

*Teoria e Prática*



Preencha a **ficha de cadastro** no final deste livro  
e receba gratuitamente informações  
sobre os lançamentos e as promoções da  
Editora Campus.

Consulte também nosso catálogo  
completo e últimos lançamentos em  
**[www.campus.com.br](http://www.campus.com.br)**

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

# **ALGORITMOS**

**TRADUÇÃO DA 2<sup>a</sup> EDIÇÃO AMERICANA**

***Teoria e Prática***

**COMPRA**

**REVISORA TÉCNICA**

**JUSSARA PIMENTA MATOS**

*Departamento de Engenharia de Computação  
e Sistemas Digitais da Escola Politécnica da USP  
e Consultora em Engenharia de Software*

**TRADUÇÃO**

**VANDENBERG D. DE SOUZA**

**6<sup>a</sup> Tiragem**



**Do original**

*Introduction to algorithms – Second Edition*

Tradução autorizada do idioma inglês da edição publicada por The MIT Press

Copyright 2001 by The Massachusetts Institute of Technology

© 2002, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

**Editoração Eletrônica**

Estúdio Castellani

**Revisão Gráfica**

Jane Castellani

**Projeto Gráfico**

Elsevier Editora Ltda.

A Qualidade da Informação.

Rua Sete de Setembro, 111 – 16º andar

20050-006 Rio de Janeiro RJ Brasil

Telefone: (21) 3970-9300 FAX: (21) 2507-1991

E-mail: [info@elsevier.com.br](mailto:info@elsevier.com.br)

**Escritório São Paulo:**

Rua Elvira Ferraz, 198

04552-040 Vila Olímpia São Paulo SP

Tel.: (11) 3841-8555

ISBN 85-352-0926-3

(Edição original: ISBN 0-07-013151-1)

CIP-Brasil. Catalogação-na-fonte.  
Sindicato Nacional dos Editores de Livros, F.

A385

Algoritmos : teoria e prática / Thomas H. Cormen... [et al.];  
tradução da segunda edição [americana] Vandenberg D. de  
Souza. – Rio de Janeiro : Elsevier, 2002 – 6ª Reimpressão.

Tradução de: *Introduction to algorithms*  
ISBN 85-352-0926-3

1. Programação (Computadores). 2. Algoritmos de computador.  
I. Cormen, Thomas H.

01-1674

CDD – 005.1

CDU – 004.421

04 05 06 07

9 8 7 6

---

# *Sumário*

Prefácio .....	XI
----------------	----

---

## **Parte I Fundamentos**

<b>Introdução .....</b>	<b>1</b>
<b>1 A função dos algoritmos na computação .....</b>	<b>3</b>
1.1 Algoritmos .....	3
1.2 Algoritmos como uma tecnologia .....	7
<b>2 Conceitos básicos .....</b>	<b>11</b>
2.1 Ordenação por inserção .....	11
2.2 Análise de algoritmos .....	16
2.3 Projeto de algoritmos .....	21
<b>3 Crescimento de funções .....</b>	<b>32</b>
3.1 Notação assintótica .....	32
3.2 Notações padrão e funções comuns .....	40
<b>4 Recorrências .....</b>	<b>50</b>
4.1 O método de substituição .....	51
4.2 O método de árvore de recursão .....	54
4.3 O método mestre .....	59
★ 4.4 Prova do teorema mestre .....	61
<b>5 Análise probabilística e algoritmos aleatórios .....</b>	<b>73</b>
5.1 O problema da contratação .....	73
5.2 Indicadores de variáveis aleatórias .....	76
5.3 Algoritmos aleatórios .....	79
★ 5.4 Análise probabilística e usos adicionais de indicadores de variáveis aleatórias ..	85

---

## **Parte II Ordenação e estatísticas de ordem**

<b>Introdução .....</b>	<b>99</b>
<b>6 Heapsort .....</b>	<b>103</b>
6.1 Heaps .....	103
6.2 Manutenção da propriedade de heap .....	105
6.3 A construção de um heap .....	107
6.4 O algoritmo heapsort .....	110
6.5 Filas de prioridades .....	111
<b>7 Quicksort .....</b>	<b>117</b>
7.1 Descrição do quicksort .....	117
7.2 O desempenho de quicksort .....	120
7.3 Uma versão aleatória de quicksort .....	124
7.4 Análise de quicksort .....	125

<b>8</b>	<b>Ordenação em tempo linear . . . . .</b>	<b>133</b>
8.1	Limites inferiores para ordenação . . . . .	133
8.2	Ordenação por contagem . . . . .	135
8.3	Radix sort . . . . .	137
8.4	Bucket sort . . . . .	140
<b>9</b>	<b>Medianas e estatísticas de ordem . . . . .</b>	<b>147</b>
9.1	Mínimo e máximo . . . . .	147
9.2	Seleção em tempo esperado linear . . . . .	149
9.3	Seleção em tempo linear no pior caso . . . . .	152

### **Parte III Estruturas de dados**

	<b>Introdução . . . . .</b>	<b>159</b>
<b>10</b>	<b>Estruturas de dados elementares . . . . .</b>	<b>163</b>
10.1	Pilhas e filas . . . . .	163
10.2	Listas ligadas . . . . .	166
10.3	Implementação de ponteiros e objetos . . . . .	170
10.4	Representação de árvores enraizadas . . . . .	173
<b>11</b>	<b>Tabelas hash . . . . .</b>	<b>179</b>
11.1	Tabelas de endereço direto . . . . .	179
11.2	Tabelas hash . . . . .	181
11.3	Funções hash . . . . .	185
11.4	Endereçamento aberto . . . . .	192
★ 11.5	Hash perfeito . . . . .	198
<b>12</b>	<b>Árvores de pesquisa binária . . . . .</b>	<b>204</b>
12.1	O que é uma árvore de pesquisa binária? . . . . .	204
12.2	Consultas em uma árvore de pesquisa binária . . . . .	207
12.3	Inserção e eliminação . . . . .	210
★ 12.4	Árvores de pesquisa binária construídas aleatoriamente . . . . .	213
<b>13</b>	<b>Árvores vermelho-preto . . . . .</b>	<b>220</b>
13.1	Propriedades de árvores vermelho-preto . . . . .	220
13.2	Rotações . . . . .	223
13.3	Inserção . . . . .	225
13.4	Eliminação . . . . .	231
<b>14</b>	<b>Ampliando estruturas de dados . . . . .</b>	<b>242</b>
14.1	Estatísticas de ordem dinâmicas . . . . .	242
14.2	Como ampliar uma estrutura de dados . . . . .	247
14.3	Árvores de intervalos . . . . .	249

### **Parte IV Técnicas avançadas de projeto e análise**

	<b>Introdução . . . . .</b>	<b>257</b>
<b>15</b>	<b>Programação dinâmica . . . . .</b>	<b>259</b>
15.1	Programação de linha de montagem . . . . .	260
15.2	Multiplicação de cadeias de matrizes . . . . .	266
15.3	Elementos de programação dinâmica . . . . .	272
15.3	Subseqüência comum mais longa . . . . .	281
15.5	Árvores de pesquisa binária ótimas . . . . .	285

<b>16</b>	<b>Algoritmos gulosos</b>	<b>296</b>
16.1	Um problema de seleção de atividade	297
16.2	Elementos da estratégia gulosa	303
16.3	Códigos de Huffman	307
★ 16.4	Fundamentos teóricos de métodos gulosos	314
★ 16.5	Um problema de programação de tarefas	319
<b>17</b>	<b>Análise amortizada</b>	<b>324</b>
17.1	A análise agregada	325
17.2	O método de contabilidade	328
17.3	O método potencial	330
17.4	Tabelas dinâmicas	333

## **Parte V Estruturas de dados avançadas**

<b>Introdução</b>	<b>345</b>	
<b>18 Árvores B</b>	<b>349</b>	
18.1	Definição de árvores B	352
18.2	Operações básicas sobre árvores B	354
18.3	Eliminação de uma chave de uma árvore B	360
<b>19 Heaps binomiais</b>	<b>365</b>	
19.1	Árvores binomiais e heaps binomiais	366
19.2	Operações sobre heaps binomiais	370
<b>20 Heaps de Fibonacci</b>	<b>381</b>	
20.1	Estrutura de heaps de Fibonacci	382
20.2	Operações de heaps intercaláveis	384
20.3	Como decrementar uma chave e eliminar um nó	390
20.4	Como limitar o grau máximo	394
<b>21 Estruturas de dados para conjuntos disjuntos</b>	<b>398</b>	
21.1	Operações de conjuntos disjuntos	398
21.2	Representação de conjuntos disjuntos por listas ligadas	400
21.3	Florestas de conjuntos disjuntos	403
★ 21.4	Análise da união por ordenação com compressão de caminho	406

## **Parte VI Algoritmos de grafos**

<b>Introdução</b>	<b>417</b>	
<b>22 Algoritmos elementares de grafos</b>	<b>419</b>	
22.1	Representações de grafos	419
22.2	Busca em largura	422
22.3	Busca em profundidade	429
22.4	Ordenação topológica	436
22.5	Componentes fortemente conectados	438
<b>23 Árvores espalhadas mínimas</b>	<b>445</b>	
23.1	Como aumentar uma árvore espalhada mínima	446
23.2	Os algoritmos de Kruskal e Prim	450
<b>24 Caminhos mais curtos de única origem</b>	<b>459</b>	
24.1	O algoritmo de Bellman-Ford	465

24.2	Caminhos mais curtos de única origem em grafos acíclicos orientados . . . . .	468
24.3	Algoritmo de Dijkstra . . . . .	470
24.4	Restrições de diferenças e caminhos mais curtos . . . . .	475
24.5	Provas de propriedades de caminhos mais curtos . . . . .	480
<b>25</b>	<b>Caminhos mais curtos de todos os pares . . . . .</b>	<b>490</b>
25.1	Caminhos mais curtos e multiplicação de matrizes . . . . .	492
25.2	O algoritmo de Floyd-Warshall . . . . .	497
25.3	Algoritmo de Johnson para grafos esparsos . . . . .	503
<b>26</b>	<b>Fluxo máximo . . . . .</b>	<b>509</b>
26.1	Fluxo em redes . . . . .	510
26.2	O método de Ford-Fulkerson . . . . .	515
26.3	Emparelhamento bipartido máximo . . . . .	525
★ 26.4	Algoritmos de push-relabel . . . . .	529
★ 26.5	O algoritmo de relabel-to-front . . . . .	538

## Parte VII Tópicos selecionados

<b>Introdução . . . . .</b>	<b>553</b>
<b>27 Redes de ordenação . . . . .</b>	<b>555</b>
27.1 Redes de comparação . . . . .	555
27.2 O princípio zero um . . . . .	559
27.3 Uma rede de ordenação bitônica . . . . .	561
27.4 Uma rede de intercalação . . . . .	564
27.5 Uma rede de ordenação . . . . .	566
<b>28 Operações sobre matrizes . . . . .</b>	<b>571</b>
28.1 Propriedades de matrizes . . . . .	571
28.2 Algoritmo de Strassen para multiplicação de matrizes . . . . .	579
28.3 Resolução de sistemas de equações lineares . . . . .	585
28.4 Inversão de matrizes . . . . .	597
28.5 Matrizes simétricas definidas como positivas e aproximação de mínimos quadrados . . . . .	601
<b>29 Programação linear . . . . .</b>	<b>610</b>
29.1 Formas padrão e relaxada . . . . .	616
29.2 Formulação de problemas como programas lineares . . . . .	622
29.3 O algoritmo simplex . . . . .	626
29.4 Dualidade . . . . .	638
29.5 A solução básica inicial possível . . . . .	643
<b>30 Polinômios e a FFT . . . . .</b>	<b>651</b>
30.1 Representação de polinômios . . . . .	653
30.2 A DFT e a FFT . . . . .	658
30.3 Implementações eficientes de FFT . . . . .	664
<b>31 Algoritmos de teoria dos números . . . . .</b>	<b>672</b>
31.1 Noções de teoria elementar dos números . . . . .	673
31.2 Máximo divisor comum . . . . .	678
31.3 Aritmética modular . . . . .	682
31.4 Resolução de equações lineares modulares . . . . .	688
31.5 O teorema chinês do resto . . . . .	691
31.6 Potências de um elemento . . . . .	693

31.7	O sistema de criptografia de chave pública RSA.....	697
★ 31.8	Como testar o caráter primo .....	702
★ 31.9	Fatoração de inteiros.....	709
<b>32</b>	<b>Emparelhamento de cadeias.....</b>	<b>717</b>
32.1	O algoritmo simples de emparelhamento de cadeias .....	719
32.2	O algoritmo de Rabin-Karp.....	721
32.3	Emparelhamento de cadeias com autômatos finitos.....	725
★ 32.4	O algoritmo de Knuth-Morris-Pratt.....	730
<b>33</b>	<b>Geometria computacional.....</b>	<b>738</b>
33.1	Propriedades de segmentos de linha .....	738
33.2	Como determinar se dois segmentos quaisquer se cruzam .....	743
33.3	Como encontrar a envoltória convexa.....	749
33.4	Localização do par de pontos mais próximos.....	756
<b>34</b>	<b>Problemas NP-completos .....</b>	<b>763</b>
34.1	Tempo polinomial.....	767
34.2	Verificação de tempo polinomial.....	773
34.3	Caráter NP-completo e redutibilidade.....	776
34.4	Provas do caráter NP-completo .....	785
34.5	Problemas NP-completos.....	791
<b>35</b>	<b>Algoritmos de aproximação .....</b>	<b>806</b>
35.1	O problema de cobertura de vértices.....	808
35.2	O problema do caixeiro-viajante .....	810
35.3	O problema de cobertura de conjuntos .....	815
35.4	Aleatoriedade e programação linear.....	819
35.5	O problema de soma de subconjuntos .....	823

## Parte VIII Apêndice: Fundamentos de matemática

<b>Introdução .....</b>	<b>833</b>
<b>A Somatórios .....</b>	<b>835</b>
A.1 Fórmulas e propriedades de somatórios.....	835
A.2 Como limitar somatórios.....	838
<b>B Conjuntos e outros temas .....</b>	<b>845</b>
B.1 Conjuntos .....	845
B.2 Relações .....	849
B.3 Funções.....	851
B.4 Grafos.....	853
B.5 Árvores .....	856
<b>C Contagem e probabilidade .....</b>	<b>863</b>
C.1 Contagem .....	863
C.2 Probabilidade .....	868
C.3 Variáveis aleatórias discretas .....	873
C.4 As distribuições geométrica e binomial .....	878
★ C.5 As extremidades da distribuição binomial .....	883
<b>Bibliografia .....</b>	<b>890</b>
<b>Índice .....</b>	<b>898</b>



---

# Prefácio

Este livro oferece uma introdução abrangente ao estudo moderno de algoritmos de computador. Ele apresenta muitos algoritmos e os examina com uma profundidade considerável, tornando seu projeto e sua análise acessíveis aos leitores de todos os níveis. Tentamos manter as explicações em um nível elementar sem sacrificar a profundidade do enfoque ou o rigor matemático.

Cada capítulo apresenta um algoritmo, uma técnica de projeto, uma área de aplicação ou um tópico relacionado. Os algoritmos são descritos em linguagem comum e em um “pseudocódigo” projetado para ser legível por qualquer pessoa que tenha um pouco de experiência em programação. O livro contém mais de 230 figuras ilustrando como os algoritmos funcionam. Tendo em vista que enfatizamos a *eficiência* como um critério de projeto, incluímos análises cuidadosas dos tempos de execução de todos os nossos algoritmos.

O texto foi planejado principalmente para uso em graduação e pós-graduação em algoritmos ou estruturas de dados. Pelo fato de discutir questões de engenharia relacionadas ao projeto de algoritmos, bem como aspectos matemáticos, o livro é igualmente adequado para auto-estudo de profissionais técnicos.

Nesta segunda edição, atualizamos o livro inteiro. As mudanças variam da adição de novos capítulos até a reestruturação de frases individuais.

## Para o professor

Este livro foi projetado para ser ao mesmo tempo versátil e completo. Você descobrirá sua utilidade para uma variedade de cursos, desde a graduação em estruturas de dados, até a pós-graduação em algoritmos. Pelo fato de fornecermos uma quantidade de material consideravelmente maior do que poderia caber em um curso típico de um período, você deve imaginar o livro como um “bufê” ou “depósito”, do qual pode selecionar e extrair o material que melhor atender ao curso que desejar ministrar.

Você achará fácil organizar seu curso apenas em torno dos capítulos de que necessitar. Tornamos os capítulos relativamente autônomos, para que você não precise se preocupar com uma dependência inesperada e desnecessária de um capítulo em relação a outro. Cada capítulo apresenta primeiro o material mais fácil e mostra os assuntos mais difíceis em seguida, com limites de seções assinalando pontos de parada naturais. Na graduação, poderão ser utilizadas apenas as primeiras seções de um capítulo; na pós-graduação, será possível estudar o capítulo inteiro.

Incluímos mais de 920 exercícios e mais de 140 problemas. Cada seção termina com exercícios, e cada capítulo com problemas. Em geral, os exercícios são perguntas curtas que testam o domínio básico do assunto. Alguns são exercícios simples criados para a sala de aula, enquanto outros são mais substanciais e apropriados para uso como dever de casa. Os problemas são estudos de casos mais elaborados que, com freqüência, apresentam novos assuntos; normalmente, eles consistem em várias perguntas que conduzem o aluno por etapas exigidas para chegar a uma solução.

Assinalamos com asteriscos (\*) as seções e os exercícios mais adequados para alunos avançados. Uma seção com asteriscos não é necessariamente mais difícil que outra que não tenha asteriscos, mas pode exigir a compreensão de matemática em um nível mais profundo. Da mesma forma, os exercícios com asteriscos podem exigir um conhecimento mais avançado ou criatividade acima da média.

## **Para o aluno**

Esperamos que este livro-texto lhe proporcione uma introdução agradável ao campo dos algoritmos. Tentamos tornar cada algoritmo acessível e interessante. Para ajudá-lo quando você encontrar algoritmos pouco familiares ou difíceis, descrevemos cada um deles passo a passo. Também apresentamos explicações cuidadosas dos conhecimentos matemáticos necessários à compreensão da análise dos algoritmos. Se já tiver alguma familiaridade com um tópico, você perceberá que os capítulos estão organizados de modo que seja possível passar os olhos pelas seções introdutórias e seguir rapidamente para o material mais avançado.

Este é um livro extenso, e sua turma provavelmente só examinará uma parte de seu conteúdo. Porém, procuramos torná-lo útil para você agora como um livro-texto, e também mais tarde em sua carreira, sob a forma de um guia de referência de matemática ou um manual de engenharia.

Quais são os pré-requisitos para a leitura deste livro?

- Você deve ter alguma experiência em programação. Em particular, deve entender procedimentos recursivos e estruturas de dados simples como arranjos e listas ligadas.
- Você deve ter alguma facilidade com a realização de demonstrações por indução matemática. Algumas partes do livro se baseiam no conhecimento de cálculo elementar. Além disso, as Partes I e VIII deste livro ensinam todas as técnicas matemáticas de que você irá necessitar.

## **Para o profissional**

A ampla variedade de tópicos deste livro o torna um excelente manual sobre algoritmos. Como cada capítulo é relativamente autônomo, você pode se concentrar nos tópicos que mais o interessem.

A maioria dos algoritmos que discutimos tem grande utilidade prática. Por essa razão, abordamos conceitos de implementação e outras questões de engenharia. Em geral, oferecemos alternativas práticas para os poucos algoritmos que têm interesse principalmente teórico.

Se desejar implementar algum dos algoritmos, você irá achar a tradução do nosso pseudocódigo em sua linguagem de programação favorita uma tarefa bastante objetiva. O pseudocódigo foi criado para apresentar cada algoritmo de forma clara e sucinta. Consequentemente, não nos preocupamos com o tratamento de erros e outras questões ligadas à engenharia de software que exigem suposições específicas sobre o seu ambiente de programação. Tentamos apresentar cada algoritmo de modo simples e direto sem permitir que as idiossincrasias de uma determinada linguagem de programação obscurecessem sua essência.

## **Para os nossos colegas**

Fornecemos uma bibliografia e ponteiros extensivos para a literatura corrente. Cada capítulo termina com um conjunto de “notas do capítulo” que fornecem detalhes e referências históricas. Contudo, as notas dos capítulos não oferecem uma referência completa para o campo inteiro de algoritmos. Porém, pode ser difícil acreditar que, em um livro com este tamanho, muitos algoritmos interessantes não puderam ser incluídos por falta de espaço.

Apesar dos inúmeros pedidos de soluções para problemas e exercícios feitos pelos alunos, escolhemos como norma não fornecer referências para problemas e exercícios, a fim de evitar que os alunos cedessem à tentação de olhar uma solução pronta em lugar de encontrá-la eles mesmos.

## Mudanças na segunda edição

O que mudou entre a primeira e a segunda edição deste livro?

Dependendo de como você o encara, o livro pode não ter mudado muito ou ter mudado bastante.

Um rápido exame no sumário mostra que a maior parte dos capítulos e das seções da primeira edição também estão presentes na segunda edição. Removemos dois capítulos e algumas seções, mas adicionamos três novos capítulos e quatro novas seções além desses novos capítulos. Se fosse julgar o escopo das mudanças pelo sumário, você provavelmente concluiria que as mudanças foram modestas.

Porém, as alterações vão muito além do que aparece no sumário. Sem qualquer ordem particular, aqui está um resumo das mudanças mais significativas da segunda edição:

- Cliff Stein foi incluído como co-autor.
- Os erros foram corrigidos. Quantos erros? Vamos dizer apenas que foram vários.
- Há três novos capítulos:
  - O Capítulo 1 discute a função dos algoritmos em informática.
  - O Capítulo 5 abrange a análise probabilística e os algoritmos aleatórios. Como na primeira edição, esses tópicos aparecem em todo o livro.
  - O Capítulo 29 é dedicado à programação linear.
- Dentro dos capítulos que foram trazidos da primeira edição, existem novas seções sobre os seguintes tópicos:
  - Hash perfeito (Seção 11.5).
  - Duas aplicações de programação dinâmica (Seções 15.1 e 15.5).
  - Algoritmos de aproximação que usam técnicas aleatórias e programação linear (Seção 35.4).
- Para permitir que mais algoritmos apareçam mais cedo no livro, três dos capítulos sobre fundamentos matemáticos foram reposicionados, migrando da Parte I para os apêndices, que formam a Parte VIII.
- Há mais de 40 problemas novos e mais de 185 exercícios novos.
- Tornamos explícito o uso de loops invariantes para demonstrar a correção. Nossa primeiro loop invariante aparece no Capítulo 2, e nós os utilizamos algumas dezenas de vezes ao longo do livro.
- Muitas das análises probabilísticas foram reescritas. Em particular, usamos em aproximadamente uma dezena de lugares a técnica de “variáveis indicadoras aleatórias” que simplificam as análises probabilísticas, em especial quando as variáveis aleatórias são dependentes.
- Expandimos e atualizamos as notas dos capítulos e a bibliografia. A bibliografia cresceu mais de 50%, e mencionamos muitos novos resultados algorítmicos que surgiram após a impressão da primeira edição.

Também fizemos as seguintes mudanças:

- O capítulo sobre resolução de recorrências não contém mais o método de iteração. Em vez disso, na Seção 4.2, “promovemos” as árvores de recursão, que passaram a constituir um método por si só. Concluímos que criar árvores de recursão é menos propenso a erros do que fazer a iteração de recorrências. Porém, devemos assinalar que as árvores de recursão são mais bem usadas como um modo de gerar hipóteses que serão então verificadas através do método de substituição.

- O método de particionamento usado para ordenação rápida (Seção 7.1) e no algoritmo de ordem estatística de tempo linear esperado (Seção 9.2) é diferente. Agora usamos o método desenvolvido por Lomuto que, junto com variáveis indicadoras aleatórias, permite uma análise um pouco mais simples. O método da primeira edição, devido a Hoare, é apresentado como um problema no Capítulo 7.
- Modificamos a discussão sobre o hash universal da Seção 11.3.3 de forma que ela se integre à apresentação do hash perfeito.
- Você encontrará na Seção 12.4 uma análise muito mais simples da altura de uma árvore de pesquisa binária construída aleatoriamente.
- As discussões sobre os elementos de programação dinâmica (Seção 15.3) e os elementos de algoritmos gulosos (Seção 16.2) foram significativamente expandidas. A exploração do problema de seleção de atividade, que inicia o capítulo de algoritmos gulosos, ajuda a esclarecer a relação entre programação dinâmica e algoritmos gulosos.
- Substituímos a prova do tempo de execução da estrutura de dados de união de conjuntos disjuntos na Seção 21.4 por uma prova que emprega o método potencial para derivar um limite restrito.
- A prova de correção do algoritmo para componentes fortemente conectados na Seção 22.5 é mais simples, mais clara e mais direta.
- O Capítulo 24, que aborda os caminhos mais curtos de origem única, foi reorganizado com a finalidade de mover as provas das propriedades essenciais para suas próprias seções. A nova organização permite que nos concentremos mais cedo nos algoritmos.
- A Seção 34.5 contém uma visão geral ampliada do caráter NP-completo, e também novas provas de caráter NP-completo para os problemas do ciclo hamiltoniano e da soma de subconjuntos.

Por fim, virtualmente todas as seções foram editadas para corrigir, simplificar e tornar mais claras as explicações e demonstrações.

## Agradecimentos da primeira edição

Muitos amigos e colegas contribuíram bastante para a qualidade deste livro. Agradecemos a todos por sua ajuda e suas críticas construtivas.

O laboratório de ciência da computação do MIT proporcionou um ambiente de trabalho ideal. Nossos colegas do grupo de teoria da computação do laboratório foram particularmente incentivadores e tolerantes em relação aos nossos pedidos incessantes de avaliação crítica dos capítulos. Agradecemos especificamente a Baruch Awerbuch, Shafi Goldwasser, Leo Guibas, Tom Leighton, Albert Meyer, David Shmoys e Éva Tardos. Agradecemos a William Ang, Sally Bemus, Ray Hirschfeld e Mark Reinhold por manterem nossas máquinas (equipamentos DEC Microvax, Apple Macintosh e Sun Sparcstation) funcionando e por recompilarem TEX sempre que excedemos um limite de prazo de compilação. A Thinking Machines Corporation forneceu suporte parcial a Charles Leiserson para trabalhar neste livro durante um período de ausência do MIT.

Muitos colegas usaram rascunhos deste texto em cursos realizados em outras faculdades. Eles sugeriram numerosas correções e revisões. Desejamos agradecer em particular a Richard Beigel, Andrew Goldberg, Joan Lucas, Mark Overmars, Alan Sherman e Diane Souvaine.

Muitos assistentes de ensino em nossos cursos apresentaram contribuições significativas para o desenvolvimento deste material. Agradecemos especialmente a Alan Baratz, Bonnie Berger, Aditi Dhagat, Burt Kaliski, Arthur Lent, Andrew Moulton, Marios Papaefthymiou, Cindy Phillips, Mark Reinhold, Phil Rogaway, Flavio Rose, Arie Rudich, Alan Sherman, Cliff Stein, Susmita Sur, Gregory Troxel e Margaret Tuttle.

Uma valiosa assistência técnica adicional foi fornecida por muitas pessoas. Denise Sergent passou muitas horas nas bibliotecas do MIT pesquisando referências bibliográficas. Maria Sensale, a bibliotecária de nossa sala de leitura, foi sempre atenciosa e alegre. O acesso à biblioteca pessoal de Albert Meyer nos poupou muitas horas na biblioteca durante a preparação das anotações dos capítulos. Shlomo Kipnis, Bill Niehaus e David Wilson revisaram os exercícios antigos, desenvolveram novos e escreveram notas sobre suas soluções. Marios Papaefthymiou e Gregory Troxel contribuíram para a indexação. Ao longo dos anos, nossas secretárias Inna Radzihovsky, Denise Sergent, Gayle Sherman e especialmente Be Blackburn proporcionaram apoio infinável a este projeto, e por isso somos gratos a elas.

Muitos erros nos rascunhos iniciais foram relatados por alunos. Agradecemos especificamente a Bobby Blumofe, Bonnie Eisenberg, Raymond Johnson, John Keen, Richard Lethin, Mark Lillibridge, John Pezaris, Steve Ponzio e Margaret Tuttle por sua leitura cuidadosa dos originais.

Nossos colegas também apresentaram resenhas críticas de capítulos específicos, ou informações sobre determinados algoritmos, pelos quais somos gratos. Agradecemos ainda a Bill Aiello, Alok Aggarwal, Eric Bach, Vašek Chvátal, Richard Cole, Johan Hastad, Alex Ishii, David Johnson, Joe Kilian, Dina Kravets, Bruce Maggs, Jim Orlin, James Park, Thane Plambeck, Herschel Safer, Jeff Shallit, Cliff Stein, Gil Strang, Bob Tarjan e Paul Wang. Vários de nossos colegas gentilmente também nos forneceram problemas; agradecemos em particular a Andrew Goldberg, Danny Sleator e Umesh Vazirani.

Foi um prazer trabalhar com The MIT Press e a McGraw-Hill no desenvolvimento deste texto. Agradecemos especialmente a Frank Satlow, Terry Ehling, Larry Cohen e Lorrie Lejeune da The MIT Press, e a David Shapiro da McGraw-Hill por seu encorajamento, apoio e paciência. Somos particularmente agradecidos a Larry Cohen por seu excelente trabalho de edição.

### Agradecimentos da segunda edição

Quando pedimos a Julie Sussman, P. P. A., que atuasse como editora técnica da segunda edição, não sabíamos que bom negócio estávamos fazendo. Além de realizar a edição do conteúdo técnico, Julie editou entusiasticamente nosso texto. É humilhante pensar em quantos erros Julie encontrou em nossos esboços antigos; entretanto, considerando a quantidade de erros que ela achou na primeira edição (depois de impressa, infelizmente), isso não surpreende. Além disso, Julie sacrificou sua própria programação para se adaptar à nossa – chegou até a levar capítulos do livro com ela em uma viagem às Ilhas Virgens! Julie, nunca conseguiremos agradecer-lhe o bastante pelo trabalho fantástico que você realizou.

O trabalho para a segunda edição foi feito enquanto os autores eram membros do departamento de ciência da computação no Dartmouth College e no laboratório de ciência da computação do MIT. Ambos foram ambientes de trabalho estimulantes, e agradecemos a nossos colegas por seu apoio.

Amigos e colegas do mundo inteiro ofereceram sugestões e opiniões que orientaram nossa escrita. Muito obrigado a Sanjeev Arora, Javed Aslam, Guy Blelloch, Avrim Blum, Scot Drysdale, Hany Farid, Hal Gabow, Andrew Goldberg, David Johnson, Yanlin Liu, Nicolas Schabanel, Alexander Schrijver, Sasha Shen, David Shmoys, Dan Spielman, Gerald Jay Sussman, Bob Tarjan, Mikkel Thorup e Vijay Vazirani.

Vários professores e colegas nos ensinaram muito sobre algoritmos. Em particular, reconhecemos o esforço e a dedicação de nossos professores Jon L. Bentley, Bob Floyd, Don Knuth, Harold Kuhn, H. T. Kung, Richard Lipton, Arnold Ross, Larry Snyder, Michael I. Shamos, David Shmoys, Ken Steiglitz, Tom Szymanski, Éva Tardos, Bob Tarjan e Jeffrey Ullman.

Reconhecemos o trabalho dos muitos assistentes de ensino dos cursos de algoritmos no MIT e em Dartmouth, inclusive Joseph Adler, Craig Barrack, Bobby Blumofe, Roberto De Prisco, Matteo Frigo, Igal Galperin, David Gupta, Raj D. Iyer, Nabil Kahale, Sarfraz Khurshid, Stavros Kolliopoulos, Alain Leblanc, Yuan Ma, Maria Minkoff, Dimitris Mitsouras, Alin Popescu, Harald Pro-

kop, Sudipta Sengupta, Donna Slonim, Joshua A. Tauber, Sivan Toledo, Elisheva Werner-Reiss, Lea Wittie, Qiang Wu e Michael Zhang.

O suporte de informática foi oferecido por William Ang, Scott Blomquist e Greg Shomo no MIT e por Wayne Cripps, John Konkle e Tim Tregubov em Dartmouth. Agradecemos também a Be Blackburn, Don Dailey, Leigh Deacon, Irene Sebeda e Cheryl Patton Wu no MIT, e a Phyllis Bellmore, Kelly Clark, Delia Mauceli, Sammie Travis, Deb Whiting e Beth Young, de Dartmouth, pelo suporte administrativo. Michael Fromberger, Brian Campbell, Amanda Eubanks, Sung Hoon Kim e Neha Narula também ofereceram apoio oportuno em Dartmouth.

Muitas pessoas fizeram a gentileza de informar sobre erros cometidos na primeira edição. Agradecemos aos leitores mencionados na lista a seguir; cada um deles foi o primeiro a relatar um erro da primeira edição: Len Adleman, Selim Akl, Richard Anderson, Juan Andrade-Cetto, Gregory Bachelis, David Barrington, Paul Beame, Richard Beigel, Margrit Betke, Alex Blakemore, Bobby Blumofe, Alexander Brown, Xavier Cazin, Jack Chan, Richard Chang, Chienhua Chen, Ien Cheng, Hoon Choi, Drue Coles, Christian Collberg, George Collins, Eric Conrad, Peter Csaszar, Paul Dietz, Martin Dietzfelbinger, Scot Drysdale, Patricia Ealy, Yaakov Eisenberg, Michael Ernst, Michael Formann, Nedim Fresko, Hal Gabow, Marek Galecki, Igal Galperin, Luisa Gargano, John Gately, Rosario Genario, Mihaly Gereb, Ronald Greenberg, Jerry Grossman, Stephen Guattery, Alexander Hartemik, Anthony Hill, Thomas Hofmeister, Mathew Hostetter, Yih-Chun Hu, Dick Johnsonbaugh, Marcin Jurdzinski, Nabil Kahale, Fumiaki Kamiya, Anand Kanagala, Mark Kantrowitz, Scott Karlin, Dean Kelley, Sanjay Khanna, Haluk Konuk, Dina Kravets, Jon Kroger, Bradley Kuszmaul, Tim Lambert, Hang Lau, Thomas Lengauer, George Madrid, Bruce Maggs, Victor Miller, Joseph Muskat, Tung Nguyen, Michael Orlov, James Park, Seongbin Park, Ioannis Paschalidis, Boaz Patt-Shamir, Leonid Peshkin, Patricio Poblete, Ira Pohl, Stephen Ponzio, Kjell Post, Todd Poinor, Colin Prepscious, Sholom Rosen, Dale Russell, Hershel Safer, Karen Seidel, Joel Seiferas, Erik Seligman, Stanley Selkow, Jeffrey Shallit, Greg Shannon, Micha Sharir, Sasha Shen, Norman Shulman, Andrew Singer, Daniel Sleator, Bob Sloan, Michael Sofka, Volker Strumpen, Lon Sunshine, Julie Sussman, Asterio Tanaka, Clark Thomborson, Nils Thommesen, Homer Tilton, Martin Tompa, Andrei Toom, Felzer Torsten, Hirendu Vaishnav, M. Veldhorst, Luca Venuti, Jian Wang, Michael Wellman, Gerry Wiener, Ronald Williams, David Wolfe, Jeff Wong, Richard Woundy, Neal Young, Huaiyuan Yu, Tian Yuxing, Joe Zachary, Steve Zhang, Florian Zschoke e Uri Zwick.

Muitos de nossos colegas apresentaram críticas atentas ou preencheram um longo formulário de pesquisa. Agradecemos aos revisores Nancy Amato, Jim Aspnes, Kevin Compton, William Evans, Peter Gacs, Michael Goldwasser, Andrzej Proskurowski, Vijaya Ramachandran e John Reif. Também agradecemos às seguintes pessoas por devolverem a pesquisa: James Abello, Josh Benaloh, Bryan Beresford-Smith, Kenneth Blaha, Hans Bodlaender, Richard Borie, Ted Brown, Domenico Cantone, M. Chen, Robert Cimikowski, William Clocksin, Paul Cull, Rick Decker, Matthew Dickerson, Robert Douglas, Margaret Fleck, Michael Goodrich, Susanne Hambrusch, Dean Hendrix, Richard Johnsonbaugh, Kyriakos Kalorkoti, Srinivas Kankanhalli, Hikyoo Koh, Steven Lindell, Errol Lloyd, Andy Lopez, Dian Rae Lopez, George Lucke, David Maier, Charles Martel, Xiannong Meng, David Mount, Alberto Policriti, Andrzej Proskurowski, Kirk Pruhs, Yves Robert, Guna Seetharaman, Stanley Selkow, Robert Sloan, Charles Steele, Gerard Tel, Murali Varanasi, Bernd Walter e Alden Wright. Gostaríamos que tivesse sido possível implementar todas as suas sugestões. O único problema é que, se isso fosse feito, a segunda edição teria mais ou menos 3.000 páginas!

A segunda edição foi produzida em  $\text{\LaTeX} 2_{\epsilon}$ . Michael Downes converteu as macros de  $\text{\LaTeX}$  do  $\text{\LaTeX}$  “clássico” para  $\text{\LaTeX} 2_{\epsilon}$  e converteu os arquivos de texto para usar essas novas macros. David Jones também forneceu suporte para  $\text{\LaTeX} 2_{\epsilon}$ . As figuras da segunda edição foram produzidas pelos autores usando o MacDraw Pro. Como na primeira edição, o índice foi compilado com o uso de Windex, um programa em C escrito pelos autores, e a bibliografia foi preparada com a utilização do BIB $\text{\TeX}$ . Ayorkor Mills-Tettey e Rob Leatherne ajudaram a converter as figuras para MacDraw Pro, e Ayorkor também conferiu nossa bibliografia.

Como também aconteceu na primeira edição, trabalhar com The MIT Press e com a McGraw-Hill foi um prazer. Nossos editores, Bob Prior da MIT Press e Betsy Jones da McGraw-Hill, toleraram nossos gracejos e nos mantiveram no rumo.

Finalmente, agradecemos a nossas esposas – Nicole Cormen, Gail Rivest e Rebecca Ivry – a nossos filhos – Ricky, William e Debby Leiserson, Alex e Christopher Rivest, e Molly, Noah e Benjamin Stein – e a nossos pais – Renee e Perry Cormen, Jean e Mark Leiserson, Shirley e Lloyd Rivest, e Irene e Ira Stein – por seu carinho e apoio durante a elaboração deste livro. O amor, a paciência e o incentivo de nossos familiares tornaram este projeto possível. Dedicamos afetuosamente este livro a eles.

Thomas H. Cormen  
Charles E. Leiserson  
Ronald L. Rivest  
Clifford Stein

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

*Maio de 2001*



# *Parte I*

## *Fundamentos*

### **Introdução**

Esta parte o fará refletir sobre o projeto e a análise de algoritmos. Ela foi planejada para ser uma introdução suave ao modo como especificamos algoritmos, a algumas das estratégias de projeto que usaremos ao longo deste livro e a muitas das idéias fundamentais empregadas na análise de algoritmos. As partes posteriores deste livro serão elaboradas sobre essa base.

O Capítulo 1 é uma visão geral dos algoritmos e de seu lugar nos modernos sistemas de computação. Esse capítulo define o que é um algoritmo e lista alguns exemplos. Ele também apresenta os algoritmos como uma tecnologia, da mesma maneira que um hardware rápido, interfaces gráficas do usuário, sistemas orientados a objetos e redes de computadores.

No Capítulo 2, veremos nossos primeiros algoritmos, que resolvem o problema de ordenar uma seqüência de  $n$  números. Eles são escritos em um pseudocódigo que, embora não possa ser traduzido diretamente para qualquer linguagem de programação convencional, transmite a estrutura do algoritmo com clareza suficiente para que um programador competente possa implementá-la na linguagem de sua escolha. Os algoritmos de ordenação que examinaremos são a ordenação por inserção, que utiliza uma abordagem incremental, e a ordenação por intercalação, que usa uma técnica recursiva conhecida como “dividir e conquistar”. Embora o tempo exigido por cada uma aumente com o valor  $n$ , a taxa de aumento difere entre os dois algoritmos. Determinaremos esses tempos de execução no Capítulo 2 e desenvolveremos uma notação útil para expressá-los.

O Capítulo 3 define com exatidão essa notação, que chamaremos notação assintótica. Ele começa definindo diversas notações assintóticas que utilizaremos para delimitar os tempos de execução dos algoritmos acima e/ou abaixo. O restante do Capítulo 3 é principalmente uma apresentação da notação matemática. Seu propósito maior é o de assegurar que o uso que você fará da notação corresponderá à utilização deste livro, em vez de ensinar-lhe novos conceitos matemáticos.

O Capítulo 4 mergulha mais profundamente no método de dividir e conquistar introduzido no Capítulo 2. Em particular, o Capítulo 4 contém métodos para solução de recorrências que são úteis para descrever os tempos de execução de algoritmos recursivos. Uma técnica eficiente é o “método mestre”, que pode ser usado para resolver recorrências que surgem dos algoritmos de dividir e conquistar. Grande parte do Capítulo 4 é dedicada a demonstrar a correção do método mestre, embora essa demonstração possa ser ignorada sem problemas.

O Capítulo 5 introduz a análise probabilística e os algoritmos aleatórios. Em geral, usaremos a análise probabilística para determinar o tempo de execução de um algoritmo nos casos em que, devido à presença de uma distribuição de probabilidades inerente, o tempo de execução pode diferir em diversas entradas do mesmo tamanho. Em alguns casos, vamos supor que as entradas obedecem a uma distribuição de probabilidades conhecida, e assim calcularemos o tempo de execução médio sobre todas as entradas possíveis. Em outros casos, a distribuição de probabilidades não vem das entradas, mas sim das escolhas aleatórias feitas durante o curso do algoritmo. Um algoritmo cujo comportamento é determinado não apenas por sua entrada, mas também pelos valores produzidos por um gerador de números aleatórios, é um algoritmo aleatório. Podemos usar algoritmos aleatórios para impor uma distribuição de probabilidade sobre as entradas – assegurando assim que nenhuma entrada específica sempre causará um fraco desempenho – ou mesmo para limitar a taxa de erros de algoritmos que têm permissão para produzir resultados incorretos de forma limitada.

Os Apêndices A, B e C contêm outros materiais matemáticos que você irá considerar úteis à medida que ler este livro. É provável que você tenha visto grande parte do material dos apêndices antes de encontrá-los neste livro (embora as convenções específicas de notação que usamos possam diferir em alguns casos daquelas que você já viu), e assim você deve considerar os apêndices um guia de referência. Por outro lado, é provável que você ainda não tenha visto a maior parte do material contido na Parte I. Todos os capítulos da Parte I e os apêndices foram escritos com um “toque” de tutorial.

---

# *Capítulo 1*

## *A função dos algoritmos na computação*

O que são algoritmos? Por que o estudo dos algoritmos vale a pena? Qual é a função dos algoritmos em relação a outras tecnologias usadas em computadores? Neste capítulo, responderemos a essas perguntas.

### **1.1 Algoritmos**

Informalmente, um **algoritmo** é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como **entrada** e produz algum valor ou conjunto de valores como **saída**. Portanto, um algoritmo é uma seqüência de passos computacionais que transformam a entrada na saída.

Também podemos visualizar um algoritmo como uma ferramenta para resolver um **problema computacional** bem especificado. O enunciado do problema especifica em termos gerais o relacionamento entre a entrada e a saída desejada. O algoritmo descreve um procedimento computacional específico para se alcançar esse relacionamento da entrada com a saída.

Por exemplo, poderia ser necessário ordenar uma seqüência de números em ordem não decrescente. Esse problema surge com freqüência na prática e oferece um solo fértil para a introdução de muitas técnicas de projeto padrão e ferramentas de análise. Vejamos como definir formalmente o **problema de ordenação**:

**Entrada:** Uma seqüência de  $n$  números  $\langle a_1, a_2, \dots, a_n \rangle$ .

**Saída:** Uma permutação (reordenação)  $\langle a'_1, a'_2, \dots, a'_n \rangle$  da seqüência de entrada, tal que  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Dada uma seqüência de entrada como  $\langle 31, 41, 59, 26, 41, 58 \rangle$ , um algoritmo de ordenação retorna como saída a seqüência  $\langle 26, 31, 41, 41, 58, 59 \rangle$ . Uma seqüência de entrada como essa é chamada uma **instância** do problema de ordenação. Em geral, uma **instância de um problema** consiste na entrada (que satisfaz a quaisquer restrições impostas no enunciado do problema) necessária para se calcular uma solução para o problema.

A ordenação é uma operação fundamental em ciência da computação (muitos programas a utilizam como uma etapa intermediária) e, como resultado, um grande número de bons algoritmos de ordenação tem sido desenvolvido. O melhor algoritmo para uma determinada aplicação |  
3

depende – entre outros fatores – do número de itens a serem ordenados, da extensão em que os itens já estão ordenados de algum modo, de possíveis restrições sobre os valores de itens e da espécie de dispositivo de armazenamento a ser usado: memória principal, discos ou fitas.

Um algoritmo é dito *correto* se, para cada instância de entrada, ele pára com a saída correta. Dizemos que um algoritmo correto *resolve* o problema computacional dado. Um algoritmo incorreto pode não parar em algumas instâncias de entrada, ou então pode parar com outra resposta que não a desejada. Ao contrário do que se poderia esperar, às vezes os algoritmos incorretos podem ser úteis, se sua taxa de erros pode ser controlada. Veremos um exemplo desse fato no Capítulo 31, quando estudarmos algoritmos para localizar grandes números primos. Porém, em situações comuns, iremos nos concentrar apenas no estudo de algoritmos corretos.

Um algoritmo pode ser especificado em linguagem comum, como um programa de computador, ou mesmo como um projeto de hardware. O único requisito é que a especificação deve fornecer uma descrição precisa do procedimento computacional a ser seguido.

## Que tipos de problemas são resolvidos por algoritmos?

A ordenação não é de modo algum o único problema computacional para o qual foram desenvolvidos algoritmos. (Você provavelmente suspeitou disso quando viu o tamanho deste livro.) As aplicações práticas de algoritmos são onipresentes e incluem os exemplos a seguir:

- O Projeto Genoma Humano tem como objetivos identificar todos os 100.000 genes do DNA humano, determinar as seqüências dos 3 bilhões de pares de bases químicas que constituem o DNA humano, armazenar essas informações em bancos de dados e desenvolver ferramentas para análise de dados. Cada uma dessas etapas exige algoritmos sofisticados. Embora as soluções para os vários problemas envolvidos estejam além do escopo deste livro, idéias de muitos capítulos do livro são usadas na solução desses problemas biológicos, permitindo assim aos cientistas realizarem tarefas ao mesmo tempo que utilizam com eficiência os recursos. As economias são de tempo, tanto humano quanto da máquina, e de dinheiro, à medida que mais informações podem ser extraídas de técnicas de laboratório.
- A Internet permite que pessoas espalhadas por todo o mundo acessem e obtenham com rapidez grandes quantidades de informações. Para isso, são empregados algoritmos inteligentes com a finalidade de gerenciar e manipular esse grande volume de dados. Os exemplos de problemas que devem ser resolvidos incluem a localização de boas rotas pelas quais os dados viajarão (as técnicas para resolver tais problemas são apresentadas no Capítulo 24) e o uso de um mecanismo de pesquisa para encontrar com rapidez páginas em que residem informações específicas (as técnicas relacionadas estão nos Capítulos 11 e 32).
- O comércio eletrônico permite que mercadorias e serviços sejam negociados e trocados eletronicamente. A capacidade de manter privativas informações como números de cartão de crédito, senhas e extratos bancários é essencial para a ampla utilização do comércio eletrônico. A criptografia de chave pública e as assinaturas digitais (estudadas no Capítulo 31) estão entre as tecnologias centrais utilizadas e se baseiam em algoritmos numéricos e na teoria dos números.
- Na indústria e em outras instalações comerciais, muitas vezes é importante alocar recursos escassos da maneira mais benéfica. Uma empresa petrolífera talvez deseje saber onde localizar seus poços para tornar máximo o lucro esperado. Um candidato à presidência da República talvez queira determinar onde gastar dinheiro em publicidade de campanha com a finalidade de ampliar as chances de vencer a eleição. Uma empresa de transporte aéreo pode designar as tripulações para os vôos da forma menos dispendiosa possível, certificando-se de que cada vôo será atendido e que as regulamentações do governo relativas à escala

das tripulações serão obedecidas. Um provedor de serviços da Internet talvez queira definir onde instalar recursos adicionais para servir de modo mais eficiente a seus clientes. Todos esses são exemplos de problemas que podem ser resolvidos com o uso da programação linear, que estudaremos no Capítulo 29.

Embora alguns dos detalhes desses exemplos estejam além do escopo deste livro, fornecere-mos técnicas básicas que se aplicam a esses problemas e a essas áreas de problemas. Também mostraremos neste livro como resolver muitos problemas concretos, inclusive os seguintes:

- Temos um mapa rodoviário no qual a distância entre cada par de interseções adjacentes é marcada, e nossa meta é determinar a menor rota de uma interseção até outra. O número de rotas possíveis pode ser enorme, ainda que sejam descartadas as rotas que cruzam sobre si mesmas. Como escolher qual de todas as rotas possíveis é a mais curta? Aqui, modelamos o mapa rodoviário (que é ele próprio um modelo das estradas reais) como um grafo (o que veremos no Capítulo 10 e no Apêndice B) e desejamos encontrar o caminho mais curto de um vértice até outro no grafo. Veremos como resolver esse problema de forma eficiente no Capítulo 24.
- Temos uma seqüência  $\langle A_1, A_2, \dots, A_n \rangle$  de  $n$  matrizes e desejamos determinar seu produto  $A_1 A_2 \dots A_n$ . Como a multiplicação de matrizes é associativa, existem várias ordens de multiplicação válidas. Por exemplo, se  $n = 4$ , podemos executar as multiplicações de matrizes como se o produto estivesse entre parênteses em qualquer das seguintes ordens:  $(A_1(A_2(A_3A_4)))$ ,  $(A_1((A_2A_3)A_4))$ ,  $((A_1A_2)(A_3A_4))$ ,  $((A_1(A_2A_3))A_4)$  ou  $((((A_1A_2)A_3)A_4))$ . Se essas matrizes forem todas quadradas (e portanto tiverem o mesmo tamanho), a ordem de multiplicação não afetará o tempo de duração das multiplicações de matrizes. Porém, se essas matrizes forem de tamanhos diferentes (ainda que seus tamanhos sejam compatíveis para a multiplicação de matrizes), então a ordem de multiplicação pode fazer uma diferença muito grande. O número de ordens de multiplicação possíveis é exponencial em  $n$ , e assim tentar todas as ordens possíveis pode levar um tempo muito longo. Veremos no Capítulo 15 como usar uma técnica geral conhecida como programação dinâmica para resolver esse problema de modo muito mais eficiente.
- Temos uma equação  $ax \equiv b \pmod{n}$ , onde  $a$ ,  $b$  e  $n$  são inteiros, e desejamos encontrar todos os inteiros  $x$ , módulo  $n$ , que satisfazem à equação. Pode haver zero, uma ou mais de uma solução. Podemos simplesmente experimentar  $x = 0, 1, \dots, n - 1$  em ordem, mas o Capítulo 31 mostra um método mais eficiente.
- Temos  $n$  pontos no plano e desejamos encontrar a envoltória convexa desses pontos. A envoltória convexa é o menor polígono convexo que contém os pontos. Intuitivamente, podemos imaginar que cada ponto é representado por um prego fixado a uma tábua. A envoltória convexa seria representada por um elástico apertado que cercasse todos os pregos. Cada prego pelo qual o elástico passa é um vértice da envoltória convexa. (Veja um exemplo na Figura 33.6.) Quaisquer dos  $2^n$  subconjuntos dos pontos poderiam ser os vértices da envoltória convexa. Saber quais pontos são vértices da envoltória convexa não é suficiente, pois também precisamos conhecer a ordem em que eles aparecem. Portanto, há muitas escolhas para os vértices da envoltória convexa. O Capítulo 33 apresenta dois bons métodos para se encontrar a envoltória convexa.

Essas listas estão longe de esgotar os exemplos (como você novamente já deve ter imaginado pelo peso deste livro), mas exibem duas características comuns a muitos algoritmos interessantes.

1. Existem muitas soluções candidatas, a maioria das quais não é aquilo que desejamos. Encontrar a solução que queremos pode representar um desafio.

- Existem aplicações práticas. Dos problemas da lista anterior, o caminho mais curto fornece os exemplos mais fáceis. Uma empresa de transportes que utiliza caminhões ou vagões ferroviários tem interesse financeiro em encontrar os caminhos mais curtos em uma rede ferroviária ou rodoviária, porque percursos menores resultam em menor trabalho e menor consumo de combustível. Ou então, um nó de roteamento na Internet pode precisar encontrar o caminho mais curto através da rede, a fim de rotear uma mensagem com rapidez.

## Estruturas de dados

Este livro também contém várias estruturas de dados. Uma *estrutura de dados* é um meio para armazenar e organizar dados com o objetivo de facilitar o acesso e as modificações. Nenhuma estrutura de dados única funciona bem para todos os propósitos, e assim é importante conhecer os pontos fortes e as limitações de várias delas.

## Técnica

Embora possa usar este livro como um “livro de receitas” para algoritmos, algum dia você poderá encontrar um problema para o qual não seja possível descobrir prontamente um algoritmo publicado (muitos dos exercícios e problemas deste livro, por exemplo!). Este livro lhe ensinará técnicas de projeto e análise de algoritmos, de forma que você possa desenvolver algoritmos por conta própria, mostrar que eles fornecem a resposta correta e entender sua eficiência.

## Problemas difíceis

A maior parte deste livro trata de algoritmos eficientes. Nossa medida habitual de eficiência é a velocidade, isto é, quanto tempo um algoritmo demora para produzir seu resultado. Porém, existem alguns problemas para os quais não se conhece nenhuma solução eficiente. O Capítulo 34 estuda um subconjunto interessante desses problemas, conhecidos como NP-completos.

Por que os problemas NP-completos são interessantes? Primeiro, embora ainda não tenha sido encontrado nenhum algoritmo eficiente para um problema NP-completo, ninguém jamais provou que não é possível existir um algoritmo eficiente para esse fim. Em outras palavras, desconhecemos se existem ou não algoritmos eficientes para problemas NP-completos. Em segundo lugar, o conjunto de problemas NP-completos tem a propriedade notável de que, se existe um algoritmo eficiente para qualquer um deles, então existem algoritmos eficientes para todos. Esse relacionamento entre os problemas NP-completos torna a falta de soluções eficientes ainda mais torturante. Em terceiro lugar, vários problemas NP-completos são semelhantes, mas não idênticos, a problemas para os quais conhecemos algoritmos eficientes. Uma pequena mudança no enunciado do problema pode provocar uma grande alteração na eficiência do melhor algoritmo conhecido.

É valioso conhecer os problemas NP-completos, porque alguns deles surgem com freqüência surpreendente em aplicações reais. Se for chamado a produzir um algoritmo eficiente para um problema NP-completo, é provável que você perca muito tempo em uma busca infrutífera. Por outro lado, se conseguir mostrar que o problema é NP-completo, você poderá em vez disso dedicar seu tempo ao desenvolvimento de um algoritmo eficiente que ofereça uma solução boa, embora não seja a melhor possível.

Como um exemplo concreto, considere uma empresa de transporte por caminhão com um armazém central. A cada dia, ela carrega o caminhão no armazém e o envia a diversos locais para efetuar entregas. No final do dia, o caminhão tem de estar de volta ao armazém, a fim de ser preparado para receber a carga do dia seguinte. Para reduzir custos, a empresa deve selecionar uma ordem de paradas de entrega que represente a menor distância total a ser percorrida pelo caminhão. Esse problema é o famoso “problema do caixeiro-viajante”, e é NP-completo. Ele não

tem nenhum algoritmo eficiente conhecido. Contudo, sob certas hipóteses, há algoritmos eficientes que fornecem uma distância total não muito acima da menor possível. O Capítulo 35 discute esses “algoritmos de aproximação”.

## Exercícios

### 1.1-1

Forneça um exemplo real no qual apareça um dos problemas computacionais a seguir: ordenação, determinação da melhor ordem para multiplicação de matrizes ou localização da envoltória convexa.

### 1.1-2

Além da velocidade, que outras medidas de eficiência poderiam ser usadas em uma configuração real?

### 1.1-3

Selecione uma estrutura de dados que você já tenha visto antes e discuta seus pontos fortes e suas limitações.

### 1.1-4

Em que aspectos os problemas do caminho mais curto e do caixeiro-viajante anteriores são semelhantes? Em que aspectos eles são diferentes?

### 1.1-5

Mostre um problema real no qual apenas a melhor solução servirá. Em seguida, apresente um problema em que baste uma solução que seja “aproximadamente” a melhor.

## 1.2 Algoritmos como uma tecnologia

Suponha que os computadores fossem infinitamente rápidos e que a memória do computador fosse livre. Você teria alguma razão para estudar algoritmos? A resposta é sim, se não por outra razão, pelo menos porque você ainda gostaria de demonstrar que o método da sua solução termina, e o faz com a resposta correta.

Se os computadores fossem infinitamente rápidos, qualquer método correto para resolver um problema serviria. É provável que você quisesse que sua implementação estivesse dentro dos limites da boa prática de engenharia de software (isto é, que ela fosse bem documentada e projetada) mas, com maior freqüência, você utilizaria o método que fosse o mais fácil de implementar.

É claro que os computadores podem ser rápidos, mas não são infinitamente rápidos. A memória pode ser de baixo custo, mas não é gratuita. Assim, o tempo de computação é um recurso limitado, bem como o espaço na memória. Esses recursos devem ser usados de forma sensata, e algoritmos eficientes em termos de tempo ou espaço ajudarão você a usá-los.

## Eficiência

Algoritmos criados para resolver o mesmo problema muitas vezes diferem de forma drástica em sua eficiência. Essas diferenças podem ser muito mais significativas que as diferenças relativas a hardware e software.

Veremos no Capítulo 2, como exemplo, dois algoritmos para ordenação. O primeiro, conhecido como *ordenação por inserção*, leva um tempo aproximadamente igual a  $c_1 n^2$  para ordenar  $n$  itens, onde  $c_1$  é uma constante que não depende de  $n$ . Isto é, ela demora um tempo aproximadamente proporcional a  $n^2$ . O segundo, de *ordenação por intercalação*, leva um tempo aproximadamente igual a  $c_2 n \lg n$ , onde  $\lg n$  representa  $\log_2 n$  e  $c_2$  é outra constante que tam-

bém não depende de  $n$ . A ordenação por inserção normalmente tem um fator constante menor que a ordenação por intercalação; e assim,  $c_1 < c_2$ . Veremos que os fatores constantes podem ser muito menos significativos no tempo de execução que a dependência do tamanho da entrada  $n$ . Onde a ordenação por intercalação tem um fator  $\lg n$  em seu tempo de execução, a ordenação por inserção tem um fator  $n$ , que é muito maior. Embora a ordenação por inserção em geral seja mais rápida que a ordenação por intercalação para pequenos tamanhos de entradas, uma vez que o tamanho da entrada  $n$  se tornar grande o suficiente, a vantagem da ordenação por intercalação de  $\lg n$  contra  $n$  compensará com sobras a diferença em fatores constantes. Independente do quanto  $c_1$  seja menor que  $c_2$ , sempre haverá um ponto de passagem além do qual a ordenação por intercalação será mais rápida.

Como um exemplo concreto, vamos comparar um computador mais rápido (computador A) que executa a ordenação por inserção com um computador mais lento (computador B) que executa a ordenação por intercalação. Cada um deles deve ordenar um arranjo de um milhão de números.

Suponha que o computador A execute um bilhão de instruções por segundo e o computador B execute apenas dez milhões de instruções por segundo; assim, o computador A será 100 vezes mais rápido que o computador B em capacidade bruta de computação. Para tornar a diferença ainda mais drástica, suponha que o programador mais astucioso do mundo codifique a ordenação por inserção em linguagem de máquina para o computador A, e que o código resultante exija  $2n^2$  instruções para ordenar  $n$  números. (Aqui,  $c_1 = 2$ .) Por outro lado, a ordenação por intercalação é programada para o computador B por um programador médio que utiliza uma linguagem de alto nível com um compilador ineficiente, com o código resultante totalizando  $50n \lg n$  instruções (de forma que  $c_2 = 50$ ). Para ordenar um milhão de números, o computador A demora

$$\frac{2 \cdot (10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} = 2000 \text{ segundos ,}$$

enquanto o computador B demora

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100 \text{ segundos .}$$

Usando um algoritmo cujo tempo de execução cresce mais lentamente, até mesmo com um compilador fraco, o computador B funciona 20 vezes mais rápido que o computador A! A vantagem da ordenação por intercalação é ainda mais pronunciada quando ordenamos dez milhões de números: onde a ordenação por inserção demora aproximadamente 2,3 dias, a ordenação por intercalação demora menos de 20 minutos. Em geral, à medida que o tamanho do problema aumenta, também aumenta a vantagem relativa da ordenação por intercalação.

## Algoritmos e outras tecnologias

O exemplo anterior mostra que os algoritmos, como o hardware de computadores, constituem uma **tecnologia**. O desempenho total do sistema depende da escolha de algoritmos eficientes tanto quanto da escolha de hardware rápido. Da mesma maneira que estão havendo rápidos avanços em outras tecnologias computacionais, eles também estão sendo obtidos em algoritmos.

Você poderia indagar se os algoritmos são verdadeiramente tão importantes nos computadores contemporâneos em comparação com outras tecnologias avançadas, como:

- Hardware com altas taxas de clock, pipelines e arquiteturas superescalares.
- Interfaces gráficas do usuário (GUIs) intuitivas e fáceis de usar.

- Sistemas orientados a objetos.
- Redes locais e remotas.

A resposta é sim. Embora existam algumas aplicações que não exigem explicitamente conteúdo algorítmico no nível da aplicação (por exemplo, algumas aplicações simples baseadas na Web), a maioria também requer um certo grau de conteúdo algorítmico por si só. Por exemplo, considere um serviço da Web que determina como viajar de um local para outro. (Havia diversos serviços desse tipo no momento em que este livro foi escrito.) Sua implementação dependeria de hardware rápido, de uma interface gráfica do usuário, de redes remotas e também, possivelmente, de orientação a objetos. Contudo, ele também exigiria algoritmos para certas operações, como localização de rotas (talvez empregando um algoritmo de caminho mais curto), interpretação de mapas e interpolação de endereços.

Além disso, até mesmo uma aplicação que não exige conteúdo algorítmico no nível da aplicação depende muito de algoritmos. Será que a aplicação depende de hardware rápido? O projeto de hardware utilizou algoritmos. A aplicação depende de interfaces gráficas do usuário? O projeto de qualquer GUI depende de algoritmos. A aplicação depende de rede? O roteamento em redes depende muito de algoritmos. A aplicação foi escrita em uma linguagem diferente do código de máquina? Então, ela foi processada por um compilador, um interpretador ou um assembler, e todos fazem uso extensivo de algoritmos. Os algoritmos formam o núcleo da maioria das tecnologias usadas em computadores contemporâneos.

Além disso, com a capacidade cada vez maior dos computadores, nós os utilizamos para resolver problemas maiores do que nunca. Como vimos na comparação anterior entre ordenação por inserção e ordenação por intercalação, em problemas de tamanhos maiores, as diferenças na eficiência dos algoritmos se tornam particularmente importantes.

Uma sólida base de conhecimento e técnica de algoritmos é uma característica que separa os programadores verdadeiramente qualificados dos novatos. Com a moderna tecnologia computacional, você pode executar algumas tarefas sem saber muito sobre algoritmos; porém, com uma boa base em algoritmos, é possível fazer muito, muito mais.

## Exercícios

### 1.2-1

Forneça um exemplo de aplicação que exige conteúdo algorítmico no nível da aplicação e discuta a função dos algoritmos envolvidos.

### 1.2-2

Vamos supor que estamos comparando implementações de ordenação por inserção e ordenação por intercalação na mesma máquina. Para entradas de tamanho  $n$ , a ordenação por inserção é executada em  $8n^2$  etapas, enquanto a ordenação por intercalação é executada em  $64n \lg n$  etapas. Para que valores de  $n$  a ordenação por inserção supera a ordenação por intercalação?

### 1.2-3

Qual é o menor valor de  $n$  tal que um algoritmo cujo tempo de execução é  $100n^2$  funciona mais rápido que um algoritmo cujo tempo de execução é  $2^n$  na mesma máquina?

## Problemas

### 1-1 Comparação entre tempos de execução

Para cada função  $f(n)$  e cada tempo  $t$  na tabela a seguir, determine o maior tamanho  $n$  de um problema que pode ser resolvido no tempo  $t$ , supondo-se que o algoritmo para resolver o problema demore  $f(n)$  microssegundos.

	1 segundo	1 minuto	1 hora	1 dia	1 mês	1 ano	1 século
$\lg n$							
$\sqrt{n}$							
$n$							
$n \lg n$							
$n^2$							
$n^3$							
$2^n$							
$n!$							

## Notas do capítulo

Existem muitos textos excelentes sobre o tópico geral de algoritmos, inclusive os de Aho, Hopcroft e Ullman [5, 6], Baase e Van Gelder [26], Brassard e Bratley [46, 47], Goodrich e Tamassia [128], Horowitz, Sahni e Rajasekaran [158], Kingston [179], Knuth [182, 183, 185], Kozen [193], Manber [210], Mehlhorn [217, 218, 219], Purdom e Brown [252], Reingold, Nievergelt e Deo [257], Sedgewick [269], Skiena [280] e Wilf [315]. Alguns dos aspectos mais práticos do projeto de algoritmos são discutidos por Bentley [39, 40] e Gonnet [126]. Pesquisas sobre o campo dos algoritmos também podem ser encontradas no Handbook of Theoretical Computer Science, Volume A [302] e no CRC Handbook on Algorithms and Theory of Computation [24]. Avaliações dos algoritmos usados em biologia computacional podem ser encontradas em livros-texto de Gusfield [136], Pevzner [240], Setubal e Medinas [272], e Waterman [309].

---

## *Capítulo 2*

# *Conceitos básicos*

Este capítulo tem o objetivo de familiarizá-lo com a estrutura que usaremos em todo o livro para refletir sobre o projeto e a análise de algoritmos. Ele é autônomo, mas inclui diversas referências ao material que será apresentado nos Capítulos 3 e 4. (E também contém diversos somatórios, que o Apêndice A mostra como resolver.)

Começaremos examinando o problema do algoritmo de ordenação por inserção para resolver o problema de ordenação apresentado no Capítulo 1. Definiremos um “pseudocódigo” que deverá ser familiar aos leitores que tenham estudado programação de computadores, e o empregaremos com a finalidade de mostrar como serão especificados nossos algoritmos. Tendo especificado o algoritmo, demonstraremos então que ele efetua a ordenação corretamente e analisaremos seu tempo de execução. A análise introduzirá uma notação centrada no modo como o tempo aumenta com o número de itens a serem ordenados. Seguindo nossa discussão da ordenação por inserção, introduziremos a abordagem de dividir e conquistar para o projeto de algoritmos e a utilizaremos com a finalidade de desenvolver um algoritmo chamado ordenação por intercalação. Terminaremos com uma análise do tempo de execução da ordenação por intercalação.

### **2.1 Ordenação por inserção**

Nosso primeiro algoritmo, o de ordenação por inserção, resolve o *problema de ordenação* introduzido no Capítulo 1:

**Entrada:** Uma seqüência de  $n$  números  $\langle a_1, a_2, \dots, a_n \rangle$ .

**Saída:** Uma permutação (reordenação)  $\langle a'_1, a'_2, \dots, a'_n \rangle$  da seqüência de entrada, tal que  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

Os números que desejamos ordenar também são conhecidos como *chaves*.

Neste livro, descreveremos tipicamente algoritmos como programas escritos em um *pseudocódigo* muito semelhante em vários aspectos a C, Pascal ou Java. Se já conhece qualquer dessas linguagens, você deverá ter pouca dificuldade para ler nossos algoritmos. O que separa o pseudocódigo do código “real” é que, no pseudocódigo, empregamos qualquer método expressivo para especificar de forma mais clara e concisa um dado algoritmo. Às vezes, o método mais claro é a linguagem comum; assim, não se surpreenda se encontrar uma frase ou sentença em nosso idioma (ou em inglês) embutida no interior de uma seção de código “real”. Outra diferen-

ça entre o pseudocódigo e o código real é que o pseudocódigo em geral não se relaciona com questões de engenharia de software. As questões de abstração de dados, modularidade e tratamento de erros são freqüentemente ignoradas, com a finalidade de transmitir a essência do algoritmo de modo mais conciso.

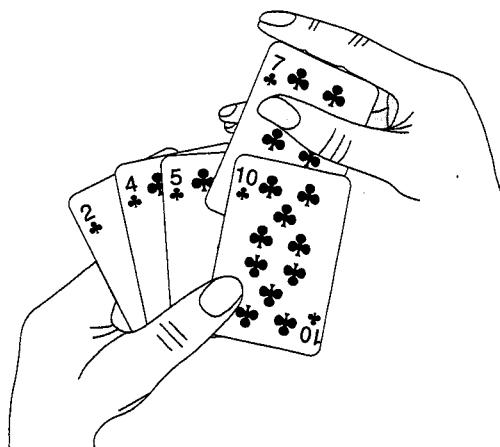


FIGURA 2.1 Ordenando cartas com o uso da ordenação por inserção

Começaremos com a *ordenação por inserção*, um algoritmo eficiente para ordenar um número pequeno de elementos. A ordenação por inserção funciona da maneira como muitas pessoas ordenam as cartas em um jogo de bridge ou pôquer. Iniciaremos com a mão esquerda vazia e as cartas viradas com a face para baixo na mesa. Em seguida, removeremos uma carta de cada vez da mesa, inserindo-a na posição correta na mão esquerda. Para encontrar a posição correta de uma carta, vamos compará-la a cada uma das cartas que já estão na mão, da direita para a esquerda, como ilustra a Figura 2.1. Em cada instante, as cartas seguras na mão esquerda são ordenadas; essas cartas eram originalmente as cartas superiores da pilha na mesa.

Nosso pseudocódigo para ordenação por inserção é apresentado como um procedimento chamado **INSERTION-SORT**, que toma como parâmetro um arranjo  $A[1..n]$  contendo uma seqüência de comprimento  $n$  que deverá ser ordenada. (No código, o número  $n$  de elementos em  $A$  é denotado por *comprimento*[ $A$ ].) Os números da entrada são *ordenados no local*: os números são reorganizados dentro do arranjo  $A$ , com no máximo um número constante deles armazenado fora do arranjo em qualquer instante. O arranjo de entrada  $A$  conterá a seqüência de saída ordenada quando **INSERTION-SORT** terminar.

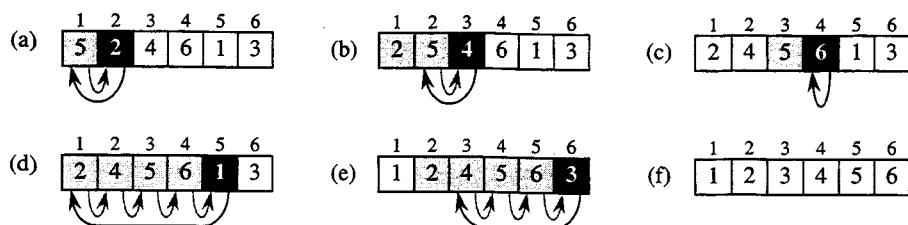


FIGURA 2.2 A operação de **INSERTION-SORT** sobre o arranjo  $A = \langle 5, 2, 4, 6, 1, 3 \rangle$ . Os índices do arranjo aparecem acima dos retângulos e os valores armazenados nas posições do arranjo aparecem dentro dos retângulos. (a)–(e) As iterações do loop **for** das linhas 1 a 8. Em cada iteração, o retângulo preto contém a chave obtida de  $A[j]$ , que é comparada aos valores contidos nos retângulos sombreados à sua esquerda, no teste da linha 5. Setas sombreadas mostram os valores do arranjo deslocados uma posição à direita na linha 6, e setas pretas indicam para onde a chave é deslocada na linha 8. (f) O arranjo ordenado final

```

INSERTION-SORT( $A$ )
1 for  $j \leftarrow 2$  to  $\text{comprimento}[A]$ 
2   do  $chave \leftarrow A[j]$ 
3      $\triangleright$  Inserir  $A[j]$  na seqüência ordenada  $A[1..j - 1]$ .
4      $i \leftarrow j - 1$ 
5     while  $i > 0$  e  $A[i] > chave$ 
6       do  $A[i + 1] \leftarrow A[i]$ 
7        $i \leftarrow i - 1$ 
8    $A[i + 1] \leftarrow chave$ 

```

## Loops invariantes e a correção da ordenação por inserção

A Figura 2.2 mostra como esse algoritmo funciona para  $A = \langle 5, 2, 4, 6, 1, 3 \rangle$ . O índice  $j$  indica a “carta atual” sendo inserida na mão. No início de cada iteração do loop **for** “externo”, indexado por  $j$ , o subarranjo que consiste nos elementos  $A[1..j - 1]$  constitui a mão atualmente ordenada, e os elementos  $A[j + 1..n]$  correspondem à pilha de cartas ainda na mesa. Na verdade, os elementos  $A[1..j - 1]$  são os elementos que estavam *originalmente* nas posições de 1 a  $j - 1$ , mas agora em seqüência ordenada. Enunciamos formalmente essas propriedades de  $A[1..j - 1]$  como um **loop invariante**:

No começo de cada iteração do loop **for** das linhas 1 a 8, o subarranjo  $A[1..j - 1]$  consiste nos elementos contidos originalmente em  $A[1..j - 1]$ , mas em seqüência ordenada.

Usamos loops invariantes para nos ajudar a entender por que um algoritmo é correto. Devemos mostrar três detalhes sobre um loop invariante:

**Inicialização:** Ele é verdadeiro antes da primeira iteração do loop.

**Manutenção:** Se for verdadeiro antes de uma iteração do loop, ele permanecerá verdadeiro antes da próxima iteração.

**Término:** Quando o loop termina, o invariante nos fornece uma propriedade útil que ajuda a mostrar que o algoritmo é correto.

Quando as duas primeiras propriedades são válidas, o loop invariante é verdadeiro antes de toda iteração do loop. Note a semelhança em relação à indução matemática; nesta última, para provar que uma propriedade é válida, você demonstra um caso básico e uma etapa indutiva. Aqui, mostrar que o invariante é válido antes da primeira iteração é equivalente ao caso básico, e mostrar que o invariante é válido de uma iteração para outra equivale à etapa indutiva.

A terceira propriedade talvez seja a mais importante, pois estamos usando o loop invariante para mostrar a correção. Ela também difere do uso habitual da indução matemática, em que a etapa indutiva é usada indefinidamente; aqui, paramos a “indução” quando o loop termina.

Vamos ver como essas propriedades são válidas para ordenação por inserção:

**Inicialização:** Começamos mostrando que o loop invariante é válido antes da primeira iteração do loop, quando  $j = 2$ .<sup>1</sup> Então, o subarranjo  $A[1..j - 1]$  consiste apenas no único elemento  $A[1]$ , que é de fato o elemento original em  $A[1]$ . Além disso, esse subarranjo é ordenado (de forma trivial, é claro), e isso mostra que o loop invariante é válido antes da primeira iteração do loop.

---

<sup>1</sup> Quando o loop é um loop **for**, o momento em que verificamos o loop invariante imediatamente antes da primeira iteração ocorre logo após a atribuição inicial à variável do contador de loop e imediatamente antes do primeiro teste no cabeçalho do loop. No caso de INSERTION-SORT, esse instante ocorre após a atribuição de 2 à variável  $j$ , mas antes do primeiro teste para verificar se  $j \leq \text{comprimento}[A]$ .

**Manutenção:** Em seguida, examinamos a segunda propriedade: a demonstração de que cada iteração mantém o loop invariante. Informalmente, o corpo do loop **for** exterior funciona deslocando-se  $A[j-1], A[j-2], A[j-3]$  e daí por diante uma posição à direita, até ser encontrada a posição adequada para  $A[j]$  (linhas 4 a 7), e nesse ponto o valor de  $A[j]$  é inserido (linha 8). Um tratamento mais formal da segunda propriedade nos obrigaria a estabelecer e mostrar um loop invariante para o loop **while** “interno”. Porém, nesse momento, preferimos não nos prender a tal formalismo, e assim contamos com nossa análise informal para mostrar que a segunda propriedade é válida para o loop exterior.

**Término:** Finalmente, examinamos o que ocorre quando o loop termina. No caso da ordenação por inserção, o loop **for** externo termina quando  $j$  excede  $n$ , isto é, quando  $j = n + 1$ . Substituindo  $j$  por  $n + 1$  no enunciado do loop invariante, temos que o subarranjo  $A[1..n]$  consiste nos elementos originalmente contidos em  $A[1..n]$ , mas em seqüência ordenada. Contudo, o subarranjo  $A[1..n]$  é o arranjo inteiro! Desse modo, o arranjo inteiro é ordenado, o que significa que o algoritmo é correto.

Empregaremos esse método de loops invariantes para mostrar a correção mais adiante neste capítulo e também em outros capítulos.

## Convenções de pseudocódigo

Utilizaremos as convenções a seguir em nosso pseudocódigo.

1. O recuo (ou endentação) indica uma estrutura de blocos. Por exemplo, o corpo do loop **for** que começa na linha 1 consiste nas linhas 2 a 8, e o corpo do loop **while**\* que começa na linha 5 contém as linhas 6 e 7, mas não a linha 8. Nossa estilo de recuo também se aplica a instruções **if-then-else**. O uso de recuo em lugar de indicadores convencionais de estrutura de blocos, como instruções **begin** e **end**, reduz bastante a desordem ao mesmo tempo que preserva, ou até mesmo aumenta, a clareza.<sup>2</sup>
2. As construções de loops **while**, **for** e **repeat** e as construções condicionais **if**, **then** e **else** têm interpretações semelhantes às que apresentam em Pascal.<sup>3</sup> Porém, existe uma diferença sutil com respeito a loops **for**: em Pascal, o valor da variável do contador de loop é indefinido na saída do loop mas, neste livro, o contador do loop retém seu valor após a saída do loop. Desse modo, logo depois de um loop **for**, o valor do contador de loop é o valor que primeiro excedeu o limite do loop **for**. Usamos essa propriedade em nosso argumento de correção para a ordenação por inserção. O cabeçalho do loop **for** na linha 1 é **for**  $j \leftarrow 2$  **to** **comprimento**[ $A$ ], e assim, quando esse loop termina,  $j = \text{comprimento}[A] + 1$  (ou, de forma equivalente,  $j = n + 1$ , pois  $n = \text{comprimento}[A]$ ).
3. O símbolo “ $\triangleright$ ” indica que o restante da linha é um comentário.
4. Uma atribuição múltipla da forma  $i \leftarrow j \leftarrow e$  atribui às variáveis  $i$  e  $j$  o valor da expressão  $e$ ; ela deve ser tratada como equivalente à atribuição  $j \leftarrow e$  seguida pela atribuição  $i \leftarrow j$ .
5. Variáveis (como  $i$ ,  $j$  e *chave*) são locais para o procedimento dado. Não usaremos variáveis globais sem indicação explícita.

---

<sup>2</sup>Em linguagens de programação reais, em geral não é aconselhável usar o recuo sozinho para indicar a estrutura de blocos, pois os níveis de recuo são difíceis de descobrir quando o código se estende por várias páginas.

<sup>3</sup>A maioria das linguagens estruturadas em blocos tem construções equivalentes, embora a sintaxe exata possa diferir da sintaxe de Pascal.

\* Manteremos na edição brasileira os nomes das instruções e dos comandos de programação (destacados em negrito) em inglês, bem como os títulos dos algoritmos, conforme a edição original americana, a fim de facilitar o processo de conversão para uma linguagem de programação qualquer, caso necessário. Por exemplo, usaremos **while** em vez de **enquanto**. (N.T.)

6. Elementos de arranjos são acessados especificando-se o nome do arranjo seguido pelo índice entre colchetes. Por exemplo,  $A[i]$  indica o  $i$ -ésimo elemento do arranjo  $A$ . A notação “ $\dots$ ” é usada para indicar um intervalo de valores dentro de um arranjo. Desse modo,  $A[1 \dots j]$  indica o subarranjo de  $A$  que consiste nos  $j$  elementos  $A[1], A[2], \dots, A[j]$ .
7. Dados compostos estão organizados tipicamente em **objetos**, os quais são constituídos por **atributos** ou **campos**. Um determinado campo é acessado usando-se o nome do campo seguido pelo nome de seu objeto entre colchetes. Por exemplo, tratamos um arranjo como um objeto com o atributo **comprimento** indicando quantos elementos ele contém. Para especificar o número de elementos em um arranjo  $A$ , escrevemos  $comprimento[A]$ . Embora sejam utilizados colchetes para indexação de arranjos e atributos de objetos, normalmente ficará claro a partir do contexto qual a interpretação pretendida.

Uma variável que representa um arranjo ou um objeto é tratada como um ponteiro para os dados que representam o arranjo ou objeto. Para todos os campos  $f$  de um objeto  $x$ , a definição de  $y \leftarrow x$  causa  $f[y] = f[x]$ . Além disso, se definirmos agora  $f[x] \leftarrow 3$ , então daí em diante não apenas  $f[x] = 3$ , mas também  $f[y] = 3$ . Em outras palavras,  $x$  e  $y$  apontarão para (“serão”) o mesmo objeto após a atribuição  $y \leftarrow x$ .

Às vezes, um ponteiro não fará referência a nenhum objeto. Nesse caso, daremos a ele o valor especial NIL.

8. Parâmetros são passados a um procedimento **por valor**: o procedimento chamado recebe sua própria cópia dos parâmetros e, se ele atribuir um valor a um parâmetro, a mudança *não* será vista pela rotina de chamada. Quando objetos são passados, o ponteiro para os dados que representam o objeto é copiado, mas os campos do objeto não o são. Por exemplo, se  $x$  é um parâmetro de um procedimento chamado, a atribuição  $x \leftarrow y$  dentro do procedimento chamado não será visível para o procedimento de chamada. Contudo, a atribuição  $f[x] \leftarrow 3$  será visível.
9. Os operadores booleanos “e” e “ou” são operadores de **curto-circuito**. Isto é, quando avaliamos a expressão “ $x$  e  $y$ ”, avaliamos primeiro  $x$ . Se  $x$  for avaliado como FALSE, então a expressão inteira não poderá ser avaliada como TRUE, e assim não avaliaremos  $y$ . Se, por outro lado,  $x$  for avaliado como TRUE, teremos de avaliar  $y$  para determinar o valor da expressão inteira. De forma semelhante, na expressão “ $x$  ou  $y$ ”, avaliamos a expressão  $y$  somente se  $x$  for avaliado como FALSE. Os operadores de curto-circuito nos permitem escrever expressões booleanas como “ $x \dots \text{NIL}$  e  $f[x] = y$ ” sem nos preocuparmos com o que acontece ao tentarmos avaliar  $f[x]$  quando  $x$  é NIL.

## Exercícios

### 2.1-1

Usando a Figura 2.2 como modelo, ilustre a operação de INSERTION-SORT no arranjo  $A = \langle 31, 41, 59, 26, 41, 58 \rangle$ .

### 2.1-2

Reescreva o procedimento INSERTION-SORT para ordenar em ordem não crescente, em vez da ordem não decrescente.

### 2.1-3

Considere o **problema de pesquisa**:

**Entrada:** Uma sequência de  $n$  números  $A = \langle a_1, a_2, \dots, a_n \rangle$  e um valor  $v$ .

**Saída:** Um índice  $i$  tal que  $v = A[i]$  ou o valor especial NIL, se  $v$  não aparecer em  $A$ .

Escreva o pseudocódigo para **pesquisa linear**, que faça a varredura da seqüência, procurando por  $v$ . Usando um loop invariante, prove que seu algoritmo é correto. Certifique-se de que seu loop invariante satisfaz às três propriedades necessárias.

### 2.1-4

Considere o problema de somar dois inteiros binários de  $n$  bits, armazenados em dois arranjos de  $n$  elementos  $A$  e  $B$ . A soma dos dois inteiros deve ser armazenada em forma binária em um arranjo de  $(n + 1)$  elementos  $C$ . Enuncie o problema de modo formal e escreva o pseudocódigo para somar os dois inteiros.

## 2.2 Análise de algoritmos

**Analizar** um algoritmo significa prever os recursos de que o algoritmo necessitará. Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, mas com freqüência é o tempo de computação que desejamos medir. Em geral, pela análise de vários algoritmos candidatos para um problema, pode-se identificar facilmente um algoritmo mais eficiente. Essa análise pode indicar mais de um candidato viável, mas vários algoritmos de qualidade inferior em geral são descartados no processo.

Antes de podermos analisar um algoritmo, devemos ter um modelo da tecnologia de implementação que será usada, inclusive um modelo dos recursos dessa tecnologia e seus custos. Na maior parte deste livro, faremos a suposição de um modelo de computação genérico com um único processador, a **RAM** (*random-access machine* – máquina de acesso aleatório), como nossa tecnologia de implementação e entenderemos que nossos algoritmos serão implementados sob a forma de programas de computador. No modelo de RAM, as instruções são executadas uma após outra, sem operações concorrentes (ou simultâneas). Porém, em capítulos posteriores teremos oportunidade de investigar modelos de hardware digital.

No sentido estrito, devemos definir com precisão as instruções do modelo de RAM e seus custos. Porém, isso seria tedioso e daria pouco percepção do projeto e da análise de algoritmos. Também devemos ter cuidado para não abusar do modelo de RAM. Por exemplo, e se uma RAM tivesse uma instrução de ordenação? Então, poderíamos ordenar com apenas uma instrução. Tal RAM seria irreal, pois os computadores reais não têm tais instruções. Portanto, nosso guia é o modo como os computadores reais são projetados. O modelo de RAM contém instruções comumente encontradas em computadores reais: instruções aritméticas (soma, subtração, multiplicação, divisão, resto, piso, teto), de movimentação de dados (carregar, armazenar, copiar) e de controle (desvio condicional e incondicional, chamada e retorno de sub-rotinas). Cada uma dessas instruções demora um período constante.

Os tipos de dados no modelo de RAM são inteiros e de ponto flutuante. Embora normalmente não nos preocupemos com a precisão neste livro, em algumas aplicações a precisão é crucial. Também supomos um limite sobre o tamanho de cada palavra de dados. Por exemplo, ao trabalharmos com entradas de tamanho  $n$ , em geral supomos que os inteiros são representados por  $c \lg n$  bits para alguma constante  $c \geq 1$ . Exigimos  $c \geq 1$  para que cada palavra possa conter o valor de  $n$ , permitindo-nos indexar os elementos de entradas individuais, e limitamos  $c$  a uma constante para que o tamanho da palavra não cresça arbitrariamente. (Se o tamanho da palavra pudesse crescer arbitrariamente, seria possível armazenar enormes quantidades de dados em uma única palavra e operar sobre toda ela em tempo constante – claramente um cenário impraticável.)

Computadores reais contêm instruções não listadas anteriormente, e tais instruções representam uma área cinza no modelo de RAM. Por exemplo, a exponenciação é uma instrução de tempo constante? No caso geral, não; são necessárias várias instruções para calcular  $x^y$  quando  $x$  e  $y$  são números reais. Porém, em situações restritas, a exponenciação é uma operação de tempo constante. Muitos computadores têm uma instrução “deslocar à esquerda” que desloca em tempo constante os bits de um inteiro  $k$  posições à esquerda. Na maioria dos computadores, deslocar os bits de um inteiro uma posição à esquerda é equivalente a efetuar a multiplicação por 2.

Deslocar os bits  $k$  posições à esquerda é equivalente a multiplicar por  $2^k$ . Portanto, tais computadores podem calcular  $2^k$  em uma única instrução de tempo constante, deslocando o inteiro 1  $k$  posições à esquerda, desde que  $k$  não seja maior que o número de bits em uma palavra de computador. Procuraremos evitar essas áreas cinza no modelo de RAM, mas trataremos a computação de  $2^k$  como uma operação de tempo constante quando  $k$  for um inteiro positivo suficientemente pequeno.

No modelo de RAM, não tentaremos modelar a hierarquia da memória que é comum em computadores contemporâneos. Isto é, não modelaremos caches ou memória virtual (que é implementada com maior freqüência com paginação por demanda). Vários modelos computacionais tentam levar em conta os efeitos da hierarquia de memória, que às vezes são significativos em programas reais de máquinas reais. Alguns problemas neste livro examinam os efeitos da hierarquia de memória mas, em sua maioria, as análises neste livro não irão considerá-los.

Os modelos que incluem a hierarquia de memória são bem mais complexos que o modelo de RAM, de forma que pode ser difícil utilizá-los. Além disso, as análises do modelo de RAM em geral permitem previsões excelentes do desempenho em máquinas reais.

Até mesmo a análise de um algoritmo simples no modelo de RAM pode ser um desafio. As ferramentas matemáticas exigidas podem incluir análise combinatória, teoria das probabilidades, destreza em álgebra e a capacidade de identificar os termos mais significativos em uma fórmula. Tendo em vista que o comportamento de um algoritmo pode ser diferente para cada entrada possível, precisamos de um meio para resumir esse comportamento em fórmulas simples, de fácil compreensão.

Embora normalmente selezionemos apenas um único modelo de máquina para analisar um determinado algoritmo, ainda estaremos diante de muitas opções na hora de decidir como expressar nossa análise. Um objetivo imediato é encontrar um meio de expressão que seja simples de escrever e manipular, que mostre as características importantes de requisitos de recurso de um algoritmo e que suprima os detalhes tediosos.

## Análise da ordenação por inserção

O tempo despendido pelo procedimento INSERTION-SORT depende da entrada: a ordenação de mil números demora mais que a ordenação de três números. Além disso, INSERTION-SORT pode demorar períodos diferentes para ordenar duas seqüências de entrada do mesmo tamanho, dependendo do quanto elas já estejam ordenadas. Em geral, o tempo de duração de um algoritmo cresce com o tamanho da entrada; assim, é tradicional descrever o tempo de execução de um programa como uma função do tamanho de sua entrada. Para isso, precisamos definir os termos “tempo de execução” e “tamanho da entrada” com mais cuidado.

A melhor noção de *tamanho da entrada* depende do problema que está sendo estudado. No caso de muitos problemas, como a ordenação ou o cálculo de transformações discretas de Fourier, a medida mais natural é o *número de itens na entrada* – por exemplo, o tamanho do arranjo  $n$  para ordenação. Para muitos outros problemas, como a multiplicação de dois inteiros, a melhor medida do tamanho da entrada é o *número total de bits* necessários para representar a entrada em notação binária comum. Às vezes, é mais apropriado descrever o tamanho da entrada com dois números em lugar de um. Por exemplo, se a entrada para um algoritmo é um grafo, o tamanho da entrada pode ser descrito pelos números de vértices e arestas no grafo. Indicaremos qual medida de tamanho da entrada está sendo usada com cada problema que estudarmos.

O *tempo de execução* de um algoritmo em uma determinada entrada é o número de operações primitivas ou “etapas” executadas. É conveniente definir a noção de etapa (ou passo) de forma que ela seja tão independente da máquina quanto possível. Por enquanto, vamos adotar a visão a seguir. Um período constante de tempo é exigido para executar cada linha do nosso pseudocódigo. Uma única linha pode demorar um período diferente de outra linha, mas vamos considerar que cada execução da  $i$ -ésima linha leva um tempo  $c_i$ , onde  $c_i$  é uma constante. Esse pon-

to de vista está de acordo com o modelo de RAM, e também reflete o modo como o pseudocódigo seria implementado na maioria dos computadores reais.<sup>4</sup>

Na discussão a seguir, nossa expressão para o tempo de execução de INSERTION-SORT evoluirá desde uma fórmula confusa que utiliza todos os custos da instrução  $c_i$  até uma notação mais simples, mais concisa e mais facilmente manipulada. Essa notação mais simples também facilitará a tarefa de descobrir se um algoritmo é mais eficiente que outro.

Começaremos apresentando o procedimento INSERTION-SORT com o “custo” de tempo de cada instrução e o número de vezes que cada instrução é executada. Para cada  $j = 2, 3, \dots, n$ , onde  $n = \text{comprimento}[A]$ , seja  $t_j$  o número de vezes que o teste do loop `while` na linha 5 é executado para esse valor de  $j$ . Quando um loop `for` ou `while` termina da maneira usual (isto é, devido ao teste no cabeçalho loop), o teste é executado uma vez além do corpo do loop. Supomos que comentários não são instruções executáveis e, portanto, não demandam nenhum tempo.

INSERTION-SORT( $A$ )	<i>custo</i>	<i>vezes</i>
1 <b>for</b> $j \leftarrow 2$ <b>to</b> <i>comprimento</i> [ $A$ ]	$c_1$	$n$
2 <b>do</b> <i>chave</i> $\leftarrow A[j]$	$c_2$	$n - 1$
3    ▷ Inserir $A[j]$ na seqüência ordenada $A[1..j - 1]$ .	0	$n - 1$
4 <i>i</i> $\leftarrow j - 1$	$c_4$	$n - 1$
5 <b>while</b> $i > 0$ e $A[i] > \text{chave}$	$c_5$	$\sum_{j=2}^n t_j$
6 <b>do</b> $A[i + 1] \leftarrow A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
7 <i>i</i> $\leftarrow i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
8 <i>A</i> [ $i + 1$ ] $\leftarrow \text{chave}$	$c_8$	$n - 1$

O tempo de execução do algoritmo é a soma dos tempos de execução para cada instrução executada; uma instrução que demanda  $c_i$  passos para ser executada e é executada  $n$  vezes, contribuirá com  $c_i n$  para o tempo de execução total.<sup>5</sup> Para calcular  $T(n)$ , o tempo de execução de INSERTION-SORT, somamos os produtos das *colunas custo* e *vezes*, obtendo

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

Mesmo para entradas de um dado tamanho, o tempo de execução de um algoritmo pode depender de *qual* entrada desse tamanho é dada. Por exemplo, em INSERTION-SORT, o melhor caso ocorre se o arranjo já está ordenado. Para cada  $j = 2, 3, \dots, n$ , descobrimos então que  $A[i] \leq \text{chave}$  na linha 5 quando  $i$  tem seu valor inicial  $j - 1$ . Portanto,  $t_j = 1$  para  $j = 2, 3, \dots, n$ , e o tempo de execução do melhor caso é

$$T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1) \\ = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

<sup>4</sup>Há algumas sutilezas aqui. As etapas computacionais que especificamos em linguagem comum freqüentemente são variantes de um procedimento que exige mais que apenas uma quantidade constante de tempo. Por exemplo, mais adiante neste livro, poderíamos dizer “ordene os pontos pela coordenada  $x$ ” que, como veremos, demora mais que uma quantidade constante de tempo. Além disso, observe que uma instrução que chama uma sub-rotina demora um tempo constante, embora a sub-rotina, uma vez invocada, possa durar mais. Ou seja, separamos o processo de *chamar* a sub-rotina – passar parâmetros a ela etc. – do processo de *executar* a sub-rotina.

<sup>5</sup>Essa característica não se mantém necessariamente para um recurso como a memória. Uma instrução que referencia  $m$  palavras de memória e é executada  $n$  vezes não consome necessariamente  $mn$  palavras de memória no total.

Esse tempo de execução pode ser expresso como  $an + b$  para constantes  $a$  e  $b$  que dependem dos custos de instrução  $c_i$ ; assim, ele é uma **função linear** de  $n$ .

Se o arranjo estiver ordenado em ordem inversa – ou seja, em ordem decrescente –, resulta o pior caso. Devemos comparar cada elemento  $A[j]$  com cada elemento do subarranjo ordenado inteiro,  $A[1 .. j - 1]$ , e então  $t_j = j$  para  $2, 3, \dots, n$ . Observando que

$$\sum_{j=2}^n j = \frac{n(n-1)}{2} - 1$$

e

$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

(veremos no Apêndice A como resolver esses somatórios), descobrimos que, no pior caso, o tempo de execução de INSERTION-SORT é

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

Esse tempo de execução no pior caso pode ser expresso como  $an^2 + bn + c$  para constantes  $a, b$  e  $c$  que, mais uma vez, dependem dos custos de instrução  $c_i$ ; portanto, ele é uma **função quadrática** de  $n$ .

Em geral, como na ordenação por inserção, o tempo de execução de um algoritmo é fixo para uma determinada entrada, embora em capítulos posteriores devamos ver alguns algoritmos “aleatórios” interessantes, cujo comportamento pode variar até mesmo para uma entrada fixa.

## Análise do pior caso e do caso médio

Em nossa análise da ordenação por inserção, observamos tanto o melhor caso, no qual o arranjo de entrada já estava ordenado, quanto o pior caso, no qual o arranjo de entrada estava ordenado em ordem inversa. Porém, no restante deste livro, em geral nos concentraremos apenas na descoberta do **tempo de execução do pior caso**; ou seja, o tempo de execução mais longo para *qualquer* entrada de tamanho  $n$ . Apresentaremos três razões para essa orientação.

- O tempo de execução do pior caso de um algoritmo é um limite superior sobre o tempo de execução para qualquer entrada. Conhecê-lo nos dá uma garantia de que o algoritmo nunca irá demorar mais tempo. Não precisamos fazer nenhuma suposição baseada em fatos sobre o tempo de execução, e temos a esperança de que ele nunca seja muito pior.
- Para alguns algoritmos, o pior caso ocorre com bastante freqüência. Por exemplo, na pesquisa de um banco de dados em busca de um determinado fragmento de informação, o pior caso do algoritmo de pesquisa ocorrerá freqüentemente quando a informação não estiver presente no banco de dados. Em algumas aplicações de pesquisa, a busca de informações ausentes pode ser freqüente.

- Muitas vezes, o “caso médio” é quase tão ruim quanto o pior caso. Suponha que sejam escolhidos aleatoriamente  $n$  números e que se aplique a eles a ordenação por inserção. Quanto tempo irá demorar para se descobrir o lugar no subarranjo  $A[1..j-1]$  em que se deve inserir o elemento  $A[j]$ ? Em média, metade dos elementos em  $A[1..j-1]$  são menores que  $A[j]$ , e metade dos elementos são maiores. Assim, em média, verificamos que metade do subarranjo  $A[1..j-1]$ , e então  $t_j = j/2$ . Se desenvolvemos o tempo de execução do caso médio resultante, ele será uma função quadrática do tamanho da entrada, exatamente como o tempo de execução do pior caso.

Em alguns casos particulares, estaremos interessados no tempo de execução do **caso médio** ou **esperado** de um algoritmo. Contudo, um problema na realização de uma análise do caso médio é que pode não ser显而易见的, o que constitui uma entrada “média” para um determinado problema. Freqüentemente, iremos supor que todas as entradas de um dado tamanho são igualmente prováveis. Na prática, é possível que essa suposição seja violada, mas às vezes podemos utilizar um **algoritmo aleatório**, que efetua escolhas ao acaso, a fim de permitir uma análise probabilística.

## Ordem de crescimento

Usamos algumas abstrações simplificadoras para facilitar nossa análise do procedimento INSERTION-SORT. Primeiro, ignoramos o custo real de cada instrução, usando as constantes  $c_i$  para representar esses custos. Em seguida, observamos que até mesmo essas constantes nos oferecem mais detalhes do que realmente necessitamos: o tempo de execução do pior caso é  $an^2 + bn + c$  para constantes  $a, b$  e  $c$  que dependem dos custos de instrução  $c_i$ . Desse modo, ignoramos não apenas os custos reais de instrução, mas também os custos abstratos  $c_i$ .

Agora, faremos mais uma abstração simplificadora. É a **taxa de crescimento**, ou **ordem de crescimento**, do tempo de execução que realmente nos interessa. Assim, consideramos apenas o termo inicial de uma fórmula (por exemplo,  $an^2$ ), pois os termos de mais baixa ordem são relativamente insignificantes para grandes valores de  $n$ . Também ignoramos o coeficiente constante do termo inicial, tendo em vista que fatores constantes são menos significativos que a taxa de crescimento na determinação da eficiência computacional para grandes entradas. Portanto, escrevemos que a ordenação por inserção, por exemplo, tem um tempo de execução do pior caso igual a  $\Theta(n^2)$  (lido como “theta de  $n$  ao quadrado”). Usaremos informalmente neste capítulo a notação  $\Theta$ ; ela será definida com precisão no Capítulo 3.

Em geral, consideramos um algoritmo mais eficiente que outro se o tempo de execução do seu pior caso apresenta uma ordem de crescimento mais baixa. Essa avaliação pode ser incorreta para entradas pequenas; porém, para entradas suficientemente grandes, um algoritmo  $\Theta(n^2)$ , por exemplo, será executado mais rapidamente no pior caso que um algoritmo  $\Theta(n^3)$ .

## Exercícios

### 2.2-1

Expresse a função  $n^3/1000 - 100n^2 - 100n + 3$  em termos da notação  $\Theta$ .

### 2.2-2

Considere a ordenação de  $n$  números armazenados no arranjo  $A$ , localizando primeiro o menor elemento de  $A$  e permutando esse elemento com o elemento contido em  $A[1]$ . Em seguida, encontre o segundo menor elemento de  $A$  e o troque pelo elemento  $A[2]$ . Continue dessa maneira para os primeiros  $n - 1$  elementos de  $A$ . Escreva o pseudocódigo para esse algoritmo, conhecido como **ordenação por seleção**. Que loop invariante esse algoritmo mantém? Por que ele só precisa ser executado para os primeiros  $n - 1$  elementos, e não para todos os  $n$  elementos? Forneça os tempos de execução do melhor caso e do pior caso da ordenação por seleção em notação  $\Theta$ .

### 2.2-3

Considere mais uma vez a pesquisa linear (ver Exercício 2.1-3). Quantos elementos da seqüência de entrada precisam ser verificados em média, supondo-se que o elemento que está sendo procurado tenha a mesma probabilidade de ser qualquer elemento no arranjo? E no pior caso? Quais são os tempos de execução do caso médio e do pior caso da pesquisa linear em notação  $\Theta$ ? Justifique suas respostas.

### 2.2-4

Como podemos modificar praticamente qualquer algoritmo para ter um bom tempo de execução no melhor caso?

## 2.3 Projeto de algoritmos

Existem muitas maneiras de projetar algoritmos. A ordenação por inserção utiliza uma abordagem **incremental**: tendo ordenado o subarranjo  $A[1..j-1]$ , inserimos o elemento isolado  $A[j]$  em seu lugar apropriado, formando o subarranjo ordenado  $A[1..j]$ .

Nesta seção, examinaremos uma abordagem de projeto alternativa, conhecida como “dividir e conquistar”. Usaremos o enfoque de dividir e conquistar para projetar um algoritmo de ordenação cujo tempo de execução do pior caso é muito menor que o da ordenação por inserção. Uma vantagem dos algoritmos de dividir e conquistar é que seus tempos de execução são freqüentemente fáceis de determinar com a utilização de técnicas que serão introduzidas no Capítulo 4.

### 2.3.1 A abordagem de dividir e conquistar

Muitos algoritmos úteis são **recursivos** em sua estrutura: para resolver um dado problema, eles chamam a si mesmos recursivamente uma ou mais vezes para lidar com subproblemas intimamente relacionados. Em geral, esses algoritmos seguem uma abordagem de **dividir e conquistar**: eles desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

O paradigma de dividir e conquistar envolve três passos em cada nível da recursão:

**Dividir** o problema em um determinado número de subproblemas.

**Conquistar** os subproblemas, resolvendo-os recursivamente. Porém, se os tamanhos dos subproblemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta.

**Combinar** as soluções dadas aos subproblemas, a fim de formar a solução para o problema original.

O algoritmo de **ordenação por intercalação** a seguir obedece ao paradigma de dividir e conquistar. Intuitivamente, ele opera do modo ilustrado a seguir.

**Dividir:** Divide a seqüência de  $n$  elementos a serem ordenados em duas subseqüências de  $n/2$  elementos cada uma.

**Conquistar:** Classifica as duas subseqüências recursivamente, utilizando a ordenação por intercalação.

**Combinar:** Faz a intercalação das duas seqüências ordenadas, de modo a produzir a resposta ordenada.

A recursão “não funciona” quando a seqüência a ser ordenada tem comprimento 1, pois nesse caso não há nenhum trabalho a ser feito, tendo em vista que toda seqüência de comprimento 1 já está ordenada.

A operação chave do algoritmo de ordenação por intercalação é a intercalação de duas sequências ordenadas, no passo de “combinação”. Para executar a intercalação, usamos um procedimento auxiliar  $\text{MERGE}(A, p, q, r)$ , onde  $A$  é um arranjo e  $p, q$  e  $r$  são índices de enumeração dos elementos do arranjo, tais que  $p \leq q < r$ . O procedimento pressupõe que os subarranjos  $A[p .. q]$  e  $A[q + 1 .. r]$  estão em seqüência ordenada. Ele os *intercala* (ou *mescla*) para formar um único subarranjo ordenado que substitui o subarranjo atual  $A[p .. r]$ .

Nosso procedimento MERGE leva o tempo  $\Theta(n)$ , onde  $n = r - p + 1$  é o número de elementos que estão sendo intercalados, e funciona como a seguir. Retornando ao nosso exemplo de motivação do jogo de cartas, vamos supor que temos duas pilhas de cartas com a face para cima sobre uma mesa. Cada pilha está ordenada, com as cartas de menor valor em cima. Desejamos juntar as duas pilhas (fazendo a intercalação) em uma única pilha de saída ordenada, que ficará com a face para baixo na mesa. Nosso passo básico consiste em escolher a menor das duas cartas superiores nas duas pilhas viradas para cima, removê-la de sua pilha (o que irá expor uma nova carta superior) e colocar essa carta com a face voltada para baixo sobre a pilha de saída. Repetimos esse passo até uma pilha de entrada se esvaziar e, nesse momento, simplesmente pegamos a pilha de entrada restante e a colocamos virada para baixo sobre a pilha de saída. Em termos computacionais, cada passo básico demanda um tempo constante, pois estamos verificando apenas duas cartas superiores. Tendo em vista que executamos no máximo  $n$  passos básicos, a intercalação demorará um tempo  $\Theta(n)$ .

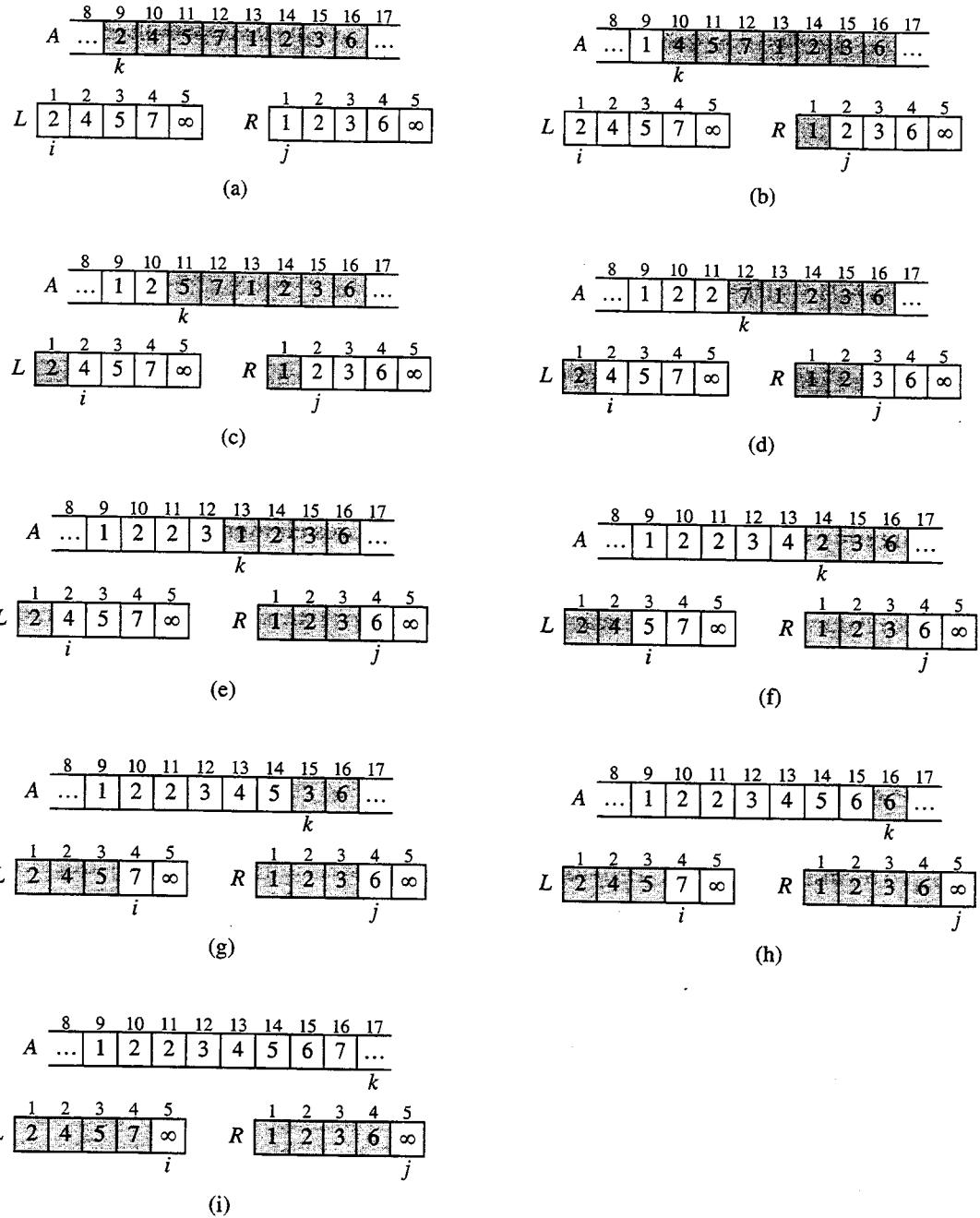
O pseudocódigo a seguir implementa a idéia anterior, mas tem uma alteração adicional que evita a necessidade de verificar se uma das pilhas está vazia em cada etapa básica. A idéia é colocar na parte inferior de cada pilha uma carta *sentinela*, que contém um valor especial que empregamos para simplificar nosso código. Aqui, usamos  $\infty$  como valor de sentinela de forma que, sempre que uma carta com  $\infty$  for exposta, ela não poderá ser a carta menor, a menos que ambas as pilhas tenham suas cartas sentinela expostas. Porém, uma vez que isso acontecer, todas as cartas que não são sentinelas já terão sido colocadas sobre a pilha de saída. Como sabemos com antecedência que exatamente  $r - p + 1$  cartas serão colocadas sobre a pilha de saída, podemos parar após a execução dessas muitas etapas básicas.

$\text{MERGE}(A, P, q, r)$

```

1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3  criar arranjos  $L[1..n_1 + 1]$  e  $R[1..n_2 + 1]$ 
4  for  $i \leftarrow 1$  to  $n_1$ 
5    do  $L[i] \leftarrow A[p + i - 1]$ 
6  for  $j \leftarrow 1$  to  $n_2$ 
7    do  $R[j] \leftarrow A[q + j]$ 
8   $L[n_1 + 1] \leftarrow \infty$ 
9   $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 for  $k \leftarrow p$  to  $r$ 
13   do if  $L[i] \leq R[j]$ 
14     then  $A[k] \leftarrow L[i]$ 
15      $i \leftarrow i + 1$ 
16   else  $A[k] \leftarrow R[j]$ 
17      $j \leftarrow j + 1$ 

```



**FIGURA 2.3** A operação das linhas 10 a 17 na chamada  $\text{MERGE}(A, 9, 12, 16)$  quando o subarranjo  $A[9..16]$  contém a seqüência  $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$ . Depois de copiar e inserir sentinelas, o arranjo  $L$  contém  $\langle 2, 4, 5, 7, \infty \rangle$ , e o arranjo  $R$  contém  $\langle 1, 2, 3, 6, \infty \rangle$ . Posições levemente sombreadas em  $A$  contêm seus valores finais, e posições levemente sombreadas em  $L$  e  $R$  contêm valores que ainda têm de ser copiados de volta em  $A$ . Juntas, as posições levemente sombreadas sempre incluem os valores contidos originalmente em  $A[9..16]$ , além das duas sentinelas. Posições fortemente sombreadas em  $A$  contêm valores que serão copiados, e posições fortemente sombreadas em  $L$  e  $R$  contêm valores que já foram copiados de volta em  $A$ . (a)-(h) Os arranjos  $A, L$  e  $R$  e seus respectivos índices  $k, i$ , e  $j$  antes de cada iteração do loop das linhas 12 a 17. (i) Os arranjos e índices no final. Nesse momento, o subarranjo em  $A[9..16]$  está ordenado, e as duas sentinelas em  $L$  e  $R$  são os dois únicos elementos nesses arranjos que não foram copiados em  $A$ .

Em detalhes, o procedimento MERGE funciona da maneira ilustrada a seguir. A linha 1 calcula o comprimento  $n_1$  do subarranjo  $A[p..q]$ , e a linha 2 calcula o comprimento  $n_2$  do subarranjo  $A[q+1..r]$ . Criamos os arranjos  $L$  e  $R$  (de “left” e “right”, ou “direita” e “esquerda” em inglês) de comprimentos  $n_1 + 1$  e  $n_2 + 1$ , respectivamente, na linha 3. O loop for das linhas 4 e 5 copia o

subarranjo  $A[p..q]$  em  $L[1..n_1]$ , e o loop **for** das linhas 6 e 7 copia o subarranjo  $A[q+1..r]$  em  $R[1..n_2]$ . As linhas 8 e 9 colocam as sentinelas nas extremidades dos arranjos  $L$  e  $R$ . As linhas 10 a 17, ilustradas na Figura 2.3, executam as  $r-p+1$  etapas básicas, mantendo o loop invariante a seguir:

No início de cada iteração do loop **for** das linhas 12 a 17, o subarranjo  $A[p..k-1]$  contém os  $k-p$  menores elementos de  $L[1..n_1+1]$  e  $R[1..n_2+1]$ , em seqüência ordenada.

Além disso,  $L[i]$  e  $R[j]$  são os menores elementos de seus arranjos que não foram copiados de volta em  $A$ .

Devemos mostrar que esse loop invariante é válido antes da primeira iteração do loop **for** das linhas 12 a 17, que cada iteração do loop mantém o invariante, e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

**Inicialização:** Antes da primeira iteração do loop, temos  $k=p$ , de forma que o subarranjo  $A[p..k-1]$  está vazio. Esse subarranjo vazio contém os  $k-p=0$  menores elementos de  $L$  e  $R$  e, desde que  $i=j=1$ , tanto  $L[i]$  quanto  $R[j]$  são os menores elementos de seus arranjos que não foram copiados de volta em  $A$ .

**Manutenção:** Para ver que cada iteração mantém o loop invariante, vamos supor primeiro que  $L[i] \leq R[j]$ . Então  $L[i]$  é o menor elemento ainda não copiado de volta em  $A$ . Como  $A[p..k-1]$  contém os  $k-p$  menores elementos, depois da linha 14 copiar  $L[i]$  em  $A[k]$ , o subarranjo  $A[p..k]$  conterá os  $k-p+1$  menores elementos. O incremento de  $k$  (na atualização do loop **for**) e de  $i$  (na linha 15) restabelece o loop invariante para a próxima iteração. Se, em vez disso,  $L[i] > R[j]$ , então as linhas 16 e 17 executam a ação apropriada para manter o loop invariante.

**Término:** No término,  $k=r+1$ . Pelo loop invariante, o subarranjo  $A[p..k-1]$ , que é  $A[p..r]$ , contém os  $k-p=r-p+1$  menores elementos de  $L[1..n_1+1]$  e  $R[1..n_2+1]$  em seqüência ordenada. Os arranjos  $L$  e  $R$  contêm juntos  $n_1+n_2+2=r-p+3$  elementos. Todos os elementos, exceto os dois maiores, foram copiados de volta em  $A$ , e esses dois maiores elementos são as sentinelas.

Para ver que o procedimento MERGE é executado no tempo  $\Theta(n)$ , onde  $n=r-p+1$ , observe que cada uma das linhas 1 a 3 e 8 a 11 demora um tempo constante, que os loops **for** das linhas 4 a 7 demoram o tempo  $\Theta(n_1+n_2)=\Theta(n)$ ,<sup>6</sup> e que há  $n$  iterações do loop **for** das linhas 12 a 17, cada uma demorando um tempo constante.

Agora podemos usar o procedimento MERGE como uma sub-rotina no algoritmo de ordenação por intercalação. O procedimento MERGE-SORT( $A, p, r$ ) ordena os elementos do subarranjo  $A[p..r]$ . Se  $p \geq r$ , o subarranjo tem no máximo um elemento e, portanto, já está ordenado. Caso contrário, a etapa de divisão simplesmente calcula um índice  $q$  que partitiona  $A[p..r]$  em dois subarranjos:  $A[p..q]$ , contendo  $\lceil n/2 \rceil$  elementos, e  $A[q+1..r]$ , contendo  $\lfloor n/2 \rfloor$  elementos.<sup>7</sup>

```
MERGE-SORT( $A, p, r$ )
1 if  $p < r$ 
2   then  $q \leftarrow \lfloor (p+r)/2 \rfloor$ 
3     MERGE-SORT( $A, p, q$ )
4     MERGE-SORT( $A, q+1, r$ )
5   MERGE( $A, p, q, r$ )
```

<sup>6</sup> Veremos no Capítulo 3 como interpretar formalmente equações contendo notação  $\Theta$ .

<sup>7</sup> A expressão  $\lceil x \rceil$  denota o menor inteiro maior que ou igual a  $x$ , e  $\lfloor x \rfloor$  denota o maior inteiro menor que ou igual a  $x$ . Essas notações são definidas no Capítulo 3. O caminho mais fácil para verificar que definir  $q$  como  $\lfloor (p+r)/2 \rfloor$  produz os subarranjos  $A[p..q]$  e  $A[q+1..r]$  de tamanhos  $\lceil n/2 \rceil$  e  $\lfloor n/2 \rfloor$ , respectivamente, é examinar os quatro casos que surgem dependendo do fato de cada valor de  $p$  e  $r$  ser ímpar ou par.

Para ordenar a seqüência  $A = \langle A[1], A[2], \dots, A[n] \rangle$  inteira, efetuamos a chamada inicial  $\text{MERGE-SORT}(A, 1, \text{comprimento}[A])$ , onde, mais uma vez,  $\text{comprimento}[A] = n$ . A Figura 2.4 ilustra a operação do procedimento de baixo para cima quando  $n$  é uma potência de 2. O algoritmo consiste em intercalar pares de seqüências de um item para formar seqüências ordenadas de comprimento 2, intercalar pares de seqüências de comprimento 2 para formar seqüências ordenadas de comprimento 4 e assim por diante, até duas seqüências de comprimento  $n/2$  serem intercaladas para formar a seqüência ordenada final de comprimento  $n$ .

### **2.3.2 Análise de algoritmos de dividir e conquistar**

Quando um algoritmo contém uma chamada recursiva a si próprio, seu tempo de execução frequentemente pode ser descrito por uma *equação de recorrência* ou *recorrência*, que descreve o tempo de execução global sobre um problema de tamanho  $n$  em termos do tempo de execução sobre entradas menores. Então, podemos usar ferramentas matemáticas para resolver a recorrência e estabelecer limites sobre o desempenho do algoritmo.

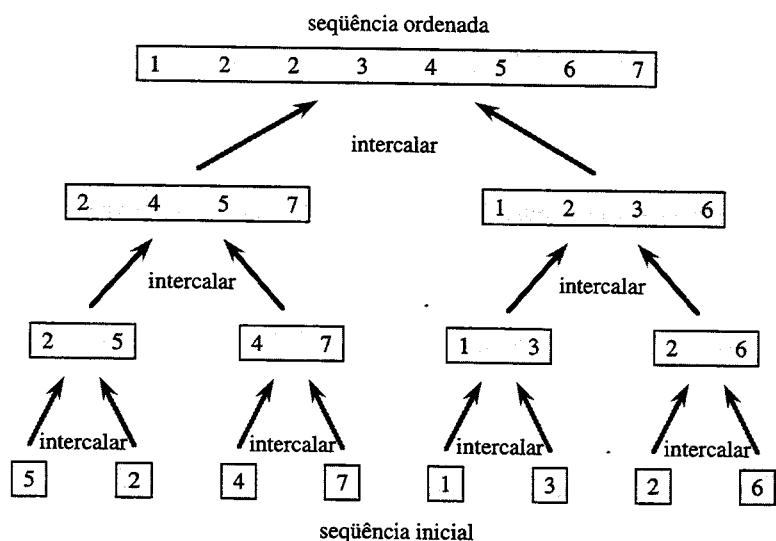


FIGURA 2.4 A operação de ordenação por intercalação sobre o arranjo  $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$ . Os comprimentos das sequências ordenadas que estão sendo intercaladas aumentam com a progressão do algoritmo da parte inferior até a parte superior

Uma recorrência para o tempo de execução de um algoritmo de dividir e conquistar se baseia nos três passos do paradigma básico. Como antes, consideramos  $T(n)$  o tempo de execução sobre um problema de tamanho  $n$ . Se o tamanho do problema for pequeno o bastante, digamos  $n \leq c$  para alguma constante  $c$ , a solução direta demorará um tempo constante, que consideraremos  $\Theta(1)$ . Vamos supor que o problema seja dividido em  $a$  subproblemas, cada um dos quais com  $1/b$  do tamanho do problema original. (No caso da ordenação por intercalação, tanto  $a$  quanto  $b$  são iguais a 2, mas veremos muitos algoritmos de dividir e conquistar nos quais  $a \dots b$ .) Se levarmos o tempo  $D(n)$  para dividir o problema em subproblemas e o tempo  $C(n)$  para combinar as soluções dadas aos subproblemas na solução para o problema original, obteremos a recorrência

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

No Capítulo 4, veremos como resolver recorrências comuns dessa maneira.

## Análise da ordenação por intercalação

Embora o pseudocódigo para MERGE-SORT funcione corretamente quando o número de elementos não é par, nossa análise baseada na recorrência será simplificada se fizermos a suposição de que o tamanho do problema original é uma potência de dois. Então, cada passo de dividir produzirá duas subsequências de tamanho exatamente  $n/2$ . No Capítulo 4, veremos que essa premissa não afeta a ordem de crescimento da solução para a recorrência.

Apresentaremos em seguida nossas razões para configurar a recorrência correspondente a  $T(n)$ , o tempo de execução do pior caso da ordenação por intercalação sobre  $n$  números. A ordenação por intercalação sobre um único elemento demora um tempo constante. Quando temos  $n > 1$  elementos, desmembramos o tempo de execução do modo explicado a seguir.

**Dividir:** A etapa de dividir simplesmente calcula o ponto médio do subarranjo, o que demora um tempo constante. Portanto,  $D(n) = \Theta(1)$ .

**Conquistar:** Resolvemos recursivamente dois subproblemas; cada um tem o tamanho  $n/2$  e contribui com  $2T(n/2)$  para o tempo de execução.

**Combinar:** Já observamos que o procedimento MERGE em um subarranjo de  $n$  elementos leva o tempo  $\Theta(n)$ ; assim,  $C(n) = \Theta(n)$ .

Quando somamos as funções  $D(n)$  e  $C(n)$  para a análise da ordenação por intercalação, estamos somando uma função que é  $\Theta(n)$  a uma função que é  $\Theta(1)$ . Essa soma é uma função linear de  $n$ , ou seja,  $\Theta(n)$ . A adição dessa função ao termo  $2T(n/2)$  da etapa de “conquistar” fornece a recorrência para o tempo de execução do pior caso  $T(n)$  da ordenação por intercalação:

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

No Capítulo 4, mostraremos o “teorema mestre”, que podemos utilizar para demonstrar que  $T(n)$  é  $\Theta(n \lg n)$ , onde  $\lg n$  significa  $\log_2 n$ . Para entradas suficientemente grandes, a ordenação por intercalação, com seu tempo de execução  $\Theta(n \lg n)$ , supera a ordenação por inserção, cujo tempo de execução é  $\Theta(n^2)$ , no pior caso.

Não precisamos do teorema mestre para entender intuitivamente por que a solução para a recorrência (2.1) é  $T(n) = \Theta(n \lg n)$ . Vamos reescrever a recorrência (2.1) como

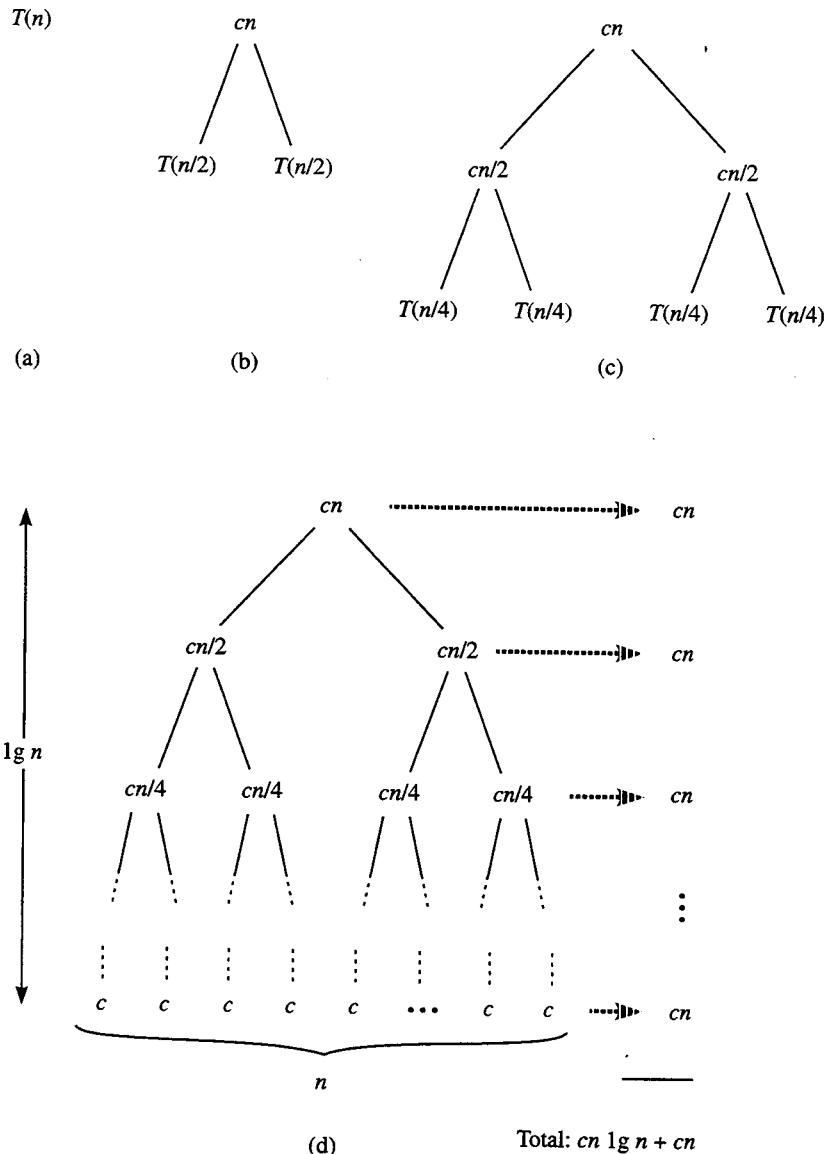
$$T(n) = \begin{cases} c & \text{se } n = 1, \\ 2T(n/2) + cn & \text{se } n > 1. \end{cases} \quad (2.2)$$

onde a constante  $c$  representa o tempo exigido para resolver problemas de tamanho 1, como também o tempo por elemento do arranjo para as etapas de dividir e combinar.<sup>8</sup>

A Figura 2.5 mostra como podemos resolver a recorrência (2.2). Por conveniência, supomos que  $n$  é uma potência exata de 2. A parte (a) da figura mostra  $T(n)$  que, na parte (b), foi expandida em uma árvore equivalente representando a recorrência. O termo  $cn$  é a raiz (o custo no nível superior da recursão), e as duas subárvore da raiz são as duas recorrências menores  $T(n/2)$ . A parte (c) mostra esse processo levado uma etapa adiante pela expansão de  $T(n/2)$ . O custo para cada um dos dois subnós no segundo nível de recursão é  $cn/2$ . Continuamos a expandir cada nó na árvore, desmembrando-o em suas partes constituintes como determina a recorrência, até os tamanhos de problemas se reduzirem a 1, cada qual com o custo  $c$ . A parte (d) mostra a árvore resultante.

---

<sup>8</sup> É improvável que a mesma constante represente exatamente o tempo para resolver problemas de tamanho 1 e também o tempo por elemento do arranjo para as etapas de dividir e combinar. Podemos contornar esse problema permitindo que  $c$  seja o maior desses tempos e reconhecendo que nossa recorrência fornece um limite superior sobre o tempo de execução, ou permitindo a  $c$  ser o menor desses tempos e reconhecendo que nossa recorrência fornece um limite inferior sobre o tempo de execução. Ambos os limites serão estabelecidos sobre a ordem de  $n \lg n$  e, tomados juntos, corresponderão ao tempo de execução  $\Theta(n \lg n)$ .



**FIGURA 2.5** A construção de uma árvore de recursão para a recorrência  $T(n) = 2T(n/2) + cn$ . A parte (a) mostra  $T(n)$ , que é progressivamente expandido em (b)–(d) para formar a árvore de recursão. A árvore completamente expandida da parte (d) tem  $\lg n + 1$  níveis (isto é, tem altura  $\lg n$ , como indicamos), e cada nível contribui com o custo total  $cn$ . Então, o custo total é  $cn \lg n + cn$ , que é  $\Theta(n \lg n)$ .

Em seguida, somamos os custos através de cada nível da árvore. O nível superior tem o custo total  $cn$ , o próximo nível abaixo tem custo total  $c(n/2) + c(n/2) = cn$ , o nível após esse tem custo total  $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$  e assim por diante. Em geral, o nível  $i$  abaixo do topo tem  $2^i$  nós, cada qual contribuindo com um custo  $c(n/2^i)$ , de forma que o  $i$ -ésimo nível abaixo do topo tem custo total  $2^i c(n/2^i) = cn$ .

No nível inferior existem  $n$  nós, cada um contribuindo com um custo  $c_i$  para um custo total  $c_m$ .

O número total de níveis da “árvore de recursão” da Figura 2.5 é  $\lg n + 1$ . Esse fato é facilmente visto por um argumento induutivo informal. O caso básico ocorre quando  $n = 1$ , e nesse caso só há um nível. Como  $\lg 1 = 0$ , temos que  $\lg n + 1$  fornece o número correto de níveis.

Agora suponha, como uma hipótese indutiva, que o número de níveis de uma árvore de recursão para  $2^i$  nós seja  $\lg 2^i + 1 = i + 1$  (pois, para qualquer valor de  $i$ , temos  $\lg 2^i = i$ ). Como estamos supondo que o tamanho da entrada original é uma potência de 2, o tamanho da próxima entrada a considerar é  $2^{i+1}$ . Uma árvore com  $2^{i+1}$  nós tem um nível a mais em relação a uma árvore de  $2^i$  nós, e então o número total de níveis é  $(i + 1) + 1 = \lg 2^{i+1} + 1$ .

Para calcular o custo total representado pela recorrência (2.2), simplesmente somamos os custos de todos os níveis. Há  $\lg n + 1$  níveis, cada um com o custo  $cn$ , o que nos dá o custo total  $cn(\lg n + 1) = cn \lg n + cn$ . Ignorando o termo de baixa ordem e a constante  $c$ , obtemos o resultado desejado,  $\Theta(n \lg n)$ .

## Exercícios

### 2.3-1

Usando a Figura 2.4 como modelo, ilustre a operação de ordenação por intercalação sobre o arranjo  $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$ .

### 2.3-2

Reescreva o procedimento MERGE de modo que ele não utilize sentinelas e, em vez disso, se interrompa depois que o arranjo  $L$  ou  $R$  tiver todos os seus elementos copiados de volta para  $A$ , e então copie o restante do outro arranjo de volta em  $A$ .

### 2.3-3

Use indução matemática para mostrar que, quando  $n$  é uma potência exata de 2, a solução da recorrência

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

é  $T(n) = n \lg n$ .

### 2.3-4

A ordenação por inserção pode ser expressa sob a forma de um procedimento recursivo como a seguir. Para ordenar  $A[1 .. n]$ , ordenamos recursivamente  $A[1 .. n - 1]$  e depois inserimos  $A[n]$  no arranjo ordenado  $A[1 .. n - 1]$ . Escreva uma recorrência para o tempo de execução dessa versão recursiva da ordenação por inserção.

### 2.3-5

Voltando ao problema da pesquisa (ver Exercício 2.1-3) observe que, se a seqüência  $A$  estiver ordenada, poderemos comparar o ponto médio da seqüência com  $v$  e eliminar metade da seqüência de consideração posterior. A **pesquisa binária** é um algoritmo que repete esse procedimento, dividindo ao meio o tamanho da porção restante da seqüência a cada vez. Escreva pseudocódigo, sendo ele iterativo ou recursivo, para pesquisa binária. Demonstre que o tempo de execução do pior caso da pesquisa binária é  $\Theta(\lg n)$ .

### 2.3-6

Observe que o loop **while** das linhas 5 a 7 do procedimento INSERTION-SORT na Seção 2.1 utiliza uma pesquisa linear para varrer (no sentido inverso) o subarranjo ordenado  $A[1 .. j - 1]$ . Podemos usar em vez disso uma pesquisa binária (ver Exercício 2.3-5) para melhorar o tempo de execução global do pior caso da ordenação por inserção para  $\Theta(n \lg n)$ ?

### 2.3-7 \*

Descreva um algoritmo de tempo  $\Theta(n \lg n)$  que, dado um conjunto  $S$  de  $n$  inteiros e outro inteiro  $x$ , determine se existem ou não dois elementos em  $S$  cuja soma seja exatamente  $x$ .

## Problemas

### 2-1 Ordenação por inserção sobre arranjos pequenos na ordenação por intercalação

Embora a ordenação por intercalação funcione no tempo de pior caso  $\Theta(n \lg n)$  e a ordenação por inserção funcione no tempo de pior caso  $\Theta(n^2)$ , os fatores constantes na ordenação por inserção a tornam mais rápida para  $n$  pequeno. Assim, faz sentido usar a ordenação por inserção dentro da ordenação por intercalação quando os subproblemas se tornam suficientemente pequenos. Considere uma modificação na ordenação por intercalação, na qual  $n/k$  sublistas de comprimento  $k$  são ordenadas usando-se a ordenação por inserção, e depois intercaladas com o uso do mecanismo padrão de intercalação, onde  $k$  é um valor a ser determinado.

- a. Mostre que as  $n/k$  sublistas, cada uma de comprimento  $k$ , podem ser ordenadas através da ordenação por inserção no tempo de pior caso  $\Theta(nk)$ .
- b. Mostre que as sublistas podem ser intercaladas no tempo de pior caso  $\Theta(n \lg(n/k))$ .
- c. Dado que o algoritmo modificado é executado no tempo de pior caso  $\Theta(nk + n \lg(n/k))$ , qual é o maior valor assintótico (notação  $\Theta$ ) de  $k$  como uma função de  $n$  para a qual o algoritmo modificado tem o mesmo tempo de execução assintótico que a ordenação por intercalação padrão?
- d. Como  $k$  deve ser escolhido na prática?

### 2-2 Correção do bubblesort

O bubblesort é um algoritmo de ordenação popular. Ele funciona permutando repetidamente elementos adjacentes que estão fora de ordem.

BUBBLESORT( $A$ )

```
1 for  $i \leftarrow 1$  to comprimento[ $A$ ]  
2   do for  $j \leftarrow \text{comprimento}[A]$  downto  $i + 1$   
3     do if  $A[j] < A[j - 1]$   
4       then trocar  $A[j] \leftrightarrow A[j - 1]$ 
```

- a. Seja  $A'$  um valor que denota a saída de BUBBLESORT( $A$ ). Para provar que BUBBLESORT é correto, precisamos provar que ele termina e que

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

onde  $n = \text{comprimento}[A]$ . O que mais deve ser provado para mostrar que BUBBLESORT realmente realiza a ordenação?

As duas próximas partes provarão a desigualdade (2.3).

- b. Enuncie com precisão um loop invariante para o loop **for** das linhas 2 a 4 e prove que esse loop invariante é válido. Sua prova deve usar a estrutura da prova do loop invariante apresentada neste capítulo.
- c. Usando a condição de término do loop invariante demonstrado na parte (b), enuncie um loop invariante para o loop **for** das linhas 1 a 4 que lhe permita provar a desigualdade (2.3). Sua prova deve empregar a estrutura da prova do loop invariante apresentada neste capítulo.
- d. Qual é o tempo de execução do pior caso de bubblesort? Como ele se compara ao tempo de execução da ordenação por inserção?

### 2-3 Correção da regra de Horner

O fragmento de código a seguir implementa a regra de Horner para avaliar um polinômio

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

dados os coeficientes  $a_0, a_1, \dots, a_n$  e um valor para  $x$ :

```
1  $y \leftarrow 0$ 
2  $i \leftarrow n$ 
3 while  $i \geq 0$ 
4   do  $y \leftarrow a_i + x \cdot y$ 
5    $i \leftarrow i - 1$ 
```

- a. Qual é o tempo de execução assintótico desse fragmento de código para a regra de Horner?
- b. Escreva pseudocódigo para implementar o algoritmo ingênuo de avaliação polinomial que calcula cada termo do polinômio desde o início. Qual é o tempo de execução desse algoritmo? Como ele se compara à regra de Horner?
- c. Prove que a expressão a seguir é um loop invariante para o loop **while** das linhas 3 a 5.

No início de cada iteração do loop **while** das linhas 3 a 5,

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

Interprete um somatório sem termos como igual a 0. Sua prova deve seguir a estrutura da prova do loop invariante apresentada neste capítulo e deve mostrar que, no término,  $y = \sum_{k=0}^n a_k x^k$ .

- d. Conclua demonstrando que o fragmento de código dado avalia corretamente um polinômio caracterizado pelos coeficientes  $a_0, a_1, \dots, a_n$ .

### 2-4 Inversões

Seja  $A[1..n]$  um arranjo de  $n$  números distintos. Se  $i < j$  e  $A[i] > A[j]$ , então o par  $(i, j)$  é chamado uma **inversão** de  $A$ .

- a. Liste as cinco inversões do arranjo  $\langle 2, 3, 8, 6, 1 \rangle$ .
- b. Qual arranjo com elementos do conjunto  $\{1, 2, \dots, n\}$  tem o número máximo de inversões? Quantas inversões ele tem?
- c. Qual é o relacionamento entre o tempo de execução da ordenação por inserção e o número de inversões no arranjo de entrada? Justifique sua resposta.
- d. Dê um algoritmo que determine o número de inversões em qualquer permutação sobre  $n$  elementos no tempo do pior caso de  $\Theta(n \lg n)$ . (Sugestão: modifique a ordenação por intercalação.)

## Notas do capítulo

Em 1968, Knuth publicou o primeiro de três volumes com o título geral *The Art of Computer Programming* [182, 183, 185]. O primeiro volume iniciou o estudo moderno de algoritmos de computador com um foco na análise do tempo de execução, e a série inteira continua a ser uma referência valiosa e interessante para muitos dos tópicos apresentados aqui. De acordo com Knuth, a palavra “algoritmo” é derivada do nome “al-Khowârizmî”, um matemático persa do século IX.

Aho, Hopcroft e Ullman [5] defenderam a análise assintótica dos algoritmos como um meio de comparar o desempenho relativo. Eles também popularizaram o uso de relações de recorrência para descrever os tempos de execução de algoritmos recursivos.

Knuth [185] oferece um tratamento enciclopédico de muitos algoritmos de ordenação. Sua comparação dos algoritmos de ordenação (página 381) inclui análises exatas de contagem de passos, como a que realizamos aqui para a ordenação por inserção. A discussão por Knuth da ordenação por inserção engloba diversas variações do algoritmo. A mais importante delas é a ordenação de Shell, introduzida por D. L. Shell, que utiliza a ordenação por inserção sobre subsequências periódicas da entrada para produzir um algoritmo de ordenação mais rápido.

A ordenação por intercalação também é descrita por Knuth. Ele menciona que um intercalador mecânico capaz de mesclar dois decks de cartões perfurados em uma única passagem foi inventado em 1938. J. von Neumann, um dos pioneiros da informática, aparentemente escreveu um programa para fazer a ordenação por intercalação no computador EDVAC em 1945.

A primeira referência à demonstração de programas corretos é descrita por Gries [133], que credita a P. Naur o primeiro artigo nesse campo. Gries atribui loops invariantes a R. W. Floyd. O livro-texto de Mitchell [222] descreve o progresso mais recente na demonstração de programas corretos.

# Crescimento de funções

A ordem de crescimento do tempo de execução de um algoritmo, definida no Capítulo 2, fornece uma caracterização simples da eficiência do algoritmo, e também nos permite comparar o desempenho relativo de algoritmos alternativos. Uma vez que o tamanho da entrada  $n$  se torna grande o suficiente, a ordenação por intercalação, com seu tempo de execução do pior caso  $\Theta(n \lg n)$ , vence a ordenação por inserção, cujo tempo de execução do pior caso é  $\Theta(n^2)$ . Embora às vezes seja possível determinar o tempo exato de execução de um algoritmo, como fizemos no caso da ordenação por inserção no Capítulo 2, a precisão extra em geral não vale o esforço de calculá-la. Para entradas grandes o bastante, as constantes multiplicativas e os termos de mais baixa ordem de um tempo de execução exato são dominados pelos efeitos do próprio tamanho da entrada.

Quando observamos tamanhos de entrada grandes o suficiente para tornar relevante apenas a ordem de crescimento do tempo de execução, estamos estudando a eficiência **assintótica** dos algoritmos. Ou seja, estamos preocupados com a maneira como o tempo de execução de um algoritmo aumenta com o tamanho da entrada *no limite*, à medida que o tamanho da entrada aumenta indefinidamente (sem limitação). Em geral, um algoritmo que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, exceto as muito pequenas.

Este capítulo oferece vários métodos padrão para simplificar a análise assintótica de algoritmos. A próxima seção começa definindo diversos tipos de “notação assintótica”, da qual já vimos um exemplo na notação  $\Theta$ . Várias convenções de notação usadas em todo este livro serão então apresentadas, e finalmente faremos uma revisão do comportamento de funções que surgem comumente na análise de algoritmos.

### 3.1 Notação assintótica

As notações que usamos para descrever o tempo de execução assintótica de um algoritmo são definidas em termos de funções cujos domínios são o conjunto dos números naturais  $N = \{0, 1, 2, \dots\}$ . Tais notações são convenientes para descrever a função do tempo de execução do pior caso  $T(n)$ , que em geral é definida somente sobre tamanhos de entrada inteiros. Contudo, às vezes é conveniente *abusar* da notação assintótica de várias maneiras. Por exemplo, a notação é estendida com facilidade ao domínio dos números reais ou, de modo alternativo, limitado a um subconjunto dos números naturais. Porém, é importante entender o significado preciso da notação para que, quando houver um abuso em seu uso, ela não seja *mal utilizada*. Esta seção define as notações assintóticas básicas e também apresenta alguns abusos comuns.

## Notação $\Theta$

No Capítulo 2, descobrimos que o tempo de execução do pior caso da ordenação por inserção é  $T(n) = \Theta(n^2)$ . Vamos definir o que significa essa notação. Para uma dada função  $g(n)$ , denotamos por  $\Theta(g(n))$  o *conjunto de funções*

$$\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\}.$$

<sup>1</sup>

Uma função  $f(n)$  pertence ao conjunto  $\Theta(g(n))$  se existem constantes positivas  $c_1$  e  $c_2$  tais que ela possa ser “imprensada” entre  $c_1 g(n)$  e  $c_2 g(n)$ , para um valor de  $n$  suficientemente grande. Como  $\Theta(g(n))$  é um conjunto, poderíamos escrever “ $f(n) \in \Theta(g(n))$ ” para indicar que  $f(n)$  é um membro de (ou pertence a)  $\Theta(g(n))$ . Em vez disso, em geral escreveremos “ $f(n) = \Theta(g(n))$ ” para expressar a mesma noção. Esse abuso da igualdade para denotar a condição de membro de um conjunto (pertinência) pode a princípio parecer confuso, mas veremos adiante nesta seção que ele tem suas vantagens.

A Figura 3.1(a) apresenta um quadro intuitivo das funções  $f(n)$  e  $g(n)$ , onde  $f(n) = \Theta(g(n))$ . Para todos os valores de  $n$  à direita de  $n_0$ , o valor de  $f(n)$  reside em  $c_1 g(n)$  ou acima dele, e em  $c_2 g(n)$  ou abaixo desse valor. Em outras palavras, para todo  $n \geq n_0$ , a função  $f(n)$  é igual a  $g(n)$  dentro de um fator constante. Dizemos que  $g(n)$  é um *limite assintoticamente restrito* para  $f(n)$ .

A definição de  $\Theta(g(n))$  exige que todo membro  $f(n) \in \Theta(g(n))$  seja *assintoticamente não negativo*, isto é, que  $f(n)$  seja não negativo sempre que  $n$  for suficientemente grande. (Uma função *assintoticamente positiva* é uma função positiva para todo  $n$  suficientemente grande.) Em consequência disso, a própria função  $g(n)$  deve ser assintoticamente não negativa, ou então o conjunto  $\Theta(g(n))$  é vazio. Por essa razão, vamos supor que toda função usada dentro da notação  $\Theta$  é assintoticamente não negativa. Essa premissa também se mantém para as outras notações assintóticas definidas neste capítulo.

No Capítulo 2, introduzimos uma noção informal da notação  $\Theta$  que consistia em descartar os termos de mais baixa ordem e ignorar o coeficiente inicial do termo de mais alta ordem. Vamos justificar brevemente essa intuição, usando a definição formal para mostrar que  $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ . Para isso, devemos definir constantes positivas  $c_1, c_2$  e  $n_0$  tais que

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

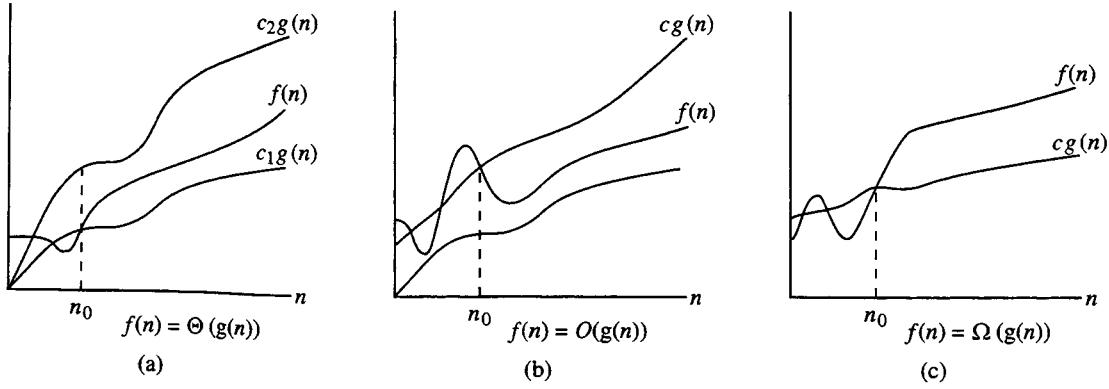
para todo  $n \geq n_0$ . A divisão por  $n^2$  produz

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

A desigualdade do lado direito pode ser considerada válida para qualquer valor de  $n \geq 1$ , escolhendo-se  $c_2 \geq 1/2$ . Do mesmo modo, a desigualdade da esquerda pode ser considerada válida para qualquer valor de  $n \geq 7$ , escolhendo-se  $c_1 \leq 1/14$ . Assim, escolhendo  $c_1 = 1/14$ ,  $c_2 = 1/2$  e  $n_0 = 7$ , podemos verificar que  $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ . Certamente, existem outras opções para as constantes, mas o fato importante é que existe *alguma* opção. Observe que essas constantes dependem da função  $\frac{1}{2}n^2 - 3n$ ; uma função diferente pertencente a  $\Theta(n^2)$  normalmente exigiria constantes distintas.

---

<sup>1</sup> Na notação de conjuntos, um sinal de dois-pontos deve ser lido como “tal que”.



**FIGURA 3.1** Exemplos gráficos das notações  $\Theta$ ,  $O$  e  $\Omega$ . Em cada parte, o valor de  $n_0$  mostrado é o valor mínimo possível; qualquer valor maior também funcionaria. (a) A notação  $\Theta$  limita uma função ao intervalo entre fatores constantes. Escrevemos  $f(n) = \Theta(g(n))$  se existem constantes positivas  $n_0$ ,  $c_1$  e  $c_2$  tais que, à direita de  $n_0$ , o valor de  $f(n)$  sempre reside entre  $c_1g(n)$  e  $c_2g(n)$  inclusive. (b) A notação  $O$  dá um limite superior para uma função dentro de um fator constante. Escrevemos  $f(n) = O(g(n))$  se existem constantes positivas  $n_0$  e  $c$  tais que, à direita de  $n_0$ , o valor de  $f(n)$  sempre reside em ou abaixo de  $cg(n)$ . (c) A notação  $\Omega$  dá um limite inferior para uma função dentro de um fator constante. Escrevemos  $f(n) = \Omega(g(n))$  se existem constantes positivas  $n_0$  e  $c$  tais que, à direita de  $n_0$ , o valor de  $f(n)$  sempre reside em ou acima de  $cg(n)$ .

Também podemos usar a definição formal para verificar que  $6n^3 \neq \Theta(n^2)$ . Vamos supor, a título de contradição, que existam  $c_2$  e  $n_0$  tais que  $6n^3 \leq c_2n^2$  para todo  $n \geq n_0$ . Mas então  $n \leq c_2/6$ , o que não pode ser válido para um valor de  $n$  arbitrariamente grande, pois  $c_2$  é constante.

Intuitivamente, os termos de mais baixa ordem de uma função assintoticamente positiva podem ser ignorados na determinação de limites assintoticamente restritos, porque eles são insignificantes para grandes valores de  $n$ . Uma minúscula fração do termo de mais alta ordem é suficiente para dominar os termos de mais baixa ordem. Desse modo, a definição de  $c_1$  com um valor ligeiramente menor que o coeficiente do termo de mais alta ordem e a definição de  $c_2$  com um valor ligeiramente maior permite que as desigualdades na definição da notação  $\Theta$  sejam satisfeitas. O coeficiente do termo de mais alta ordem pode do mesmo modo ser ignorado, pois ele só muda  $c_1$  e  $c_2$  por um fator constante igual ao coeficiente.

Como exemplo, considere qualquer função quadrática  $f(n) = an^2 + bn + c$ , onde  $a, b$  e  $c$  são constantes e  $a > 0$ . Descartando os termos de mais baixa ordem e ignorando a constante, produzimos  $f(n) = \Theta(n^2)$ . Formalmente, para mostrar a mesma coisa, tomamos as constantes  $c_1 = a/4$ ,  $c_2 = 7a/4$  e  $n_0 = 2 \cdot \max(|b|/a, \sqrt{|c|/a})$ . O leitor poderá verificar que  $0 \leq c_1n^2 \leq an^2 + bn + c \leq c_2n^2$  para todo  $n \geq n_0$ . Em geral, para qualquer polinômio  $p(n) = \sum_{i=0}^d a_i n^i$ , onde  $a_i$  são constantes e  $a_d > 0$ , temos  $p(n) = \Theta(n^d)$  (ver Problema 3-1).

Tendo em vista que qualquer constante é um polinômio de grau 0, podemos expressar qualquer função constante como  $\Theta(n^0)$ , ou  $\Theta(1)$ . Porém, essa última notação é um abuso secundário, porque não está claro qual variável está tendendo a infinito.<sup>2</sup> Usaremos com frequência a notação  $\Theta(1)$  para indicar uma constante ou uma função constante em relação a alguma variável.

<sup>2</sup> O problema real é que nossa notação comum para funções não distingue funções de valores. No cálculo de  $\lambda$ , os parâmetros para uma função estão claramente especificados: a função  $n^2$  poderia ser escrita como  $\lambda n \cdot n^2$ , ou até mesmo  $\lambda r \cdot r^2$ . Porém, a adoção de uma notação mais rigorosa complicaria manipulações algébricas, e assim optamos por tolerar o abuso.

## Notação O

A notação  $\Theta$  limita assintoticamente uma função acima e abaixo. Quando temos apenas um *limite assintótico superior*, usamos a notação  $O$ . Para uma dada função  $g(n)$ , denotamos por  $O(g(n))$  (lê-se “O maiúsculo de  $g$  de  $n$ ” ou, às vezes, apenas “o de  $g$  de  $n$ ”) o conjunto de funções

$$O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\}.$$

Usamos a notação  $O$  para dar um limite superior sobre uma função, dentro de um fator constante. A Figura 3.1(b) mostra a intuição por trás da notação  $O$ . Para todos os valores  $n$  à direita de  $n_0$ , o valor da função  $f(n)$  está em ou abaixo de  $g(n)$ .

Para indicar que uma função  $f(n)$  é um membro de  $O(g(n))$ , escrevemos  $f(n) = O(g(n))$ . Observe que  $f(n) = \Theta(g(n))$  implica  $f(n) = O(g(n))$ , pois a notação  $\Theta$  é uma noção mais forte que a notação  $O$ . Em termos da teoria de conjuntos, temos  $\Theta(g(n)) \subseteq O(g(n))$ . Desse modo, nossa prova de que qualquer função quadrática  $an^2 + bn + c$ , onde  $a > 0$ , está em  $\Theta(n^2)$  também mostra que qualquer função quadrática está em  $O(n^2)$ . O que pode ser mais surpreendente é o fato de que qualquer função linear  $an + b$  está em  $O(n^2)$ , o que é facilmente verificado fazendo-se  $c = a + |b|$  e  $n_0 = 1$ .

Alguns leitores que viram antes a notação  $O$  podem achar estranho que devamos escrever, por exemplo,  $n = O(n^2)$ . Na literatura, a notação  $O$  é usada às vezes de modo informal para descrever limites assintoticamente restritos, ou seja, o que definimos usando a notação  $\Theta$ . Contudo, neste livro, quando escrevermos  $f(n) = O(g(n))$ , estamos simplesmente afirmado que algum múltiplo constante de  $g(n)$  é um limite assintótico superior sobre  $f(n)$ , sem qualquer menção sobre o quanto um limite superior é restrito. A distinção entre limites assintóticos superiores e limites assintoticamente restritos agora se tornou padrão na literatura de algoritmos.

Usando a notação  $O$ , podemos descrever freqüentemente o tempo de execução de um algoritmo apenas inspecionando a estrutura global do algoritmo. Por exemplo, a estrutura de loop duplamente aninhado do algoritmo de ordenação por inserção vista no Capítulo 2 produz imediatamente um limite superior  $O(n^2)$  sobre o tempo de execução do pior caso: o custo do loop interno é limitado na parte superior por  $O(1)$  (constante), os índices  $i$  e  $j$  são ambos no máximo  $n$ , e o loop interno é executado no máximo uma vez para cada um dos  $n^2$  pares de valores correspondentes a  $i$  e  $j$ .

Tendo em vista que a notação  $O$  descreve um limite superior, quando a empregamos para limitar o tempo de execução do pior caso de um algoritmo, temos um limite sobre o tempo de execução do algoritmo em cada entrada. Desse modo, o limite  $O(n^2)$  no tempo de execução do pior caso da ordenação por inserção também se aplica a seu tempo de execução sobre toda entrada. Porém, o limite  $\Theta(n^2)$  no tempo de execução do pior caso da ordenação por inserção não implica um limite  $\Theta(n^2)$  no tempo de execução da ordenação por inserção em *toda* entrada. Por exemplo, vimos no Capítulo 2 que, quando a entrada já está ordenada, a ordenação por inserção funciona no tempo  $\Theta(n)$ .

Tecnicamente, é um abuso dizer que o tempo de execução da ordenação por inserção é  $O(n^2)$ , pois, para um dado  $n$ , o tempo de execução real varia, dependendo da entrada específica de tamanho  $n$ . Quando afirmamos que “o tempo de execução é  $O(n^2)$ ”, queremos dizer que existe uma função  $f(n)$  que é  $O(n^2)$  tal que, para qualquer valor de  $n$ , não importando que entrada específica de tamanho  $n$  seja escolhida, o tempo de execução sobre essa entrada tem um limite superior determinado pelo valor  $f(n)$ . De modo equivalente, dizemos que o tempo de execução do pior caso é  $O(n^2)$ .

## Notação $\Omega$

Da mesma maneira que a notação  $O$  fornece um limite assintótico *superior* sobre uma função, a notação  $\Omega$  fornece um *limite assintótico inferior*. Para uma determinada função  $g(n)$ , denotamos por  $\Omega(g(n))$  (lê-se “ômega maiúsculo de  $g$  de  $n$ ” ou, às vezes, “ômega de  $g$  de  $n$ ”) o conjunto de funções

$$\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que} \\ 0 \leq cg(n) \leq f(n) \text{ para todo } n \geq n_0\}.$$

A intuição por trás da notação  $\Omega$  é mostrada na Figura 3.1(c). Para todos os valores  $n$  à direita de  $n_0$ , o valor de  $f(n)$  está em ou acima de  $g(n)$ .

A partir das definições das notações assintóticas que vimos até agora, é fácil demonstrar o importante teorema a seguir (ver Exercício 3.1-5).

### Teorema 3.1

Para duas funções quaisquer  $f(n)$  e  $g(n)$ , temos  $f(n) = \Theta(g(n))$  se e somente se  $f(n) = O(g(n))$  e  $f(n) = \Omega(g(n))$ . ■

Como exemplo de aplicação desse teorema, nossa demonstração de que  $an^2 + bn + c = \Theta(n^2)$  para quaisquer constantes  $a, b$  e  $c$ , onde  $a > 0$ , implica imediatamente que  $an^2 + bn + c = \Omega(n^2)$  e  $an^2 + bn + c = O(n^2)$ . Na prática, em lugar de usar o Teorema 3.1 para obter limites assintóticos superiores e inferiores a partir de limites assintoticamente restritos, como fizemos nesse exemplo, nós o utilizamos normalmente para demonstrar limites assintoticamente restritos a partir de limites assintóticos superiores e inferiores.

Considerando-se que a notação  $\Omega$  descreve um limite inferior, quando a usamos para limitar o tempo de execução do melhor caso de um algoritmo, por implicação também limitamos o tempo de execução do algoritmo sobre entradas arbitrárias. Por exemplo, o tempo de execução no melhor caso da ordenação por inserção é  $\Omega(n)$ , o que implica que o tempo de execução da ordenação por inserção é  $\Omega(n)$ .

Assim, o tempo de execução da ordenação por inserção recai entre  $\Omega(n)$  e  $O(n^2)$ , pois ele fica em qualquer lugar entre uma função linear de  $n$  e uma função quadrática de  $n$ . Além disso, esses limites são assintoticamente tão restritos quanto possível: por exemplo, o tempo de execução da ordenação por inserção não é  $\Omega(n^2)$ , pois existe uma entrada para a qual a ordenação por inserção é executada no tempo  $\Theta(n)$  (por exemplo, quando a entrada já está ordenada). Contudo, não é contraditório dizer que o tempo de execução do *pior caso* da ordenação por inserção é  $\Omega(n^2)$ , tendo em vista que existe uma entrada que faz o algoritmo demorar o tempo  $\Omega(n^2)$ . Quando afirmamos que o *tempo de execução* (sem modificador) de um algoritmo é  $\Omega(g(n))$ , queremos dizer que, *independentemente da entrada específica de tamanho n escolhida para cada valor de n*, o tempo de execução sobre essa entrada é pelo menos uma constante vezes  $g(n)$ , para um valor de  $n$  suficientemente grande.

## Notação assintótica em equações e desigualdades

Já vimos como a notação assintótica pode ser usada dentro de fórmulas matemáticas. Por exemplo, na introdução da notação  $O$ , escrevemos “ $n = O(n^2)$ ”. Também poderíamos escrever  $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ . Como interpretaremos tais fórmulas?

Quando a notação assintótica está sozinha no lado direito de uma equação (ou desigualdade), como em  $n = O(n^2)$ , já definimos o sinal de igualdade como símbolo de pertinência a um conjunto:  $n \in O(n^2)$ . Porém, em geral, quando a notação assintótica aparecer em uma fórmula, nós a interpretaremos com o significado de alguma função anônima que não nos preocupare-

mos em nomear. Por exemplo, a fórmula  $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$  significa que  $2n^2 + 3n + 1 = 2n^2 + f(n)$ , onde  $f(n)$  é alguma função no conjunto  $\Theta(n)$ . Nesse caso,  $f(n) = 3n + 1$ , que de fato está em  $\Theta(n)$ .

O uso da notação assintótica dessa maneira pode ajudar a eliminar detalhes não essenciais e a desordem em uma equação. Por exemplo, expressamos no Capítulo 2 o tempo de execução do pior caso da ordenação por intercalação como a recorrência

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

Se estivermos interessados apenas no comportamento assintótico de  $T(n)$ , não haverá sentido em especificar exatamente todos os termos de mais baixa ordem; todos eles serão considerados incluídos na função anônima denotada pelo termo  $\Theta(n)$ .

O número de funções anônimas em uma expressão é entendido como igual ao número de vezes que a notação assintótica aparece. Por exemplo, na expressão

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

existe apenas uma única função anônima (uma função de  $i$ ). Portanto, essa expressão *não* é o mesmo que  $O(1) + O(2) + \dots + O(n)$  que, na realidade, não tem uma interpretação clara.

Em alguns casos, a notação assintótica aparece no lado esquerdo de uma equação, como em

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

Interpretamos tais equações usando a seguinte regra: *Independentemente de como as funções anônimas são escolhidas no lado esquerdo do sinal de igualdade, existe um modo de escolher as funções anônimas no lado direito do sinal de igualdade para tornar a equação válida.* Desse modo, o significado de nosso exemplo é que, para *qualquer* função  $f(n) \in \Theta(n)$ , existe *alguma* função  $g(n) \in \Theta(n^2)$ , tal que  $2n^2 + f(n) = g(n)$  para todo  $n$ . Em outras palavras, o lado direito de uma equação fornece um nível mais grosso de detalhes que o lado esquerdo.

Vários desses relacionamentos podem ser encadeados, como em

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

Podemos interpretar cada equação separadamente pela regra anterior. A primeira equação diz que existe *alguma* função  $f(n) \in \Theta(n)$  tal que  $2n^2 + 3n + 1 = 2n^2 + f(n)$  para todo  $n$ . A segunda equação afirma que, para *qualquer* função  $g(n) \in \Theta(n)$  (como a função  $f(n)$  que acabamos de mencionar), existe *alguma* função  $b(n) \in \Theta(n^2)$  tal que  $2n^2 + g(n) = b(n)$  para todo  $n$ . Observe que essa interpretação implica que  $2n^2 + 3n + 1 = \Theta(n^2)$ , que é aquilo que o encadeamento de equações nos fornece intuitivamente.

## Notação $o$

O limite assintótico superior fornecido pela notação  $O$  pode ser ou não assintoticamente restrito. O limite  $2n^2 = O(n^2)$  é assintoticamente restrito, mas o limite  $2n = O(n^2)$  não o é. Usamos a notação  $o$  para denotar um limite superior que não é assintoticamente restrito. Definimos formalmente  $o(g(n))$  (lê-se “ $o$  minúsculo de  $g$  de  $n$ ”) como o conjunto

$o(g(n)) = \{f(n) : \text{ para qualquer constante positiva } c > 0, \text{ existe uma constante } n_0 > 0 \text{ tal que } 0 \leq f(n) < cg(n) \text{ para todo } n \geq n_0\}$ .

Por exemplo,  $2n = o(n^2)$ , mas  $2n^2 \neq o(n^2)$ .

As definições da notação  $O$  e da notação  $o$  são semelhantes. A principal diferença é que em  $f(n) = O(g(n))$ , o limite  $0 \leq f(n) \leq cg(n)$  se mantém válido para *alguma* constante  $c > 0$  mas, em  $f(n) = o(g(n))$ , o limite  $0 \leq f(n) < cg(n)$  é válido para *todas* as constantes  $c > 0$ . Intuitivamente, na notação  $o$ , a função  $f(n)$  se torna insignificante em relação a  $g(n)$  à medida que  $n$  se aproxima do infinito; isto é,

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

Alguns autores usam esse limite como uma definição da notação  $o$ ; a definição neste livro também restringe as funções anônimas a serem assintoticamente não negativas.

### Notação $\omega$

Por analogia, a notação  $\omega$  está para a notação  $\Omega$  como a notação  $o$  está para a notação  $O$ . Usamos a notação  $\omega$  para denotar um limite inferior que não é assintoticamente restrito. Um modo de defini-la é

$f(n) \in \omega(g(n))$  se e somente se  $g(n) \in o(f(n))$ .

Porém, formalmente, definimos  $\omega(g(n))$  (lê-se “ômega minúsculo de  $g$  de  $n$ ”) como o conjunto

$\omega(g(n)) = \{f(n) : \text{ para qualquer constante positiva } c > 0, \text{ existe uma constante } n_0 > 0 \text{ tal que } 0 \leq cg(n) < f(n) \text{ para todo } n \geq n_0\}.$

Por exemplo,  $n^2/2 = \omega(n)$ , mas  $n^2/2 \neq \omega(n^2)$ . A relação  $f(n) = \omega(g(n))$  implica que

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

se o limite existe. Isto é,  $f(n)$  se torna arbitrariamente grande em relação a  $g(n)$  à medida que  $n$  se aproxima do infinito.

## Comparação de funções

Muitas das propriedades relacionais de números reais também se aplicam a comparações assintóticas. No caso das propriedades seguintes, suponha que  $f(n)$  e  $g(n)$  sejam assintoticamente positivas.

### Transitividade:

$f(n) = \Theta(g(n))$	e	$g(n) = \Theta(b(n))$	implicam	$f(n) = \Theta(b(n))$ ,
$f(n) = O(g(n))$	e	$g(n) = O(b(n))$	implicam	$f(n) = O(b(n))$ ,
$f(n) = \Omega(g(n))$	e	$g(n) = \Omega(b(n))$	implicam	$f(n) = \Omega(b(n))$ ,
$f(n) = o(g(n))$	e	$g(n) = o(b(n))$	implicam	$f(n) = o(b(n))$ ,
$f(n) = \omega(g(n))$	e	$g(n) = \omega(b(n))$	implicam	$f(n) = \omega(b(n))$ .

### Reflexividade:

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

### Simetria:

$$f(n) = \Theta(g(n)) \text{ se e somente se } g(n) = \Theta(f(n))$$

### Simetria de transposição:

$$\begin{aligned}f(n) &= O(g(n)) \text{ se e somente se } g(n) = \Omega(f(n)) \\f(n) &= o(g(n)) \text{ se e somente se } g(n) = \omega(f(n))\end{aligned}$$

Pelo fato dessas propriedades se manterem válidas para notações assintóticas, é possível traçar uma analogia entre a comparação assintótica de duas funções  $f$  e  $g$  e a comparação de dois números reais  $a$  e  $b$ :

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

Dizemos que  $f(n)$  é **assintoticamente menor** que  $g(n)$  se  $f(n) = o(g(n))$ , e que  $f(n)$  é **assintoticamente maior** que  $g(n)$  se  $f(n) = \omega(g(n))$ .

Contudo, uma propriedade de números reais não é transportada para a notação assintótica:

**Tricotomia:** Para dois números reais quaisquer  $a$  e  $b$ , exatamente uma das propriedades a seguir deve ser válida:  $a < b$ ,  $a = b$  ou  $a > b$ .

Embora dois números reais quaisquer possam ser comparados, nem todas as funções são assintoticamente comparáveis. Ou seja, para duas funções  $f(n)$  e  $g(n)$ , pode acontecer que nem  $f(n) = O(g(n))$ , nem  $f(n) = \Omega(g(n))$  seja válida. Por exemplo, as funções  $n$  e  $n^{1 + \sin n}$  não podem ser comparadas utilizando-se a notação assintótica, pois o valor do expoente em  $n^{1 + \sin n}$  oscila entre 0 e 2, assumindo todos os valores intermediários.

## Exercícios

### 3.1-1

Sejam  $f(n)$  e  $g(n)$  funções assintoticamente não negativas. Usando a definição básica da notação  $\Theta$ , prove que  $\max(f(n), g(n)) = \Theta(f(n) + g(n))$ .

### 3.1-2

Mostre que, para quaisquer constantes reais  $a$  e  $b$ , onde  $b > 0$ ,

$$(n + a)^b = \Theta(n^b). \tag{3.2}$$

### 3.1-3

Explique por que a declaração “O tempo de execução no algoritmo A é no mínimo  $O(n^2)$ ” é isenta de significado.

### 3.1-4

É verdade que  $2^{n+1} = O(2^n)$ ? É verdade que  $2^{2n} = O(2^n)$ ?

### 3.1-5

Demonstre o Teorema 3.1.

### 3.1-6

Prove que o tempo de execução de um algoritmo é  $\Theta(g(n))$  se e somente se seu tempo de execução do pior caso é  $O(g(n))$  e seu tempo de execução do melhor caso é  $\Omega(g(n))$ .

### 3.1-7

Prove que  $o(g(n)) \cap \omega(g(n))$  é o conjunto vazio.

### 3.1-8

Podemos estender nossa notação ao caso de dois parâmetros  $n$  e  $m$  que podem tender a infinito independentemente a taxas distintas. Para uma dada função  $g(n, m)$ , denotamos por  $O(g(n, m))$  o conjunto de funções

$$O(g(n, m)) = \{f(n, m) : \text{existem constantes positivas } c, n_0 \text{ e } m_0 \text{ tais que } 0 \leq f(n, m) \leq cg(n, m) \text{ para todo } n \geq n_0 \text{ e } m \geq m_0\}.$$

Forneça definições correspondentes para  $\Omega(g(n, m))$  e  $\Theta(g(n, m))$ .

## 3.2 Notações padrão e funções comuns

Esta seção revê algumas funções e notações matemáticas padrão e explora os relacionamentos entre elas. A seção também ilustra o uso das notações assintóticas.

### Monotonidade

Uma função  $f(n)$  é **monotonicamente crescente** (ou **monotonamente crescente**) se  $m \leq n$  implica  $f(m) \leq f(n)$ . De modo semelhante, ela é **monotonicamente decrescente** (ou **monotonamente decrescente**) se  $m \leq n$  implica  $f(m) \geq f(n)$ . Uma função  $f(n)$  é **estritamente crescente** se  $m < n$  implica  $f(m) < f(n)$  e **estritamente decrescente** se  $m < n$  implica  $f(m) > f(n)$ .

### Pisos e tetos

Para qualquer número real  $x$ , denotamos o maior inteiro menor que ou igual a  $x$  por  $\lfloor x \rfloor$  (lê-se “o piso de  $x$ ”) e o menor inteiro maior que ou igual a  $x$  por  $\lceil x \rceil$  (lê-se “o teto de  $x$ ”). Para todo  $x$  real,

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

Para qualquer inteiro  $n$ ,

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

e, para qualquer número real  $n \geq 0$  e inteiros  $a, b > 0$ ,

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

$$40 \lceil \lfloor n/a \rfloor/b \rceil = \lfloor n/ab \rfloor, \quad (3.5)$$

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

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

A função piso  $f(x) = \lfloor x \rfloor$  é monotonicamente crescente, como também a função teto  $f(x) = \lceil x \rceil$ .

## Aritmética modular

Para qualquer inteiro  $a$  e qualquer inteiro positivo  $n$ , o valor  $a \bmod n$  é o **resto** (ou **resíduo**) do quociente  $a/n$ :

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

Dada uma noção bem definida do resto da divisão de um inteiro por outro, é conveniente fornecer uma notação especial para indicar a igualdade de restos. Se  $(a \bmod n) = (b \bmod n)$ , escrevemos  $a \equiv b \pmod{n}$  e dizemos que  $a$  é **equivalente a  $b$** , módulo  $n$ . Em outras palavras,  $a \equiv b \pmod{n}$  se  $a$  e  $b$  têm o mesmo resto quando divididos por  $n$ . De modo equivalente,  $a \equiv b \pmod{n}$  se e somente se  $n$  é um divisor de  $b - a$ . Escrevemos  $a \not\equiv b \pmod{n}$  se  $a$  não é equivalente a  $b$ , módulo  $n$ .

## Polinômios

Dado um inteiro não negativo  $d$ , um **polinômio em  $n$  de grau  $d$**  é uma função  $p(n)$  da forma

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

onde as constantes  $a_0, a_1, \dots, a_d$  são os **coeficientes** do polinômio e  $a_d \neq 0$ . Um polinômio é assintoticamente positivo se e somente se  $a_d > 0$ . No caso de um polinômio assintoticamente positivo  $p(n)$  de grau  $d$ , temos  $p(n) = \Theta(n^d)$ . Para qualquer constante real  $a \geq 0$ , a função  $n^a$  é monotonicamente crescente, e para qualquer constante real  $a \leq 0$ , a função  $n^a$  é monotonicamente decrescente. Dizemos que uma função  $f(n)$  é **polinomialmente limitada** se  $f(n) = O(n^k)$  para alguma constante  $k$ .

## Exponenciais

Para todos os valores  $a \neq 0$ ,  $m$  e  $n$  reais, temos as seguintes identidades:

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

Para todo  $n$  e  $a \geq 1$ , a função  $a^n$  é monotonicamente crescente em  $n$ . Quando conveniente, consideraremos  $0^0 = 1$ .

As taxas de crescimento de polinômios e exponenciais podem ser relacionadas pelo fato a seguir. Para todas as constantes reais  $a$  e  $b$  tais que  $a > 1$ ,

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

da qual podemos concluir que

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

Portanto, qualquer função exponencial com uma base estritamente maior que 1 cresce mais rapidamente que qualquer função polinomial.

Usando  $e$  para denotar 2,71828 ..., a base da função logaritmo natural, temos para todo  $x$  real,

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

onde “!” denota a função factorial, definida mais adiante nesta seção. Para todo  $x$  real, temos a desigualdade

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

onde a igualdade se mantém válida somente quando  $x = 0$ . Quando  $|x| \leq 1$ , temos a aproximação

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

Quando  $x \rightarrow 0$ , a aproximação de  $e^x$  por  $1 + x$  é bastante boa:

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

(Nessa equação, a notação assintótica é usada para descrever o comportamento de limite como  $x \rightarrow 0$  em lugar de  $x \rightarrow \infty$ .) Temos, para todo  $x$ ,

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

## Logaritmos

Utilizaremos as seguintes notações:

$$\lg n = \log_2 n \quad (\text{logaritmo binário}),$$

$$\ln n = \log_e n \quad (\text{logaritmo natural}),$$

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

$$42 \quad | \quad \lg \lg n = \lg(\lg n) \quad (\text{composição}).$$

Uma importante convenção notacional que adotaremos é que as *funções logarítmicas se aplicarão apenas ao próximo termo na fórmula*; assim,  $\lg n + k$  significará  $(\lg n) + k$  e não  $\lg(n + k)$ . Se mantivermos  $b > 1$  constante, então para  $n > 0$ , a função  $\log_b n$  será estritamente crescente.

Para todo  $a > 0$ ,  $b > 0$ ,  $c > 0$  e  $n$  real,

$$a = b^{\log_b a}, \quad (3.14)$$

$$\log_c(ab) = \log_c a + \log_c b,$$

$$\log_b a^n = n \log_b a,$$

$$\log_b a = \frac{\log_c a}{\log_c b},$$

$$\log_b(1/a) = -\log_b a, \quad (3.15)$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a},$$

onde, em cada equação anterior, as bases de logaritmos não são iguais a 1.

Pela equação (3.14), a mudança da base de um logaritmo de uma constante para outra só altera o valor do logaritmo por um fator constante, e assim usaremos com freqüência a notação “ $\lg n$ ” quando não nos importarmos com fatores constantes, como na notação  $O$ . Os cientistas da computação consideram 2 a base mais natural para logaritmos, porque muitos algoritmos e estruturas de dados envolvem a divisão de um problema em duas partes.

Existe uma expansão de série simples para  $\ln(1 + x)$  quando  $|x| < 1$ :

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

Também temos as seguintes desigualdades para  $x > -1$ :

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

onde a igualdade é válida somente para  $x = 0$ .

Dizemos que uma função  $f(n)$  é *polilogaritmicamente limitada* se  $f(n) = O(\lg^k n)$  para alguma constante  $k$ . Podemos relacionar o crescimento de polinômios e polilogaritmos substituindo  $n$  por  $\lg n$  e  $\alpha$  por  $2^\alpha$  na equação (3.9), resultando em:

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

A partir desse limite, podemos concluir que

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

para qualquer constante  $\alpha > 0$ . Desse modo, qualquer função polinomial positiva cresce mais rapidamente que qualquer função polilogarítmica.

## Fatoriais

A notação  $n!$  (lê-se “ $n$  fatorial”) é definida para inteiros  $n \geq 0$  como

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

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

Um limite superior fraco na função fatorial é  $n! \leq n^n$ , pois cada um dos  $n$  termos no produto do fatorial é no máximo  $n$ . A *aproximação de Stirling*,

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

onde  $e$  é a base do logaritmo natural, nos dá um limite superior mais restrito, e um limite inferior também. Pode-se demonstrar que (consulte o Exercício 3.2-3)

$$n! = o(n^n), \quad (3.18)$$

$$n! = \omega(2^n),$$

$$\lg(n!) = \Theta(n \lg n),$$

onde a aproximação de Stirling é útil na demonstração da equação (3.18). A equação a seguir também é válida para todo  $n \geq 1$ :

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

onde

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

## Iteração funcional

Usamos a notação  $f^{(i)}(n)$  para denotar a função  $f(n)$  aplicada iterativamente  $i$  vezes a um valor inicial  $n$ . Formalmente, seja  $f(n)$  uma função sobre os reais. Para inteiros não negativos  $i$ , definimos recursivamente:

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

Por exemplo, se  $f(n) = 2n$ , então  $f^{(i)}(n) = 2^i n$ .

## A função logaritmo repetido

Usamos a notação  $\lg^* n$  (lê-se “log asterisco de  $n$ ”) para denotar o logaritmo repetido, que é definido como a seguir. Seja  $\lg^{(i)} n$  definida da maneira anterior, com  $f(n) \lg n$ . Como o logaritmo de um número não positivo é indefinido,  $\lg^{(i)} n$  só é definido se  $\lg^{(i-1)} n > 0$ . Certifique-se de distinguir  $\lg^{(i)} n$  (a função logaritmo aplicada  $i$  vezes em sucessão, começando com o argumento  $n$ ) de  $\lg^i n$  (o logaritmo de  $n$  elevado à  $i$ -ésima potência). A função logaritmo repetido é definida como

$$44 \quad \lg^* n = \min \{i \geq 0 : \lg^{(i)} n \leq 1\}.$$

O logaritmo repetido é uma função que cresce *muito* lentamente:

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

Tendo em vista que o número de átomos no universo visível é estimado em cerca  $10^{80}$ , que é muito menor que  $2^{65536}$ , raramente encontraremos uma entrada de tamanho  $n$  tal que  $\lg^* n > 5$ .

## Números de Fibonacci

Os *números de Fibonacci* são definidos pela seguinte recorrência:

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

Portanto, cada número de Fibonacci é a soma dos dois números anteriores, produzindo a sequência

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

Os números de Fibonacci estão relacionados com a *razão áurea*  $\phi$  e a seu conjugado  $\hat{\phi}$ , que são dados pelas seguintes fórmulas:

$$\begin{aligned}\phi &= \frac{1 + \sqrt{5}}{2} \\ &= 1,61803\dots, \\ \hat{\phi} &= \frac{1 - \sqrt{5}}{2} \\ &= -0,61803\dots.\end{aligned}\tag{3.22}$$

Especificamente, temos

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}\tag{3.23}$$

que pode ser demonstrada por indução (Exercício 3.2-6). Como  $|\hat{\phi}| < 1$ , temos  $|\hat{\phi}|/\sqrt{5} < 1/\sqrt{5} < 1/2$ , de modo que o  $i$ -ésimo número de Fibonacci  $F_i$  é igual a  $\phi^i/\sqrt{5}$  arredondado para o inteiro mais próximo. Desse modo, os números de Fibonacci crescem exponencialmente.

## Exercícios

### 3.2-1

Mostre que, se  $f(n)$  e  $g(n)$  são funções monotonicamente crescentes, então também o são as funções  $f(n) + g(n)$  e  $f(g(n))$ , e se  $f(n)$  e  $g(n)$  são além disso não negativas, então  $f(n) \cdot g(n)$  é monotonicamente crescente.

### 3.2-2

Prove a equação (3.15).

### 3.2-3

Prove a equação (3.18). Prove também que  $n! = \omega(2^n)$  e que  $n! = o(n^n)$ .

### 3.2-4 \*

A função  $\lceil \lg n \rceil!$  é polinomialmente limitada? A função  $\lceil \lg \lg n \rceil!$  é polinomialmente limitada?

### 3.2-5 \*

Qual é assintoticamente maior:  $\lg(\lg^* n)$  ou  $\lg^*(\lg n)$ ?

### 3.2-6

Prove por indução que o  $i$ -ésimo número de Fibonacci satisfaz à igualdade

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

onde  $\phi$  é a razão áurea e  $\hat{\phi}$  é seu conjugado.

### 3.2-7

Prove que, para  $i \geq 0$ , o  $(i+2)$ -ésimo número de Fibonacci satisfaz a  $F_{i+2} \geq \phi^i$ .

## Problemas

### 3-1 Comportamento assintótico de polinômios

Seja

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

onde  $a_d > 0$ , um polinômio de grau  $d$  em  $n$ , e seja  $k$  uma constante. Use as definições das notações assintóticas para provar as propriedades a seguir.

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

### 3-2 Crescimentos assintóticos relativos

Indique, para cada par de expressões  $(A, B)$  na tabela a seguir, se  $A$  é  $O$ ,  $o$ ,  $\Omega$ ,  $\omega$  ou  $\Theta$  de  $B$ . Suponha que  $k \geq 1$ ,  $\epsilon > 0$  e  $c > 1$  são constantes. Sua resposta deve estar na forma da tabela, com “sim” ou “não” escrito em cada retângulo.

	<i>A</i>	<i>B</i>	<i>O</i>	<i>o</i>	$\Omega$	$\omega$	$\Theta$
a.	$\lg^k n$	$n^\epsilon$					
b.	$n^k$	$c^n$					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	$2^n$	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

### 3-3 Ordenação por taxas de crescimento assintóticas

- a. Ordene as funções a seguir por ordem de crescimento; ou seja, encontre um arranjo  $g_1, g_2, \dots, g_{30}$  das funções que satisfazem a  $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \dots, g_{29} = \Omega(g_{30})$ . particione sua lista em classes de equivalência tais que  $f(n)$  e  $g(n)$  estejam na mesma classe se e somente se  $f(n) = \Theta(g(n))$ .

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

- b. Dê um exemplo de uma única função não negativa  $f(n)$  tal que, para todas as funções  $g_i(n)$  da parte (a),  $f(n)$  não seja nem  $O(g_i(n))$ , nem  $\Omega(g_i(n))$ .

### 3-4 Propriedades da notação assintótica

Sejam  $f(n)$  e  $g(n)$  funções assintoticamente positivas. Prove ou conteste cada uma das seguintes conjecturas.

- a.  $f(n) = O(g(n))$  implica  $g(n) = O(f(n))$ .
- b.  $f(n) + g(n) = \Theta(\min(f(n), g(n)))$ .
- c.  $f(n) = O(g(n))$  implica  $\lg(f(n)) = O(\lg(g(n)))$ , onde  $\lg(g(n)) \geq 1$  e  $f(n) \geq 1$  para todo  $n$  suficientemente grande.
- d.  $f(n) = O(g(n))$  implica  $2^{f(n)} = O(2^{g(n)})$ .
- e.  $f(n) = O((f(n))^2)$ .
- f.  $f(n) = O(g(n))$  implica  $g(n) = \Omega(f(n))$ .
- g.  $f(n) = \Theta(f(n/2))$ .
- h.  $f(n) + o(f(n)) = \Theta(f(n))$ .

### 3-5 Variações sobre O e Ω

Alguns autores definem  $\Omega$  de um modo ligeiramente diferente do modo como nós definimos; vamos usar  $\tilde{\Omega}$  (lê-se “ômega infinito”) para essa definição alternativa. Dizemos que  $f(n) = \tilde{\Omega}(g(n))$  se existe uma constante positiva  $c$  tal que  $f(n) \geq cg(n) \geq 0$  para infinitamente muitos inteiros  $n$ .

- Mostre que, para duas funções quaisquer  $f(n)$  e  $g(n)$  que são assintoticamente não negativas,  $f(n) = O(g(n))$  ou  $f(n) = \tilde{\Omega}(g(n))$  ou ambas, enquanto isso não é verdade se usamos  $\Omega$  em lugar de  $\tilde{\Omega}$ .
- Descreva as vantagens e as desvantagens potenciais de se usar  $\tilde{\Omega}$  em vez de  $\Omega$  para caracterizar os tempos de execução de programas.

Alguns autores também definem  $O'$  de um modo ligeiramente diferente; vamos usar  $O'$  para a definição alternativa. Dizemos que  $f(n) = O'(g(n))$  se e somente se  $|f(n)| = O(g(n))$ .

- O que acontece para cada sentido de “se e somente se” no Teorema 3.1 se substituirmos  $O$  por  $O'$ , mas ainda usarmos  $\Omega$ ?

Alguns autores definem  $\tilde{O}$  (lê-se “o’ suave”) para indicar  $O$  com fatores logarítmicos ignorados:

$$\tilde{O}(g(n)) = \{f(n) : \text{existem constantes positivas } c, k \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \lg^k(n) \text{ para todo } n \geq n_0\}.$$

- Defina  $\tilde{\Omega}$  e  $\tilde{\Theta}$  de maneira semelhante. Prove a analogia correspondente ao Teorema 3.1.

### 3-6 Funções repetidas

O operador de iteração \* usado na função  $\lg^*$  pode ser aplicado a qualquer função monotonicamente crescente  $f(n)$  sobre os reais. Para uma dada constante  $c \in \mathbb{R}$ , definimos a função repetida (ou iterada)  $f_c^*$  por

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

que não necessita ser bem definida em todos os casos. Em outras palavras, a quantidade  $f_c^*(n)$  é o número de aplicações repetidas da função  $f$  necessárias para reduzir seu argumento a  $c$  ou menos.

Para cada uma das funções  $f(n)$  e constantes  $c$  a seguir, forneça um limite tão restrito quanto possível sobre  $f_c^*(n)$ .

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

## Notas do capítulo

Knuth [182] traça a origem da notação  $O$  até um texto de teoria dos números escrito por P. Bachmann em 1892. A notação  $o$  foi criada por E. Landau em 1909, para sua descrição da distribuição de números primos. As notações  $\Omega$  e  $\Theta$  foram defendidas por Knuth [186] para corrigir a prática popular, mas tecnicamente ruim, de se usar na literatura a notação  $O$  para limites superiores e inferiores. Muitas pessoas continuam a usar a notação  $O$  onde a notação  $\Theta$  é mais precisa tecnicamente. Uma discussão adicional da história e do desenvolvimento de notações assintóticas pode ser encontrada em Knuth [182, 186] e em Brassard e Bratley [46].

Nem todos os autores definem as notações assintóticas do mesmo modo, embora as várias definições concordem na maioria das situações comuns. Algumas das definições alternativas envolvem funções que não são assintoticamente não negativas, desde que seus valores absolutos sejam adequadamente limitados.

A equação (3.19) se deve a Robbins [260]. Outras propriedades de funções matemáticas elementares podem ser encontradas em qualquer bom manual de referência de matemática, como Abramowitz e Stegun [1] ou Zwillinger [320], ou em um livro de cálculo, como Apostol [18] ou Thomas e Finney [296]. Knuth [182] e também Graham, Knuth e Patashnik [132] contêm uma grande quantidade de material sobre matemática discreta, como a que se utiliza em ciência da computação.

---

## *Capítulo 4*

### *Recorrências*

Como observamos na Seção 2.3.2, quando um algoritmo contém uma chamada recursiva a ele próprio, seu tempo de execução pode freqüentemente ser descrito por uma recorrência. Uma **recorrência** é uma equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores. Por exemplo, vimos na Seção 2.3.2 que o tempo de execução do pior caso  $T(n)$  do procedimento MERGE-SORT poderia ser descrito pela recorrência

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

cuja solução se afirmava ser  $T(n) = \Theta(n \lg n)$ .

Este capítulo oferece três métodos para resolver recorrências – ou seja, para obter limites assintóticos “ $\Theta$ ” ou “ $O$ ” sobre a solução. No **método de substituição**, supomos um limite hipotético, e depois usamos a indução matemática para provar que nossa suposição era correta. O **método de árvore de recursão** converte a recorrência em uma árvore cujos nós representam os custos envolvidos em diversos níveis da recursão; usamos técnicas para limitar somatórios com a finalidade de solucionar a recorrência. O **método mestre** fornece limites para recorrências da forma

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

onde  $a \geq 1$ ,  $b > 1$  e  $f(n)$  é uma função dada; o método requer memorização de três casos mas, depois que você o fizer, será fácil descobrir limites assintóticos para muitas recorrências simples.

#### **Detalhes técnicos**

Na prática, negligenciamos certos detalhes técnicos quando enunciamos e resolvemos recorrências. Um bom exemplo de um detalhe que freqüentemente é ignorado é a suposição de argumentos inteiros para funções. Normalmente, o tempo de execução  $T(n)$  de um algoritmo só é definido quando  $n$  é um inteiro, pois, para a maior parte dos algoritmos, o tamanho da entrada é sempre um inteiro. Por exemplo, a recorrência que descreve o tempo de execução do pior caso de MERGE-SORT é na realidade

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

As condições limite representam outra classe de detalhes que em geral ignoramos. Tendo em vista que o tempo de execução de um algoritmo sobre uma entrada de dimensões constantes é uma constante, as recorrências que surgem dos tempos de execução de algoritmos geralmente têm  $T(n) = \Theta(1)$  para  $n$  suficientemente pequeno. Em consequência disso, por conveniência, em geral omitiremos declarações sobre as condições limite de recorrências e iremos supor que  $T(n)$  é constante para  $n$  pequeno. Por exemplo, normalmente enunciamos a recorrência (4.1) como

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

sem fornecer explicitamente valores para  $n$  pequeno. A razão é que, embora a mudança no valor de  $T(1)$  altere a solução para a recorrência, normalmente a solução não muda por mais de um fator constante, e assim a ordem de crescimento não é alterada.

Quando enunciamos e resolvemos recorrências, com freqüência omitimos pisos, tetos e condições limite. Seguimos em frente sem esses detalhes, e mais tarde definimos se eles têm importância ou não. Em geral, eles não têm importância, mas é importante saber quando têm. A experiência ajuda, e também alguns teoremas declarando que esses detalhes não afetam os limites assintóticos de muitas recorrências encontradas na análise de algoritmos (ver Teorema 4.1). Portanto, neste capítulo, examinaremos alguns desses detalhes para mostrar os ótimos métodos de solução de pontos de recorrência.

## 4.1 O método de substituição

O método de substituição para resolver recorrências envolve duas etapas:

1. Pressupor a forma da solução.
2. Usar a indução matemática para encontrar as constantes e mostrar que a solução funciona.

O nome vem da substituição da resposta pressuposta para a função quando a hipótese induktiva é aplicada a valores menores. Esse método é eficiente, mas obviamente só pode ser aplicado em casos nos quais é fácil pressupor a forma da resposta.

O método de substituição pode ser usado para estabelecer limites superiores ou inferiores sobre uma recorrência. Como exemplo, vamos determinar um limite superior sobre a recorrência

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

que é semelhante às recorrências (4.2) e (4.3). Supomos que a solução é  $T(n) = O(n \lg n)$ . Nossa tarefa é provar que  $T(n) \leq cn \lg n$  para uma escolha adequada da constante  $c > 0$ . Começamos supondo que esse limite se mantém válido para  $\lfloor n/2 \rfloor$ , ou seja, que  $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$ . A substituição na recorrência produz

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

onde o último passo é válido desde que  $c \geq 1$ .

Agora, a indução matemática exige que mostremos que nossa solução se mantém válida para as condições limite. Normalmente, fazemos isso mostrando que as condições limite são adequadas para casos básicos da prova indutiva. No caso da recorrência (4.4), devemos mostrar que podemos escolher uma constante  $c$  grande o suficiente para que o limite  $T(n) \leq cn \lg n$  também funcione para as condições limite. Às vezes, essa exigência pode levar a problemas. Vamos supor, como argumento, que  $T(1) = 1$  seja a única condição limite da recorrência. Então, para  $n = 1$ , o limite  $T(n) \leq cn \lg n$  produz  $T(1) \leq c 1 \lg 1 = 0$ , o que está em desacordo com  $T(1) = 1$ . Consequentemente, o caso básico de nossa prova indutiva deixa de ser válido.

Essa dificuldade para provar uma hipótese indutiva para uma condição limite específica pode ser facilmente contornada. Por exemplo, na recorrência (4.4), tiramos proveito do fato de que a notação assintótica só exige que demonstremos que  $T(n) \leq cn \lg n$  para  $n \geq n_0$ , onde  $n_0$  é uma constante de nossa escolha. A idéia é remover a difícil condição limite  $T(1) = 1$  da consideração na prova indutiva. Observe que, para  $n > 3$ , a recorrência não depende diretamente de  $T(1)$ . Desse modo, podemos substituir  $T(1)$  por  $T(2)$  e  $T(3)$  como os casos básicos na prova indutiva, deixando  $n_0 = 2$ . Observe que fazemos uma distinção entre o caso básico da recorrência ( $n = 1$ ) e os casos básicos da prova indutiva ( $n = 2$  e  $n = 3$ ). Da recorrência, derivamos  $T(2) = 4$  e  $T(3) = 5$ . A prova indutiva de que  $T(n) \leq cn \lg n$  para alguma constante  $c \geq 1$  pode agora ser completada escolhendo-se um valor de  $c$  grande o bastante para que  $T(2) \leq c 2 \lg 2$  e  $T(3) \leq c 3 \lg 3$ . Como observamos, qualquer valor de  $c \geq 2$  é suficiente para que os casos básicos de  $n = 2$  e  $n = 3$  sejam válidos. Para a maioria das recorrências que examinaremos, é simples estender as condições limite para fazer a hipótese indutiva funcionar para  $n$  pequeno.

## Como fazer um bom palpite

Infelizmente, não há nenhum modo geral de adivinhar as soluções corretas para recorrências. Pressupor uma solução exige experiência e, ocasionalmente, criatividade. Entretanto, por sorte, existem algumas heurísticas que podem ajudá-lo a se tornar um bom criador de suposições. Você também poderá usar árvores de recursão, que veremos na Seção 4.2, para gerar boas hipóteses.

Se uma recorrência for semelhante a uma que você já tenha visto antes, então será razoável supor uma solução semelhante. Como exemplo, considere a recorrência

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

que parece difícil devido ao “17” acrescentado ao argumento de  $T$  no lado direito. Intuitivamente, porém, esse termo adicional não pode afetar de maneira substancial a solução para a recorrência. Quando  $n$  é grande, a diferença entre  $T(\lfloor n/2 \rfloor)$  e  $T(\lfloor n/2 \rfloor + 17)$  não é tão grande: ambos os termos cortam  $n$  quase uniformemente pela metade. Em consequência disso, fazemos a suposição de que  $T(n) = O(n \lg n)$ , que você pode verificar como correto usando o método de substituição (ver Exercício 4.1-5).

Outro caminho para fazer uma boa suposição é provar informalmente limites superiores e inferiores sobre a recorrência, e então reduzir o intervalo de incerteza. Por exemplo, podemos começar com o limite inferior  $T(n) = \Omega(n)$  para a recorrência (4.4), pois temos o termo  $n$  na recorrência, e podemos demonstrar um limite superior inicial  $T(n) = O(n^2)$ . Então, podemos diminuir gradualmente o limite superior e elevar o limite inferior, até convergirmos sobre a solução correta, assintoticamente restrita,  $T(n) = \Theta(n \lg n)$ .

## Sutilidades

Há momentos em que é possível fazer uma suposição correta em um limite assintótico sobre a solução de uma recorrência, mas, de algum modo, a matemática não parece funcionar na indu-

ção. Em geral, o problema é que a hipótese indutiva não é forte o bastante para provar o limite detalhado. Quando você chegar a um nó como esse, revisar a suposição subtraindo um termo de mais baixa ordem freqüentemente permitirá que a matemática funcione.

Considere a recorrência

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

Supomos por hipótese que a solução seja  $O(n)$  e tentamos mostrar que  $T(n) \leq cn$  para uma escolha apropriada da constante  $c$ . Substituindo nossa suposição na recorrência, obtemos

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

o que não implica  $T(n) \leq cn$  para qualquer escolha de  $c$ . É uma tentação experimentar um palpite maior, digamos  $T(n) = O(n^2)$ , o que poderia funcionar; porém, de fato, nossa suposição de que a solução é  $T(n) = O(n)$  é correta. No entanto, para mostrar isso, devemos criar uma hipótese indutiva mais forte.

Intuitivamente, nossa suposição é quase correta: a única diferença é a constante 1, um termo de mais baixa ordem. Apesar disso, a indução matemática não funciona, a menos que provemos a forma exata da hipótese indutiva. Superamos nossa dificuldade *subtraindo* um termo de mais baixa ordem da nossa suposição anterior. Nossa nova suposição é  $T(n) \leq cn - b$ , onde  $b \geq 0$  é constante. Agora temos

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

desde que  $b \geq 1$ . Como antes, a constante  $c$  deve ser escolhida com um valor grande o suficiente para tratar as condições limite.

A maioria das pessoas acha antiintuitiva a idéia de subtrair um termo de mais baixa ordem. Afinal, se a matemática não funciona, não deveríamos estar aumentando nossa suposição? A chave para entender esse passo é lembrar que estamos usando a indução matemática: podemos provar algo mais forte para um determinado valor, supondo algo mais forte para valores menores.

## Como evitar armadilhas

É fácil errar na utilização da notação assintótica. Por exemplo, na recorrência (4.4), podemos “provar” falsamente que  $T(n) = O(n)$ , supondo  $T(n) \leq cn$ , e então argumentando que

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

pois  $c$  é uma constante. O erro é que não provamos a *forma exata* da hipótese indutiva, ou seja, que  $T(n) \leq cn$ .

## Como trocar variáveis

Às vezes, um pouco de manipulação algébrica pode tornar uma recorrência desconhecida semelhante a uma que você já viu antes. Como exemplo, considere a recorrência

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

que parece difícil. Entretanto, podemos simplificar essa recorrência com uma troca de variáveis. Por conveniência, não nos preocuparemos em arredondar valores, como  $\sqrt{n}$ , para inteiros. Renomear  $m = \lg n$  produz

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

Agora podemos renomear  $S(m) = T(2^m)$  para produzir a nova recorrência

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

que é muito semelhante à recorrência (4.4). De fato, essa nova recorrência tem a mesma solução:  $S(m) = O(m \lg m)$ . Trocando de volta de  $S(m)$  para  $T(n)$ , obtemos  $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$ .

## Exercícios

### 4.1-1

Mostre que a solução de  $T(n) = T(\lceil n/2 \rceil) + 1$  é  $O(\lg n)$ .

### 4.1-2

Vimos que a solução de  $T(n) = 2T(\lfloor n/2 \rfloor) + n$  é  $O(n \lg n)$ . Mostre que a solução dessa recorrência também é  $\Omega(n \lg n)$ . Conclua que a solução é  $\Theta(n \lg n)$ .

### 4.1-3

Mostre que, supondo uma hipótese indutiva diferente, podemos superar a dificuldade com a condição limite  $T(1) = 1$  para a recorrência (4.4), sem ajustar as condições limite para a prova induktiva.

### 4.1-4

Mostre que  $\Theta(n \lg n)$  é a solução para a recorrência “exata” (4.2) para a ordenação por intercalação.

### 4.1-5

Mostre que a solução para  $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$  é  $O(n \lg n)$ .

### 4.1-6

Resolva a recorrência  $T(n) = 2T(\sqrt{n})$ , fazendo uma troca de variáveis. Sua solução deve ser assintoticamente restrita. Não se preocupe em saber se os valores são integrais.

## 4.2 O método de árvore de recursão

Embora o método de substituição possa fornecer uma prova sucinta de que uma solução para uma recorrência é correta, às vezes é difícil apresentar uma boa suposição. Traçar uma árvore de recursão, como fizemos em nossa análise da recorrência de ordenação por intercalação na Seção 2.3.2, é um caminho direto para se criar uma boa suposição. Em uma **árvore de recursão**, cada nó representa o custo de um único subproblema em algum lugar no conjunto de invocações de funções recursivas. Somamos os custos dentro de cada nível da árvore para obter um conjunto de custos por nível, e então somamos todos os custos por nível para determinar o custo total de todos os níveis da recursão. As árvores de recursão são particularmente úteis quando a recorrência descreve o tempo de execução de um algoritmo de dividir e conquistar.

Uma árvore de recursão é mais bem usada para gerar uma boa suposição, que é então verificada pelo método de substituição. Quando utiliza uma árvore de recursão para gerar uma boa suposição, você freqüentemente pode tolerar uma pequena quantidade de “sujeira”, pois irá verificar sua suposição mais tarde. Porém, se for muito cuidadoso ao criar uma árvore de recursão e somar os custos, você poderá usar a árvore de recursão como uma prova direta de uma solução para uma recorrência. Nesta seção, usaremos árvores de recursão para gerar boas suposições e, na Seção 4.4, utilizaremos árvores de recursão diretamente para provar o teorema que forma a base do método mestre.

Por exemplo, vamos ver como uma árvore de recursão forneceria uma boa suposição para a recorrência  $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ . Começamos concentrando nossa atenção em encontrar um limite superior para a solução. Considerando que sabemos que pisos e tetos são normalmente insatisfatórios para resolver recorrências (aqui está um exemplo de sujeira que podemos tolerar), criamos uma árvore de recursão para a recorrência  $T(n) = 3T(n/4) + cn^2$ , tendo estabelecido o coeficiente constante implícito  $c > 0$ .

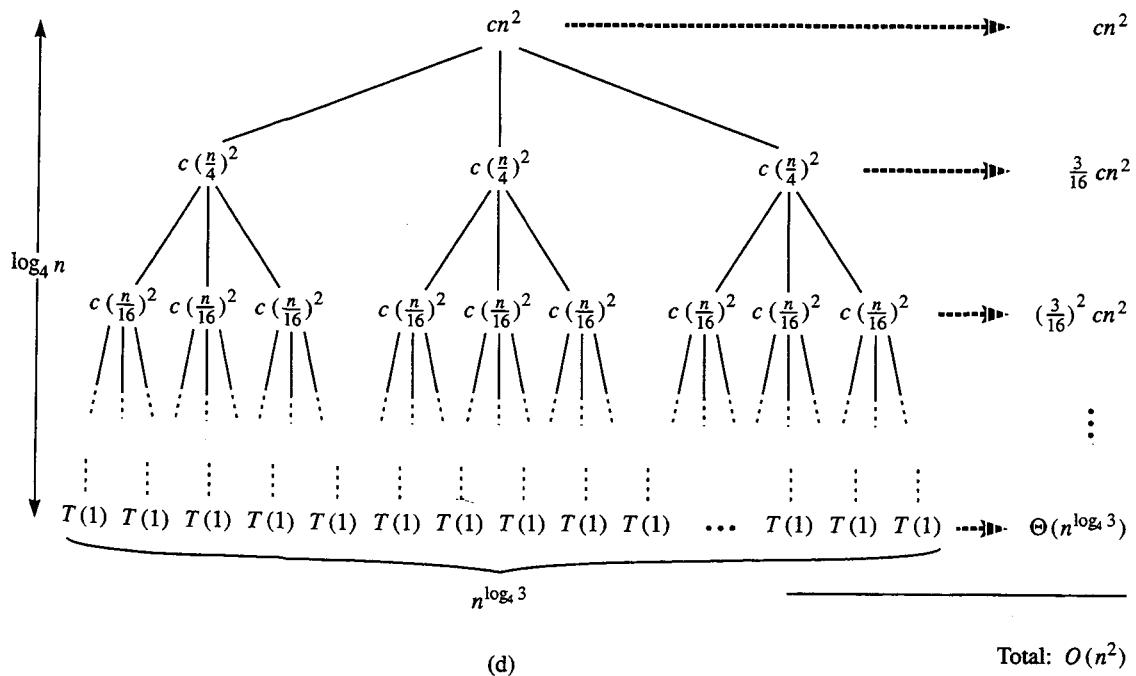
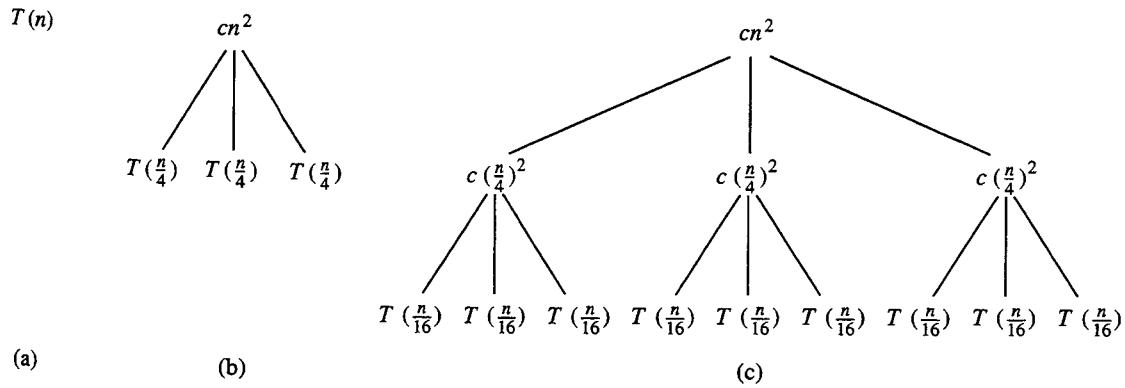
A Figura 4.1 mostra a derivação da árvore de recursão para  $T(n) = 3T(n/4) + cn^2$ . Por conveniência, supomos que  $n$  é uma potência exata de 4 (outro exemplo de sujeira tolerável). A parte (a) da figura mostra  $T(n)$  que, na parte (b), foi expandida até uma árvore equivalente representando a recorrência. O termo  $cn^2$  na raiz representa o custo no nível de recursão superior, e as três subárvores da raiz representam os custos resultantes dos subproblemas de tamanho  $n/4$ . A parte (c) mostra esse processo levado um passo adiante pela expansão de cada nó com custo  $T(n/4)$  da parte (b). O custo para cada um dos três filhos da raiz é  $c(n/4)^2$ . Continuamos a expandir cada nó na árvore, desmembrando-o em suas partes constituintes conforme determinado pela recorrência.

Tendo em vista que os tamanhos de subproblemas diminuem à medida que nos afastamos da raiz, eventualmente temos de alcançar uma condição limite. A que distância da raiz nós a encontramos? O tamanho do subproblema para um nó na profundidade  $i$  é  $n/4^i$ . Desse modo, o tamanho do subproblema chega a  $n = 1$  quando  $n/4^i = 1$  ou, de modo equivalente, quando  $i = \log_4 n$ . Assim, a árvore tem  $\log_4 n + 1$  níveis  $(0, 1, 2, \dots, \log_4 n)$ .

Em seguida, determinamos o custo em cada nível da árvore. Cada nível tem três vezes mais nós que o nível acima dele, e assim o número de nós na profundidade  $i$  é  $3^i$ . Como os tamanhos de subproblemas se reduzem por um fator de 4 para cada nível que descemos a partir da raiz, cada nó na profundidade  $i$ , para  $i = 0, 1, 2, \dots, \log_4 n - 1$ , tem o custo de  $c(n/4^i)^2$ . Multiplicando, vemos que o custo total sobre todos os nós na profundidade  $i$ , para  $i = 0, 1, 2, \dots, \log_4 n - 1$ , é  $3^i c(n/4^i)^2 = (3/16)^i cn^2$ . O último nível, na profundidade  $\log_4 n$ , tem  $3^{\log_4 n} = n^{\log_4 3}$  nós, cada qual contribuindo com o custo  $T(1)$ , para um custo total de  $n^{\log_4 3} T(1)$ , que é  $\Theta(n^{\log_4 3})$ .

Agora somamos os custos sobre todos os níveis para determinar o custo correspondente à árvore inteira:

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



**FIGURA 4.1** A construção de uma árvore de recursão para a recorrência  $T(n) = 3T(n/4) + cn^2$ . A parte (a) mostra  $T(n)$ , que é progressivamente expandido nas partes b a d para formar a árvore de recursão. A árvore completamente expandida da parte (d) tem altura  $\log_4 n$  (ela tem  $\log_4 n + 1$  níveis)

Essa última fórmula parece um pouco confusa até percebermos que é possível mais uma vez tirar proveito de pequenas porções de sujeira e usar uma série geométrica decrescente infinita como limite superior. Voltando um passo atrás e aplicando a equação (A.6), temos

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

Desse modo, derivamos uma suposição de  $T(n) = O(n^2)$  para nossa recorrência original  $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ . Nesse exemplo, os coeficientes de  $cn^2$  formam uma série geométrica decrescente e, pela equação (A.6), a soma desses coeficientes é limitada na parte superior pela constante  $16/13$ . Tendo em vista que a contribuição da raiz para o custo total é  $cn^2$ , a raiz contribui com uma fração constante do custo total. Em outras palavras, o custo total da árvore é dominado pelo custo da raiz.

De fato, se  $O(n^2)$  é realmente um limite superior para a recorrência (como verificaremos em breve), então ele deve ser um limite restrito. Por quê? A primeira chamada recursiva contribui com o custo  $\Theta(n^2)$ , e então  $\Omega(n^2)$  deve ser um limite inferior para a recorrência.

Agora podemos usar o método de substituição para verificar que nossa suposição era correta, isto é,  $T(n) = O(n^2)$  é um limite superior para a recorrência  $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ . Queremos mostrar que  $T(n) \leq dn^2$  para alguma constante  $d > 0$ . Usando a mesma constante  $c > 0$  de antes, temos

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

onde a última etapa é válida desde que  $d \geq (16/13)c$ .

Como outro exemplo mais complicado, a Figura 4.2 mostra a árvore de recursão para

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

(Novamente, omitimos as funções piso e teto por simplicidade.) Como antes,  $c$  representa o fator constante no termo  $O(n)$ . Quando somamos os valores em todos os níveis da árvore de recursão, obtemos um valor  $cn$  para cada nível. O caminho mais longo da raiz até uma folha é  $n \rightarrow (2/3)n \rightarrow (2/3)^2n \rightarrow \dots \rightarrow 1$ . Tendo em vista que  $(2/3)^k n = 1$  quando  $k = \log_{3/2} n$ , a altura da árvore é  $\log_{3/2} n$ .

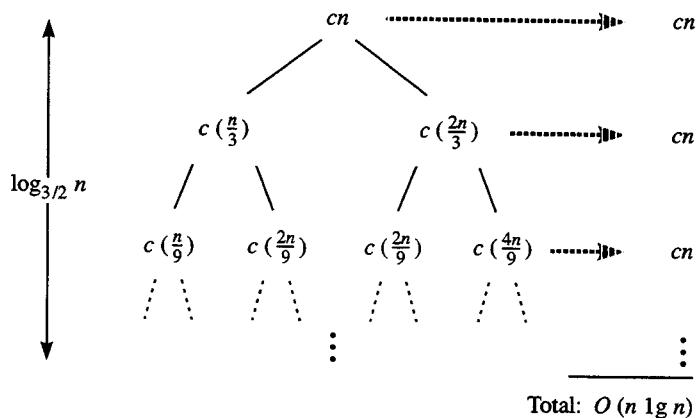


FIGURA 4.2 Uma árvore de recursão para a recorrência  $T(n) = T(n/3) + T(2n/3) + cn$

Intuitivamente, esperamos que a solução para a recorrência seja no máximo o número de níveis vezes o custo de cada nível, ou  $O(cn \log_{3/2} n) = O(n \lg n)$ . O custo total é distribuído de

modo uniforme ao longo dos níveis da árvore de recursão. Existe uma complicação aqui: ainda temos de considerar o custo das folhas. Se essa árvore de recursão fosse uma árvore binária completa de altura  $\log_{3/2} n$ , haveria  $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$  folhas. Como o custo de cada folha é uma constante, o custo total de todas as folhas seria então  $\Theta(n^{\log_{3/2} 2})$ , que é  $\omega(n \lg n)$ . Contudo, essa árvore de recursão não é uma árvore binária completa, e assim ela tem menos de  $n^{\log_{3/2} 2}$  folhas. Além disso, à medida que descemos a partir da raiz, mais e mais nós internos estão ausentes. Consequentemente, nem todos os níveis contribuem com um custo exatamente igual a  $cn$ ; os níveis em direção à parte inferior contribuem menos. Poderíamos desenvolver uma contabilidade precisa de todos os custos, mas lembre-se de que estamos apenas tentando apresentar uma suposição para usar no método de substituição. Vamos tolerar a sujeira e tentar mostrar que uma suposição  $O(n \lg n)$  para o limite superior é correta.

Na verdade, podemos usar o método de substituição para verificar aquele  $O(n \lg n)$  é um limite superior para a solução da recorrência. Mostramos que  $T(n) \leq dn \lg n$ , onde  $d$  é uma constante positiva apropriada. Temos

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

desde que  $d \geq c/(\lg 3 - (2/3))$ . Desse modo, não tivemos de executar uma contabilidade de custos mais precisa na árvore de recursão.

## Exercícios

### 4.2-1

Use uma árvore de recursão para determinar um bom limite superior assintótico na recorrência  $T(n) = 3T(\lfloor n/2 \rfloor) + n$ . Use o método de substituição para verificar sua resposta.

### 4.2-2

Demonstre que a solução para a recorrência  $T(n) = T(n/3) + T(2n/3) + cn$ , onde  $c$  é uma constante, é  $\Omega(n \lg n)$ , apelando para uma árvore de recursão.

### 4.2-3

Trace a árvore de recursão para  $T(n) = 4T(\lfloor n/2 \rfloor) + cn$ , onde  $c$  é uma constante, e forneça um limite assintótico restrito sobre sua solução. Verifique o limite pelo método de substituição.

### 4.2-4

Use uma árvore de recursão com o objetivo de fornecer uma solução assintoticamente restrita para a recorrência  $T(n) = T(n - a) + T(a) + cn$ , onde  $a \geq 1$  e  $c > 0$  são constantes.

### 4.2-5

Use uma árvore de recursão para fornecer uma solução assintoticamente restrita para a recorrência  $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$ , onde  $\alpha$  é uma constante no intervalo  $0 < \alpha < 1$  e  $c > 0$  também é uma constante.

## 4.3 O método mestre

O método mestre fornece um processo de “livro de receitas” para resolver recorrências da forma

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

onde  $a \geq 1$  e  $b > 1$  são constantes e  $f(n)$  é uma função assintoticamente positiva. O método mestre exige a memorização de três casos, mas, daí em diante, a solução de muitas recorrências pode ser descoberta com grande facilidade, freqüentemente sem lápis e papel.

A recorrência (4.5) descreve o tempo de execução de um algoritmo que divide um problema de tamanho  $n$  em  $a$  subproblemas, cada um do tamanho  $n/b$ , onde  $a$  e  $b$  são constantes positivas. Os  $a$  subproblemas são resolvidos recursivamente, cada um no tempo  $T(n/b)$ . O custo de dividir o problema e combinar os resultados dos subproblemas é descrito pela função  $f(n)$ . (Ou seja, usando a notação da Seção 2.3.2,  $f(n) = D(n) + C(n)$ .) Por exemplo, a recorrência que surge do procedimento MERGE-SORT tem  $a = 2$ ,  $b = 2$  e  $f(n) = \Theta(n)$ .

Como uma questão de correção técnica, na realidade a recorrência não está bem definida, porque  $n/b$  não poderia ser um inteiro. Porém, a substituição de cada um dos  $a$  termos  $T(n/b)$  por  $T(\lfloor n/b \rfloor)$  ou  $T(\lceil n/b \rceil)$  não afeta o comportamento assintótico da recorrência. (Provaremos isso na próxima seção.) Por essa razão, em geral, consideramos conveniente omitir as funções piso e teto ao se escreverem recorrências de dividir e conquistar dessa forma.

### O teorema mestre

O método mestre depende do teorema a seguir.

#### Teorema 4.1 (Teorema mestre)

Sejam  $a \geq 1$  e  $b > 1$  constantes, seja  $f(n)$  uma função e seja  $T(n)$  definida sobre os inteiros não negativos pela recorrência

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

onde interpretamos  $n/b$  com o significado de  $\lfloor n/b \rfloor$  ou  $\lceil n/b \rceil$ . Então,  $T(n)$  pode ser limitado assintoticamente como a seguir.

1. Se  $f(n) = O(n^{\log_b a - \epsilon})$  para alguma constante  $\epsilon > 0$ , então  $T(n) = \Theta(n^{\log_b a})$ .
2. Se  $f(n) = \Theta(n^{\log_b a})$ , então  $T(n) = \Theta(n^{\log_b a} \lg n)$ .
3. Se  $f(n) = \Omega(n^{\log_b a + \epsilon})$  para alguma constante  $\epsilon > 0$ , e se  $af(n/b) \leq cf(n)$  para alguma constante  $c < 1$  e para todo  $n$  suficientemente grande, então  $T(n) = \Theta(f(n))$ . ■

Antes de aplicar o teorema mestre a alguns exemplos, vamos passar algum tempo tentando entender o que ele significa. Em cada um dos três casos, estamos comparando a função  $f(n)$  com a função  $n^{\log_b a}$ . Intuitivamente, a solução para a recorrência é determinada pela maior das duas funções. Se, como no caso 1, a função  $n^{\log_b a}$  for a maior, então a solução será  $T(n) = \Theta(n^{\log_b a})$ . Se, como no caso 3, a função  $f(n)$  for a maior, então a solução será  $T(n) = \Theta(f(n))$ . Se, como no caso 2, as duas funções tiverem o mesmo tamanho, faremos a multiplicação por um fator logarítmico e a solução será  $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$ .

Além dessa intuição, existem alguns detalhes técnicos que devem ser entendidos. No primeiro caso,  $f(n)$  não só deve ser menor que  $n^{\log_b a}$ , mas tem de ser *polinomialmente* menor. Ou seja,  $f(n)$  deve ser assintoticamente menor que  $n^{\log_b a}$  por um fator  $n^\epsilon$  para alguma constante  $\epsilon > 0$ . No terceiro caso,  $f(n)$  não apenas deve ser maior que  $n^{\log_b a}$ ; ela tem de ser polinomialmente maior e, além disso, satisfazer à condição de “regularidade” de que  $af(n/b) \leq cf(n)$ . Essa condição é satisfeita pela maioria das funções polinomialmente limitadas que encontraremos.

É importante perceber que os três casos não abrangem todas as possibilidades para  $f(n)$ . Existe uma lacuna entre os casos 1 e 2 quando  $f(n)$  é menor que  $n^{\log_b a}$ , mas não polinomialmente menor. De modo semelhante, há uma lacuna entre os casos 2 e 3 quando  $f(n)$  é maior que  $n^{\log_b a}$ , mas não polinomialmente maior. Se a função  $f(n)$  recair em uma dessas lacunas, ou se a condição de regularidade no caso 3 deixar de ser válida, o método mestre não poderá ser usado para resolver a recorrência.

## Como usar o método mestre

Para usar o método mestre, simplesmente determinamos qual caso (se houver algum) do teorema mestre se aplica e anotamos a resposta.

Como primeiro exemplo, considere

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

Para essa recorrência, temos  $a = 9$ ,  $b = 3$ ,  $f(n) = n$  e, portanto, temos  $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$ . Como  $f(n) = O(n^{\log_3 9 - \epsilon})$ , onde  $\epsilon = 1$ , podemos aplicar o caso 1 do teorema mestre e concluir que a solução é  $T(n) = \Theta(n^2)$ .

Agora, considere

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

na qual  $a = 1$ ,  $b = 3/2$ ,  $f(n) = 1$  e  $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$ . Aplica-se o caso 2, pois  $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$ , e portanto a solução para a recorrência é  $T(n) = \Theta(\lg n)$ .

Para a recorrência

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

temos  $a = 3$ ,  $b = 4$ ,  $f(n) = n \lg n$  e  $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$ . Como  $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ , onde  $\epsilon \approx 0.2$ , aplica-se o caso 3 se podemos mostrar que a condição de regularidade é válida para  $f(n)$ . Para  $n$  suficientemente grande,  $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$  para  $c = 3/4$ . Consequentemente, de acordo com o caso 3, a solução para a recorrência é  $T(n) = \Theta(n \lg n)$ .

O método mestre não se aplica à recorrência

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

mesmo que ela tenha a forma adequada:  $a = 2$ ,  $b = 2$ ,  $f(n) = n \lg n$  e  $n^{\log_b a} = n$ . Parece que o caso 3 deve se aplicar, pois  $f(n) = n \lg n$  é assintoticamente maior que  $n^{\log_b a} = n$ . O problema é que ela não é *polinomialmente* maior. A razão  $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$  é assintoticamente menor que  $n^\epsilon$  para qualquer constante positiva  $\epsilon$ . Consequentemente, a recorrência cai na lacuna entre o caso 2 e o caso 3. (Veja no Exercício 4.4-2 uma solução.)

## Exercícios

### 4.3-1

Use o método mestre para fornecer limites assintóticos restritos para as recorrências a seguir.

a.  $T(n) = 4T(n/2) + n$ .

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

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

### 4.3-2

O tempo de execução de um algoritmo  $A$  é descrito pela recorrência  $T(n) = 7T(n/2) + n^2$ . Um algoritmo concorrente  $A'$  tem um tempo de execução  $T'(n) = \alpha T'(n/4) + n^2$ . Qual é o maior valor inteiro para  $\alpha$  tal que  $A'$  seja assintoticamente mais rápido que  $A$ ?

### 4.3-3

Use o método mestre para mostrar que a solução para a recorrência de pesquisa binária  $T(n) = T(n/2) + \Theta(1)$  é  $T(n) = \Theta(\lg n)$ . (Veja no Exercício 2.3-5 uma descrição da pesquisa binária.)

### 4.3-4

O método mestre pode ser aplicado à recorrência  $T(n) = 4T(n/2) + n^2 \lg n$ ? Por que ou por que não? Forneça um limite superior assintótico para essa recorrência.

### 4.3-5 \*

Considere a condição de regularidade  $af(n/b) \leq cf(n)$  para alguma constante  $c < 1$ , que faz parte do caso 3 do teorema mestre. Dê um exemplo de uma função simples  $f(n)$  que satisfaça a todas as condições no caso 3 do teorema mestre, exceto à condição de regularidade.

## ★ 4.4 Prova do teorema mestre

Esta seção contém uma prova do teorema mestre (Teorema 4.1). A prova não precisa ser entendida para se aplicar o teorema.

A prova tem duas partes. A primeira parte analisa a recorrência “mestre” (4.5), sob a hipótese simplificadora de que  $T(n)$  é definida apenas sobre potências exatas de  $b > 1$ ; ou seja, para  $n = 1, b, b^2, \dots$ . Essa parte fornece toda a intuição necessária para se entender por que o teorema mestre é verdadeiro. A segunda parte mostra como a análise pode ser estendida a todos os inteiros positivos  $n$  e é apenas a técnica matemática aplicada ao problema do tratamento de pisos e tetos.

Nesta seção, algumas vezes abusaremos ligeiramente de nossa notação assintótica, usando-a para descrever o comportamento de funções que só são definidas sobre potências exatas de  $b$ . Lembre-se de que as definições de notações assintóticas exigem que os limites sejam provados para todos os números grandes o suficiente, não apenas para aqueles que são potências de  $b$ . Tendo em vista que poderíamos produzir novas notações assintóticas que se aplicassem ao conjunto  $\{b^i : i = 0, 1, \dots\}$  em lugar dos inteiros negativos, esse abuso é secundário.

Apesar disso, sempre deveremos nos resguardar quando estivermos usando a notação assintótica sobre um domínio limitado, a fim de não chegarmos a conclusões inadequadas. Por exemplo, provar que  $T(n) = O(n)$  quando  $n$  é uma potência exata de 2 não garante que  $T(n) = O(n)$ . A função  $T(n)$  poderia ser definida como

$$T(n) = \begin{cases} n & \text{se } n = 1, 2, 4, 8, \dots, \\ n^2 & \text{em caso contrário,} \end{cases}$$

e, nesse caso, o melhor limite superior que pode ser provado é  $T(n) = O(n^2)$ . Devido a esse tipo de consequência drástica, nunca empregaremos a notação assintótica sobre um domínio limitado sem tornar absolutamente claro a partir do contexto que estamos fazendo isso.

### 4.4.1 A prova para potências exatas

A primeira parte da prova do teorema mestre analisa a recorrência (4.5),

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

para o método mestre, sob a hipótese de que  $n$  é uma potência exata de  $b > 1$ , onde  $b$  não precisa ser um inteiro. A análise está dividida em três lemas. O primeiro reduz o problema de resolver a recorrência mestre ao problema de avaliar uma expressão que contém um somatório. O segundo determina limites sobre esse somatório. O terceiro lema reúne os dois primeiros para provar uma versão do teorema mestre correspondente ao caso em que  $n$  é uma potência exata de  $b$ .

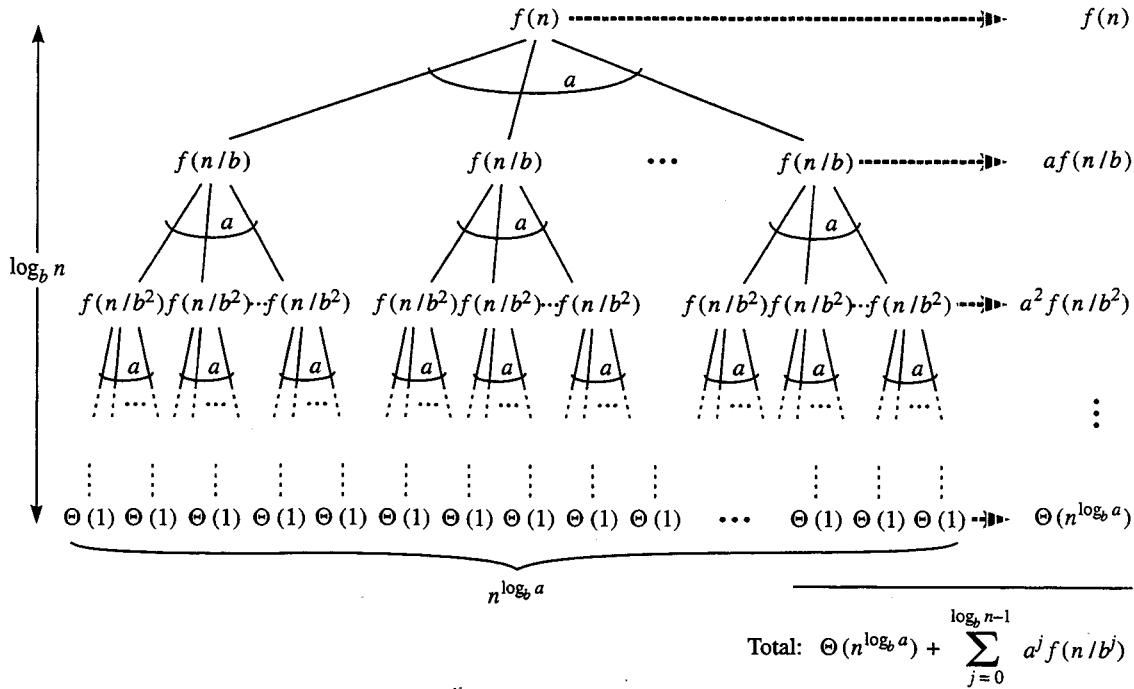


FIGURA 4.3 A árvore de recursão gerada por  $T(n) = aT(n/b) + f(n)$ . A árvore é uma árvore  $a$ -ária completa com  $n^{\log_b a}$  folhas e altura  $\log_b n$ . O custo de cada nível é mostrado à direita, e sua soma é dada na equação (4.6)

#### Lema 4.2

Sejam  $a \geq 1$  e  $b > 1$  constantes, e seja  $f(n)$  uma função não negativa definida sobre potências exatas de  $b$ . Defina  $T(n)$  sobre potências exatas de  $b$  pela recorrência

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

onde  $i$  é um inteiro positivo. Então

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

**Prova** Usamos a árvore de recursão da Figura 4.3. A raiz da árvore tem custo  $f(n)$ , e ela tem  $a$  filhas, cada uma com o custo  $f(n/b)$ . (É conveniente imaginar  $a$  como sendo um inteiro, especialmente ao se visualizar a árvore de recursão, mas a matemática não o exige.) Cada uma dessas filhas tem  $a$  filhas com o custo  $f(n/b^2)$  e, desse modo, existem  $a^2$  nós que estão à distância 2 da raiz. Em geral, há  $a^j$  nós à distância  $j$  da raiz, e cada um tem o custo  $f(n/b^j)$ . O custo de cada folha é  $T(1) = \Theta(1)$ , e cada folha está a uma distância  $\log_b n$  da raiz, pois  $n/b^{\log_b n} = 1$ . Existem  $a^{\log_b n} = n^{\log_b a}$  folhas na árvore.

Podemos obter a equação (4.6) fazendo o somatório dos custos de cada nível da árvore, como mostra a figura. O custo para um nível  $j$  de nós internos é  $a^j f(n/b^j)$ , e assim o total de todos os níveis de nós internos é

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

No algoritmo básico de dividir e conquistar, essa soma representa os custos de dividir problemas em subproblemas, e depois combinar novamente os subproblemas. O custo de todas as folhas, que é o custo de efetuar todos os  $n^{\log_b a}$  subproblemas de tamanho 1, é  $\Theta(n^{\log_b a})$ . ■

Em termos da árvore de recursão, os três casos do teorema mestre correspondem a casos nos quais o custo total da árvore é (1) dominado pelos custos nas folhas, (2) distribuído uniformemente entre os níveis da árvore ou (3) dominado pelo custo da raiz.

O somatório na equação (4.6) descreve o custo dos passos de divisão e combinação no algoritmo básico de dividir e conquistar. O lema seguinte fornece limites assintóticos sobre o crescimento do somatório.

### Lema 4.3

Sejam  $a \geq 1$  e  $b > 1$  constantes, e seja  $f(n)$  uma função não negativa definida sobre potências exatas de  $b$ . Uma função  $g(n)$  definida sobre potências exatas de  $b$  por

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

pode então ser limitada assintoticamente para potências exatas de  $b$  como a seguir.

1. Se  $f(n) = O(n^{\log_b a - \epsilon})$  para alguma constante  $\epsilon > 0$ , então  $g(n) = O(n^{\log_b a})$ .
2. Se  $f(n) = \Theta(n^{\log_b a})$ , então  $g(n) = \Theta(n^{\log_b a} \lg n)$ .
3. Se  $a f(n/b) \leq c f(n)$  para alguma constante  $c < 1$  e para todo  $n \geq b$ , então  $g(n) = \Theta(f(n))$ .

**Prova** Para o caso 1, temos  $f(n) = O(n^{\log_b a - \epsilon})$ , implicando que  $f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon})$ . A substituição na equação (4.7) resulta em

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

Limitamos o somatório dentro da notação  $O$  pela fatoração dos termos e simplificação, o que resulta em uma série geométrica crescente:

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

Tendo em vista que  $b$  e  $\epsilon$  são constantes, podemos reescrever a última expressão como  $n^{\log_b a - \epsilon} O(n^\epsilon) = O(n^{\log_b a})$ . Substituindo por essa expressão o somatório na equação (4.8), temos

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

e o caso 1 fica provado.

Sob a hipótese de que  $f(n) = \Theta(n^{\log_b a})$  para o caso 2, temos que  $f(n/b^j) = \Theta((n/b^j)^{\log_b a - \epsilon})$ . A substituição na equação (4.7) produz

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

Limitamos o somatório dentro de  $\Theta$  como no caso 1 mas, dessa vez, não obtemos uma série geométrica. Em vez disso, descobrimos que todo termo do somatório é o mesmo:

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

Substituindo por essa expressão o somatório da equação (4.9), obtemos

$$\begin{aligned} g(n) &= \Theta(n^{\log_b a} \log_b n) \\ &= \Theta(n^{\log_b a} \lg n), \end{aligned}$$

e caso 2 fica provado.

O caso 3 é provado de forma semelhante. Como  $f(n)$  aparece na definição (4.7) de  $g(n)$  e todas os termos de  $g(n)$  são não negativos, podemos concluir que  $g(n) = \Omega(f(n))$  para potências exatas de  $b$ . Sob a hipótese de que  $af(n/b) \leq cf(n)$  para alguma constante  $c < 1$  e todo  $n \geq b$ , temos  $f(n/b) \leq (c/a)f(n)$ . Iteragindo  $j$  vezes, temos  $f(n/b^j) \leq (c/a)^j f(n)$  ou, de modo equivalente,  $a^j f(n/b^j) \leq c^j f(n)$ . A substituição na equação (4.7) e a simplificação produzem uma série geométrica mas, diferente da série no caso 1, esta tem termos decrescentes:

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

pois  $c$  é constante. Desse modo, podemos concluir que  $g(n) = \Theta(f(n))$  para potências exatas de  $b$ . O caso 3 fica provado, o que completa a prova do lema. ■

Agora podemos provar uma versão do teorema mestre para o caso em que  $n$  é uma potência exata de  $b$ .

**Lema 4.4**

Sejam  $a \geq 1$  e  $b > 1$  constantes, e seja  $f(n)$  uma função não negativa definida sobre potências exatas de  $b$ . Defina  $T(n)$  sobre potências exatas de  $b$  pela recorrência

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

onde  $i$  é um inteiro positivo. Então,  $T(n)$  pode ser limitado assintoticamente para potências exatas de  $b$  como a seguir.

1. Se  $f(n) = O(n^{\log_b a - \epsilon})$  para alguma constante  $\epsilon > 0$ , então  $T(n) = \Theta(n^{\log_b a})$ .
2. Se  $f(n) = \Theta(n^{\log_b a})$ , então  $T(n) = \Theta(n^{\log_b a} \lg n)$ .
3. Se  $f(n) = \Omega(n^{\log_b a + \epsilon})$  para alguma constante  $\epsilon > 0$ , e se  $af(n/b) \leq cf(n)$  para alguma constante  $c < 1$  e todo  $n$  suficientemente grande, então  $T(n) = \Theta(f(n))$ .

**Prova** Empregamos os limites do Lema 4.3 para avaliar o somatório (4.6) a partir do Lema 4.2. Para o caso 1, temos

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

e, para o caso 2,

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

Para o caso 3,

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

porque  $f(n) = \Omega(n^{\log_b a + \epsilon})$ . ■

#### 4.4.2 Pisos e tetos

Para completar a prova do teorema mestre, devemos agora estender nossa análise à situação em que pisos e tetos são usados na recorrência mestre, de forma que a recorrência seja definida para todos os inteiros, não apenas para potências exatas de  $b$ . Obter um limite inferior sobre

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

e um limite superior sobre

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.11} |_{65}$$

é rotina, pois o limite  $\lceil n/b \rceil \geq n/b$  pode ser forçado no primeiro caso para produzir o resultado desejado, e o limite  $\lfloor n/b \rfloor \leq n/b$  pode ser forçado no segundo caso. A imposição de um limite inferior sobre a recorrência (4.11) exige quase a mesma técnica que a imposição de um limite superior sobre a recorrência (4.10); assim, apresentaremos somente esse último limite.

Modificamos a árvore de recursão da Figura 4.3 para produzir a árvore de recursão da Figura 4.4. À medida que nos aprofundamos na árvore de recursão, obtemos uma seqüência de invocações recursivas sobre os argumentos

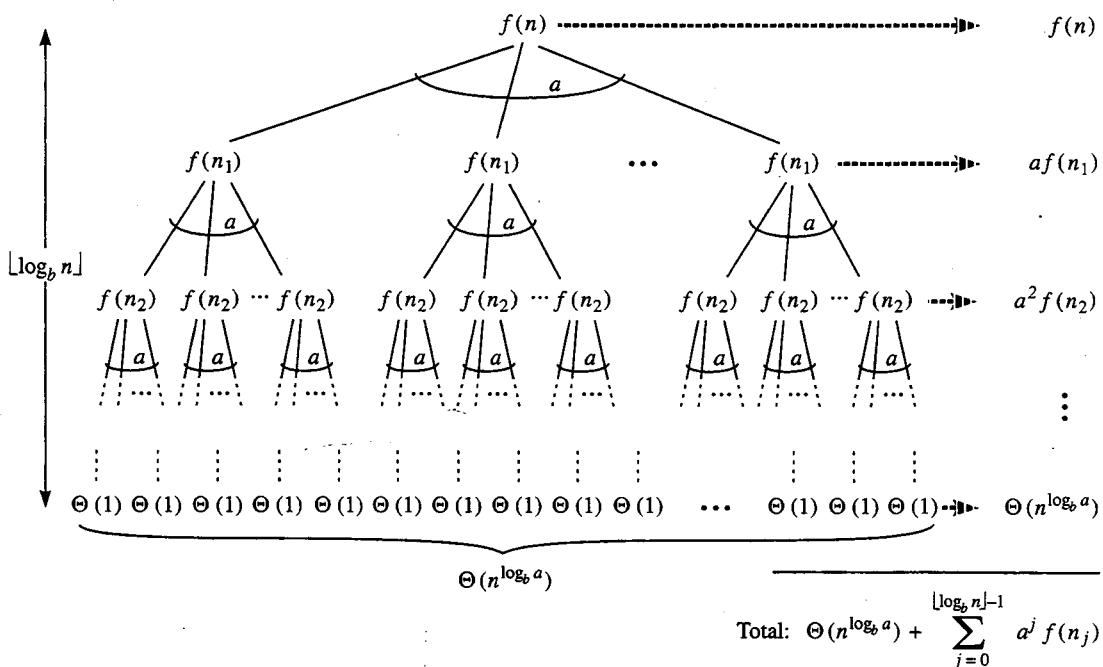
$$\begin{aligned} & n, \\ & \lceil n/b \rceil, \\ & \lceil \lceil n/b \rceil / b \rceil, \\ & \lceil \lceil \lceil n/b \rceil / b \rceil / b \rceil, \\ & \vdots \end{aligned}$$


FIGURA 4.4 A árvore de recursão gerada por  $T(n) = aT(\lceil n/b \rceil) + f(n)$ . O argumento recursivo  $n_j$  é dado pela equação (4.12)

Vamos denotar o  $j$ -ésimo elemento na seqüência por  $n_j$ , onde

$$n_j = \begin{cases} n & \text{se } j = 0, \\ \lceil n_{j-1}/b \rceil & \text{se } j > 0. \end{cases} \quad (4.12)$$

Nossa primeira meta é determinar a profundidade  $k$  tal que  $n_k$  é uma constante. Usando a desigualdade  $\lceil x \rceil \leq x + 1$ , obtemos

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

Em geral,

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

Fazendo  $j = \lfloor \log_b n \rfloor$ , obtemos

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

e desse modo vemos que, à profundidade  $\lfloor \log_b n \rfloor$ , o tamanho do problema é no máximo uma constante.

Da Figura 4.4, observamos que

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

que é quase igual à equação (4.6), a não ser pelo fato de  $n$  ser um inteiro arbitrário e não se restringir a ser uma potência exata de  $b$ .

Agora podemos avaliar o somatório

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

a partir de (4.13) de maneira análoga à prova do Lema 4.3. Começando com o caso 3, se  $af(\lceil n/b \rceil) \leq cf(n)$  para  $n > b + b/(b-1)$ , onde  $c < 1$  é uma constante, segue-se que  $a^j f(n_j) \leq c^j f(n)$ . Portanto, a soma na equação (4.14) pode ser avaliada da mesma maneira que no Lema 4.3. Para o caso 2, temos  $f(n) = \Theta(n^{\log_b a})$ . Se pudermos mostrar que  $f(n_j) = O(n^{\log_b a} / a^j) = O((n/b^j)^{\log_b a})$ , então a prova para caso 2 do Lema 4.3 passará. Observe que  $j \leq \lfloor \log_b n \rfloor$  implica  $b^j/n \leq 1$ . O limite  $f(n) = O(n^{\log_b a})$  implica que existe uma constante  $c > 0$  tal que, para  $n_j$  suficientemente grande,

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

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

pois  $c(1 + b / (b - 1))^{\log_b a}$  é uma constante. Portanto, o caso 2 fica provado. A prova do caso 1 é quase idêntica. A chave é provar o limite  $f(n_j) = O(n^{\log_b a - \epsilon})$ , semelhante à prova correspondente do caso 2, embora a álgebra seja mais complicada.

Agora provamos os limites superiores no teorema mestre para todos os inteiros  $n$ . A prova dos limites inferiores é semelhante.

## Exercícios

### 4.4-1 \*

Forneça uma expressão simples e exata para  $n_i$  na equação (4.12) para o caso em que  $b$  é um inteiro positivo, em vez de um número real arbitrário.

### 4.4-2 \*

Mostre que, se  $f(n) = \Theta(n^{\log_b a} \lg^k n)$ , onde  $k \geq 0$ , a recorrência mestre tem a solução  $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$ . Por simplicidade, restrinja sua análise a potências exatas de  $b$ .

### 4.4-3 \*

Mostre que o caso 3 do teorema mestre é exagerado, no sentido de que a condição de regularidade  $af(n/b) \leq cf(n)$  para alguma constante  $c < 1$  implica que existe uma constante  $\epsilon > 0$  tal que  $f(n) = \Omega(n^{\log_b a + \epsilon})$ .

## Problemas

### 4-1 Exemplos de recorrência

Forneça limites assintóticos superiores e inferiores para  $T(n)$  em cada uma das recorrências a seguir. Suponha que  $T(n)$  seja constante para  $n \leq 2$ . Torne seus limites tão restritos quanto possível e justifique suas respostas.

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

### 4-2 Como localizar o inteiro perdido

Um arranjo  $A[1 .. n]$  contém todos os inteiros de 0 a  $n$ , exceto um. Seria fácil descobrir o inteiro que falta no tempo  $O(n)$ , usando um arranjo auxiliar  $B[0 .. n]$  para registrar os números que aparecem em  $A$ . Porém, neste problema, não podemos ter acesso a todo um inteiro em  $A$  com uma única operação. Os elementos de  $A$  são representados em binário, e a única operação que pode-

mos usar para obter acesso a eles é “buscar o  $j$ -ésimo bit de  $A[i]$ ”, que demora um tempo constante.

Mostre que, se usarmos apenas essa operação, ainda poderemos descobrir o inteiro que falta no tempo  $O(n)$ .

#### 4-3 Custos de passagem de parâmetros

Em todo este livro, partimos da hipótese de que a passagem de parâmetros durante as chamadas de procedimentos demora um tempo constante, mesmo que um arranjo de  $N$  elementos esteja sendo passado. Essa hipótese é válida na maioria dos sistemas, porque é passado um ponteiro para o arranjo, e não o próprio arranjo. Este problema examina as implicações de três estratégias de passagem de parâmetros:

1. Um arranjo é passado por ponteiro. Tempo =  $\Theta(1)$ .
2. Um arranjo é passado por cópia. Tempo =  $\Theta(N)$ , onde  $N$  é o tamanho do arranjo.
3. Um arranjo é passado por cópia somente do subintervalo ao qual o procedimento chamado poderia ter acesso. Tempo =  $\Theta(p - q + 1)$  se o subarranjo  $A[p .. q]$  for passado.
  - a. Considere o algoritmo de pesquisa binária recursiva para localizar um número em um arranjo ordenado (ver Exercício 2.3-5). Forneça recorrências para os tempos de execução do pior caso de pesquisa binária quando os arranjos são passados com o uso de cada um dos três métodos anteriores, e forneça bons limites superiores nas soluções das recorrências. Seja  $N$  o tamanho do problema original e  $n$  o tamanho de um subproblema.
  - b. Repita a parte (a) para o algoritmo MERGE-SORT da Seção 2.3.1.

#### 4-4 Outros exemplos de recorrência

Forneça limites assintóticos superiores e inferiores para  $T(n)$  em cada uma das recorrências a seguir. Suponha que  $T(n)$  seja constante para  $n$  suficientemente pequeno. Torne seus limites tão restritos quanto possível e justifique suas respostas.

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

#### 4-5 Números de Fibonacci

Este problema desenvolve propriedades dos números de Fibonacci, que são definidos pela recorrência (3.21). Usaremos a técnica de gerar funções para resolver a recorrência de Fibonacci. Defina a *função geradora* (ou *série de potências formais*)  $\mathcal{F}$  como]

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} F_i z^i$$

$$= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \dots,$$

onde  $F_i$  é o  $i$ -ésimo número de Fibonacci.

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

b. Mostre que

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

onde

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

e

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

c. Mostre que

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

d. Prove que  $F_i = \phi^i/\sqrt{5}$  para  $i > 0$ , arredondado para o inteiro mais próximo. (Sugestão: Observe que  $|\hat{\phi}| < 1$ .)

e. Prove que  $F_{i+2} \geq \phi^i$  para  $i \geq 0$ .

#### 4-6 Testes de chips VLSI

O professor Diógenes tem  $n$  chips VLSI<sup>1</sup> supostamente idênticos que, em princípio, são capazes de testar uns aos outros. O aparelho de teste do professor acomoda dois chips de cada vez. Quando o aparelho está carregado, cada chip testa o outro e informa se ele está bom ou ruim. Um chip bom sempre informa com precisão se o outro chip está bom ou ruim, mas a resposta de um chip ruim não é confiável. Portanto, os quatro resultados possíveis de um teste são:

Chip A informa	Chip B informa	Conclusão
B está bom	A está bom	Ambos estão bons ou ambos ruins
B está bom	A está ruim	Pelo menos um está ruim
B está ruim	A está bom	Pelo menos um está ruim
B está ruim	A está ruim	Pelo menos um está ruim

<sup>1</sup>VLSI significa *very large scale integration*, ou integração em escala muito grande, que é a tecnologia de chips de circuitos integrados usada para fabricar a maioria dos microprocessadores de hoje.

- a. Mostre que, se mais de  $n/2$  chips estão ruins, o professor não pode necessariamente descobrir quais chips estão bons usando qualquer estratégia baseada nessa espécie de teste aos pares. Suponha que os chips ruins possam conspirar para enganar o professor.
- b. Considere o problema de descobrir um único chip bom entre  $n$  chips, supondo que mais de  $n/2$  dos chips estejam bons. Mostre que  $\lfloor n/2 \rfloor$  testes de pares sejam suficientes para reduzir o problema a outro com praticamente metade do tamanho.
- c. Mostre que os chips bons podem ser identificados com  $\Theta(n)$  testes de pares, supondo-se que mais de  $n/2$  dos chips estejam bons. Forneça e resolva a recorrência que descreve o número de testes.

#### 4-7 Arranjos de Monge

Um arranjo  $m \times n$  de números reais é um **arranjo de Monge** se, para todo  $i, j, k$  e  $l$  tais que  $1 \leq i < k \leq m$  e  $1 \leq j < l \leq n$ , temos

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

Em outras palavras, sempre que escolhemos duas linhas e duas colunas de um arranjo de Monge e consideramos os quatro elementos nas interseções das linhas e das colunas, a soma dos elementos superior esquerdo e inferior direito é menor ou igual à soma dos elementos inferior esquerdo e superior direito. Por exemplo, o arranjo a seguir é um arranjo de Monge:

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

- a. Prove que um arranjo é de Monge se e somente se para todo  $i = 1, 2, \dots, m-1$  e  $j = 1, 2, \dots, n-1$ , temos

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

(*Sugestão:* Para a parte “somente se”, use a indução separadamente sobre linhas e colunas.)

- b. O arranjo a seguir não é de Monge. Troque um elemento para transformá-lo em um arranjo de Monge. (*Sugestão:* Use a parte (a).)

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

- c. Seja  $f(i)$  o índice da coluna que contém o elemento mínimo mais à esquerda da linha  $i$ . Prove que  $f(1) \leq f(2) \leq \dots \leq f(m)$  para qualquer arranjo de Monge  $m \times n$ .
- d. Aqui está uma descrição de um algoritmo de dividir e conquistar que calcula o elemento mínimo mais à esquerda em cada linha de um arranjo de Monge  $m \times n$ :

Construa uma submatriz  $A'$  de  $A$  consistindo nas linhas de numeração par de  $A$ . Determine recursivamente o mínimo mais à esquerda para cada linha de  $A'$ . Em seguida, calcule o mínimo mais à esquerda nas linhas de numeração ímpar de  $A$ .

Explique como calcular o mínimo mais à esquerda nas linhas de numeração ímpar de  $A$  ( dado que o mínimo mais à esquerda das linhas de numeração seja conhecido) no tempo  $O(m + n)$ .

- e. Escreva a recorrência que descreve o tempo de execução do algoritmo descrito na parte (d). Mostre que sua solução é  $O(m + n \log m)$ .

## Notas do capítulo

As recorrências foram estudadas desde 1202 aproximadamente, por L. Fibonacci, e em sua homenagem os números de Fibonacci receberam essa denominação. A. De Moivre introduziu o método de funções geradoras (ver Problema 4.5) para resolver recorrências. O método mestre foi adaptado de Bentley, Haken e Saxe [41], que fornecem o método estendido justificado pelo Exercício 4.4-2. Knuth [182] e Liu [205] mostram como resolver recorrências lineares usando o método de funções geradoras. Purdom e Brown [252] e Graham, Knuth e Patashnik [132] contêm discussões extensas da resolução de recorrências.

Vários pesquisadores, inclusive Akra e Bazzi [13], Roura [262] e Verma [306], forneceram métodos para resolução de recorrências de dividir e conquistar mais gerais do que as que são resolvidas pelo método mestre. Descrevemos aqui o resultado de Akra e Bazzi, que funciona para recorrências da forma

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

onde  $k \geq 1$ ; todos os coeficientes  $a_i$  são positivos e somam pelo menos 1; todos os valores  $b_i$  são maiores que 1;  $f(n)$  é limitada, positiva e não decrescente; e, para todas as constantes  $c > 1$ , existem constantes  $n_0, d > 0$  tais que  $f(n/c) \geq df(n)$  para todo  $n \geq n_0$ . Esse método funcionaria sobre uma recorrência como  $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$ , para a qual o método mestre não se aplica. Para resolver a recorrência (4.15), primeiro encontramos o valor de  $p$  tal que  $\sum_{i=1}^k a_i b_i^{-p} = 1$ . (Tal  $p$  sempre existe, ele é único e positivo.) A solução para a recorrência é então

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

para uma constante  $n'$  suficientemente grande. O método de Akra-Bazzi pode ser um tanto difícil de usar, mas ele serve para resolver recorrências que modelam a divisão do problema em subproblemas de tamanhos substancialmente desiguais. O método mestre é mais simples para usar, mas só se aplica quando os tamanhos dos subproblemas são iguais.

---

## *Capítulo 5*

# *Análise probabilística e algoritmos aleatórios*

Este capítulo introduz a análise probabilística e os algoritmos aleatórios. Se estiver pouco familiarizado com os fundamentos da teoria das probabilidades, leia o Apêndice C, que apresenta uma revisão desse assunto. A análise probabilística e os algoritmos aleatórios serão revistos várias vezes ao longo deste livro.

### **5.1 O problema da contratação**

Suponha que você precise contratar um novo auxiliar de escritório. Suas tentativas anteriores de contratação foram malsucedidas, e você decide usar uma agência de empregos. A agência de empregos lhe enviará um candidato por dia. Você entrevistará a pessoa e então decidirá se deve contratá-la ou não. Você terá de pagar à agência de empregos uma pequena taxa para entrevistar um candidato. Porém, a contratação real de um candidato é mais onerosa, pois você deve despedir seu auxiliar de escritório atual e pagar uma grande taxa de contratação à agência de empregos. Seu compromisso é ter, a cada momento, a melhor pessoa possível para realizar o serviço. Assim, você decide que, depois de entrevistar cada candidato, se esse candidato for mais bem qualificado que o auxiliar de escritório atual, o auxiliar de escritório atual será despedido e o novo candidato será contratado. Você está disposto a pagar o preço resultante dessa estratégia, mas deseja avaliar qual será esse preço.

O procedimento HIRE-ASSISTANT, dado a seguir, expressa em pseudocódigo essa estratégia para contratação. Ele pressupõe que os candidatos ao emprego de auxiliar de escritório são numerados de 1 a  $n$ . O procedimento presume que você pode, depois de entrevistar o candidato  $i$ , determinar se o candidato  $i$  é o melhor candidato que viu até agora. Para inicializar, o procedimento cria um candidato fictício, com o número 0, menos qualificado que cada um dos outros candidatos.

HIRE-ASSISTANT ( $n$ )

```
1 melhor  $\leftarrow$  0       $\triangleright$  candidato 0 é um candidato fictício menos qualificado
2 for  $i \leftarrow 1$  to  $n$ 
3   do entrevistar candidato  $i$ 
4     if candidato  $i$  é melhor que candidato melhor
5       then melhor  $\leftarrow i$ 
6       contratar candidato  $i$ 
```

O modelo de custo para esse problema difere do modelo descrito no Capítulo 2. Não estamos preocupados com o tempo de execução de HIRE-ASSISTANT, mas sim com o custo referente a entrevistar e contratar. Na superfície, a análise do custo desse algoritmo pode parecer muito diferente de analisar o tempo de execução, digamos, da ordenação por intercalação. Porém, as técnicas analíticas usadas são idênticas, quer estejamos analisando o custo ou o tempo de execução. Em um ou outro caso, estamos contando o número de vezes que certas operações básicas são executadas.

Entrevistar tem um custo baixo, digamos  $c_i$ , enquanto contratar é dispendioso, custando  $c_b$ . Seja  $m$  o número de pessoas contratadas. Então, o custo total associado a esse algoritmo é  $O(nc_i + mc_b)$ . Independente de quantas pessoas contratamos, sempre entrevistamos  $n$  candidatos e, portanto, sempre incorremos no custo  $nc_i$  associado a entrevistar. Assim, vamos nos concentrar na análise de  $mc_b$ , o custo de contratação. Essa quantidade varia com cada execução do algoritmo.

Esse cenário serve como um modelo para um paradigma computacional comum. Com frequência, precisamos encontrar o valor máximo ou mínimo em uma seqüência, examinando cada elemento da seqüência e mantendo um “vencedor” atual. O problema da contratação modela a forma como atualizamos freqüentemente nossa noção de qual elemento está vencendo no momento.

## Análise do pior caso

No pior caso, realmente contratamos cada candidato que entrevistamos. Essa situação ocorre se os candidatos chegam em ordem crescente de qualidade, e nesse caso contratamos  $n$  vezes, com o custo total de contratação  $O(nc_b)$ .

Contudo, talvez seja razoável esperar que os candidatos nem sempre cheguem em ordem crescente de qualidade. De fato, não temos nenhuma idéia sobre a ordem em que eles chegam, nem temos qualquer controle sobre essa ordem. Portanto, é natural perguntar o que esperamos que aconteça em um caso típico ou médio.

## Análise probabilística

A *análise probabilística* é o uso da probabilidade na análise de problemas. De modo mais comum, usamos a análise probabilística para analisar o tempo de execução de um algoritmo. Às vezes, nós a usamos para analisar outras quantidades, como o custo da contratação no procedimento HIRE-ASSISTANT. Para executar uma análise probabilística, devemos usar o conhecimento da distribuição das entradas ou fazer suposições sobre ela. Em seguida, analisamos nosso algoritmo, calculando um tempo de execução esperado. A expectativa é tomada sobre a distribuição das entradas possíveis. Desse modo estamos, na verdade, calculando a média do tempo de execução sobre todas as entradas possíveis.

Devemos ser muito cuidadosos ao decidir sobre a distribuição das entradas. Para alguns problemas, é razoável supor algo sobre o conjunto de todas as entradas possíveis, e podemos usar a análise probabilística como uma técnica para projetar um algoritmo eficiente e como um meio de obter informações para resolver um problema. Em outros problemas, não podemos descrever uma distribuição de entradas razoável e, nesses casos, não podemos utilizar a análise probabilística.

No caso do problema da contratação, podemos pressupor que os candidatos chegam em uma ordem aleatória. O que isso significa para esse problema? Supomos que é possível separar dois candidatos quaisquer e decidir qual é o candidato mais bem qualificado; isto é, existe uma ordem total sobre os candidatos. (Consulte o Apêndice B para ver a definição de uma ordem total.) Portanto, podemos ordenar cada candidato com um número exclusivo de 1 a  $n$ , usando *ordenação( $i$ )* para denotar a ordenação do candidato  $i$ , e adotar a convenção de que uma ordenação mais alta corresponde a um candidato mais bem qualificado. A lista ordenada  $\langle \text{ordenação}(1), \text{ordenação}(2), \dots, \text{ordenação}(n) \rangle$  é uma permutação da lista  $\langle 1, 2, \dots, n \rangle$ . Dizer que os candidatos chegam em uma ordem aleatória é equivalente a dizer que essa lista de ordenações

tem igual probabilidade de ser qualquer uma das  $n!$  permutações dos números 1 a  $n$ . Como alternativa, dizemos que as ordenações formam uma **permutação aleatória uniforme**; ou seja, cada uma das  $n!$  permutações possíveis aparece com igual probabilidade.

A Seção 5.2 contém uma análise probabilística do problema da contratação.

## Algoritmos aleatórios

Para utilizar a análise probabilística, precisamos saber algo a respeito da distribuição sobre as entradas. Em muitos casos, sabemos bem pouco sobre a distribuição das entradas. Mesmo se soubermos algo sobre a distribuição, talvez não possamos modelar esse conhecimento em termos computacionais. Ainda assim, freqüentemente podemos usar a probabilidade e o caráter aleatório como uma ferramenta para projeto e análise de algoritmos, tornando aleatório o comportamento de parte do algoritmo.

No problema da contratação, talvez pareça que os candidatos estão sendo apresentados em ordem aleatória, mas não temos nenhum meio de saber se isso está ou não ocorrendo. Desse modo, para desenvolver um algoritmo aleatório para o problema da contratação, devemos ter maior controle sobre a ordem em que entrevistamos os candidatos. Assim, vamos alterar um pouco o modelo. Diremos que a agência de empregos tem  $n$  candidatos, e ela nos envia uma lista dos candidatos com antecedência. A cada dia, escolhemos ao acaso qual candidato entrevistar. Embora não conheçamos nada sobre os candidatos (além de seus nomes), fizemos uma mudança significativa. Em vez de contar com a suposição de que os candidatos virão em ordem aleatória, obtivemos o controle do processo e impusemos uma ordem aleatória.

De modo mais geral, dizemos que um algoritmo é **aleatório** se o seu comportamento é determinado não apenas por sua entrada, mas também por valores produzidos por um **gerador de números aleatórios**. Vamos supor que temos à nossa disposição um gerador de números aleatórios RANDOM. Uma chamada a  $\text{RANDOM}(a, b)$  retorna um inteiro entre  $a$  e  $b$  inclusive, sendo cada inteiro igualmente provável. Por exemplo,  $\text{RANDOM}(0, 1)$  produz 0 com probabilidade  $1/2$  e produz 1 com probabilidade  $1/2$ . Uma chamada a  $\text{RANDOM}(3, 7)$  retorna 3, 4, 5, 6 ou 7, cada um com probabilidade  $1/5$ . Cada inteiro retornado por RANDOM é independente dos inteiros retornados em chamadas anteriores. Imagine RANDOM como se fosse o lançamento de um dado de  $(b - a + 1)$  lados para obter sua saída. (Na prática, a maioria dos ambientes de programação oferece um **gerador de números pseudo-aleatórios**: um algoritmo determinístico retornando números que “parecem” estatisticamente aleatórios.)

## Exercícios

### 5.1-1

Mostre que a hipótese de que sempre somos capazes de descobrir qual candidato é o melhor na linha 4 do procedimento HIRE-ASSISTANT implica que conhecemos uma ordem total sobre as ordenações dos candidatos.

### 5.1-2 \*

Descreva uma implementação do procedimento  $\text{RANDOM}(a, b)$  que só faça chamadas a  $\text{RANDOM}(0, 1)$ . Qual é o tempo de execução esperado de seu procedimento, como uma função de  $a$  e  $b$ ?

### 5.1-3 \*

Suponha que você deseja dar saída a 0 com probabilidade  $1/2$  e a 1 com probabilidade  $1/2$ . Há um procedimento BIASED-RANDOM à sua disposição que dá saída a 0 ou 1. A saída é 1 com alguma probabilidade  $p$  e 0 com probabilidade  $1 - p$ , onde  $0 < p < 1$ , mas você não sabe qual é o valor de  $p$ . Forneça um algoritmo que utilize BIASED-RANDOM como uma sub-rotina e forneça uma resposta imparcial, retornando 0 com probabilidade  $1/2$  e 1 com probabilidade  $1/2$ . Qual é o tempo de execução esperado de seu algoritmo como uma função de  $p$ ?

## 5.2 Indicadores de variáveis aleatórias

Para analisar muitos algoritmos, inclusive o problema da contratação, usaremos indicadores de variáveis aleatórias. Os indicadores de variáveis aleatórias fornecem um método conveniente para conversão entre probabilidades e expectativas. Vamos supor que temos um espaço amostral  $S$  e um evento  $A$ . Então, o *indicador de variável aleatória*  $I\{A\}$  associado ao evento  $A$  é definido como

$$I\{A\} = \begin{cases} 1 & \text{se } A \text{ ocorre ,} \\ 0 & \text{se } A \text{ não ocorre .} \end{cases} \quad (5.1)$$

Como um exemplo simples, vamos determinar o número esperado de caras que obtemos quando lançamos uma moeda comum. Nossa espaço amostral é  $S = \{H, T\}$ , e definimos uma variável aleatória  $Y$  que assume os valores  $H$  e  $T$ , cada um com probabilidade  $1/2$ . Podemos então definir um indicador de variável aleatória  $X_H$ , associado ao resultado cara no lançamento da moeda, o que podemos expressar como o evento  $Y = H$ . Essa variável conta o número de caras obtidas nesse lançamento, e tem o valor 1 se a moeda mostra cara e 0 em caso contrário. Escrevemos

$$X_H = I\{Y = H\} = \begin{cases} 1 & \text{se } Y = H , \\ 0 & \text{se } Y = T . \end{cases}$$

O número esperado de caras obtidas em um lançamento da moeda é simplesmente o valor esperado de nosso indicador de variável  $X_H$ :

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

Desse modo, o número esperado de caras obtidas por um lançamento de uma moeda comum é  $1/2$ . Como mostra o lema a seguir, o valor esperado de um indicador de variável aleatória associado a um evento  $A$  é igual à probabilidade de  $A$  ocorrer.

### **Lema 5.1**

Dado um espaço amostral  $S$  e um evento  $A$  no espaço amostral  $S$ , seja  $X_A = I\{A\}$ . Então,  $E\{X_A\} = \Pr\{A\}$ .

**Prova** Pela definição de indicador de variável aleatória da equação (5.1) e pela definição de valor esperado, temos

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

onde  $\bar{A}$  denota  $S - A$ , o complemento de  $A$ . ■

Embora os indicadores de variáveis aleatórias possam parecer incômodos para uma aplicação como a contagem do número esperado de caras no lançamento de uma única moeda, eles

são úteis para analisar situações em que realizamos testes aleatórios repetidos. Por exemplo, indicadores de variáveis aleatórias nos dão um caminho simples para chegar ao resultado da equação (C.36). Nessa equação, calculamos o número de caras em  $n$  lançamentos de moedas, considerando separadamente a probabilidade de obter 0 cara, 1 cara, 2 caras etc. Contudo, o método mais simples proposto na equação (C.37) na realidade utiliza implicitamente indicadores de variáveis aleatórias. Tornando esse argumento mais explícito, podemos fazer de  $X_i$  o indicador de variável aleatória associado ao evento no qual o  $i$ -ésimo lançamento mostra cara. Sendo  $Y_i$  a variável aleatória que denota o resultado do  $i$ -ésimo lançamento, temos que  $X_i = I\{Y_i = H\}$ . Seja  $X$  a variável aleatória que denota o número total de caras em  $n$  lançamentos de moedas; assim,

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

Desejamos calcular o número esperado de caras; para isso, tomamos a expectativa de ambos os lados da equação anterior para obter

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

O lado esquerdo da equação anterior é a expectativa da soma de  $n$  variáveis aleatórias. Pelo Lema 5.1, podemos calcular facilmente a expectativa de cada uma das variáveis aleatórias. Pela equação (C.20) – linearidade de expectativa – é fácil calcular a expectativa da soma: ela é igual à soma das expectativas das  $n$  variáveis aleatórias.

A linearidade de expectativa torna o uso de indicadores de variáveis aleatórias uma técnica analítica poderosa; ela se aplica até mesmo quando existe dependência entre as variáveis aleatórias. Agora podemos calcular com facilidade o número esperado de caras:

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

Desse modo, em comparação com o método empregado na equação (C.36), os indicadores de variáveis aleatórias simplificam muito o cálculo. Utilizaremos indicadores de variáveis aleatórias em todo este livro.

### Análise do problema da contratação com o uso de indicadores de variáveis aleatórias

Voltando ao problema da contratação, agora desejamos calcular o número esperado de vezes que contratamos um novo auxiliar de escritório. Com a finalidade de usar uma análise probabilística, supomos que os candidatos chegam em uma ordem aleatória, como discutimos na seção anterior. (Veremos na Seção 5.3 como remover essa suposição.) Seja  $X$  a variável aleatória cujo valor é igual ao número de vezes que contratamos um novo auxiliar de escritório. Poderíamos então aplicar a definição de valor esperado da equação (C.19) para obter

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

mas esse cálculo seria incômodo. Em vez disso, utilizaremos indicadores de variáveis aleatórias para simplificar bastante o cálculo.

Para usar indicadores de variáveis aleatórias, em lugar de calcular  $E[X]$  definindo uma única variável associada ao número de vezes que contratamos um novo auxiliar de escritório, definimos  $n$  variáveis relacionadas ao fato de cada candidato específico ser ou não contratado. Em particular, fazemos de  $X_i$  o indicador de variável aleatória associado ao evento em que o  $i$ -ésimo candidato é contratado. Desse modo,

$$X_i = I\{\text{candidato } i \text{ é contratado}\} = \begin{cases} 1 & \text{se o candidato } i \text{ é contratado,} \\ 0 & \text{se o candidato } i \text{ não é contratado.} \end{cases} \quad (5.2)$$

e

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

Pelo Lema 5.1, temos que

$$E[X_i] = \Pr\{\text{candidato } i \text{ é contratado}\},$$

e devemos então calcular a probabilidade de que as linhas 5 e 6 de HIRE-ASSISTANT sejam executadas.

O candidato  $i$  é contratado, na linha 5, exatamente quando o candidato  $i$  é melhor que cada um dos candidatos 1 a  $i - 1$ . Como presumimos que os candidatos chegam em uma ordem aleatória, os primeiros  $i$  candidatos apareceram em uma ordem aleatória. Qualquer um desses  $i$  primeiros candidatos tem igual probabilidade de ser o mais bem qualificado até o momento. O candidato  $i$  tem uma probabilidade  $1/i$  de ser mais bem qualificado que os candidatos 1 a  $i - 1$  e, desse modo, uma probabilidade  $1/i$  de ser contratado. Pela Lema 5.1, concluímos que

$$E[X_i] = 1/i. \quad (5.4)$$

Agora podemos calcular  $E[X]$ :

$$E[X] = E\left[\sum_{i=1}^n X_i\right] \quad (\text{pela equação (5.3)}) \quad (5.5)$$

$$= \sum_{i=1}^n E[X_i] \quad (\text{pela linearidade de expectativa})$$

$$= \sum_{i=1}^n 1/i \quad (\text{pela equação (5.4)})$$

$$78 \quad = \ln n + O(1) \quad (\text{pela equação (A.7)}). \quad (5.6)$$

Apesar de entrevistarmos  $n$  pessoas, na realidade só contratamos aproximadamente  $\ln n$  delas, em média. Resumimos esse resultado no lema a seguir.

### Lema 5.2

Supondo que os candidatos sejam apresentados em uma ordem aleatória, o algoritmo HIRE-ASSISTANT tem um custo total de contratação  $O(c_b \ln n)$ .

**Prova** O limite decorre imediatamente de nossa definição do custo de contratação e da equação (5.6).

O custo esperado de contratação é uma melhoria significativa sobre o custo de contratação do pior caso,  $O(n c_b)$ .

## Exercícios

### 5.2-1

Em HIRE-ASSISTANT, supondo que os candidatos sejam apresentados em uma ordem aleatória, qual é a probabilidade de você contratar exatamente uma vez? Qual é a probabilidade de você contratar exatamente  $n$  vezes?

### 5.2-2

Em HIRE-ASSISTANT, supondo que os candidatos sejam apresentados em uma ordem aleatória, qual é a probabilidade de você contratar exatamente duas vezes?

### 5.2-3

Use indicadores de variáveis aleatórias para calcular o valor esperado da soma de  $n$  dados.

### 5.2-4

Use indicadores de variáveis aleatórias para resolver o problema a seguir, conhecido como o **problema da chapelaria**. Cada um entre  $n$  clientes entrega um chapéu ao funcionário da chapelaria em um restaurante. O funcionário devolve os chapéus aos clientes em ordem aleatória. Qual é o número esperado de clientes que recebem de volta seus próprios chapéus?

### 5.2-5

Seja  $A[1 .. n]$  um arranjo de  $n$  números distintos. Se  $i < j$  e  $A[i] > A[j]$ , então o par  $(i, j)$  é chamado uma **inversão** de  $A$ . (Veja no Problema 2-4 mais informações sobre inversões.) Suponha que cada elemento de  $A$  seja escolhido ao acaso, independentemente e de maneira uniforme no intervalo de 1 a  $n$ . Use indicadores de variáveis aleatórias para calcular o número esperado de inversões.

## 5.3 Algoritmos aleatórios

Na seção anterior, mostramos como o conhecimento de uma distribuição sobre as entradas pode nos ajudar a analisar o comportamento de um algoritmo no caso médio. Muitas vezes, não temos tal conhecimento, e nenhuma análise de caso médio é possível. Como mencionamos na Seção 5.1, talvez possamos usar um algoritmo aleatório.

No caso de um problema como o problema da contratação, no qual é útil supor que todas as permutações da entrada são igualmente prováveis, uma análise probabilística orientará o desenvolvimento de um algoritmo aleatório. Em vez de pressupor uma distribuição de entradas, imponemos uma distribuição. Em particular, antes de executar o algoritmo, permutamos ao acaso os candidatos, a fim de impor a propriedade de que cada permutação é igualmente provável. Essa modificação não altera nossa expectativa de contratar um novo auxiliar de escritório aproximadamente  $\ln n$  vezes. Contudo, ela significa que, para *qualquer* entrada, esperamos que seja esse o caso, em vez de entradas obtidas a partir de uma distribuição particular.

Agora, exploramos um pouco mais a distinção entre a análise probabilística e os algoritmos aleatórios. Na Seção 5.2 afirmamos que, supondo-se que os candidatos se apresentem em uma ordem aleatória, o número esperado de vezes que contratamos um novo auxiliar de escritório é cerca de  $\ln n$ . Observe que o algoritmo é determinístico nesse caso; para qualquer entrada particular, o número de vezes que um novo auxiliar de escritório é contratado sempre será o mesmo. Além disso, o número de vezes que contratamos um novo auxiliar de escritório difere para entradas distintas e depende das ordenações dos diversos candidatos. Tendo em vista que esse número depende apenas das ordenações dos candidatos, podemos representar uma entrada particular listando, em ordem, as ordenações dos candidatos, ou seja,  $\langle \text{ordenação}(1), \text{ordenação}(2), \dots, \text{ordenação}(n) \rangle$ . Dada lista de ordenação  $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$ , um novo auxiliar de escritório sempre será contratado 10 vezes, pois cada candidato sucessivo é melhor que o anterior, e as linhas 5 e 6 serão executadas em cada iteração do algoritmo. Dada a lista de ordenações  $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 \rangle$ , um novo auxiliar de escritório será contratado apenas uma vez, na primeira iteração. Dada uma lista de ordenações  $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6 \rangle$ , um novo auxiliar de escritório será contratado três vezes, após a entrevista com os candidatos de ordenações 5, 8 e 10. Lembrando que o custo de nosso algoritmo depende de quantas vezes contratamos um novo auxiliar de escritório, vemos que existem entradas dispendiosas, como  $A_1$ , entradas econômicas, como  $A_2$  e entradas moderadamente dispendiosas, como  $A_3$ .

Por outro lado, considere o algoritmo aleatório que primeiro permuta os candidatos, e depois determina o melhor candidato. Nesse caso, a aleatoriedade está no algoritmo, e não na distribuição de entradas. Dada uma entrada específica, digamos a entrada  $A_3$  anterior, não podemos dizer quantas vezes o máximo será atualizado, porque essa quantidade é diferente a cada execução do algoritmo. A primeira vez em que executamos o algoritmo sobre  $A_3$ , ele pode produzir a permutação  $A_1$  e executar 10 atualizações enquanto, na segunda vez em que executamos o algoritmo, podemos produzir a permutação  $A_2$  e executar apenas uma atualização. Na terceira vez em que o executamos, podemos produzir algum outro número de atualizações. A cada vez que executamos o algoritmo, a execução depende das escolhas aleatórias feitas e ela provavelmente irá diferir da execução anterior do algoritmo. Para esse algoritmo e muitos outros algoritmos aleatórios, *nenhuma entrada específica induz seu comportamento no pior caso*. Nem mesmo pior inimigo poderá produzir um arranjo de entrada ruim, pois a permutação aleatória torna irrelevante a ordem de entrada. O algoritmo aleatório só funcional mal se o gerador de números aleatórios produzir uma permutação “sem sorte”.

No caso do problema da contratação, a única alteração necessária no código tem a finalidade de permitir o arranjo ao acaso.

#### RANDOMIZED-HIRE-ASSISTANT( $n$ )

```

1  permutar aleatoriamente a lista de candidatos
2  melhor  $\leftarrow 0$        $\triangleright$  candidato 0 é um candidato fictício menos qualificado
3  for  $i \leftarrow 1$  to  $n$ 
4    do entrevistar candidato  $i$ 
5    if candidato  $i$  é melhor que candidato melhor
6      then melhor  $\leftarrow i$ 
7      contratar candidato  $i$ 
```

Com essa mudança simples, criamos um algoritmo aleatório cujo desempenho corresponde ao que obtivemos supondo que os candidatos se apresentavam em uma ordem aleatória.

#### **Lema 5.3**

O custo esperado de contratação do procedimento RANDOMIZED-HIRE-ASSISTANT é  $O(c_b \ln n)$ .

**Prova** Depois de permitir o arranjo de entrada, chegamos a uma situação idêntica à da análise probabilística de HIRE-ASSISTANT.

A comparação entre os Lemas 5.2 e 5.3 evidencia a diferença entre a análise probabilística e os algoritmos aleatórios. No Lema 5.2, fizemos uma suposição sobre a entrada. No Lema 5.3, não fizemos tal suposição, embora a aleatoriedade da entrada demore algum tempo extra. No restante desta seção, discutiremos algumas questões relacionadas com a permutação aleatória das entradas.

## Permutação aleatória de arranjos

Muitos algoritmos aleatórios fazem a aleatoriedade da entrada permutando o arranjo de entrada dado. (Existem outras maneiras de usar a aleatoriedade.) Aqui, descreveremos dois métodos para esse fim. Supomos que temos um arranjo  $A$  que, sem perda de generalidade, contém os elementos 1 a  $n$ . Nossa meta é produzir uma permutação aleatória do arranjo.

Um método comum é atribuir a cada elemento  $A[i]$  do arranjo uma prioridade aleatória  $P[i]$ , e depois ordenar os elementos de  $A$  de acordo com essas prioridades. Por exemplo, se nosso arranjo inicial fosse  $A = \langle 1, 2, 3, 4 \rangle$  e escolhêssemos prioridades aleatórias  $P = \langle 36, 3, 97, 19 \rangle$ , produziríamos um arranjo  $B = \langle 2, 4, 1, 3 \rangle$ , pois a segunda prioridade é a menor, seguida pela quarta, depois pela primeira e finalmente pela terceira. Denominamos esse procedimento PERMUTE-BY-SORTING:

**PERMUTE-BY-SORTING( $A$ )**

```

1  $n \leftarrow \text{comprimento}[A]$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   do  $P[i] = \text{RANDOM}(1, n^3)$ 
4 ordenar  $A$ , usando  $P$  como chaves de ordenação
5 return  $A$ 
```

A linha 3 escolhe um número aleatório entre 1 e  $n^3$ . Usamos um intervalo de 1 a  $n^3$  para tornar provável que todas as prioridades em  $P$  sejam exclusivas.

(O Exercício 5.3-5 lhe pede para provar que a probabilidade de todas as entradas serem exclusivas é pelo menos  $1 - 1/n$ , e o Exercício 5.3-6 pergunta como implementar o algoritmo ainda que duas ou mais prioridades sejam idênticas.) Vamos supor que todas as prioridades sejam exclusivas.

A etapa demorada nesse procedimento é a ordenação na linha 4. Como veremos no Capítulo 8, se usarmos uma ordenação por comparação, a ordenação demorará o tempo  $\Omega(n \lg n)$ . Podemos alcançar esse limite inferior, pois vimos que a ordenação por intercalação demora o tempo  $\Theta(n \lg n)$ . (Veremos na Parte II outras ordenações por comparação que tomam o tempo  $\Theta(n \lg n)$ .) Depois da ordenação, se  $P[i]$  for a  $j$ -ésima menor prioridade, então  $A[i]$  estará na posição  $j$  da saída. Dessa maneira, obteremos uma permutação. Resta provar que o procedimento produz uma **permutação aleatória uniforme**, isto é, que toda permutação dos números de 1 a  $n$  tem igual probabilidade de ser produzida.

### Lema 5.4

O procedimento PERMUTE-BY-SORTING produz uma permutação aleatória uniforme da entrada, supondo que todas as prioridades são distintas.

**Prova** Começamos considerando a permutação particular em que cada elemento  $A[i]$  recebe a  $i$ -ésima menor prioridade. Mostraremos que essa permutação ocorre com probabilidade exatamente igual a  $1/n!$ . Para  $i = 1, 2, \dots, n$ , seja  $X_i$  o evento em que o elemento  $A[i]$  recebe a  $i$ -ésimo menor prioridade. Então, desejamos calcular a probabilidade de que, para todo  $i$ , o evento  $X_i$  ocorra, que é

$$\Pr \{X_1 \cap X_2 \cap X_3 \cap \dots \cap X_{n-1} \cap X_n\}.$$

Usando o Exercício C.2-6, essa probabilidade é igual a

$$\Pr \{X_1\} \cdot \Pr \{X_2 | X_1\} \cdot \Pr \{X_3 | X_2 \cap X_1\} \cdot \Pr \{X_4 | X_3 \cap X_2 \cap X_1\} \\ \cdots \Pr \{X_i | X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} \cdots \Pr \{X_n | X_{n-1} \cap \cdots \cap X_1\}.$$

Temos que  $\Pr \{X_1\} = 1/n$  porque essa é a probabilidade de que uma prioridade escolhida ao acaso em um conjunto de  $n$  seja a menor. Em seguida, observamos que  $\Pr \{X_2 | X_1\} = 1/(n-1)$  porque, dado que o elemento  $A[1]$  tem a menor prioridade, cada um dos  $n-1$  elementos restantes apresenta uma chance igual de ter a segunda menor prioridade. Em geral, para  $i = 2, 3, \dots, n$ , temos que  $\Pr \{X_i | X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} = 1/(n-i+1)$  pois, dado que os elementos  $A[1]$  até  $A[i-1]$  têm as  $i-1$  menores prioridades (em ordem), cada um dos  $n-(i-1)$  elementos restantes apresenta uma chance igual de ter a  $i$ -ésima menor prioridade. Desse modo, temos

$$\Pr \{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} = \left(\frac{1}{n}\right) \left(\frac{1}{n-1}\right) \cdots \left(\frac{1}{2}\right) \left(\frac{1}{1}\right) \\ = \frac{1}{n!},$$

e mostramos que a probabilidade de obtermos a permutação de identidade é  $1/n!$ .

Podemos estender essa prova a qualquer permutação de prioridades. Considere qualquer permutação fixa  $\sigma = \langle \sigma(1), \sigma(2), \dots, \sigma(n) \rangle$  do conjunto  $\{1, 2, \dots, n\}$ . Vamos denotar por  $r_i$  a ordenação da prioridade atribuída ao elemento  $A[i]$ , onde o elemento com a  $j$ -ésima menor prioridade tem a ordenação  $j$ . Se definirmos  $X_i$  como o evento em que o elemento  $A[i]$  recebe a  $\sigma(i)$ -ésima menor prioridade, ou  $r_i = \sigma(i)$ , a mesma prova ainda se aplicará. Então, se calcularmos a probabilidade de obter qualquer permutação específica, o cálculo será idêntico ao anterior, de forma que a probabilidade de se obter essa permutação também será  $1/n!$ .

Poderíamos imaginar que, para provar que uma permutação é uma permutação aleatória uniforme é suficiente mostrar que, para cada elemento  $A[i]$ , a probabilidade de que ele termine na posição  $j$  é  $1/n$ . O Exercício 5.3-4 mostra que essa condição mais fraca é, de fato, insuficiente.

Um método melhor para gerar uma permutação aleatória é permutar o arranjo dado no local. O procedimento RANDOMIZE-IN-PLACE faz isso no tempo  $O(n)$ . Na iteração  $i$ , o elemento  $A[i]$  é escolhido ao acaso entre os elementos  $A[i]$  a  $A[n]$ . Depois da iteração  $i$ ,  $A[i]$  nunca é alterado.

```
RANDOMIZE-IN-PLACE( $A$ )
1  $n \leftarrow \text{comprimento}[A]$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   do trocar  $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$ 
```

Usaremos um loop invariante para mostrar que o procedimento RANDOMIZE-IN-PLACE produz uma permutação aleatória uniforme. Dado um conjunto de  $n$  elementos, uma permutação de  $k$  é uma seqüência contendo  $k$  dos  $n$  elementos. (Consulte o Apêndice B.) Existem  $n!/(n-k)!$  dessas permutações de  $k$  possíveis.

### Lema 5.5

O procedimento RANDOMIZE-IN-PLACE calcula uma permutação aleatória uniforme.

**Prova** Usamos o seguinte loop invariante:

Imediatamente antes da  $i$ -ésima iteração do loop **for** das linhas 2 e 3, para cada permutação de  $(i-1)$  possível, o subarranjo  $A[1..i-1]$  contém essa permutação de  $(i-1)$  com probabilidade  $(n-i+1)!/n!$ .

Precisamos mostrar que esse invariante é verdadeiro antes da primeira iteração do loop, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

**Inicialização:** Considere a situação imediatamente antes da primeira iteração do loop, de forma que  $i = 1$ . O loop invariante nos diz que, para cada permutação de 0 possível, o subarranjo  $A[1..0]$  contém essa permutação de 0 com probabilidade  $(n-i+1)!/n! = n!/n! = 1$ . O subarranjo  $A[1..0]$  é um subarranjo vazio e uma permutação de 0 não tem nenhum elemento. Desse modo,  $A[1..0]$  contém qualquer permutação de 0 com probabilidade 1, e o loop invariante é válido antes da primeira iteração.

**Manutenção:** Supomos que, imediatamente antes da  $(i-1)$ -ésima iteração, cada permutação de  $(i-1)$  possível aparece no subarranjo  $A[1..i-1]$  com probabilidade  $(n-i+1)!/n!$ , e mostraremos que, após a  $i$ -ésima iteração, cada permutação de  $i$  possível aparece no subarranjo  $A[1..i]$  com probabilidade  $(n-i)!/n!$ . Incrementar  $i$  para a próxima iteração manterá então o loop invariante.

Vamos examinar a  $i$ -ésima iteração. Considere uma permutação de  $i$  específica e denote os elementos que ela contém por  $\langle x_1, x_2, \dots, x_i \rangle$ . Essa permutação consiste em uma permutação de  $(i-1) \langle x_1, \dots, x_{i-1} \rangle$  seguida pelo valor  $x_i$  que o algoritmo insere em  $A[i]$ . Seja  $E_1$  o evento em que as primeiras  $i-1$  iterações criaram a permutação de  $(i-1) \langle x_1, \dots, x_{i-1} \rangle$  específica em  $A[1..i-1]$ . Pelo loop invariante,  $\Pr\{E_1\} = (n-i+1)!/n!$ . Seja  $E_2$  o evento em que a  $i$ -ésima iteração insere  $x_i$  na posição  $A[i]$ . A permutação de  $i \langle x_1, \dots, x_i \rangle$  é formada em  $A[1..i]$  precisamente quando tanto  $E_1$  quanto  $E_2$  ocorrem, e assim desejamos calcular  $\Pr\{E_2 \cap E_1\}$ . Usando a equação (C.14), temos

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 | E_1\} \Pr\{E_1\}.$$

A probabilidade  $\Pr\{E_2 | E_1\}$  é igual a  $1/(n-i+1)$  porque, na linha 3, o algoritmo escolhe  $x_i$  ao acaso entre os  $n-i+1$  valores nas posições  $A[i..n]$ . Desse modo, temos

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 | E_1\} \Pr\{E_1\}$$

$$\begin{aligned} &= \frac{1}{n-i+1} \cdot \frac{(n-i+1)!}{n!} \\ &= \frac{(n-i)!}{n!}. \end{aligned}$$

**Término:** No término,  $i = n+1$ , e temos que o subarranjo  $A[1..n]$  é uma permutação de  $n$  dada com probabilidade  $(n-n)!/n! = 1/n!$ .

Portanto, RANDOMIZE-IN-PLACE produz uma permutação aleatória uniforme. ■

Com freqüência, um algoritmo aleatório é a maneira mais simples e eficiente de resolver um problema. Usaremos os algoritmos aleatórios ocasionalmente em todo o livro.

## Exercícios

### 5.3-1

O professor Marceau faz objeções ao loop invariante usado na prova do Lema 5.5. Ele questiona se o loop invariante é verdadeiro antes da primeira iteração. Seu raciocínio é que seria possível | 83

quase com a mesma facilidade declarar que um subarranjo vazio não contém nenhuma permutação de 0. Assim, a probabilidade de um subarranjo vazio conter uma permutação de 0 deve ser 0, invalidando assim o loop invariante antes da primeira iteração. Reescreva o procedimento RANDOMIZE-IN-PLACE, de forma que seu loop invariante associado se aplique a um subarranjo não vazio antes da primeira iteração, e modifique a prova do Lema 5.5 de acordo com seu procedimento.

### 5.3-2

O professor Kelp decide escrever um procedimento que produzirá ao acaso qualquer permutação além da permutação de identidade. Ele propõe o seguinte procedimento:

PERMUTE-WITHOUT-IDENTITY( $A$ )

- 1  $n \leftarrow \text{comprimento}[A]$
- 2 **for**  $i \leftarrow 1$  **to**  $n$
- 3   **do trocar**  $A[i] \leftrightarrow A[\text{RANDOM}(i + 1, n)]$

Esse código faz o que professor Kelp deseja?

### 5.3-3

Suponha que, em vez de trocar o elemento  $A[i]$  por um elemento aleatório do subarranjo  $A[i .. n]$ , nós o trocamos por um elemento aleatório de qualquer lugar no arranjo:

PERMUTE-WITH-ALL( $A$ )

- 1  $n \leftarrow \text{comprimento}[A]$
- 2 **for**  $i \leftarrow 1$  **to**  $n$
- 3   **do trocar**  $A[i] \leftarrow A[\text{RANDOM}(1, n)]$

Esse código produz uma permutação aleatória uniforme? Por que ou por que não?

### 5.3-4

O professor Armstrong sugere o procedimento a seguir para gerar uma permutação aleatória uniforme:

PERMUTE-BY-CYCLIC( $A$ )

- 1  $n \leftarrow \text{comprimento}[A]$
- 2  $\text{deslocamento} \leftarrow \text{RANDOM}(1, n)$
- 3 **for**  $i \leftarrow 1$  **to**  $n$
- 4   **do dest**  $\leftarrow i + \text{deslocamento}$
- 5    **if**  $\text{dest} > n$
- 6      **then**  $\text{dest} \leftarrow \text{dest} - n$
- 7       $B[\text{dest}] \leftarrow A[i]$
- 8 **return**  $B$

Mostre que cada elemento  $A[i]$  tem uma probabilidade  $1/n$  de terminar em qualquer posição particular em  $B$ . Em seguida, mostre que o professor Armstrong está equivocado, demonstrando que a permutação resultante não é uniformemente aleatória.

### 5.3-5 \*

Prove que, no arranjo  $P$  do procedimento PERMUTE-BY-SORTING, a probabilidade de todos os elementos serem exclusivos é pelo menos  $1 - 1/n$ .

### 5.3-6

Explique como implementar o algoritmo PERMUTE-BY-SORTING para tratar o caso em que duas ou mais prioridades são idênticas. Isto é, seu algoritmo deve produzir uma permutação aleatória uniforme, ainda que duas ou mais prioridades sejam idênticas.

## ★ 5.4 Análise probabilística e usos adicionais de indicadores de variáveis aleatórias

Esta seção avançada ilustra um pouco mais a análise probabilística por meio de quatro exemplos. O primeiro determina a probabilidade de, em uma sala com  $k$  pessoas, algum par compartilhar a mesma data de aniversário. O segundo exemplo examina o lançamento aleatório de bolas em caixas. O terceiro investiga “seqüências” de caras consecutivas no lançamento de moedas. O exemplo final analisa uma variante do problema da contratação, na qual você tem de tomar decisões sem entrevistar realmente todos os candidatos.

### 5.4.1 O paradoxo do aniversário

Nosso primeiro exemplo é o *paradoxo do aniversário*. Quantas pessoas devem estar em uma sala antes de existir uma chance de 50% de duas delas terem nascido no mesmo dia do ano? A resposta é um número de pessoas surpreendentemente pequeno. O paradoxo é que esse número é de fato muito menor que o número de dias do ano, ou até menor que metade do número de dias de um ano, como veremos.

Para responder à pergunta, indexamos as pessoas na sala com os inteiros  $1, 2, \dots, k$ , onde  $k$  é o número de pessoas na sala. Ignoramos a questão dos anos bissextos e supomos que todos os anos têm  $n = 365$  dias. Para  $i = 1, 2, \dots, k$ , seja  $b_i$  o dia do ano em que recai o aniversário da pessoa  $i$ , onde  $1 \leq b_i \leq n$ . Supomos também que os aniversários estão uniformemente distribuídos ao longo dos  $n$  dias do ano, de tal forma que  $\Pr\{b_i = r\} = 1/n$  para  $i = 1, 2, \dots, k$  e  $r = 1, 2, \dots, n$ .

A probabilidade de que duas pessoas, digamos  $i$  e  $j$ , tenham datas de aniversário coincidentes depende do fato de ser independente a seleção aleatória dos aniversários. Supomos de agora em diante que os aniversários são independentes, e então a probabilidade de que o aniversário de  $i$  e o aniversário de  $j$  recaiam ambos no dia  $r$  é

$$\begin{aligned}\Pr\{b_i = r \text{ e } b_j = r\} &= \Pr\{b_i = r\} \Pr\{b_j = r\} \\ &= 1/n^2.\end{aligned}$$

Desse modo, a probabilidade de que ambos recaiam no mesmo dia é

$$\begin{aligned}\Pr\{b_i = b_j\} &= \sum_{r=1}^n \Pr\{b_i = r \text{ e } b_j = r\} \\ &= \sum_{r=1}^n (1/n^2) \\ &= 1/n.\end{aligned}\tag{5.7}$$

Mais intuitivamente, uma vez que  $b_i$  é escolhido, a probabilidade de  $b_j$  ser escolhido com o mesmo valor é  $1/n$ . Desse modo, a probabilidade de  $i$  e  $j$  terem o mesmo dia de aniversário é igual à probabilidade de o aniversário de um deles recair em um determinado dia. Porém, observe que essa coincidência depende da suposição de que os dias de aniversário são independentes.

Podemos analisar a probabilidade de pelo menos 2 entre  $k$  pessoas terem aniversários coincidentes examinando o evento complementar. A probabilidade de pelo menos dois aniversários coincidirem é 1 menos a probabilidade de todos os aniversários serem diferentes. O evento em que  $k$  pessoas têm aniversários distintos é

$$B_k = \bigcap_{i=1}^k A_i,$$

onde  $A_i$  é o evento em que o dia do aniversário da pessoa  $i$  é diferente do aniversário da pessoa  $j$  para todo  $j < i$ . Tendo em vista que podemos escrever  $B_k = A_k \cap B_{k-1}$ , obtemos da equação (C.16) a recorrência

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_k | B_{k-1}\}, \quad (5.8)$$

onde consideramos  $\Pr\{B_1\} = \Pr\{A_1\} = 1$  uma condição inicial. Em outras palavras, a probabilidade de que  $b_1, b_2, \dots, b_k$  sejam aniversários distintos é a probabilidade de  $b_1, b_2, \dots, b_{k-1}$  serem aniversários distintos vezes a probabilidade de que  $b_k \neq b_i$  para  $i = 1, 2, \dots, k-1$ , dado que  $b_1, b_2, \dots, b_{k-1}$  são distintos.

Se  $b_1, b_2, \dots, b_{k-1}$  são distintos, a probabilidade condicional de que  $b_k \neq b_i$  para  $i = 1, 2, \dots, k-1$  é  $\Pr\{A_k | B_{k-1}\} = (n-k+1)/n$ , pois, dos  $n$  dias, existem  $n-(k-1)$  que não são tomados. Aplicamos de forma iterativa a recorrência (5.8) para obter

$$\begin{aligned} \Pr\{B_k\} &= \Pr\{B_{k-1}\} \Pr\{A_k | B_{k-1}\} \\ &= \Pr\{B_{k-1}\} \Pr\{A_{k-1} | B_{k-2}\} \Pr\{A_k | B_{k-1}\} \\ &\vdots \\ &= \Pr\{B_1\} \Pr\{A_2 | B_1\} \Pr\{A_3 | B_2\} \dots \Pr\{A_k | B_{k-1}\} \\ &= 1 \cdot \left(\frac{n-1}{n}\right) \left(\frac{n-2}{n}\right) \dots \left(\frac{n-k+1}{n}\right) \\ &= 1 \cdot \left(1 - \frac{1}{n}\right) \left(1 - \frac{2}{n}\right) \dots \left(1 - \frac{k-1}{n}\right). \end{aligned}$$

A desigualdade (3.11),  $1 + x \leq e^x$ , nos fornece

$$\begin{aligned} \Pr\{B_k\} &\leq e^{-1/n} e^{-2/n} \dots e^{-(k-1)/n} \\ &= e^{-\sum_{i=1}^{k-1} i/n} \\ &= e^{-k(k-1)/2n} \\ &\leq 1/2 \end{aligned}$$

quando  $-k(k-1)/2n \leq \ln(1/2)$ . A probabilidade de que todos os  $k$  aniversários sejam distintos é no máximo  $1/2$  quando  $k(k-1) \geq 2n \ln 2$  ou, resolvendo a equação quadrática, quando  $k \geq (1 + \sqrt{1 + (8 \ln 2)n})/2$ . Para  $n = 365$ , devemos ter  $k \geq 23$ . Portanto, se pelo menos 23 pessoas estão em uma sala, a probabilidade é de pelo menos  $1/2$  de que no mínimo duas pessoas tenham a mesma data de aniversário. Em Marte, um ano tem a duração de 669 dias marcianos; então, seriam necessários 31 marcianos para conseguirmos o mesmo efeito.

## Uma análise usando indicadores de variáveis aleatórias

Podemos usar indicadores de variáveis aleatórias para fornecer uma análise mais simples, embora aproximada, do paradoxo do aniversário. Para cada par  $(i, j)$  das  $k$  pessoas na sala, vamos definir o indicador de variável aleatória  $X_{ij}$ , para  $1 \leq i < j \leq k$ , por

$$X_{ij} = I\{\text{a pessoa } i \text{ e a pessoa } j \text{ têm o mesmo dia de aniversário}\}$$

$$= \begin{cases} 1 & \text{se a pessoa } i \text{ e a pessoa } j \text{ têm o mesmo dia de aniversário,} \\ 0 & \text{em caso contrário.} \end{cases}$$

Pela equação (5.7), a probabilidade de que duas pessoas tenham aniversários coincidentes é  $1/n$  e, desse modo, pelo Lema 5.1, temos

$$\begin{aligned} E[X_{ij}] &= \Pr\{\text{a pessoa } i \text{ e a pessoa } j \text{ têm o mesmo dia aniversário}\} \\ &= 1/n. \end{aligned}$$

Sendo  $X$  a variável aleatória que conta o número de pares de indivíduos que têm a mesma data de aniversário, temos

$$X = \sum_{i=1}^k \sum_{j=i+1}^k X_{ij}.$$

Tomando as expectativas de ambos os lados e aplicando a linearidade de expectativa, obtemos

$$E[X] = E\left[\sum_{i=1}^k \sum_{j=i+1}^k X_{ij}\right]$$

$$= \sum_{i=1}^k \sum_{j=i+1}^k E[X_{ij}]$$

$$= \binom{k}{2} \frac{1}{n}$$

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

Quando  $k(k-1) \geq 2n$ , o número esperado de pares de pessoas com a mesma data de aniversário é pelo menos 1. Desse modo, se tivermos pelo menos  $\sqrt{2n} + 1$  indivíduos em uma sala, poderemos esperar que no mínimo dois deles façam aniversário no mesmo dia. Para  $n = 365$ , se  $k = 28$ , o número esperado de pares com o mesmo dia de aniversário é  $(28 \cdot 27)/(2 \cdot 365) \approx 1,0356$ . Assim, com pelo menos 28 pessoas, esperamos encontrar no mínimo um par de aniversários coincidentes. Em Marte, onde um ano corresponde a 669 dias marcianos, precisaríamos de pelo menos 38 marcianos.

A primeira análise, que usou somente probabilidades, determinou o número de pessoas necessárias para que a probabilidade de existir um par de datas de aniversário coincidentes exceda  $1/2$ , e a segunda análise, que empregou indicadores de variáveis aleatórias, determinou o número tal que a quantidade esperada de aniversários coincidentes é 1. Embora os números exatos de pessoas sejam diferentes nas duas situações, eles são assintoticamente iguais:  $\Theta(\sqrt{n})$ .

### 5.4.2 Bolas e caixas

Considere o processo de lançar aleatoriamente bolas idênticas em  $b$  caixas, com a numeração 1, 2, ...,  $b$ . Os lançamentos são independentes, e em cada lançamento a bola tem igual probabilidade de terminar em qualquer caixa. A probabilidade de uma bola lançada cair em qualquer caixa dada é  $1/b$ . Desse modo, o processo de lançamento de bolas é uma seqüência de experiências de Bernoulli (consulte o Apêndice C, Seção C.4) com uma probabilidade de sucesso  $1/b$ , onde sucesso significa que a bola cai na caixa dada. Esse modelo é particularmente útil para analisar o hash (consulte o Capítulo 11), e podemos responder a uma variedade de perguntas interessantes sobre o processo de lançamento de bolas. (O problema C-1 formula perguntas adicionais sobre bolas e caixas.)

*Quantas bolas caem em uma determinada caixa?* O número de bolas que caem em uma caixa dada segue a distribuição binomial  $b(k; n, 1/b)$ . Se  $n$  bolas são lançadas, a equação (C.36) nos informa que o número esperado de bolas que caem na caixa dada é  $n/b$ .

*Quantas bolas devem ser lançadas, em média, até uma caixa dada conter uma bola?* O número de lances até a caixa dada receber uma bola segue a distribuição geométrica com probabilidade  $1/b$  e, pela equação (C.31), o número esperado de lances até o sucesso é  $1/(1/b) = b$ .

*Quantas bolas devem ser lançadas até toda caixa conter pelo menos uma bola?* Vamos chamar um lançamento em que uma bola cai em uma caixa vazia de “acerto”. Queremos saber o número esperado  $n$  de lançamentos necessários para se conseguir  $b$  acertos.

Os acertos podem ser usados para partitionar os  $n$  lançamentos em fases. A  $i$ -ésima fase consiste nos lançamentos depois do  $(i - 1)$ -ésimo acerto até o  $i$ -ésimo acerto. A primeira fase consiste no primeiro lançamento, pois temos a garantia de um acerto quando todas as caixas estão vazias. Para cada lançamento durante a  $i$ -ésima fase, existem  $i - 1$  caixas que contêm bolas e  $b - i + 1$  caixas vazias. Desse modo, para cada lançamento na  $i$ -ésima fase, a probabilidade de se obter um acerto é  $(b - i + 1)/b$ .

Seja  $n_i$  o número de lançamentos na  $i$ -ésima fase. Portanto, o número de lançamentos exigidos para se conseguir  $b$  acertos é  $n = \sum_{i=1}^b n_i$ . Cada variável aleatória  $n_i$  tem uma distribuição geométrica com probabilidade de sucesso  $(b - i + 1)/b$  e, pela equação (C.31),

$$E[n_i] = \frac{b}{b - i + 1}.$$

Por linearidade de expectativa,

$$\begin{aligned} E[n] &= E\left[\sum_{i=1}^b n_i\right] \\ &= \sum_{i=1}^b E[n_i] \\ &= \sum_{i=1}^b \frac{b}{b - i + 1} \\ &= b \sum_{i=1}^b \frac{1}{i} \\ &= b(\ln b + O(1)). \end{aligned}$$

A última linha decorre do limite (A.7) sobre a série harmônica. Então, ocorrem aproximadamente  $b \ln b$  lançamentos antes de podermos esperar que toda caixa tenha uma bola. Esse pro-

blema também é conhecido como o *problema do colecionador de cupons*, e nos diz que uma pessoa que tenta colecionar cada um de  $b$  cupons diferentes deve adquirir aproximadamente  $b \ln b$  cupons obtidos ao acaso para ter sucesso.

### 5.4.3 Seqüências

Suponha que você lance uma moeda comum  $n$  vezes. Qual é a seqüência mais longa de caras consecutivas que você espera ver? A resposta é  $\Theta(\lg n)$ , como mostra a análise a seguir.

Primeiro, provamos que o comprimento esperado da mais longa seqüência de caras é  $O(\lg n)$ . A probabilidade de que cada lançamento de moeda seja uma cara é  $1/2$ . Seja  $A_{ik}$  o evento em que uma seqüência de caras de comprimento no mínimo  $k$  começa com o  $i$ -ésimo lançamento de moeda ou, mais precisamente, o evento em que os  $k$  lançamentos consecutivos de moedas  $i, i+1, \dots, i+k-1$  produzem somente caras, onde  $1 \leq k \leq n$  e  $1 \leq i \leq n-k+1$ . Como os lançamentos de moedas são mutuamente independentes, para qualquer evento dado  $A_{ik}$ , a probabilidade de que todos os  $k$  lançamentos sejam caras é

$$\Pr\{A_{ik}\} = 1/2^k. \quad (5.9)$$

Para  $k = 2 \lceil \lg n \rceil$ ,

$$\begin{aligned} \Pr\{A_{i,2\lceil \lg n \rceil}\} &= 1/2^{2\lceil \lg n \rceil} \\ &\leq 1/2^{2\lg n} \\ &= 1/n^2, \end{aligned}$$

e, portanto, a probabilidade de uma seqüência de caras de comprimento pelo menos igual a  $2 \lceil \lg n \rceil$  começar na posição  $i$  é bastante pequena. Há no máximo  $n - 2 \lceil \lg n \rceil + 1$  posições onde tal seqüência pode começar. A probabilidade de uma seqüência de caras de comprimento pelo menos  $2 \lceil \lg n \rceil$  começar em qualquer lugar é então

$$\begin{aligned} \Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil}\right\} &\leq \sum_{i=1}^{n-2\lceil \lg n \rceil+1} 1/n^2 \\ &< \sum_{i=1}^n 1/n^2 \\ &= 1/n, \end{aligned} \quad (5.10)$$

pois, pela desigualdade de Boole (C.18), a probabilidade de uma união de eventos é no máximo a soma das probabilidades dos eventos individuais. (Observe que a desigualdade de Boole é válida até mesmo para eventos como esses, que não são independentes.)

Agora usamos a desigualdade (5.10) para limitar o comprimento da seqüência mais longa. Para  $j = 0, 1, 2, \dots, n$ , seja  $L_j$  o evento em que a seqüência mais longa de caras tem comprimento exatamente  $j$ , e seja  $L$  o comprimento da seqüência mais longa. Pela definição de valor esperado,

$$E[L] = \sum_{j=0}^n j \Pr\{L_j\}. \quad (5.11) \quad | 89$$

Poderíamos tentar avaliar essa soma usando limites superiores sobre cada  $\Pr\{L_j\}$  semelhantes aos que foram calculados na desigualdade (5.10). Infelizmente, esse método produziria limites fracos. Porém, podemos usar alguma intuição obtida pela análise anterior para obter um bom limite. Informalmente, observamos que para nenhum termo individual no somatório da equação (5.11) ambos os fatores,  $j$  e  $\Pr\{L_j\}$ , são grandes. Por quê? Quando  $j \geq 2\lceil \lg n \rceil$ , então  $\Pr\{L_j\}$  é muito pequeno e, quando  $j < 2\lceil \lg n \rceil$ , então  $j$  é bastante pequeno. De maneira mais formal, notamos que os eventos  $L_j$  para  $j = 0, 1, \dots, n$  são disjuntos, e assim a probabilidade de uma seqüência de caras de comprimento no mínimo  $2\lceil \lg n \rceil$  começar em qualquer lugar é  $\sum_{j=2\lceil \lg n \rceil}^n \Pr\{L_j\}$ . Pela desigualdade (5.10), temos  $\sum_{j=2\lceil \lg n \rceil}^n \Pr\{L_j\} < 1/n$ . Além disso, notando que  $\sum_{j=0}^n \Pr\{L_j\} = 1$ , temos  $\sum_{j=0}^{2\lceil \lg n \rceil-1} \Pr\{L_j\} \leq 1$ . Desse modo, obtemos

$$\begin{aligned} E[L] &= \sum_{j=0}^n j \Pr\{L_j\} \\ &= \sum_{j=0}^{2\lceil \lg n \rceil-1} j \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^n j \Pr\{L_j\} \\ &= \sum_{j=0}^{2\lceil \lg n \rceil-1} (2\lceil \lg n \rceil) \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^n n \Pr\{L_j\} \\ &= 2\lceil \lg n \rceil \sum_{j=0}^{2\lceil \lg n \rceil-1} \Pr\{L_j\} + n \sum_{j=2\lceil \lg n \rceil}^n \Pr\{L_j\} \\ &< 2\lceil \lg n \rceil \cdot 1 + n \cdot (1/n) \\ &= O(\lg n). \end{aligned}$$

As chances de que uma seqüência de caras exceda  $r\lceil \lg n \rceil$  lançamentos diminui rapidamente com  $r$ . Para  $r \geq 1$ , a probabilidade de uma seqüência de  $r\lceil \lg n \rceil$  caras começar na posição  $i$  é

$$\begin{aligned} \Pr\{A_{i, r\lceil \lg n \rceil}\} &= 1/2^{r\lceil \lg n \rceil} \\ &\leq 1/n^r. \end{aligned}$$

Desse modo, a probabilidade de que a seqüência mais longa seja pelo menos  $r\lceil \lg n \rceil$  é no máximo igual a  $n/n^r = 1/n^{r-1}$  ou, de modo equivalente, a probabilidade de que a seqüência mais longa tenha comprimento menor que  $r\lceil \lg n \rceil$  é no mínimo  $1 - 1/n^{r-1}$ .

Como um exemplo, para  $n = 1000$  lançamentos de moedas, a probabilidade de haver uma seqüência de pelo menos  $2\lceil \lg n \rceil = 20$  caras é no máximo  $1/n = 1/1000$ . As chances de haver uma seqüência mais longa que  $3\lceil \lg n \rceil = 30$  caras é no máximo  $1/n^2 = 1/1.000.000$ .

Agora, vamos provar um limite complementar inferior: o comprimento esperado da seqüência mais longa de caras em  $n$  lançamentos de moedas é  $\Omega(\lg n)$ . Para provar esse limite, procuramos por seqüências de comprimento  $s$  particionando os  $n$  lançamentos em aproximadamente  $n/s$  grupos de  $s$  lançamentos cada. Se escolhermos  $s = \lfloor (\lg n)/2 \rfloor$ , poderemos mostrar que é provável que pelo menos um desses grupos mostre apenas caras e, consequentemente, é provável que a seqüência mais longa tenha comprimento pelo menos igual a  $s = \Omega(\lg n)$ . Mostraremos então que a seqüência mais longa tem comprimento esperado  $\Omega(\lg n)$ .

Particionamos os  $n$  lançamentos de moedas em pelo menos  $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$  grupos de  $\lfloor (\lg n)/2 \rfloor$  lançamentos sucessivos, e limitamos a probabilidade de não surgir nenhum grupo formado apenas por caras. Pela equação (5.9), a probabilidade do grupo que começa na posição  $i$  mostrar apenas caras é

$$\Pr\{A_{i,\lfloor(\lg n)/2\rfloor}\} = 1/2^{\lfloor(\lg n)/2\rfloor} \geq 1/\sqrt{n}.$$

A probabilidade de uma seqüência de caras de comprimento pelo menos igual a  $\lfloor(\lg n)/2\rfloor$  não começar na posição  $i$  é então no máximo  $1 - 1/\sqrt{n}$ . Tendo em vista que os  $\lfloor n/\lfloor(\lg n)/2\rfloor \rfloor$  grupos são formados a partir de lançamentos de moedas mutuamente exclusivos e independentes, a probabilidade de que cada um desses grupos *deixe* de ser uma seqüência de comprimento  $\lfloor(\lg n)/2\rfloor$  é no máximo

$$(1 - 1/\sqrt{n})^{\lfloor n/\lfloor(\lg n)/2\rfloor \rfloor} \leq (1 - 1/\sqrt{n})^{\lfloor n/\lfloor(\lg n)/2\rfloor \rfloor - 1}$$

$$= (1 - 1/\sqrt{n})^{2n/\lg n - 1}$$

$$= e^{-(2n/\lg n - 1)/\sqrt{n}}$$

$$= O(e^{-\lg n})$$

$$= O(1/n).$$

Para esse argumento, usamos a desigualdade (3.11),  $1 + x \leq e^x$  e o fato, que talvez você deseje verificar, de que  $(2n/\lg n - 1)/\sqrt{n} \geq \lg n$  para  $n$  suficientemente grande.

Desse modo, a probabilidade de que a seqüência mais longa exceda  $\lfloor(\lg n)/2\rfloor$  é

$$\sum_{j=\lfloor(\lg n)/2\rfloor+1}^n \Pr\{L_j\} \geq 1 - O(1/n). \quad (5.12)$$

Agora podemos calcular um limite inferior sobre o comprimento esperado da seqüência mais longa, começando com a equação (5.11) e procedendo de forma semelhante à nossa análise do limite superior:

$$\begin{aligned} E[L] &= \sum_{j=0}^n j \Pr\{L_j\} \\ &= \sum_{j=0}^{\lceil(\lg n)/2\rceil} j \Pr\{L_j\} + \sum_{j=\lceil(\lg n)/2\rceil+1}^n j \Pr\{L_j\} \\ &= \sum_{j=0}^{\lceil(\lg n)/2\rceil} 0 \cdot \Pr\{L_j\} + \sum_{j=\lceil(\lg n)/2\rceil+1}^n \lfloor(\lg n)/2\rfloor \Pr\{L_j\} \\ &= 0 \cdot \sum_{j=0}^{\lceil(\lg n)/2\rceil} \Pr\{L_j\} + \lfloor(\lg n)/2\rfloor \sum_{j=\lceil(\lg n)/2\rceil+1}^n \Pr\{L_j\} \\ &\geq 0 + \lfloor(\lg n)/2\rfloor (1 - O(1/n)) \quad (\text{pela desigualdade (5.12)}) \\ &= \Omega(\lg n). \end{aligned}$$

Como ocorre no caso do paradoxo do aniversário, podemos obter uma análise mais simples, embora aproximada, usando indicadores de variáveis aleatórias. Seja  $X_{ik} = I\{A_{ik}\}$  o indicador de variável aleatória associado a uma seqüência de caras de comprimento no mínimo  $k$  que começam com o  $i$ -ésimo lançamento de moeda. Para contar o número total de tais seqüências, definimos

$$X = \sum_{i=1}^{n-k+1} X_{ik}.$$

Tomando as expectativas e usando a linearidade de expectativa, temos

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right]$$

$$= \sum_{i=1}^{n-k+1} \mathbb{E}[X_{ik}]$$

$$= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\}$$

$$= \sum_{i=1}^{n-k+1} 1/2^k$$

$$\frac{n-k+1}{2^k}.$$

Conectando diversos valores para  $k$ , podemos calcular o número esperado de seqüências de comprimento  $k$ . Se esse número é grande (muito maior que 1), então espera-se que ocorram muitas seqüências de comprimento  $k$ , e a probabilidade de ocorrer uma é alta. Se esse número é pequeno (muito menor que 1), então espera-se que ocorram muito poucas seqüências de comprimento  $k$ , e a probabilidade de ocorrer uma é baixa. Se  $k = c \lg n$ , para alguma constante positiva  $c$ , obtemos

$$\begin{aligned} \mathbb{E}[X] &= \frac{n - c \lg n + 1}{2^{c \lg n}} \\ &= \frac{n - c \lg n + 1}{n^c} \\ &= \frac{1}{n^{c-1}} - \frac{(c \lg n - 1)/n}{n^{c-1}} \\ &= \Theta(1/n^{c-1}). \end{aligned}$$

Se  $c$  é grande, o número esperado de seqüências de comprimento  $c \lg n$  é muito pequeno, e concluímos que é improvável que elas ocorram. Por outro lado, se  $c < 1/2$ , então obtemos  $\mathbb{E}[X] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$ , e esperamos que exista um número grande de seqüências de comprimento  $(1/2) \lg n$ . Portanto, é muito provável que ocorra uma seqüência de tal comprimento. Sómente a partir dessas estimativas grosseiras, podemos concluir que o comprimento esperado da seqüência mais longa é  $\Theta(\lg n)$ .

#### 5.4.4 O problema da contratação on-line

Como um exemplo final, examinaremos uma variante do problema da contratação. Suponha agora que não desejamos entrevistar todos os candidatos a fim de encontrar o melhor. Também não desejamos contratar e despedir à medida que encontrarmos candidatos cada vez melhores. Em vez disso, estamos dispostos a aceitar um candidato próximo do melhor, em troca de contratar exatamente uma vez. Devemos obedecer a um requisito da empresa: depois de cada entrevista, devemos oferecer imediatamente o cargo ao candidato ou dizer a ele que não será possível contratá-lo. Qual é o compromisso entre minimizar a quantidade de entrevistas e maximizar a qualidade do candidato contratado?

Podemos modelar esse problema da maneira ilustrada a seguir. Após encontrar um candidato, somos capazes de dar a cada um deles uma pontuação; seja  $pontuação(i)$  a pontuação dada ao  $i$ -ésimo candidato, e suponha que não existam dois candidatos que recebam a mesma pontuação. Depois de ver  $j$  candidatos, sabemos qual dos  $j$  candidatos tem a pontuação mais alta, mas não sabemos se qualquer dos  $n - j$  candidatos restantes terá uma pontuação mais alta. Decidimos adotar a estratégia de selecionar um inteiro positivo  $k < n$ , entrevistando e depois rejeitando os primeiros  $k$  candidatos, e contratando daí em diante o primeiro candidato que tem uma pontuação mais alta que todos os candidatos anteriores. Se notarmos que o candidato mais bem qualificado se encontrava entre os  $k$  primeiros entrevistados, então contrataremos o  $n$ -ésimo candidato. Essa estratégia é formalizada no procedimento ON-LINE-MAXIMUM( $k, n$ ), que aparece a seguir. O procedimento ON-LINE-MAXIMUM retorna o índice do candidato que desejamos contratar.

```

ON-LINE-MAXIMUM( $k, n$ )
1 melhorpontuação  $\leftarrow -\infty$ 
2 for  $i \leftarrow 1$  to  $k$ 
3   do if pontuação( $i$ )  $>$  melhorpontuação
4     then melhorpontuação  $\leftarrow$  pontuação( $i$ )
5 for  $i \leftarrow k + 1$  to  $n$ 
6   do if pontuação( $i$ )  $>$  melhorpontuação
7     then return  $i$ 
8 return  $n$ 
```

Desejamos determinar, para cada valor possível de  $k$ , a probabilidade de contratarmos o candidato mais bem qualificado. Em seguida, escolheremos o melhor possível  $k$ , e implementaremos a estratégia com esse valor. Por enquanto, suponha que  $k$  seja fixo. Seja  $M(j) = \max_{1 \leq i \leq j} \{pontuação(i)\}$  a pontuação máxima entre os candidatos 1 a  $j$ . Seja  $S$  o evento em que temos sucesso na escolha do candidato mais bem qualificado, e seja  $S_i$  o evento em que temos sucesso quando o candidato mais bem qualificado for o  $i$ -ésimo entrevistado. Tendo em vista que diversos  $S_i$  são disjuntos, temos que  $\Pr\{S\} = \sum_{i=1}^n \Pr\{S_i\}$ . Observando que nunca temos sucesso quando o candidato mais bem qualificado é um dos  $k$  primeiros, temos que  $\Pr\{S_i\} = 0$  para  $i = 1, 2, \dots, k$ . Desse modo, obtemos

$$\Pr\{S\} = \sum_{i=k+1}^n \Pr\{S_i\}. \quad (5.13)$$

Agora calculamos  $\Pr\{S_i\}$ . Para se ter sucesso quando o candidato mais bem qualificado é o  $i$ -ésimo, duas coisas devem acontecer. Primeiro, o candidato mais bem qualificado deve estar na posição  $i$ , um evento que denotamos por  $B_i$ . Segundo, o algoritmo não deve selecionar quaisquer dos candidatos nas posições  $k + 1$  a  $i - 1$ , o que acontece somente se, para cada  $j$  tal que  $k + 1 \leq j \leq i - 1$ , encontramos  $pontuação(j) < melhorpontuação$  na linha 6. (Como as pontuações são exclusivas, podemos ignorar a possibilidade de  $pontuação(j) = melhorpontuação$ .) Em ou-

tras palavras, deve ser o caso de que todos os valores  $pontuação(k+1)$  até  $pontuação(i-1)$  são menores que  $M(k)$ ; se quaisquer deles forem maiores que  $M(k)$ , retornaremos em vez disso o índice do primeiro que for maior. Usamos  $O_i$  para denotar o evento em que nenhum dos candidatos nas posições  $k+1$  a  $i-1$  é escolhido. Felizmente, os dois eventos  $B_i$  e  $O_i$  são independentes. O evento  $O_i$  depende apenas da ordenação relativa dos valores nas posições 1 a  $i-1$ , enquanto  $B_i$  depende apenas do fato de o valor na posição  $i$  SER maior que todos os valores de 1 até  $i-1$ . A ordenação das posições 1 a  $i-1$  não afeta o fato de  $i$  ser maior que todas elas, e o valor de  $i$  não afeta a ordenação das posições 1 a  $i-1$ . Desse modo, podemos aplicar a equação (C.15) para obter

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\} \Pr\{O_i\}.$$

A probabilidade  $\Pr\{B_i\}$  é claramente  $1/n$ , pois o máximo tem igual probabilidade de estar em qualquer uma das  $n$  posições. Para o evento  $O_i$  ocorrer, o valor máximo nas posições 1 a  $i-1$  deve estar em uma das  $k$  primeiras posições, e é igualmente provável que esteja em qualquer dessas  $i-1$  posições. Conseqüentemente,  $\Pr\{O_i\} = k/(i-1)$  e  $\Pr\{S_i\} = k/(n(i-1))$ . Usando a equação (5.13), temos

$$\begin{aligned}\Pr\{S\} &= \sum_{i=k+1}^n \Pr\{S_i\} \\ &= \sum_{i=k+1}^n \frac{k}{n(i-1)} \\ &= \frac{k}{n} \sum_{i=k+1}^n \frac{1}{i-1} \\ &= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i}.\end{aligned}$$

Fazemos a aproximação por integrais para limitar esse somatório acima e abaixo. Pelas desigualdades (A.12), temos

$$\int_k^n \frac{1}{x} dx \leq \sum_{i=k}^{n-1} \frac{1}{i} \leq \int_{k-1}^{n-1} \frac{1}{x} dx.$$

A avaliação dessas integrais definidas nos dá os limites

$$\frac{k}{n}(\ln n - \ln k) \leq \Pr\{S\} \leq \frac{k}{n}(\ln(n-1) - \ln(k-1)),$$

que fornecem um limite bastante restrito para  $\Pr\{S\}$ . Como desejamos maximizar nossa probabilidade de sucesso, vamos nos concentrar na escolha do valor de  $k$  que maximiza o limite inferior sobre  $\Pr\{S\}$ . (Além disso, a expressão do limite inferior é mais fácil de maximizar que a expressão do limite superior.) Fazendo a diferenciação da expressão  $(k/n)(\ln n - \ln k)$  com relação a  $k$ , obtemos

$$94 \left| \frac{1}{n}(\ln n - \ln k - 1).$$

Definindo essa derivada como igual a 0, vemos que o limite inferior sobre a probabilidade é maximizado quando  $\ln k = \ln n - 1 = \ln(n/e)$  ou, de modo equivalente, quando  $k = n/e$ . Desse modo, se implementarmos nossa estratégia com  $k = n/e$ , teremos sucesso na contratação do nosso candidato mais bem qualificado com probabilidade pelo menos  $1/e$ .

## Exercícios

### 5.4-1

Quantas pessoas deve haver em uma sala para que a probabilidade de que alguém tenha a mesma data de aniversário que você seja pelo menos  $1/2$ ? Quantas pessoas deve haver para que a probabilidade de que pelo menos duas pessoas façam aniversário em 7 de setembro seja maior que  $1/2$ ?

### 5.4-2

Suponha que sejam lançadas bolas em  $b$  caixas. Cada lançamento é independente, e cada bola tem a mesma probabilidade de cair em qualquer caixa. Qual é o número esperado de lançamentos de bolas antes que pelo menos uma das caixas contenha duas bolas?

### 5.4-3 \*

Para a análise do paradoxo do aniversário, é importante que os aniversários sejam mutuamente independentes, ou é suficiente a independência aos pares? Justifique sua resposta.

### 5.4-4 \*

Quantas pessoas devem ser convidadas para uma festa, a fim de tornar provável que existam *três* pessoas com a mesma data de aniversário?

### 5.4-5 \*

Qual é a probabilidade de uma cadeia de  $k$  elementos sobre um conjunto de tamanho  $n$  ser na realidade uma permutação de  $k$  elementos? De que modo essa pergunta se relaciona ao paradoxo do aniversário?

### 5.4-6 \*

Suponha que  $n$  bolas sejam lançadas em  $n$  caixas, onde cada lançamento é independente e a bola tem igual probabilidade de cair em qualquer caixa. Qual é o número esperado de caixas vazias? Qual é o número esperado de caixas com exatamente uma bola?

### 5.4-7 \*

Torne mais nítido o limite inferior sobre o comprimento da seqüência mostrando que, em  $n$  lançamentos de uma moeda comum, a probabilidade de não ocorrer nenhuma seqüência mais longa que  $\lg n - 2 \lg \lg n$  caras consecutivas é menor que  $1/n$ .

## Problemas

### 5-1 Contagem probabilística

Com um contador de  $b$  bits, só podemos contar normalmente até  $2^b - 1$ . Com a *contagem probabilística* de R. Morris, podemos contar até um valor muito maior, a expensas de alguma perda de precisão.

Seja um valor de contador  $i$  que representa uma contagem  $n_i$  para  $i = 0, 1, \dots, 2^b - 1$ , onde os  $n_i$  formam uma seqüência crescente de valores não negativos. Supomos que o valor inicial do contador é 0, representando uma contagem de  $n_0 = 0$ . A operação de INCREMENT atua de maneira probabilística sobre um contador que contém o valor  $i$ . Se  $i = 2^b - 1$ , então é relatado um erro de estouro (overflow). Caso contrário, o contador é incrementado em 1 com probabilidade  $1/(n_{i+1} - n_i)$ , e permanece inalterado com probabilidade  $1 - 1/(n_{i+1} - n_i)$ .

Se selecionarmos  $n_i = i$  para todo  $i \geq 0$ , então o contador será um contador comum. Surgirão situações mais interessantes se selecionarmos, digamos,  $n_i = 2^{i-1}$  para  $i > 0$  ou  $n_i = F_i$  (o  $i$ -ésimo número de Fibonacci – consulte a Seção 3.2).

Para este problema, suponha que  $n_{2^b-1}$  seja grande o suficiente para que a probabilidade de um erro de estouro seja desprezível.

- Mostre que o valor esperado representado pelo contador depois de  $n$  operações de INCREMENT serem executadas é exatamente  $n$ .
- A análise da variância da contagem representada pelo contador depende da seqüência dos  $n_i$ . Vamos considerar um caso simples:  $n_i = 100i$  para todo  $i \geq 0$ . Estime a variância no valor representado pelo registrador depois de  $n$  opções de INCREMENT terem sido executadas.

### 5-2 Pesquisa em um arranjo não ordenado

Este problema examina três algoritmos para pesquisar um valor  $x$  em um arranjo não ordenado  $A$  que consiste em  $n$  elementos.

Considere a estratégia aleatória a seguir: escolha um índice aleatório  $i$  em  $A$ . Se  $A[i] = x$ , então terminamos; caso contrário, continuamos a pesquisa escolhendo um novo índice aleatório em  $A$ . Continuamos a escolher índices aleatórios em  $A$  até encontrarmos um índice  $j$  tal que  $A[j] = x$  ou até verificarmos todos os elementos de  $A$ . Observe que cada escolha é feita a partir do conjunto inteiro de índices, de forma que podemos examinar um dado elemento mais de uma vez.

- Escreva pseudocódigo para um procedimento RANDOM-SEARCH para implementar a estratégia anterior. Certifique-se de que o algoritmo termina depois que todos os índices em  $A$  são escolhidos.
- Suponha que existe exatamente um índice  $i$  tal que  $A[i] = x$ . Qual é o número esperado de índices em  $A$  que devem ser escolhidos antes de  $x$  ser encontrado e RANDOM-SEARCH terminar?
- Generalizando sua solução para a parte (b), suponha que existam  $k \geq 1$  índices  $i$  tais que  $A[i] = x$ . Qual é o número esperado de índices em  $A$  que devem ser escolhidos antes de  $x$  ser encontrado e RANDOM-SEARCH terminar? Sua resposta deve ser uma função de  $n$  e  $k$ .
- Suponha que não exista nenhum índice  $i$  tal que  $A[i] = x$ . Qual é o número esperado de índices em  $A$  que devem ser escolhidos antes de todos os elementos de  $A$  terem sido verificados e RANDOM-SEARCH terminar?

Agora, considere um algoritmo de pesquisa linear determinística, que denominamos DETERMINISTIC-SEARCH. Especificamente, o algoritmo pesquisa  $A$  para  $x$  em ordem, considerando  $A[1], A[2], A[3], \dots, A[n]$  até  $A[i] = x$  ser encontrado ou alcançar o fim do arranjo. Suponha que todas as permutações possíveis do arranjo de entrada sejam igualmente prováveis.

- Suponha que existe exatamente um índice  $i$  tal que  $A[i] = x$ . Qual é o tempo de execução esperado de DETERMINISTIC-SEARCH? Qual é o tempo de execução no pior caso de DETERMINISTIC-SEARCH?
- Generalizando sua solução para parte (e), suponha que existam  $k \geq 1$  índices  $i$  tais que  $A[i] = x$ . Qual é o tempo de execução esperado de DETERMINISTIC-SEARCH? Qual é tempo de execução no pior caso de DETERMINISTIC-SEARCH? Sua resposta deve ser uma função de  $n$  e  $k$ .
- Suponha que não exista nenhum índice  $i$  tal que  $A[i] = x$ . Qual é o tempo de execução esperado de DETERMINISTIC-SEARCH? Qual é o tempo de execução no pior caso de DETERMINISTIC-SEARCH?

Finalmente, considere um algoritmo aleatório SCRAMBLE-SEARCH que funciona primeiro permitando aleatoriamente o arranjo de entrada e depois executando a pesquisa linear determinística anterior sobre o arranjo permutado resultante.

- b.** Sendo  $k$  o número de índices  $i$  tais que  $A[i] = x$ , forneça os tempos de execução no pior caso e esperado de SCRAMBLE-SEARCH para os casos em que  $k = 0$  e  $k = 1$ . Generalize sua solução para tratar o caso em que  $k \geq 1$ .
- i.* Qual dos três algoritmos de pesquisa você usaria? Explique sua resposta.

## Notas do capítulo

Bollobás [44], Hofri [151] e Spencer [283] contêm um grande número de técnicas probabilísticas avançadas. As vantagens dos algoritmos aleatórios são discutidas e pesquisadas por Karp [174] e Rabin [253]. O livro-texto de Motwani e Raghavan [228] apresenta um tratamento extensivo de algoritmos aleatórios.

Têm sido amplamente estudadas diversas variantes do problema da contratação. Esses problemas são mais comumente referidos como “problemas da secretária”. Um exemplo de trabalho nessa área é o artigo de Ajtai, Meggido e Waarts [12].



---

## *Parte II*

# *Ordenação e estatísticas de ordem*

### **Introdução**

Esta parte apresenta vários algoritmos que resolvem o **problema de ordenação** a seguir:

**Entrada:** Uma seqüência de  $n$  números  $\langle a_1, a_2, \dots, a_n \rangle$ .

**Saída:** Uma permutação (reordenação)  $\langle a'_1, a'_2, \dots, a'_n \rangle$  da seqüência de entrada, tal que  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

A seqüência de entrada normalmente é um arranjo de  $n$  elementos, embora possa ser representada de algum outro modo, como uma lista ligada.

### **A estrutura dos dados**

Na prática, os números a serem ordenados raramente são valores isolados. Em geral, cada um deles faz parte de uma coleção de dados chamada **registro**. Cada registro contém uma **chave**, que é o valor a ser ordenado, e o restante do registro consiste em **dados satélite**, que quase sempre são transportados junto com a chave. Na prática, quando um algoritmo de ordenação permuta as chaves, ele também deve permutar os dados satélite. Se cada registro inclui uma grande quantidade de dados satélite, muitas vezes permutamos um arranjo de ponteiros para os registros em lugar dos próprios registros, a fim de minimizar a movimentação de dados.

De certo modo, são esses detalhes de implementação que distinguem um algoritmo de um programa completamente implementado. O fato de ordenarmos números individuais ou grandes registros que contêm números é irrelevante para o **método** pelo qual um procedimento de ordenação determina a seqüência ordenada. Desse modo, quando nos concentramos no problema de ordenação, em geral supomos que a entrada consiste apenas em números. A tradução de um algoritmo para ordenação de números em um programa para ordenação de registros é conceitualmente direta, embora em uma situação específica de engenharia possam surgir outras sutilezas que fazem da tarefa real de programação um desafio.

## Por que ordenar?

Muitos cientistas de computação consideram a ordenação o problema mais fundamental no estudo de algoritmos. Há várias razões:

- Às vezes, a necessidade de ordenar informações é inherente a uma aplicação. Por exemplo, para preparar os extratos de clientes, os bancos precisam ordenar os cheques pelo número do cheque.
- Os algoritmos freqüentemente usam a ordenação como uma sub-rotina chave. Por exemplo, um programa que apresenta objetos gráficos dispostos em camadas uns sobre os outros talvez tenha de ordenar os objetos de acordo com uma relação “acima”, de forma a poder desenhar esses objetos de baixo para cima. Neste texto, veremos numerosos algoritmos que utilizam a ordenação como uma sub-rotina.
- Existe uma ampla variedade de algoritmos de ordenação, e eles empregam um rico conjunto de técnicas. De fato, muitas técnicas importantes usadas ao longo do projeto de algoritmos são representadas no corpo de algoritmos de ordenação que foram desenvolvidos ao longo dos anos. Desse modo, a ordenação também é um problema de interesse histórico.
- A ordenação é um problema para o qual podemos demonstrar um limite inferior não trivial (como faremos no Capítulo 8). Nossos melhores limites superiores correspondem ao limite inferior assintoticamente, e assim sabemos que nossos algoritmos de ordenação são assintoticamente ótimos. Além disso, podemos usar o limite inferior da ordenação com a finalidade de demonstrar limites inferiores para alguns outros problemas.
- Muitas questões de engenharia surgem quando se implementam algoritmos de ordenação. O programa de ordenação mais rápido para uma determinada situação pode depender de muitos fatores, como o conhecimento anterior a respeito das chaves e dos dados satélite, da hierarquia de memória (caches e memória virtual) do computador host e do ambiente de software. Muitas dessas questões são mais bem tratadas no nível algorítmico, em vez de ser necessário “mexer” no código.

## Algoritmos de ordenação

Introduzimos no Capítulo 2 dois algoritmos para ordenação de  $n$  números reais. A ordenação de inserção leva o tempo  $\Theta(n^2)$  no pior caso. Porém, pelo fato de seus loops internos serem compactos, ela é um rápido algoritmo de ordenação local para pequenos tamanhos de entrada. (Lembre-se de que um algoritmo de ordenação efetua a ordenação *local* se somente um número constante de elementos do arranjo de entrada sempre são armazenados fora do arranjo.) A ordenação por intercalação tem um tempo assintótico de execução melhor,  $\Theta(n \lg n)$ , mas o procedimento MERGE que ela utiliza não opera no local.

Nesta parte, apresentaremos mais dois algoritmos que ordenam números reais arbitrários. O heapsort, apresentado no Capítulo 6, efetua a ordenação de  $n$  números localmente, no tempo  $\Theta(n \lg n)$ . Ele usa uma importante estrutura de dados, chamada heap (monte), com a qual também podemos implementar uma fila de prioridades.

O quicksort, no Capítulo 7, também ordena  $n$  números localmente, mas seu tempo de execução no pior caso é  $\Theta(n^2)$ . Porém, seu tempo de execução no caso médio é  $\Theta(n \lg n)$ , e ele em geral supera o heapsort na prática. Como a ordenação por inserção, o quicksort tem um código compacto, e assim o fator constante oculto em seu tempo de execução é pequeno. Ele é um algoritmo popular para ordenação de grandes arranjos de entrada.

A ordenação por inserção, a ordenação por intercalação, o heapsort e o quicksort são todos ordenações por comparação: eles determinam a seqüência ordenada de um arranjo de entrada pela comparação dos elementos. O Capítulo 8 começa introduzindo o modelo de árvore de de-

cisão, a fim de estudar as limitações de desempenho de ordenações por comparação. Usando esse modelo, provamos um limite inferior igual a  $\Omega(n \lg n)$  no tempo de execução do pior caso de qualquer ordenação por comparação sobre  $n$  entradas, mostrando assim que o heapsort e a ordenação por intercalação são ordenações por comparação assintoticamente ótimas.

Em seguida, o Capítulo 8 mostra que poderemos superar esse limite inferior  $\Omega(n \lg n)$  se for possível reunir informações sobre a seqüência ordenada da entrada por outros meios além da comparação dos elementos. Por exemplo, o algoritmo de ordenação por contagem pressupõe que os números da entrada estão no conjunto  $\{1, 2, \dots, k\}$ . Usando a indexação de arranjos como uma ferramenta para determinar a ordem relativa, a ordenação por contagem pode ordenar  $n$  números no tempo  $(k + n)$ . Desse modo, quando  $k = O(n)$ , a ordenação por contagem é executada em um tempo linear no tamanho do arranjo de entrada. Um algoritmo relacionado, a radix sort (ordenação da raiz), pode ser usado para estender o intervalo da ordenação por contagem. Se houver  $n$  inteiros para ordenar, cada inteiro tiver  $d$  dígitos e cada dígito estiver no conjunto  $\{1, 2, \dots, k\}$ , a radix sort poderá ordenar os números em um tempo  $O(d(n + k))$ . Quando  $d$  é uma constante e  $k$  é  $O(n)$ , a radix sort é executada em tempo linear. Um terceiro algoritmo, bucket sort (ordenação por balde), requer o conhecimento da distribuição probabilística dos números no arranjo de entrada. Ele pode ordenar  $n$  números reais distribuídos uniformemente no intervalo meio aberto  $[0, 1]$  no tempo do caso médio  $O(n)$ .

## Estatísticas de ordem

A  $i$ -ésima estatística de ordem de um conjunto de  $n$  números é o  $i$ -ésimo menor número no conjunto. É claro que uma pessoa pode selecionar a  $i$ -ésima estatística de ordem, ordenando a entrada e indexando o  $i$ -ésimo elemento da saída. Sem quaisquer suposições a respeito da distribuição da entrada, esse método é executado no tempo  $\Omega(n \lg n)$ , como mostra o limite inferior demonstrado no Capítulo 8.

No Capítulo 9, mostramos que é possível encontrar o  $i$ -ésimo menor elemento no tempo  $O(n)$ , mesmo quando os elementos são números reais arbitrários. Apresentamos um algoritmo com pseudocódigo compacto que é executado no tempo  $\Theta(n^2)$  no pior caso, mas em tempo linear no caso médio. Também fornecemos um algoritmo mais complicado que é executado em um tempo  $O(n)$  no pior caso.

## Experiência necessária

Embora a maioria das seções desta parte não dependa de conceitos matemáticos difíceis, algumas seções exigem uma certa sofisticação matemática. Em particular, as análises do caso médio do quicksort, do bucket sort e do algoritmo de estatística de ordem utilizam a probabilidade, o que revisamos no Apêndice C; o material sobre a análise probabilística e os algoritmos aleatórios é estudado no Capítulo 5. A análise do algoritmo de tempo linear do pior caso para estatísticas de ordem envolve matemática um pouco mais sofisticada que as análises do pior caso desta parte.



---

# *Capítulo 6*

## *Heapsort*

Neste capítulo, introduzimos outro algoritmo de ordenação. Como a ordenação por intercalação, mas diferente da ordenação por inserção, o tempo de execução do heapsort é  $O(n \lg n)$ . Como a ordenação por inserção, mas diferente da ordenação por intercalação, o heapsort ordena localmente: apenas um número constante de elementos do arranjo é armazenado fora do arranjo de entrada em qualquer instante. Desse modo, o heapsort combina os melhores atributos dos dois algoritmos de ordenação que já discutimos.

O heapsort também introduz outra técnica de projeto de algoritmos: o uso de uma estrutura de dados, nesse caso uma estrutura que chamamos “heap” (ou “monte”) para gerenciar informações durante a execução do algoritmo. A estrutura de dados heap não é útil apenas para o heapsort (ou ordenação por heap); ela também cria uma eficiente fila de prioridades. A estrutura de dados heap reaparecerá em algoritmos de capítulos posteriores.

Observamos que o termo “heap” foi cunhado originalmente no contexto do heapsort, mas, desde então, ele passou a se referir ao “espaço para armazenamento do lixo coletado” como o espaço proporcionado pelas linguagens de programação Lisp e Java. Nossa estrutura de dados heap *não* é um espaço para armazenamento do lixo coletado e, sempre que mencionarmos heaps neste livro, estaremos fazendo referência à estrutura de dados definida neste capítulo.

### **6.1 Heaps**

A estrutura de dados **heap (binário)** é um objeto arranjo que pode ser visto como uma árvore binária praticamente completa (ver Seção B.5.3), como mostra a Figura 6.1. Cada nó da árvore corresponde a um elemento do arranjo que armazena o valor no nó. A árvore está completamente preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda até certo ponto. Um arranjo  $A$  que representa um heap é um objeto com dois atributos:  $\text{comprimento}[A]$ , que é o número de elementos no arranjo, e  $\text{tamanho-do-heap}[A]$ , o número de elementos no heap armazenado dentro do arranjo  $A$ . Ou seja, embora  $A[1 .. \text{comprimento}[A]]$  possa conter números válidos, nenhum elemento além de  $A[\text{tamanho-do-heap}[A]]$ , onde  $\text{tamanho-do-heap}[A] \leq \text{comprimento}[A]$ , é um elemento do heap. A raiz da árvore é  $A[1]$  e, dado o índice  $i$  de um nó, os índices de seu pai  $\text{PARENT}(i)$ , do filho da esquerda  $\text{LEFT}(i)$  e do filho da direita  $\text{RIGHT}(i)$  podem ser calculados de modo simples:

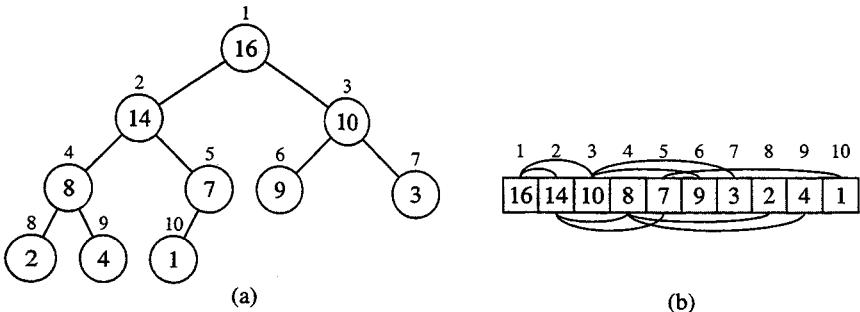


FIGURA 6.1 Um heap máximo visto como (a) uma árvore binária e (b) um arranjo. O número dentro do círculo em cada nó na árvore é o valor armazenado nesse nó. O número acima de um nó é o índice correspondente no arranjo. Acima e abaixo do arranjo encontramos linhas mostrando relacionamentos pai-filho; os pais estão sempre à esquerda de seus filhos. A árvore tem altura três; o nó no índice 4 (com o valor 8) tem altura um

```
PARENT( $i$ )
return  $\lfloor i/2 \rfloor$ 
```

```
LEFT( $i$ )
return  $2i$ 
```

```
RIGHT( $i$ )
return  $2i + 1$ 
```

Na maioria dos computadores, o procedimento LEFT pode calcular  $2i$  em uma única instrução, simplesmente deslocando a representação binária de  $i$  uma posição de bit para a esquerda. De modo semelhante, o procedimento RIGHT pode calcular rapidamente  $2i + 1$  deslocando a representação binária de  $i$  uma posição de bit para a esquerda e inserindo 1 como valor do bit de baixa ordem. O procedimento PARENT pode calcular  $\lfloor i/2 \rfloor$  deslocando  $i$  uma posição de bit para a direita. Em uma boa implementação de heapsort, esses três procedimentos são executados frequentemente como “macros” ou como procedimentos “em linha”.

Existem dois tipos de heaps binários: heaps máximos e heaps mínimos. Em ambos os tipos, os valores nos nós satisfazem a uma **propriedade de heap**, cujos detalhes específicos dependem do tipo de heap. Em um **heap máximo**, a **propriedade de heap máximo** é que, para todo nó  $i$  diferente da raiz,

$$A[\text{PARENT}(i)] \geq A[i],$$

isto é, o valor de um nó é no máximo o valor de seu pai. Desse modo, o maior elemento em um heap máximo é armazenado na raiz, e a subárvore que tem raiz em um nó contém valores menores que o próprio nó. Um **heap mínimo** é organizado de modo oposto; a **propriedade de heap mínimo** é que, para todo nó  $i$  diferente da raiz,

$$A[\text{PARENT}(i)] \leq A[i].$$

O menor elemento em um heap mínimo está na raiz.

Para o algoritmo de heapsort, usamos heaps máximos. Heaps mínimos são comumente empregados em filas de prioridades, que discutimos na Seção 6.5. Seremos precisos ao especificar se necessitamos de um heap máximo ou de um heap mínimo para qualquer aplicação específica e, quando as propriedades se aplicarem tanto a heaps máximos quanto a heaps mínimos, simplesmente usaremos o termo “heap”.

Visualizando um heap como uma árvore, definimos a *altura* de um nó em um heap como o número de arestas no caminho descendente simples mais longo desde o nó até uma folha, e definimos a altura do heap como a altura de sua raiz. Tendo em vista que um heap de  $n$  elementos é baseado em uma árvore binária completa, sua altura é  $\Theta(\lg n)$  (ver Exercício 6.1-2). Veremos que as operações básicas sobre heaps são executadas em um tempo máximo proporcional à altura da árvore, e assim demoram um tempo  $O(\lg n)$ . O restante deste capítulo apresenta alguns procedimentos básicos e mostra como eles são usados em um algoritmo de ordenação e em uma estrutura de dados de fila de prioridades.

- O procedimento MAX-HEAPIFY, executado no tempo  $O(\lg n)$ , é a chave para manter a propriedade de heap máximo (6.1).
- O procedimento BUILD-MAX-HEAP, executado em tempo linear, produz um heap a partir de um arranjo de entrada não ordenado.
- O procedimento HEAPSORT, executado no tempo  $O(n \lg n)$ , ordena um arranjo localmente.
- Os procedimentos MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY e HEAP-MAXIMUM, executados no tempo  $O(\lg n)$ , permitem que a estrutura de dados heap seja utilizada como uma fila de prioridades.

## Exercícios

### 6.1-1

Quais são os números mínimo e máximo de elementos em um heap de altura  $b$ ?

### 6.1-2

Mostre que um heap de  $n$  elementos tem altura  $\lfloor \lg n \rfloor$ .

### 6.1-3

Mostre que, em qualquer subárvore de um heap máximo, a raiz da subárvore contém o maior valor que ocorre em qualquer lugar nessa subárvore.

### 6.1-4

Onde em um heap máximo o menor elemento poderia residir, supondo-se que todos os elementos sejam distintos?

### 6.1-5

Um arranjo que está em seqüência ordenada é um heap mínimo?

### 6.1-6

A seqüência  $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$  é um heap máximo?

### 6.1-7

Mostre que, com a representação de arranjo para armazenar um heap de  $n$  elementos, as folhas são os nós indexados por  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ .

## 6.2 Manutenção da propriedade de heap

MAX-HEAPIFY é uma sub-rotina importante para manipulação de heaps máximos. Suas entradas são um arranjo  $A$  e um índice  $i$  para o arranjo. Quando MAX-HEAPIFY é chamado, supomos que as árvores binárias com raízes em  $\text{LEFT}(i)$  e  $\text{RIGHT}(i)$  são heaps máximos, mas que  $A[i]$  pode ser menor que seus filhos, violando assim a propriedade de heap máximo. A função de MAX-HEAPIFY é deixar que o valor em  $A[i]$  “flutue para baixo” no heap máximo, de tal forma que a subárvore com raiz no índice  $i$  se torne um heap máximo.

```

MAX-HEAPIFY( $A$ ,  $i$ )
1  $l \leftarrow \text{LEFT}(i)$ 
2  $r \leftarrow \text{RIGHT}(i)$ 
3 if  $l \leq \text{tamanho-do-heap}[A]$  e  $A[l] > A[i]$ 
4   then  $\text{maior} \leftarrow l$ 
5   else  $\text{maior} \leftarrow i$ 
6 if  $r \leq \text{tamanho-do-heap}[A]$  e  $A[r] > A[\text{maior}]$ 
7   then  $\text{maior} \leftarrow r$ 
8 if  $\text{maior} \neq i$ 
9   then trocar  $A[i] \leftrightarrow A[\text{maior}]$ 
10  MAX-HEAPIFY( $A$ ,  $\text{maior}$ )

```

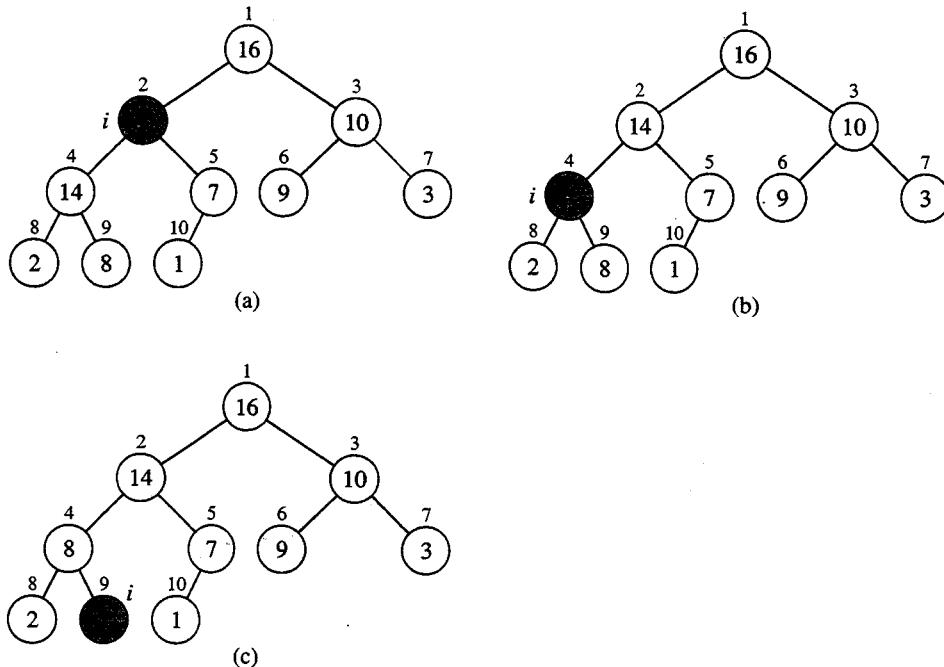


FIGURA 6.2 A ação de MAX-HEAPIFY( $A$ , 2), onde  $\text{tamanho-do-heap}[A] = 10$ . (a) A configuração inicial, com  $A[2]$  no nó  $i = 2$ , violando a propriedade de heap máximo, pois ele não é maior que ambos os filhos. A propriedade de heap máximo é restabelecida para o nó 2 em (b) pela troca de  $A[2]$  por  $A[4]$ , o que destrói a propriedade de heap máximo para o nó 4. A chamada recursiva MAX-HEAPIFY( $A$ , 4) agora define  $i = 4$ . Após a troca de  $A[4]$  por  $A[9]$ , como mostramos em (c), o nó 4 é corrigido, e a chamada recursiva a MAX-HEAPIFY( $A$ , 9) não produz nenhuma mudança adicional na estrutura de dados

A Figura 6.2 ilustra a ação de MAX-HEAPIFY. Em cada passo, o maior entre os elementos  $A[i]$ ,  $A[\text{LEFT}(i)]$  e  $A[\text{RIGHT}(i)]$  é determinado, e seu índice é armazenado em  $\text{maior}$ . Se  $A[i]$  é maior, então a subárvore com raiz no nó  $i$  é um heap máximo e o procedimento termina. Caso contrário, um dos dois filhos tem o maior elemento, e  $A[i]$  é trocado por  $A[\text{maior}]$ , o que faz o nó  $i$  e seus filhos satisfazerem à propriedade de heap máximo. Porém, agora o nó indexado por  $\text{maior}$  tem o valor original  $A[i]$  e, desse modo, a subárvore com raiz em  $\text{maior}$  pode violar a propriedade de heap máximo. Em consequência disso, MAX-HEAPIFY deve ser chamado recursivamente nessa subárvore.

O tempo de execução de MAX-HEAPIFY em uma subárvore de tamanho  $n$  com raiz em um dado nó  $i$  é o tempo  $\Theta(1)$  para corrigir os relacionamentos entre os elementos  $A[i]$ ,  $A[\text{LEFT}(i)]$  e  $A[\text{RIGHT}(i)]$ , mais o tempo para executar MAX-HEAPIFY em uma subárvore com raiz em um dos filhos do nó  $i$ . As subárvore de cada filho têm tamanho máximo igual a  $2n/3$  – o pior caso ocorre quando a última linha da árvore está exatamente metade cheia – e o tempo de execução de MAX-HEAPIFY pode então ser descrito pela recorrência

$$T(n) \leq T(2n/3) + \Theta(1)$$

A solução para essa recorrência, de acordo com o caso 2 do teorema mestre (Teorema 4.1), é  $T(n) = O(\lg n)$ . Como alternativa, podemos caracterizar o tempo de execução de MAX-HEAPIFY em um nó de altura  $b$  como  $O(b)$ .

## Exercícios

### 6.2-1

Usando a Figura 6.2 como modelo, ilustre a operação de MAX-HEAPIFY( $A, 3$ ) sobre o arranjo  $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$ .

### 6.2-2

Começando com o procedimento MAX-HEAPIFY, escreva pseudocódigo para o procedimento MIN-HEAPIFY( $A, i$ ), que executa a manipulação correspondente sobre um heap mínimo. Como o tempo de execução de MIN-HEAPIFY se compara ao de MAX-HEAPIFY?

### 6.2-3

Qual é o efeito de chamar MAX-HEAPIFY( $A, i$ ) quando o elemento  $A[i]$  é maior que seus filhos?

### 6.2-4

Qual é o efeito de chamar MAX-HEAPIFY( $A, i$ ) para  $i > \text{tamanho-do-heap}[A]/2$ ?

### 6.2-5

O código de MAX-HEAPIFY é bastante eficiente em termos de fatores constantes, exceto possivelmente para a chamada recursiva da linha 10, que poderia fazer alguns compiladores produzirem um código ineficiente. Escreva um MAX-HEAPIFY eficiente que use uma construção de controle iterativa (um loop) em lugar da recursão.

### 6.2-6

Mostre que o tempo de execução do pior caso de MAX-HEAPIFY sobre um heap de tamanho  $n$  é  $\Omega(\lg n)$ . (Sugestão: Para um heap com  $n$  nós, forneça valores de nós que façam MAX-HEAPIFY ser chamado recursivamente em todo nó sobre um caminho desde a raiz até uma folha.)

## 6.3 A construção de um heap

Podemos usar o procedimento MAX-HEAPIFY de baixo para cima, a fim de converter um arranjo  $A[1..n]$ , onde  $n = \text{comprimento}[A]$ , em um heap máximo. Pelo Exercício 6.1-7, os elementos no subarranjo  $A[\lfloor n/2 \rfloor + 1]..n]$  são todos folhas da árvore, e então cada um deles é um heap de 1 elemento com o qual podemos começar. O procedimento BUILD-MAX-HEAP percorre os nós restantes da árvore e executa MAX-HEAPIFY sobre cada um.

### BUILD-MAX-HEAP( $A$ )

- 1  $\text{tamanho-do-heap}[A] \leftarrow \text{comprimento}[A]$
- 2 **for**  $i \leftarrow \lfloor \text{comprimento}[A]/2 \rfloor$  **downto** 1
- 3   **do** MAX-HEAPIFY( $A, i$ )

A Figura 6.3 ilustra um exemplo da ação de BUILD-MAX-HEAP.

Para mostrar por que BUILD-MAX-HEAP funciona corretamente, usamos o seguinte loop invariante:

No começo de cada iteração do loop **for** das linhas 2 e 3, cada nó  $i+1, i+2, \dots, n$  é a raiz de um heap máximo.

A	4	1	3	2	16	9	10	14	8	7
---	---	---	---	---	----	---	----	----	---	---

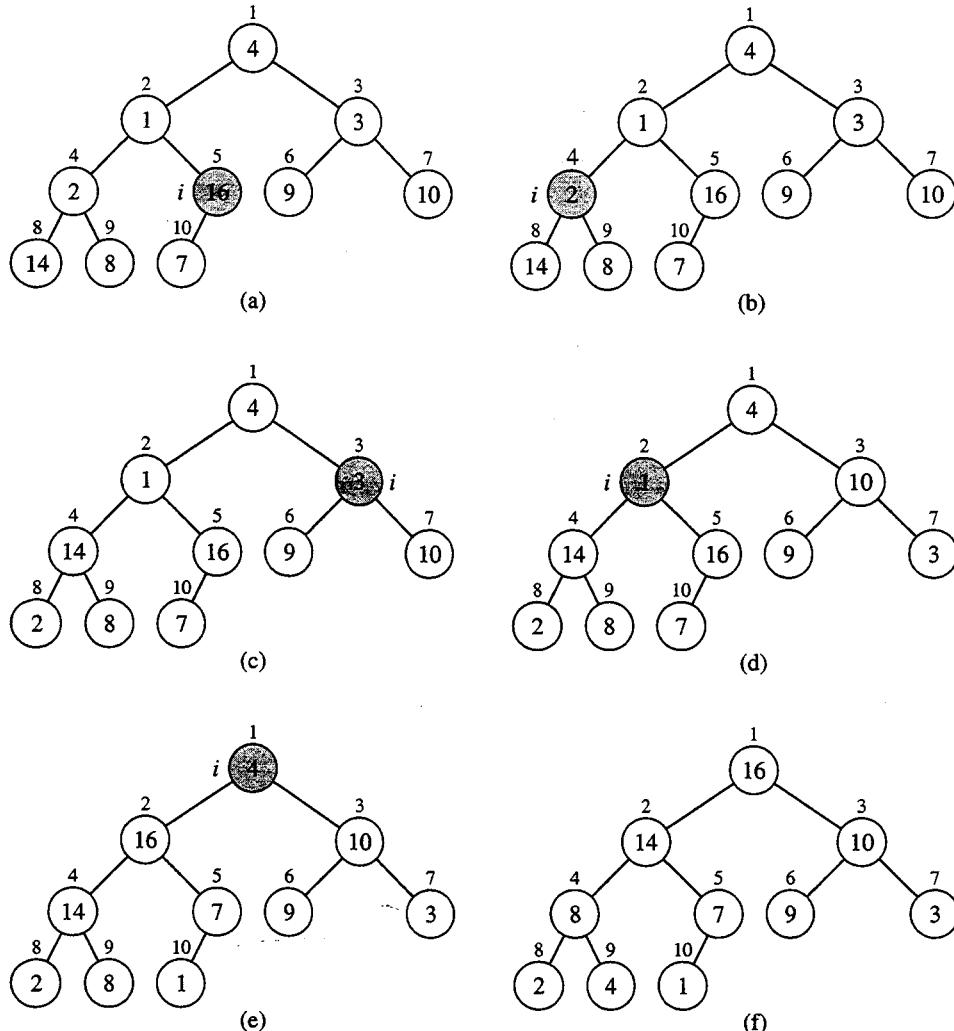


FIGURA 6.3 A operação de BUILD-MAX-HEAP, mostrando a estrutura de dados antes da chamada a MAX-HEAPIFY na linha 3 de BUILD-MAX-HEAP. (a) Um arranjo de entrada de 10 elementos  $A$  e a árvore binária que ele representa. A figura mostra que o índice de loop  $i$  se refere ao nó 5 antes da chamada MAX-HEAPIFY( $A, i$ ). (b) A estrutura de dados que resulta. O índice de loop  $i$  para a próxima iteração aponta para o nó 4. (c)–(e) Iterações subsequentes do loop for em BUILD-MAX-HEAP. Observe que, sempre que MAX-HEAPIFY é chamado em um nó, as duas subárvore desse nó são ambas heaps máximos. (f) O heap máximo após o término de BUILD-MAX-HEAP

Precisamos mostrar que esse invariante é verdade antes da primeira iteração do loop, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

**Inicialização:** Antes da primeira iteração do loop,  $i = \lfloor n/2 \rfloor$ . Cada nó  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$  é uma folha, e é portanto a raiz de um heap máximo trivial.

**Manutenção:** Para ver que cada iteração mantém o loop invariante, observe que os filhos do nó  $i$  são numerados com valores mais altos que  $i$ . Assim, pelo loop invariante, ambos são raízes de heaps máximos. Essa é precisamente a condição exigida para a chamada MAX-HEAPIFY( $A, i$ ) para tornar o nó  $i$  a raiz de um heap máximo. Além disso, a chamada a MAX-HEAPIFY preserva a propriedade de que os nós  $i + 1, i + 2, \dots, n$  são todos raízes de heaps máximos. Decrementar  $i$  na atualização do loop for restabelece o loop invariante para a próxima iteração.

**Término:** No término,  $i = 0$ . Pelo loop invariante, cada nó  $1, 2, \dots, n$  é a raiz de um heap máximo. Em particular, o nó 1 é uma raiz.

Podemos calcular um limite superior simples sobre o tempo de execução de BUILD-MAX-HEAP como a seguir. Cada chamada a MAX-HEAPIFY custa o tempo  $O(\lg n)$ , e existem  $O(n)$  dessas chamadas. Desse modo, o tempo de execução é  $O(n \lg n)$ . Esse limite superior, embora correto, não é assintoticamente restrito.

Podemos derivar um limite mais restrito observando que o tempo de execução de MAX-HEAPIFY sobre um nó varia com a altura do nó na árvore, e as alturas na maioria dos nós são pequenas. Nossa análise mais restrita se baseia nas propriedades de que um heap de  $n$  elementos tem altura  $\lfloor \lg n \rfloor$  (ver Exercício 6.1-2) e no máximo  $\lceil n/2^b + 1 \rceil$  nós de qualquer altura  $b$  (ver Exercício 6.3-3).

O tempo exigido por MAX-HEAPIFY quando é chamado em um nó de altura  $b$  é  $O(b)$ ; assim, podemos expressar o custo total de BUILD-MAX-HEAP por

$$\sum_{b=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{b+1}} \right\rceil O(b) = O\left(n \sum_{b=0}^{\lfloor \lg n \rfloor} \frac{b}{2^b}\right).$$

O último somatório pode ser calculado substituindo-se  $x = 1/2$  na fórmula (A.8), o que produz

$$\sum_{b=0}^{\infty} \frac{b}{2^b} = \frac{1/2}{(1 - 1/2)^2}$$

$$= 2.$$

Desse modo, o tempo de execução de BUILD-MAX-HEAP pode ser limitado como

$$\begin{aligned} O\left(n \sum_{b=0}^{\lfloor \lg n \rfloor} \frac{b}{2^b}\right) &= O\left(n \sum_{b=0}^{\infty} \frac{b}{2^b}\right) \\ &= O(n). \end{aligned}$$

Conseqüentemente, podemos construir um heap máximo a partir de um arranjo não ordenado em tempo linear.

Podemos construir um heap mínimo pelo procedimento BUILD-MIN-HEAP, que é igual a BUILD-MAX-HEAP, mas tem a chamada a MAX-HEAPIFY na linha 3 substituída por uma chamada a MIN-HEAPFY (ver Exercício 6.2-2). BUILD-MIN-HEAP produz um heap mínimo a partir de um arranjo linear não ordenado em tempo linear.

## Exercícios

### 6.3-1

Usando a Figura 6.3 como modelo, ilustre a operação de BUILD-MAX-HEAP sobre o arranjo  $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$ .

### 6.3-2

Por que queremos que o índice de loop  $i$  na linha 2 de BUILD-MAX-HEAP diminua de  $\lfloor \text{comprimento}[A]/2 \rfloor$  até 1, em vez de aumentar de 1 até  $\lfloor \text{comprimento}[A]/2 \rfloor$ ?

### 6.3-3

Mostre que existem no máximo  $\lceil n/2^b + 1 \rceil$  nós de altura  $b$  em qualquer heap de  $n$  elementos. | 109

## 6.4 O algoritmo heapsort

O algoritmo heapsort (ordenação por monte) começa usando BUILD-MAX-HEAP para construir um heap no arranjo de entrada  $A[1..n]$ , onde  $n = \text{comprimento}[A]$ . Tendo em vista que o elemento máximo do arranjo está armazenado na raiz  $A[1]$ , ele pode ser colocado em sua posição final correta, trocando-se esse elemento por  $A[n]$ . Se agora “descartarmos” o nó  $n$  do heap (diminuindo  $\text{tamanho-do-heap}[A]$ ), observaremos que  $A[1 .. (n - 1)]$  pode ser facilmente transformado em um heap máximo. Os filhos da raiz continuam sendo heaps máximos, mas o novo elemento raiz pode violar a propriedade de heap máximo. Porém, tudo o que é necessário para restabelecer a propriedade de heap máximo é uma chamada a MAX-HEAPIFY( $A, 1$ ), que deixa um heap máximo em  $A[1 .. (n - 1)]$ . Então, o algoritmo heapsort repete esse processo para o heap de tamanho  $n - 1$ , descendo até um heap de tamanho 2. (Veja no Exercício 6.4-2 um loop invariante preciso.)

**HEAPSORT( $A$ )**

- 1 BUILD-MAX-HEAP( $A$ )
- 2 **for**  $i \leftarrow \text{comprimento}[A]$  **downto** 2
- 3   **do** trocar  $A[1] \leftrightarrow A[i]$
- 4     $\text{tamanho-do-heap}[A] \leftarrow \text{tamanho-do-heap}[A] - 1$
- 5    MAX-HEAPIFY( $A, 1$ )

A Figura 6.4 mostra um exemplo da operação de heapsort depois do heap máximo ser inicialmente construído. Cada heap máximo é mostrado no princípio de uma iteração do loop **for** das linhas 2 a 5.

O procedimento HEAPSORT demora o tempo  $O(n \lg n)$ , pois a chamada a BUILD-MAX-HEAP demora o tempo  $O(n)$ , e cada uma das  $n - 1$  chamadas a MAX-HEAPIFY demora o tempo  $O(\lg n)$ .

### Exercícios

#### 6.4-1

Usando a Figura 6.4 como modelo, ilustre a operação de HEAPSORT sobre o arranjo  $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$ .

#### 6.4-2

Discuta a correção de HEAPSORT usando o loop invariante a seguir:

No início de cada iteração do loop **for** das linhas 2 a 5, o subarranjo  $A[1 .. i]$  é um heap máximo contendo os  $i$  menores elementos de  $A[1 .. n]$ , e o subarranjo  $A[i + 1 .. n]$  contém os  $n - i$  maiores elementos de  $A[1 .. n]$ , ordenados.

#### 6.4-3

Qual é o tempo de execução de heapsort sobre um arranjo  $A$  de comprimento  $n$  que já está ordenado em ordem crescente? E em ordem decrescente?

#### 6.4-4

Mostre que o tempo de execução do pior caso de heapsort é  $\Omega(n \lg n)$ .

#### 6.4-5

Mostre que, quando todos os elementos são distintos, o tempo de execução do melhor caso de heapsort é  $\Omega(n \lg n)$ .

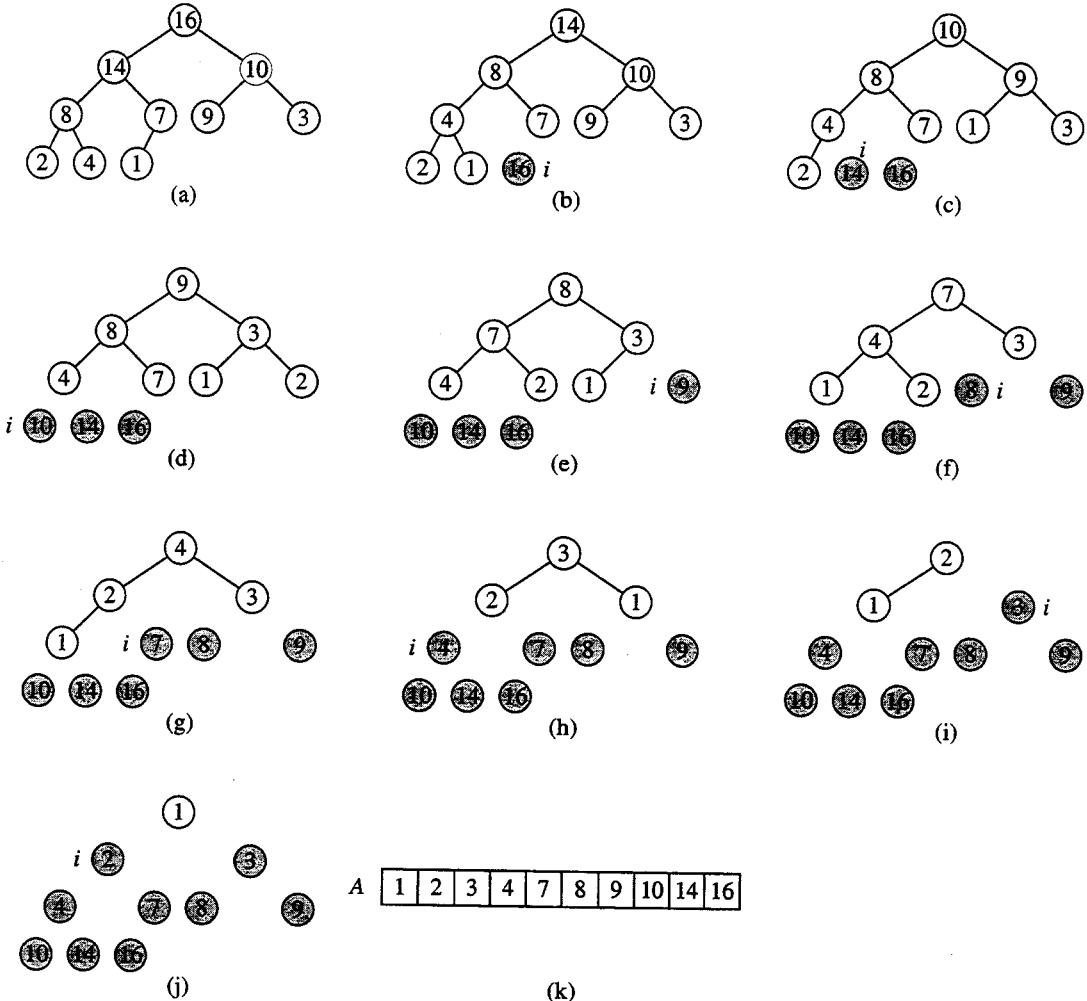


FIGURA 6.4 A operação de HEAPSORT. (a) A estrutura de dados heap máximo, logo após ter sido construída por BUILD-MAX-HEAP. (b)–(j) O heap máximo logo após cada chamada de MAX-HEAPIFY na linha 5. O valor de  $i$  nesse instante é mostrado. Apenas os nós levemente sombreados permanecem no heap. (k) O arranjo ordenado resultante  $A$

## 6.5 Filas de prioridades

O heapsort é um algoritmo excelente, mas uma boa implementação de quicksort, apresentado no Capítulo 7, normalmente o supera na prática. Não obstante, a estrutura de dados de heap propriamente dita tem uma utilidade enorme. Nesta seção, apresentaremos uma das aplicações mais populares de um heap: seu uso como uma fila de prioridades eficiente. Como ocorre no caso dos heaps, existem dois tipos de filas de prioridades: as filas de prioridade máxima e as filas de prioridade mínima. Focalizaremos aqui a implementação das filas de prioridade máxima, que por sua vez se baseiam em heaps máximos; o Exercício 6.5-3 lhe pede para escrever os procedimentos correspondentes a filas de prioridade mínima.

Uma **fila de prioridades** é uma estrutura de dados para manutenção de um conjunto  $S$  de elementos, cada qual com um valor associado chamado **chave**. Uma fila de prioridade máxima admite as operações a seguir.

$\text{INSERT}(S, x)$  insere o elemento  $x$  no conjunto  $S$ . Essa operação poderia ser escrita como  $S \leftarrow S \cup \{x\}$ .

$\text{MAXIMUM}(S)$  retorna o elemento de  $S$  com a maior chave.

$\text{EXTRACT-MAX}(S)$  remove e retorna o elemento de  $S$  com a maior chave.

**INCREASE-KEY( $S, x, k$ )** aumenta o valor da chave do elemento  $x$  para o novo valor  $k$ , que se presume ser pelo menos tão grande quanto o valor da chave atual de  $x$ .

Uma aplicação de filas de prioridade máxima é programar trabalhos em um computador compartilhado. A fila de prioridade máxima mantém o controle dos trabalhos a serem executados e de suas prioridades relativas. Quando um trabalho termina ou é interrompido, o trabalho de prioridade mais alta é selecionado dentre os trabalhos pendentes, com o uso de EXTRACT-MAX. Um novo trabalho pode ser adicionado à fila em qualquer instante, com a utilização de INSERT.

Como alternativa, uma *fila de prioridade mínima* admite as operações INSERT, MINIMUM, EXTRACT-MIN e DECREASE-KEY. Uma fila de prioridade mínima pode ser usada em um simulador orientado a eventos. Os itens na fila são eventos a serem simulados, cada qual com um tempo de ocorrência associado que serve como sua chave. Os eventos devem ser simulados em ordem de seu momento de ocorrência, porque a simulação de um evento pode provocar outros eventos a serem simulados no futuro. O programa de simulação utiliza EXTRACT-MIN em cada etapa para escolher o próximo evento a simular. À medida que novos eventos são produzidos, eles são inseridos na fila de prioridade mínima com o uso de INSERT. Veremos outros usos de filas de prioridade mínima destacando a operação DECREASE-KEY, nos Capítulos 23 e 24.

Não surpreende que possamos usar um heap para implementar uma fila de prioridades. Em uma dada aplicação, como a programação de trabalhos ou a simulação orientada a eventos, os elementos de uma fila de prioridades correspondem a objetos na aplicação. Frequentemente é necessário determinar que objeto da aplicação corresponde a um dado elemento de fila de prioridades e vice-versa. Então, quando um heap é usado para implementar uma fila de prioridades, com freqüência precisamos armazenar um *descriptor* para o objeto da aplicação correspondente em cada elemento do heap. A constituição exata do descriptor (isto é, um ponteiro, um inteiro etc.) depende da aplicação. De modo semelhante, precisamos armazenar um descriptor para o elemento do heap correspondente em cada objeto da aplicação. Aqui, o descriptor em geral seria um índice de arranjo. Como os elementos do heap mudam de posições dentro do arranjo durante operações de heap, uma implementação real, ao reposicionar um elemento do heap, também teria de atualizar o índice do arranjo no objeto da aplicação correspondente. Tendo em vista que os detalhes de acesso a objetos de aplicações dependem muito da aplicação e de sua implementação, não os examinaremos aqui, exceto por observar que na prática esses descritores precisam ser mantidos corretamente.

Agora descreveremos como implementar as operações de uma fila de prioridade máxima. O procedimento HEAP-MAXIMUM implementa a operação MAXIMUM no tempo  $\Theta(1)$ .

#### HEAP-MAXIMUM( $A$ )

1 **return**  $A[1]$

O procedimento HEAP-EXTRACT-MAX implementa a operação EXTRACT-MAX. Ele é semelhante ao corpo do loop **for** (linhas 3 a 5) do procedimento HEAPSORT:

#### HEAP-EXTRACT-MAX( $A$ )

1 **if** *tamanho-do-beap*[ $A$ ] < 1  
2   **then error** “heap underflow”  
3  $max \leftarrow A[1]$   
4  $A[1] \leftarrow A[tamanho-do-beap[A]]$   
5 *tamanho-do-beap*[ $A$ ]  $\leftarrow$  *tamanho-do-beap*[ $A$ ] – 1  
6 MAX-HEAPIFY( $A$ , 1)  
7 **return**  $max$

O tempo de execução de HEAP-EXTRACT-MAX é  $O(\lg n)$ , pois ele executa apenas uma porção constante do trabalho sobre o tempo  $O(\lg n)$  para MAX-HEAPIFY.

O procedimento HEAP-INCREASE-KEY implementa a operação INCREASE-KEY. O elemento da fila de prioridades cuja chave deve ser aumentada é identificado por um índice  $i$  no arranjo. Primeiro, o procedimento atualiza a chave do elemento  $A[i]$  para seu novo valor. Em seguida, como o aumento da chave de  $A[i]$  pode violar a propriedade de heap máximo, o procedimento, de um modo que é uma reminiscência do loop de inserção (linhas 5 a 7) de INSERTION-SORT da Seção 2.1, percorre um caminho desde esse nó em direção à raiz, até encontrar um lugar apropriado para o elemento recém-aumentado. Durante essa travessia, ele compara repetidamente um elemento a seu pai, permutando suas chaves e continuando se a chave do elemento é maior, e terminando se a chave do elemento é menor, pois a propriedade de heap máximo agora é válida. (Veja no Exercício 6.5-5 um loop invariante preciso.)

**HEAP-INCREASE-KEY( $A, i, chave$ )**

- 1 **if**  $chave < A[i]$
- 2   **then error** “nova chave é menor que chave atual”
- 3    $A[i] \leftarrow chave$
- 4   **while**  $i > 1$  e  $A[\text{PARENT}(i)] < A[i]$
- 5     **do troca**  $A[i] \leftrightarrow A[\text{PARENT}(i)]$
- 6      $i \leftarrow \text{PARENT}(i)$

A Figura 6.5 mostra um exemplo de uma operação de HEAP-INCREASE-KEY. O tempo de execução de HEAP-INCREASE-KEY sobre um heap de  $n$  elementos é  $O(\lg n)$ , pois o caminho traçado desde o nó atualizado na linha 3 até a raiz tem o comprimento  $O(\lg n)$ .

O procedimento MAX-HEAP-INSERT implementa a operação INSERT. Ele toma como uma entrada a chave do novo elemento a ser inserido no heap máximo  $A$ . Primeiro, o procedimento expande o heap máximo, adicionando à árvore uma nova folha cuja chave é  $-\infty$ . Em seguida, ele chama HEAP-INCREASE-KEY para definir a chave desse novo nó com seu valor correto e manter a propriedade de heap máximo.

**MAX-HEAP-INSERT( $A, chave$ )**

- 1 **tamanbo-do-heap**[ $A$ ]  $\leftarrow$  **tamanbo-do-heap**[ $A$ ] + 1
- 2  $A[\text{tamanbo-do-heap}[A]] \leftarrow -\infty$
- 3 **HEAP-INCREASE-KEY**( $A, \text{tamanbo-do-heap}[A], chave$ )

O tempo de execução de MAX-HEAP-INSERT sobre um heap de  $n$  elementos é  $O(\lg n)$ .

Em resumo, um heap pode admitir qualquer operação de fila de prioridades em um conjunto de tamanho  $n$  no tempo  $O(\lg n)$ .

## Exercícios

### 6.5-1

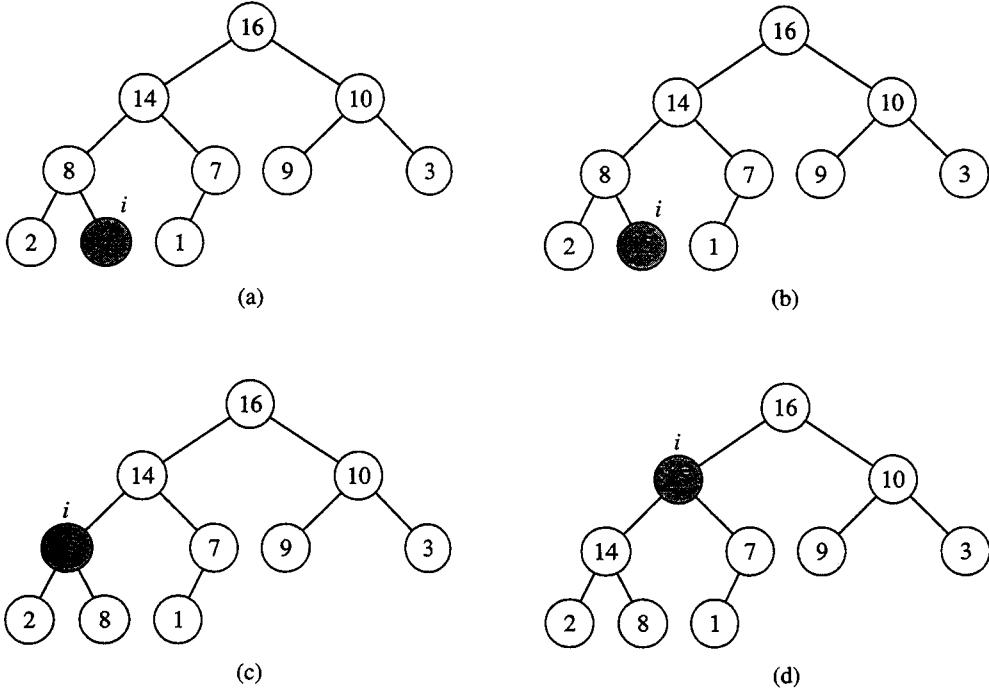
Ilustre a operação de HEAP-EXTRACT-MAX sobre o heap  $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$ .

### 6.5-2

Ilustre a operação de MAX-HEAP-INSERT( $A, 10$ ) sobre o heap  $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$ . Use o heap da Figura 6.5 como modelo para a chamada de HEAP-INCREASE-KEY.

### 6.5-3

Escreva pseudocódigo para os procedimentos HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY e MIN-HEAP-INSERT que implementam uma fila de prioridade mínima com um heap mínimo.



**FIGURA 6.5** A operação de `HEAP-INCREASE-KEY`. (a) O heap máximo da Figura 6.4(a) com um nó cujo índice é  $i$ , fortemente sombreado. (b) Esse nó tem sua chave aumentada para 15. (c) Depois de uma iteração do loop `while` das linhas 4 a 6, o nó e seu pai trocaram chaves, e o índice  $i$  sobe para o pai. (d) O heap máximo depois de mais uma iteração do loop `while`. Nesse momento,  $A[\text{PARENT}(i)] \geq A[i]$ . Agora, a propriedade de heap máximo é válida, e o procedimento termina

#### 6.5-4

Por que nos preocupamos em definir a chave do nó inserido como  $-\infty$  na linha 2 de `MAX-HEAP-INSERT` quando a nossa próxima ação é aumentar sua chave para o valor desejado?

#### 6.5-5

Demonstre a correção de `HEAP-INCREASE-KEY` usando este loop invariante:

No começo de cada iteração do loop `while` das linhas 4 a 6, o arranjo  $A[1 \dots \text{tamanho-do-beap}[A]]$  satisfaz à propriedade de heap máximo, a não ser pelo fato de que é possível haver uma violação:  $A[i]$  pode ser maior que  $A[\text{PARENT}(i)]$ .

#### 6.5-6

Mostre como implementar uma fila de primeiro a entrar, primeiro a sair com uma fila de prioridades. Mostre como implementar uma pilha com uma fila de prioridades. (Filas e pilhas são definidas na Seção 10.1.)

#### 6.5-7

A operação `HEAP-DELETE(A, i)` elimina o item do nó  $i$  do heap  $A$ . Forneça uma implementação de `HEAP-DELETE` que seja executada no tempo  $O(\lg n)$  para um heap máximo de  $n$  elementos.

#### 6.5-8

Forneça um algoritmo de tempo  $O(n \lg k)$  para intercalar  $k$  listas ordenadas em uma única lista ordenada, onde  $n$  é o número total de elementos em todas as listas de entrada. (*Sugestão:* Use um heap mínimo para fazer a intercalação de  $k$  modos.)

## Problemas

### 6-1 Construção de um heap com o uso de inserção

O procedimento BUILD-MAX-HEAP na Seção 6.3 pode ser implementado pelo uso repetido de MAX-HEAP-INSERT para inserir os elementos no heap. Considere a implementação a seguir:

BUILD-MAX-HEAP'(A)

- 1  $tamanho-do-heap[A] \leftarrow 1$
- 2 **for**  $i \leftarrow 2$  **to**  $comprimento[A]$
- 3   **do** MAX-HEAP-INSERT( $A, A[i]$ )

- a. Os procedimentos BUILD-MAX-HEAP e BUILD-MAX-HEAP' sempre criam o mesmo heap quando são executados sobre o mesmo arranjo de entrada? Prove que eles o fazem, ou então forneça um contra-exemplo.
- b. Mostre que no pior caso, BUILD-MAX-HEAP' exige o tempo  $\Theta(n \lg n)$  para construir um heap de  $n$  elementos.

### 6-2 Análise de heaps d-ários

Um **heap d-ário** é semelhante a um heap binário, mas (com uma única exceção possível) nós que não são de folhas têm  $d$  filhos em vez de dois filhos.

- a. Como você representaria um heap  $d$ -ário em um arranjo?
- b. Qual é a altura de um heap  $d$ -ário de  $n$  elementos em termos de  $n$  e  $d$ ?
- c. Dê uma implementação eficiente de EXTRACT-MAX em um heap máximo  $d$ -ário. Analise seu tempo de execução em termos de  $d$  e  $n$ .
- d. Forneça uma implementação eficiente de INSERT em um heap máximo  $d$ -ário. Analise seu tempo de execução em termos de  $d$  e  $n$ .
- e. Dê uma implementação eficiente de HEAP-INCREASE-KEY( $A, i, k$ ), que primeiro configura  $A[i] \leftarrow \max(A[i], k)$  e depois atualiza adequadamente a estrutura de heap máximo  $d$ -ário. Analise seu tempo de execução em termos de  $d$  e  $n$ .

### 6-3 Quadros de Young

Um **quadro de Young**  $m \times n$  é uma matriz  $m \times n$  tal que as entradas de cada linha estão em seqüência ordenada da esquerda para a direita, e as entradas de cada coluna estão em seqüência ordenada de cima para baixo. Algumas das entradas de um quadro de Young podem ser  $\infty$ , o que tratamos como elementos inexistentes. Desse modo, um quadro de Young pode ser usado para conter  $r \leq mn$  números finitos.

- a. Trace um quadro de Young  $4 \times 4$  contendo os elementos  $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$ .
- b. Demonstre que um quadro de Young  $Y m \times n$  é vazio se  $Y[1, 1] = \infty$ . Demonstre que  $Y$  é completo (contém  $mn$  elementos) se  $Y[m, n] < \infty$ .
- c. Forneça um algoritmo para implementar EXTRACT-MIN em um quadro de Young  $m \times n$  não vazio que funcione no tempo  $O(m+n)$ . Seu algoritmo deve usar uma sub-rotina recursiva que solucione um problema  $m \times n$  resolvendo recursivamente um subproblema  $(m-1) \times n$  ou  $m \times (n-1)$ . (Sugestão: Pense em MAX-HEAPIFY.) Defina  $T(p)$ , onde  $p = m + n$ , como o tempo de execução máximo de EXTRACT-MIN em qualquer quadro de Young  $m \times n$ . Forneça e resolva uma recorrência para  $T(p)$  que produza o limite de tempo  $O(m+n)$ .
- d. Mostre como inserir um novo elemento em um quadro de Young  $m \times n$  não completo no tempo  $O(m+n)$ .
- e. Sem usar nenhum outro método de ordenação como uma sub-rotina, mostre como utilizar um quadro de Young  $n \times n$  para ordenar  $n^2$  números no tempo  $O(n^3)$ .
- f. Forneça um algoritmo de tempo  $O(m+n)$  para determinar se um dado número está armazenado em um determinado quadro de Young  $m \times n$ .

## Notas do capítulo

O algoritmo heapsort foi criado por Williams [316], que também descreveu como implementar uma fila de prioridades com um heap. O procedimento BUILD-MAX-HEAP foi sugerido por Floyd [90].

Usamos heaps mínimos para implementar filas de prioridade mínima nos Capítulos 16, 23 e 24. Também damos uma implementação com limites de tempo melhorados para certas operações nos Capítulos 19 e 20.

Implementações mais rápidas de filas de prioridades são possíveis para dados inteiros. Uma estrutura de dados criada por van Emde Boas [301] admite as operações MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT-MIN, EXTRACT-MAX, PREDECESSOR e SUCCESSOR, no tempo de pior caso  $O(\lg \lg C)$ , sujeito à restrição de que o universo de chaves é o conjunto  $\{1, 2, \dots, C\}$ . Se os dados são inteiros de  $b$  bits e a memória do computador consiste em palavras de  $b$  bits endereçáveis, Fredman e Willard [99] mostraram como implementar MINIMUM no tempo  $O(1)$ , e INSERT e EXTRACT-MIN no tempo  $O(\sqrt{\lg n})$ . Thorup [299] melhorou o limite  $O(\sqrt{\lg n})$  para o tempo  $O((\lg \lg n)^2)$ . Esse limite usa uma quantidade de espaço ilimitada em  $n$ , mas pode ser implementado em espaço linear com o uso de hash aleatório.

Um caso especial importante de filas de prioridades ocorre quando a seqüência de operações de EXTRACT-MIN é *monotônica* ou *monótona*, ou seja, os valores retornados por operações sucessivas de EXTRACT-MIN são monotonicamente crescentes com o tempo. Esse caso surge em várias aplicações importantes, como o algoritmo de caminhos mais curtos de origem única de Dijkstra, discutido no Capítulo 24, e na simulação de eventos discretos. Para o algoritmo de Dijkstra é particularmente importante que a operação DECREASE-KEY seja implementada de modo eficiente.

No caso monotônico, se os dados são inteiros no intervalo  $1, 2, \dots, C$ , Ahuja, Melhorn, Orlin e Tarjan [8] descrevem como implementar EXTRACT-MIN e INSERT no tempo amortizado  $O(\lg C)$  (consulte o Capítulo 17 para obter mais informações sobre análise amortizada) e DECREASE-KEY no tempo  $O(1)$ , usando uma estrutura de dados chamada heap de raiz. O limite  $O(\lg C)$  pode ser melhorado para  $O(\sqrt{\lg C})$  com o uso de heaps de Fibonacci (consulte o Capítulo 20) em conjunto com heaps de raiz. O limite foi melhorado ainda mais para o tempo esperado  $O(\lg^{1/3 + \varepsilon} C)$  por Cherkassky, Goldberg e Silverstein [58], que combinam a estrutura de baldes em vários níveis de Denardo e Fox [72] com o heap de Thorup mencionado antes. Raman [256] melhorou mais ainda esses resultados para obter um limite de  $O(\min(\lg^{1/4 + \varepsilon} C, \lg^{1/3 + \varepsilon} n))$ , para qualquer  $\varepsilon > 0$ . Discussões mais detalhadas desses resultados podem ser encontradas em trabalhos de Raman [256] e Thorup [299].

---

## *Capítulo 7*

# *Quicksort*

O quicksort (ordenação rápida) é um algoritmo de ordenação cujo tempo de execução do pior caso é  $\Theta(n^2)$  sobre um arranjo de entrada de  $n$  números. Apesar desse tempo de execução lento no pior caso, o quicksort com freqüência é a melhor opção prática para ordenação, devido a sua notável eficiência na média: seu tempo de execução esperado é  $\Theta(n \lg n)$ , e os fatores constantes ocultos na notação  $\Theta(n \lg n)$  são bastante pequenos. Ele também apresenta a vantagem da ordenação local (ver Capítulo 2) e funciona bem até mesmo em ambientes de memória virtual.

A Seção 7.1 descreve o algoritmo e uma sub-rotina importante usada pelo quicksort para particionamento. Pelo fato do comportamento de quicksort ser complexo, começaremos com uma discussão intuitiva de seu desempenho na Seção 7.2 e adiaremos sua análise precisa até o final do capítulo. A Seção 7.3 apresenta uma versão de quicksort que utiliza a amostragem aleatória. Esse algoritmo tem um bom tempo de execução no caso médio, e nenhuma entrada específica induz seu comportamento do pior caso. O algoritmo aleatório é analisado na Seção 7.4, onde mostraremos que ele é executado no tempo  $\Theta(n^2)$  no pior caso e no tempo  $O(n \lg n)$  em média.

### **7.1 Descrição do quicksort**

O quicksort, como a ordenação por intercalação, se baseia no paradigma de dividir e conquistar introduzido na Seção 2.3.1. Aqui está o processo de dividir e conquistar em três passos para ordenar um subarranjo típico  $A[p .. r]$ .

**Dividir:** O arranjo  $A[p .. r]$  é particionado (reorganizado) em dois subarranjos (possivelmente vazios)  $A[p .. q-1]$  e  $A[q+1 .. r]$  tais que cada elemento de  $A[p .. q-1]$  é menor que ou igual a  $A[q]$  que, por sua vez, é igual ou menor a cada elemento de  $A[q+1 .. r]$ . O índice  $q$  é calculado como parte desse procedimento de particionamento.

**Conquistar:** Os dois subarranjos  $A[p .. q-1]$  e  $A[q+1 .. r]$  são ordenados por chamadas recursivas a quicksort.

**Combinar:** Como os subarranjos são ordenados localmente, não é necessário nenhum trabalho para combiná-los: o arranjo  $A[p .. r]$  inteiro agora está ordenado.

O procedimento a seguir implementa o quicksort.

```

QUICKSORT( $A, p, r$ )
1 if  $p < r$ 
2   then  $q \leftarrow \text{PARTITION}(A, p, r)$ 
3     QUICKSORT( $A, p, q - 1$ )
4     QUICKSORT( $A, q + 1, r$ )

```

Para ordenar um arranjo  $A$  inteiro, a chamada inicial é  $\text{QUICKSORT}(A, 1, \text{comprimento}[A])$ .

## Particionamento do arranjo

A chave para o algoritmo é o procedimento PARTITION, que reorganiza o subarranjo  $A[p .. r]$  localmente.

```

PARTITION( $A, p, r$ )
1  $x \leftarrow A[r]$ 
2  $i \leftarrow p - 1$ 
3 for  $j \leftarrow p$  to  $r - 1$ 
4   do if  $A[j] \leq x$ 
5     then  $i \leftarrow i + 1$ 
6     trocar  $A[i] \leftarrow A[j]$ 
7 trocar  $A[i + 1] \leftrightarrow A[r]$ 
8 return  $i + 1$ 

```

A Figura 7.1 mostra a operação de PARTITION sobre um arranjo de 8 elementos. PARTITION sempre seleciona um elemento  $x = A[r]$  como um elemento *pivô* ao redor do qual será feito o particionamento do subarranjo  $A[p .. r]$ . À medida que o procedimento é executado, o arranjo é particionado em quatro regiões (possivelmente vazias). No início de cada iteração do loop for nas linhas 3 a 6, cada região satisfaz a certas propriedades, que podemos enunciar como um loop invariante:

No início de cada iteração do loop das linhas 3 a 6, para qualquer índice de arranjo  $k$ ,

1. Se  $p \leq k \leq i$ , então  $A[k] \leq x$ .
2. Se  $i + 1 \leq k \leq j - 1$ , então  $A[k] > x$ .
3. Se  $k = r$ , então  $A[k] = x$ .

A Figura 7.2 resume essa estrutura. Os índices entre  $j$  e  $r - 1$  não são cobertos por quaisquer dos três casos, e os valores nessas entradas não têm nenhum relacionamento particular para o pivô  $x$ .

Precisamos mostrar que esse loop invariante é verdadeiro antes da primeira iteração, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

**Inicialização:** Antes da primeira iteração do loop,  $i = p - 1$  e  $j = p$ . Não há nenhum valor entre  $p$  e  $i$ , e nenhum valor entre  $i + 1$  e  $j - 1$ ; assim, as duas primeiras condições do loop invariante são satisfeitas de forma trivial. A atribuição na linha 1 satisfaz à terceira condição.

**Manutenção:** Como mostra a Figura 7.3, existem dois casos a considerar, dependendo do resultado do teste na linha 4. A Figura 7.3(a) mostra o que acontece quando  $A[j] > x$ ; a única ação no loop é incrementar  $j$ . Depois que  $j$  é incrementado, a condição 2 é válida para  $A[j - 1]$  e todas as outras entradas permanecem inalteradas.

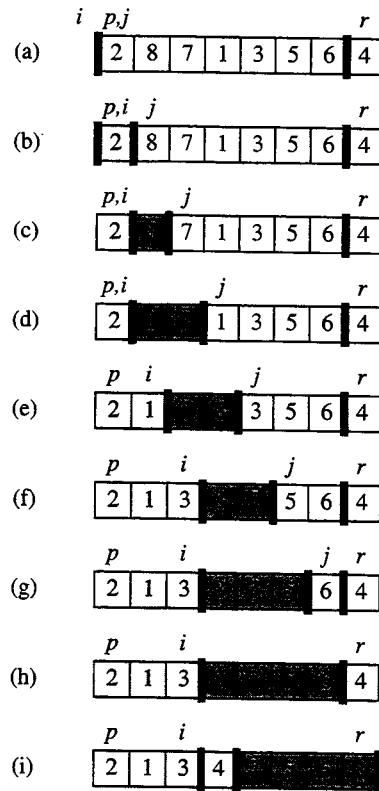


FIGURA 7.1 A operação de PARTITION sobre um exemplo de arranjo. Os elementos do arranjo ligeiramente sombreados estão todos na primeira partição com valores não maiores que  $x$ . Elementos fortemente sombreados estão na segunda partição com valores maiores que  $x$ . Os elementos não sombreados ainda não foram inseridos em uma das duas primeiras partições, e o elemento final branco é o pivô. (a) O arranjo inicial e as configurações de variáveis. Nenhum dos elementos foi inserido em qualquer das duas primeiras partições. (b) O valor 2 é “trocado com ele mesmo” e inserido na partição de valores menores. (c)–(d) Os valores 8 e 7 foram acrescentados à partição de valores maiores. (e) Os valores 1 e 8 são permutados, e a partição menor cresce. (f) Os valores 3 e 8 são permutados, e a partição menor cresce. (g)–(h) A partição maior cresce até incluir 5 e 6 e o loop termina. (i) Nas linhas 7 e 8, o elemento pivô é permutado de forma a residir entre as duas partições

A Figura 7.3(b) mostra o que acontece quando  $A[j] \leq x$ ;  $i$  é incrementado,  $A[i]$  e  $A[j]$  são permutados, e então  $j$  é incrementado. Por causa da troca, agora temos que  $A[i] \leq x$ , e a condição 1 é satisfeita. De modo semelhante, também temos que  $A[j-1] > x$ , pois o item que foi trocado em  $A[j-1]$  é, pelo loop invariante, maior que  $x$ .



FIGURA 7.2 As quatro regiões mantidas pelo procedimento PARTITION em um subarranjo  $A[p..r]$ . Os valores em  $A[p..i]$  são todos menores que ou iguais a  $x$ , os valores em  $A[i+1..j-1]$  são todos maiores que  $x$ , e  $A[r] = x$ . Os valores em  $A[j..r-1]$  podem ser quaisquer valores

**Término:** No término,  $j = r$ . Então, toda entrada no arranjo está em um dos três conjuntos descritos pelo invariante, e particionamos os valores no arranjo em três conjuntos: os que são menores que ou iguais a  $x$ , os maiores que  $x$ , e um conjunto unitário contendo  $x$ .

As duas linhas finais de PARTITION movem o elemento pivô para seu lugar no meio do arranjo, permutando-o com o elemento mais à esquerda que é maior que  $x$ . A saída de PARTITION agora satisfaz às especificações dadas para a etapa de dividir.

O tempo de execução de PARTITION sobre o subarranjo  $A[p..r]$  é  $\Theta(n)$ , onde  $n = r - p + 1$  (ver Exercício 7.1-3).

## Exercícios

### 7.1-1

Usando a Figura 7.1 como modelo, ilustre a operação de PARTITION sobre o arranjo  $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$ .

### 7.1-2

Que valor de  $q$  PARTITION retorna quando todos os elementos no arranjo  $A[p..r]$  têm o mesmo valor? Modifique PARTITION de forma que  $q = (p + r)/2$  quando todos os elementos no arranjo  $A[p..r]$  têm o mesmo valor.

### 7.1-3

Forneça um breve argumento mostrando que o tempo de execução de PARTITION sobre um subarranjo de tamanho  $n$  é  $\Theta(n)$ .

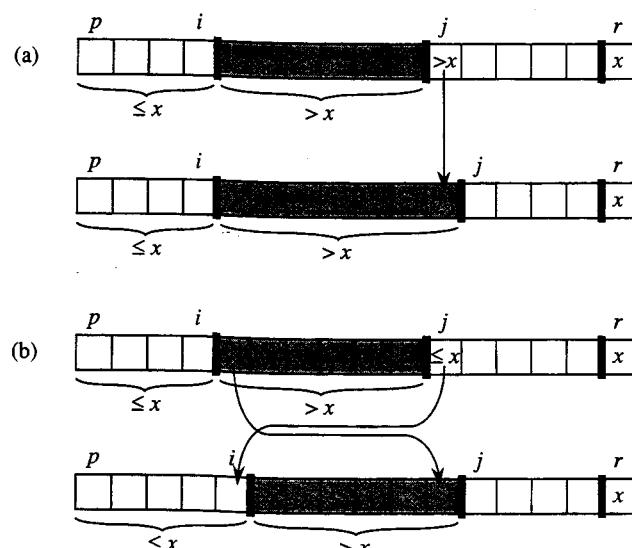


FIGURA 7.3 Os dois casos para uma iteração do procedimento PARTITION. (a) Se  $A[j] > x$ , a única ação é incrementar  $j$ , o que mantém o loop invariante. (b) Se  $A[j] \leq x$ , o índice  $i$  é incrementado,  $A[i]$  e  $A[j]$  são permutados, e então  $j$  é incrementado. Novamente, o loop invariante é mantido

### 7.1-4

De que maneira você modificaria QUICKSORT para fazer a ordenação em ordem não crescente?

## 7.2 O desempenho de quicksort

O tempo de execução de quicksort depende do fato de o particionamento ser balanceado ou não balanceado, e isso por sua vez depende de quais elementos são usados para particionar. Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto a ordenação por intercalação. Contudo, se o particionamento é não balanceado, ele pode ser executado assintoticamente de forma tão lenta quanto a ordenação por inserção. Nesta seção, investigaremos informalmente como o quicksort é executado sob as hipóteses de particionamento balanceado e particionamento não balanceado.

## Particionamento no pior caso

O comportamento do pior caso para o quicksort ocorre quando a rotina de particionamento produz um subproblema com  $n - 1$  elementos e um com 0 elementos. (Essa afirmativa é demonstrada na Seção 7.4.1.) Vamos supor que esse particionamento não balanceado surge em cada chamada recursiva. O particionamento custa o tempo  $\Theta(n)$ . Tendo em vista que a chamada recursiva sobre um arranjo de tamanho 0 simplesmente retorna,  $T(0) = \Theta(1)$ , e a recorrência para o tempo de execução é

$$\begin{aligned}T(n) &= T(n - 1) + T(0) + \Theta(n) \\&= T(n - 1) + \Theta(n).\end{aligned}$$

Intuitivamente, se somarmos os custos incorridos a cada nível da recursão, conseguimos uma série aritmética (equação (A.2)), que tem o valor  $\Theta(n^2)$ . Na realidade, é simples usar o método de substituição para provar que a recorrência  $T(n) = T(n - 1) + \Theta(n)$  tem a solução  $T(n) = \Theta(n^2)$ . (Veja o Exercício 7.2-1.)

Portanto, se o particionamento é não balanceado de modo máximo em cada nível recursivo do algoritmo, o tempo de execução é  $\Theta(n^2)$ . Por conseguinte, o tempo de execução do pior caso do quicksort não é melhor que o da ordenação por inserção. Além disso, o tempo de execução  $\Theta(n^2)$  ocorre quando o arranjo de entrada já está completamente ordenado – uma situação comum na qual a ordenação por inserção é executada no tempo  $O(n)$ .

## Particionamento do melhor caso

Na divisão mais uniforme possível, PARTITION produz dois subproblemas, cada um de tamanho não maior que  $n/2$ , pois um tem tamanho  $\lfloor n/2 \rfloor$  e o outro tem tamanho  $\lceil n/2 \rceil - 1$ . Nesse caso, o quicksort é executado com muito maior rapidez. A recorrência pelo tempo de execução é então

$$T(n) \leq 2T(n/2) + \Theta(n)$$

que, pelo caso 2 do teorema mestre (Teorema 4.1) tem a solução  $T(n) = O(n \lg n)$ . Desse modo, o balanceamento equilibrado dos dois lados da partição em cada nível da recursão produz um algoritmo assintoticamente mais rápido.

## Particionamento balanceado

O tempo de execução do caso médio de quicksort é muito mais próximo do melhor caso que do pior caso, como mostraram as análises da Seção 7.4. A chave para compreender por que isso poderia ser verdade é entender como o equilíbrio do particionamento se reflete na recorrência que descreve o tempo de execução.

Por exemplo, suponha que o algoritmo de particionamento sempre produza uma divisão proporcional de 9 para 1, que a princípio parece bastante desequilibrada. Então, obtemos a recorrência

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

no tempo de execução de quicksort, onde incluímos explicitamente a constante  $c$  oculta no termo  $\Theta(n)$ . A Figura 7.4 mostra a árvore de recursão para essa recorrência. Note que todo nível da árvore tem custo  $cn$ , até ser alcançada uma condição limite à profundidade  $\log_{10} n = \Theta(\lg n)$ , e então os níveis têm o custo máximo  $cn$ . A recursão termina na profundidade  $\log_{10/9} n = \Theta(\lg n)$ . O custo total do quicksort é portanto  $O(n \lg n)$ . Desse modo, com uma divisão na proporção de | 121

9 para 1 em todo nível de recursão, o que intuitivamente parece bastante desequilibrado, o quicksort é executado no tempo  $O(n \lg n)$  – assintoticamente o mesmo tempo que teríamos se a divisão fosse feita exatamente ao meio. De fato, até mesmo uma divisão de 99 para 1 produz um tempo de execução igual a  $O(n \lg n)$ . A razão é que qualquer divisão de proporcionalidade *constante* produz uma árvore de recursão de profundidade  $\Theta(\lg n)$ , onde o custo em cada nível é  $O(n)$ . Portanto, o tempo de execução será então  $O(n \lg n)$  sempre que a divisão tiver proporcionalidade constante.

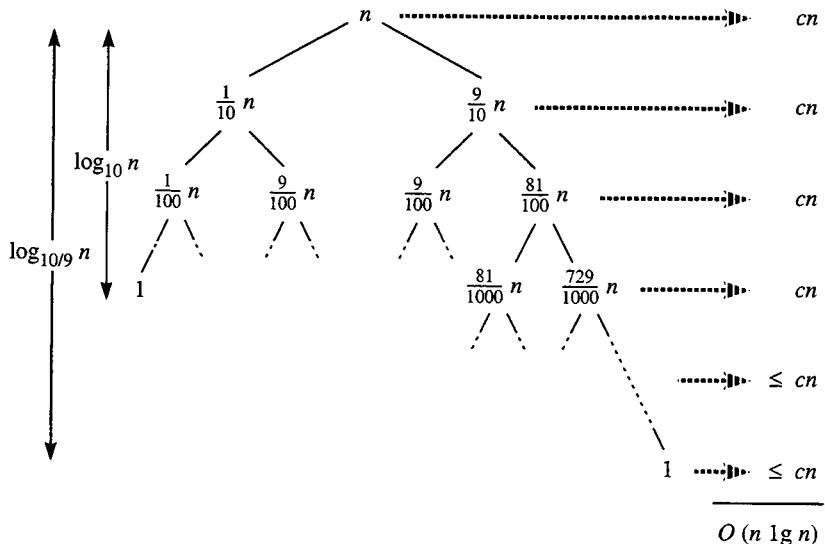


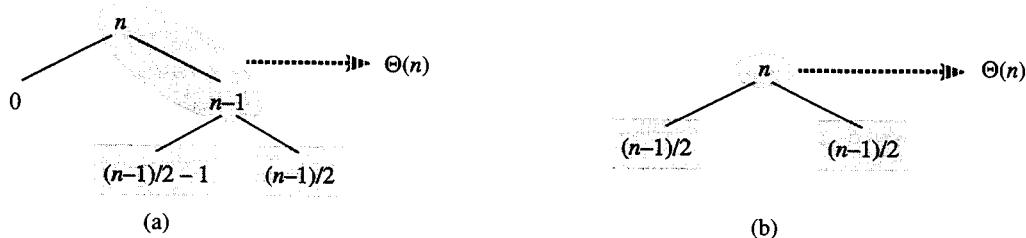
FIGURA 7.4 Uma árvore de recursão para QUICKSORT, na qual PARTITION sempre produz uma divisão de 9 para 1, resultando no tempo de execução  $O(n \lg n)$ . Os nós mostram tamanhos de subproblemas, com custos por nível à direita. Os custos por nível incluem a constante  $c$  implícita no termo  $\Theta(n)$ .

## Intuição para o caso médio

Para desenvolver uma noção clara do caso médio de quicksort, devemos fazer uma suposição sobre a freqüência com que esperamos encontrar as várias entradas. O comportamento de quicksort é determinado pela ordenação relativa dos valores nos elementos do arranjo dados como entrada, e não pelos valores específicos no arranjo. Como em nossa análise probabilística do problema da contratação na Seção 5.2, iremos supor por enquanto que todas as permutações dos números de entrada são igualmente prováveis.

Quando executamos o quicksort sobre um arranjo de entrada aleatório, é improvável que o particionamento sempre ocorra do mesmo modo em todo nível, como nossa análise informal pressupõe. Esperamos que algumas divisões sejam razoavelmente bem equilibradas e que algumas sejam bastante desequilibradas. Por exemplo, o Exercício 7.2-6 lhe pede para mostrar que, em cerca de 80% do tempo, PARTITION produz uma divisão mais equilibrada que 9 para 1, e mais ou menos em 20% do tempo ele produz uma divisão menos equilibrada que 9 para 1.

No caso médio, PARTITION produz uma mistura de divisões “boas” e “ruins”. Em uma árvore de recursão para uma execução do caso médio de PARTITION, as divisões boas e ruins estão distribuídas aleatoriamente ao longo da árvore. Porém, suponha para fins de intuição, que as divisões boas e ruins alternem seus níveis na árvore, e que as divisões boas sejam divisões do melhor caso e as divisões ruins sejam divisões do pior caso. A Figura 7.5(a) mostra as divisões em dois níveis consecutivos na árvore de recursão. Na raiz da árvore, o custo é  $n$  para particionamento e os subarranjos produzidos têm tamanhos  $n - 1$  e 1: o pior caso. No nível seguinte, o subarranjo de tamanho  $n - 1$  é particionado no melhor caso em dois subarranjos de tamanho  $(n - 1)/2 - 1$  e  $(n - 1)/2$ . Vamos supor que o custo da condição limite seja 1 para o subarranjo de tamanho 0.



**FIGURA 7.5** (a) Dois níveis de uma árvore de recursão para quicksort. O particionamento na raiz custa  $n$  e produz uma divisão “ruim”: dois subarranjos de tamanhos 0 e  $n - 1$ . O particionamento do subarranjo de tamanho  $n - 1$  custa  $n - 1$  e produz uma divisão “boa”: subarranjos de tamanho  $(n - 1)/2 - 1$  e  $(n - 1)/2$ . (b) Um único nível de uma árvore de recursão que está muito bem equilibrada. Em ambas as partes, o custo de particionamento para os subproblemas mostrados com sombreamento elíptico é  $\Theta(n)$ . Ainda assim, os subproblemas que restam para serem resolvidos em (a), mostrados com sombreamento retangular, não são maiores que os subproblemas correspondentes que restam para serem resolvidos em (b)

A combinação da divisão ruim seguida pela divisão boa produz três subarranjos de tamanhos 0,  $(n - 1)/2 - 1$  e  $(n - 1)/2$ , a um custo de particionamento combinado de  $\Theta(n) + \Theta(n - 1) = \Theta(n)$ . Certamente, essa situação não é pior que a da Figura 7.5(b), ou seja, um único nível de particionamento que produz dois subarranjos de tamanho  $(n - 1)/2$ , ao custo  $\Theta(n)$ . Ainda assim, essa última situação é equilibrada! Intuitivamente, o custo  $\Theta(n - 1)$  da divisão ruim pode ser absorvido no custo  $\Theta(n)$  da divisão boa, e a divisão resultante é boa. Desse modo, o tempo de execução do quicksort, quando os níveis se alternam entre divisões boas e ruins, é semelhante ao tempo de execução para divisões boas sozinhas: ainda  $O(n \lg n)$ , mas com uma constante ligeiramente maior oculta pela notação de  $O$ . Faremos uma análise rigorosa do caso médio na Seção 7.4.2.

## Exercícios

### 7.2-1

Use o método de substituição para provar que a recorrência  $T(n) = T(n - 1) + \Theta(n)$  tem a solução  $T(n) = \Theta(n^2)$ , como afirmamos no início da Seção 7.2.

### 7.2-2

Qual é o tempo de execução de QUICKSORT quando todos os elementos do arranjo  $A$  têm o mesmo valor?

### 7.2-3

Mostre que o tempo de execução de QUICKSORT é  $\Theta(n^2)$  quando o arranjo  $A$  contém elementos distintos e está ordenado em ordem decrescente.

### 7.2-4

Os bancos freqüentemente registram transações em uma conta na ordem dos horários das transações, mas muitas pessoas gostam de receber seus extratos bancários com os cheques relacionados na ordem de número do cheque. Em geral, as pessoas preenchem seus cheques na ordem do número do cheque, e os comerciantes normalmente os descontam com uma presteza razoável. Portanto, o problema de converter a ordenação pela hora da transação na ordenação pelo número do cheque é o problema de ordenar uma entrada quase ordenada. Demonstre que o procedimento INSERTION-SORT tenderia a superar o procedimento QUICKSORT nesse problema.

### 7.2-5

Suponha que as divisões em todo nível de quicksort estejam na proporção  $1 - \alpha$  para  $\alpha$ , onde  $0 < \alpha \leq 1/2$  é uma constante. Mostre que a profundidade mínima de uma folha na árvore de recursão é aproximadamente  $-\lg n / \lg \alpha$  e a profundidade máxima é aproximadamente  $-\lg n / \lg(1 - \alpha)$ . (Não se preocupe com o arredondamento até inteiro.)

### 7.2-6 \*

Demonstre que, para qualquer constante  $0 < \alpha \leq 1/2$ , a probabilidade de que, em um arranjo de entradas aleatórias, PARTITION produza uma divisão mais equilibrada que  $1 - \alpha$  para  $\alpha$  é aproximadamente  $1 - 2\alpha$ .

## 7.3 Uma versão aleatória de quicksort

Na exploração do comportamento do caso médio de quicksort, fizemos uma suposição de que todas as permutações dos números de entrada são igualmente prováveis. Porém, em uma situação de engenharia nem sempre podemos esperar que ela se mantenha válida. (Ver Exercício 7.2-4.) Como vimos na Seção 5.3, às vezes adicionamos um caráter aleatório a um algoritmo para obter bom desempenho no caso médio sobre todas as entradas. Muitas pessoas consideram a versão aleatória resultante de quicksort o algoritmo de ordenação preferido para entrada grandes o suficiente.

Na Seção 5.3, tornamos nosso algoritmo aleatório permutando explicitamente a entrada. Também poderíamos fazer isso para quicksort, mas uma técnica de aleatoriedade diferente, chamada amostragem aleatória, produz uma análise mais simples. Em vez de sempre usar  $A[r]$  como pivô, usaremos um elemento escolhido ao acaso a partir do subarranjo  $A[p..r]$ . Faremos isso permutando o elemento  $A[r]$  com um elemento escolhido ao acaso de  $A[p..r]$ . Essa modificação, em que fazemos a amostragem aleatória do intervalo  $p, \dots, r$ , assegura que o elemento pivô  $x = A[r]$  tem a mesma probabilidade de ser qualquer um dos  $r-p+1$  elementos do subarranjo. Como o elemento pivô é escolhido ao acaso, esperamos que a divisão do arranjo de entrada seja razoavelmente bem equilibrada na média.

As mudanças em PARTITION e QUICKSORT são pequenas. No novo procedimento de partição, simplesmente implementamos a troca antes do particionamento real:

```
RANDOMIZED-PARTITION( $A, p, r$ )
1  $i \leftarrow \text{RANDOM}(p, r)$ 
2 trocar  $A[p] \leftrightarrow A[i]$ 
3 return PARTITION( $A, p, r$ )
```

O novo quicksort chama RANDOMIZED-PARTITION em lugar de PARTITION:

```
RANDOMIZED-QUICKSORT( $A, p, r$ )
1 if  $p < r$ 
2   then  $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$ 
3     RANDOMIZED-QUICKSORT( $A, p, q - 1$ )
4     RANDOMIZED-QUICKSORT( $A, q + 1, r$ )
```

Analisaremos esse algoritmo na próxima seção.

## Exercícios

### 7.3-1

Por que analisamos o desempenho do caso médio de um algoritmo aleatório e não seu desempenho no pior caso?

### 7.3-2

Durante a execução do procedimento RANDOMIZED-QUICKSORT, quantas chamadas são feitas ao gerador de números aleatórios RANDOM no pior caso? E no melhor caso? Dê a resposta em termos de notação  $\Theta$ .

## 7.4 Análise de quicksort

A Seção 7.2 forneceu alguma intuição sobre o comportamento do pior caso do quicksort e sobre o motivo pelo qual esperamos que ele funcione rapidamente. Nesta seção, analisaremos o comportamento do quicksort de forma mais rigorosa. Começaremos com uma análise do pior caso, que se aplica a QUICKSORT ou a RANDOMIZED-QUICKSORT, e concluiremos com uma análise do caso médio de RANDOMIZED-QUICKSORT.

### 7.4.1 Análise do pior caso

Vimos na Seção 7.2 que uma divisão do pior caso em todo nível de recursão do quicksort produz um tempo de execução igual a  $\Theta(n^2)$  que, intuitivamente, é o tempo de execução do pior caso do algoritmo. Agora, vamos provar essa afirmação.

Usando o método de substituição (ver Seção 4.1), podemos mostrar que o tempo de execução de quicksort é  $O(n^2)$ . Seja  $T(n)$  o tempo no pior caso para o procedimento QUICKSORT sobre uma entrada de tamanho  $n$ . Então, temos a recorrência

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n), \quad (7.1)$$

onde o parâmetro  $q$  varia de 0 a  $n-1$ , porque o procedimento PARTITION produz dois subproblemas com tamanho total  $n-1$ . Supomos que  $T(n) \leq cn^2$  para alguma constante  $c$ . Pela substituição dessa suposição na recorrência (7.1), obtemos

$$\begin{aligned} T(n) &= \max_{0 \leq q \leq n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) + \Theta(n). \end{aligned}$$

A expressão  $q^2 + (n-q-1)^2$  alcança um máximo sobre o intervalo  $0 \leq q \leq n-1$  do parâmetro em um dos pontos extremos, como pode ser visto pelo fato da segunda derivada da expressão com relação a  $q$  ser positiva (ver Exercício 7.4-3). Essa observação nos dá o limite  $\max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) \leq (n-1)^2 = n^2 - 2n + 1$ . Continuando com nossa definição do limite de  $T(n)$ , obtemos

$$\begin{aligned} T(n) &\leq cn^2 - c(2n-1) + \Theta(n) \\ &\leq cn^2, \end{aligned}$$

pois podemos escolher a constante  $c$  grande o suficiente para que o termo  $c(2n-1)$  domine o termo  $\Theta(n)$ . Portanto,  $T(n) = O(n^2)$ . Vimos na Seção 7.2 um caso específico em que quicksort demora o tempo  $\Omega(n^2)$ : quando o particionamento é desequilibrado. Como alternativa, o Exercício 7.4-1 lhe pede para mostrar que a recorrência (7.1) tem uma solução  $T(n) = \Omega(n^2)$ . Desse modo, o tempo de execução (no pior caso) de quicksort é  $\Theta(n^2)$ .

### 7.4.2 Tempo de execução esperado

Já fornecemos um argumento intuitivo sobre o motivo pelo qual o tempo de execução do caso médio de RANDOMIZED-QUICKSORT é  $O(n \lg n)$ : se, em cada nível de recursão, a divisão induzida por RANDOMIZED-PARTITION colocar qualquer fração constante dos elementos em um lado da partição, então a árvore de recursão terá a profundidade  $\Theta(\lg n)$ , e o trabalho  $O(n)$  será

executado em cada nível. Ainda que sejam adicionados novos níveis com a divisão mais desequilibrada. Possível entre esses níveis, o tempo total continuará a ser  $O(n \lg n)$ . Podemos analisar o tempo de execução esperado de RANDOMIZED-QUICKSORT com precisão, compreendendo primeiro como o procedimento de particionamento opera, e depois usando essa compreensão para derivar um limite  $O(n \lg n)$  sobre o tempo de execução esperado. Esse limite superior no tempo de execução esperado, combinado com o limite no melhor caso  $\Theta(n \lg n)$  que vimos na Seção 7.2, resulta em um tempo de execução esperado  $\Theta(n \lg n)$ .

## Tempo de execução e comparações

O tempo de execução de QUICKSORT é dominado pelo tempo gasto no procedimento PARTITION. Toda vez que o procedimento PARTITION é chamado, um elemento pivô é selecionado, e esse elemento nunca é incluído em quaisquer chamadas recursivas futuras a QUICKSORT e PARTITION. Desse modo, pode haver no máximo  $n$  chamadas a PARTITION durante a execução inteira do algoritmo de quicksort. Uma chamada a PARTITION demora o tempo  $O(1)$  mais um período de tempo proporcional ao número de iterações do loop for das linhas 3 a 6. Cada iteração desse loop for executa uma comparação na linha 4, comparando o elemento pivô a outro elemento do arranjo  $A$ . Assim, se pudermos contar o número total de vezes que a linha 4 é executada, poderemos limitar o tempo total gasto no loop for durante toda a execução de QUICKSORT.

### Lema 7.1

Seja  $X$  o número de comparações executadas na linha 4 de PARTITION por toda a execução de QUICKSORT sobre um arranjo de  $n$  elementos. Então, o tempo de execução de QUICKSORT é  $O(n + X)$ .

**Prova** Pela discussão anterior, existem  $n$  chamadas a PARTITION, cada uma das quais faz uma proporção constante do trabalho e depois executa o loop for um certo número de vezes. Cada iteração do loop for executa a linha 4. ■

Portanto, nossa meta é calcular  $X$ , o número total de comparações executadas em todas as chamadas a PARTITION. Não tentaremos analisar quantas comparações são feitas em cada chamada a PARTITION. Em vez disso, derivaremos um limite global sobre o número total de comparações. Para fazê-lo, devemos reconhecer quando o algoritmo compara dois elementos do arranjo e quando ele não o faz. Para facilitar a análise, renomeamos os elementos do arranjo  $A$  como  $z_1, z_2, \dots, z_n$ , com  $z_i$  sendo o  $i$ -ésimo menor elemento. Também definimos o conjunto  $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$  como o conjunto de elementos entre  $z_i$  e  $z_j$ , inclusive.

Quando o algoritmo compara  $z_i$  e  $z_j$ ? Para responder a essa pergunta, primeiro observamos que cada par de elementos é comparado no máximo uma vez. Por quê? Os elementos são comparados apenas ao elemento pivô e, depois que uma chamada específica de PARTITION termina, o elemento pivô usado nessa chamada nunca é comparado novamente a quaisquer outros elementos.

Nossa análise utiliza variáveis indicadoras aleatórias (consulte a Seção 5.2). Definimos

$$X_{ij} = I\{z_i \text{ é comparado a } z_j\},$$

onde estamos considerando se a comparação tem lugar em qualquer instante durante a execução do algoritmo, não apenas durante uma iteração ou uma chamada de PARTITION. Tendo em vista que cada par é comparado no máximo uma vez, podemos caracterizar facilmente o número total de comparações executadas pelo algoritmo:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}.$$

Tomando as expectativas em ambos os lados, e depois usando a linearidade de expectativa e o Lema 5.1, obtemos

$$\begin{aligned}
 E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] \\
 &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] \\
 &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr\{z_i \text{ é comparado a } z_j\}. \tag{7.2}
 \end{aligned}$$

Resta calcular  $\Pr\{z_i \text{ é comparado a } z_j\}$ . Nossa análise parte do princípio de que cada pivô é escolhido ao acaso e de forma independente.

É útil imaginar quando dois itens *não* são comparados. Considere uma entrada para quick-sort dos números 1 a 10 (em qualquer ordem) e suponha que o primeiro elemento pivô seja 7. Então, a primeira chamada a PARTITION separa os números em dois conjuntos: {1, 2, 3, 4, 5, 6} e {8, 9, 10}. Fazendo-se isso, o elemento pivô 7 é comparado a todos os outros elementos, mas nenhum número do primeiro conjunto (por exemplo, 2) é ou jamais será comparado a qualquer número do segundo conjunto (por exemplo, 9).

Em geral, uma vez que um pivô  $x$  é escolhido com  $z_i < x < z_j$ , sabemos que  $z_i$  e  $z_j$  não podem ser comparados em qualquer momento subsequente. Se, por outro lado,  $z_i$  for escolhido como um pivô antes de qualquer outro item em  $Z_{ij}$ , então  $z_i$  será comparado a cada item em  $Z_{ij}$ , exceto ele mesmo. De modo semelhante, se  $z_j$  for escolhido como pivô antes de qualquer outro item em  $Z_{ij}$ , então  $z_j$  será comparado a cada item em  $Z_{ij}$ , exceto ele próprio. Em nosso exemplo, os valores 7 e 9 são comparados porque 7 é o primeiro item de  $Z_{7,9}$  a ser escolhido como pivô. Em contraste, 2 e 9 nunca serão comparados, porque o primeiro elemento pivô escolhido de  $Z_{2,9}$  é 7. Desse modo,  $z_i$  e  $z_j$  são comparados se e somente se o primeiro elemento a ser escolhido como pivô de  $Z_{ij}$  é  $z_i$  ou  $z_j$ .

Agora, calculamos a probabilidade de que esse evento ocorra. Antes do ponto em que um elemento de  $Z_{ij}$  é escolhido como pivô, todo o conjunto  $Z_{ij}$  está reunido na mesma partição. Por conseguinte, qualquer elemento de  $Z_{ij}$  tem igual probabilidade de ser o primeiro escolhido como pivô. Pelo fato do conjunto  $Z_{ij}$  ter  $j - i + 1$  elementos e tendo em vista que os pivôs são escolhidos ao acaso e de forma independente, a probabilidade de qualquer elemento dado ser o primeiro escolhido como pivô é  $1/(j - i + 1)$ . Desse modo, temos

$$\begin{aligned}
 \Pr\{z_i \text{ é comparado a } z_j\} &= \Pr\{z_i \text{ ou } z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \tag{7.3} \\
 &= \Pr\{z_i \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\
 &\quad + \Pr\{z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\
 &= \frac{1}{j - i + 1} + \frac{1}{j - i + 1} \\
 &= \frac{2}{j - i + 1}.
 \end{aligned}$$

A segunda linha se segue porque os dois eventos são mutuamente exclusivos. Combinando as equações (7.2) e (7.3), obtemos

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1}.$$

Podemos avaliar essa soma usando uma troca de variáveis ( $k = j - i$ ) e o limite sobre a série harmônica na equação (A.7):

$$\begin{aligned}
 E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\
 &= \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k+1} \\
 &< \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} \\
 &= \sum_{i=1}^{n-1} O(\lg n) \\
 &= O(n \lg n). \tag{7.4}
 \end{aligned}$$

Desse modo concluímos que, usando-se RANDOMIZED-PARTITION, o tempo de execução esperado de quicksort é  $O(n \lg n)$ .

## Exercícios

### 7.4-1

Mostre que, na recorrência

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n),$$

$$T(n) = \Omega(n^2).$$

### 7.4-2

Mostre que o tempo de execução do quicksort no melhor caso é  $\Omega(n \lg n)$ .

### 7.4-3

Mostre que  $q^2 + (n-q-1)^2$  alcança um máximo sobre  $q = 0, 1, \dots, n-1$  quando  $q = 0$  ou  $q = n-1$ .

### 7.4-4

Mostre que o tempo de execução esperado do procedimento RANDOMIZED-QUICKSORT é  $\Omega(n \lg n)$ .

### 7.4-5

O tempo de execução do quicksort pode ser melhorado na prática, aproveitando-se o tempo de execução muito pequeno da ordenação por inserção quando sua entrada se encontra “quase” ordenada. Quando o quicksort for chamado em um subarranjo com menos de  $k$  elementos, deixe-o simplesmente retornar sem ordenar o subarranjo. Após o retorno da chamada de alto nível a quicksort, execute a ordenação por inserção sobre o arranjo inteiro, a fim de concluir o processo de ordenação. Mostre que esse algoritmo de ordenação é executado no tempo esperado  $O(nk + n \lg(n/k))$ . Como  $k$  deve ser escolhido, tanto na teoria quanto na prática?

### 7.4-6 \*

Considere a modificação do procedimento PARTITION pela escolha aleatória de três elementos do arranjo  $A$  e pelo particionamento sobre sua mediana (o valor médio dos três elementos). Faça a aproximação da probabilidade de se obter na pior das hipóteses uma divisão de  $\alpha$  para  $(1-\alpha)$ , como uma função de  $\alpha$  no intervalo  $0 < \alpha < 1$ .

## Problemas

### 7-1 Correção da partição de Hoare

A versão de PARTITION dada neste capítulo não é o algoritmo de particionamento original. Aqui está o algoritmo de partição original, devido a T. Hoare:

**HOARE-PARTITION( $A, p, r$ )**

```

1  $x \leftarrow A[p]$ 
2  $i \leftarrow p - 1$ 
3  $j \leftarrow r + 1$ 
4 while TRUE
5   do repeat  $j \leftarrow j - 1$ 
6     until  $A[j] \leq x$ 
7     repeat  $i \leftarrow i + 1$ 
8     until  $A[i] \geq x$ 
9     if  $i < j$ 
10    then trocar  $A[i] \leftrightarrow A[j]$ 
11    else return  $j$ 

```

- a. Demonstre a operação de HOARE-PARTITION sobre o arranjo  $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$ , mostrando os valores do arranjo e os valores auxiliares depois de cada iteração do loop **for** das linhas 4 a 11.

As três perguntas seguintes lhe pedem para apresentar um argumento cuidadoso de que o procedimento HOARE-PARTITION é correto. Prove que:

- b. Os índices  $i$  e  $j$  são tais que nunca acessamos um elemento de  $A$  fora do subarranjo  $A[p..r]$ .
- c. Quando HOARE-PARTITION termina, ele retorna um valor  $j$  tal que  $p \leq j < r$ .
- d. Todo elemento de  $A[p..j]$  é menor que ou igual a todo elemento de  $A[j+1..r]$  quando HOARE-PARTITION termina.

O procedimento PARTITION da Seção 7.1 separa o valor do pivô (originalmente em  $A[r]$ ) das duas partições que ele forma. Por outro lado, o procedimento HOARE-PARTITION sempre insere o valor do pivô (originalmente em  $A[p]$ ) em uma das duas partições  $A[p..j]$  e  $A[j+1..r]$ . Como  $p \leq j < r$ , essa divisão é sempre não trivial.

- e. Reescreva o procedimento QUICKSORT para usar HOARE-PARTITION.

### 7-2 Análise alternativa de quicksort

Uma análise alternativa do tempo de execução de quicksort aleatório se concentra no tempo de execução esperado de cada chamada recursiva individual a QUICKSORT, em vez de se ocupar do número de comparações executadas.

- a. Demonstre que, dado um arranjo de tamanho  $n$ , a probabilidade de que qualquer elemento específico seja escolhido como pivô é  $1/n$ . Use isso para definir variáveis indicadoras aleatórias  $X_i = I\{i\text{-ésimo menor elemento é escolhido como pivô}\}$ . Qual é  $E[X_i]$ ?
- b. Seja  $T(n)$  uma variável aleatória denotando o tempo de execução de quicksort sobre um arranjo de tamanho  $n$ . Demonstre que

$$E[T(n)] = E\left[\sum_{q=1}^n X_q (T(q-1) + T(n-q) + \Theta(n))\right]. \quad (7.5)$$

- c. Mostre que a equação (7.5) é simplificada para

$$E[T(n)] = \frac{2}{n} \sum_{q=0}^{n-1} E[T(q)] + \Theta(n) . \quad (7.6)$$

d. Mostre que

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 . \quad (7.7)$$

(Sugestão: Divida o somatório em duas partes, uma para  $k = 1, 2, \dots, \lceil n/2 \rceil - 1$  e uma para  $k = \lceil n/2 \rceil, \dots, n - 1$ .)

- e. Usando o limite da equação (7.7), mostre que a recorrência na equação (7.6) tem a solução  $E[T(n)] = \Theta(n \lg n)$ . (Sugestão: Mostre, por substituição, que  $E[T(n)] \leq an \log n - bn$  para algumas constantes positivas  $a$  e  $b$ .)

### 7-3 Ordenação do pateta

Os professores Howard, Fine e Howard propuseram o seguinte algoritmo de ordenação “ele-gante”:

```
STOOGE-SORT( $A, i, j$ )
1 if  $A[i] > A[j]$ 
2   then trocar  $A[i] \leftrightarrow A[j]$ 
3   if  $i + 1 \geq j$ 
4     then return
5    $k \leftarrow \lfloor (j - i + 1)/3 \rfloor$             $\triangleright$  Arredonda para menos.
6   STOOGE-SORT( $A, i, j - k$ )                  $\triangleright$  Primeiros dois terços.
7   STOOGE-SORT( $A, i + k, j$ )                  $\triangleright$  Últimos dois terços.
8   STOOGE-SORT( $A, i, j - k$ )                  $\triangleright$  Primeiros dois terços novamente.
```

- a. Mostre que, se  $n = \text{comprimento}[A]$ , então STOOGE-SORT( $A, 1, \text{comprimento}[A]$ ) ordena corretamente o arranjo de entrada  $A[1 .. n]$ .
- b. Forneça uma recorrência para o tempo de execução no pior caso de STOOGE-SORT e um limite assintótico restrito (notação  $\Theta$ ) sobre o tempo de execução no pior caso.
- c. Compare o tempo de execução no pior caso de STOOGE-SORT com o da ordenação por inserção, da ordenação por intercalação, de heapsort e de quicksort. Os professores merecem a estabilidade no emprego?

### 7-4 Profundidade de pilha para quicksort

O algoritmo QUICKSORT da Seção 7.1 contém duas chamadas recursivas a ele próprio. Após a chamada a PARTITION, o subarranjo da esquerda é ordenado recursivamente, e depois o subarranjo da direita é ordenado recursivamente. A segunda chamada recursiva em QUICKSORT não é realmente necessária; ela pode ser evitada pelo uso de uma estrutura de controle iterativa. Essa técnica, chamada **recursão do final**, é automaticamente fornecida por bons compiladores. Considere a versão de quicksort a seguir, que simula a recursão do final.

```
QUICKSORT'( $A, p, r$ )
1 while  $p < r$ 
2   do  $\triangleright$  Particiona e ordena o subarranjo esquerdo
3      $q \leftarrow \text{PARTITION}(A, p, r)$ 
4     QUICKSORT'( $A, p, q - 1$ )
5      $p \leftarrow q + 1$ 
```

- a. Mostre que  $\text{QUICKSORT}'(A, 1, \text{comprimento}[A])$  ordena corretamente o arranjo  $A$ .

Os compiladores normalmente executam procedimentos recursivos usando uma *pilha* que contém informações pertinentes, inclusive os valores de parâmetros, para cada chamada recursiva. As informações para a chamada mais recente estão na parte superior da pilha, e as informações para a chamada inicial encontram-se na parte inferior. Quando um procedimento é invocado, suas informações são *empurradas* sobre a pilha; quando ele termina, suas informações são *extraídas*. Tendo em vista nossa suposição de que os parâmetros de arranjos são na realidade representados por ponteiros, as informações correspondentes a cada chamada de procedimento na pilha exigem o espaço de pilha  $O(1)$ . A *profundidade de pilha* é a quantidade máxima de espaço da pilha usado em qualquer instante durante uma computação.

- b. Descreva um cenário no qual a profundidade de pilha de  $\text{QUICKSORT}'$  é  $\Theta(n)$  sobre um arranjo de entrada de  $n$  elementos.
- c. Modifique o código de  $\text{QUICKSORT}'$  de tal modo que a profundidade de pilha no pior caso seja  $\Theta(\lg n)$ . Mantenha o tempo de execução esperado  $O(n \lg n)$  do algoritmo.

### 7-5 Partição de mediana de 3

Um modo de melhorar o procedimento RANDOMIZED-QUICKSORT é criar uma partição em torno de um elemento  $x$  escolhido com maior cuidado que a simples escolha de um elemento aleatório do subarranjo. Uma abordagem comum é o método da *mediana de 3*: escolha  $x$  como a mediana (o elemento intermediário) de um conjunto de 3 elementos selecionados aleatoriamente a partir do subarranjo. Para esse problema, vamos supor que os elementos no arranjo de entrada  $A[1..n]$  sejam distintos e que  $n \geq 3$ . Denotamos o arranjo de saída ordenado por  $A'[1..n]$ . Usando o método da mediana de 3 para escolher o elemento pivô  $x$ , defina  $p_i = \Pr\{x = A'[i]\}$ .

- a. Dê uma fórmula exata para  $p_i$  como uma função de  $n$  e  $i$  para  $i = 2, 3, \dots, n - 1$ . (Observe que  $p_1 = p_n = 0$ .)
- b. Por qual valor aumentamos a probabilidade de escolher  $x = A'[\lfloor (n + 1)/2 \rfloor]$ , a mediana de  $A[1..n]$ , em comparação com a implementação comum? Suponha que  $n \rightarrow \infty$  e forneça a razão de limitação dessas probabilidades.
- c. Se definimos uma “boa” divisão com o significado de escolher o pivô como  $x = A'[i]$ , onde  $n/3 \leq i \leq 2n/3$ , por qual quantidade aumentamos a probabilidade de se obter uma boa divisão em comparação com a implementação comum? (Sugestão: Faça uma aproximação da soma por uma integral.)
- d. Mostre que, no tempo de execução  $\Omega(n \lg n)$  de quicksort, o método da mediana de 3 só afeta o fator constante.

### 7-6 Ordenação nebulosa de intervalos

Considere um problema de ordenação no qual os números não são conhecidos exatamente. Em vez disso, para cada número, conhecemos um intervalo sobre a linha real a que ele pertence. Ou seja, temos  $n$  intervalos fechados da forma  $[a_i, b_i]$ , onde  $a_i \leq b_i$ . O objetivo é fazer a *ordenação nebulosa* desses intervalos, isto é, produzir uma permutação  $\langle i_1, i_2, \dots, i_n \rangle$  dos intervalos tal que existe  $c_j \in [a_{i_j}, b_{i_j}]$  que satisfaça a  $c_1 \leq c_2 \leq \dots \leq c_n$ .

- a. Projete um algoritmo para ordenação do pateta de  $n$  intervalos. Seu algoritmo devia ter a estrutura geral de um algoritmo que faz o quicksort das extremidades esquerdas (os valores  $a_i$ ), mas deve tirar proveito da sobreposição de intervalos para melhorar o tempo de execução. (À medida que os intervalos se sobrepõem cada vez mais, o problema de fazer a ordenação do pateta dos intervalos fica cada vez mais fácil. Seu algoritmo deve aproveitar tal sobreposição, desde que ela exista.)

- b. Demonstre que seu algoritmo é executado no tempo esperado  $\Theta(n \lg n)$  em geral, mas funciona no tempo esperado  $\Theta(n)$  quando todos os intervalos se sobrepõem (isto é, quando existe um valor  $x$  tal que  $x \in [a_i, b_i]$  para todo  $i$ ). O algoritmo não deve verificar esse caso de forma explícita; em vez disso, seu desempenho deve melhorar naturalmente à medida que aumentar a proporção de sobreposição.

## Notas do capítulo

O procedimento quicksort foi criado por Hoare [147]; a versão de Hoare aparece no Problema 7-1. O procedimento PARTITION dado na Seção 7.1 se deve a N. Lomuto. A análise da Seção 7.4 se deve a Avrim Blum, Sedgewick [268] e Bentley [40] fornecem uma boa referência sobre os detalhes de implementação e como eles são importantes.

McIlroy [216] mostrou como engenheiro um “adversário matador” que produz um arranjo sobre o qual virtualmente qualquer implementação de quicksort demora o tempo  $\Theta(n^2)$ . Se a implementação é aleatória, o adversário produz o arranjo depois de ver as escolhas ao acaso do algoritmo de quicksort.

---

# *Capítulo 8*

## *Ordenação em tempo linear*

Apresentamos até agora diversos algoritmos que podem ordenar  $n$  números no tempo  $O(n \lg n)$ . A ordenação por intercalação e o heapsort alcançam esse limite superior no pior caso; quicksort o alcança na média. Além disso, para cada um desses algoritmos, podemos produzir uma seqüência de  $n$  números de entrada que faz o algoritmo ser executado no tempo  $\Omega(n \lg n)$ .

Esse algoritmos compartilham uma propriedade interessante: *a seqüência ordenada que eles determinam se baseia apenas em comparações entre os elementos de entrada*. Chamamos esses algoritmos de ordenação de **ordenações por comparação**. Todos os algoritmos de ordenação apresentados até agora são portanto ordenações por comparação.

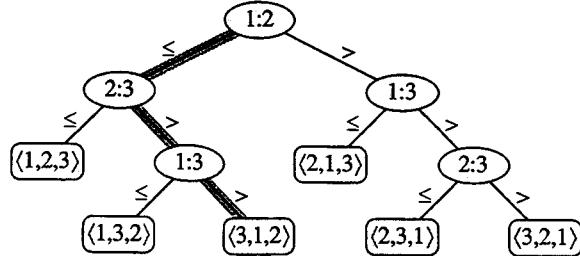
Na Seção 8.1, provaremos que qualquer ordenação por comparação deve efetuar  $\Omega(n \lg n)$  comparações no pior caso para ordenar  $n$  elementos. Desse modo, a ordenação por intercalação e heapsort são assintoticamente ótimas, e não existe nenhuma ordenação por comparação que seja mais rápida por mais de um fator constante.

As Seções 8.2, 8.3 e 8.4 examinam três algoritmos de ordenação – ordenação por contagem, radix sort (ordenação da raiz) e bucket sort (ordenação por balde) – que são executados em tempo linear. É desnecessário dizer que esses algoritmos utilizam outras operações diferentes de comparações para determinar a seqüência ordenada. Em consequência disso, o limite inferior  $\Omega(n \lg n)$  não se aplica a eles.

### **8.1 Limites inferiores para ordenação**

Em uma ordenação por comparação, usamos apenas comparações entre elementos para obter informações de ordem sobre uma seqüência de entrada  $\langle a_1, a_2, \dots, a_n \rangle$ . Ou seja, dados dois elementos  $a_i$  e  $a_j$ , executamos um dos testes  $a_i < a_j$ ,  $a_i \leq a_j$ ,  $a_i = a_j$ ,  $a_i \geq a_j$  ou  $a_i > a_j$ , para determinar sua ordem relativa. Podemos inspecionar os valores dos elementos ou obter informações de ordem sobre eles de qualquer outro modo.

Nesta seção, vamos supor sem perda de generalidade que todos os elementos de entrada são distintos. Dada essa hipótese, as comparações da forma  $a_i = a_j$  são inúteis; assim, podemos supor que não é feita nenhuma comparação dessa forma. Também observamos que as comparações  $a_i \leq a_j$ ,  $a_i \geq a_j$ ,  $a_i > a_j$  e  $a_i < a_j$  são todas equivalentes, em virtude de produzirem informações idênticas sobre a ordem relativa de  $a_i$  e  $a_j$ . Por essa razão, vamos supor que todas as comparações têm a forma  $a_i \leq a_j$ .



**FIGURA 8.1** A árvore de decisão para ordenação por inserção, operando sobre três elementos. Um nó interno anotado por  $i:j$  indica uma comparação entre  $a_i$  e  $a_j$ . Uma folha anotada pela permutação  $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$  indica a ordenação  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ . O caminho sombreado indica as decisões tomadas durante a ordenação da seqüência de entrada  $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$ ; a permutação  $(3, 1, 2)$  na folha indica que a seqüência ordenada é  $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$ . Existem  $3! = 6$  permutações possíveis dos elementos de entrada; assim, a árvore de decisão deve ter no mínimo 6 folhas.

## O modelo de árvore de decisão

As ordenações por comparação podem ser vistas de modo abstrato em termos de *árvores de decisão*. Uma árvore de decisão é uma árvore binária cheia que representa as comparações executadas por um algoritmo de ordenação quando ele opera sobre uma entrada de um tamanho dado. Controle, movimentação de dados e todos os outros aspectos do algoritmo são ignorados. A Figura 8.1 mostra a árvore de decisão correspondente ao algoritmo de ordenação por inserção da Seção 2.1, operando sobre uma seqüência de entrada de três elementos.

Em uma árvore de decisão, cada nó interno é anotado por  $i:j$  para algum  $i$  e  $j$  no intervalo  $1 \leq i, j \leq n$ , onde  $n$  é o número de elementos na seqüência de entrada. Cada folha é anotada por uma permutação  $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ . (Consulte a Seção C.1 para adquirir experiência em permutações.) A execução do algoritmo de ordenação corresponde a traçar um caminho desde a raiz da árvore de decisão até uma folha. Em cada nó interno, é feita uma comparação  $a_i \leq a_j$ . A subárvore da esquerda determina então comparações subsequentes para  $a_i \leq a_j$ , e a subárvore da direita determina comparações subsequentes para  $a_i > a_j$ . Quando chegamos a uma folha, o algoritmo de ordenação estabelece a ordem  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ . Como qualquer algoritmo de ordenação correto deve ser capaz de produzir cada permutação de sua entrada, uma condição necessária para uma ordenação por comparação ser correta é que cada uma das  $n!$  permutações sobre  $n$  elementos deve aparecer como uma das folhas da árvore de decisão, e que cada uma dessas folhas deve ser acessível a partir da raiz por um caminho correspondente a uma execução real da ordenação por comparação. (Iremos nos referir a tais folhas como “acessíveis”.) Desse modo, consideraremos apenas árvores de decisão em que cada permutação aparece como uma folha acessível.

## Um limite inferior para o pior caso

O comprimento do caminho mais longo da raiz de uma árvore de decisão até qualquer de suas folhas acessíveis representa o número de comparações do pior caso que o algoritmo de ordenação correspondente executa. Conseqüentemente, o número de comparações do pior caso para um dado algoritmo de ordenação por comparação é igual à altura de sua árvore de decisão. Um limite inferior sobre as alturas de todas as árvores de decisão em que cada permutação aparece como uma folha acessível é portanto um limite inferior sobre o tempo de execução de qualquer algoritmo de ordenação por comparação. O teorema a seguir estabelece esse limite inferior.

### **Teorema 8.1**

Qualquer algoritmo de ordenação por comparação exige  $\Omega(n \lg n)$  comparações no pior caso.

**Prova** Da discussão precedente, basta determinar a altura de uma árvore de decisão em que cada permutação aparece como uma folha acessível. Considere uma árvore de decisão de altura  $b$  com  $l$  folhas acessíveis correspondente a uma ordenação por comparação sobre  $n$  elementos. Como cada uma das  $n!$  permutações da entrada aparece como alguma folha, temos  $n! \leq l$ . Tendo em vista que uma árvore binária de altura  $b$  não tem mais de  $2^b$  folhas, temos

$$n! \leq l \leq 2^b,$$

que, usando-se logaritmos, implica

$$\begin{aligned} b &\geq \lg(n!) && \text{(pois a função } \lg \text{ é monotonicamente crescente)} \\ &= \Omega(n \lg n) && \text{(pela equação (3.18)) .} \end{aligned}$$

■

### Corolário 8.2

O heapsort e a ordenação por intercalação são ordenações por comparação assintoticamente ótimas.

**Prova** Os  $O(n \lg n)$  limites superiores sobre os tempos de execução para heapsort e ordenação por intercalação correspondem ao limite inferior  $\Omega(n \lg n)$  do pior caso do Teorema 8.1. ■

## Exercícios

### 8.1-1

Qual é a menor profundidade possível de uma folha em uma árvore de decisão para uma ordenação por comparação?

### 8.1-2

Obtenha limites assintoticamente restritos sobre  $\lg(n!)$  sem usar a aproximação de Stirling. Em vez disso, avalie o somatório  $\sum_{k=1}^n \lg k$ , empregando técnicas da Seção A.2.

### 8.1-3

Mostre que não existe nenhuma ordenação por comparação cujo tempo de execução seja linear para pelo menos metade das  $n!$  entradas de comprimento  $n$ . E no caso de uma fração  $1/n$  das entradas de comprimento  $n$ ? E no caso de uma fração  $1/2^n$ ?

### 8.1-4

Você recebeu uma seqüência de  $n$  elementos para ordenar. A seqüência de entrada consiste em  $n/k$  subseqüências, cada uma contendo  $k$  elementos. Os elementos em uma dada subseqüência são todos menores que os elementos na subseqüência seguinte e maiores que os elementos na subseqüência precedente. Desse modo, tudo que é necessário para ordenar a seqüência inteira de comprimento  $n$  é ordenar os  $k$  elementos em cada uma das  $n/k$  subseqüências. Mostre um limite inferior  $\Omega(n \lg k)$  sobre o número de comparações necessárias para resolver essa variação do problema de ordenação. (Sugestão: Não é rigoroso simplesmente combinar os limites inferiores para as subseqüências individuais.)

## 8.2 Ordenação por contagem

A *ordenação por contagem* pressupõe que cada um dos  $n$  elementos de entrada é um inteiro no intervalo de 1 a  $k$ , para algum inteiro  $k$ . Quando  $k = O(n)$ , a ordenação é executada no tempo  $\Theta(n)$ .

A idéia básica da ordenação por contagem é determinar, para cada elemento de entrada  $x$ , o número de elementos menores que  $x$ . Essa informação pode ser usada para inserir o elemento  $x$

diretamente em sua posição no arranjo de saída. Por exemplo, se há 17 elementos menores que  $x$ , então  $x$  é colocado na posição de saída 18. Esse esquema deve ser ligeiramente modificado para manipular a situação na qual vários elementos têm o mesmo valor, pois não queremos inserir todos eles na mesma posição.

No código para ordenação por contagem, partimos da suposição de que a entrada é um arranjo  $A[1 .. n]$  e, portanto,  $\text{comprimento}[A] = n$ . Exigimos dois outros arranjos: o arranjo  $B[1 .. n]$  contém a saída ordenada, e o arranjo  $C[0 .. k]$  fornece um espaço de armazenamento de trabalho temporário.

#### COUNTING-SORT( $A, B, k$ )

```

1 for  $i \leftarrow 0$  to  $k$ 
2   do  $C[i] \leftarrow 0$ 
3 for  $j \leftarrow 1$  to  $\text{comprimento}[A]$ 
4   do  $C[A[j]] \leftarrow C[A[j]] + 1$ 
5 ▷ Agora  $C[i]$  contém o número de elementos iguais a  $i$ .
6 for  $i \leftarrow 2$  to  $k$ 
7   do  $C[i] \leftarrow C[i] + C[i - 1]$ 
8 ▷ Agora  $C[i]$  contém o número de elementos menores que ou iguais a  $i$ .
9 for  $j \leftarrow \text{comprimento}[A]$  downto 1
10  do  $B[C[A[j]]] \leftarrow A[j]$ 
11    $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

	1	2	3	4	5	6	7	8
$A$	2	5	3	0	2	3	0	3
	0	1	2	3	4	5		
$C$	2	0	2	3	0	1		

(a)

	0	1	2	3	4	5
$C$	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
$B$								3
	0	1	2	3	4	5		
$C$	2	2	4	6	7	8		

(c)

	1	2	3	4	5	6	7	8
$B$		0						3
	0	1	2	3	4	5		
$C$	1	2	4	6	7	8		

(d)

	1	2	3	4	5	6	7	8
$B$		0				3	3	
	0	1	2	3	4	5		
$C$	1	2	4	5	7	8		

(e)

	1	2	3	4	5	6	7	8
$B$	0	0	2	2	3	3	3	5

(f)

FIGURA 8.2 A operação de COUNTING-SORT sobre um arranjo de entrada  $A[1 .. 8]$ , onde cada elemento de  $A$  é um inteiro não negativo não maior que  $k = 5$ . (a) O arranjo  $A$  e o arranjo auxiliar  $C$  após a linha 4. (b) O arranjo  $C$  após a linha 7. (c)–(e) O arranjo de saída  $B$  e o arranjo auxiliar  $C$  após uma, duas e três iterações do loop nas linhas 9 a 11, respectivamente. Apenas os elementos levemente sombreados do arranjo  $B$  foram preenchidos. (f) O arranjo de saída final ordenado  $B$

A ordenação por contagem é ilustrada na Figura 8.2. Após a inicialização no loop **for** das linhas 1 e 2, inspecionamos cada elemento de entrada no loop **for** das linhas 3 e 4. Se o valor de um elemento de entrada é  $i$ , incrementamos  $C[i]$ . Desse modo, depois da linha 4,  $C[i]$  contém um número de elementos de entrada igual a  $i$  para cada inteiro  $i = 0, 1, \dots, k$ . Nas linhas 6 e 7 determinamos, para cada  $i = 0, 1, \dots, k$ , quantos elementos de entrada são menores que ou iguais a  $i$ ; mantendo uma soma atualizada do arranjo  $C$ .

Finalmente, no loop **for** das linhas 9 a 11, colocamos cada elemento  $A[j]$  em sua posição ordenada correta no arranjo de saída  $B$ . Se todos os  $n$  elementos forem distintos, então, quando entrarmos pela primeira vez na linha 9, para cada  $A[j]$ , o valor  $C[A[j]]$  será a posição final correta

de  $A[j]$  no arranjo de saída, pois existem  $C[A[j]]$  elementos menores que ou iguais a  $A[j]$ . Como os elementos podem não ser distintos, decrementamos  $C[A[j]]$  toda vez que inserimos um valor  $A[j]$  no arranjo  $B$ ; isso faz com que o próximo elemento de entrada com um valor igual a  $A[j]$ , se existir algum, vá para a posição imediatamente anterior a  $A[j]$  no arranjo de saída.

Quanto tempo a ordenação por contagem exige? O loop `for` das linhas 1 e 2 demora o tempo  $\Theta(k)$ , o loop `for` das linhas 3 e 4 demora o tempo  $\Theta(n)$ , o loop `for` das linhas 6 e 7 demora o tempo  $\Theta(k)$  e o loop `for` das linhas 9 a 11 demora o tempo  $\Theta(n)$ . Portanto, o tempo total é  $\Theta(k + n)$ . Na prática, normalmente usamos a ordenação por contagem quando temos  $k = O(n)$ , em cujo caso o tempo de execução é  $\Theta(n)$ .

A ordenação por contagem supera o limite inferior de  $\Omega(n \lg n)$  demonstrado na Seção 8.1, porque não é uma ordenação por comparação. De fato, nenhuma comparação entre elementos de entrada ocorre em qualquer lugar no código. Em vez disso, a ordenação por contagem utiliza os valores reais dos elementos para efetuar a indexação em um arranjo. O limite inferior  $\Omega(n \lg n)$  para ordenação não se aplica quando nos afastamos do modelo de ordenação por comparação.

Uma propriedade importante da ordenação por contagem é o fato de ela ser *estável*: números com o mesmo valor aparecem no arranjo de saída na mesma ordem em que se encontram no arranjo de entrada. Ou seja, os vínculos entre dois números são rompidos pela regra de que qualquer número que aparecer primeiro no arranjo de entrada aparecerá primeiro no arranjo de saída. Normalmente, a propriedade de estabilidade só é importante quando dados satélite são transportados juntamente com o elemento que está sendo ordenado. A estabilidade da ordenação por contagem é importante por outra razão: a ordenação por contagem é usada frequentemente como uma sub-rotina em radix sort. Como veremos na próxima seção, a estabilidade da ordenação por contagem é crucial para a correção da radix sort.

## Exercícios

### 8.2-1

Usando a Figura 8.2 como modelo, ilustre a operação de COUNTING-SORT sobre o arranjo  $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$ .

### 8.2-2

Prove que COUNTING-SORT é estável.

### 8.2-3

Suponha que o cabeçalho do loop `for` na linha 9 do procedimento COUNTING-SORT seja reescrito:

9 `for`  $j \leftarrow 1$  to `comprimento[A]`

Mostre que o algoritmo ainda funciona corretamente. O algoritmo modificado é estável?

### 8.2-4

Descreva um algoritmo que, dados  $n$  inteiros no intervalo de 0 a  $k$ , realiza o pré-processamento de sua entrada e depois responde a qualquer consulta sobre quantos dos  $n$  inteiros recaem em um intervalo  $[a .. b]$  no tempo  $O(1)$ . Seu algoritmo deve utilizar o tempo de pré-processamento  $\Theta(n + k)$ .

## 8.3 Radix sort

A *radix sort* (ou *ordenação da raiz*) é o algoritmo usado pelas máquinas de ordenação de cartões que agora são encontradas apenas nos museus de informática. Os cartões estão organizados em 80 colunas, e em cada coluna pode ser feita uma perfuração em uma de 12 posições. O

ordenador pode ser “programado” mecanicamente para examinar uma determinada coluna de cada cartão em uma pilha e distribuir o cartão em uma de 12 caixas, dependendo de qual foi o local perfurado. Um operador pode então juntar os cartões caixa por caixa, de modo que os cartões com a primeira posição perfurada fiquem sobre os cartões com a segunda posição perfurada e assim por diante.

329	720	720	329
457	355	329	355
657	436	436	436
839	.....»	457	.....»
436	657	355	657
720	329	457	720
355	839	657	839

FIGURA 8.3 A operação de radix sort sobre uma lista de sete números de 3 dígitos. A primeira coluna é a entrada. As colunas restantes mostram a lista após ordenações sucessivas sobre posições de dígitos cada vez mais significativas. As setas verticais indicam a posição do dígito sobre o qual é feita a ordenação para produzir cada lista a partir da anterior

No caso de dígitos decimais, apenas 10 posições são utilizadas em cada coluna. (As outras duas posições são usadas para codificação de caracteres não numéricos.) Assim, um número de  $d$  dígitos ocuparia um campo de  $d$  colunas. Tendo em vista que o ordenador de cartões pode examinar apenas uma coluna de cada vez, o problema de ordenar  $n$  cartões em um número de  $d$  dígitos requer um algoritmo de ordenação.

Intuitivamente, poderíamos ordenar números sobre seu dígito *mais significativo*, ordenar cada uma das caixas resultantes recursivamente, e então combinar as pilhas em ordem. Infelizmente, como os cartões em 9 das 10 caixas devem ser postos de lado para se ordenar cada uma das caixas, esse procedimento gera muitas pilhas intermediárias de cartões que devem ser controladas. (Ver Exercício 8.3-5.)

A radix sort resolve o problema da ordenação de cartões de modo contra-intuitivo ordenando primeiro sobre o dígito *menos significativo*. Os cartões são então combinados em uma única pilha, com os cartões na caixa 0 precedendo os cartões na caixa 1, que precedem os cartões na caixa 2 e assim por diante. Então, a pilha inteira é ordenada novamente sobre o segundo dígito menos significativo e recombinação de maneira semelhante. O processo continua até os cartões terem sido ordenados sobre todos os  $d$  dígitos. É interessante observar que, nesse ponto, os cartões estão completamente ordenados sobre o número de  $d$  dígitos. Desse modo, apenas  $d$  passagens pela pilha são necessárias para se fazer a ordenação. A Figura 8.3 mostra como a radix sort opera sobre uma “pilha” (ou um “deck”) de sete números de 3 dígitos.

É essencial que as ordenações de dígitos nesse algoritmo sejam estáveis. A ordenação executada por um ordenador de cartões é estável, mas o operador tem de ser cauteloso para não alterar a ordem dos cartões à medida que eles são retirados de uma caixa, ainda que todos os cartões em uma caixa tenham o mesmo dígito na coluna escolhida.

Em um computador típico, que é uma máquina seqüencial de acesso aleatório, a radix sort é usada às vezes para ordenar registros de informações chaveados por vários campos. Por exemplo, talvez fosse desejável ordenar datas por três chaves: ano, mês e dia. Poderíamos executar um algoritmo de ordenação com uma função de comparação que, dadas duas datas, comparasse anos e, se houvesse uma ligação, comparasse meses e, se ocorresse outra ligação, comparasse dias. Como outra alternativa, poderíamos ordenar as informações três vezes com uma ordenação estável: primeiro sobre o dia, em seguida sobre o mês, e finalmente sobre o ano.

O código para radix sort é direto. O procedimento a seguir supõe que cada elemento no arranjo de  $n$  elementos  $A$  tem  $d$  dígitos, onde o dígito 1 é o dígito de mais baixa ordem e o dígito  $d$  é o dígito de mais alta ordem.

```

RADIX-SORT( $A, d$ )
1 for  $i \leftarrow 1$  to  $d$ 
2 do usar uma ordenação estável para ordenar o arranjo  $A$  sobre o dígito  $i$ 

```

### Lema 8.3

Dados  $n$  números de  $d$  dígitos em que cada dígito pode assumir até  $k$  valores possíveis, RADIX-SORT ordena corretamente esses números no tempo  $\Theta(d(n + k))$ .

**Prova** A correção de radix sort se segue por indução sobre a coluna que está sendo ordenada (ver Exercício 8.3-3). A análise do tempo de execução depende da ordenação estável usada como algoritmo de ordenação intermediária. Quando cada dígito está no intervalo de 0 a  $k - 1$  (de modo que possa assumir até  $k$  valores possíveis) e  $k$  não é muito grande, a ordenação por contagem é a escolha óbvia. Cada passagem sobre  $n$  números de  $d$  dígitos leva então o tempo  $\Theta(n + k)$ . Há  $d$  passagens; assim, o tempo total para radix sort é  $\Theta(d(n + k))$ . ■

Quando  $d$  é constante e  $k = O(n)$ , radix sort é executada em tempo linear. Mais geralmente, temos alguma flexibilidade em como quebrar cada chave em dígitos.

### Lema 8.4

Dados  $n$  números de  $b$  bits e qualquer inteiro positivo  $r \leq b$ , RADIX-SORT ordena corretamente esses números no tempo  $\Theta((b/r)(n + 2^r))$ .

**Prova** Para um valor  $r \leq b$ , visualizamos cada chave como tendo  $d = \lceil b/r \rceil$  dígitos de  $r$  bits cada. Cada dígito é um inteiro no intervalo 0 a  $2^r - 1$ , de forma que podemos usar a ordenação por contagem com  $k = 2^r - 1$ . (Por exemplo, podemos visualizar uma palavra de 32 bits como tendo 4 dígitos de 8 bits, de forma que  $b = 32$ ,  $r = 8$ ,  $k = 2^r - 1 = 255$  e  $d = b/r = 4$ .) Cada passagem da ordenação por contagem leva o tempo  $\Theta(n + k) = \Theta(n + 2^r)$  e há  $d$  passagens, dando um tempo de execução total igual a  $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$ . ■

Para valores de  $n$  e  $b$  dados, desejamos escolher o valor de  $r$ , com  $r \leq b$ , que minimiza a expressão  $(b/r)(n + 2^r)$ . Se  $b < \lfloor \lg n \rfloor$ , para qualquer valor de  $r \leq b$ , temos  $(n + 2^r) = \Theta(n)$ . Desse modo, a escolha de  $r = b$  produz um tempo de execução  $(b/b)(n + 2^b) = \Theta(n)$ , que é assintoticamente ótimo. Se  $b \geq \lfloor \lg n \rfloor$ , então a escolha de  $r = \lfloor \lg n \rfloor$  fornece o melhor tempo dentro de um fator constante, que podemos ver como a seguir. A escolha de  $r = \lfloor \lg n \rfloor$  produz um tempo de execução  $\Theta(bn/\lg n)$ . À medida que aumentamos  $r$  acima de  $\lfloor \lg n \rfloor$ , o termo  $2^r$  no numerador aumenta mais rápido que o termo  $r$  no denominador, e assim o aumento de  $r$  acima de  $\lfloor \lg n \rfloor$  resulta em um tempo de execução  $\Omega(bn/\lg n)$ . Se, em vez disso, diminuirmos  $r$  abaixo de  $\lfloor \lg n \rfloor$ , então o termo  $b/r$  aumentará, e o termo  $n + 2^r$  permanecerá em  $\Theta(n)$ .

É preferível radix sort a um algoritmo de ordenação baseado em comparação, como quicksort? Se  $b = O(\lg n)$ , como é freqüentemente o caso, e escolhemos  $r \approx \lg n$ , então o tempo de execução de radix sort é  $\Theta(n)$ , que parece ser melhor que o tempo do caso médio de quicksort,  $\Theta(n \lg n)$ . Porém, os fatores constantes ocultos na notação  $\Theta$  são diferentes. Embora radix sort possa fazer menos passagens que quicksort sobre as  $n$  chaves, cada passagem de radix sort pode tomar um tempo significativamente maior. Determinar o algoritmo de ordenação preferível depende das características das implementações, da máquina subjacente (por exemplo, quicksort utiliza com freqüência caches de hardware de modo mais eficaz que radix sort) e dos dados de entrada. Além disso, a versão de radix sort que utiliza a ordenação por contagem como ordenação estável intermediária não efetua a ordenação local, o que é feito por muitas das ordenações por comparação no tempo  $\Theta(n \lg n)$ . Desse modo, quando o espaço de armazenamento da memória primária é importante, um algoritmo local como quicksort pode ser preferível.

## Exercícios

### 8.3-1

Usando a Figura 8.3 como modelo, ilustre a operação de RADIX-SORT sobre a seguinte lista de palavras em inglês: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

### 8.3-2

Quais dos seguintes algoritmos de ordenação são estáveis: ordenação por inserção, ordenação por intercalação, heapsort e quicksort? Forneça um esquema simples que torne estável qualquer algoritmo de ordenação. Quanto tempo e espaço adicional seu esquema requer?

### 8.3-3

Use a indução para provar que radix sort funciona. Onde sua prova necessita da hipótese de que a ordenação intermediária é estável?

### 8.3-4

Mostre como ordenar  $n$  inteiros no intervalo de 0 a  $n^2 - 1$  no tempo  $O(n)$ .

### 8.3-5 \*

No primeiro algoritmo de ordenação de cartões desta seção, exatamente quantas passagens de ordenação são necessárias para ordenar números decimais de  $d$  dígitos no pior caso? Quantas pilhas de cartões um operador precisaria controlar no pior caso?

## 8.4 Bucket sort

A *bucket sort* (ou *ordenação por balde*) funciona em tempo linear quando a entrada é gerada a partir de uma distribuição uniforme. Como a ordenação por contagem, a bucket sort é rápida porque pressupõe algo sobre a entrada. Enquanto a ordenação por contagem presume que a entrada consiste em inteiros em um intervalo pequeno, bucket sort presume que a entrada é gerada por um processo aleatório que distribui elementos uniformemente sobre o intervalo  $[0, 1]$ . (Consulte a Seção C.2 para ver uma definição de distribuição uniforme.)

A idéia de bucket sort é dividir o intervalo  $[0, 1]$  em  $n$  subintervalos de igual tamanho, ou *baldes*, e depois distribuir os  $n$  números de entrada entre os baldes. Tendo em vista que as entradas são uniformemente distribuídas sobre  $[0, 1]$ , não esperamos que muitos números caiam em cada balde. Para produzir a saída, simplesmente ordenamos os números em cada balde, e depois percorremos os baldes em ordem, listando os elementos contidos em cada um.

Nosso código para bucket sort pressupõe que a entrada é um arranjo de  $n$  elementos  $A$ , e que cada elemento  $A[i]$  no arranjo satisfaz a  $0 \leq A[i] < 1$ . O código exige um arranjo auxiliar  $B[0 \dots n - 1]$  de listas ligadas (baldes) e pressupõe que existe um mecanismo para manter tais listas. (A Seção 10.2 descreve como implementar operações básicas sobre listas ligadas.)

### BUCKET-SORT( $A$ )

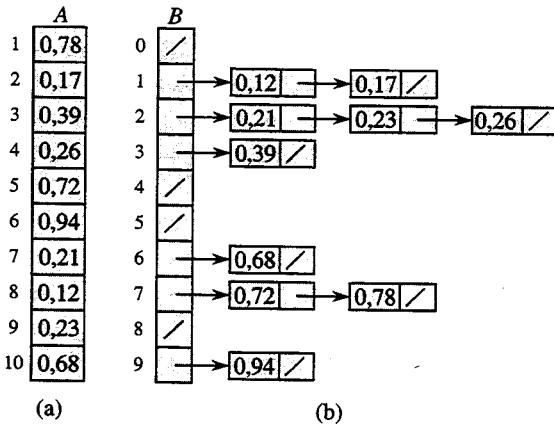
```
1  $n \leftarrow \text{comprimento}[A]$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   do inserir  $A[i]$  na lista  $B[\lfloor nA[i] \rfloor]$ 
4 for  $i \leftarrow 0$  to  $n - 1$ 
5   do ordenar lista  $B[i]$  com ordenação por inserção
6 concatenar as listas  $B[0], B[1], \dots, B[n - 1]$  juntas em ordem
```

A Figura 8.4 mostra a operação de bucket sort sobre um arranjo de entrada de 10 números.

Para ver que esse algoritmo funciona, considere dois elementos  $A[i]$  e  $A[j]$ . Suponha sem perda de generalidade que  $A[i] \leq A[j]$ . Tendo em vista que  $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$ , o elemento  $A[i]$  é inserido no mesmo balde que  $A[j]$  ou em um balde com índice mais baixo. Se  $A[i]$  e  $A[j]$  são inseri-

dos no mesmo balde, então o loop **for** das linhas 4 e 5 os coloca na ordem adequada. Se  $A[i]$  e  $A[j]$  são inseridos em baldes diferentes, a linha 6 os coloca na ordem adequada. Portanto, a bucket sort funciona corretamente.

Para analisar o tempo de execução, observe que todas as linhas exceto a linha 5 demoram o tempo  $O(n)$  no pior caso. Resta equilibrar o tempo total ocupado pelas  $n$  chamadas à ordenação por inserção na linha 5.



**FIGURA 8.4** A operação de BUCKET-SORT. (a) O arranjo de entrada  $A[1 \dots 10]$ . (b) O arranjo  $B[0 \dots 9]$  de listas ordenadas (baldes) depois da linha 5 do algoritmo. O balde  $i$  contém valores no intervalo  $[i/10, (i+1)/10)$ . A saída ordenada consiste em uma concatenação em ordem das listas  $B[0], B[1], \dots, B[9]$

Para analisar o custo das chamadas para ordenação por inserção, seja  $n_i$  a variável aleatória que denota o número de elementos inseridos no balde  $B[i]$ . Tendo em vista que a ordenação por inserção funciona em tempo quadrático (consulte a Seção 2.2), o tempo de execução de bucket sort é

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2).$$

Tomando as expectativas de ambos os lados e usando a linearidade de expectativa, temos

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \quad (\text{por linearidade de expectativa}) \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad (\text{pela equação (C.21)}). \end{aligned} \tag{8.1}$$

Afirmamos que

$$E[n_i^2] = 2 - 1/n \tag{8.2}$$

para  $i = 0, 1, \dots, n-1$ . Não surpreende que cada balde  $i$  tenha o mesmo valor de  $E[n_i^2]$ , pois cada valor no arranjo de entrada  $A$  tem igual probabilidade de cair em qualquer balde. Para provar a equação (8.2), definimos variáveis indicadoras aleatórias

$$X_{ij} = I\{A[j] \text{ recai no balde } i\}$$

para  $i = 0, 1, \dots, n - 1$  e  $j = 1, 2, \dots, n$ . Desse modo,

$$n_i = \sum_{j=1}^n X_{ij} .$$

Para calcular  $E[n_i^2]$ , expandimos o quadrado e reagrupamos os termos:

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] \\ &= E\left[\sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik}\right] \\ &= E\left[\sum_{j=1}^n X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} X_{ij} X_{ik}\right] \\ &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij} X_{ik}], \end{aligned} \tag{8.3}$$

onde a última linha se segue por linearidade de expectativa. Avaliamos os dois somatórios separadamente. A variável indicadora aleatória  $X_{ij}$  é 1 com probabilidade  $1/n$  e 0 em caso contrário e, portanto,

$$\begin{aligned} E[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{n} . \end{aligned}$$

Quando  $k \neq j$ , as variáveis  $X_{ij}$  e  $X_{ik}$  são independentes e, por conseguinte,

$$\begin{aligned} E[X_{ij} X_{ik}] &= E[X_{ij}] E[X_{ik}] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n^2} . \end{aligned}$$

Substituindo esses dois valores esperados na equação (8.3), obtemos

$$\begin{aligned} E[X_{ij}^2] &= \sum_{j=1}^n \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} \frac{1}{n^2} \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \end{aligned}$$

$$\begin{aligned}
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n},
\end{aligned}$$

o que prova a equação (8.2).

Usando esse valor esperado na equação (8.1), concluímos que o tempo esperado para bucket sort é  $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$ . Desse modo, o algoritmo de bucket sort inteiro funciona no tempo esperado linear.

Mesmo que a entrada não seja obtida a partir de uma distribuição uniforme, bucket sort ainda pode ser executada em tempo linear. Como a entrada tem a propriedade de que a soma dos quadrados dos tamanhos de baldes é linear no número total de elementos, a equação (8.1) nos diz que bucket sort funcionará em tempo linear.

## Exercícios

### 8.4-1

Usando a Figura 8.4 como modelo, ilustre a operação de BUCKET-SORT no arranjo  $A = \langle 0,79, 0,13, 0,16, 0,64, 0,39, 0,20, 0,89, 0,53, 0,71, 0,42 \rangle$ .

### 8.4-2

Qual é o tempo de execução do pior caso para o algoritmo de bucket sort? Que alteração simples no algoritmo preserva seu tempo de execução esperado linear e torna seu tempo de execução no pior caso igual a  $O(n \lg n)$ ?

### 8.4-3

Seja  $X$  uma variável aleatória que é igual ao número de caras em dois lançamentos de uma moeda comum. Qual é  $E[X^2]$ ? Qual é  $E^2[X]$ ?

### 8.4-4 \*

Temos  $n$  pontos no círculo unitário,  $p_i = (x_i, y_i)$ , tais que  $0 < x_i^2 + y_i^2 \leq 1$  para  $i = 1, 2, \dots, n$ . Suponha que os pontos estejam distribuídos uniformemente; ou seja, a probabilidade de se encontrar um ponto em qualquer região do círculo é proporcional à área dessa região. Projete um algoritmo de tempo esperado  $\Theta(n)$  para ordenar os  $n$  pontos por suas distâncias  $d_i = \sqrt{x_i^2 + y_i^2}$  a partir da origem. (Sugestão: Projete os tamanhos de baldes em BUCKET-SORT para refletir a distribuição uniforme dos pontos no círculo unitário.)

### 8.4-5 \*

Uma função de distribuição de probabilidades  $P(x)$  para uma variável aleatória  $X$  é definida por  $P(x) = \Pr\{X \leq x\}$ . Suponha que uma lista de  $n$  variáveis aleatórias  $X_1, X_2, \dots, X_n$  seja obtida a partir de uma função de distribuição de probabilidades contínuas  $P$  que possa ser calculada no tempo  $O(1)$ . Mostre como ordenar esses números em tempo esperado linear.

## Problemas

### 8-1 Limites inferiores do caso médio na ordenação por comparação

Neste problema, provamos um limite inferior  $\Omega(n \lg n)$  sobre o tempo de execução esperado de qualquer ordenação por comparação determinística ou aleatória sobre  $n$  elementos de entrada distintos. Começamos examinando uma ordenação por comparação determinística  $A$  com árvore de decisão  $T_A$ . Supomos que toda permutação de entradas de  $A$  é igualmente provável.

- Suponha que cada folha de  $T_A$  seja identificada com a probabilidade de ser alcançada dada uma entrada aleatória. Prove que exatamente  $n!$  folhas são identificadas com  $1/n!$  e que as restantes são identificadas com 0.

- b. Seja  $D(T)$  um valor que denota o comprimento do caminho externo de uma árvore de decisão  $T$ ; isto é,  $D(T)$  é a soma das profundidades de todas as folhas de  $T$ . Seja  $T$  uma árvore de decisão com  $k > 1$  folhas, e sejam  $RT$  e  $LT$  as subárvore direita e esquerda de  $T$ . Mostre que  $D(T) = D(LT) + D(RT) + k$ .
- c. Seja  $d(k)$  o valor mínimo de  $D(T)$  sobre todas as árvores de decisão  $T$  com  $k > 1$  folhas. Mostre que  $d(k) = \min_{1 \leq i \leq k-1} \{d(i) + d(k-i) + k\}$ . (Sugestão: Considere uma árvore de decisão  $T$  com  $k$  folhas que alcance o mínimo. Seja  $i_0$  o número de folhas em  $LT$  e  $k - i_0$  o número de folhas em  $RT$ .)
- d. Prove que, para um dado valor de  $k > 1$  e  $i$  no intervalo  $1 \leq i \leq k-1$ , a função  $i \lg i + (k-i) \lg (k-i)$  é minimizada em  $i = k/2$ . Conclua que  $d(k) = \Omega(k \lg k)$ .
- e. Prove que  $D(T_A) = \Omega(n! \lg(n!))$  e conclua que o tempo esperado para ordenar  $n$  elementos é  $\Omega(n \lg n)$ .

Agora, considere uma ordenação por comparação *aleatória B*. Podemos estender o modelo de árvore de decisão para tratar a aleatoriedade, incorporando dois tipos de nós: os nós de comparação comum e os nós “aleatórios”. Um nó aleatório modela uma opção aleatória da forma  $RANDOM(1, r)$  feita pelo algoritmo  $B$ ; o nó tem  $r$  filhos, cada um dos quais tem igual probabilidade de ser escolhido durante uma execução do algoritmo.

- f. Mostre que, para qualquer ordenação por comparação aleatória  $B$ , existe uma ordenação por comparação determinística  $A$  que não faz mais comparações sobre a média que  $B$ .

### 8-2 Ordenação local em tempo linear

Vamos supor que temos um arranjo de  $n$  registros de dados para ordenar e que a chave de cada registro tem o valor 0 ou 1. Um algoritmo para ordenar tal conjunto de registros poderia ter algum subconjunto das três características desejáveis a seguir:

1. O algoritmo é executado no tempo  $O(n)$ .
  2. O algoritmo é estável.
  3. O algoritmo ordena localmente, sem utilizar mais que uma quantidade constante de espaço de armazenamento além do arranjo original.
- a. Dê um algoritmo que satisfaça aos critérios 1 e 2 anteriores.
  - b. Dê um algoritmo que satisfaça aos critérios 1 e 3 anteriores.
  - c. Dê um algoritmo que satisfaça aos critérios 2 e 3 anteriores.
  - d. Algum dos seus algoritmos de ordenação das partes (a)-(c) pode ser usado para ordenar  $n$  registros com chaves de  $b$  bits usando radix sort no tempo  $O(bn)$ ? Explique como ou por que não.
  - e. Suponha que os  $n$  registros tenham chaves no intervalo de 1 a  $k$ . Mostre como modificar a ordenação por contagem de tal forma que os registros possam ser ordenados localmente no tempo  $O(n + k)$ . Você pode usar o espaço de armazenamento  $O(k)$  fora do arranjo de entrada. Seu algoritmo é estável? (Sugestão: Como você faria isso para  $k = 3$ ?)

### 8-3 Ordenação de itens de comprimento variável

- a. Você tem um arranjo de inteiros, no qual diferentes inteiros podem ter números de dígitos distintos, mas o número total de dígitos sobre *todos* os inteiros no arranjo é  $n$ . Mostre como ordenar o arranjo no tempo  $O(n)$ .
- b. Você tem um arranjo de cadeias, no qual diferentes cadeias podem ter números de caracteres distintos, mas o número total de caracteres em todas as cadeias é  $n$ . Mostre como ordenar as cadeias no tempo  $O(n)$ .

(Observe que a ordem desejada aqui é a ordem alfabética padrão; por exemplo,  $a < ab < b$ .)

### 8-4 Jarros de água

Vamos supor que você tem  $n$  jarros de água vermelhos e  $n$  jarros azuis, todos de diferentes formas e tamanhos. Todos os jarros vermelhos contêm quantidades diferentes de água, como também os jarros azuis. Além disso, para todo jarro vermelho, existe um jarro azul que contém a mesma quantidade de água e vice-versa.

Sua tarefa é encontrar um agrupamento dos jarros em pares de jarros vermelhos e azuis que contêm a mesma quantidade de água. Para isso, você pode executar a seguinte operação: escolher um par de jarros no qual um é vermelho e um é azul, encher o jarro vermelho com água, e depois despejar a água no jarro azul. Essa operação lhe informará se o jarro vermelho ou o jarro azul pode conter mais água, ou se eles têm o mesmo volume. Suponha que tal comparação demore uma unidade de tempo. Seu objetivo é encontrar um algoritmo que faça um número mínimo de comparações para determinar o agrupamento. Lembre-se de que você não pode comparar diretamente dois jarros vermelhos ou dois jarros azuis.

- a. Descreva um algoritmo determinístico que use  $\Theta(n^2)$  comparações para agrupar os jarros em pares.
- b. Prove um limite inferior  $\Omega(n \lg n)$  para o número de comparações que um algoritmo que resolve esse problema deve efetuar.
- c. Dê um algoritmo aleatório cujo número esperado de comparações seja  $O(n \lg n)$  e prove que esse limite é correto. Qual é o número de comparações no pior caso do seu algoritmo?

### 8-5 Ordenação por média

Suponha que, em vez de ordenar um arranjo, simplesmente exigimos que os elementos aumentem na média. De modo mais preciso, chamamos um arranjo de  $n$  elementos  $A$  de  **$k$ -ordenado** se, para todo  $i = 1, 2, \dots, n - k$ , é válida a desigualdade a seguir:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}.$$

- a. O que significa um arranjo ser 1-ordenado?
- b. Forneça uma permutação dos números 1, 2, ..., 10 que seja 2-ordenada, mas não ordenada.
- c. Prove que um arranjo de  $n$  elementos é  $k$ -ordenado se e somente se  $A[i] \leq A[i + k]$  para todo  $i = 1, 2, \dots, n - k$ .
- d. Forneça um algoritmo que faça a  $k$ -ordenação de um arranjo de  $n$  elementos no tempo  $O(n \lg(n/k))$ .

Também podemos mostrar um limite inferior sobre o tempo para produzir um arranjo  $k$ -ordenado, quando  $k$  é uma constante.

- e. Mostre que um arranjo  $k$ -ordenado de comprimento  $n$  pode ser ordenado no tempo  $O(n \lg k)$ . (Sugestão: Use a solução do Exercício 6.5-8.)
- f. Mostre que, quando  $k$  é uma constante, é necessário o tempo  $\Omega(n \lg n)$  para fazer a  $k$ -ordenação de um arranjo de  $n$  elementos. (Sugestão: Use a solução para a parte anterior, juntamente com o limite inferior sobre ordenações por comparação.)

### 8-6 Limite inferior sobre a intercalação de listas ordenadas

O problema de intercalar duas listas ordenadas surge com freqüência. Ele é usado como uma sub-rotina de MERGE-SORT, e o procedimento para intercalar duas listas ordenadas é dado como MERGE na Seção 2.3.1. Neste problema, mostraremos que existe um limite inferior  $2n - 1$  sobre o número de comparações no pior caso exigidas para intercalar duas listas ordenadas, cada uma contendo  $n$  itens.

Primeiro, mostraremos um limite inferior de  $2n - o(n)$  comparações, usando uma árvore de decisão.

- a. Mostre que, dados  $2n$  números, existem  $\binom{2n}{n}$  maneiras possíveis de dividi-los em duas listas ordenadas, cada uma com  $n$  números.
  - b. Usando uma árvore de decisão, mostre que qualquer algoritmo que intercala corretamente duas listas ordenadas utiliza pelo menos  $2n - o(n)$  comparações.
- Agora, mostraremos um limite  $2n - 1$ , ligeiramente mais restrito.
- c. Mostre que, se dois elementos são consecutivos na seqüência ordenada e vêm de listas opositas, então eles devem ser comparados.
  - d. Use sua resposta à parte anterior para mostrar um limite inferior de  $2n - 1$  comparações para intercalar duas listas ordenadas.

## Notas do capítulo

O modelo de árvore de decisão para o estudo de ordenações por comparação foi introduzido por Ford e Johnson [94]. O tratamento abrangente de Knuth sobre a ordenação [185] cobre muitas variações sobre o problema da ordenação, inclusive o limite inferior teórico de informações sobre a complexidade da ordenação fornecida aqui. Os limites inferiores para ordenação com o uso de generalizações do modelo de árvore de decisão foram estudados amplamente por Ben-Or [36].

Knuth credita a H. H. Seward a criação da ordenação por contagem em 1954, e também a idéia de combinar a ordenação por contagem com a radix sort. A radix sort começando pelo dígito menos significativo parece ser um algoritmo popular amplamente utilizado por operadores de máquinas mecânicas de ordenação de cartões. De acordo com Knuth, a primeira referência ao método publicada é um documento de 1929 escrito por L. J. Comrie que descreve o equipamento de perfuração de cartões. A bucket sort esteve em uso desde 1956, quando a idéia básica foi proposta por E. J. Isaac e R. C. Singleton.

Munro e Raman [229] fornecem um algoritmo de ordenação estável que executa  $O(n^{1+\varepsilon})$  comparações no pior caso, onde  $0 < \varepsilon \leq 1$  é qualquer constante fixa. Embora qualquer dos algoritmos de tempo  $O(n \lg n)$  efetue um número menor de comparações, o algoritmo de Munro e Raman move os dados apenas  $O(n)$  vezes e opera localmente.

O caso de ordenar  $n$  inteiros de  $b$  bits no tempo  $o(n \lg n)$  foi considerado por muitos pesquisadores. Vários resultados positivos foram obtidos, cada um sob hipóteses um pouco diferentes a respeito do modelo de computação e das restrições impostas sobre o algoritmo. Todos os resultados pressupõem que a memória do computador está dividida em palavras endereçáveis de  $b$  bits. Fredman e Willard [99] introduziram a estrutura de dados de árvores de fusão e a empregaram para ordenar  $n$  inteiros no tempo  $O(n \lg n / \lg \lg n)$ . Esse limite foi aperfeiçoado mais tarde para o tempo  $O(n \sqrt{\lg n})$  por Andersson [16]. Esses algoritmos exigem o uso de multiplicação e de várias constantes pré-calculadas. Andersson, Hagerup, Nilsson e Raman [17] mostraram como ordenar  $n$  inteiros no tempo  $O(n \lg \lg n)$  sem usar multiplicação, mas seu método exige espaço de armazenamento que pode ser ilimitado em termos de  $n$ . Usando-se o hash multiplicativo, é possível reduzir o espaço de armazenamento necessário para  $O(n)$ , mas o limite  $O(n \lg \lg n)$  do pior caso sobre o tempo de execução se torna um limite de tempo esperado. Generalizando as árvores de pesquisa exponencial de Andersson [16], Thorup [297] forneceu um algoritmo de ordenação de tempo  $O(n(\lg \lg n)^2)$  que não usa multiplicação ou aleatoriedade, e que utiliza espaço linear. Combinando essas técnicas com algumas idéias novas, Han [137] melhorou o limite para ordenação até o tempo  $O(n \lg \lg n \lg \lg \lg n)$ . Embora esses algoritmos sejam inovações teóricas importantes, todos eles são bastante complicados e neste momento parece improvável que venham a competir na prática com algoritmos de ordenação existentes.

---

## Capítulo 9

# Medianas e estatísticas de ordem

A  $i$ -ésima **estatística de ordem** de um conjunto de  $n$  elementos é o  $i$ -ésimo menor elemento. Por exemplo, o **mínimo** de um conjunto de elementos é a primeira estatística de ordem ( $i = 1$ ), e o **máximo** é a  $n$ -ésima estatística de ordem ( $i = n$ ). Informalmente, uma **mediana** é o “ponto médio” do conjunto. Quando  $n$  é ímpar, a mediana é única, ocorrendo em  $i = (n + 1)/2$ . Quando  $n$  é par, existem duas medianas, ocorrendo em  $i = n/2$  e  $i = n/2 + 1$ . Desse modo, independentemente da paridade de  $n$ , as medianas ocorrem em  $i = \lfloor (n + 1)/2 \rfloor$  (a **mediana inferior**) e  $i = \lceil (n + 1)/2 \rceil$  (a **mediana superior**). Porém, por simplicidade neste texto, usaremos de forma coerente a expressão “a mediana” para nos referirmos à mediana inferior.

Este capítulo focaliza o problema de selecionar a  $i$ -ésima estatística de ordem de um conjunto de  $n$  números distintos. Supomos por conveniência que o conjunto contém números distintos, embora virtualmente tudo que fizermos se estenda à situação na qual um conjunto contém valores repetidos. O **problema de seleção** pode ser especificado formalmente do seguinte modo:

**Entrada:** Um conjunto  $A$  de  $n$  números (distintos) e um número  $i$ , com  $1 \leq i \leq n$ .

**Saída:** O elemento  $x \in A$  que é maior que exatamente  $i - 1$  outros elementos de  $A$ .

O problema de seleção pode ser resolvido no tempo  $O(n \lg n)$ , pois podemos ordenar os números usando heapsort ou ordenação por intercalação, e depois simplesmente indexar o  $i$ -ésimo elemento no arranjo de saída. Contudo, existem algoritmos mais rápidos.

Na Seção 9.1, examinamos o problema de selecionar o mínimo e o máximo de um conjunto de elementos. Mais interessante é o problema de seleção geral, que é investigado nas duas seções subsequentes. A Seção 9.2 analisa um algoritmo prático que alcança um limite  $O(n)$  sobre o tempo de execução no caso médio. A Seção 9.3 contém um algoritmo de maior interesse teórico, que alcança o tempo de execução  $O(n)$  no pior caso.

### 9.1 Mínimo e máximo

Quantas comparações são necessárias para determinar o mínimo de um conjunto de  $n$  elementos? Podemos obter facilmente um limite superior de  $n - 1$  comparações: examine cada elemento do conjunto isoladamente e mantenha o controle do menor elemento visto até então. No procedimento a seguir, vamos supor que o conjunto reside no arranjo  $A$ , onde  $\text{comprimento}[A] = n$ .

```

MINIMUM( $A$ )
1  $min \leftarrow A[1]$ 
2 for  $i \leftarrow 2$  to  $comprimento[A]$ 
3   do if  $min > A[i]$ 
4     then  $min \leftarrow A[i]$ 
5 return  $min$ 

```

A localização do máximo também pode, é claro, ser realizada com  $n - 1$  comparações.

Isso é o melhor que podemos fazer? Sim, desde que possamos obter um limite inferior de  $n - 1$  comparações para o problema de determinar o mínimo. Imagine qualquer algoritmo que determina o mínimo como um torneio entre os elementos. Cada comparação é uma partida no torneio, na qual o menor dos dois elementos vence. A observação chave é que todo elemento, exceto o vencedor, deve perder pelo menos uma partida. Conseqüentemente,  $n - 1$  comparações são necessárias para determinar o mínimo, e o algoritmo MINIMUM é ótimo com relação ao número de comparações executadas.

## Mínimo e máximo simultâneos

Em algumas aplicações, devemos localizar tanto o mínimo quanto o máximo de um conjunto de  $n$  elementos. Por exemplo, um programa gráfico talvez precise ajustar a escala de um conjunto de  $(x, y)$  dados para se encaixar em uma tela de exibição retangular ou em outro dispositivo de saída gráfica. Para fazer isso, o programa deve primeiro determinar o mínimo e o máximo de cada coordenada.

Não é difícil criar um algoritmo que possa encontrar tanto o mínimo quanto o máximo de  $n$  elementos usando  $\Theta(n)$  comparações, que é o número assintoticamente ótimo de comparações. Simplesmente localize o mínimo e o máximo independentemente, usando  $n - 1$  comparações para cada um deles, o que fornece um total de  $2n - 2$  comparações.

De fato, no máximo  $3\lfloor n/2 \rfloor$  comparações são suficientes para se encontrar tanto o mínimo quanto o máximo. A estratégia é manter os elementos mínimo e máximo vistos até agora. Porém, em lugar de processar cada elemento da entrada comparando-o com o mínimo e o máximo atuais, a um custo de duas comparações por elemento, processamos os elementos aos pares. Primeiro, comparamos pares de elementos da entrada *uns com os outros*, depois comparamos o menor com o mínimo atual e o maior com o máximo atual, a um custo de três comparações para cada dois elementos.

A definição de valores iniciais para o mínimo e o máximo atuais depende do fato de  $n$  ser ímpar ou par. Se  $n$  é ímpar, definimos tanto o mínimo quanto o máximo com o valor do primeiro elemento e, em seguida, processamos os elementos restantes aos pares. Se  $n$  é par, executamos uma comparação sobre os dois primeiros elementos para determinar os valores iniciais do mínimo e do máximo, e depois processamos os elementos restantes aos pares, como no caso de  $n$  ímpar.

Vamos analisar o número total de comparações. Se  $n$  é ímpar, executamos  $3\lfloor n/2 \rfloor$  comparações. Se  $n$  é par, executamos uma comparação inicial seguida por  $3(n - 2)/2$  comparações, dando um total de  $3n/2 - 2$ . Desse modo, em qualquer caso, o número total de comparações é no máximo  $3\lfloor n/2 \rfloor$ .

## Exercícios

### 9.1-1

Mostre que o segundo menor entre  $n$  elementos pode ser encontrado com  $n + \lceil \lg n \rceil - 2$  comparações no pior caso. (Sugestão: Encontre também o menor elemento.)

### 9.1-2 \*

Mostre que  $\lceil 3n/2 \rceil - 2$  comparações são necessárias no pior caso para localizar tanto o máximo quanto o mínimo entre  $n$  números. (Sugestão: Considere quantos números são potencialmente o máximo ou o mínimo, e investigue como uma comparação afeta essas contagens.)

## 9.2 Seleção em tempo esperado linear

O problema de seleção geral parece mais difícil que o problema simples de se achar um mínimo, ainda que surpreendentemente o tempo de execução assintótico para ambos os problemas seja o mesmo:  $\Theta(n)$ . Nesta seção, apresentamos um algoritmo de dividir e conquistar para o problema de seleção. O algoritmo RANDOMIZED-SELECT é modelado sobre o algoritmo quicksort do Capítulo 7. Como no quicksort, a idéia é particionar o arranjo de entrada recursivamente. Porém, diferente de quicksort, que processa recursivamente ambos os lados da partição, RANDOMIZED-SELECT só funciona sobre um lado da partição. Essa diferença fica evidente na análise: enquanto quicksort tem um tempo de execução esperado  $\Theta(n \lg n)$ , o tempo esperado de RANDOMIZED-SELECT é  $\Theta(n)$ .

RANDOMIZED-SELECT utiliza o procedimento RANDOMIZED-PARTITION introduzido na Seção 7.3. Desse modo, como RANDOMIZED-QUICKSORT, ele é um algoritmo aleatório, pois seu comportamento é determinado em parte pela saída de um gerador de números aleatórios. O código a seguir para RANDOMIZED-SELECT retorna o  $i$ -ésimo menor elemento do arranjo  $A[p .. r]$ .

```
RANDOMIZED-SELECT( $A, p, r, i$ )
1 if  $p = r$ 
2   then return  $A[p]$ 
3    $q \leftarrow$  RANDOMIZED-PARTITION( $A, p, r$ )
4    $k \leftarrow q - p + 1$ 
5   if  $i = k$        $\triangleright$  O valor pivô é a resposta
6     then return  $A[q]$ 
7   elseif  $i < k$ 
8     then return RANDOMIZED-SELECT( $A, p, q - 1, i$ )
9   else return RANDOMIZED-SELECT( $A, q + 1, r, i - k$ )
```

Após RANDOMIZED-PARTITION ser executado na linha 3 do algoritmo, o arranjo  $A[p .. r]$  é particionado em dois subarranjos (possivelmente vazios)  $A[p .. q - 1]$  e  $A[q + 1 .. r]$  tais que cada elemento de  $A[p .. q - 1]$  é menor que ou igual a  $A[q]$  que, por sua vez, é menor que cada elemento de  $A[q + 1 .. r]$ . Como em quicksort, vamos nos referir a  $A[q]$  como o elemento *pivô*. A linha 4 de RANDOMIZED-SELECT calcula o número  $k$  de elementos no subarranjo  $A[p .. q]$ , ou seja, o número de elementos no lado baixo da partição, mais uma unidade para o elemento pivô. A linha 5 verifica então se  $A[q]$  é o  $i$ -ésimo menor elemento. Se for, então  $A[q]$  é retornado. Caso contrário, o algoritmo determina em qual dos dois subarranjos  $A[p .. q - 1]$  e  $A[q + 1 .. r]$  o  $i$ -ésimo menor elemento se encontra. Se  $i < k$ , então o elemento desejado está no lado baixo da partição, e é recursivamente selecionado do subarranjo na linha 8. Porém, se  $i > k$ , o elemento desejado reside no lado alto da partição. Como já conhecemos  $k$  valores que são menores que o  $i$ -ésimo menor elemento de  $A[p .. r]$  – isto é, os elementos de  $A[p .. q]$  – o elemento desejado é o  $(i - k)$ -ésimo menor elemento de  $A[q + 1 .. r]$ , encontrado recursivamente na linha 9. O código parece permitir chamadas recursivas a subarranjos com 0 elementos, mas o Exercício 9.2-1 lhe pede para mostrar que essa situação não pode acontecer.

O tempo de execução do pior caso para RANDOMIZED-SELECT é  $\Theta(n^2)$ , mesmo para se encontrar o mínimo, porque poderíamos estar extremamente sem sorte e sempre efetuar a partição em torno do maior elemento restante, e o particionamento levará o tempo  $\Theta(n)$ . Entretanto, o algoritmo funciona bem no caso médio e, porque ele é aleatório, nenhuma entrada específica surge do comportamento no pior caso.

O tempo exigido por RANDOMIZED-SELECT em um arranjo de entrada  $A[p .. r]$  de  $n$  elementos é uma variável aleatória que denotamos por  $T(n)$ , e obtemos um limite superior sobre  $E[T(n)]$  como a seguir. O procedimento RANDOMIZED-PARTITION tem igual probabilidade de retornar qualquer elemento como pivô. Assim, para cada  $k$  tal que  $1 \leq k \leq n$ , o subarranjo  $A[p .. q]$  tem  $k$  elementos (todos menores que ou iguais ao pivô) com probabilidade  $1/n$ . Para  $k = 1, 2, \dots, n$ , definimos variáveis indicadoras aleatórias  $X_k$ , nas quais

$$X_k = I\{\text{o subarranjo } A[p .. q] \text{ tem exatamente } k \text{ elementos}\},$$

e assim temos

$$E[X_k] = 1/n. \quad (9.1)$$

Quando chamamos RANDOMIZED-SELECT e escolhemos  $A[q]$  como o elemento pivô, não sabemos *a priori* se terminaremos imediatamente com a resposta correta, faremos a recursão no subarranjo  $A[p .. q-1]$  ou faremos a recursão no subarranjo  $A[q+1 .. r]$ . Essa decisão depende de onde o  $i$ -ésimo menor elemento ficará em relação a  $A[q]$ . Supondo que  $T(n)$  seja monotonicamente crescente, podemos limitar o tempo necessário para a chamada recursiva pelo tempo necessário para a chamada recursiva sobre a maior entrada possível. Em outras palavras, supomos, para obter um limite superior, que o  $i$ -ésimo elemento está sempre no lado da partição com o maior número de elementos. Para uma dada chamada de RANDOMIZED-SELECT, a variável indicadora aleatória  $X_k$  tem o valor 1 para exatamente um valor de  $k$ , e é 0 para todos os outros  $k$ . Quando  $X_k = 1$ , os dois subarranjos sobre os quais podemos fazer a recursão têm tamanhos  $k-1$  e  $n-k$ . Conseqüentemente, temos a recorrência

$$\begin{aligned} T(n) &\leq \sum_{k=1}^n X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^n (X_k \cdot T(\max(k-1, n-k)) + O(n)). \end{aligned}$$

Tomando valores esperados, temos

$$E[T(n)]$$

$$\begin{aligned} &\leq E\left[\sum_{k=1}^n X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\ &= \sum_{k=1}^n E[X_k] \cdot T(\max(k-1, n-k)) + O(n) \quad (\text{por linearidade de expectativa}) \\ &= \sum_{k=1}^n E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n) \quad (\text{pela equação (C.23)}) \\ &= \sum_{k=1}^n \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n) \quad (\text{pela equação (9.1)}). \end{aligned}$$

Para aplicar a equação (C.23), dependemos do fato de  $X_k$  e  $T(\max(k-1, n-k))$  serem variáveis aleatórias independentes. O Exercício 9.2-2 lhe pede para justificar essa afirmação.

Vamos considerar a expressão  $\max(k - 1, n - k)$ . Temos

$$\max(k - 1, n - k) = \begin{cases} k - 1 & \text{se } k > \lceil n/2 \rceil, \\ n - k & \text{se } k \leq \lceil n/2 \rceil. \end{cases}$$

Se  $n$  é par, cada termo de  $T(\lceil n/2 \rceil)$  até  $T(n - 1)$  aparece exatamente duas vezes no somatório e, se  $n$  é ímpar, todos esses termos aparecem duas vezes e o termo  $T(\lfloor n/2 \rfloor)$  aparece uma vez. Desse modo, temos

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n).$$

Resolvemos a recorrência por substituição. Suponha que  $T(n) \leq cn$  para alguma constante  $c$  que satisfaça às condições iniciais da recorrência. Supomos que  $T(n) = O(1)$  para  $n$  menor que alguma constante; escolheremos essa constante mais adiante. Também escolheremos uma constante  $a$  tal que a função descrita pelo termo  $O(n)$  anterior (que descreve o componente não recursivo do tempo de execução do algoritmo) seja limitado acima por  $an$  para todo  $n > 0$ . Usando essa hipótese induutiva, temos

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\ &= \frac{2c}{n} \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\ &= \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\ &= \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\ &= \frac{2c}{n} \left( \frac{n^2 - n}{2} - \frac{(n^2/4 - 3n/2 + 2)}{2} \right) + an \\ &= \frac{c}{n} \left( \frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\ &= c \left( \frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\ &\leq \frac{3cn}{4} + \frac{c}{2} + an. \\ &= cn \left( \frac{cn}{4} - \frac{c}{2} - an \right). \end{aligned}$$

Para completar a prova, precisamos mostrar que, para  $n$  suficientemente grande, essa última expressão é no máximo  $cn$  ou, de modo equivalente, que  $cn/4 - c/2 - an \geq 0$ . Se adicionarmos  $c/2$  a ambos os lados e fatorarmos  $n$ , obteremos  $n(c/4 - a) \geq c/2$ . Desde que a constante  $c$  seja escolhida de modo que  $c/4 - a > 0$ , isto é,  $c > 4a$ , poderemos dividir ambos os lados por  $c/4 - a$ , obtendo

$$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}.$$

Desse modo, se considerarmos que  $T(n) = O(1)$  para  $n < 2c/(c - 4\alpha)$ , teremos  $T(n) = O(n)$ . Concluímos que qualquer estatística de ordem, e em particular a mediana, pode ser determinada em média no tempo linear.

## Exercícios

### 9.2-1

Mostre que, em RANDOMIZED-SELECT, não é feita nenhuma chamada recursiva para um arranjo de comprimento 0.

### 9.2-2

Demonstre que a variável indicadora aleatória  $X_k$  e o valor  $T(\max(k-1, n-k))$  são independentes.

### 9.2-3

Escreva uma versão iterativa de RANDOMIZED-SELECT.

### 9.2-4

Suponhamos que RANDOMIZED-SELECT seja utilizado para selecionar o elemento mínimo do arranjo  $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$ . Descreva uma seqüência de partições que resulte em um desempenho do pior caso de RANDOMIZED-SELECT.

## 9.3 Seleção em tempo linear no pior caso

Agora, vamos examinar um algoritmo de seleção cujo tempo de execução é  $O(n)$  no pior caso. Como RANDOMIZED-SELECT, o algoritmo SELECT localiza o elemento desejado particionando recursivamente o arranjo de entrada. Porém, a idéia que rege o algoritmo é *garantir* uma boa divisão quando o arranjo é particionado. SELECT utiliza o algoritmo de particionamento determinístico PARTITION de quicksort (ver Seção 7.1), modificado para tomar o elemento a particionar como um parâmetro de entrada.

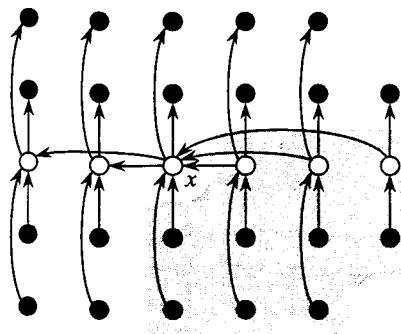


FIGURA 9.1 Análise do algoritmo SELECT. Os  $n$  elementos são representados por pequenos círculos, e cada grupo ocupa uma coluna. As medianas dos grupos são brancas, e a mediana de medianas está identificada como  $x$ . (Quando encontramos a mediana de um número par de elementos, usamos a mediana inferior.) São traçadas setas de elementos maiores para elementos menores e, a partir disso, podemos ver que 3 elementos em cada grupo de 5 elementos à direita de  $x$  são maiores que  $x$ , e 3 em cada grupo de 5 elementos à esquerda de  $x$  são menores que  $x$ . Os elementos maiores que  $x$  são mostrados sobre um plano de fundo sombreado

O algoritmo SELECT determina o  $i$ -ésimo menor elemento de um arranjo de entrada de  $n > 1$  elementos, executando as etapas a seguir. (Se  $n = 1$ , então SELECT simplesmente retorna seu único valor de entrada como o  $i$ -ésimo menor.)

1. Dividir os  $n$  elementos do arranjo de entrada em  $\lfloor n/5 \rfloor$  grupos de 5 elementos cada e no máximo um grupo formado pelos  $n \bmod 5$  elementos restantes.
2. Encontrar a mediana de cada um dos  $\lceil n/5 \rceil$  grupos, primeiro através da ordenação por inserção dos elementos de cada grupo (dos quais existem 5 no máximo), e depois escolhendo a mediana da lista ordenada de elementos de grupos.
3. Usar SELECT recursivamente para encontrar a mediana  $x$  das  $\lceil n/5 \rceil$  medianas localizadas na Etapa 2. (Se existe um número par de medianas, então, por nossa convenção,  $x$  é a mediana inferior.)
4. Partitionar o arranjo de entrada em torno da mediana de medianas  $x$ , usando uma versão modificada de PARTITION. Seja  $k$  uma unidade maior que o número de elementos no lado baixo da partição, de forma que  $x$  seja o  $k$ -ésimo menor elemento e existam  $n - k$  elementos no lado alto da partição.
5. Se  $i = k$ , então retornar  $x$ . Caso contrário, usar SELECT recursivamente para encontrar o  $i$ -ésimo menor elemento no lado baixo se  $i \leq k$ , ou então o  $(i - k)$ -ésimo menor elemento no lado alto, se  $i > k$ .

Para analisar o tempo de execução de SELECT, primeiro determinamos um limite inferior sobre o número de elementos que são maiores que o elemento de particionamento  $x$ . A Figura 9.1 é útil na visualização dessa contabilidade. Pelo menos metade das medianas encontradas na Etapa 2 é maior que<sup>1</sup> a mediana de medianas  $x$ . Portanto, pelo menos metade dos  $\lceil n/5 \rceil$  grupos contribui com 3 elementos maiores que  $x$ , exceto pelo único grupo que tem menos de 5 elementos se 5 não dividir  $n$  exatamente, e pelo único grupo contendo o próprio  $x$ .

Descontando esses dois grupos, segue-se que o número de elementos maiores que  $x$  é pelo menos

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6.$$

De modo semelhante, o número de elementos menores que  $x$  é no mínimo  $3n/10 - 6$ . Desse modo, no pior caso, SELECT é chamado recursivamente sobre no máximo  $7n/10 + 6$  elementos na Etapa 5.

Agora, podemos desenvolver uma recorrência para o tempo de execução do pior caso  $T(n)$  do algoritmo SELECT. As Etapas 1, 2 e 4 demoram o tempo  $O(n)$ . (A Etapa 2 consiste em  $O(n)$  chamadas de ordenação por inserção sobre conjuntos de tamanho  $O(1)$ .) A Etapa 3 demora o tempo  $T(\lceil n/5 \rceil)$ , e a Etapa 5 demora no máximo o tempo  $T(7n/10 + 6)$ , supondo-se que  $T$  seja monotonicamente crescente. Adotamos a hipótese, que a princípio parece sem motivo, de que qualquer entrada de 140 ou menos elementos exige o tempo  $O(1)$ ; a origem da constante mágica 140 ficará clara em breve. Portanto, podemos obter a recorrência

$$T(n) \leq T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 140, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{se } n > 140. \end{cases}$$

Mostramos que o tempo de execução é linear por substituição. Mais especificamente, mostraremos que  $T(n) \leq cn$  para alguma constante  $c$  grande o bastante e para todo  $n > 0$ . Começamos supondo que  $T(n) \leq cn$  para alguma constante  $c$  grande o bastante e para todo  $n \leq 140$ ; essa hipótese se mantém válida se  $c$  é suficientemente grande. Também escolhemos uma constante  $a$  tal que a função descrita pelo termo  $O(n)$  anterior (que descreve o componente não recursivo do tempo de execução do algoritmo) é limitado acima por  $an$  para todo  $n > 0$ . Substituindo essa hipótese induativa no lado direito da recorrência, obtemos

---

<sup>1</sup> Em consequência de nossa hipótese de que os números são distintos, podemos dizer “maior que” e “menor que” sem nos preocuparmos com a igualdade.

$$\begin{aligned}
T(n) &\leq c \lceil n/5 \rceil + c(7n/10 + 6) + an \\
&\leq cn/5 + c + 7cn/10 + 6c + an \\
&= 9cn/10 + 7c + an \\
&= cn + (-cn/10 + 7c + an),
\end{aligned}$$

que é no máximo  $cn$  se

$$-cn/10 + 7c + an \leq 0. \quad (9.2)$$

A desigualdade (9.2) é equivalente à desigualdade  $c \geq 10a(n/(n - 70))$  quando  $n > 70$ . Considerando que supomos  $n \geq 140$ , temos  $n/(n - 70) \leq 2$ , e assim a escolha de  $c \geq 20a$  satisfará à desigualdade (9.2). (Observe que não existe nada de especial sobre a constante 140; poderíamos substituí-la por qualquer inteiro estritamente maior que 70 e depois escolher  $c$  de acordo.) Então, o tempo de execução do pior caso de SELECT é linear.

Como em uma ordenação por comparação (ver Seção 8.1), SELECT e RANDOMIZED-SELECT descobrem informações sobre a ordem relativa de elementos apenas pela comparação de elementos. Vimos no Capítulo 8 que a ordenação exige o tempo  $\Omega(n \lg n)$  no modelo de comparação, mesmo na média (ver Problema 8-1). Os algoritmos de ordenação de tempo linear do Capítulo 8 fazem suposições sobre a entrada. Em contraste, os algoritmos de seleção de tempo linear deste capítulo não exigem quaisquer hipóteses sobre a entrada. Eles não estão sujeitos ao limite inferior  $\Omega(n \lg n)$  porque conseguem resolver o problema de seleção sem ordenação.

Desse modo, o tempo de execução é linear porque esses algoritmos não efetuam a ordenação; o comportamento de tempo linear não é um resultado de hipóteses sobre a entrada, como foi o caso para os algoritmos de ordenação do Capítulo 8. A ordenação requer o tempo  $\Omega(n \lg n)$  no modelo de comparação, mesmo na média (ver Problema 8-1), e assim o método de ordenação e indexação apresentado na introdução a este capítulo é assintoticamente ineficiente.

## Exercícios

### 9.3-1

No algoritmo SELECT, os elementos de entrada estão divididos em grupos de 5. O algoritmo funcionará em tempo linear se eles forem divididos em grupos de 7? Demonstre que SELECT não será executado em tempo linear se forem usados grupos de 3 elementos.

### 9.3-2

Analise SELECT para mostrar que, se  $n \geq 140$ , pelo menos  $\lceil n/4 \rceil$  elementos são maiores que a mediana de medianas  $x$  e pelo menos  $\lceil n/4 \rceil$  elementos são menores que  $x$ .

### 9.3-3

Mostre como quicksort pode ser desenvolvido para ser executado no tempo  $O(n \lg n)$  no pior caso.

### 9.3-4 \*

Suponha que um algoritmo utilize apenas comparações para encontrar o  $i$ -ésimo menor elemento em um conjunto de  $n$  elementos. Mostre que ele também pode encontrar os  $i - 1$  menores elementos e os  $n - i$  maiores elementos sem executar quaisquer comparações adicionais.

### 9.3-5

Dada uma sub-rotina de “caixa-preta” de mediana de tempo linear no pior caso, forneça um algoritmo simples de tempo linear que resolva o problema de seleção para uma estatística de ordem arbitrária.

### 9.3-6

Os  $k$ -ésimos **quantis** de um conjunto de  $n$  elementos são as  $k - 1$  estatísticas de ordem que dividem o conjunto ordenado em  $k$  conjuntos de igual tamanho (até dentro de 1). Forneça um algoritmo de tempo  $O(n \lg k)$  para listar os  $k$ -ésimos quantis de um conjunto.

### 9.3-7

Descreva um algoritmo de tempo  $O(n)$  que, dados um conjunto  $S$  de  $n$  números distintos e um inteiro positivo  $k \leq n$ , determine os  $k$  números em  $S$  que estão mais próximos da mediana de  $S$ .

### 9.3-8

Sejam  $X[1 \dots n]$  e  $Y[1 \dots n]$  dois arranjos, cada um contendo  $n$  números já em seqüência ordenada. Forneça um algoritmo de tempo  $O(\lg n)$  para localizar a mediana de todos os  $2n$  elementos nos arranjos  $X$  e  $Y$ .

### 9.3-9

O professor Olavo é consultor de uma empresa petrolífera que está planejando um grande oleoduto de leste para oeste através de um campo petrolífero de  $n$  poços. De cada poço, um oleoduto auxiliar deve ser conectado diretamente ao oleoduto principal ao longo de um caminho mais curto (para o norte ou para o sul), como mostra a Figura 9.2. Dadas as coordenadas  $x$  e  $y$  dos poços, de que modo o professor deve escolher a localização ótima do oleoduto principal (aquele que minimiza o comprimento total dos dutos auxiliares)? Mostre que a localização ótima pode ser determinada em tempo linear.

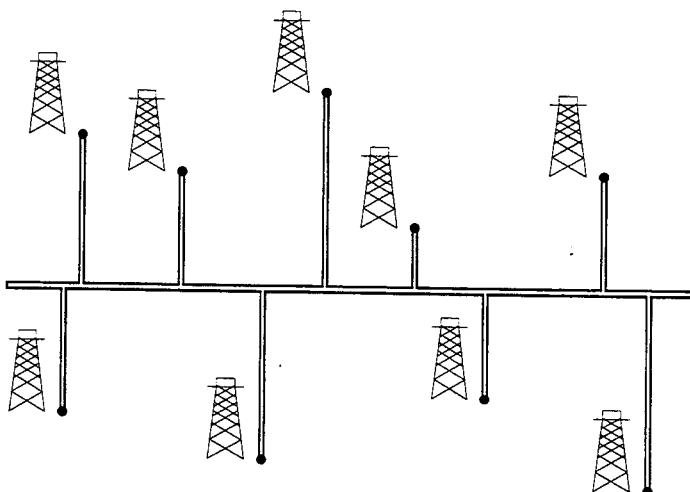


FIGURA 9.2 O professor Olavo precisa determinar a posição do oleoduto no sentido leste-oeste que minimiza o comprimento total dos dutos auxiliares norte-sul

## Problemas

### 9-1 Os $i$ maiores números em seqüência ordenada

Dado um conjunto de  $n$  números, queremos encontrar os  $i$  maiores em seqüência ordenada, usando um algoritmo baseado em comparação. Descubra o algoritmo que implementa cada um dos métodos a seguir com o melhor tempo de execução assintótico no pior caso e analise os tempos de execução dos algoritmos em termos de  $n$  e  $i$ .

- Classifique os números e liste os  $i$  maiores.
- Construa uma fila de prioridade a partir dos números e chame EXTRACT-MAX  $i$  vezes.
- Use um algoritmo de estatística de ordem para localizar o  $i$ -ésimo maior número, particionar e ordenar os  $i$  maiores números.

## 9-2 Mediana ponderada

Para  $n$  elementos distintos  $x_1, x_2, \dots, x_n$ , com pesos positivos  $w_1, w_2, \dots, w_n$  tais que  $\sum_{i=1}^n w_i = 1$ , a **mediana ponderada (inferior)** é o elemento  $x_k$  que satisfaz a

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

e

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}.$$

- a. Mostre que a mediana de  $x_1, x_2, \dots, x_n$  é a mediana ponderada dos  $x_i$  com pesos  $w_i = 1/n$  para  $i = 1, 2, \dots, n$ .
- b. Mostre como calcular a mediana ponderada de  $n$  elementos no tempo  $O(n \lg n)$  no pior caso usando ordenação.
- c. Mostre como calcular a mediana ponderada no tempo  $\Theta(n)$  no pior caso, usando um algoritmo de mediana de tempo linear como SELECT da Seção 9.3.

O **problema da localização da agência postal** é definido como a seguir. Temos  $n$  pontos  $p_1, p_2, \dots, p_n$  com pesos associados  $w_1, w_2, \dots, w_n$ . Desejamos encontrar um ponto  $p$  (não necessariamente um dos pontos de entrada) que minimize o somatório  $\sum_{i=1}^n w_i d(p, p_i)$ , onde  $d(a, b)$  é a distância entre os pontos  $a$  e  $b$ .

- d. Mostre que a mediana ponderada é uma solução melhor para o problema da localização de agência postal unidimensional, no qual os pontos são simplesmente números reais e a distância entre os pontos  $a$  e  $b$  é  $d(a, b) = |a - b|$ .
- e. Encontre a melhor solução para o problema de localização da agência postal bidimensional, no qual os pontos são pares de coordenadas  $(x, y)$  e a distância entre os pontos  $a = (x_1, y_1)$  e  $b = (x_2, y_2)$  é a **distância de Manhattan** dada por  $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$ .

## 9-3 Estatísticas de ordem menores

Mostramos que o número  $T(n)$  de comparações no pior caso usadas por SELECT para selecionar a  $i$ -ésima estatística de ordem de  $n$  números satisfaz a  $T(n) = \Theta(n)$ , mas a constante oculta pela notação  $\Theta$  é bastante grande. Quando  $i$  é pequena em relação a  $n$ , podemos implementar um procedimento diferente que utiliza SELECT como uma sub-rotina, mas que efetua menos comparações no pior caso.

- a. Descreva um algoritmo que utilize  $U_i(n)$  comparações para encontrar o  $i$ -ésimo menor de  $n$  elementos, onde

$$U_i(n) = T(n) = \begin{cases} T(n) & \text{se } i \geq n/2 , \\ \lfloor n/2 \rfloor + U_i(\lfloor n/2 \rfloor) + T(2i) & \text{em caso contrário.} \end{cases}$$

(Sugestão: Comece com  $\lfloor n/2 \rfloor$  comparações de pares disjuntos e efetue a recursão sobre o conjunto que contém o menor elemento de cada par.)

- b. Mostre que, se  $i < n/2$ , então  $U_i(n) = n + O(T(2i) \lg(n/i))$ .
- c. Mostre que, se  $i$  é uma constante menor que  $n/2$ , então  $U_i(n) = n + O(\lg n)$ .
- d. Mostre que, se  $i = n/k$  para  $k \geq 2$ , então  $U_i(n) = n + O(T(2n/k) \lg k)$ .

## Notas do capítulo

O algoritmo de tempo linear no pior caso para localização da mediana foi criado por Blum, Floyd, Pratt, Rivest e Tarjan [43]. A versão de tempo médio mais rápido se deve a Hoare [146]. Floyd e Rivest [92] desenvolveram uma versão de tempo médio melhor que efetua a partição em torno de um elemento selecionado recursivamente a partir de uma amostra pequena dos elementos.

Ainda não se sabe exatamente quantas comparações são necessárias para determinar a mediana. Um limite inferior de  $2n$  comparações para a localização de medianas foi dado por Bent e John [38]. Um limite superior de  $3n$  foi dado por Schonhage, Paterson e Pippenger [265]. Dor e Zwick [79] fizeram melhorias nesses dois limites; seu limite superior é ligeiramente menor que  $2,95n$  e o limite inferior é ligeiramente maior que  $2n$ . Paterson [239] descreve esses resultados juntamente com outro trabalho relacionado.



---

## *Parte III*

# *Estruturas de dados*

### **Introdução**

Os conjuntos são tão fundamentais para a ciência da computação quanto o são para a matemática. Enquanto os conjuntos matemáticos são invariáveis, os conjuntos manipulados por algoritmos podem crescer, encolher ou sofrer outras mudanças ao longo do tempo. Chamamos tais conjuntos de conjuntos **dinâmicos**. Os próximos cinco capítulos apresentam algumas técnicas básicas para representar conjuntos dinâmicos finitos e para manipular esses conjuntos em um computador.

Os algoritmos podem exigir vários tipos diferentes de operações a serem executadas sobre conjuntos. Por exemplo, muitos algoritmos precisam apenas da capacidade de inserir elementos em, eliminar elementos de, e testar a pertinência de elementos a um conjunto. Um conjunto dinâmico que admite essas operações é chamado **dicionário**. Outros algoritmos exigem operações mais complicadas. Por exemplo, filas de prioridade mínima, que foram introduzidas no Capítulo 6 no contexto da estrutura de dados do tipo heap (monte), admitem as operações de inserção de um elemento em e de extração do menor elemento de um conjunto. A melhor maneira de implementar um conjunto dinâmico depende das operações que devem ser admitidas.

### **Os elementos de um conjunto dinâmico**

Em uma implementação típica de um conjunto dinâmico, cada elemento é representado por um objeto cujos campos podem ser examinados e manipulados se tivermos um ponteiro para o objeto. (A Seção 10.3 discute a implementação de objetos e ponteiros em ambientes de programação que não contêm esses objetos e ponteiros como tipos de dados básicos.) Alguns tipos de conjuntos dinâmicos pressupõem que um dos campos do objeto é um campo de **chave** de identificação. Se as chaves são todas diferentes, podemos imaginar o conjunto dinâmico como um conjunto de valores de chaves. O objeto pode conter **dados satélite**, que são transportados em campos de outro objeto, mas não são utilizados de outro modo pela implementação do conjunto. Ele também pode ter campos que são manipulados pelas operações de conjuntos; esses campos podem conter dados ou ponteiros para outros objetos no conjunto.

Alguns conjuntos dinâmicos pressupõem que as chaves são extraídas de um conjunto totalmente ordenado, como o dos números reais, ou ainda o conjunto de todas as palavras que se-

guem a ordenação alfabética usual. (Um conjunto totalmente ordenado satisfaz à propriedade de tricotomia, definida no Capítulo 3.) Uma ordenação total nos permite definir o elemento mínimo do conjunto, por exemplo, ou falar do próximo elemento maior que um dado elemento em um conjunto.

## Operações sobre conjuntos dinâmicos

As operações sobre um conjunto dinâmico podem ser agrupadas em duas categorias: **consultas**, que simplesmente retornam informações sobre o conjunto, e **operações de modificação**, que alteram o conjunto. Aqui está uma lista de operações típicas. Qualquer aplicação específica normalmente exigirá que apenas algumas dessas operações sejam implementadas.

### SEARCH( $S, k$ )

Uma consulta que, dado um conjunto  $S$  e um valor de chave  $k$ , retorna um ponteiro  $x$  para um elemento em  $S$  tal que  $chave[x] = k$ , ou NIL se nenhum elemento desse tipo pertencer a  $S$ .

### INSERT( $S, x$ )

Uma operação de modificação que aumenta o conjunto  $S$  com o elemento apontado por  $x$ . Normalmente, supomos que quaisquer campos no elemento  $x$  necessários para a implementação do conjunto já tenham sido inicializados.

### DELETE( $S, x$ )

Uma operação de modificação que, dado um ponteiro  $x$  para um elemento no conjunto  $S$ , remove  $x$  de  $S$ . (Observe que essa operação utiliza um ponteiro para um elemento  $x$ , não um valor de chave.)

### MINIMUM( $S$ )

Uma consulta sobre um conjunto totalmente ordenado  $S$  que retorna um ponteiro para o elemento de  $S$  com a menor chave.

### MAXIMUM( $S$ )

Uma consulta sobre um conjunto totalmente ordenado  $S$  que retorna um ponteiro para o elemento de  $S$  com a maior chave.

### SUCCESSOR( $S, x$ )

Uma consulta que, dado um elemento  $x$  cuja chave é de um conjunto totalmente ordenado  $S$ , retorna um ponteiro para o maior elemento seguinte em  $S$ , ou NIL se  $x$  é o elemento máximo.

### PREDECESSOR( $S, x$ )

Uma consulta que, dado um elemento  $x$  cuja chave é de um conjunto totalmente ordenado  $S$ , retorna um ponteiro para o menor elemento seguinte em  $S$ , ou NIL se  $x$  é o elemento mínimo.

As consultas SUCCESSOR e PREDECESSOR freqüentemente são estendidas a conjuntos com chaves não-distintas. Para um conjunto sobre  $n$  chaves, a suposição normal é que uma chamada a MINIMUM seguida por  $n - 1$  chamadas a SUCCESSOR enumera os elementos no conjunto em seqüência ordenada.

O tempo empregado para executar uma operação de conjunto é medido normalmente em termos do tamanho do conjunto dado como um de seus argumentos. Por exemplo, o Capítulo 13 descreve uma estrutura de dados que pode admitir quaisquer das operações listadas anteriormente sobre um conjunto de tamanho  $n$  no tempo  $O(\lg n)$ .

## Visão geral da Parte III

Os Capítulos 10 a 14 descrevem várias estruturas de dados que podem ser usadas para implementar conjuntos dinâmicos; muitas dessas estruturas serão utilizadas mais tarde para construir algoritmos eficientes destinados à solução de uma variedade de problemas. Outra estrutura de dados importante – o heap (ou monte) – já foi apresentada no Capítulo 6.

O Capítulo 10 apresenta os detalhes essenciais do trabalho com estruturas de dados simples como pilhas, filas, listas ligadas e árvores enraizadas. Ele também mostra como objetos e ponteiros podem ser implementados em ambientes de programação que não os admitem como primitivas. Grande parte desse material deve ser familiar para qualquer pessoa que tenha freqüentado um curso introdutório de programação.

O Capítulo 11 introduz as tabelas hash, que admitem as operações de dicionário INSERT, DELETE e SEARCH. No pior caso, o hash exige o tempo  $1(n)$  para executar uma operação SEARCH, mas o tempo esperado para operações sobre tabelas hash é  $O(1)$ . A análise do hash se baseia na probabilidade, mas a maior parte do capítulo não requer nenhuma experiência no assunto.

As árvores de pesquisa binária, que são focalizadas no Capítulo 12, admitem todas as operações sobre conjuntos dinâmicos listadas anteriormente. No pior caso, cada operação demora um tempo  $1(n)$  em uma árvore com  $n$  elementos, mas, em uma árvore de pesquisa binária construída aleatoriamente, o tempo esperado para cada operação é  $O(\lg n)$ . As árvores de pesquisa binária servem como base para muitas outras estruturas de dados.

As árvores vermelho-preto, uma variante de árvores de pesquisa binária, são introduzidas no Capítulo 13. Diferentes das árvores de pesquisa binária comuns, as árvores vermelho-preto oferecem a garantia de funcionar bem: as operações demoram o tempo  $O(\lg n)$  no pior caso. Uma árvore vermelho-preto é uma árvore de pesquisa balanceada; o Capítulo 18 apresenta outro tipo de árvore de pesquisa balanceada, chamada árvore B. Embora a mecânica das árvores vermelho-preto seja um pouco complicada, você pode descobrir a maior parte de suas propriedades a partir do capítulo, sem estudar a mecânica em detalhes. Apesar disso, o exame do código pode ser bastante instrutivo.

No Capítulo 14, mostramos como aumentar as árvores vermelho-preto para oferecer suporte a operações diferentes das operações básicas listadas antes. Primeiro, aumentamos essas árvores de modo a podermos manter dinamicamente estatísticas de ordem para um conjunto de chaves. Em seguida, nós as aumentamos de modo diferente, a fim de manter intervalos de números reais.



---

# *Capítulo 10*

## *Estruturas de dados elementares*

Neste capítulo, examinaremos a representação de conjuntos dinâmicos por estruturas de dados simples que usam ponteiros. Embora muitas estruturas de dados complexas possam ser modeladas com a utilização de ponteiros, apresentaremos apenas as estruturas rudimentares: pilhas, filas, listas ligadas e árvores enraizadas. Também discutiremos um método pelo qual objetos e ponteiros podem ser sintetizados a partir de arranjos.

### **10.1 Pilhas e filas**

As pilhas e filas são conjuntos dinâmicos nos quais o elemento removido do conjunto pela operação DELETE é especificado previamente. Em uma **pilha**, o elemento eliminado do conjunto é o mais recentemente inserido: a pilha implementa uma norma de **último a entrar, primeiro a sair**, ou **LIFO** (last-in, first-out). De modo semelhante, em uma **fila**, o elemento eliminado é sempre o que esteve no conjunto pelo tempo mais longo: a fila implementa uma norma de **primeiro a entrar, primeiro a sair**, ou **FIFO** (first-in, first-out). Existem vários modos eficientes de implementar pilhas e filas em um computador. Nesta seção, mostraremos como usar um arranjo simples para implementar cada uma dessas estruturas.

#### **Pilhas**

A operação INSERT sobre uma pilha é chamada com freqüência PUSH, e a operação DELETE, que não toma um argumento de elemento, é freqüentemente chamada POP. Esses nomes são alusões a pilhas físicas, como as pilhas de pratos usados em restaurantes. A ordem em que os pratos são retirados da pilha é o oposto da ordem em que eles são colocados sobre a pilha e, como consequência, apenas o prato do topo está acessível.

Como mostra a Figura 10.1, podemos implementar uma pilha de no máximo  $n$  elementos com um arranjo  $S[1..n]$ . O arranjo tem um atributo  $\text{topo}[S]$  que realiza a indexação do elemento inserido mais recentemente. A pilha consiste nos elementos  $S[1 .. \text{topo}[S]]$ , onde  $S[1]$  é o elemento na parte inferior da pilha e  $S[\text{topo}[S]]$  é o elemento na parte superior (ou no topo).

Quando  $\text{topo}[S] = 0$ , a pilha não contém nenhum elemento e está **vazia**. É possível testar se a pilha está vazia, através da operação de consulta STACK-EMPTY. Se uma pilha vazia sofre uma operação de extração, dizemos que a pilha tem um **estouro negativo**, que é normalmente um erro. Se  $\text{topo}[S]$  excede  $n$ , a pilha tem um **estouro positivo**. (Em nossa implementação de pseudocódigo, não nos preocuparemos com o estouro de pilhas.)

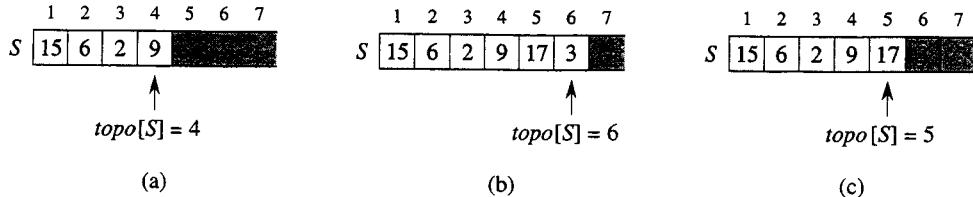


FIGURA 10.1 Uma implementação de arranjo de uma pilha  $S$ . Os elementos da pilha só aparecem nas posições levemente sombreadas. (a) A pilha  $S$  tem 4 elementos. O elemento do topo é 9. (b) A pilha  $S$  após as chamadas  $\text{PUSH}(S, 17)$  e  $\text{PUSH}(S, 3)$ . (c) A pilha  $S$  após a chamada  $\text{POP}(S)$  retornou o elemento 3, que é o elemento mais recentemente inserido na pilha. Embora o elemento 3 ainda apareça no arranjo, ele não está mais na pilha; o elemento do topo é o elemento 17

Cada uma das operações sobre pilhas pode ser implementada com algumas linhas de código.

#### STACK-EMPTY( $S$ )

```

1 if  $\text{topo}[S] = 0$ 
2   then return TRUE
3   else return FALSE

```

#### PUSH( $S, x$ )

```

1  $\text{topo}[S] \leftarrow \text{topo}[S] + 1$ 
2  $S[\text{topo}[S]] \leftarrow x$ 

```

#### POP( $S$ )

```

1 if STACK-EMPTY( $S$ )
2   then error "underflow"
3   else  $\text{topo}[S] \leftarrow \text{topo}[S] - 1$ 
4     return  $S[\text{topo}[S] + 1]$ 

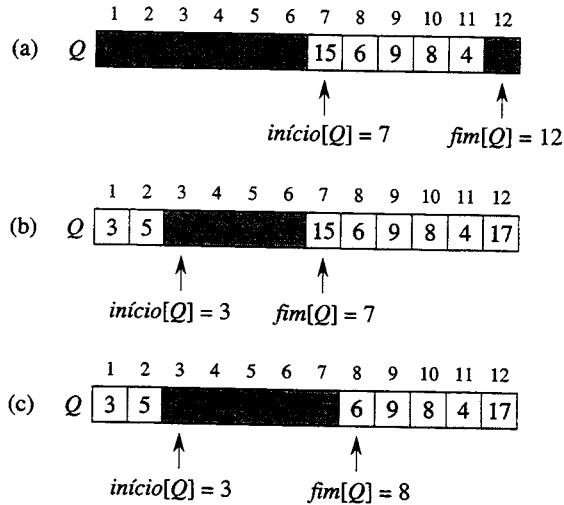
```

A Figura 10.1 mostra os efeitos das operações de modificação PUSH (EMPILHAR) e POP (DESEMPILHAR). Cada uma das três operações sobre pilhas demora o tempo  $O(1)$ .

## Filas

Chamamos a operação INSERT sobre uma fila de ENQUEUE (ENFILEIRAR), e também a operação DELETE de DEQUEUE (DESINFILEIRAR); como a operação sobre pilhas POP, DEQUEUE não tem nenhum argumento de elemento. A propriedade FIFO de uma fila faz com que ela opere como uma fileira de pessoas no posto de atendimento da previdência social. A fila tem um *índice* (ou cabeça) e um *fim* (ou cauda). Quando um elemento é colocado na fila, ele ocupa seu lugar no fim da fila, como um aluno recém-chegado que ocupa um lugar no final da fileira. O elemento retirado da fila é sempre aquele que está no início da fila, como o aluno que se encontra no começo da fileira e que esperou por mais tempo. (Felizmente, não temos de nos preocupar com a possibilidade de elementos computacionais “furarem” a fila.)

A Figura 10.2 mostra um modo de implementar uma fila de no máximo  $n - 1$  elementos usando um arranjo  $Q[1..n]$ . A fila tem um atributo  $\text{início}[Q]$  que indexa ou aponta para seu início. O atributo  $\text{fim}[Q]$  realiza a indexação da próxima posição na qual um elemento recém-chegado será inserido na fila. Os elementos na fila estão nas posições  $\text{início}[Q], \text{início}[Q] + 1, \dots, \text{fim}[Q] - 1$ , onde “retornamos”, no sentido de que a posição 1 segue imediatamente a posição  $n$  em uma ordem circular. Quando  $\text{início}[Q] = \text{fim}[Q]$ , a fila está vazia. Inicialmente, temos  $\text{início}[Q] = \text{fim}[Q] = 1$ . Quando a fila está vazia, uma tentativa de retirar um elemento da fila provoca o estouro negativo da fila. Quando  $\text{início}[Q] = \text{fim}[Q] + 1$ , a fila está cheia, e uma tentativa de colocar um elemento na fila provoca o estouro positivo da fila.



**FIGURA 10.2** Uma fila implementada com a utilização de um arranjo  $Q[1..12]$ . Os elementos da fila aparecem apenas nas posições levemente sombreadas. (a) A fila tem 5 elementos, nas localizações  $Q[7..11]$ . (b) A configuração da fila depois das chamadas  $\text{ENQUEUE}(Q, 17)$ ,  $\text{ENQUEUE}(Q, 3)$  e  $\text{ENQUEUE}(Q, 5)$ . (c) A configuração da fila depois da chamada  $\text{DEQUEUE}(Q)$  retorna o valor de chave 15 que se encontrava anteriormente no início da fila. O novo início tem a chave 6

Em nossos procedimentos  $\text{ENQUEUE}$  e  $\text{DEQUEUE}$ , a verificação de erros de estouro negativo (underflow) e estouro positivo (overflow) foi omitida. (O Exercício 10.1-4 lhe pede para fornecer o código que efetua a verificação dessas duas condições de erro.)

$\text{ENQUEUE}(Q, x)$

- 1  $Q[\text{fim}[Q]] \leftarrow x$
- 2 **if**  $\text{fim}[Q] = \text{comprimento}[Q]$
- 3   **then**  $\text{fim}[Q] \leftarrow 1$
- 4   **else**  $\text{fim}[Q] \leftarrow \text{fim}[Q] + 1$

$\text{DEQUEUE}(Q)$

- 1  $x \leftarrow Q[\text{início}[Q]]$
- 2 **if**  $\text{início}[Q] = \text{comprimento}[Q]$
- 3   **then**  $\text{início}[Q] \leftarrow 1$
- 4   **else**  $\text{início}[Q] \leftarrow \text{início}[Q] + 1$
- 5 **return**  $x$

A Figura 10.2 mostra os efeitos das operações  $\text{ENQUEUE}$  e  $\text{DEQUEUE}$ . Cada operação demora o tempo  $O(1)$ .

## Exercícios

### 10.1-1

Usando a Figura 10.1 como modelo, ilustre o resultado de cada operação na seqüência  $\text{PUSH}(S, 4)$ ,  $\text{PUSH}(S, 1)$ ,  $\text{PUSH}(S, 3)$ ,  $\text{POP}(S)$ ,  $\text{PUSH}(S, 8)$  e  $\text{POP}(S)$  sobre uma pilha  $S$  inicialmente vazia armazenada no arranjo  $S[1 .. 6]$ .

### 10.1-2

Explique como implementar duas pilhas em um único arranjo  $A[1 .. n]$  de tal modo que nenhuma das pilhas sofra um estouro positivo, a menos que o número total de elementos em ambas as pilhas juntas seja  $n$ . As operações  $\text{PUSH}$  e  $\text{POP}$  devem ser executadas no tempo  $O(1)$ .

### 10.1-3

Usando a Figura 10.2 como modelo, ilustre o resultado de cada operação na seqüência ENQUEUE( $Q, 4$ ), ENQUEUE( $Q, 1$ ), ENQUEUE( $Q, 3$ ), DEQUEUE( $Q$ ), ENQUEUE( $Q, 8$ ) e DEQUEUE( $Q$ ) sobre uma fila  $Q$  inicialmente vazia armazenada no arranjo  $Q[1 .. 6]$ .

### 10.1-4

Reescreva ENQUEUE e DEQUEUE para detectar o estouro negativo e o estouro positivo de uma fila.

### 10.1-5

Enquanto uma pilha permite a inserção e a eliminação de elementos em apenas uma extremidade e uma fila permite a inserção em uma extremidade e a eliminação na outra extremidade, uma **deque** (double-ended queue, ou fila de extremidade dupla) permite a inserção e a eliminação em ambas as extremidades. Escreva quatro procedimentos de tempo  $O(1)$  para inserir elementos e eliminar elementos de ambas as extremidades de uma deque construída a partir de um arranjo.

### 10.1-6

Mostre como implementar uma fila usando duas pilhas. Analise o tempo de execução das operações sobre filas.

### 10.1-7

Mostre como implementar uma pilha usando duas filas. Analise o tempo de execução das operações sobre pilhas.

## 10.2 Listas ligadas

Uma **lista ligada** é uma estrutura de dados em que os objetos estão organizados em uma ordem linear. Entretanto, diferente de um arranjo, no qual a ordem linear é determinada pelos índices do arranjo, a ordem em uma lista ligada é determinada por um ponteiro em cada objeto. As listas ligadas fornecem uma representação simples e flexível para conjuntos dinâmicos, admitindo (embora não necessariamente de modo eficiente) todas as operações listadas na introdução à Parte III, seção “Operações sobre conjuntos dinâmicos”.

Como mostra a Figura 10.3, cada elemento de uma **lista duplamente ligada**  $L$  é um objeto com um campo de *chave* e dois outros campos de ponteiros: *próximo* e *anterior*. O objeto também pode conter outros dados satélite. Sendo dado um elemento  $x$  na lista,  $\text{próximo}[x]$  aponta para seu sucessor na lista ligada, e  $\text{anterior}[x]$  aponta para seu predecessor. Se  $\text{anterior}[x] = \text{NIL}$ , o elemento  $x$  não tem nenhum predecessor e portanto é o primeiro elemento, ou *início*, da lista. Se  $\text{próximo}[x] = \text{NIL}$ , o elemento  $x$  não tem nenhum sucessor e assim é o último elemento, ou *fim*, da lista. Um atributo  $\text{início}[L]$  aponta para o primeiro elemento da lista. Se  $\text{início}[L] = \text{NIL}$ , a lista está vazia.

Uma lista pode ter uma entre várias formas. Ela pode ser simplesmente ligada ou duplamente ligada, pode ser ordenada ou não, e pode ser circular ou não. Se uma lista é **simplesmente ligada**, omitimos o ponteiro *anterior* em cada elemento. Se uma lista é **ordenada**, a ordem linear da lista corresponde à ordem linear de chaves armazenadas em elementos da lista; o elemento mínimo é o início da lista, e o elemento máximo é o fim. Se a lista é **não ordenada**, os elementos podem aparecer em qualquer ordem. Em uma **lista circular**, o ponteiro *anterior* do início da lista aponta para o fim, e o ponteiro *próximo* do fim da lista aponta para o início. Desse modo, a lista pode ser vista como um anel de elementos. No restante desta seção, supomos que as listas com as quais estamos trabalhando são listas não ordenadas e duplamente ligadas.

## Como pesquisar em uma lista ligada

O procedimento  $\text{LIST-SEARCH}(L, k)$  encontra o primeiro elemento com a chave  $k$  na lista  $L$  através de uma pesquisa linear simples, retornando um ponteiro para esse elemento. Se nenhum objeto com a chave  $k$  aparecer na lista, então  $\text{NIL}$  será retornado. No caso da lista ligada da Figura 10.3(a), a chamada  $\text{LIST-SEARCH}(L, 4)$  retorna um ponteiro para o terceiro elemento, e a chamada  $\text{LIST-SEARCH}(L, 7)$  retorna  $\text{NIL}$ .

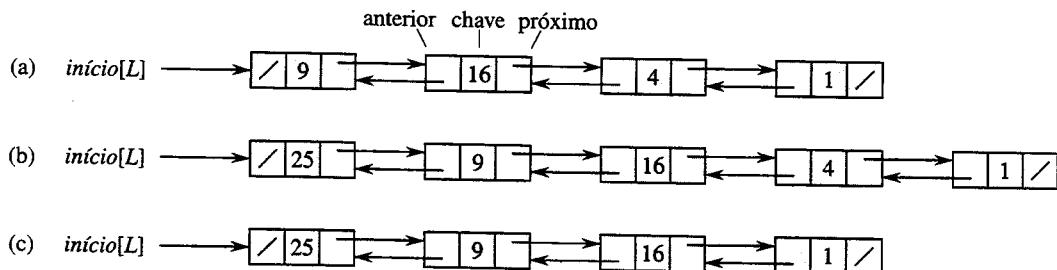


FIGURA 10.3 (a) Uma lista duplamente ligada  $L$  representando o conjunto dinâmico  $\{1, 4, 9, 16\}$ . Cada elemento na lista é um objeto com campos para a chave e ponteiros (mostrados por setas) para os objetos próximo e anterior. O campo *próximo* do fim e o campo *anterior* do início são  $\text{NIL}$ , indicados por uma barra em diagonal. O atributo  $\text{início}[L]$  aponta para o início. (b) Seguindo a execução de  $\text{LIST-INSERT}(L, x)$ , onde  $\text{chave}[x] = 25$ , a lista ligada tem um novo objeto com chave 25 como o novo início. Esse novo objeto aponta para o antigo início com chave 9. (c) O resultado da chamada subsequente  $\text{LIST-DELETE}(L, x)$ , onde  $x$  aponta para o objeto com chave 4

$\text{LIST-SEARCH}(L, k)$

- 1  $x \leftarrow \text{início}[L]$
- 2 **while**  $x \neq \text{NIL}$  e  $\text{chave}[x] \neq k$
- 3   **do**  $x \leftarrow \text{próximo}[x]$
- 4 **return**  $x$

Para pesquisar em uma lista de  $n$  objetos, o procedimento  $\text{LIST-SEARCH}$  demora o tempo  $\Theta(n)$  no pior caso, pois pode ter de pesquisar a lista inteira.

## Inserção de elementos em uma lista ligada

Dado um elemento  $x$  cujo campo de *chave* já foi definido, o procedimento  $\text{LIST-INSERT}$  “junta”  $x$  à frente da lista ligada, como mostra a Figura 10.3(b).

$\text{LIST-INSERT}(L, x)$

- 1  $\text{próximo}[x] \leftarrow \text{início}[L]$
- 2 **if**  $\text{início}[L] \neq \text{NIL}$
- 3   **then**  $\text{anterior}[\text{início}[L]] \leftarrow x$
- 4  $\text{início}[L] \leftarrow x$
- 5  $\text{anterior}[x] \leftarrow \text{NIL}$

O tempo de execução para  $\text{LIST-INSERT}$  sobre uma lista de  $n$  elementos é  $O(1)$ .

## Eliminação de elementos de uma lista ligada

O procedimento  $\text{LIST-DELETE}$  remove um elemento  $x$  de uma lista ligada  $L$ . Ele deve receber um ponteiro para  $x$ , e depois “retirar”  $x$  da lista, atualizando os ponteiros. Se desejarmos eliminar um elemento com uma determinada chave, deveremos primeiro chamar  $\text{LIST-SEARCH}$ , a fim de recuperar um ponteiro para o elemento.

```

LIST-DELETE( $L, x$ )
1 if  $anterior[x] \neq NIL$ 
2   then  $próximo[anterior[x]] \leftarrow próximo[x]$ 
3   else  $início[L] \leftarrow próximo[x]$ 
4 if  $próximo[x] \neq NIL$ 
5   then  $anterior[próximo[x]] \leftarrow anterior[x]$ 

```

A Figura 10.3(c) mostra como um elemento é eliminado de uma lista ligada. LIST-DELETE é executado no tempo  $O(1)$  mas, se desejarmos eliminar um elemento com uma dada chave, será necessário o tempo  $\Theta(n)$  no pior caso, porque primeiro devemos chamar LIST-SEARCH.

## Sentinelas

O código para LIST-DELETE seria mais simples se pudéssemos ignorar as condições limite no início e no fim da lista.

```

LIST-DELETE'( $L, x$ )

```

```

1  $próximo[anterior[x]] \leftarrow próximo[x]$ 
2  $anterior[próximo[x]] \leftarrow anterior[x]$ 

```

Uma **sentinela** é um objeto fictício que nos permite simplificar condições limites. Por exemplo, vamos supor que fornecemos com a lista  $L$  um objeto  $nulo[L]$  que representa NIL, mas tem todos os campos dos outros elementos da lista. Onde quer que tenhamos uma referência NIL no código da lista, vamos substituí-la por uma referência à sentinela  $nulo[L]$ . Como vemos na Figura 10.4, isso transforma uma lista duplamente ligada normal em uma **lista circular, duplamente ligada com uma sentinela**, na qual a sentinela  $nulo[L]$  é colocada entre o início e o fim; o campo  $próximo[nulo[L]]$  aponta para o início da lista, enquanto  $anterior[nulo[L]]$  aponta para o fim. De modo semelhante, tanto o campo  $próximo$  do fim quanto o campo  $anterior$  do início apontam para  $nulo[L]$ . Tendo em vista que  $próximo[nulo[L]]$  aponta para o início, podemos eliminar totalmente o atributo  $início[L]$ , substituindo as referências a ele por referências a  $próximo[nulo[L]]$ . Uma lista vazia consiste apenas na sentinela, pois tanto  $próximo[nulo[L]]$  quanto  $anterior[nulo[L]]$  podem ser definidos como  $nulo[L]$ .

O código para LIST-SEARCH permanece o mesmo de antes, mas tem as referências a NIL e  $início[L]$  modificadas do modo especificado antes.

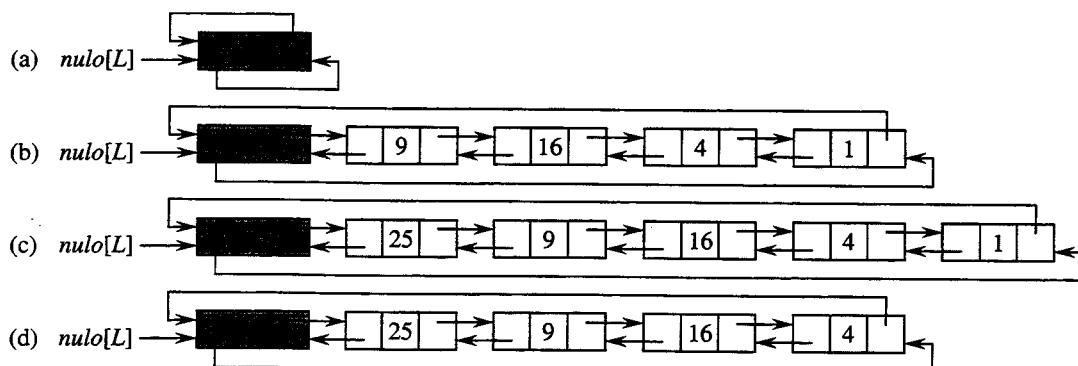


FIGURA 10.4 Uma lista circular duplamente ligada com uma sentinela. A sentinela  $nulo[L]$  aparece entre o início e o fim. O atributo  $início[L]$  não é mais necessário, pois podemos obter acesso ao início da lista por  $próximo[nulo[L]]$ . (a) Uma lista vazia. (b) A lista ligada da Figura 10.3(a), com chave 9 no início e chave 1 no fim. (c) A lista após a execução de LIST-INSERT'( $L, x$ ), onde  $chave[x] = 25$ . O novo objeto se torna o início da lista. (d) A lista após a eliminação do objeto com chave 1. O novo fim é o objeto com chave 4

```

LIST-SEARCH'(L, k)
1  $x \leftarrow \text{próximo}[\text{nulo}[L]]$ 
2 while  $x \neq \text{nulo}[L]$  e  $\text{chave}[x] \neq k$ 
3   do  $x \leftarrow \text{próximo}[x]$ 
4 return  $x$ 

```

Usamos o procedimento de duas linhas LIST-DELETE' para eliminar um elemento da lista. Utilizamos o procedimento a seguir para inserir um elemento na lista.

```

LIST-INSERT'(L, x)
1  $\text{próximo}[x] \leftarrow \text{próximo}[\text{nulo}[L]]$ 
2  $\text{anterior}[\text{próximo}[\text{nulo}[L]]] \leftarrow x$ 
3  $\text{próximo}[\text{nulo}[L]] \leftarrow x$ 
4  $\text{anterior}[x] \leftarrow \text{nulo}[L]$ 

```

A Figura 10.4 mostra os efeitos de LIST-INSERT' e LIST-DELETE' sobre uma amostra de lista.

As sentinelas raramente reduzem os limites assintóticos de tempo de operações de estrutura de dados, mas podem reduzir fatores constantes. O ganho de se utilizar sentinelas dentro de loops em geral é uma questão de clareza de código, em vez de velocidade; por exemplo, o código da lista ligada é simplificado pelo uso de sentinelas, mas pouparamos apenas o tempo  $O(1)$  nos procedimentos LIST-INSERT' e LIST-DELETE'. Contudo, em outras situações, o uso de sentinelas ajuda a tornar mais compacto o código em um loop, reduzindo assim o coeficiente de, digamos,  $n$  ou  $n^2$  no tempo de execução.

As sentinelas não devem ser usadas indiscriminadamente. Se houver muitas listas pequenas, o espaço de armazenamento extra usado por suas sentinelas poderá representar um desperdício significativo de memória. Neste livro, só utilizaremos sentinelas quando elas realmente simplificarem o código.

## Exercícios

### 10.2-1

A operação sobre conjuntos dinâmicos INSERT pode ser implementada sobre uma lista simplesmente ligada em tempo  $O(1)$ ? E no caso de DELETE?

### 10.2-2

Implemente uma pilha usando uma lista simplesmente ligada  $L$ . As operações PUSH e POP ainda devem demorar o tempo  $O(1)$ .

### 10.2-3

Implemente uma fila através de uma lista simplesmente ligada  $L$ . As operações ENQUEUE e DEQUEUE ainda devem demorar o tempo  $O(1)$ .

### 10.2-4

Como está escrita, cada iteração de loop no procedimento LIST-SEARCH' exige dois testes: um para  $x \neq \text{nulo}[L]$  e um para  $\text{chave}[x] \neq k$ . Mostre como eliminar o teste para  $x \neq \text{nulo}[L]$  em cada iteração.

### 10.2-5

Implemente as operações de dicionário INSERT, DELETE e SEARCH, usando listas circulares simplesmente ligadas. Quais são os tempos de execução dos seus procedimentos?

### 10.2-6

A operação sobre conjuntos dinâmicos UNION utiliza como entrada dois conjuntos disjuntos  $S_1$  e  $S_2$  e retorna um conjunto  $S = S_1 \cup S_2$  que consiste em todos os elementos de  $S_1$  e  $S_2$ . Os conjuntos  $S_1$  e  $S_2$  são normalmente destruídos pela operação. Mostre como oferecer suporte a UNION no tempo  $O(1)$  usando uma estrutura de dados de lista apropriada.

### 10.2-7

Forneça um procedimento não recursivo de tempo  $\Theta(n)$  que inverta uma lista simplesmente ligada de  $n$  elementos. O procedimento não deve usar nada mais além do espaço de armazenamento constante necessário para a própria lista.

### 10.2-8 \*

**Explique como implementar listas duplamente ligadas usando apenas um valor de ponteiro  $np[x]$  por item, em lugar dos dois valores usuais (*próximo* e *anterior*).** Suponha que todos os valores de ponteiros possam ser interpretados como inteiros de  $k$  bits e defina  $np[x]$  como  $np[x] = \text{próximo}[x] \text{ XOR } \text{anterior}[x]$ , o “ou exclusivo” de  $k$  bits de *próximo*[ $x$ ] e *anterior*[ $x$ ]. (O valor NIL é representado por 0.) Certifique-se de descrever as informações necessárias para obter acesso ao início da lista. Mostre como implementar as operações SEARCH, INSERT e DELETE em tal lista. Mostre também como inverter tal lista em tempo  $O(1)$ .

## 10.3 Implementação de ponteiros e objetos

De que modo implementamos ponteiros e objetos em linguagens como Fortran, que não os oferecem? Nesta seção, veremos duas maneiras de implementar estruturas de dados ligadas sem um tipo de dados ponteiro explícito. Sintetizaremos objetos e ponteiros a partir de arranjos e índices de arranjos.

### Uma representação de objetos com vários arranjos

Podemos representar uma coleção de objetos que têm os mesmos campos usando um arranjo para cada campo. Como exemplo, a Figura 10.5 mostra como podemos implementar a lista ligada da Figura 10.3(a) com três arranjos. A *chave* do arranjo contém os valores das chaves presentes atualmente no conjunto dinâmico, e os ponteiros são armazenados nos arranjos *próximo* e *anterior*. Para um dado índice de arranjo  $x$ , *chave*[ $x$ ], *próximo*[ $x$ ] e *anterior*[ $x$ ] representam um objeto na lista ligada. Sob essa interpretação, um ponteiro  $x$  é simplesmente um índice comum para os arranjos *chave*, *próximo* e *anterior*.

Na Figura 10.3(a), o objeto com chave 4 segue o objeto com chave 16 na lista ligada. Na Figura 10.5, a chave 4 aparece em *chave*[2] e a chave 16 aparece em *chave*[5]; assim, temos *próximo*[5] = 2 e *anterior*[2] = 5. Embora a constante NIL apareça no campo *próximo* do fim e no campo *anterior* do início, em geral usamos um inteiro (como 0 ou -1) que não tem possibilidade de representar um índice real para os arranjos. Uma variável  $L$  contém o índice do início da lista.

Em nosso pseudocódigo, temos usado colchetes para denotar tanto a indexação de um arranjo quanto a seleção de um campo (atributo) de um objeto. De qualquer modo, os significados de *chave*[ $x$ ], *próximo*[ $x$ ] e *anterior*[ $x$ ] são consistentes com a prática de implementação.

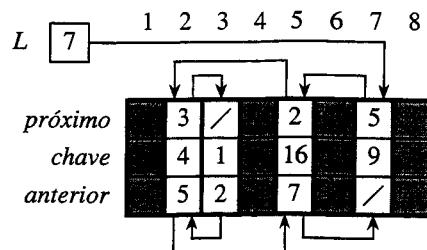


FIGURA 10.5 A lista ligada da Figura 10.3(a) representada pelos arranjos *chave*, *próximo* e *anterior*. Cada fatia vertical dos arranjos representa um objeto único. Os ponteiros armazenados correspondem aos índices do arranjo mostrados na parte superior; as setas mostram como interpretá-los. As posições de objetos levemente sombreadas contêm elementos de listas. A variável  $L$  mantém o índice do início

## Uma representação de objetos com um único arranjo

As palavras na memória de um computador normalmente são endereçadas por inteiros desde 0 até  $M - 1$ , onde  $M$  é um inteiro adequadamente grande. Em muitas linguagens de programação, um objeto ocupa um conjunto contíguo de posições na memória do computador. Um ponteiro é simplesmente o endereço da primeira posição de memória do objeto, e outras posições de memória dentro do objeto podem ser indexadas pela inclusão de um deslocamento para o ponteiro.

Podemos utilizar a mesma estratégia para implementar objetos em ambientes de programação que não fornecem tipos de dados ponteiro explícito. Por exemplo, a Figura 10.6 mostra como um único arranjo  $A$  pode ser usado para armazenar a lista ligada das Figuras 10.3(a) e 10.5. Um objeto ocupa um subarranjo contíguo  $A[j .. k]$ . Cada campo do objeto corresponde a um deslocamento no intervalo de 0 a  $k - j$ , e um ponteiro para o objeto é o índice  $j$ . Na Figura 10.6, os deslocamentos correspondentes a *chave*, *próximo* e *anterior* são 0, 1 e 2, respectivamente. Para ler o valor de  $anterior[i]$ , dado um ponteiro  $i$ , adicionamos o valor  $i$  do ponteiro ao deslocamento 2, lendo assim  $A[i + 2]$ .

A representação de um único arranjo é flexível pelo fato de permitir que objetos de diferentes comprimentos sejam armazenados no mesmo arranjo. O problema de administrar tal coleção heterogênea de objetos é mais difícil que o problema de administrar uma coleção homogênea, onde todos os objetos têm os mesmos campos. Tendo em vista que a maioria das estruturas de dados que consideraremos são compostas por elementos homogêneos, será suficiente para nossos propósitos empregar a representação de objetos de vários arranjos.

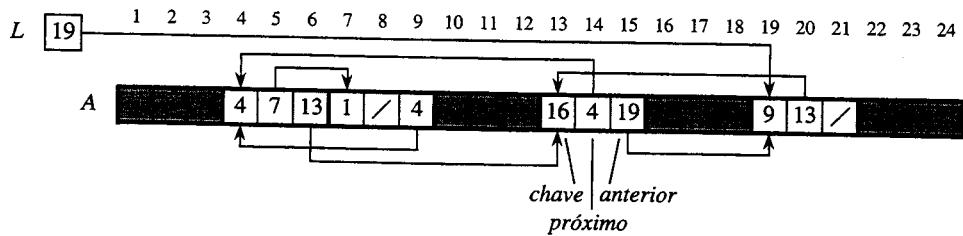


FIGURA 10.6 A lista ligada das Figuras 10.3(a) e 10.5, representada em um único arranjo  $A$ . Cada elemento da lista é um objeto que ocupa um subarranjo contíguo de comprimento 3 dentro do arranjo. Os três campos *chave*, *próximo* e *anterior* correspondem aos deslocamentos 0, 1 e 2, respectivamente. Um ponteiro para um objeto é um índice do primeiro elemento do objeto. Os objetos contendo elementos da lista estão levemente sombreados, e as setas mostram a ordenação da lista

## Alocação e liberação de objetos

Para inserir uma chave em um conjunto dinâmico representado por uma lista duplamente ligada, devemos alocar um ponteiro para um objeto atualmente não utilizado na representação da lista ligada. Assim, é útil gerenciar o espaço de armazenamento de objetos não utilizados no momento na representação da lista ligada, de tal modo que um objeto possa ser alocado. Em alguns sistemas, um *coletor de lixo* é responsável por determinar quais objetos não serão utilizados. Porém, muitas aplicações são simples o bastante para poderem assumir a responsabilidade pela devolução de um objeto não utilizado a um gerenciador de espaço de armazenamento. Agora, exploraremos o problema de alocar e liberar (ou desalocar) objetos homogêneos utilizando o exemplo de uma lista duplamente ligada representada por vários arranjos.

Suponha que os arranjos na representação de vários arranjos têm comprimento  $m$  e que em algum momento o conjunto dinâmico contém  $n \leq m$  elementos. Então,  $n$  objetos representam elementos que se encontram atualmente no conjunto dinâmico, e os  $m - n$  objetos restantes são *livres*; os objetos livres podem ser usados para representar elementos inseridos no conjunto dinâmico no futuro.

Mantemos os objetos livres em uma lista simplesmente ligada, que chamamos *lista livre*. A lista livre usa apenas o arranjo *próximo*, que armazena os ponteiros *próximo* dentro da lista. O início da lista livre está contido na variável global *livre*. Quando o conjunto dinâmico representado pela lista ligada *L* é não vazio, a lista livre pode ser entrelaçada com a lista *L*, como mostra a Figura 10.7. Observe que cada objeto na representação está na lista *L* ou na lista livre, mas não em ambas.

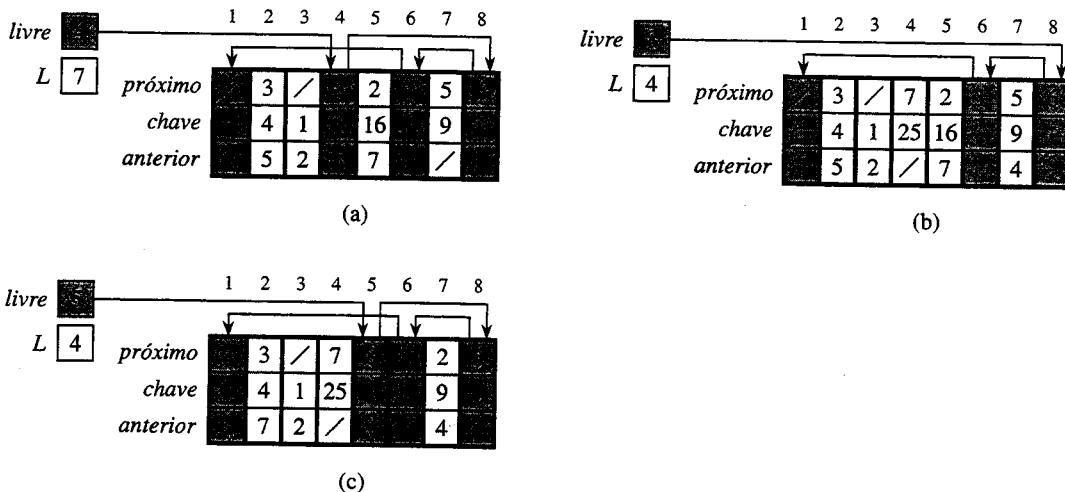


FIGURA 10.7 O efeito dos procedimentos ALLOCATE-OBJECT e FREE-OBJECT. (a) A lista da Figura 10.5 (levemente sombreada) e uma lista livre (fortemente sombreada). As setas mostram a estrutura da lista livre. (b) O resultado da chamada *ALLOCATE-OBJECT()* (que retorna o índice 4), definindo *chave[4]* como 25, e da chamada a *LIST-INSERT(L, 4)*. O novo início da lista livre é o objeto 8, que era *próximo[4]* na lista livre. (c) Depois de executar *LIST-DELETE(L, 5)*, chamamos *FREE-OBJECT(5)*. O objeto 5 se torna o novo início da lista livre, seguido pelo objeto 8 na lista livre

A lista livre é uma pilha: o próximo objeto alocado é o último objeto liberado. Podemos usar uma implementação de lista das operações de pilhas PUSH e POP, a fim de implementar os procedimentos para alocar e liberar objetos, respectivamente. Supomos que a variável global *livre* usada nos procedimentos a seguir aponta para o primeiro elemento da lista livre.

```
ALLOCATE-OBJECT()
1 if livre = NIL
2 then error "out of space"
3 else x ← livre
4     livre ← próximo[x]
5 return x
```

```
FREE-OBJECT(x)
1 próximo[x] ← livre
2 livre ← x
```

A lista livre contém inicialmente todos os *n* objetos não alocados. Quando a lista livre é esgotada, o procedimento ALLOCATE-OBJECT assinala um erro. É comum o uso de uma única lista livre para servir várias listas ligadas. A Figura 10.8 mostra duas listas ligadas e uma lista livre entrelaçada através dos arranjos *chave*, *próximo* e *anterior*.

Os dois procedimentos são executados no tempo  $O(1)$ , o que os torna bastante práticos. Eles podem ser modificados com o objetivo de funcionarem para qualquer coleção homogênea de objetos, permitindo que qualquer um dos campos no objeto atue como um campo *próximo* na lista livre.

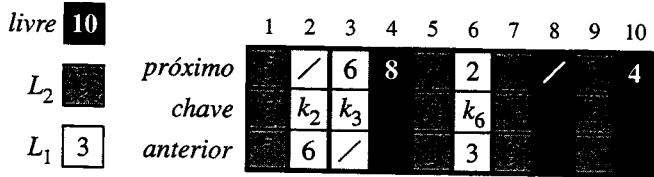


FIGURA 10.8 Duas listas ligadas,  $L_1$  (levemente sombreada) e  $L_2$  (fortemente sombreada), e uma lista livre (escurecida) entrelaçada

## Exercícios

### 10.3-1

Trace um quadro da seqüência  $\langle 13, 4, 8, 19, 5, 11 \rangle$  armazenada como uma lista duplamente ligada, utilizando a representação de vários arranjos. Faça o mesmo para a representação de um único arranjo.

### 10.3-2

Escreva os procedimentos ALLOCATE-OBJECT e FREE-OBJECT para uma coleção homogênea de objetos implementada pela representação de um único arranjo.

### 10.3-3

Por que não precisamos definir ou redefinir os campos *anterior* de objetos na implementação dos procedimentos ALLOCATE-OBJECT e FREE-OBJECT?

### 10.3-4

Freqüentemente é desejável manter todos os elementos de uma lista duplamente ligada compactos no espaço de armazenamento, usando-se, por exemplo, as primeiras posições do índice  $m$  na representação de vários arranjos. (Esse é o caso em um ambiente de computação de memória virtual paginada.) Explique de que modo os procedimentos ALLOCATE-OBJECT e FREE-OBJECT podem ser implementados de forma que a representação seja compacta. Suponha que não existam ponteiros para elementos da lista ligada fora da própria lista. (Sugestão: Use a implementação de arranjo de uma pilha.)

### 10.3-5

Seja  $L$  uma lista duplamente ligada de comprimento  $m$  armazenada nos arranjos *chave*, *anterior* e *próximo* de comprimento  $n$ . Suponha que esses arranjos sejam gerenciados pelos procedimentos ALLOCATE-OBJECT e FREE-OBJECT, que mantêm uma lista livre duplamente ligada  $F$ . Suponha ainda que, dos  $n$  itens, exatamente  $m$  itens estejam na lista  $L$  e  $n - m$  estejam na lista livre. Escreva um procedimento COMPACTIFY-LIST( $L, F$ ) que, dada a lista  $L$  e a lista livre  $F$ , desloque os itens em  $L$  de forma que eles ocupem as posições de arranjos  $1, 2, \dots, m$  e ajuste a lista livre  $F$  para que ela permaneça correta, ocupando as posições de arranjos  $m + 1, m + 2, \dots, n$ . O tempo de execução do seu procedimento deve ser  $\Theta(m)$ , e ele só deve utilizar uma quantidade constante de espaço extra. Forneça um argumento cuidadoso para justificar a correção do seu procedimento.

## 10.4 Representação de árvores enraizadas

Os métodos para representação de listas dados na seção anterior se estendem a qualquer estrutura de dados homogêneos. Nesta seção, examinaremos especificamente o problema da representação de árvores enraizadas por estruturas de dados ligadas. Primeiro, veremos as árvores binárias e depois apresentaremos um método para árvores enraizadas nas quais os nós podem ter um número arbitrário de filhos.

Representamos cada nó de uma árvore por um objeto. Como no caso das listas ligadas, vamos supor que cada nó contém um campo *chave*. Os campos restantes de interesse são ponteiros para outros nós, e eles variam de acordo com o tipo de árvore.

## Árvores binárias

Como mostra a Figura 10.9, usamos os campos *p*, *esquerdo* e *direito* com o objetivo de armazenar ponteiros para o pai, o filho da esquerda e o filho da direita de cada nó em uma árvore binária *T*. Se  $p[x] = \text{NIL}$ , então  $x$  é a raiz. Se o nó  $x$  não tem nenhum filho da esquerda, então  $\text{esquerdo}[x] = \text{NIL}$ , e temos uma situação semelhante para o filho da direita. A raiz da árvore *T* inteira é apontada pelo atributo *raiz[T]*. Se *raiz[T]* = NIL, então a árvore é vazia.

## Árvores enraizadas com ramificações ilimitadas

O esquema para representar uma árvore binária pode ser estendido a qualquer classe de árvores na qual o número de filhos de cada nó seja no máximo alguma constante *k*: substituímos os campos *esquerdo* e *direito* por *filho<sub>1</sub>*, *filho<sub>2</sub>*, ..., *filho<sub>k</sub>*. Esse esquema não funciona mais quando o número de filhos de um nó é ilimitado, pois não sabemos quantos campos (arranjos na representação de vários arranjos) devemos alocar antecipadamente. Além disso, mesmo que o número de filhos *k* seja limitado por uma constante grande, mas a maioria dos nós tenha um número pequeno de filhos, poderemos desperdiçar uma grande quantidade de memória.

Felizmente, existe um esquema inteligente para usar árvores binárias com a finalidade de representar árvores com números arbitrários de filhos. Ele tem a vantagem de utilizar apenas o espaço  $O(n)$  para qualquer árvore enraizada de  $n$  nós. A *representação de filho da esquerda, irmão da direita* é mostrada na Figura 10.10. Como antes, cada nó contém um ponteiro superior *p*, e a *raiz[T]* aponta para a raiz da árvore *T*. Contudo, em vez de ter um ponteiro para cada um de seus filhos, cada nó  $x$  tem apenas dois ponteiros:

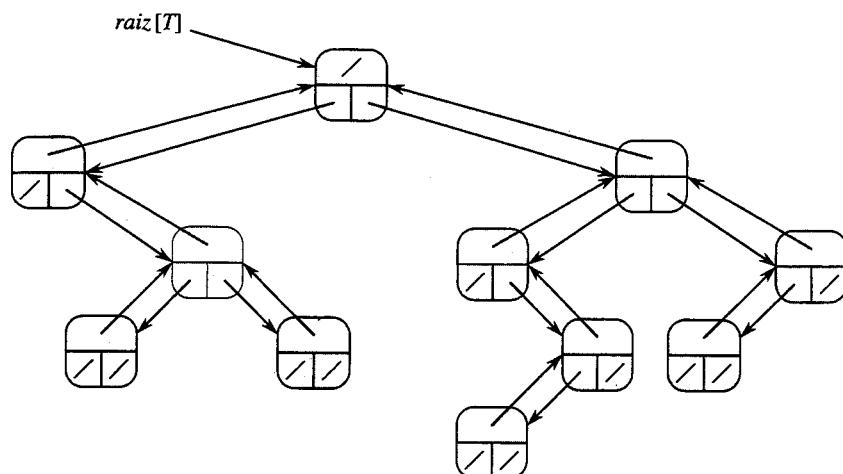


FIGURA 10.9 A representação de uma árvore binária *T*. Cada nó  $x$  tem os campos *p[x]* (superior), *esquerdo[x]* (inferior esquerdo) e *direito[x]* (inferior direito). Os campos *chave* não são mostrados

1. *Filho da esquerda[x]* aponta para o filho da extremidade esquerda do nó  $x$ , e
2. *Irmão da direita[x]* aponta para o irmão de  $x$  situado imediatamente à direita.

Se o nó  $x$  não tem nenhum filho, então *filho da esquerda[x] = NIL* e, se o nó  $x$  é o filho da extremidade direita de seu pai, então *irmão da direita[x] = NIL*.

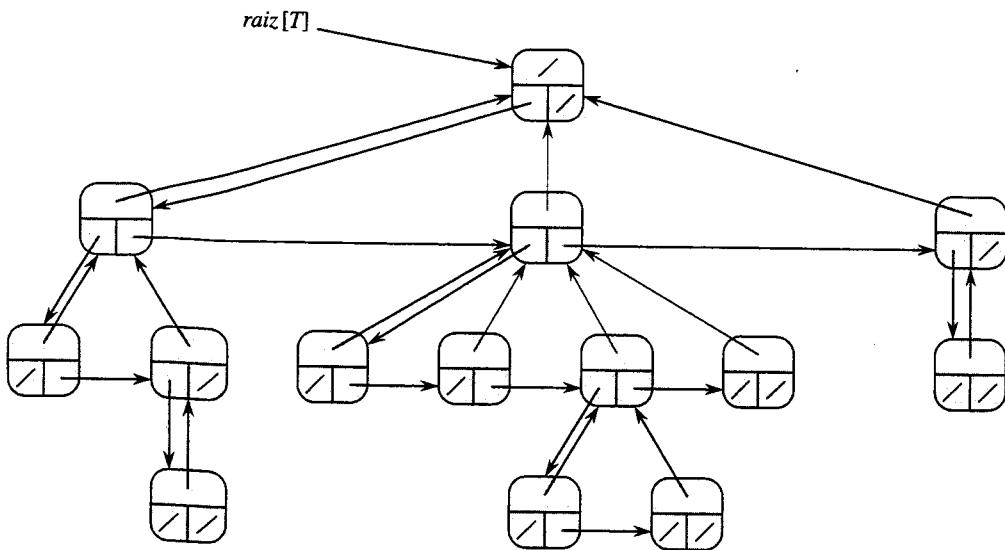


FIGURA 10.10 A representação de filho da esquerda, irmão da direita de uma árvore  $T$ . Cada nó  $x$  tem campos  $p[x]$  (superior),  $filho\ da\ esquerda[x]$  (inferior esquerdo) e  $irmão\ da\ direita[x]$  (inferior direito). As chaves não são mostradas.

## Outras representações de árvores

Algumas vezes, representamos árvores enraizadas de outras maneiras. Por exemplo, no Capítulo 6, representamos um heap (monte), que se baseia em uma árvore binária completa, por um único arranjo e mais um índice. As árvores que aparecem no Capítulo 21 são atravessadas somente em direção à raiz; assim, apenas os ponteiros superiores estão presentes: não há ponteiros para filhos. Muitos outros esquemas são possíveis. O melhor esquema dependerá da aplicação.

## Exercícios

### 10.4-1

Desenhe a árvore binária enraizada no índice 6 que é representada pelos campos a seguir.

índice	chave	esquerdo	direito
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

### 10.4-2

Escreva um procedimento recursivo de tempo  $O(n)$  que, dada uma árvore binária de  $n$  nós, imprima a chave de cada nó na árvore.

### 10.4-3

Escreva um procedimento não recursivo de tempo  $O(n)$  que, dada uma árvore binária de  $n$  nós, imprima a chave de cada nó na árvore. Use uma pilha como estrutura de dados auxiliar.

#### 10.4-4

Escreva um procedimento de tempo  $O(n)$  que imprima todas as chaves de uma árvore enraizada arbitrária com  $n$  nós, onde a árvore é armazenada usando a representação de filho da esquerda, irmão da direita.

#### 10.4-5 \*

Escreva um procedimento não recursivo de tempo  $O(n)$  que, dada uma árvore binária de  $n$  nós, imprima a chave de cada nó. Não utilize mais que um espaço extra constante fora da própria árvore e não modifique a árvore, mesmo temporariamente, durante o procedimento.

#### 10.4-6 \*

A representação de filho da esquerda, irmão da direita de uma árvore enraizada arbitrária utiliza três ponteiros em cada nó: *filho da esquerda*, *irmão da direita* e *pai*. A partir de qualquer nó, seu pai pode ser alcançado e identificado em tempo constante, e todos os seus filhos podem ser alcançados e identificados em tempo linear no número de filhos. Mostre como usar apenas dois ponteiros e um valor booleano em cada nó para que o pai de um nó ou todos os seus filhos possam ser alcançados e identificados em tempo linear no número de filhos.

## Problemas

### 10-1 *Comparações entre listas*

Para cada um dos quatro tipos de listas na tabela a seguir, qual é o tempo de execução assintótico no pior caso para cada operação sobre conjuntos dinâmicos listada?

	não ordenada, simplesmente ligada	ordenada, simplesmente ligada	não ordenada, duplamente ligada	ordenada, duplamente ligada
SEARCH( $L, k$ )				
INSERT( $L, x$ )				
DELETE( $L, x$ )				
SUCCESSOR( $L, x$ )				
PREDECESSOR( $L, x$ )				
MINIMUM( $L$ )				
MAXIMUM( $L$ )				

### 10-2 *Heaps intercaláveis com o uso de listas ligadas*

Um heap intercalável admite as seguintes operações: MAKE-HEAP (que cria um heap intercalável vazio), INSERT, MINIMUM, EXTRACT-MIN e UNION.<sup>1</sup> Mostre como implementar heaps intercaláveis com o uso de listas ligadas em cada um dos casos a seguir. Procure tornar cada operação tão eficiente quanto possível. Analise o tempo de execução de cada operação em termos do tamanho do(s) conjunto(s) dinâmico(s) sobre o(s) qual(is) é realizada a operação.

<sup>1</sup> Tendo em vista que definimos um heap intercalável para dar suporte a MINIMUM e EXTRACT-MIN, também podemos nos referir a ele como um heap intercalável mínimo. Como outra alternativa, se o heap admitisse MAXIMUM e EXTRACT-MAX, ele seria um heap intercalável máximo.

- a. As listas são ordenadas.
- b. As listas são não ordenadas.
- c. As listas são não ordenadas, e os conjuntos dinâmicos a serem intercalados são disjuntos.

### 10-3 Pesquisa em uma lista compacta ordenada

O Exercício 10.3-4 perguntou como poderíamos manter compacta uma lista de  $n$  elementos nas primeiras  $n$  posições de um arranjo. Vamos supor que todas as chaves sejam distintas e que a lista compacta também seja ordenada; ou seja  $chave[i] < chave[próximo[i]]$  para todo  $i = 1, 2, \dots, n$  tal que  $próximo[i] \neq \text{NIL}$ . Sob essas hipóteses, você mostrará que o algoritmo aleatório a seguir pode ser usado para pesquisar a lista no tempo esperado  $O(\sqrt{n})$ .

**COMPACT-LIST-SEARCH( $L, n, k$ )**

```

1  $i \leftarrow \text{início}[L]$ 
2 while  $i \neq \text{NIL}$  e  $chave[i] < k$ 
3   do  $j \leftarrow \text{RANDOM}(1, n)$ 
4     if  $chave[i] < chave[j]$  e  $chave[j] \leq k$ 
5       then  $i \leftarrow j$ 
6       if  $chave[i] = k$ 
7         then return  $i$ 
8      $i \leftarrow \text{próximo}[i]$ 
9 if  $i = \text{NIL}$  or  $chave[i] > k$ 
10  then return NIL
11  else return  $i$ 
```

Se ignorarmos as linhas 3 a 7 do procedimento, teremos um algoritmo comum para pesquisa em uma lista ligada ordenada, na qual o índice  $i$  aponta por sua vez para cada posição da lista. A pesquisa termina uma vez que o índice  $i$  “cai” no final da lista ou uma vez que  $chave[i] \geq k$ . Nesse último caso, se  $chave[i] = k$ , torna-se claro que encontramos uma chave com o valor  $k$ . Se, porém,  $chave[i] > k$ , isso significa que nunca encontraremos uma chave com o valor  $k$ , e então encerrar a pesquisa é a ação correta.

As linhas 3 a 7 tentam saltar à frente, até uma posição  $j$  escolhida aleatoriamente. Esse salto será vantajoso se  $chave[j]$  for maior que  $chave[i]$  e não maior que  $k$ ; em tal caso,  $j$  marcará uma posição na lista pela qual  $i$  teria de passar durante uma pesquisa comum na lista. Pelo fato de a lista ser compacta, sabemos que qualquer escolha de  $j$  entre 1 e  $n$  efetua a indexação de algum objeto na lista, em vez de uma abertura na lista livre.

Em lugar de analisar o desempenho de COMPACT-LIST-SEARCH diretamente, analisaremos um algoritmo relacionado, COMPACT-LIST-SEARCH', que executa dois loops separados. Esse algoritmo utiliza um parâmetro adicional  $t$  que determina um limite superior sobre o número de iterações do primeiro loop.

**COMPACT-LIST-SEARCH'( $L, n, k, t$ )**

```

1  $i \leftarrow \text{início}[L]$ 
2 for  $q \leftarrow 1$  to  $t$ 
3   do  $j \leftarrow \text{RANDOM}(1, n)$ 
4     if  $chave[i] < chave[j]$  e  $chave[j] \leq k$ 
5       then  $i \leftarrow j$ 
6       if  $chave[i] = k$ 
7         then return  $i$ 
8 while  $i \neq \text{NIL}$  e  $chave[i] < k$ 
9   do  $i \leftarrow \text{próximo}[i]$ 
10 if  $i = \text{NIL}$  or  $chave[i] > k$ 
11   then return NIL
12   else return  $i$ 
```

Para comparar a execução dos algoritmos COMPACT-LIST-SEARCH( $L, k$ ) e COMPACT-LIST-SEARCH'( $L, k, t$ ), vamos supor que a sequência de inteiros retornados pelas chamadas de RANDOM(1,  $n$ ) seja a mesma para ambos os algoritmos.

- a. Suponha que COMPACT-LIST-SEARCH( $L, k$ ) execute  $t$  iterações do loop **while** das linhas 2 a 8. Demonstre que COMPACT-LIST-SEARCH'( $L, k, t$ ) retorna a mesma resposta e que o número total de iterações dos loops **for** e **while** dentro de COMPACT-LIST-SEARCH' é pelo menos  $t$ .

Na chamada COMPACT-LIST-SEARCH'( $L, k, t$ ), seja  $X_t$  a variável aleatória que descreve a distância na lista ligada (isto é, através da cadeia de ponteiros *próximo*) desde a posição  $i$  até a chave desejada  $k$ , após terem ocorrido  $t$  iterações do loop **for** das linhas 2 a 7.

- b. Mostre que o tempo de execução esperado de COMPACT-LIST-SEARCH'( $L, k, t$ ) é  $O(t + E[X_t])$ .
- c. Mostre que  $E[X_t] \leq \sum_{r=1}^n (1 - r/n)^t$ . (*Sugestão:* Use a equação (C.24).)
- d. Mostre que  $\sum_{r=0}^{n-1} r^t \leq n^{t+1} / (t + 1)$ .
- e. Prove que  $E[X_t] \leq n/(t + 1)$ .
- f. Mostre que COMPACT-LIST-SEARCH'( $L, k, t$ ) é executado no tempo esperado  $O(t + n/t)$ .
- g. Conclua que COMPACT-LIST-SEARCH é executado no tempo esperado  $O(\sqrt{n})$ .
- b. Por que supomos que todas as chaves são distintas em COMPACT-LIST-SEARCH? Demonstre que não é obrigatório que saltos aleatórios ajudem assintoticamente quando a lista contém valores de chave repetidos.

## Notas do capítulo

Aho, Hopcroft e Ullman [6] e Knuth [182] são excelentes referências sobre estruturas de dados elementares. Muitos outros textos focalizam tanto estruturas de dados básicos quanto sua implementação em uma linguagem de programação particular. Os exemplos desses tipos de livros-texto incluem Goodrich e Tamassia [128], Main [209], Shaffer [273] e Weiss [310, 312, 313]. Gonnet [126] fornece dados experimentais a respeito do desempenho de muitas operações sobre estruturas de dados.

A origem de pilhas e filas como estruturas de dados em ciência da computação não é clara, pois já existiam noções correspondentes em matemática e em práticas comerciais baseadas em papéis antes da introdução dos computadores digitais. Knuth [182] cita A. M. Turing sobre o desenvolvimento de pilhas para o encadeamento de sub-rotinas em 1947.

Estruturas de dados baseadas em ponteiros também parecem ser uma criação popular. De acordo com Knuth, os ponteiros foram usados aparentemente nos primeiros computadores com memórias de tambor. A linguagem A-1, desenvolvida por G. M. Hopper em 1951, representava fórmulas algébricas como árvores binárias. Knuth credita à linguagem IPL-II, desenvolvida em 1956 por A. Newell, J. C. Shaw e H. A. Simon, o reconhecimento da importância e a promoção do uso de ponteiros. Sua linguagem IPL-III, desenvolvida em 1957, incluía operações explícitas sobre pilhas.

# *Capítulo 11*

## *Tabelas hash*

Muitas aplicações exigem um conjunto dinâmico que admite apenas as operações de dicionário **INSERT**, **SEARCH** e **DELETE**. Por exemplo, um compilador para uma linguagem de computador mantém uma tabela de símbolos, na qual as chaves de elementos são cadeias de caracteres arbitrários que correspondem a identificadores na linguagem. Uma tabela hash é uma estrutura de dados eficiente para implementar dicionários. Embora a busca por um elemento em uma tabela hash possa demorar tanto quanto procurar por um elemento em uma lista ligada – o tempo  $\Theta(n)$  no pior caso – na prática, o hash funciona extremamente bem. Sob hipóteses razoáveis, o tempo esperado para a busca por um elemento em uma tabela hash é  $O(1)$ .

Uma tabela hash é uma generalização da noção mais simples de um arranjo comum. O endereçamento direto em um arranjo comum faz uso eficiente de nossa habilidade para examinar uma posição arbitrária em um arranjo no tempo  $O(1)$ . A Seção 11.1 discute com mais detalhes o endereçamento direto. O endereçamento direto é aplicável quando temos condições de alocar um arranjo que tem uma única posição para cada chave possível.

Quando o número de chaves realmente armazenadas é pequeno em relação ao número total de chaves possíveis, as tabelas hash se tornam uma alternativa eficiente para se endereçar diretamente um arranjo, pois em geral uma tabela hash utiliza um arranjo de tamanho proporcional ao número de chaves realmente armazenadas. Em vez de usar a chave diretamente como um índice de arranjo, o índice de arranjo é *computado* a partir da chave. A Seção 11.2 apresenta as principais idéias, concentrando-se no “encadeamento” como um meio para tratar “colisões” no qual mais de uma chave é mapeada para o mesmo índice de arranjo. A Seção 11.3 descreve como os índices de arranjos podem ser computados a partir de chaves com o uso de funções hash. São apresentadas e analisadas diversas variações sobre o tema básico. A Seção 11.4 examina o “endereçamento aberto”, outro modo de lidar com colisões. A conclusão é que o hash é uma técnica extremamente eficaz e prática: as operações básicas de dicionário exigem apenas o tempo  $O(1)$  em média. A Seção 11.5 explica como o “hash perfeito” pode dar suporte a pesquisas no tempo de *pior caso*  $O(1)$ , quando o conjunto de chaves que está sendo armazenado é estático (isto é, quando o conjunto de chaves nunca se altera uma vez armazenado).

### **11.1 Tabelas de endereço direto**

O endereçamento direto é uma técnica simples que funciona bem quando o universo  $U$  de chaves é razoavelmente pequeno. Suponha que uma aplicação necessite de um conjunto dinâmico no qual cada elemento tenha uma chave definida a partir do universo  $U = \{0, 1, \dots, m - 1\}$ , onde  $m$  não é muito grande. Iremos supor que não há dois elementos com a mesma chave.

Para representar o conjunto dinâmico, usamos um arranjo ou uma **tabela de endereço direto**  $T[0 \dots m - 1]$ , na qual cada abertura (slot), ou **posição**, corresponde a uma chave no universo  $U$ . A Figura 11.1 ilustra a abordagem; a posição  $k$  aponta para um elemento no conjunto com chave  $k$ . Se o conjunto não contém nenhum elemento com chave  $k$ , então  $T[k] = \text{NIL}$ .

A implementação das operações de dicionário é trivial.

**DIRECT-ADDRESS-SEARCH( $T, k$ )**

**return**  $T[k]$

**DIRECT-ADDRESS-INSERT( $T, x$ )**

$T[\text{chave}[x]] \leftarrow x$

**DIRECT-ADDRESS-DELETE( $T, x$ )**

$T[\text{chave}[x]] \leftarrow \text{NIL}$

Cada uma dessas operações é rápida: é necessário apenas o tempo  $O(1)$ .

Para algumas aplicações, os elementos no conjunto dinâmico podem ser armazenados na própria tabela de endereço direto. Ou seja, em vez de armazenar a chave e os dados satélite de um elemento em um objeto externo à tabela de endereço direto, com um ponteiro de uma posição na tabela até o objeto, podemos armazenar o objeto na própria posição, e portanto economizar espaço. Além disso, freqüentemente é desnecessário armazenar o campo de chave do objeto pois, se temos o índice de um objeto na tabela, temos sua chave. Entretanto, se as chaves não forem armazenadas, devemos ter algum modo de saber se a posição está vazia.

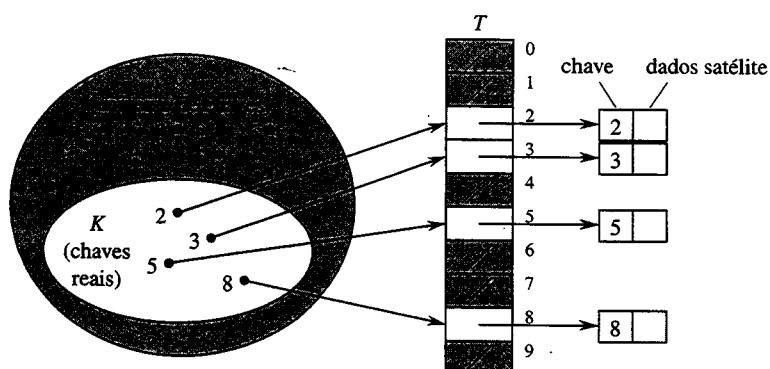


FIGURA 11.1 Implementação de um conjunto dinâmico por uma tabela de endereço direto  $T$ . Cada chave no universo  $U = \{0, 1, \dots, 9\}$  corresponde a um índice na tabela. O conjunto  $K = \{2, 3, 5, 8\}$  de chaves reais determina as posições na tabela que contêm ponteiros para elementos. As outras posições, fortemente sombreadas, contêm NIL.

## Exercícios

### 11.1-1

Considere um conjunto dinâmico  $S$  que é representado por uma tabela de endereço direto  $T$  de comprimento  $m$ . Descreva um procedimento que encontre o elemento máximo de  $S$ . Qual é o desempenho de seu procedimento no pior caso?

### 11.1-2

Um **vetor de bits** é simplesmente um arranjo de bits (0s e 1s). Um vetor de bits de comprimento  $m$  ocupa muito menos espaço que um arranjo de  $m$  ponteiros. Descreva como usar um vetor de bits para representar um conjunto dinâmico de elementos distintos sem dados satélite. As operações de dicionário devem ser executadas em tempo  $O(1)$ .

### 11.1-3

Sugira um modo de implementar uma tabela de endereço direto na qual as chaves de elementos armazenados não precisem ser distintas, e os elementos possam ter dados satélite. Todas as três operações de dicionário (INSERT, DELETE e SEARCH) devem ser executadas no tempo  $O(1)$ . (Não se esqueça de que DELETE toma como argumento um ponteiro para um objeto a ser eliminado, e não uma chave.)

### 11.1-4 \*

Desejamos implementar um dicionário usando endereçamento direto sobre um arranjo enorme. No início, as entradas do arranjo podem conter lixo e a inicialização do arranjo inteiro é impraticável devido a seu tamanho. Descreva um esquema para implementar um dicionário de endereço direto sobre um arranjo enorme. Cada objeto armazenado deve utilizar o espaço  $O(1)$ ; as operações SEARCH, INSERT e DELETE devem demorar o tempo  $O(1)$  cada uma; e a inicialização da estrutura de dados deve demorar o tempo  $O(1)$ . (Sugestão: Use uma pilha adicional, cujo tamanho é o número de chaves realmente armazenadas no dicionário, a fim de ajudar a determinar se uma dada entrada no arranjo enorme é válida ou não.)

## 11.2 Tabelas hash

A dificuldade com o endereçamento direto é óbvia: se o universo  $U$  é grande, o armazenamento de uma tabela  $T$  de tamanho  $|U|$  pode ser impraticável, ou mesmo impossível, em virtude da memória disponível em um computador típico. Além disso, o conjunto  $K$  de chaves *realmente armazenadas* pode ser tão pequeno em relação a  $U$  que a maior parte do espaço alocado para  $T$  seria desperdiçada.

Quando o conjunto  $K$  de chaves armazenadas em um dicionário é muito menor que o universo  $U$  de todas as chaves possíveis, uma tabela hash exige muito menos espaço de armazenamento que uma tabela de endereço direto. Especificamente, os requisitos de armazenamento podem ser reduzidos a  $\Theta(|K|)$ , embora a pesquisa de um elemento na tabela hash ainda exija apenas o tempo  $O(1)$ . (A única desvantagem é que esse limite corresponde ao *tempo médio*, enquanto no caso do endereçamento direto ele se mantém válido para o *tempo do pior caso*.) Com o endereçamento direto, um elemento com a chave  $k$  é armazenado na posição  $k$ . No caso do hash, esse elemento é armazenado na posição  $b(k)$ ; ou seja, uma **função hash**  $b$  é utilizada para calcular a posição a partir da chave  $k$ . Aqui,  $b$  mapeia o universo  $U$  de chaves nas posições de uma **tabela hash**  $T[0 .. m - 1]$ :

$$b : U \rightarrow \{0, 1, \dots, m - 1\}.$$

Dizemos que um elemento com a chave  $k$  **efetua o hash** para a posição  $b(k)$ ; dizemos também que  $b(k)$  é o **valor hash** da chave  $k$ . A Figura 11.2 ilustra a idéia básica. A finalidade da função hash é reduzir o intervalo de índices de arranjos que precisam ser tratados. Em vez de  $|U|$  valores, precisamos manipular apenas  $m$  valores. Os requisitos de armazenamento são reduzidos de modo correspondente.

O detalhe fundamental dessa boa idéia é que duas chaves podem ter o hash na mesma posição. Chamamos essa situação de **colisão**. Felizmente, existem técnicas eficientes para resolver o conflito criado por colisões.

É claro que a solução ideal seria evitar por completo as colisões. Poderíamos tentar alcançar essa meta escolhendo uma função hash adequada  $b$ . Uma idéia é fazer  $b$  parecer “aleatória”, evitando assim as colisões, ou pelo menos minimizando seu número. A expressão “efetuar o hash”, que evoca imagens de misturas e recortes aleatórios, capta o espírito dessa abordagem. (É claro que uma função hash  $b$  deve ser determinística, no sentido de que uma dada entrada  $k$  sempre deve produzir a mesma saída  $b(k)$ .) Porém, tendo em vista que  $|U| > m$ , devem existir duas cha-

ves que tenham o mesmo valor hash; assim, é impossível evitar totalmente as colisões. Desse modo, embora uma função hash bem projetada e de aparência “aleatória” possa minimizar o número de colisões, ainda precisaremos de um método para resolver as colisões que ocorrerem.

O restante desta seção apresenta a técnica mais simples para resolução de colisões, chamada encadeamento. A Seção 11.4 introduz um método alternativo para solucionar colisões, denominado endereçamento aberto.

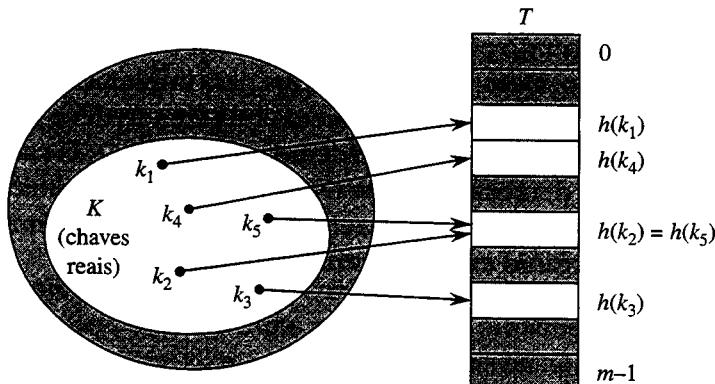


FIGURA 11.2 O uso de uma função hash  $h$  para mapear chaves como posições de uma tabela hash. As chaves  $k_2$  e  $k_5$  são mapeadas para a mesma posição, e assim elas colidem

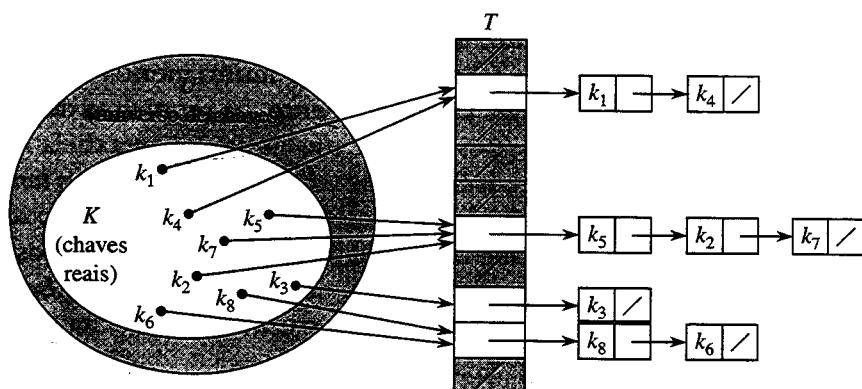


FIGURA 11.3 Resolução de colisões por encadeamento. Cada posição  $T[j]$  da tabela hash contém uma lista ligada de todas as chaves cujo valor hash é  $j$ . Por exemplo,  $b(k_1) = b(k_4)$  e  $b(k_5) = b(k_2) = b(k_7)$

### Resolução de colisões por encadeamento

No **encadeamento**, colocamos todos os elementos que efetuam hash para a mesma posição em uma lista ligada, como mostra a Figura 11.3. A posição  $j$  contém um ponteiro para o início da lista de todos os elementos armazenados que efetuam hash para  $j$ ; se não houver nenhum desses elementos, a posição  $j$  conterá NIL.

As operações de dicionário sobre uma tabela hash  $T$  são fáceis de implementar quando as colisões são resolvidas por encadeamento.

**CHAINED-HASH-INSERT( $T, x$ )**

insere  $x$  no início da lista  $T[b(chave[x])]$

**CHAINED-HASH-SEARCH( $T, k$ )**

procura por um elemento com a chave  $k$  na lista  $T[b(k)]$

**CHAINED-HASH-DELETE( $T, x$ )**  
elimina  $x$  da lista  $T[b(chave[x])]$

O tempo de execução no pior caso para a inserção é  $O(1)$ . No caso da pesquisa, o tempo de execução no pior caso é proporcional ao comprimento da lista; analisaremos esse assunto com mais detalhes em seguida. A eliminação de um elemento  $x$  pode ser alcançada no tempo  $O(1)$  se as listas forem duplamente ligadas. (Se as listas são simplesmente ligadas, primeiro devemos encontrar  $x$  na lista  $T[b(chave[x])]$ , de forma que o *próximo* vínculo do predecessor de  $x$  possa ser definido de modo apropriado para unir  $x$ ; nesse caso, a eliminação e a pesquisa terão essencialmente o mesmo tempo de execução.)

## Análise do hash com encadeamento

Qual é a qualidade da execução do hash com encadeamento? Em particular, quanto tempo ele leva para procurar por um elemento com uma determinada chave?

Dada uma tabela hash  $T$  com  $m$  posições que armazena  $n$  elementos, definimos o **fator de carga**  $\alpha$  para  $T$  como  $n/m$ , isto é, o número médio de elementos armazenados em uma cadeia. Nossa análise será em termos de  $\alpha$ , que pode ser menor que, igual a ou maior que 1.

O comportamento no pior caso do hash com encadeamento é terrível: todas as  $n$  chaves executam o hash na mesma posição, criando uma lista de comprimento  $n$ . Portanto, o tempo no pior caso para a pesquisa é  $(n)$  mais o tempo necessário para calcular a função hash – não melhor do que seria se usássemos uma lista ligada para todos os elementos. É evidente que as tabelas hash não são usadas por seu desempenho no pior caso. (O hash perfeito, descrito na Seção 11.5, oferece porém um bom desempenho no pior caso quando o conjunto de chaves é estático.)

O desempenho médio do hash depende de como a função hash  $b$  distribui o conjunto de chaves a serem armazenadas entre as  $m$  posições, em média. A Seção 11.3 discute essas questões, mas, por enquanto, devemos supor que qualquer elemento dado tem igual probabilidade de efetuar o hash para qualquer das  $m$  posições, independentemente de onde qualquer outro elemento tenha efetuado o hash. Chamamos essa suposição de hipótese de **hash uniforme simples**.

Para  $j = 0, 1, \dots, m - 1$ , vamos denotar o comprimento da lista  $T[j]$  por  $n_j$ , de forma que

$$n = n_0 + n_1 + \dots + n_{m-1}, \quad (11.1)$$

e o valor médio de  $n_j$  é  $E[n_j] = \alpha = n/m$ .

Supomos que o valor hash  $b(k)$  pode ser calculado no tempo  $O(1)$ , de modo que o tempo necessário para procurar por um elemento com chave  $k$  depende linearmente do comprimento  $n_{b(k)}$  da lista  $T[b(k)]$ . Deixando de lado o tempo  $O(1)$  necessário para calcular a função hash e obter acesso à posição  $b(k)$ , vamos considerar o número esperado de elementos examinados pelo algoritmo de pesquisa, ou seja, o número de elementos na lista  $T[b(k)]$  que são examinados para ver se suas chaves são iguais a  $k$ . Consideraremos dois casos. No primeiro, a pesquisa não é bem-sucedida: nenhum elemento na tabela tem a chave  $k$ . No segundo caso, a pesquisa tem sucesso, encontrando um elemento com chave  $k$ .

### Teorema 11.1

Em uma tabela hash na qual as colisões são resolvidas por encadeamento, uma pesquisa malsucedida demora o tempo esperado  $\Theta(1 + \alpha)$ , sob a hipótese de hash uniforme simples.

**Prova** Sob a hipótese de hash uniforme simples, qualquer chave  $k$  ainda não armazenada na tabela tem igual probabilidade de efetuar o hash para qualquer das  $m$  posições. O tempo médio para pesquisa sem sucesso para uma chave  $k$  é, portanto, o tempo esperado para pesquisar até o

fim da lista  $T[b(k)]$ , que tem o comprimento esperado  $E[n_{b(k)}] = \alpha$ . Assim, o número esperado de elementos examinados em uma pesquisa malsucedida é  $\alpha$ , e o tempo total necessário (incluindo o tempo para se calcular  $b(k)$ ) é  $\Theta(1 + \alpha)$ . ■

A situação para uma pesquisa bem-sucedida é ligeiramente diferente, pois cada lista não tem igual probabilidade de ser pesquisada. Em vez disso, a probabilidade de uma lista ser pesquisada é proporcional ao número de elementos que ela contém. Todavia, o tempo de pesquisa esperado ainda é  $\Theta(1 + \alpha)$ .

### **Teorema 11.2**

Em uma tabela hash na qual as colisões são resolvidas por encadeamento, uma pesquisa bem-sucedida demora o tempo  $\Theta(1 + \alpha)$ , na média, sob a hipótese de hash uniforme simples.

**Prova** Vamos supor que o elemento que está sendo pesquisado tem igual probabilidade de ser qualquer dos  $n$  elementos armazenados na tabela. O número de elementos examinados durante uma pesquisa bem-sucedida para um elemento  $x$  é uma unidade maior que o número de elementos que aparecem antes de  $x$  na lista de  $x$ . Os elementos antes de  $x$  foram todos inseridos após  $x$  ser inserido, porque novos elementos são colocados no início da lista. Para encontrar o número esperado de elementos examinados, tomamos a média, sobre os  $n$  elementos  $x$  na tabela, de 1 mais o número esperado de elementos adicionados à lista de  $x$  depois que  $x$  foi adicionado à lista. Seja  $x_i$  o  $i$ -ésimo elemento inserido na tabela, para  $i = 1, 2, \dots, n$ , e seja  $k_i = chave[x_i]$ . Para chaves  $k_i$  e  $k_j$ , definimos a variável indicadora aleatória  $X_{ij} = I\{b(k_i) = b(k_j)\}$ . Sob a hipótese de hash uniforme simples, temos  $\Pr\{b(k_i) = b(k_j)\}$  e então, pelo Lema 5.1,  $E[X_{ij}] = 1/m$ . Desse modo, o número esperado de elementos examinados em uma pesquisa bem-sucedida é

$$\begin{aligned}
 E & \left[ \frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n X_{ij} \right) \right] \\
 &= \frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n E[X_{ij}] \right) \text{ (por linearidade de expectativa)} \\
 &= \frac{1}{n} \sum_{i=1}^n \left( 1 + \sum_{j=i+1}^n \frac{1}{m} \right) \\
 &= 1 + \frac{1}{nm} \sum_{i=1}^n (n - i) \\
 &= 1 + \frac{1}{nm} \left( \sum_{i=1}^n n - \sum_{i=1}^n i \right) \\
 &= 1 + \frac{1}{nm} \left( n^2 - \frac{n(n+1)}{2} \right) \text{ (pela equação (A.1))} \\
 &= 1 + \frac{n-1}{2m} \\
 &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.
 \end{aligned}$$

Desse modo, o tempo total exigido para uma pesquisa bem-sucedida (incluindo o tempo para calcular a função hash) é  $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$ . ■

O que significa essa análise? Se o número de posições da tabela hash é no mínimo proporcional ao número de elementos na tabela, temos  $n = O(m)$  e, consequentemente,  $\alpha = n/m = O(m)/m = O(1)$ . Assim, a pesquisa demora um tempo constante em média. Tendo em vista que a

inserção demora o tempo  $O(1)$  no pior caso e a eliminação demora o tempo  $O(1)$  no pior caso quando as listas são duplamente ligadas, todas as operações de dicionário podem ser admitidas no tempo  $O(1)$  em média.

## Exercícios

### 11.2-1

Vamos supor que utilizamos uma função hash  $b$  para efetuar o hash de  $n$  chaves distintas em um arranjo  $T$  de comprimento  $m$ . Supondo-se hash uniforme simples, qual é o número esperado de colisões? Mais precisamente, qual é a cardinalidade esperada de  $\{ \{k, l\} : k \neq l \text{ e } b(k) = b(l) \}$ ?

### 11.2-2

Demonstre a inserção das chaves 5, 28, 19, 15, 20, 33, 12, 17, 10 em uma tabela hash com colisões resolvidas por encadeamento. Seja a tabela com 9 posições, e seja a função hash  $b(k) = k \bmod 9$ .

### 11.2-3

O professor Marley apresenta a hipótese de que é possível obter ganhos substanciais de desempenho se modificarmos o esquema de encadeamento de tal modo que cada lista seja mantida em seqüência ordenada. Como a modificação do professor afeta o tempo de execução para pesquisas bem-sucedidas, pesquisas sem sucesso, inserções e eliminações?

### 11.2-4

Sugira como o armazenamento para elementos pode ser alocado e desalocado dentro da própria tabela hash, através da vinculação de todas as posições não utilizadas em uma lista livre. Suponha que uma posição possa armazenar um sinalizador e um elemento mais um ponteiro ou dois ponteiros. Todas as operações de dicionário e de lista livre devem ser executadas no tempo esperado  $O(1)$ . A lista livre precisa ser duplamente ligada, ou seria suficiente uma lista livre simplesmente ligada?

### 11.2-5

Mostre que, se  $|U| > nm$ , existe um subconjunto de  $U$  de tamanho  $n$  consistindo em chaves que efetuam o hash todas para a mesma posição, de tal forma que o tempo de pesquisa do pior caso para o hash com encadeamento é  $(n)$ .

## 11.3 Funções hash

Nesta seção, discutiremos algumas questões relacionadas ao projeto de funções hash de boa qualidade, e depois apresentaremos três esquemas para sua criação. Dois dos esquemas, o hash por divisão e o hash por multiplicação, são heurísticos por natureza, enquanto o terceiro esquema, o hash universal, utiliza a aleatoriedade para oferecer um desempenho que se pode demonstrar ser bom.

### O que faz uma função hash de boa qualidade?

Uma função hash de boa qualidade satisfaz (aproximadamente) à hipótese do hash uniforme simples: cada chave tem igual probabilidade de efetuar o hash para qualquer das  $m$  posições, não importando a posição para onde foi feito o hash de qualquer outra chave. Infelizmente, em geral não é possível verificar essa condição, pois é raro conhecer a distribuição de probabilidades segundo a qual as chaves são obtidas, e as chaves não podem ser obtidas de forma independente.

Ocasionalmente, conhecemos a distribuição. Por exemplo, suponha que as chaves sejam conhecidas como números reais aleatórios  $k$ , independente e uniformemente distribuídos no intervalo  $0 \leq k < 1$ . Nesse caso, podemos mostrar que a função hash

$$h(k) = \lfloor km \rfloor$$

satisfaz à condição de hash uniforme simples.

Na prática, podem ser usadas técnicas heurísticas para criar uma função hash que provavelmente terá bom desempenho. Informações qualitativas sobre a distribuição de chaves podem ser úteis nesse processo de projeto. Por exemplo, considere a tabela de símbolos de um compilador, na qual as chaves são cadeias de caracteres arbitrários que representam identificadores em um programa. É comum que símbolos intimamente relacionados, como `pt` e `pts`, ocorram no mesmo programa. Uma função hash de boa qualidade minimizaria a chance de que tais variações efetuassem hash para a mesma posição.

Uma boa abordagem é derivar a tabela hash de um modo supostamente independente de quaisquer padrões que pudesse existir nos dados. Por exemplo, o “método de divisão” (discutido na Seção 11.3.1) calcula o valor hash como o resto quando a chave é dividida por um número primo especificado. Esse método com freqüência fornece bons resultados, supondo-se que o número primo escolhido não esteja relacionado a quaisquer padrões na distribuição de chaves.

Finalmente, observamos que algumas aplicações das funções hash poderiam exigir propriedades mais fortes que aquelas oferecidas pelo hash uniforme simples. Por exemplo, poderíamos desejar chaves que fossem mais “próximas” em algum sentido, a fim de formar valores hash bastante afastados entre si. (Essa propriedade é especialmente desejável quando estamos usando a sondagem linear, definida na Seção 11.4.) O hash universal, descrito na Seção 11.3.3, com freqüência fornece as propriedades desejadas.

## Interpretação de chaves como números naturais

A maior parte das funções hash supõe que o universo de chaves é o conjunto  $\mathbb{N} = \{0, 1, 2, \dots\}$  de números naturais. Desse modo, se as chaves não são números naturais, deve-se encontrar um modo de interpretá-las como números naturais. Por exemplo, uma cadeia de caracteres pode ser interpretada como um inteiro expresso em uma notação de raiz apropriada. Assim, o identificador `pt` poderia ser interpretado como o par de inteiros decimais (112, 116), pois  $p = 112$  e  $t = 116$  no conjunto de caracteres ASCII; então, expresso como um inteiro de raiz 128, `pt` se torna  $(112 \cdot 128) + 116 = 14452$ . Em geral, é uma tarefa objetiva em qualquer aplicação dada elaborar algum método simples como esse para interpretar cada chave como um número natural (possivelmente grande). No texto a seguir, supomos que as chaves são números naturais.

### 11.3.1 O método de divisão

No **método de divisão** para criação de funções hash, mapeamos uma chave  $k$  para uma de  $m$  posições, tomando o resto de  $k$  dividido por  $m$ . Ou seja, a função hash é

$$h(k) = k \bmod m .$$

Por exemplo, se a tabela hash tem tamanho  $m = 12$  e a chave é  $k = 100$ , então  $h(k) = 4$ . Pelo fato de só exigir uma única operação de divisão, o hash por divisão é bastante rápido.

Quando se utiliza o método de divisão, em geral se evitam certos valores de  $m$ . Por exemplo,  $m$  não deve ser uma potência de 2 pois, se  $m = 2^p$ , então  $h(k)$  será somente o grupo de  $p$  bits de mais baixa ordem de  $k$ . A menos que seja conhecido *a priori* que a distribuição de probabilidades sobre chaves torna todos os padrões de  $p$  bits de mais baixa ordem igualmente prováveis, é melhor fazer a função hash depender de todos os bits da chave. Como o Exercício 11.3-3 lhe pede para mostrar, a escolha de  $m = 2^p - 1$  quando  $k$  é uma cadeia de caracteres interpretada em raiz  $2^p$  pode ser uma escolha ruim, porque a permutação de caracteres de  $k$  não altera seu valor hash.

Um primo não muito próximo a uma potência exata de 2 freqüentemente é uma boa escolha para  $m$ . Por exemplo, suponha que desejamos alocar uma tabela hash, com colisões resolvidas por encadeamento, para conter aproximadamente  $n = 2000$  cadeias de caracteres, onde um caractere tem 8 bits. Não nos importamos de examinar uma média de 3 elementos em uma pesquisa malsucedida, e assim alocamos uma tabela hash de tamanho  $m = 701$ . O número 701 foi escondido porque é um primo próximo a  $2000/3$ , mas não próximo a qualquer potência de 2. Tratando cada chave  $k$  como um inteiro, nossa função hash seria

$$b(k) = k \bmod 701.$$

### 11.3.2 O método de multiplicação

O **método de multiplicação** para criação de funções hash opera em duas etapas. Primeiro, multiplicamos a chave  $k$  por uma constante  $A$  no intervalo  $0 < A < 1$  e extraímos a parte fracionária de  $kA$ . Em seguida, multiplicamos esse valor por  $m$  e tomamos o piso do resultado. Em suma, a função hash é

$$b(k) = \lfloor m(kA \bmod 1) \rfloor,$$

onde “ $kA \bmod 1$ ” significa a parte fracionária de  $kA$ , ou seja,  $kA - \lfloor kA \rfloor$ .

Uma vantagem do método de multiplicação é que o valor de  $m$  não é crítico. Em geral, nós escolhemos de modo a ser uma potência de 2 ( $m = 2^p$  para algum inteiro  $p$ ) pois podemos implementar facilmente a função na maioria dos computadores como a seguir. Suponha que o tamanho da palavra da máquina seja  $w$  bits e que  $k$  se encaixe em uma única palavra. Restringimos  $A$  a ser uma fração da forma  $s/2^w$ , onde  $s$  é um inteiro no intervalo  $0 < s < 2^w$ . Consultando a Figura 11.4, primeiro multiplicamos  $k$  pelo inteiro de  $w$  bits  $s = A \cdot 2^w$ . O resultado é um valor de  $2w$  bits  $r_1 2^w + r_0$ , onde  $r_1$  é a palavra de alta ordem do produto e  $r_0$  é a palavra de baixa ordem do produto. O valor hash de  $p$  bits desejado consiste nos  $p$  bits mais significativos de  $r_0$ .

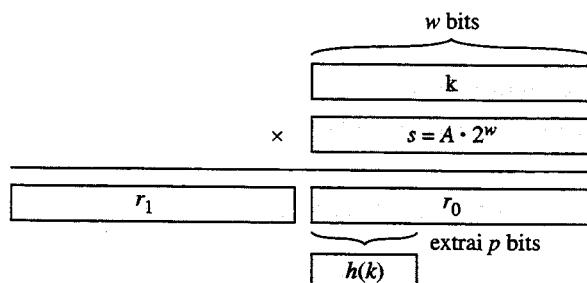


FIGURA 11.4 O método de multiplicação hash. A representação de  $w$  bits da chave  $k$  é multiplicada pelo valor de  $w$  bits  $s = A \cdot 2^w$ , onde  $0 < A < 1$  é uma constante adequada. Os  $p$  bits de mais alta ordem da metade inferior de  $w$  bits do produto formam o valor hash desejado  $b(k)$

Embora esse método funcione com qualquer valor da constante  $A$ , ele funciona melhor com alguns valores que com outros. A escolha ótima depende das características dos dados sobre os quais está sendo feito o hash. Knuth [185] sugere que

$$A \approx (\sqrt{5} - 1)/2 = 0,6180339887... \quad (11.2)$$

deverá funcionar razoavelmente bem.

Como exemplo, suponha que temos  $k = 123456$ ,  $p = 14$ ,  $m = 2^{14} = 16384$  e  $w = 32$ . Adap-tando a sugestão de Knuth, escolhemos  $A$  como a fração da forma  $s/2^{32}$  mais próxima a  $(\sqrt{5} - 1)/2$ , de forma que  $A = 2654435769/2^{32}$ . Então,  $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$ , e assim  $r_1 = 76300$  e  $r_0 = 17612864$ . Os 14 bits mais significativos de  $r_0$  formam o va-lor  $b(k) = 67$ .

### ★ 11.3.3 Hash universal

Se um adversário malicioso escolher as chaves que deverão sofrer o hash de alguma função hash fixa, então ele poderá escolher  $n$  chaves que façam todos o hash para a mesma posição, resultan-do em um tempo médio de recuperação igual a  $\Theta(n)$ . Qualquer função hash fixa é vulnerável a essa espécie terrível de comportamento do pior caso; a única maneira eficaz de melhorar a situa-ção é escolher a função hash *aleatoriamente*, de um modo que seja *independente* das chaves que na realidade estão sendo armazenadas. Essa abordagem, chamada **hash universal**, pode resultar em um desempenho provavelmente bom na média, não importando as chaves que são escolhidas pelo adversário.

A idéia principal por trás do hash universal é selecionar a função hash ao acaso a partir de uma classe de funções cuidadosamente projetada no início a execução. Como no caso de quick-sort, a aleatoriedade garante que nenhuma entrada isolada evocará sempre o comportamento do pior caso. Em virtude da aleatoriedade, o algoritmo poderá se comportar de modo diferente em cada execução, ainda que para a mesma entrada, garantindo um bom desempenho no caso médio para qualquer entrada. Retornando ao exemplo da tabela de símbolos de um compila-dor, verificamos que agora a escolha de identificadores pelo programador não pode provocar um desempenho pobre consistente no hash. O desempenho pobre ocorre apenas quando o compilador escolhe uma função hash aleatória que faz o conjunto de identificadores efetuar o hash de modo ruim, mas a probabilidade de ocorrer essa situação é pequena, e é a mesma para qualquer conjunto de identificadores de tamanho idêntico.

Seja  $\mathcal{H}$  uma coleção finita de funções hash que mapeiam um dado universo  $U$  de chaves no intervalo  $\{0, 1, \dots, m - 1\}$ . Essa coleção é dita ***universal*** se, para cada par de chaves distintas  $k, l \in U$ , o número de funções hash  $\mathcal{H}$  para as quais  $b(k) = b(l)$  é no máximo  $\mathcal{H}$ . Em outras palavras, com uma função hash escolhida aleatoriamente a partir de  $\mathcal{H}$ , a chance de uma colisão entre cha-ves distintas  $k$  e  $l$  não é maior que a chance  $1/m$  de uma colisão se  $b(k)$  e  $b(l)$  fossem escolhidas aleatória e independentemente a partir do conjunto  $\{0, 1, \dots, m - 1\}$ .

O teorema a seguir mostra que uma classe universal de funções hash oferece bom comporta-mento no caso médio. Lembre-se de que  $n_i$  denota o comprimento da lista  $T[i]$ .

#### **Teorema 11.3**

Suponha que uma função hash  $b$  seja escolhida a partir de uma coleção universal de funções hash e seja usada para efetuar o hash de  $n$  chaves em uma tabela  $T$  de tamanho  $m$ , usando o enca-deamento para resolver colisões. Se a chave  $k$  não está na tabela, então o comprimento esperado  $E[n_{b(k)}]$  da lista para a qual a chave  $k$  efetua o hash é no máximo  $\alpha$ . Se a chave  $k$  está na tabela, en-tão o comprimento esperado  $E[n_{b(k)}]$  da lista que contém a chave  $k$  é no máximo  $1 + \alpha$ .

**Prova** Notamos que as expectativas aqui estão acima da escolha da função hash e não depen-dem de quaisquer hipóteses sobre a distribuição das chaves. Para cada par  $k$  e  $l$  de chaves distin-tas, defina a variável indicadora aleatória  $X_{kl} = I\{b(k) = b(l)\}$ . Tendo em vista que, por defini-ção, um único par de chaves colide com probabilidade no máximo  $1/m$ , temos  $\Pr\{b(k) = b(l)\} \leq 1/m$  e, assim, o Lema 5.1 implica que  $E[X_{kl}] \leq 1/m$ .

Em seguida definimos, para cada chave  $k$ , a variável aleatória  $Y_k$  que é igual ao número de chaves diferentes de  $k$  que efetuam o hash para a mesma posição que  $k$ , de forma que

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}.$$

Portanto, temos

$$\begin{aligned}
 E[Y_k] &= E\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\
 &= \sum_{\substack{l \in T \\ l \neq k}} E[X_{kl}] \text{ (por linearidade de expectativa)} \\
 &\leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}.
 \end{aligned}$$

O restante da prova depende do fato da chave  $k$  estar ou não na tabela  $T$ .

- Se  $k \notin T$ , então  $n_{b(k)} = Y_k$  e  $|\{l : l \in T \text{ e } l \neq k\}| = n$ . Desse modo  $E[n_{b(k)}] = E[Y_k] \leq n/m = \alpha$ .
- Se  $k \in T$  então, como a chave  $k$  aparece na lista  $T[b(k)]$  e a contagem  $Y_k$  não inclui a chave  $k$ , temos  $n_{b(k)} = Y_k + 1$  e  $|\{l : l \in T \text{ e } l \neq k\}| = n - 1$ . Assim,  $E[n_{b(k)}] = E[Y_k] \leq (n - 1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$ .

O corolário a seguir nos diz que o hash universal fornece a compensação desejada: agora é impossível um adversário escolher uma seqüência de operações que force o tempo de execução do pior caso. Tornando habilmente aleatória a escolha da função hash em tempo de execução, garantimos que toda seqüência de operações pode ser tratada com um bom tempo de execução esperado.

#### *Corolário 11.4*

Usando-se o hash universal e a resolução de colisões pelo encadeamento em uma tabela com  $m$  posições, demora o tempo esperado  $\Theta(n)$  tratar qualquer seqüência de  $n$  operações INSERT, SEARCH e DELETE contendo  $O(m)$  operações INSERT.

**Prova** Como o número de inserções é  $O(m)$ , temos  $n = O(m)$ , e assim  $\alpha = O(1)$ . As operações INSERT e DELETE demoram um tempo constante e, pelo Teorema 11.3, o tempo esperado para cada operação SEARCH é  $O(1)$ . Desse modo, por linearidade de expectativa, o tempo esperado para a seqüência de operações inteira é  $O(n)$ .

### O projeto de uma classe universal de funções hash

É bastante fácil projetar uma classe universal de funções hash, como um pouco de teoria dos números nos ajudará a demonstrar. Talvez você queira primeiro consultar o Capítulo 31, se estiver pouco familiarizado com a teoria dos números.

Vamos começar escolhendo um número primo  $p$  grande o bastante para que toda chave  $k$  possível esteja no intervalo 0 a  $p - 1$ , inclusive. Seja  $Z_p^*$  o conjunto  $\{0, 1, \dots, p - 1\}$ , e seja  $Z_p$  o conjunto  $\{1, 2, \dots, p - 1\}$ . Tendo em vista que  $p$  é primo, podemos resolver equações de módulo  $p$  com os métodos dados no Capítulo 31. Como supomos que o tamanho do universo de chaves é maior que o número de posições na tabela hash, temos  $p > m$ .

Agora definimos a função hash  $b_{a,b}$  para qualquer  $a \in Z_p^*$  e qualquer  $b \in Z_p$ , usando uma transformação linear seguida por reduções de módulo  $p$ , e depois de módulo  $m$ :

$$b_{a,b}(k) = ((ak + b) \bmod p) \bmod m. \quad (11.3)$$

Por exemplo, com  $p = 17$  e  $m = 6$ , temos  $b_{3,4}(8) = 5$ . A família de todas essas funções hash é

$$\mathcal{H}_{p,m} = \{b_{a,b} : a \in Z_p^* \text{ e } b \in Z_p\}. \quad (11.4) \quad |_{189}$$

Cada função hash  $b_{a,b}$  mapeia  $\mathbf{Z}_p$  para  $\mathbf{Z}_m$ . Essa classe de funções hash tem a interessante propriedade de que o tamanho  $m$  do intervalo de saída é arbitrário – não necessariamente primo –, uma característica que usaremos na Seção 11.5. Tendo em vista que existem  $p-1$  escolhas para  $a$  e que há  $p$  escolhas para  $b$ , existem  $p(p-1)$  funções hash em  $\mathcal{H}_{p,m}$ .

### **Teorema 11.5**

A classe  $\mathcal{H}_{p,m}$  de funções hash definida pelas equações (11.3) e (11.4) é universal.

**Prova** Considere duas chaves distintas  $k$  e  $l$  de  $\mathbf{Z}_p$ , de forma que  $k \neq l$ . Para uma dada função hash  $b_{a,b}$  fazemos

$$\begin{aligned} r &= (ak + b) \bmod p, \\ s &= (al + b) \bmod p. \end{aligned}$$

Primeiro observamos que  $r \neq s$ . Por quê? Observe que

$$r - s \equiv a(k - l) \pmod{p}.$$

Segue-se que  $r \neq s$  porque  $p$  é primo e tanto  $a$  quanto  $(k-l)$  são diferentes de zero em módulo  $p$ , e assim seu produto também deve ser diferente de zero em módulo  $p$ , pelo Teorema 31.6. Então, durante a computação de qualquer  $b_{a,b}$  em  $\mathcal{H}_{p,m}$ , entradas distintas  $k$  e  $l$  mapeiam para valores distintos  $r$  e  $s$  em módulo  $p$ ; ainda não há nenhuma colisão no “nível de mod  $p$ ”. Além disso, cada uma das  $p(p-1)$  escolhas possíveis para o par  $(a, b)$  com  $a \neq 0$  produz um par resultante  $(r, s)$  diferente com  $r \neq s$ , pois podemos resolver para  $a$  e  $b$  dado  $r$  e  $s$ :

$$\begin{aligned} a &= ((r - s)((k - l)^{-1} \bmod p)) \bmod p, \\ b &= (r - ak) \bmod p, \end{aligned}$$

onde  $((k - l)^{-1} \bmod p)$  denota o inverso multiplicativo único em módulo  $p$ , de  $k - l$ . Como existem apenas  $p(p-1)$  pares  $(r, s)$  possíveis com  $r \neq s$ , há uma correspondência de um para um entre pares  $(a, b)$  com  $a \neq 0$  e pares  $(r, s)$  com  $r \neq s$ . Desse modo, para qualquer par de entradas  $k$  e  $l$  dado, se escolhermos  $(a, b)$  uniformemente ao acaso de  $\mathbf{Z}_p^* \times \mathbf{Z}_p$ , o par resultante  $(r, s)$  terá igual probabilidade de ser qualquer par de valores distintos em módulo  $p$ .

Então, segue-se que a probabilidade de que chaves distintas  $k$  e  $l$  colidam é igual à probabilidade de que  $r = s$  quando  $r$  e  $s$  são escolhidos ao acaso como valores distintos em módulo  $p$ . Para um dado valor de  $r$ , dos  $p-1$  valores restantes possíveis para  $s$ , o número de valores  $s$  tais que  $s \neq r$  e  $s \equiv r$  é no máximo

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p+m-1)/m) - 1 \quad (\text{pela desigualdade (3.7)}) \\ &= (p-1)/m. \end{aligned}$$

A probabilidade de  $s$  colidir com  $r$  quando reduzido em módulo  $m$  é no máximo  $((p-1)/m)/(p-1) = 1/m$ .

Assim, para qualquer par de valores distintos  $k, l \in \mathbf{Z}_p$ ,

$$\Pr\{b_{a,b}(k) = b_{a,b}(l)\} \leq 1/m,$$

de forma que  $\mathcal{H}_{p,m}$  é realmente universal.

## Exercícios

### 11.3-1

Suponha que desejamos pesquisar uma lista ligada de comprimento  $n$ , onde cada elemento contém um chave  $k$  juntamente com um valor hash  $b(k)$ . Cada chave é uma cadeia de caracteres longa. Como poderíamos tirar proveito dos valores hash ao procurar na lista por um elemento com uma chave específica?

### 11.3-2

Vamos supor que uma cadeia de  $r$  caracteres sofra hash em  $m$  posições, tratando-a como um número de raiz 128, e depois usando o método de divisão. O número  $m$  é facilmente representado como uma palavra de computador de 32 bits, mas a cadeia de  $r$  caracteres, tratada como um número de raiz 128, ocupa muitas palavras. Como podemos aplicar o método de divisão para calcular o valor hash da cadeia de caracteres sem usar mais de um número constante de palavras de espaço de armazenamento fora da própria cadeia?

### 11.3-3

Considere uma versão do método de divisão, na qual  $b(k) = k \bmod m$ , onde  $m = 2^p - 1$  e  $k$  é uma cadeia de caracteres interpretada na raiz  $2^p$ . Mostre que, se a cadeia  $x$  puder ser derivada da cadeia  $y$  pela permutação de seus caracteres, então  $x$  e  $y$  terão hash para o mesmo valor. Forneça um exemplo de uma aplicação na qual essa propriedade seria indesejável em uma função hash.

### 11.3-4

Considere uma tabela hash de tamanho  $m = 1000$  e a função hash correspondente  $b(k)$  igual a  $\lfloor m(kA \bmod 1) \rfloor$  para  $A = (\sqrt{5} - 1)/2$ . Calcule as localizações para as quais as chaves 61, 62, 63, 64 e 65 estão mapeadas.

### 11.3-5 \*

Defina uma família  $\mathcal{H}$  de funções hash a partir de um conjunto finito  $U$  para um conjunto finito  $B$  como  $\epsilon$ -universal se, para todos os pares de elementos distintos  $k$  e  $l$  em  $U$ ,

$$\Pr \{b(k) = b(l)\} \leq \epsilon$$

onde a probabilidade é obtida sobre a definição da função hash  $b$  ao acaso a partir da família  $\mathcal{H}$ . Mostre que uma família  $\epsilon$ -universal de funções hash deve ter

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

### 11.3-6 \*

Seja  $U$  o conjunto de  $n$ -tuplas de valores obtidos a partir de  $\mathbf{Z}_p$ , e seja  $B = \mathbf{Z}_p$ , onde  $p$  é primo. Defina a função hash  $b_b : U \rightarrow B$  para  $b \in \mathbf{Z}_p$  sobre  $n$ -tupla de entrada  $\langle a_0, a_1, \dots, a_{n-1} \rangle$  a partir de  $U$  como

$$b_b(\langle a_0, a_1, \dots, a_{n-1} \rangle) = \sum_{j=0}^{n-1} a_j b^j$$

e seja  $\mathcal{H} = \{b_b : b \in \mathbf{Z}_p\}$ . Demonstre que  $\mathcal{H}$  é  $((n-1)/p)$ -universal, de acordo com a definição de  $\epsilon$ -universal no Exercício 11.3-5. (Sugestão: Ver Exercício 31.4-4.)

## 11.4 Endereçamento aberto

No **endereçamento aberto**, todos os elementos estão armazenados na própria tabela hash. Ou seja, cada entrada da tabela contém um elemento do conjunto dinâmico ou NIL. Ao procurar por um elemento, examinamos sistematicamente as posições da tabela até encontrarmos o elemento desejado, ou até ficar claro que o elemento não está na tabela. Não existe nenhuma lista e nenhum elemento armazenado fora da tabela, como há no encadeamento. Desse modo, no endereçamento aberto, a tabela hash pode “ficar cheia”, de tal forma que não podem ser feitas inserções adicionais; o fator de carga  $\alpha$  nunca pode exceder 1.

É claro que poderíamos armazenar as listas ligadas para encadeamento no interior da tabela hash, nas posições da tabela hash não utilizadas de outro modo (ver Exercício 11.2-4), mas a vantagem do endereçamento aberto é que ele evita por completo o uso de ponteiros. Em lugar de seguir ponteiros, *calculamos* a seqüência de posições a serem examinadas. A memória extra liberada por não se armazenarem ponteiros fornece à tabela hash um número maior de posições para a mesma quantidade de memória, gerando potencialmente menor número de colisões e recuperação mais rápida.

Para executar a inserção usando o endereçamento aberto, examinamos sucessivamente, ou *sondamos*, a tabela hash até encontrarmos uma posição vazia na qual seja possível inserir a chave. Em lugar de ser fixada na ordem 0, 1, ...,  $m - 1$  (o que exige o tempo de pesquisa  $\Theta(n)$ ), a seqüência de posições demonstrada *depende da chave que está sendo inserida*. Para determinar quais serão as posições sondadas, estendemos a função hash com o objetivo de incluir o número de sondagens (a partir de 0) como uma segunda entrada. Desse modo, a função hash se torna

$$b : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}.$$

Com o endereçamento aberto, exigimos que, para toda chave  $k$ , a *seqüência de sondagem*  $\langle b(k, 0), b(k, 1), \dots, b(k, m - 1) \rangle$

seja uma permutação de  $\langle 0, 1, \dots, m - 1 \rangle$ , de modo que toda posição da tabela hash seja eventualmente considerada uma posição para uma nova chave, à medida que a tabela é preenchida. No pseudocódigo a seguir, supomos que os elementos na tabela hash  $T$  são chaves sem informações satélite; a chave  $k$  é idêntica ao elemento que contém a chave  $k$ . Cada posição contém uma chave ou NIL (se a posição é vazia).

```
HASH-INSERT( $T, k$ )
1  $i \leftarrow 0$ 
2 repeat  $j \leftarrow b(k, i)$ 
3   if  $T[j] = \text{NIL}$ 
4     then  $T[j] \leftarrow k$ 
5     return  $j$ 
6   else  $i \leftarrow i + 1$ 
7 until  $i = m$ 
8 error “hash table overflow”
```

O algoritmo para procurar pela chave  $k$  efetua a sondagem da mesma seqüência de posições que o algoritmo de inserção examinado quando a chave  $k$  foi inserida. Portanto, a pesquisa pode terminar (sem sucesso) ao encontrar uma posição vazia, pois  $k$  teria sido inserido ali e não mais adiante em sua seqüência de sondagem. (Esse argumento pressupõe que as chaves não são eliminadas da tabela hash.) O procedimento HASH-SEARCH toma como entrada uma tabela hash  $T$  e um chave  $k$ , retornando  $j$  se a posição  $j$  contendo a chave  $k$  é encontrada, ou retornando NIL se a chave  $k$  não está presente na tabela  $T$ .

```

HASH-SEARCH( $T, k$ )
1  $i \leftarrow 0$ 
2 repeat  $j \leftarrow b(k, i)$ 
3   if  $T[j] = k$ 
4     then return  $j$ 
5    $i \leftarrow i + 1$ 
6 until  $T[j] = \text{NIL}$  or  $i = m$ 
7 return  $\text{NIL}$ 

```

A eliminação de uma tabela hash de endereço aberto é difícil. Quando eliminamos uma chave da posição  $i$ , não podemos simplesmente assinalar essa posição como vazia, armazenando NIL em seu interior. Fazer isso poderia tornar impossível recuperar qualquer chave  $k$ , se durante a inserção dessa chave tivéssemos sondado a posição  $i$  e encontrado essa posição ocupada. Uma solução é assinalar a posição armazenando nela o valor especial DELETED em lugar de NIL. Então, modificariam o procedimento HASH-INSERT para tratar tal posição como se ela estivesse vazia, de modo que uma nova chave possa ser inserida. Nenhuma modificação de HASH-SEARCH é necessária, pois ele passará sobre valores DELETED enquanto estiver pesquisando. Entretanto, quando usamos o valor especial DELETED, os tempos de pesquisa não são mais dependentes do fator de carga  $\alpha$ . Por essa razão, o encadeamento é mais comumente selecionado como uma técnica de resolução de colisões quando surge a necessidade de eliminar chaves.

Em nossa análise, fazemos a suposição de **hash uniforme**: supomos que cada chave considerada tem igual probabilidade de ter qualquer das  $m!$  permutações de  $\langle 0, 1, \dots, m - 1 \rangle$  como sua sequência de sondagem. O hash uniforme generaliza a noção de hash uniforme simples definida anteriormente para a situação na qual a função hash produz não apenas um número único, mas uma sequência de sondagem inteira. Contudo, o verdadeiro hash uniforme é difícil de implementar e, na prática, são usadas aproximações adequadas (como o hash duplo, definido a seguir).

Três técnicas são usadas comumente para calcular as sequências de sondagem exigidas para o endereçamento aberto: sondagem linear, sondagem quadrática e hash duplo. Todas essas técnicas garantem que  $\langle b(k, 0), b(k, 1), \dots, b(k, m - 1) \rangle$  é uma permutação de  $\langle 0, 1, \dots, m - 1 \rangle$  para cada chave  $k$ . Porém, nenhuma dessas técnicas implementa a hipótese de hash uniforme, pois nenhuma delas é capaz de gerar mais de  $m^2$  sequências de sondagem diferentes (em vez das  $m!$  que o hash uniforme exige). O hash duplo tem o maior número de sequências de sondagem e, como se poderia esperar, parece proporcionar os melhores resultados.

## Sondagem linear

Dada uma função hash comum  $b': U \rightarrow \{0, 1, \dots, m - 1\}$ , à qual nos referimos como uma **função hash auxiliar**, o método de **sondagem linear** usa a função hash

$$b(k, i) = (b'(k) + i) \bmod m$$

para  $i = 0, 1, \dots, m - 1$ . Dada a chave  $k$ , a primeira posição sondada é  $T[b'(k)]$ , isto é, a posição dada pela função hash auxiliar. Em seguida, sondamos a posição  $T[b'(k) + 1]$  e assim por diante até a posição  $T[m - 1]$ . Depois, voltamos às posições  $T[0], T[1], \dots$ , até finalmente sondarmos a posição  $T[b'(k) - 1]$ . Tendo em vista que a posição inicial de sondagem determina toda a sequência de sondagem, só existem  $m$  sequências de sondagem distintas.

A sondagem linear é fácil de implementar, mas sofre de um problema conhecido como **agrupamento primário**. Longas sequências de posições ocupadas são construídas, aumentando o tempo médio de pesquisa. Surgem agrupamentos, pois uma posição vazia precedida por  $i$  posições completas é preenchida em seguida com probabilidade  $(i + 1)/m$ . Sequências de posições ocupadas tendem a ficar mais longas, e o tempo médio de pesquisa aumenta.

## Sondagem quadrática

A **sondagem quadrática** utiliza uma função hash da forma

$$b(k, i) = (b'(k) + c_1i + c_2i^2) \bmod m , \quad (11.5)$$

onde  $b'$  é uma função hash auxiliar,  $c_1$  e  $c_2 \neq 0$  são constantes auxiliares e  $i = 0, 1, \dots, m - 1$ . A posição inicial sondada é  $T[b'(k)]$ ; posições posteriores sondadas são deslocadas por quantidades que dependem de forma quadrática do número da sondagem  $i$ . Esse método funciona muito melhor que a sondagem linear mas, para fazer pleno uso da tabela hash, os valores de  $c_1$ ,  $c_2$  e  $m$  são limitados. O Problema 11-3 mostra um modo de selecionar esses parâmetros. Além disso, se duas chaves têm a mesma posição de sondagem inicial, então suas seqüências de sondagem são iguais, pois  $b(k_1, 0) = b(k_2, 0)$  implica  $b(k_1, i) = b(k_2, i)$ . Essa propriedade conduz a uma forma mais interessante de agrupamento, chamada **agrupamento secundário**. Como na sondagem linear, a sondagem inicial determina a seqüência inteira; assim, apenas  $m$  seqüências de sondagem distintas são utilizadas.

## Hash duplo

O hash duplo é um dos melhores métodos disponíveis para endereçamento aberto, porque as permutações produzidas têm muitas características de permutações escolhidas aleatoriamente. O **hash duplo** usa uma função hash da forma

$$b(k, i) = (b_1(k) + ib_2(k)) \bmod m ,$$

onde  $b_1$  e  $b_2$  são funções hash auxiliares. A posição inicial sondada é  $T[b_1(k)]$ ; posições de sondagem sucessivas são deslocadas a partir de posições anteriores pela quantidade  $b_2(k)$ , módulo  $m$ . Desse modo, diferente do caso de sondagem linear ou quadrática, aqui a seqüência de sondagem depende da chave  $k$  de duas maneiras, pois a posição de sondagem inicial, o deslocamento, ou ambos, podem variar. A Figura 11.5 oferece um exemplo de inserção por hash duplo.

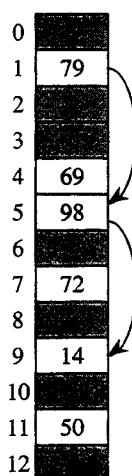


FIGURA 11.5 Inserção por hash duplo. Aqui temos uma tabela hash de tamanho 13 com  $b_1(k) = k \bmod 13$  e  $b_2(k) = 1 + (k \bmod 11)$ . Como  $14 \equiv 1 \bmod 13$  e  $14 \equiv 3 \bmod 11$ , a chave 14 será inserida na posição vazia 9, depois que as posições 1 e 5 tiverem sido examinadas e se descobrir que elas já estão ocupadas

O valor  $b_2(k)$  e o tamanho  $m$  da tabela hash devem ser primos entre si para que a tabela hash inteira possa ser pesquisada. (Ver Exercício 11.4-3.) Uma forma conveniente de assegurar essa condição é permitir que  $m$  seja uma potência de 2 e projetar  $b_2$  de modo que ele sempre produza um número ímpar. Outra maneira é permitir que  $m$  seja primo e projetar  $b_2$  de forma que ele sempre retorne um inteiro positivo menor que  $m$ . Por exemplo, poderíamos escolher  $m$  primo e fazer

$$\begin{aligned} b_1(k) &= k \bmod m, \\ b_2(k) &= 1 + (k \bmod m'), \end{aligned}$$

onde  $m'$  é escolhido com um valor ligeiramente menor que  $m$  (digamos,  $m - 1$ ). Por exemplo, se  $k = 123456$ ,  $m = 701$  e  $m' = 700$ , temos  $b_1(k) = 80$  e  $b_2(k) = 257$ ; assim a primeira sondagem ocorre na posição 80, e depois cada 257-ésima posição (módulo  $m$ ) é examinada até a chave ser encontrada ou todas as posições serem examinadas.

O hash duplo representa um aperfeiçoamento em relação à sondagem linear ou quadrática, pelo fato de serem usadas  $\Theta(m^2)$  seqüências de sondagem, em lugar de  $\Theta(m)$ , pois cada par  $(b_1(k), b_2(k))$  gera uma seqüência de sondagem distinta. Como resultado, o desempenho do hash duplo parece ser muito próximo do desempenho do esquema “ideal” de hash uniforme.

## Análise do hash de endereço aberto

Nossa análise de endereçamento aberto, como nossa análise de encadeamento, é expressa em termos do fator de carga  $\alpha = n/m$  da tabela hash, à medida que  $n$  e  $m$  tendem a infinito. É claro que, no caso do endereçamento aberto, temos no máximo um elemento por posição, e portanto  $n \leq m$ , o que implica  $\alpha \leq 1$ .

Supomos que seja usado o hash uniforme. Nesse esquema idealizado, a seqüência de sondagem  $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$  usada para inserir ou procurar por cada chave  $k$  tem igual probabilidade de ser qualquer permutação de  $\langle 0, 1, \dots, m - 1 \rangle$ . É evidente que uma dada chave tem uma seqüência de sondagem fixa única associada a ela; o que queremos dizer nesse caso é que, considerando a distribuição de probabilidades no espaço de chaves e a operação da função hash sobre as chaves, cada seqüência de sondagem possível é igualmente provável.

Agora, analisamos o número esperado de sondagens para hash com endereçamento aberto, sob a hipótese de hash uniforme, começando com uma análise do número de sondagens realizadas em uma pesquisa malsucedida.

### **Teorema 11.6**

Dada uma tabela hash de endereço aberto com fator de carga  $\alpha = n/m < 1$ , o número esperado de sondagens em uma pesquisa malsucedida é no máximo  $1/(1-\alpha)$ , supondo-se o hash uniforme.

**Prova** Em uma pesquisa malsucedida, toda sondagem exceto a última obtém acesso a uma posição ocupada que não contém a chave desejada, e a última posição sondada está vazia. Vamos definir a variável aleatória  $X$  como o número de sondagens feitas em uma pesquisa malsucedida, e vamos também definir o evento  $A_i$ , para  $i = 1, 2, \dots$ , como o evento em que existe uma  $i$ -ésima sondagem e ela é para uma posição ocupada. Então, o evento  $\{X \geq i\}$  é a interseção dos eventos  $A_1 \cap A_2 \cap \dots \cap A_{i-1}$ . Limitaremos  $\{X \geq i\}$  limitando  $\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\}$ . Pelo Exercício C.2-6,

$$\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \dots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \dots \cap A_{i-2}\}.$$

Tendo em vista que existem  $n$  elementos e  $m$  posições,  $\Pr\{A_1\} = n/m$ . Para  $j > 1$ , a probabilidade de existir uma  $j$ -ésima sondagem e ela ser para uma posição ocupada, dado que as primeiras  $j-1$  sondagens foram para posições ocupadas, é  $(n-j+1)/(m-j+1)$ . Essa probabilidade se

segue porque estariamos encontrando um dos  $(n - (j - 1))$  elementos restantes em uma das  $(m - (j - 1))$  posições não examinadas e, pela hipótese de hash uniforme, a probabilidade é a razão entre essas quantidades. Observando que  $n < m$  implica  $(n - j)/(m - j) \leq n/m$  para todo  $j$  tal que  $0 \leq j < m$ , temos para todo  $i$  tal que  $1 \leq i \leq m$ ,

$$\begin{aligned}\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1}.\end{aligned}$$

Agora, usamos a equação (C.24) para limitar o número esperado de sondagens:

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^i \\ &= \frac{1}{1-\alpha}.\end{aligned}$$

O limite anterior  $1 + \alpha + \alpha^2 + \alpha^3 + \dots$  tem uma interpretação intuitiva: uma sondagem é sempre realizada. Com probabilidade aproximadamente igual a  $\alpha$ , a primeira sondagem encontra uma posição ocupada de forma que é necessária uma segunda sondagem. Com probabilidade aproximadamente igual a  $\alpha^2$ , as duas primeiras posições estão ocupadas, de forma que é necessária uma terceira sondagem e assim por diante.

Se  $\alpha$  é uma constante, o Teorema 11.6 prevê que uma pesquisa malsucedida é executada no tempo  $O(1)$ . Por exemplo, se a tabela hash estiver cheia até a metade, o número médio de sondagens em uma pesquisa malsucedida será  $1/(1 - 0,5) = 2$ . Se ela estiver 90% cheia, o número de sondagens será no máximo  $1/(1 - 0,9) = 10$ .

O Teorema 11.6 nos dá quase imediatamente o desempenho do procedimento HASH-INSERT.

### **Corolário 11.7**

A inserção de um elemento em uma tabela hash de endereço aberto com fator de carga  $\alpha$  exige no máximo  $1/(1 - \alpha)$  sondagens em média, supondo-se o hash uniforme.

**Prova** Um elemento é inserido apenas se existe espaço na tabela, e portanto  $\alpha < 1$ . A inserção de uma chave requer uma pesquisa malsucedida seguida pela colocação da chave na primeira posição vazia encontrada. Desse modo, o número esperado de sondagens é no máximo  $1/(1 - \alpha)$ . ■

O cálculo do número esperado de sondagens para uma pesquisa bem-sucedida exige um pouco mais de trabalho.

### **Teorema 11.8**

Dada uma tabela hash de endereço aberto com fator de carga  $\alpha < 1$ , o número esperado de sondagens em uma pesquisa bem-sucedida é no máximo

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

supondo-se o hash uniforme e considerando-se que cada chave na tabela tem igual probabilidade de ser pesquisada.

**Prova** Uma pesquisa de uma chave  $k$  segue a mesma seqüência de sondagem que foi seguida quando foi inserido o elemento com chave  $k$ . De acordo com o Corolário 11.7, se  $k$  foi a  $(i+1)$ -ésima chave inserida na tabela hash, o número esperado de sondagens efetuadas em uma procura de  $k$  é no máximo  $1/(1-i/m) = m/(m-i)$ . O cálculo da média sobre todas as  $n$  chaves na tabela hash nos dá o número médio de sondagens em uma pesquisa bem-sucedida:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i}$$

$$\frac{1}{\alpha} (H_m - H_{m-n}),$$

onde  $H_i = \sum_{j=1}^i 1/j$  é o  $i$ -ésimo número harmônico (conforme foi definido na equação (A.7)). Usando a técnica de limitar um somatório por uma integral, descrita na Seção A.2, obtemos

$$\begin{aligned} \frac{1}{\alpha} (H_m - H_{m-n}) &= \frac{1}{\alpha} \sum_{k=m-n+1}^m \frac{1}{k} \\ &\leq \frac{1}{\alpha} \int_{m-n}^m (1/x) dx \quad (\text{pela desigualdade (A.12)}) \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

para um limite sobre o número esperado de sondagens em uma pesquisa bem-sucedida. ■

Se a tabela hash estiver cheia até a metade, o número esperado de sondagens em uma pesquisa bem-sucedida será menor que 1,387. Se a tabela hash estiver 90% cheia, o número esperado de sondagens será menor que 2,559.

## **Exercícios**

### **11.4-1**

Considere a inserção das chaves 10, 22, 31, 4, 15, 28, 17, 88, 59 em uma tabela hash de comprimento  $m = 11$  usando o endereçamento aberto com a função hash primário  $b'(k) = k \bmod m$ . Ilustre o resultado da inserção dessas chaves com o uso da sondagem linear, empregando a sondagem quadrática com  $c_1 = 1$  e  $c_2 = 3$ , e com a utilização do hash duplo com  $b_2(k) = 1 + (k \bmod (m-1))$ .

### **11.4-2**

Escreva pseudocódigo para HASH-DELETE da forma descrita no texto e modifique HASH-INSERT para manipular o valor especial DELETED.

### 11.4-3 \*

Suponha que utilizamos o hash duplo para resolver colisões; isto é, usamos a função hash  $b(k, i) = (b_1(k) + ib_2(k)) \text{ mod } m$ . Mostre que, se  $m$  e  $b_2(k)$  têm máximo divisor comum  $d \geq 1$  para alguma chave  $k$ , então uma pesquisa malsucedida para a chave  $k$  examina  $(1/d)$ -ésimo da tabela hash antes de retornar à posição  $b_1(k)$ . Desse modo, quando  $d = 1$ , de forma que  $m$  e  $b_2(k)$  são primos entre si, a pesquisa pode examinar a tabela hash inteira. (Sugestão: Consulte o Capítulo 31.)

### 11.4-4

Considere uma tabela hash de endereço aberto com hash uniforme. Forneça limites superiores sobre o número esperado de sondagens em uma pesquisa malsucedida e sobre o número esperado de sondagens em uma pesquisa bem-sucedida quando o fator de carga é  $3/4$  e quando ele é  $7/8$ .

### 11.4-5 \*

Considere uma tabela hash de endereço aberto com um fator de carga  $\alpha$ . Encontre o valor  $\alpha$  diferente de zero para o qual o número esperado de sondagens em uma pesquisa malsucedida é igual a duas vezes o número esperado de sondagens em uma pesquisa bem-sucedida. Use os limites superiores dados pelos Teoremas 11.6 e 11.8 para esses números esperados de sondagens.

## ★ 11.5 Hash perfeito

Embora o hash seja usado com maior freqüência por seu excelente desempenho esperado, ele pode ser usado para obter um excelente desempenho *no pior caso*, quando o conjunto de chaves é *estático*: uma vez que as chaves estão armazenadas na tabela, o conjunto de chaves nunca se altera. Algumas aplicações têm naturalmente conjuntos estáticos de chaves: considere o conjunto de palavras reservadas em uma linguagem de programação, ou o conjunto de nomes de arquivos em um CD-ROM. Chamamos uma técnica de hash de **hash perfeito** se o número de acessos de memória exigidos no pior caso para executar uma pesquisa é  $O(1)$ .

A idéia básica para criar um esquema de hash perfeito é simples. Usamos um esquema de hash de dois níveis com hash universal em cada nível. A Figura 11.6 ilustra o enfoque.

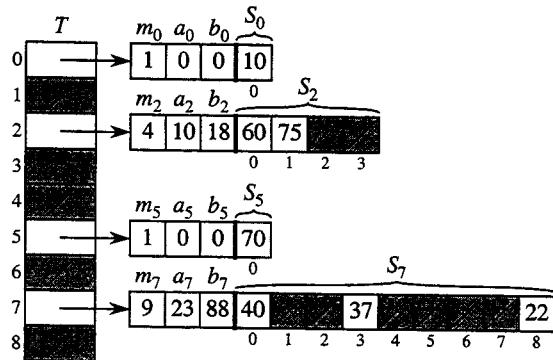
O primeiro nível é essencialmente o mesmo do hash com encadeamento: as  $n$  chaves são submetidas ao hash para  $m$  posições, com o uso de uma função hash  $b$  cuidadosamente selecionada de uma família de funções hash universal.

Porém, em vez de fazer uma lista das chaves que têm hash para a posição  $j$ , usamos uma pequena **tabela hash secundário**  $S_j$  com uma função hash associada  $b_j$ . Escolhendo as funções hash  $b_j$  cuidadosamente, podemos garantir que não haverá colisões no nível secundário.

Contudo, para garantir que não haverá nenhuma colisão no nível secundário, precisaremos fazer o tamanho  $m_j$  da tabela hash  $S_j$  ser o quadrado do número  $n_j$  de chaves com hash para a posição  $j$ . Embora pareça provável que a existência de tal dependência quadrática de  $m_j$  em relação a  $n_j$  torne excessivos os requisitos globais de armazenamento, mostraremos que, escolhendo-se bem a função hash de primeiro nível, o espaço total esperado a ser utilizado ainda será  $O(n)$ .

Usamos funções hash escolhidas a partir das classes universais de funções hash da Seção 1.3.3. A função hash de primeiro nível é escolhida a partir da classe  $\mathcal{H}_{p, m}$  onde, como na Seção 1.3.3,  $p$  é um número primo maior que qualquer valor de chave. As chaves que efetuam o hash para a posição  $j$  têm um novo hash para uma tabela hash secundário  $S_j$  de tamanho  $m_j$ , com a utilização de uma função hash  $b_j$  escolhida a partir da classe  $\mathcal{H}_{p, m_j}$ .<sup>1</sup>

<sup>1</sup> Quando  $n_j = m_j = 1$ , não precisamos realmente de uma função hash para a posição  $j$ ; quando escolhemos uma função hash  $b_{a, b}(k) = ((ak + b) \text{ mod } p) \text{ mod } m_j$  para tal posição, usamos simplesmente  $a = b = 0$ .



**FIGURA 11.6** O uso do hash perfeito para armazenar o conjunto  $K = \{10, 22, 37, 40, 60, 70, 75\}$ . A função hash exterior é  $b(k) = ((ak + b) \bmod p) \bmod m$ , onde  $a = 3$ ,  $b = 42$ ,  $p = 101$  e  $m = 9$ . Por exemplo,  $b(75) = 2$ , então a chave 75 efetua o hash para a posição 2 da tabela  $T$ . Uma tabela hash secundário  $S_j$  armazena todas as chaves que efetuam o hash para a posição  $j$ . O tamanho da tabela hash  $S_j$  é  $m_j$ , e a função hash associada é  $b_j(k) = ((ak + b_j) \bmod p) \bmod m_j$ . Tendo em vista que  $b_2(75) = 1$ , a chave 75 é armazenada na posição 1 da tabela hash secundário  $S_2$ . Não existe nenhuma colisão em quaisquer das tabelas hash secundário, e assim a pesquisa demora um tempo constante no pior caso

Prosseguiremos em duas etapas. Primeiro, vamos determinar como assegurar que as tabelas secundárias não têm nenhuma colisão. Em segundo lugar, mostraremos que a quantidade esperada de memória global utilizada – para a tabela hash primário e todas as tabelas hash secundário – é  $O(n)$ .

### Teorema 11.9

Se armazenarmos  $n$  chaves em uma tabela hash de tamanho  $m = n^2$  usando uma função hash  $b$  escolhida ao acaso de uma classe universal de funções hash, então a probabilidade de haver quaisquer colisões é menor que  $\frac{1}{2}$ .

**Prova** Há  $\binom{n}{2}$  pares de chaves que podem colidir; cada par colide com probabilidade  $1/m$  se  $b$  é escolhida ao acaso a partir de uma família universal  $\mathcal{H}$  de funções hash. Seja  $X$  uma variável aleatória que conta o número de colisões. Quando  $m = n^2$ , o número esperado de colisões é

$$\begin{aligned} E[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \end{aligned}$$

$$< 1/2.$$

(Observe que essa análise é semelhante à análise do paradoxo do aniversário na Seção 5.4.1.) A aplicação da desigualdade de Markov (C.29),  $\Pr\{X \geq t\} \leq E[X]/t$ , com  $t = 1$  completa a prova. ■

Na situação descrita no Teorema 11.9,  $m = n^2$ , segue-se que uma função hash  $b$  escolhida ao acaso a partir de  $\mathcal{H}$  tem maior probabilidade de não ter *nenhuma* colisão. Dado o conjunto  $K$  de  $n$  chaves para hash (lembre-se de que  $K$  é estático), é fácil encontrar então uma função hash  $b$  livre de colisões com algumas experiências aleatórias.

Contudo, quando  $n$  é grande, uma tabela hash de tamanho  $m = n^2$  é excessiva. Assim, adotamos a abordagem de hash de dois níveis e usamos o enfoque do Teorema 11.9 apenas para o hash das entradas em cada posição. Uma função hash  $b$  exterior, ou de primeiro nível, é usada para efetuar o hash das chaves em  $m = n$  posições. Então, se  $n$  chaves têm hash para a posição  $j$ ,

é usada uma tabela hash secundário  $S_j$  de tamanho  $m_j = n_j^2$  para proporcionar uma pesquisa de tempo constante livres de colisões.

Agora, vamos tratar da questão de assegurar que a memória global usada é  $O(n)$ . Como o tamanho  $m_j$  da  $j$ -ésima tabela hash secundário cresce de forma quadrática com o número  $n_j$  de chaves armazenadas, existe o risco de que a quantidade global de armazenamento possa ser excessiva.

Se o tamanho da tabela de primeiro nível é  $m = n$ , então a quantidade de memória usada é  $O(n)$  para a tabela hash primário, para o armazenamento dos tamanhos  $m_j$  das tabelas hash secundário e para o armazenamento dos parâmetros  $a_j$  e  $b_j$  que definem as funções hash secundário  $b_j$  obtidas a partir da classe  $\mathcal{H}_{p, m_j}$  da Seção 11.3.3 (exceto quando  $n_j = 1$  e usamos  $a = b = 0$ ). O teorema a seguir e um corolário fornecem um limite sobre os tamanhos combinados esperados de todas as tabelas hash secundário. Um segundo corolário limita a probabilidade de que o tamanho combinado de todas as tabelas hash secundário seja superlinear.

### **Teorema 11.10**

Se armazenarmos  $n$  chaves em uma tabela hash de tamanho  $m = n$  usando uma função hash  $h$  escolhida ao acaso a partir de uma classe universal de funções hash, então

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n,$$

onde  $n_j$  é o número de chaves que efetuam o hash para a posição  $j$ .

**Prova** Começamos com a identidade a seguir, válida para qualquer inteiro não negativo  $a$ :

$$a^2 = a + 2\binom{a}{2}. \quad (11.6)$$

Temos

$$\begin{aligned} E\left[\sum_{j=0}^{m-1} n_j^2\right] &= E\left[\sum_{j=0}^{m-1} \left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(pela equação (11.6))} \\ &= E\left[\sum_{j=0}^{m-1} n_j\right] + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(por linearidade de expectativa)} \\ &= E[n] + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(pela equação (11.1))} \\ &= n + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] && \text{(pois } n \text{ não é uma variável aleatória).} \end{aligned}$$

Para avaliar o somatório  $\sum_{j=0}^{m-1} \binom{n_j}{2}$ , observamos que ele é simplesmente o número total de colisões. Pelas propriedades de hash universal, o valor esperado desse somatório é no máximo

$$\binom{n}{2} \frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2},$$

200 | pois  $m = n$ . Desse modo,

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + 2 \frac{n-1}{2}$$

$$= 2n - 1$$

$$< 2n .$$

### **Corolário 11.11**

Se armazenarmos  $n$  chaves em uma tabela hash de tamanho  $m = n$  usando uma função hash  $b$  escolhida ao acaso a partir de uma classe universal de funções hash e definirmos o tamanho de cada tabela hash secundário como  $m_j = n_j^2$  para  $j = 0, 1, \dots, m - 1$ , então a quantidade esperada de armazenamento necessário para todas tabelas hash secundário em um esquema de hash perfeito é menor que  $2n$ .

**Prova** Tendo em vista que  $m_j = n_j^2$  para  $j = 0, 1, \dots, m - 1$ , o Teorema 11.10 nos dá

$$E\left[\sum_{j=0}^{m-1} m_j\right] = E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n , \quad (11.7)$$

o que completa a prova.

### **Corolário 11.12**

Se armazenarmos  $n$  chaves em uma tabela hash de tamanho  $m = n$ , usando uma função hash  $b$  escolhida ao acaso a partir de uma classe universal de funções hash, e definirmos o tamanho de cada tabela hash secundário como  $m_j = n_j^2$  para  $j = 0, 1, \dots, m - 1$ , então a probabilidade de que o armazenamento total usado para tabelas hash secundário exceda  $4n$  é menor que  $1/2$ .

**Prova** Aplicamos mais uma vez a desigualdade de Markov (C.29),  $\Pr\{X \geq t\} \leq E[X]/t$ , dessa vez à desigualdade (11.7), com  $X = \sum_{j=0}^{m-1} m_j$  e  $t = 4n$ :

$$\begin{aligned} \Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} &\leq \frac{E\left[\sum_{j=0}^{m-1} m_j\right]}{4n} \\ &< \frac{2n}{4n} \\ &= 1/2 . \end{aligned}$$

Do Corolário 11.12, vemos que testar algumas funções hash escolhidas ao acaso a partir da família universal produzirá rapidamente uma função que utilizará uma quantidade razoável de espaço de armazenamento.

## **Exercícios**

### **11.5-1 \***

Suponha que inserimos  $n$  chaves em uma tabela hash de tamanho  $m$  usando o endereçamento aberto e o hash uniforme. Seja  $p(n, m)$  a probabilidade de não ocorrer nenhuma colisão. Mostre que  $p(n, m) \leq e^{-n(n-1)/2m}$ . (Sugestão: Consulte a equação (3.11).) Demonstre que, quando  $n$  excede  $\sqrt{m}$ , a probabilidade de evitar colisões cai rapidamente a zero.

## Problemas

### 11-1 Limite de sondagem mais longo para o hash

Uma tabela hash de tamanho  $m$  é usada para armazenar  $n$  itens, com  $n \leq m/2$ . O endereçamento aberto é usado para resolução de colisões.

- Supondo hash uniforme, mostre que, para  $i = 1, 2, \dots, n$ , a probabilidade de a  $i$ -ésima inserção exigir estritamente mais de  $k$  sondagens é no máximo  $2^{-k}$ .
- Mostre que, para  $i = 1, 2, \dots, n$ , a probabilidade de a  $i$ -ésima inserção exigir mais de  $2 \lg n$  sondagens é no máximo  $1/n^2$ .

Seja a variável aleatória  $X_i$  que denota o número de sondagens exigidas pela  $i$ -ésima inserção. Você mostrou na parte (b) que  $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$ . Seja a variável aleatória  $X = \max_{1 \leq i \leq n} X_i$  que denota o número máximo de sondagens exigidas por quaisquer das  $n$  inserções.

- Mostre que  $\Pr\{X > 2 \lg n\} \leq 1/n$ .
- Mostre que o comprimento esperado  $E[X]$  da mais longa seqüência de sondagens é  $O(\lg n)$ .

### 11-2 Limite do tamanho da posição para encadeamento

Suponha que temos uma tabela hash com  $n$  posições, com colisões resolvidas por encadeamento, e suponha também que  $n$  chaves sejam inseridas na tabela. Cada chave tem igual probabilidade de sofrer hash para cada posição. Seja  $M$  o número máximo de chaves em qualquer posição após todas as chaves terem sido inseridas. Sua missão é provar um limite superior  $O(\lg n / \lg \lg n)$  sobre  $E[M]$ , o valor esperado de  $M$ .

- Mostre que a probabilidade  $Q_k$  de ocorrer o hash de  $k$  chaves para uma determinada posição é dada por

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

- Seja  $P_k$  a probabilidade de que  $M = k$ , ou seja, a probabilidade de que a posição contendo a maioria das chaves contenha  $k$  chaves. Mostre que  $P_k \leq nQ_k$ .
- Use a aproximação de Stirling, equação (3.17), para mostrar que  $Q_k < e^k / k^k$ .
- Mostre que existe uma constante  $c > 1$  tal que  $Q_{k_0} < 1/n^3$  para  $k_0 = c \lg n / \lg \lg n$ . Conclua que  $P_k < 1/n^2$  para  $k \geq k_0 = c \lg n / \lg \lg n$ .
- Mostre que

$$E[M] \leq \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \leq \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n}.$$

Conclua que  $E[M] = O(\lg n / \lg \lg n)$ .

### 11-3 Sondagem quadrática

Suponha que recebemos uma chave  $k$  para pesquisar uma tabela hash com posições  $0, 1, \dots, m-1$ , e suponha que temos uma função hash  $b$  mapeando o espaço de chaves no conjunto  $\{0, 1, \dots, m-1\}$ . O esquema de pesquisa é dado a seguir.

- 202 | 1. Calcular o valor  $i \leftarrow b(k)$  e definir  $j \leftarrow 0$ .

2. Efetuar a sondagem na posição  $i$  em busca da chave desejada  $k$ . Se a encontrar, ou se essa posição estiver vazia, encerrar a pesquisa.
3. Definir  $j \leftarrow (j + 1) \bmod m$  e  $i \leftarrow (i + j) \bmod m$ , e retornar à Etapa 2. Suponha que  $m$  seja uma potência de 2.
  - a. Mostre que esse esquema é uma instância do esquema geral de “sondagem quadrática”, exibindo as constantes  $c_1$  e  $c_2$  apropriadas para a equação (11.5).
  - b. Prove que esse algoritmo examina cada posição da tabela no pior caso.

#### 11-4 Hash de $k$ -universal e autenticação

Seja  $\mathcal{H} = \{h\}$  uma classe de funções hash na qual cada  $h$  mapeia o universo  $U$  de chaves para  $\{0, 1, \dots, m - 1\}$ . Dizemos que  $\mathcal{H}$  é  **$k$ -universal** se, para toda seqüência fixa de  $k$  chaves distintas  $\langle x^{(1)}, x^{(2)}, \dots, x^{(k)} \rangle$  e para qualquer  $h$  escolhido ao acaso a partir de  $\mathcal{H}$ , a seqüência  $\langle h(x^{(1)}), h(x^{(2)}), \dots, h(x^{(k)}) \rangle$  tem igual probabilidade de ser qualquer uma das  $m^k$  seqüências de comprimento  $k$  com elementos estabelecidos a partir de  $\{0, 1, \dots, m - 1\}$ .

- a. Mostre que, se  $\mathcal{H}$  é 2-universal, então ela é universal.
  - b. Seja  $U$  o conjunto de  $n$ -tuplas de valores obtidos a partir de  $\mathbb{Z}_p$ , e seja  $B = \mathbb{Z}_p$ , onde  $p$  é primo. Para qualquer  $n$ -tupla  $a = \langle a_0, a_1, \dots, a_{n-1} \rangle$  de valores de  $\mathbb{Z}_p$  e para qualquer  $b \in \mathbb{Z}_p$ , defina a função hash  $b_{a,b} : U \rightarrow B$  sobre uma  $n$ -tupla de entrada  $x = \langle x_0, x_1, \dots, x_{n-1} \rangle$  de  $U$  como
- $$b_{a,b}(x) = \left( \sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$
- e seja  $\mathcal{H}$ . Mostre que  $\mathcal{H}$  é 2-universal.
- c. Suponha que Alice e Bob concordam secretamente sobre uma função hash  $b_{a,b}$  de uma família 2-universal  $\mathcal{H}$  de funções hash. Mais tarde, Alice envia pela Internet uma mensagem  $m$  a Bob, na qual  $m \in U$ . Ela autentica essa mensagem para Bob enviando também uma marca de autenticação  $t = b_{a,b}(m)$ , e Bob verifica se o par  $(m, t)$  que ele recebe satisfaz a  $t = b_{a,b}(m)$ . Suponha que um adversário intercepte  $(m, t)$  em trânsito e tente iludir Bob substituindo o par por um par diferente  $(m', t')$ . Mostre que a probabilidade de o adversário ter sucesso na tentativa de fazer Bob aceitar  $(m', t')$  é no máximo  $1/p$ , independente de quanta capacidade de computação o adversário tenha.

## Notas do capítulo

Knuth [185] e Gonnet [126] são excelentes guias de referência para a análise de algoritmos de hash. Knuth credita a H. P. Luhn (1953) a criação de tabelas hash, juntamente com o método de encadeamento para resolução de colisões. Aproximadamente na mesma época, G. M. Amdahl apresentou a idéia de endereçamento aberto.

Carter e Wegman introduziram a noção de classes universais de funções hash em 1979 [52].

Fredman, Komlós e Szemerédi [96] desenvolveram o esquema de hash perfeito para conjuntos estáticos apresentado na Seção 11.5. Uma extensão de seu método para conjuntos dinâmicos, tratamento de inserções e eliminações em tempo esperado amortizado  $O(1)$ , foi apresentada por Dietzfelbinger *et al.* [73].

---

## *Capítulo 12*

# *Árvores de pesquisa binária*

As árvores de pesquisa são estruturas de dados que admitem muitas operações de conjuntos dinâmicos, inclusive SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT e DELETE. Desse modo, uma árvore de pesquisa pode ser usada como um dicionário e também como uma fila de prioridades.

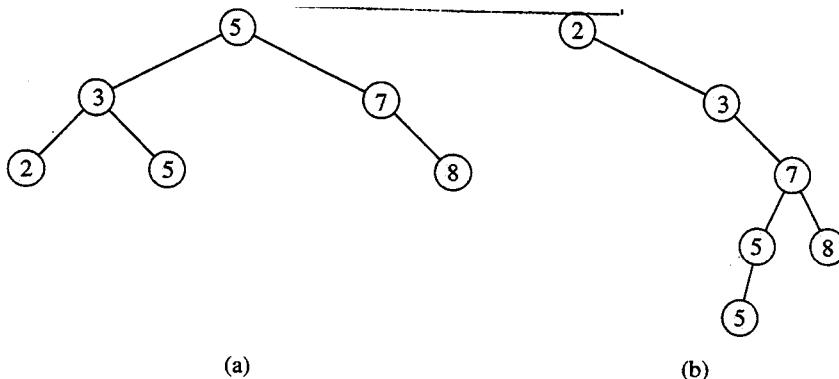
As operações básicas sobre uma árvore de pesquisa binária demoram um tempo proporcional à altura da árvore. No caso de uma árvore binária completa com  $n$  nós, tais operações são executadas em um tempo  $\Theta(\lg n)$  no pior caso. Porém, se a árvore é uma cadeia linear de  $n$  nós, as mesmas operações demoram um tempo  $\Theta(n)$  no pior caso. Veremos na Seção 12.4 que a altura de uma árvore de pesquisa binária construída aleatoriamente é  $O(\lg n)$ , de forma que as operações básicas sobre conjuntos dinâmicos demoram o tempo  $\Theta(\lg n)$  sobre tal árvore.

Na prática, nem sempre podemos garantir que as árvores de pesquisa binária são construídas aleatoriamente, mas existem variações cujo desempenho no pior caso em operações básicas tem uma boa garantia. O Capítulo 13 apresenta uma dessas variações, as árvores vermelho-preto, que têm altura  $O(\lg n)$ . O Capítulo 18 introduz as árvores B (B-trees), que são boas particularmente para manutenção de bancos de dados em um espaço de armazenamento secundário (disco) de acesso aleatório.

Depois de apresentar as propriedades básicas de árvores de pesquisa binária, as próximas seções mostram como percorrê-las para imprimir seus valores em seqüência ordenada, procurar por um valor, encontrar o elemento mínimo ou máximo, encontrar o predecessor ou o sucessor de um elemento e inserir ou eliminar elementos em uma árvore de pesquisa binária. As propriedades matemáticas básicas das árvores são apresentadas no Apêndice B.

### **12.1 O que é uma árvore de pesquisa binária?**

Uma árvore de pesquisa binária é organizada, conforme seu nome sugere, em uma árvore binária, como mostra a Figura 12.1. Uma árvore desse tipo pode ser representada por uma estrutura de dados encadeada em que cada nó é um objeto. Além de um campo *chave* e de dados satélite, cada nó contém campos *esquerdo*, *direito* e *p* que apontam para os nós correspondentes a seu filho da esquerda, seu filho da direita e a seu pai, respectivamente. Se um filho ou o pai estiver ausente, o campo apropriado conterá o valor NIL. O nó raiz é o único nó na árvore cujo campo *pai* é NIL.



**FIGURA 12.1** Árvores de pesquisa binária. Para qualquer nó  $x$ , as chaves na subárvore esquerda de  $x$  são no máximo  $\text{chave}[x]$ , e as chaves na subárvore direita de  $x$  são no mínimo  $\text{chave}[x]$ . Árvores de pesquisa binária diferentes podem representar o mesmo conjunto de valores. O tempo de execução do pior caso para a maioria das operações em árvores de pesquisa é proporcional à altura da árvore. (a) Uma árvore de pesquisa binária em 6 nós com altura 2. (b) Uma árvore de pesquisa binária menos eficiente, com altura 4 e que contém as mesmas chaves

As chaves em uma árvore de pesquisa binária são sempre armazenadas de modo a satisfazem à **propriedade de árvore de pesquisa binária**:

Seja  $x$  um nó em uma árvore de pesquisa binária. Se  $y$  é um nó na subárvore esquerda de  $x$ , então  $\text{chave}[y] \leq \text{chave}[x]$ . Se  $y$  é um nó na subárvore direita de  $x$ , então  $\text{chave}[x] \leq \text{chave}[y]$ .

Desse modo, na Figura 12.1(a), a chave da raiz é 5, as chaves 2, 3 e 5 em sua subárvore esquerda não são maiores que 5, e as chaves 7 e 8 em sua subárvore direita não são menores que 5. A mesma propriedade é válida para todo nó na árvore. Por exemplo, a chave 3 na Figura 12.1(a) não é menor que a chave 2 em sua subárvore esquerda e não é maior que a chave 5 em sua subárvore direita.

A propriedade de árvore de pesquisa binária nos permite imprimir todas as chaves em uma árvore de pesquisa binária em seqüência ordenada, por meio de um simples algoritmo recursivo, chamado **percurso de árvore em ordem**. O nome desse algoritmo deriva do fato de que a chave da raiz de uma subárvore é impressa entre os valores de sua subárvore esquerda e aqueles de sua subárvore direita. (De modo semelhante, um **percurso de árvore de pré-ordem** imprime a raiz antes dos valores em uma ou outra subárvore, e um **percurso de árvore de pós-ordem** imprime a raiz depois dos valores contidos em suas subárvore.) Para usar o procedimento a seguir com o objetivo de imprimir todos os elementos em uma árvore de pesquisa binária  $T$ , chamamos **INORDER-TREE-WALK( $raiz[T]$ )**.

#### INORDER-TREE-WALK( $x$ )

```

1  if  $x \neq \text{NIL}$ 
2    then INORDER-TREE-WALK(esquerda[ $x$ ])
3    print  $\text{chave}[x]$ 
4    INORDER-TREE-WALK(direita[ $x$ ])

```

Como um exemplo, o percurso de árvore em ordem imprime as chaves em cada uma das duas árvores de pesquisa binária da Figura 12.1 na ordem 2, 3, 5, 5, 7, 8. A correção do algoritmo decorre por indução diretamente da propriedade de árvore de pesquisa binária.

Ele demora o tempo  $\Theta(n)$  para percorrer uma árvore de pesquisa binária de  $n$  nós, pois, após a chamada inicial, o procedimento é chamado de forma recursiva exatamente duas vezes para cada nó na árvore – uma vez para seu filho da esquerda e uma vez para seu filho da direita. O

teorema a seguir apresenta uma prova mais formal de que o tempo para executar um percurso de árvore em ordem é linear.

### **Teorema 12.1**

Se  $x$  é a raiz de uma subárvore de  $n$  nós, então a chamada INORDER-TREE-WALK( $x$ ) demora o tempo  $\Theta(n)$ .

**Prova** Seja  $T(n)$  o tempo tomado por INORDER-TREE-WALK quando ele é chamado na raiz de uma subárvore de  $n$  nós. INORDER-TREE-WALK demora um espaço de tempo pequeno e constante em uma subárvore vazia (para o teste  $x \neq \text{NIL}$ ) e então  $T(0) = c$  para alguma constante positiva  $c$ .

Para  $n > 0$ , suponha que INORDER-TREE-WALK seja chamado em um nó  $x$  cuja subárvore esquerda tem  $k$  nós e cuja subárvore direita tem  $n - k - 1$  nós. O tempo para executar INORDER-TREE-WALK( $x$ ) é  $T(n) = T(k) + T(n - k - 1) + d$  para alguma constante positiva  $d$  que reflete o tempo para executar INORDER-TREE-WALK( $x$ ), excetuando-se o tempo gasto em chamadas recursivas.

Usamos o método de substituição para mostrar que  $T(n) = \Theta(n)$ , provando que  $T(n) = (c + d)n + c$ . Para  $n = 0$ , temos  $(c + d) \cdot 0 + c = c = T(0)$ . Para  $n > 0$ , temos

$$\begin{aligned} T(n) &= T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c, \end{aligned}$$

o que completa a prova.

## **Exercícios**

### **12.1-1**

Trace árvores de pesquisa binária de altura 2, 3, 4, 5 e 6 sobre o conjunto de chaves  $\{1, 4, 5, 10, 16, 17, 21\}$ .

### **12.1-2**

Qual a diferença entre a propriedade de árvore de pesquisa binária e a propriedade de heap mínimo (Seção 6.1)? A propriedade de heap pode ser usada para imprimir as chaves de uma árvore de  $n$  nós na seqüência ordenada em tempo  $O(n)$ ? Explique como, ou por que não.

### **12.1-3**

Forneça um algoritmo não recursivo que execute um percurso de árvore em ordem. (*Sugestão:* Existe uma solução fácil que usa uma pilha como uma estrutura de dados auxiliar e uma solução mais complicada, embora elegante, que não emprega nenhuma pilha mas pressupõe que é possível testar a igualdade entre dois ponteiros.)

### **12.1-4**

Forneça algoritmos recursivos que executem caminhos de árvores de pré-ordem e pós-ordem no tempo  $\Theta(n)$  em uma árvore de  $n$  nós.

### **12.1-5**

Mostre que, considerando-se que a ordenação de  $n$  elementos demora o tempo  $\Omega(n \lg n)$  no pior caso no modelo de comparação, qualquer algoritmo baseado em comparação para construção de uma árvore de pesquisa binária a partir de uma lista arbitrária de  $n$  elementos demora o tempo  $\Omega(n \lg n)$  no pior caso.

## 12.2 Consultas em uma árvore de pesquisa binária

A operação mais comum executada sobre uma árvore de pesquisa binária é procurar por uma chave armazenada na árvore. Além da operação SEARCH, árvores de pesquisa binária podem admitir consultas como MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR. Nesta seção, examinaremos essas operações e mostraremos que cada uma delas pode ser admitida no tempo  $O(b)$  sobre uma árvore de pesquisa binária de altura  $b$ .

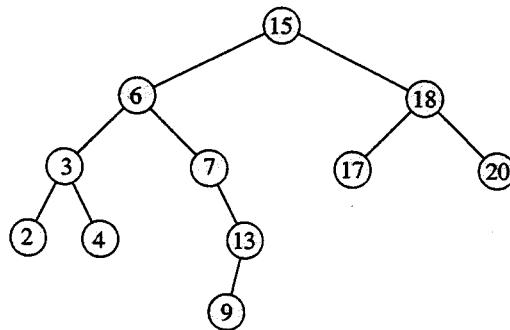


FIGURA 12.2 Consultas em uma árvore de pesquisa binária. Para procurar pela chave 13 na árvore, o caminho  $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$  é seguido a partir da raiz. A chave mínima na árvore é 2, que pode ser encontrada seguindo-se os ponteiros da *esquerda* a partir da raiz. A chave máxima 20 é encontrada seguindo-se os ponteiros da *direita* a partir da raiz. O sucessor do nó com chave 15 é o nó com chave 17, pois ele é a chave mínima na subárvore direita de 15. O nó com chave 13 não tem nenhuma subárvore direita, e assim seu sucessor é seu ancestral mais baixo cujo filho da esquerda também é um ancestral. Nesse caso, o nó com chave 15 é seu sucessor

### Como pesquisar

Usamos o procedimento a seguir para procurar por um nó com uma determinada chave em uma árvore de pesquisa binária. Dado um ponteiro para a raiz da árvore e uma chave  $k$ , TREE-SEARCH retorna um ponteiro para um nó com chave  $k$ , se existir algum; caso contrário, ele retorna NIL.

```
TREE-SEARCH( $x, k$ )
1 if  $x = \text{NIL}$  or  $k = \text{chave}[x]$ 
2   then return  $x$ 
3 if  $k < \text{chave}[x]$ 
4   then return TREE-SEARCH(esquerda[ $x$ ],  $k$ )
5 else return TREE-SEARCH(direita[ $x$ ],  $k$ )
```

O procedimento começa sua pesquisa na raiz e traça um caminho descendente na árvore, como mostra a Figura 12.2. Para cada nó  $x$  que encontra, ele compara a chave  $k$  com  $\text{chave}[x]$ . Se as duas chaves são iguais, a pesquisa termina. Se  $k$  é menor que  $\text{chave}[x]$ , a pesquisa continua na subárvore esquerda de  $x$ , pois a propriedade de árvore de pesquisa binária implica que  $k$  não poderia estar armazenada na subárvore direita. Simetricamente, se  $k$  é maior que  $\text{chave}[x]$ , a pesquisa continua na subárvore direita. Os nós encontrados durante a recursão formam um caminho descendente a partir da raiz da árvore, e portanto o tempo de execução de TREE-SEARCH é  $O(b)$ , onde  $b$  é a altura da árvore.

O mesmo procedimento pode ser escrito de forma iterativa, “estendendo-se” a recursão para o interior de um loop **while**. Na maioria dos computadores, essa versão é mais eficiente.

```

ITERATIVE-TREE-SEARCH( $x, k$ )
1 while  $x \neq \text{NIL}$  e  $k \neq \text{chave}[x]$ 
2   do if  $k < \text{chave}[x]$ 
3     then  $x \leftarrow \text{esquerda}[x]$ 
4     else  $x \leftarrow \text{direita}[x]$ 
5 return  $x$ 

```

## Mínimo e máximo

Um elemento em uma árvore de pesquisa binária cuja chave é um mínimo sempre pode ser encontrado seguindo-se ponteiros filhos da *esquerda* desde a raiz até ser encontrado um valor NIL, como mostra a Figura 12.2. O procedimento a seguir retorna um ponteiro para o elemento mínimo na subárvore com raiz em um determinado nó  $x$ .

TREE-MINIMUM( $x$ )

```

1 while  $\text{esquerda}[x] \neq \text{NIL}$ 
2   do  $x \leftarrow \text{esquerda}[x]$ 
3 return  $x$ 

```

A propriedade de árvore de pesquisa binária garante que TREE-MINIMUM é correto. Se um nó  $x$  não tem nenhuma subárvore esquerda, como toda chave na subárvore direita de  $x$  é pelo menos tão grande quanto  $\text{chave}[x]$ , a chave mínima na subárvore com raiz em  $x$  é  $\text{chave}[x]$ . Se o nó  $x$  tem uma subárvore esquerda, como nenhuma chave na subárvore direita é menor que  $\text{chave}[x]$  e toda chave na subárvore esquerda não é maior que  $\text{chave}[x]$ , a chave mínima na subárvore com raiz em  $x$  pode ser encontrada na subárvore com raiz em  $\text{esquerda}[x]$ .

O pseudocódigo para TREE-MAXIMUM é simétrico.

TREE-MAXIMUM( $x$ )

```

1 while  $\text{direita}[x] \neq \text{NIL}$ 
2   do  $x \leftarrow \text{direita}[x]$ 
3 return  $x$ 

```

Ambos os procedimentos são executados no tempo  $O(b)$  em uma árvore de altura  $b$ , pois, como em TREE-SEARCH, a seqüência de nós encontrados forma um caminho descendente a partir da raiz.

## Sucessor e predecessor

Dado um nó em uma árvore de pesquisa binária, às vezes é importante ser capaz de encontrar seu sucessor na seqüência ordenada determinada por um percurso de árvore em ordem. Se todas as chaves são distintas, o sucessor de um nó  $x$  é o nó com a menor chave maior que  $\text{chave}[x]$ . A estrutura de uma árvore de pesquisa binária nos permite descobrir o sucessor de um nó sem sequer comparar chaves. O procedimento a seguir retorna o sucessor de um nó  $x$  em uma árvore de pesquisa binária se ele existir, e NIL se  $x$  tem a maior chave na árvore.

TREE-SUCCESSOR( $x$ )

```

1 if  $\text{direita}[x] \neq \text{NIL}$ 
2   then return TREE-MINIMUM( $\text{direita}[x]$ )
3  $y \leftarrow p[x]$ 
4 while  $y \neq \text{NIL}$  e  $x = \text{direita}[y]$ 
5   do  $x \leftarrow y$ 
6      $y \leftarrow p[y]$ 
7 return  $y$ 

```

O código para TREE-SUCCESSOR se divide em dois casos. Se a subárvore direita do nó  $x$  for não vazia, então o sucessor de  $x$  será exatamente o nó da extremidade esquerda na subárvore direita, o qual é encontrado na linha 2, chamando-se TREE-MINIMUM( $\text{direita}[x]$ ). Por exemplo, o sucessor do nó com chave 15 na Figura 12.2 é o nó com chave 17.

Por outro lado, como o Exercício 12.2-6 lhe pede para mostrar, se a subárvore direita do nó  $x$  for vazia e  $x$  tiver um sucessor  $y$ , então  $y$  será o ancestral mais baixo de  $x$  cujo filho da esquerda também é um ancestral de  $x$ . Na Figura 12.2, o sucessor do nó com chave 13 é o nó com chave 15. Para encontrar  $y$ , simplesmente subimos a árvore desde  $x$  até encontrarmos um nó que seja o filho da esquerda de seu pai; isso é conseguido através das linhas 3 a 7 de TREE-SUCCESSOR.

O tempo de execução de TREE-SUCCESSOR em uma árvore de altura  $b$  é  $O(b)$ , pois seguimos um caminho para cima na árvore, ou então um caminho para baixo na árvore. O procedimento TREE-PREDECESSOR, o qual é simétrico de TREE-SUCCESSOR, também é executado no tempo  $O(b)$ .

Ainda que as chaves não sejam distintas, definimos o sucessor e o predecessor de qualquer nó  $x$  como o nó retornado por chamadas feitas a TREE-SUCCESSOR( $x$ ) e TREE-PREDECESSOR( $x$ ), respectivamente.

Em suma, demonstramos o teorema a seguir.

### **Teorema 12.2**

As operações sobre conjuntos dinâmicos SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR podem ser executadas em um tempo  $O(b)$  sobre uma árvore de pesquisa binária de altura  $b$ . ■

## **Exercícios**

### **12.2-1**

Suponha que temos números entre 1 e 1000 em uma árvore de pesquisa binária e queremos procurar pelo número 363. Qual das seqüências a seguir *não* poderia ser a seqüência de nós examinados?

- a. 2, 252, 401, 398, 330, 344, 397, 363.
- b. 924, 220, 911, 244, 898, 258, 362, 363.
- c. 925, 202, 911, 240, 912, 245, 363.
- d. 2, 399, 387, 219, 266, 382, 381, 278, 363.
- e. 935, 278, 347, 621, 299, 392, 358, 363.

### **12.2-2**

Escreva versões recursivas dos procedimentos TREE-MINIMUM e TREE-MAXIMUM.

### **12.2-3**

Escreva o procedimento TREE-PREDECESSOR.

### **12.2-4**

O professor Bunyan pensa ter descoberto uma importante propriedade de árvores de pesquisa binária. Suponha que a pesquisa da chave  $k$  em uma árvore de pesquisa binária termine em uma folha. Considere três conjuntos:  $A$ , as chaves à esquerda do caminho de pesquisa;  $B$ , as chaves no caminho de pesquisa; e  $C$ , as chaves à direita do caminho de pesquisa. O professor Bunyan afirma que três chaves quaisquer  $a \in A$ ,  $b \in B$  e  $c \in C$  devem satisfazer a  $a \leq b \leq c$ . Forneça um contra-exemplo mínimo possível para a afirmação do professor.

### **12.2-5**

Mostre que, se um nó em uma árvore de pesquisa binária tem dois filhos, então seu sucessor não tem nenhum filho da esquerda e seu predecessor não tem nenhum filho da direita.

### 12.2-6

Considere uma árvore de pesquisa binária  $T$  cujas chaves são distintas. Mostre que, se a subárvore direita de um nó  $x$  em  $T$  é vazia e  $x$  tem um sucessor  $y$ , então  $y$  é o ancestral mais baixo de  $x$  cujo filho da esquerda também é um ancestral de  $x$ . (Lembre-se de que todo nó é seu próprio ancestral.)

### 12.2-7

Um percurso de árvore em ordem de uma árvore de pesquisa binária de  $n$  nós pode ser implementado encontrando-se o elemento mínimo na árvore com TREE-MINIMUM e, em seguida, fazendo-se  $n - 1$  chamadas a TREE-SUCCESSOR. Prove que esse algoritmo é executado no tempo  $\Theta(n)$ .

### 12.2-8

Prove que, independentemente do nó em que iniciamos em uma árvore de pesquisa binária de altura  $b$ ,  $k$  chamadas sucessivas a TREE-SUCCESSOR demoram o tempo  $O(k + b)$ .

### 12.2-9

Seja  $T$  uma árvore de pesquisa binária cujas chaves são distintas, seja  $x$  um nó de folha e seja  $y$  seu pai. Mostre que  $chave[y]$  é a menor chave em  $T$  maior que  $chave[x]$ , ou a maior chave em  $T$  menor que  $chave[x]$ .

## 12.3 Inserção e eliminação

As operações de inserção e eliminação provocam mudanças no conjunto dinâmico representado por uma árvore de pesquisa binária. A estrutura de dados deve ser modificada para refletir essa mudança, mas de tal modo que a propriedade de árvore de pesquisa binária continue válida. Como veremos, a modificação da árvore para inserir um novo elemento é relativamente direta, mas o tratamento da eliminação é um pouco mais complicado.

### Inserção

Para inserir um novo valor  $v$  em uma árvore de pesquisa binária  $T$ , utilizamos o procedimento TREE-INSERT. O procedimento recebe a passagem de um nó  $z$  para o qual  $chave[z] = v$ ,  $esquerda[z] = \text{NIL}$  e  $direita[z] = \text{NIL}$ . Ele modifica  $T$  e alguns dos campos de  $z$  de tal modo que  $z$  é inserido em uma posição apropriada na árvore.

```
TREE-INSERT( $T, z$ )
1  $y \leftarrow \text{NIL}$ 
2  $x \leftarrow \text{raiz}[T]$ 
3 while  $x \neq \text{NIL}$ 
4   do  $y \leftarrow x$ 
5     if  $chave[z] < chake[x]$ 
6       then  $x \leftarrow esquerda[x]$ 
7       else  $x \leftarrow direita[x]$ 
8    $p[z] \leftarrow y$ 
9   if  $y = \text{NIL}$ 
10  then  $\text{raiz}[T] \leftarrow z$        $\triangleright$  A árvore  $T$  era vazia
11  else if  $chave[z] < chake[y]$ 
12    then  $esquerda[y] \leftarrow z$ 
13    else  $direita[y] \leftarrow z$ 
```

A Figura 12.3 mostra como TREE-INSERT funciona. De forma semelhante aos procedimentos TREE-SEARCH e ITERATIVE-TREE-SEARCH, TREE-INSERT começa na raiz da árvore e traça

um caminho descendente. O ponteiro  $x$  traça o caminho, e o ponteiro  $y$  é mantido como o pai de  $x$ . Após a inicialização, o loop **while** das linhas 3 a 7 faz esses dois ponteiros se deslocarem para baixo na árvore, indo para a esquerda ou direita dependendo da comparação de  $chave[z]$  com  $chave[x]$ , até  $x$  ser definido como NIL. Esse NIL ocupa a posição em que desejamos inserir o item de entrada  $z$ . As linhas 8 a 13 definem os ponteiros que fazem  $z$  ser inserido.

Do mesmo modo que as outras operações primitivas sobre árvores de pesquisa, o procedimento TREE-INSERT é executado em tempo  $O(b)$  sobre uma árvore de altura  $b$ .

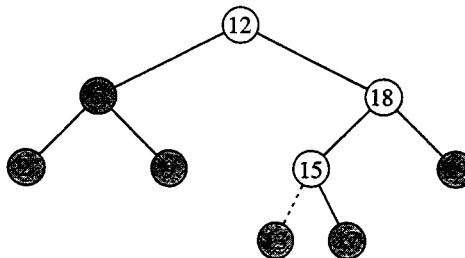


FIGURA 12.3 Inserção de um item com chave 13 em uma árvore de pesquisa binária. Os nós levemente sombreados indicam o caminho desde a raiz até a posição em que o item é inserido. A linha tracejada indica o vínculo na árvore que é acrescentado para se inserir o item

## Eliminação

O procedimento para eliminação de um determinado nó  $z$  de uma árvore de pesquisa binária toma como argumento um ponteiro para  $z$ . O procedimento considera os três casos mostrados na Figura 12.4. Se  $z$  não tem nenhum filho, modificamos seu pai  $p[z]$  para substituir  $z$  por NIL como seu filho. Se o nó tem apenas um único filho, “extraímos”  $z$ , criando um novo vínculo entre seu filho e seu pai. Finalmente, se o nó tem dois filhos, extraímos  $y$ , o sucessor de  $z$ , que não tem nenhum filho da esquerda (ver Exercício 12.2-5) e substituímos a chaves e os dados satélite de  $z$  pela chave e pelos dados satélite de  $y$ .

O código para TREE-DELETE organiza esses três casos de modo um pouco diferente.

```

TREE-DELETE( $T, z$ )
1 if  $esquerda[z] = \text{NIL}$  or  $direita[z] = \text{NIL}$ 
2   then  $y \leftarrow z$ 
3   else  $y \leftarrow \text{TREE-SUCCESSOR}(z)$ 
4 if  $esquerda[y] \neq \text{NIL}$ 
5   then  $x \leftarrow esquerda[y]$ 
6   else  $x \leftarrow direita[y]$ 
7 if  $x \neq \text{NIL}$ 
8   then  $p[x] \leftarrow p[y]$ 
9 if  $p[y] = \text{NIL}$ 
10  then  $raiz[T] \leftarrow x$ 
11  else if  $y = esquerda[p[y]]$ 
12    then  $esquerda[p[y]] \leftarrow x$ 
13    else  $direita[p[y]] \leftarrow x$ 
14 if  $y \neq z$ 
15  then  $chave[z] \leftarrow chave[y]$ 
16  copiar dados satélite de  $y$  em  $z$ 
17 return  $y$ 
  
```

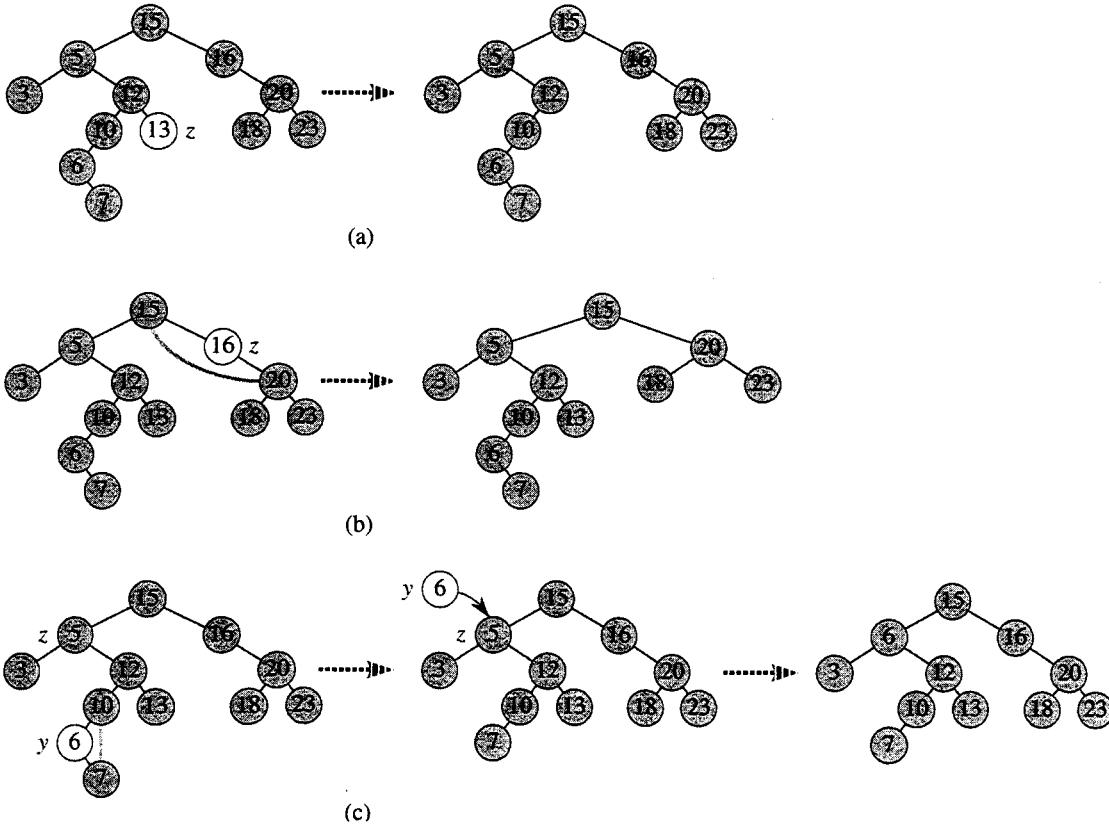


FIGURA 12.4 Eliminação de um nó  $z$  de uma árvore de pesquisa binária. O nó realmente removido depende de quantos filhos  $z$  tem; esse nó está levemente sombreado. (a) Se  $z$  não tem nenhum filho, simplesmente o removemos. (b) Se  $z$  tem apenas um filho, extraímos  $z$ . (c) Se  $z$  tem dois filhos, extraímos seu sucessor  $y$ , que tem no máximo um filho, e depois substituímos a chave e os dados satélite de  $z$  pela chave e os dados satélite de  $y$ .

Nas linhas 1 a 3, o algoritmo determina um nó  $y$  a extraírem. O nó  $y$  é o nó de entrada  $z$  (se  $z$  tem no máximo 1 filho) ou o sucessor de  $z$  (se  $z$  tem dois filhos). Em seguida, nas linhas 4 a 6,  $x$  é definido como o filho não NIL de  $y$ , ou como NIL se  $y$  não tem nenhum filho. O nó  $y$  é extraído nas linhas 7 a 13 pela modificação de ponteiros em  $p[y]$  e  $x$ . A extração de  $y$  é um tanto complicada pela necessidade de tratamento apropriado das condições limite, o que ocorre quando  $x = \text{NIL}$ , ou quando  $y$  é a raiz. Finalmente, nas linhas 14 a 16, se o sucessor de  $z$  foi o nó extraído, a chave e os dados satélite de  $z$  são movidos de  $y$  para  $z$ , sobrepondo a chave e os dados satélite anteriores. O nó  $y$  é retornado na linha 17, de modo que o procedimento de chamada possa reciclá-lo por meio da lista livre. O procedimento é executado no tempo  $O(b)$  em uma árvore de altura  $b$ .

Em suma, demonstramos o teorema a seguir.

### Teorema 12.3

As operações sobre conjuntos dinâmicos INSERT e DELETE podem ser executadas no tempo  $O(b)$  em uma árvore de pesquisa binária de altura  $b$ . ■

## Exercícios

### 12.3-1

Forneça uma versão recursiva do procedimento TREE-INSERT.

### 12.3-2

Suponha que uma árvore de pesquisa binária seja construída pela inserção repetida de valores distintos na árvore. Mostre que o número de nós examinados na pesquisa de um valor na árvore é uma unidade mais o número de nós examinados quando o valor foi inserido inicialmente na árvore.

### 12.3-3

Podemos classificar um dado conjunto de  $n$  números construindo primeiro uma árvore de pesquisa binária contendo esses números (usando TREE-INSERT repetidamente para inserir os números um a um), e depois imprimindo os números por meio de um percurso de árvore em ordem. Quais são os tempos de execução no pior caso e no melhor caso para esse algoritmo de ordenação?

### 12.3-4

Suponha que outra estrutura de dados contenha um ponteiro para um nó  $y$  em uma árvore de pesquisa binária, e suponha que o predecessor  $z$  de  $y$  seja eliminado da árvore pelo procedimento TREE-DELETE. Que problema pode surgir? Como TREE-DELETE pode ser reescrito para resolver esse problema?

### 12.3-5

A operação de eliminação é “comutativa” no sentido de que a eliminação de  $x$  e depois  $y$  de uma árvore de pesquisa binária resulta na mesma árvore que a eliminação de  $y$  e depois  $x$ ? Mostre por que ou forneça um contra-exemplo.

### 12.3-6

Quando o nó  $z$  em TREE-DELETE tem dois filhos, podemos extrair seu predecessor em lugar de seu sucessor. Algumas pessoas argumentam que uma estratégia regular, dando igual prioridade ao predecessor e ao sucessor, produz melhor desempenho empírico. De que maneira TREE-DELETE poderia ser alterado para implementar tal estratégia regular?

## ★ 12.4 Árvores de pesquisa binária construídas aleatoriamente

Mostramos que todas as operações básicas sobre uma árvore de pesquisa binária são executadas no tempo  $O(b)$ , onde  $b$  é a altura da árvore. Contudo, a altura de uma árvore de pesquisa binária varia à medida que são inseridos e eliminados itens. Se, por exemplo, os itens são inseridos em ordem estritamente crescente, a árvore será uma cadeia com altura  $n - 1$ . Por outro lado, o Exercício B.5-4 mostra que  $b \geq \lfloor \lg n \rfloor$ . Como ocorre com o quicksort, podemos mostrar que o comportamento do caso médio é muito mais próximo do melhor caso que do pior caso.

Infelizmente, sabe-se pouco sobre a altura média de uma árvore de pesquisa binária quando são usadas tanto a inserção quanto a eliminação para criá-la. Quando a árvore é criada somente por inserção, a análise se torna mais manejável. Por essa razão, vamos definir uma **árvore de pesquisa binária construída aleatoriamente** sobre  $n$  chaves distintas como aquela que surge da inserção das chaves em ordem aleatória em uma árvore inicialmente vazia, onde cada uma das  $n!$  permutações das chaves de entrada é igualmente provável. (O Exercício 12.4-3 lhe pede para mostrar que essa noção é diferente de supor que toda árvore de pesquisa binária sobre  $n$  chaves é igualmente provável.) O objetivo desta seção é mostrar que a altura esperada de uma árvore de pesquisa binária construída aleatoriamente sobre  $n$  chaves é  $O(\lg n)$ . Supomos que todas as chaves são distintas.

Começamos definindo três variáveis aleatórias que ajudam a medir a altura de uma árvore de pesquisa binária construída aleatoriamente. Denotamos a altura de uma árvore de pesquisa binária construída aleatoriamente sobre  $n$  chaves por  $X_n$ , e definimos a **altura exponencial**  $Y_n = 2^{X_n}$ . Quando construímos uma árvore de pesquisa binária sobre  $n$  chaves, escolhemos uma chave como a da raiz e fazemos  $R_n$  denotar a variável aleatória que contém a ordenação dessa chave dentro do conjunto de  $n$  chaves. O valor de  $R_n$  tem igual probabilidade de ser qualquer elemento do conjunto  $\{1, 2, \dots, n\}$ . Se  $R_n = i$ , então a subárvore esquerda da raiz é uma árvore de pesquisa binária construída aleatoriamente sobre  $i - 1$  chaves, e a subárvore direita é uma árvore de pesquisa binária construída aleatoriamente sobre  $n - i$  chaves. Como a altura de uma árvore binária é uma unidade maior que a maior das alturas das duas subárvores da raiz, a altura exponencial de uma árvore binária é duas vezes a maior das alturas exponenciais das duas subárvores da raiz. Se soubermos que  $R_n = i$ , teremos então

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}) .$$

Como casos básicos, temos  $Y_1 = 1$ , porque a altura exponencial de uma árvore com 1 nó é  $2^0 = 1$  e, por conveniência, definimos  $Y_0 = 0$ .

Em seguida, definimos variáveis indicadoras aleatórias  $Z_{n,1}, Z_{n,2}, \dots, Z_{n,n}$ , onde

$$Z_{n,i} = I\{R_n = i\} .$$

Tendo em vista que  $R_n$  tem igual probabilidade de ser qualquer elemento de  $\{1, 2, \dots, n\}$ , temos que  $\Pr\{R_n = i\} = 1/n$  para  $i = 1, 2, \dots, n$  e, consequentemente, pelo Lema 5.1,

$$E[Z_{n,i}] = 1/n , \quad (12.1)$$

para  $i = 1, 2, \dots, n$ . Como exatamente um valor de  $Z_{n,i}$  é 1 e todos os outros são 0, também temos

$$Y_n = \sum_{i=1}^n Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i})) .$$

Mostraremos que  $E[Y_n]$  é polinomial em  $n$ , que em última análise implicará que  $E[X_n] = O(\lg n)$ .

A variável indicadora aleatória  $Z_{n,i} = I\{R_n = i\}$  é independente dos valores de  $Y_{i-1}$  e  $Y_{n-i}$ . Tendo escolhido  $R_n = i$ , a subárvore da esquerda, cuja altura exponencial é  $Y_{i-1}$ , é construída aleatoriamente sobre as  $i-1$  chaves cujas ordenações são menores que 1. Essa subárvore é exatamente igual a qualquer outra árvore de pesquisa binária construída aleatoriamente sobre  $i-1$  chaves. Exceto pelo número de chaves que contém, a estrutura dessa subárvore não é afetada de modo algum pela escolha de  $R_n = i$ ; consequentemente, as variáveis aleatórias  $Y_{i-1}$  e  $Z_{n,i}$  são independentes. Do mesmo modo, a subárvore da direita, cuja altura exponencial é  $Y_{n-i}$ , é construída aleatoriamente sobre as  $n-i$  chaves cujas ordenações são maiores que  $i$ . Sua estrutura é independente do valor de  $R_n$ , e assim as variáveis aleatórias  $Y_{n-i}$  e  $Z_{n,i}$  são independentes. Daí,

$$\begin{aligned} E[Y_n] &= E\left[\sum_{i=1}^n Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\ &= \sum_{i=1}^n E[Z_{n,i}] E(2 \cdot \max(Y_{i-1}, Y_{n-i})) \quad (\text{por linearidade de expectativa}) \\ &= \sum_{i=1}^n E[Z_{n,i}] E(2 \cdot \max(Y_{i-1}, Y_{n-i})) \quad (\text{por independência}) \\ &= \sum_{i=1}^n \frac{1}{n} \cdot E(2 \cdot \max(Y_{i-1}, Y_{n-i})) \quad (\text{pela equação (12.1)}) \\ &= \frac{2}{n} \sum_{i=1}^n E(2 \cdot \max(Y_{i-1}, Y_{n-i})) \quad (\text{pela equação (C.21)}) \\ &\leq \frac{2}{n} \sum_{i=1}^n (E[Y_{i-1}] + E[Y_{n-i}]) \quad (\text{pelo Exercício C.3-4}) . \end{aligned}$$

Cada termo  $E[Y_0], E[Y_1], \dots, E[Y_{n-1}]$  aparece duas vezes no último somatório, uma vez como  $E[Y_{i-1}]$  e uma vez como  $E[Y_{n-i}]$ , e assim temos a recorrência

$$E[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i]. \quad (12.2)$$

Usando o método de substituição, mostraremos que, para todos os inteiros positivos  $n$ , a recorrência (12.2) tem a solução

$$E[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

Fazendo isso, usaremos a identidade

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{3}. \quad (12.3)$$

(O Exercício 12.4-1 lhe pede para provar essa identidade.)

No caso básico, verificamos que o limite

$$1 = Y_1 = E[Y_1] \leq \frac{1}{4} \binom{1+3}{3} = 1$$

é válido. Para a substituição, temos que

$$\begin{aligned} E[Y_n] &\leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] \\ &= \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} \quad (\text{pela hipótese indutiva}) \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3} \\ &= \frac{1}{n} \binom{n+3}{4} \quad (\text{pela equação 12.3})) \\ &= \frac{1}{n} \cdot \frac{(n+3)!}{4!(n-1)!} \\ &= \frac{1}{4} \cdot \frac{(n+3)!}{3!n!} \\ &= \frac{1}{4} \binom{n+3}{3}. \end{aligned}$$

Limitamos  $E[Y_n]$ , mas nosso objetivo final é limitar  $E[X_n]$ . Como o Exercício 12.4-4 lhe pede para mostrar, a função  $f(x) = 2^x$  é convexa (consulte a Seção C.3). Assim, podemos aplicar a desigualdade de Jensen (C.25), que nos diz que

$$2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n],$$

para derivar que

$$\begin{aligned}
2^{E[X_n]} &\leq \frac{1}{4} \binom{n+3}{3} \\
&= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\
&= \frac{n^3 + 6n^2 + 11n + 6}{24}.
\end{aligned}$$

Usando logaritmos de ambos os lados, temos  $E[X_n] = O(\lg n)$ . Desse modo, demonstramos o seguinte:

#### **Teorema 12.4**

A altura esperada de uma árvore de pesquisa binária construída aleatoriamente sobre  $n$  chaves é  $O(\lg n)$ . ■

### **Exercícios**

#### **12.4-1**

Prove a equação (12.3).

#### **12.4-2**

Descreva uma árvore de pesquisa binária sobre  $n$  nós tal que a profundidade média de um nó na árvore seja  $\Theta(\lg n)$ , mas a altura da árvore seja  $\omega(\lg n)$ . Forneça um limite superior assintótico sobre a altura de uma árvore de pesquisa binária de  $n$  nós na qual a profundidade média de um nó é  $\Theta(\lg n)$ .

#### **12.4-3**

Mostre que a noção de uma árvore de pesquisa binária escolhida aleatoriamente sobre  $n$  chaves, onde cada árvore de pesquisa binária de  $n$  chaves tem igual probabilidade de ser escolhida, é diferente da noção de uma árvore de pesquisa binária construída aleatoriamente dada nesta seção. (Sugestão: Liste as possibilidades quando  $n = 3$ .)

#### **12.4-4**

Mostre que a função  $f(x) = 2^x$  é convexa.

#### **12.4-5** \*

Considere RANDOMIZED-QUICKSORT operando sobre uma seqüência de  $n$  números de entrada. Prove que para qualquer constante  $k > 0$ , todas as  $n!$  permutações de entrada exceto  $O(1/n^k)$  produzem um tempo de execução  $O(n \lg n)$ .

### **Problemas**

#### **12-1 Árvores de pesquisa binária com chaves iguais**

Chaves iguais trazem um problema para a implementação de árvores de pesquisa binária.

- a. Qual é o desempenho assintótico de TREE-INSERT quando é usado para inserir  $n$  itens com chaves idênticas em uma árvore de pesquisa binária inicialmente vazia?

Propomos melhorar TREE-INSERT, testando antes da linha 5 se  $chave[z] = chave[x]$  ou não, e testando antes da linha 11 se  $chave[z] = chave[y]$  ou não. Se a igualdade for válida, implementaremos uma das estratégias a seguir. Para cada estratégia, encontre o desempenho assintótico da inserção de  $n$  itens com chaves idênticas em uma árvore de pesquisa binária inicialmente vazia. (As estratégias são descritas para a linha 5, na qual comparamos as chaves de  $z$  e  $x$ . Substitua  $x$  por  $y$  para chegar às estratégias da linha 11.)

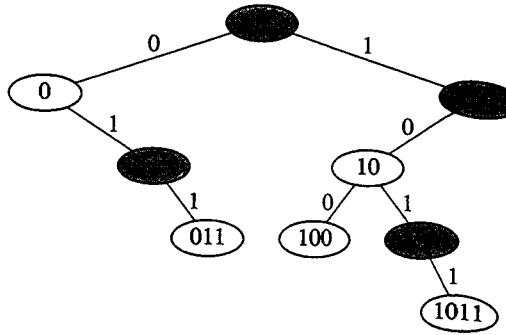


FIGURA 12.5 Uma árvore de raiz que armazena as cadeias de bits 1011, 10, 011, 100 e 0. A chave de cada nó pode ser determinada percorrendo-se o caminho desde a raiz até esse nó. Portanto, não há necessidade de armazenar as chaves nos nós: as chaves são mostradas aqui somente para fins ilustrativos. Os nós estão fortemente sombreados se as chaves correspondentes a eles não estão na árvore; estão presentes apenas para estabelecer um caminho até outros nós.

- b.** Mantenha um sinalizador booleano  $b[x]$  no nó  $x$ , e defina  $x$  como  $\text{esquerdo}[x]$  ou  $\text{direito}[x]$ , de acordo com o valor de  $b[x]$ , que se alterna entre FALSE e TRUE a cada vez que o nó é visitado durante a inserção de um nó com a mesma chave que  $x$ .
- c.** Mantenha uma lista de nós com chaves iguais em  $x$ , e insira  $z$  na lista.
- d.** Defina aleatoriamente  $x$  como  $\text{esquerdo}[x]$  ou  $\text{direito}[x]$ . (Forneça o desempenho do pior caso e derive informalmente o desempenho do caso médio.)

### 12-2 Raiz de árvores

Dadas duas cadeias  $a = a_0a_1 \dots a_p$  e  $b = b_0b_1 \dots b_q$ , onde cada  $a_i$  e cada  $b_j$  pertencem a algum conjunto ordenado de caracteres, dizemos que a cadeia  $a$  é **lexicograficamente menor que** a cadeia  $b$  se

1. existe um inteiro  $j$ , onde  $0 \leq j \leq \min(p, q)$ , tal que  $a_i = b_i$  para todo  $i = 0, 1, \dots, j-1$  e  $a_j < b_j$ , ou
2.  $p < q$  e  $a_i = b_i$  para todo  $i = 0, 1, \dots, p$ .

Por exemplo, se  $a$  e  $b$  são cadeias de bits, então  $10100 < 10110$  pela regra 1 (fazendo-se  $j = 3$ ) e  $10100 < 101000$  pela regra 2. Isso é semelhante à ordenação utilizada nos dicionários de idiomas.

A estrutura de dados **raiz de árvore** mostrada na Figura 12.5 armazena as cadeias de bits 1011, 10, 011, 100 e 0. Quando procuramos por uma chave  $a = a_0a_1 \dots a_p$ , vamos para a esquerda em um nó de profundidade  $i$  se  $a_i = 0$  e para a direita se  $a_i = 1$ . Seja  $S$  um conjunto de cadeias binárias distintas cujos comprimentos produzem a soma  $n$ . Mostre como usar uma raiz de árvore para ordenar lexicograficamente o conjunto  $S$  no tempo  $\Theta(n)$ . No caso do exemplo da Figura 12.6, a saída da ordenação deve ser a seqüência 0, 011, 10, 100, 1011.

### 12-3 Profundidade média de nó em uma árvore de pesquisa binária construída aleatoriamente

Neste problema, provamos que a profundidade média de um nó em uma árvore de pesquisa binária construída aleatoriamente com  $n$  nós é  $O(\lg n)$ . Embora esse resultado seja mais fraco que o do Teorema 12.4, a técnica que empregaremos revelará uma semelhança assombrosa entre a construção de uma árvore de pesquisa binária e a execução do algoritmo RANDOMIZED-QUICKSORT da Seção 7.3.

Definimos o comprimento do caminho total  $P(T)$  de uma árvore binária  $T$  como a soma, ao longo de todos os nós  $x$  em  $T$ , da profundidade do nó  $x$ , que denotamos por  $d(x, T)$ .

- a. Demonstre que a profundidade média de um nó em  $T$  é

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$

Desse modo, desejamos mostrar que o valor esperado de  $P(T)$  é  $O(n \lg n)$ .

- b. Sejam  $T_L$  e  $T_R$  as representações das subárvores esquerda e direita da árvore  $T$ , respectivamente. Mostre que, se  $T$  tem  $n$  nós, então

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

- c. Seja  $P(n)$  o comprimento médio do caminho total de uma árvore de pesquisa binária construída aleatoriamente com  $n$  nós. Mostre que

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n - 1).$$

- d. Mostre que  $P(n)$  pode ser reescrito como

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n).$$

- e. Recordando a análise da versão aleatória de quicksort, conclua que  $P(n) = O(n \lg n)$ .

Em cada invocação recursiva de quicksort, escolhemos um elemento pivô aleatório para partitionar o conjunto de elementos que está sendo ordenado. Cada nó de uma árvore de pesquisa binária partitiona o conjunto de elementos que recaem na subárvore com raiz nesse nó.

- f. Descreva uma implementação de quicksort na qual as comparações para se ordenar um conjunto de elementos são exatamente iguais às comparações utilizadas para inserir os elementos em uma árvore de pesquisa binária. (A ordem em que as comparações são efetuadas pode diferir, mas as mesmas comparações devem ser feitas.)

#### 12-4 Número de árvores binárias diferentes

Seja  $b_n$  o número de árvores binárias diferentes com  $n$  nós. Neste problema, você encontrará uma fórmula para  $b_n$ , como também uma estimativa assintótica.

- a. Mostre que  $b_0 = 1$  e que, para  $n \geq 1$ ,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$

- b. Consultando a definição de uma função geradora no Problema 4-5, seja  $B(x)$  a função geradora

$$B(x) = \sum_{n=0}^{\infty} b_n x^n.$$

Mostre que  $B(x) = xB(x)^2 + 1$  e, consequentemente, um modo de expressar  $B(x)$  em forma fechada é

$$B(x) = \frac{1}{2x} (1 - \sqrt{1 - 4x}) .$$

A **expansão de Taylor** de  $f(x)$  em torno do ponto  $x = a$  é dada por

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!} (x-a)^k ,$$

onde  $f^{(k)}(x)$  é a  $k$ -ésima derivada de  $f$  avaliada em  $x$ .

**c.** Mostre que

$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

(o  $n$ -ésimo **número catalão**) usando a expansão de Taylor de  $\sqrt{1 - 4x}$  em torno de  $x = 0$ . (Se desejar, em vez de usar a expansão de Taylor, você poderá empregar a generalização da expansão binomial (C.4) para expoentes não inteiros  $n$ , onde, para qualquer número real  $n$  e qualquer inteiro  $k$ , interpretamos  $\binom{n}{k}$  como  $n(n-1)\cdots(n-k+1)/k!$  se  $k \geq 0$ , e 0 em caso contrário.)

**d.** Mostre que

$$b_n = \frac{4^n}{\sqrt{\pi n^{3/2}}} (1 + O(1/n)) .$$

## Notas do capítulo

Knuth [185] contém uma boa discussão sobre árvores de pesquisa binária simples, bem como muitas variações. Presume-se que as árvores de pesquisa binária tenham sido descobertas independentemente por vários estudiosos no final da década de 1950. Elas também são discutidas por Knuth [185].

A Seção 15.5 mostrará como construir uma árvore de pesquisa binária ótima quando as freqüências de pesquisa são conhecidas antes da construção da árvore. Ou seja, dadas as freqüências de pesquisa para cada chave e as freqüências de pesquisa para valores que recaem entre chaves na árvore, construímos uma árvore de pesquisa binária para a qual um conjunto de pesquisas que segue essas freqüências examina o número mínimo de nós.

A prova da Seção 12.4 que limita a altura esperada de uma árvore de pesquisa binária construída aleatoriamente se deve a Aslam [23]. Martínez e Roura [211] fornecem algoritmos aleatórios para inserção e eliminação de árvores de pesquisa binária nos quais o resultado de qualquer operação é uma árvore de pesquisa binária aleatória. Porém, sua definição de uma árvore de pesquisa binária aleatória difere ligeiramente da definição de uma árvore de pesquisa binária construída aleatoriamente neste capítulo.

---

## *Capítulo 13*

# *Árvores vermelho-preto*

O Capítulo 12 mostrou que uma árvore de pesquisa binária de altura  $b$  pode implementar quaisquer das operações básicas de conjuntos dinâmicos – como SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT e DELETE – no tempo  $O(b)$ . Assim, as operações de conjuntos são rápidas se a altura da árvore de pesquisa é pequena; porém, se a altura da árvore é grande, o desempenho dessas operações pode não ser melhor do que seria no caso de uma lista ligada. As árvores vermelho-preto constituem um entre muitos esquemas de árvores de pesquisa que são “balanceadas” com o objetivo de garantir que as operações básicas de conjuntos dinâmicos demorem o tempo  $O(\lg n)$  no pior caso.

### **13.1 Propriedades de árvores vermelho-preto**

Uma **árvore vermelho-preto** é uma árvore de pesquisa binária com um bit extra de armazenamento por nó: sua **cor**, que pode ser VERMELHO ou PRETO. Restringindo o modo como os nós podem ser coloridos em qualquer caminho desde a raiz até uma folha, as árvores vermelho-preto asseguram que nenhum desses caminhos será maior que duas vezes o comprimento de qualquer outro, de forma que a árvore é aproximadamente **balanceada**.

Cada nó da árvore contém agora os campos *cor*, *chave*, *esquerda*, *direita* e *p*. Se um filho ou o pai de um nó não existir, o campo do ponteiro correspondente do nó conterá o valor NIL. Trataremos esses valores NIL como ponteiros para nós externos (folhas) da árvore de pesquisa binária e, portanto, os nós normais que conduzem chaves serão tratados como nós internos da árvore.

Uma árvore de pesquisa binária é uma árvore vermelho-preto se satisfaz às seguintes **propriedades vermelho-preto**:

1. Todo nó é vermelho ou preto.
2. A raiz é preta.
3. Toda folha (NIL) é preta.
4. Se um nó é vermelho, então ambos os seus filhos são pretos.
5. Para cada nó, todos os caminhos desde um nó até as folhas descendentes contêm o mesmo número de nós pretos.

A Figura 13.1 mostra um exemplo de árvore vermelho-preto.

Por questão de conveniência no uso de condições limites no código de árvores vermelho-preto, usamos uma única sentinela para representar NIL (ver Seção 10.2). Para uma árvore vermelho-preto  $T$ , a sentinela  $nil[T]$  é um objeto com os mesmos campos que um nó comum na árvore. Seu campo *cor* é PRETO e seus outros campos – *p*, *esquerda*, *direita* e *chave* – podem ser definidos como valores arbitrários. Como mostra a Figura 13.1(b), todos os ponteiros para NIL são substituídos por ponteiros para a sentinela  $nil[T]$ .

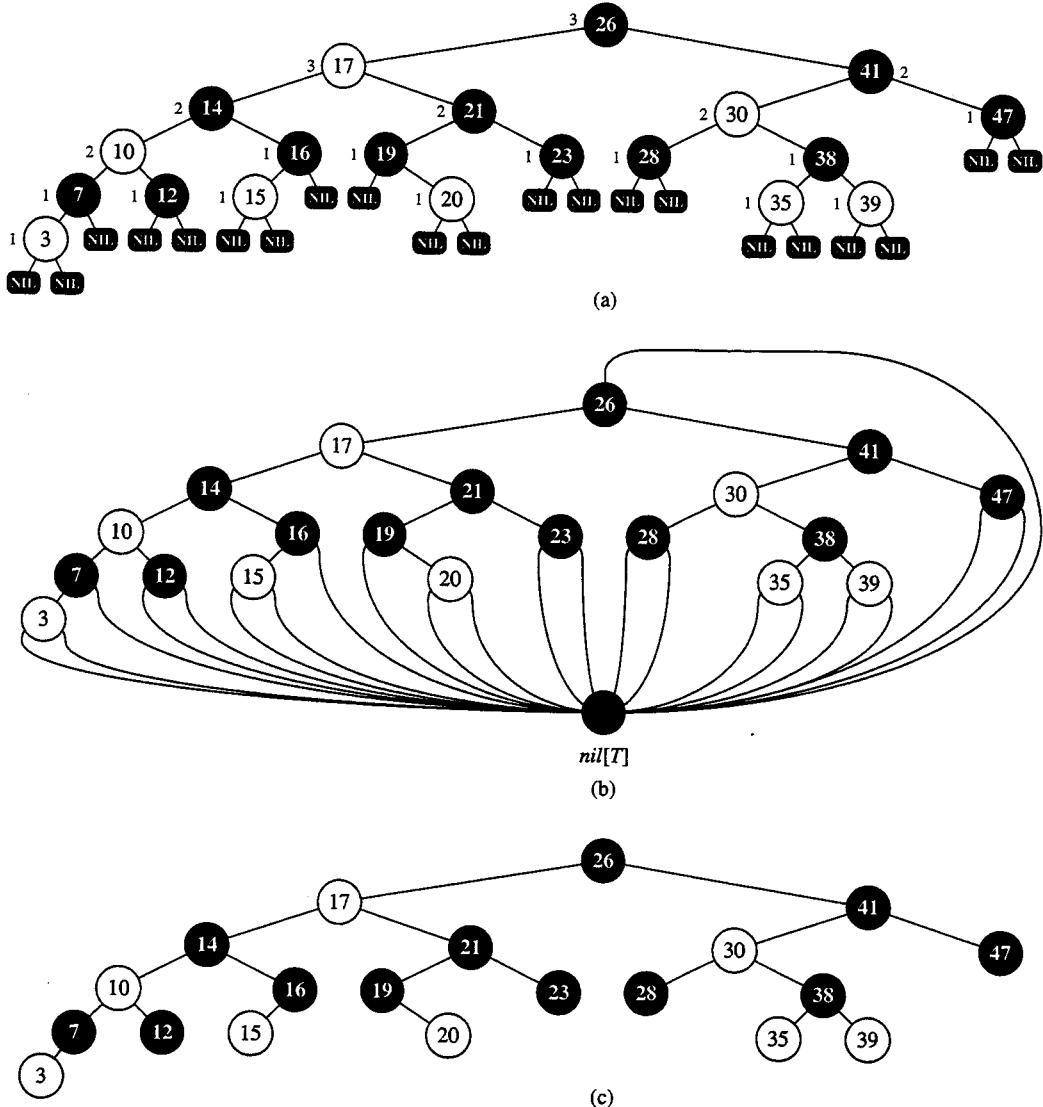


FIGURA 13.1 Uma árvore vermelho-preto com nós pretos escurecidos e nós vermelhos sombreados. Todo nó em uma árvore vermelho-preto é vermelho ou preto, os filhos de um nó vermelho são ambos pretos, e todo caminho simples desde um nó até uma folha descendente contém o mesmo número de nós pretos. (a) Toda folha, mostrada como um NIL, é preta. Cada nó não NIL é marcado com sua altura de preto: os nós NIL têm altura de preto igual a 0. (b) A mesma árvore vermelho-preto, mas com cada NIL substituído pela única sentinela  $nil[T]$ , que é sempre preta, e com alturas de preto omitidas. O pai da raiz também é a sentinela. (c) A mesma árvore vermelho-preto, mas com folhas e o pai da raiz omitidos completamente. Utilizaremos esse estilo de representação no restante deste capítulo

Usamos a sentinela de modo a podermos tratar um filho NIL de um nó  $x$  como um nó comum cujo pai é  $x$ . Poderíamos adicionar um nó de sentinela distinto para cada NIL na árvore, de forma que o pai de cada NIL fosse bem definido, mas essa abordagem desperdiçaria espaço. Em | 221

vez disso, usamos a única sentinela  $\text{nil}[T]$  para representar todos os nós NIL – todas as folhas e o pai da raiz. Os valores dos campos  $p$ , *esquerda*, *direita* e *chave* da sentinela são irrelevantes, embora possamos defini-los durante o curso de um procedimento de acordo com nossa conveniência.

Em geral, limitamos nosso interesse aos nós internos de uma árvore vermelho-preto, pois eles contêm os valores de chaves. No restante deste capítulo, omitiremos as folhas quando desenharmos árvores vermelho-preto, como mostra a Figura 13.1(c).

Chamamos o número de nós pretos em qualquer caminho desde um nó  $x$ , sem incluir esse nó, até uma folha, de **altura de preto** do nó, denotada por  $\text{bh}(x)$ . Pela propriedade 5, a noção de altura de preto é bem definida, pois todos os caminhos descendentes a partir do nó têm o mesmo número de nós pretos. Definimos a altura de preto de uma árvore vermelho-preto como a altura de preto de sua raiz.

O lema a seguir mostra por que as árvores vermelho-preto constituem boas árvores de pesquisa.

### Lema 13.1

Uma árvore vermelho-preto com  $n$  nós internos tem altura no máximo  $2 \lg(n + 1)$ .

**Prova** Primeiro, vamos mostrar que a subárvore com raiz em qualquer nó  $x$  contém pelo menos  $2^{\text{bh}(x)} - 1$  nós internos. Provamos essa afirmativa por indução sobre a altura de  $x$ . Se a altura de  $x$  é 0, então  $x$  deve ser uma folha ( $\text{nil}[T]$ ), e a subárvore com raiz em  $x$  realmente contém pelo menos  $2^{\text{bh}(x)} - 1 = 2^0 - 1 = 0$  nós internos. Para a etapa indutiva, considere um nó  $x$  que tem altura positiva e é um nó interno com dois filhos. Cada filho tem uma altura de preto  $\text{bh}(x)$  ou  $\text{bh}(x) - 1$ , dependendo de sua cor ser vermelha ou preta, respectivamente. Tendo em vista que a altura de um filho de  $x$  é menor que a altura do próprio  $x$ , podemos aplicar a hipótese indutiva para concluir que cada filho tem pelo menos  $2^{\text{bh}(x)-1} - 1$  nós internos. Desse modo, a subárvore com raiz em  $x$  contém pelo menos  $(2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$  nós internos, o que prova a afirmativa.

Para completar a prova do lema, seja  $b$  a altura da árvore. De acordo com a propriedade 4, pelo menos metade dos nós em qualquer caminho simples desde a raiz até uma folha, não incluindo a raiz, deve ser preta. Conseqüentemente, a altura de preto da raiz deve ser pelo menos  $b/2$ ; desse modo,

$$n \geq 2^{b/2} - 1.$$

Movendo-se o valor 1 para o lado esquerdo e usando-se logaritmos em ambos os lados, obtém-se  $\lg(n + 1) \geq b/2$ , ou  $b \leq 2 \lg(n + 1)$ . ■

Uma consequência imediata desse lema é que as operações sobre conjuntos dinâmicos SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR podem ser implementadas no tempo  $O(\lg n)$  em árvores vermelho-preto, pois elas podem ser criadas para execução no tempo  $O(b)$  em uma árvore de pesquisa de altura  $b$  (como mostra o Capítulo 12), e qualquer árvore vermelho-preto sobre  $n$  nós é uma árvore de pesquisa com altura  $O(\lg n)$ . (Evidentemente, referências a NIL nos algoritmos do Capítulo 12 teriam de ser substituídas por  $\text{nil}[T]$ .) Embora os algoritmos TREE-INSERT e TREE-DELETE do Capítulo 12 sejam executados no tempo  $O(\lg n)$  quando é dada uma árvore vermelho-preto como entrada, eles não oferecem suporte direto para as operações de conjuntos dinâmicos INSERT e DELETE, pois não garantem que a árvore de pesquisa binária modificada será uma árvore vermelho-preto. Porém, veremos nas Seções 13.3 e 13.4 que essas duas operações podem de fato ser admitidas no tempo  $O(\lg n)$ .

## Exercícios

### 13.1-1

Desenhe a árvore de pesquisa binária completa de altura 3 sobre as chaves  $\{1, 2, \dots, 15\}$ . Adicione as folhas NIL e defina as cores dos nós de três modos diferentes, tais que as alturas de preto das árvores vermelho-preto resultantes sejam 2, 3 e 4.

### 13.1-2

Desenhe a árvore vermelho-preto que resulta da chamada a TREE-INSERT sobre a árvore da Figura 13.1 com chave 36. Se o nó inserido for vermelho, a árvore resultante será uma árvore vermelho-preto? E se ele for preto?

### 13.1-3

Vamos definir uma **árvore vermelho-preto relaxada** como uma árvore de pesquisa binária que satisfaz às propriedades vermelho-preto 1, 3, 4 e 5. Em outras palavras, a raiz pode ser vermelha ou preta. Considere uma árvore vermelho-preto relaxada  $T$  cuja raiz é vermelha. Se colorirmos a raiz de  $T$  de preto, mas não fizermos nenhuma outra mudança em  $T$ , a árvore resultante será uma árvore vermelho-preto?

### 13.1-4

Vamos que todo nó vermelho em uma árvore vermelho-preto seja “absorvida” por seu pai preto, de forma que os filhos do nó vermelho se tornem filhos do pai preto. (Ignore o que acontece às chaves.) Quais são os graus possíveis de um nó preto depois que todos os seus filhos vermelhos são absorvidos? O que se pode dizer sobre as profundidades das folhas da árvore resultante?

### 13.1-5

Mostre que o mais longo caminho simples desde um nó  $x$  em uma árvore vermelho-preto até uma folha descendente tem comprimento no máximo duas vezes igual ao caminho simples mais curto desde o nó  $x$  até uma folha descendente.

### 13.1-6

Qual é o maior número possível de nós internos em uma árvore vermelho-preto com altura de preto  $k$ ? Qual é o menor número possível?

### 13.1-7

Descreva uma árvore vermelho-preto sobre  $n$  chaves que permita a maior razão possível de nós internos vermelhos para nós internos pretos. Qual é essa razão? Qual árvore tem a menor razão possível, e qual é essa razão?

## 13.2 Rotações

As operações sobre árvores de pesquisa TREE-INSERT e TREE-DELETE, quando executadas sobre uma árvore vermelho-preto com  $n$  chaves, demoram o tempo  $O(\lg n)$ . Considerando-se que elas modificam a árvore, o resultado pode violar as propriedades vermelho-preto enumeradas na Seção 13.1. Para restabelecer essas propriedades, devemos mudar as cores de alguns nós na árvore e também mudar a estrutura de ponteiros.

Mudamos a estrutura de ponteiros através de **rotação**, uma operação local em uma árvore de pesquisa que preserva a propriedade de árvores de pesquisa binária. A Figura 13.2 mostra os dois tipos de rotações: rotações à esquerda e rotações à direita. Quando fazemos uma rotação à esquerda em um nó  $x$ , supomos que seu filho da direita  $y$  não é  $\text{nil}[T]$ . A rotação à esquerda “faz o pivô” em torno da ligação de  $x$  para  $y$ . Ela faz de  $y$  a nova raiz da subárvore, tendo  $x$  como filho da esquerda de  $y$  e o filho da esquerda de  $y$  como filho da direita de  $x$ .

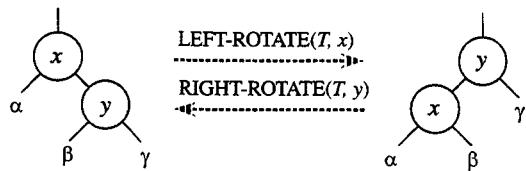


FIGURA 13.2 As operações de rotação em uma árvore de pesquisa binária. A operação  $\text{LEFT-ROTATE}(T, x)$  transforma a configuração dos dois nós da esquerda na configuração da direita, mudando um número constante de ponteiros. A configuração da direita pode ser transformada na configuração da esquerda pela operação inversa  $\text{RIGHT-ROTATE}(T, y)$ . As letras  $\alpha$ ,  $\beta$  e  $\gamma$  representam subárvores arbitrárias. Uma operação de rotação preserva a propriedade de árvores de pesquisa binária: as chaves em  $\alpha$  precedem  $\text{chave}[x]$ , que precede as chaves em  $\beta$ , que precedem  $\text{chave}[y]$ , que precede as chaves em  $\gamma$

O pseudocódigo para  $\text{LEFT-ROTATE}$  pressupõe que  $\text{direita}[x] \neq \text{nil}[T]$  e que o pai da raiz é  $\text{nil}[T]$ .

#### LEFT-ROTATE( $T, x$ )

```

1  $y \leftarrow \text{direita}[x]$             $\triangleright$  Define  $y$ .
2  $\text{direita}[x] \leftarrow \text{esquerda}[y]$      $\triangleright$  Faz da subárvore esquerda de  $y$  a subárvore direita de  $x$ .
3  $p[\text{esquerda}[y]] \leftarrow x$ 
4  $p[y] \leftarrow p[x]$             $\triangleright$  Liga o pai de  $x$  a  $y$ .
5  $\text{if } p[x] = \text{nil}[T]$ 
6    $\text{then } \text{raiz}[T] \leftarrow y$ 
7    $\text{else if } x = \text{esquerda}[p[x]]$ 
8      $\text{then } \text{esquerda}[p[x]] \leftarrow y$ 
9      $\text{else } \text{direita}[p[x]] \leftarrow y$ 
10  $\text{esquerda}[y] \leftarrow x$        $\triangleright$  Coloca  $x$  à esquerda de  $y$ .
11  $p[x] \leftarrow y$ 

```

A Figura 13.3 mostra como  $\text{LEFT-ROTATE}$  opera. O código para  $\text{RIGHT-ROTATE}$  é simétrico. Tanto  $\text{LEFT-ROTATE}$  quanto  $\text{RIGHT-ROTATE}$  são executados no tempo  $O(1)$ . Somente os ponteiros são alterados por uma rotação; todos os outros campos em um nó permanecem os mesmos.

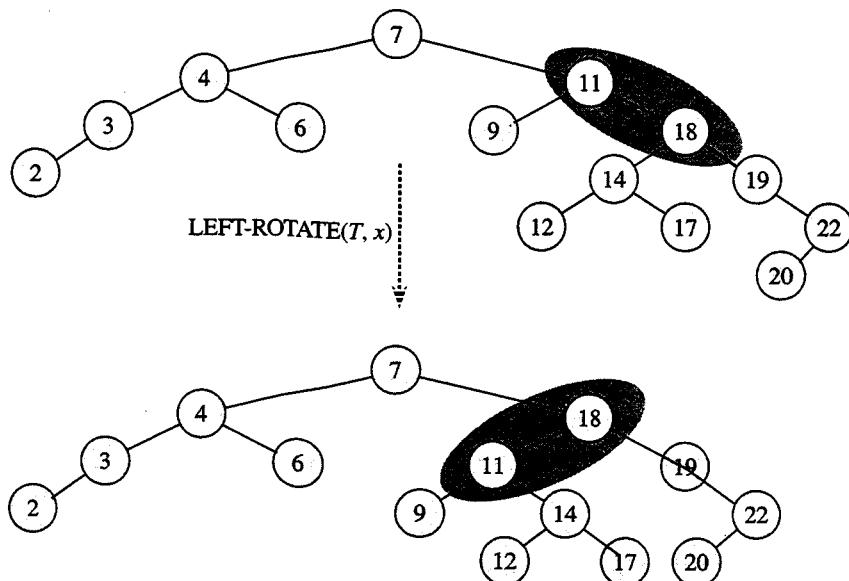


FIGURA 13.3 Um exemplo de como o procedimento  $\text{LEFT-ROTATE}(T, x)$  modifica uma árvore de pesquisa binária. Os percursos de árvore em ordem da árvore de entrada e a árvore modificada produzem a mesma listagem de valores de chaves

## Exercícios

### 13.2-1

Escreva pseudocódigo para RIGHT-ROTATE.

### 13.2-2

Demonstre que, em toda árvore de pesquisa binária de  $n$  nós, existem exatamente  $n - 1$  rotações possíveis.

### 13.2-3

Sejam  $a$ ,  $b$  e  $c$  nós arbitrários nas subárvore  $\alpha$ ,  $\beta$  e  $\gamma$ , respectivamente, na árvore da esquerda da Figura 13.2. De que modo as profundidades de  $a$ ,  $b$  e  $c$  mudam quando é realizada uma rotação à esquerda sobre o nó  $x$  na figura?

### 13.2-4

Mostre que qualquer árvore de pesquisa binária arbitrária de  $n$  nós pode ser transformada em qualquer outra árvore de pesquisa binária arbitrária de  $n$  nós com o uso de  $O(n)$  rotações. (Sugestão: Primeiro, mostre que no máximo  $n - 1$  rotações à direita são suficientes para transformar a árvore em uma cadeia indo para a direita.)

### 13.2-5 \*

Dizemos que uma árvore de pesquisa binária  $T_1$  pode ser **convertida à direita** na árvore de pesquisa binária  $T_2$  se é possível obter  $T_2$  a partir de  $T_1$  por meio de uma série de chamadas a RIGHT-ROTATE. Dê um exemplo de duas árvores  $T_1$  e  $T_2$  tais que  $T_1$  não possa ser convertida à direita em  $T_2$ . Em seguida, mostre que, se uma árvore  $T_1$  pode ser convertida à direita em  $T_2$ , ela pode ser convertida à direita com o uso de  $O(n^2)$  chamadas a RIGHT-ROTATE.

## 13.3 Inserção

A inserção de um nó em uma árvore vermelho-preto de  $n$  nós pode ser realizada no tempo  $O(\lg n)$ . Usamos uma versão ligeiramente modificada do procedimento TREE-INSERT (Seção 12.3) para inserir o nó  $z$  na árvore  $T$  como se ela fosse uma árvore de pesquisa binária comum, e depois colorimos  $z$  de vermelho. Para garantir que as propriedades vermelho-preto serão preservadas, chamamos então um procedimento auxiliar RB-INSERT-FIXUP para recolorir os nós e executar rotações. A chamada RB-INSERT( $T, z$ ) insere o nó  $z$ , cujo campo *chave* já deve ser preenchido, na árvore vermelho-preto  $T$ .

```
RB-INSERT( $T, z$ )
1  $y \leftarrow \text{nil}[T]$ 
2  $x \leftarrow \text{raiz}[T]$ 
3 while  $x \neq \text{nil}[T]$ 
4   do  $y \leftarrow x$ 
5     if chave[ $z$ ] < chave[ $x$ ]
6       then  $x \leftarrow \text{esquerda}[x]$ 
7       else  $x \leftarrow \text{direita}[x]$ 
8  $p[z] \leftarrow y$ 
9 if  $y = \text{nil}[T]$ 
10   then  $\text{raiz}[T] \leftarrow z$ 
11   else if chave[ $z$ ] < chave[ $y$ ]
12     then esquerda[ $y$ ]  $\leftarrow z$ 
13     else direita[ $y$ ]  $\leftarrow z$ 
14 esquerda[ $z$ ]  $\leftarrow \text{nil}[T]$ 
15 direita[ $z$ ]  $\leftarrow \text{nil}[T]$ 
16 cor[ $z$ ]  $\leftarrow \text{VERMELHO}$ 
17 RB-INSERT-FIXUP( $T, z$ )
```

Há quatro diferenças entre os procedimentos TREE-INSERT e RB-INSERT. Primeiro, todas as instâncias de NIL em TREE-INSERT são substituídas por  $nil[T]$ . Em segundo lugar, definimos *esquerda*[ $z$ ] e *direita*[ $z$ ] como  $nil[T]$  nas linhas 14 e 15 de RB-INSERT, a fim de manter a estrutura de árvore adequada. Em terceiro lugar, colorimos  $z$  de vermelho na linha 16. Em quarto lugar, tendo em vista que colorir  $z$  de vermelho pode causar uma violação de uma das propriedades vermelho-preto, chamamos RB-INSERT-FIXUP( $T, z$ ) na linha 17 de RB-INSERT para restaurar as propriedades vermelho-preto.

```

RB-INSERT-FIXUP( $T, z$ )
1 while  $cor[p[z]] = VERMELHO$ 
2   do if  $p[z] = esquerda[p[p[z]]]$ 
3     then  $y \leftarrow direita[p[p[z]]]$ 
4       if  $cor[y] \leftarrow VERMELHO$ 
5         then  $cor[p[z]] \leftarrow PRETO$            ▷ Caso 1
6            $cor[y] \leftarrow PRETO$                  ▷ Caso 1
7            $cor[p[p[x]]] \leftarrow VERMELHO$       ▷ Caso 1
8            $z \leftarrow p[p[z]]$                    ▷ Caso 1
9         else if  $z = direita[p[z]]$ 
10        then  $z \leftarrow p[z]$                   ▷ Caso 2
11          LEFT-ROTATE( $T, z$ )                ▷ Caso 2
12         $cor[p[z]] \leftarrow PRETO$             ▷ Caso 3
13         $cor[p[p[z]]] \leftarrow VERMELHO$       ▷ Caso 3
14        RIGHT-ROTATE( $T, p[p[z]]$ )          ▷ Caso 3
15      else (igual a cláusula then
           com "direita" e "esquerda" trocadas)
16     $cor[raiz[T]] \leftarrow PRETO$ 

```

Para entender como RB-INSERT-FIXUP funciona, desmembraremos nosso exame do código em três etapas principais. Primeiro, determinaremos que violações das propriedades vermelho-preto são introduzidas em RB-INSERT quando o nó  $z$  é inserido e colorido de vermelho. Em segundo lugar, examinaremos a meta global do loop **while** das linhas 1 a 15. Por fim, exploraremos cada um dos três casos<sup>1</sup> em que o loop **while** é dividido e veremos como eles alcançam essa meta. A Figura 13.4 mostra como RB-INSERT-FIXUP opera sobre uma amostra de árvore vermelho-preto.

Quais das propriedades vermelho-preto podem ser violadas na chamada a RB-INSERT-FIXUP? A propriedade 1 certamente continua a ser válida, bem como a propriedade 3, pois ambos os filhos do nó vermelho recém-inserido são a sentinela  $nil[T]$ . A propriedade 5, que nos diz que o número de nós pretos é igual em todo caminho a partir de um dado nó, também é satisfeita, porque o nó  $z$  substitui a sentinela (preta), e o nó  $z$  é vermelho com filhos sentinelas. Desse modo, as únicas propriedades que poderiam ser violadas são a propriedade 2, que exige que a raiz seja preta, e a propriedade 4, que nos diz que um nó vermelho não pode ter um filho vermelho. Ambas as violações possíveis se devem ao fato de  $z$  ser colorido de vermelho. A propriedade 2 é violada se  $z$  é a raiz, e a propriedade 4 é violada se o pai de  $z$  é vermelho. A Figura 13.4(a) mostra uma violação da propriedade 4 depois que o nó  $z$  é inserido.

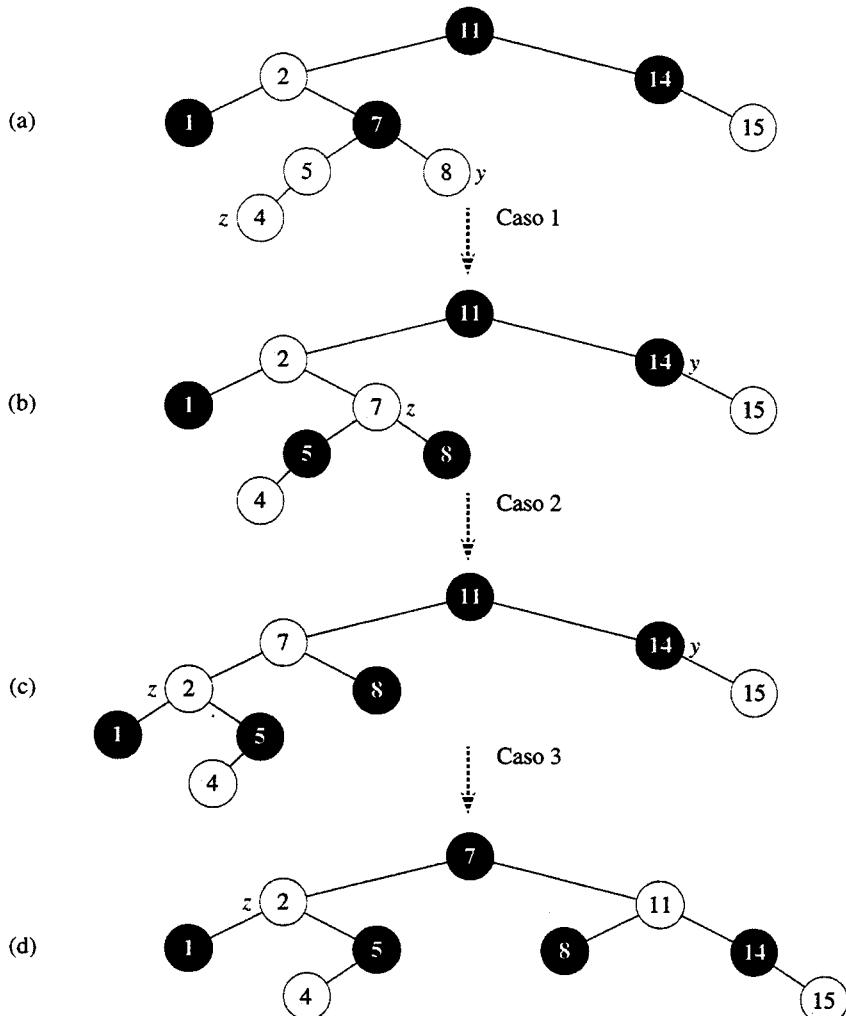
O loop **while** nas linhas 1 a 15 mantém o seguinte invariante de três partes:

No início de cada iteração do loop,

- a.** O nó  $z$  é vermelho.
- b.** Se  $p[z]$  é a raiz, então  $p[z]$  é preto.

---

<sup>1</sup> O caso 2 recai no caso 3, e por conseguinte esses dois casos não são mutuamente exclusivos.



**FIGURA 13.4** A operação de RB-INSERT-FIXUP. (a) Um nó  $z$  depois da inserção. Tendo em vista que  $z$  e seu pai  $p[z]$  são ambos vermelhos, ocorre uma violação da propriedade 4. Tendo em vista que o tio  $y$  de  $z$  é vermelho, pode ser aplicado o caso 1 no código. Os nós são novamente coloridos, e o ponteiro  $z$  é movido para cima na árvore, resultando na árvore mostrada em (b). Mais uma vez,  $z$  e seu pai são ambos vermelhos, mas o tio  $y$  de  $z$  é preto. Como  $z$  é o filho da direita de  $p[z]$ , o caso 2 pode ser aplicado. Uma rotação à esquerda é executada, e a árvore resultante é mostrada em (c). Agora,  $z$  é o filho da esquerda de seu pai, e o caso 3 pode ser aplicado. Uma rotação à direita produz a árvore em (d), que é uma árvore vermelho-preto válida

- c. Se existe uma violação das propriedades vermelho-preto, existe no máximo uma violação, e ela é uma violação da propriedade 2 ou da propriedade 4. Se há uma violação da propriedade 2, ela ocorre porque  $z$  é a raiz e é vermelho. Se há uma violação da propriedade 4, ela ocorre porque tanto  $z$  quanto  $p[z]$  são vermelhos.

A parte (c), que lida com violações de propriedades vermelho-preto, é mais importante para mostrar que RB-INSERT-FIXUP restaura as propriedades vermelho-preto que as partes (a) e (b), que utilizamos no caminho para entender situações no código. Tendo em vista que nos concentraremos no nó  $z$  e nós próximos a ele na árvore, é útil saber da parte (a) que  $z$  é vermelho. Usaremos a parte (b) para mostrar que o nó  $p[p[z]]$  existe quando fazemos referência a ele nas linhas 2, 3, 7, 8, 13 e 14.

Lembre-se de que precisamos mostrar que um loop invariante é verdadeiro antes da primeira iteração do loop, que cada iteração mantém o loop invariante e que o loop invariante nos dá uma propriedade útil ao término do loop.

Começamos com os argumentos de inicialização e término. Em seguida, à medida que examinarmos com mais detalhes como o corpo do loop funciona, demonstraremos que o loop mantém o invariante em cada iteração. Ao longo do caminho, também demonstraremos que há dois resultados possíveis de cada iteração do loop: o ponteiro  $z$  sobe a árvore, ou então algumas rotações são executadas e o loop termina.

**Inicialização:** Antes da primeira iteração do loop, começamos com uma árvore vermelho-preto sem violações e adicionamos um nó vermelho  $z$ . Mostramos que cada parte do invariante é válida no momento em que RB-INSERT-FIXUP é chamado:

- a. Quando RB-INSERT-FIXUP é chamado,  $z$  é o nó vermelho que foi adicionado.
- b. Se  $p[z]$  é a raiz, então  $p[z]$  começou preto e não mudou antes da chamada de RB-INSERT-FIXUP.
- c. Já vimos que as propriedades 1, 3 e 5 são válidas quando RB-INSERT-FIXUP é chamado.

Se há uma violação da propriedade 2, então a raiz vermelha deve ser o nó  $z$  recém-adicionado, o qual é o único nó interno na árvore. Como o pai e ambos os filhos de  $z$  são a sentinela, que é preta, também não há uma violação da propriedade 4. Assim, essa violação da propriedade 2 é a única violação de propriedades vermelho-preto na árvore inteira.

Se há uma violação da propriedade 4, como os filhos do nó  $z$  são sentinelas pretas e a árvore não tinha outras violações antes de  $z$  ser adicionado, a violação tem de ter ocorrido porque tanto  $z$  quanto  $p[z]$  são vermelhos. Além disso, não há outras violações de propriedades vermelho-preto.

**Término:** Quando o loop termina, ele o faz porque  $p[z]$  é preto. (Se  $z$  é a raiz, então  $p[z]$  é a sentinela  $nil[T]$ , que é preta.) Desse modo, não há violação da propriedade 4 no término do loop. Pelo loop invariante, a única propriedade que poderia deixar de ser válida é a propriedade 2. A linha 16 também restaura essa propriedade, de forma que, quando RB-INSERT-FIXUP termina, todas as propriedades vermelho-preto são válidas.

**Manutenção:** Na realidade, há seis casos há seis casos a considerar no loop **while**, mas três deles são simétricos aos outros três, dependendo do pai  $p[z]$  de  $z$  ser um filho da esquerda ou um filho da direita do avô  $p[p[z]]$  de  $z$ , o que é determinado na linha 2. Fornecemos o código apenas para a situação na qual  $p[z]$  é um filho da esquerda. O nó  $p[p[z]]$  existe pois, pela parte (b) do loop invariante, se  $p[z]$  é a raiz, então  $p[z]$  é preto. Tendo em vista que entramos em uma iteração de loop somente se  $p[z]$  é vermelho, sabemos que  $p[z]$  não pode ser a raiz. Conseqüentemente,  $p[p[z]]$  existe.

O caso 1 se distingue dos casos 2 e 3 pela cor do irmão do pai de  $z$ , ou “tio”. A linha 5 faz  $y$  apontar para o tio  $direita[p[p[z]]]$  de  $z$ , e é feito um teste na linha 4. Se  $y$  é vermelho, então o caso 1 é executado. Do contrário, o controle passa para os casos 2 e 3. Em todos os três casos, o avô  $p[p[z]]$  de  $z$  é preto, pois seu pai  $p[z]$  é vermelho, e a propriedade 3 é violada apenas entre  $z$  e  $p[z]$ .

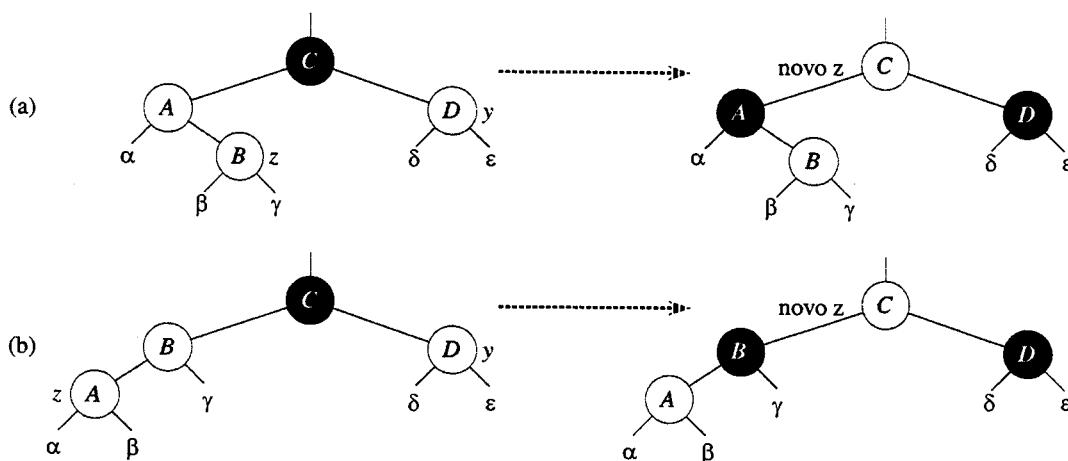
### Caso 1: o tio de $y$ de $z$ é vermelho

A situação para o caso 1 (linhas 5 a 8) é mostrada na Figura 13.5. O caso 1 é executado quando tanto  $p[z]$  quanto  $y$  são vermelhos. Tendo em vista que  $p[p[z]]$  é preto, podemos colorir tanto  $p[z]$  quanto  $y$  de preto, corrigindo assim o problema de  $z$  e  $p[z]$  serem ambos vermelhos, e também colorimos  $p[p[z]]$  de vermelho, mantendo assim a propriedade 5. Em seguida, repetimos o loop **while** com  $p[p[z]]$  como o novo nó  $z$ . O ponteiro  $z$  sobe dois níveis na árvore.

Agora mostramos que o caso 1 mantém o loop invariante no início da próxima iteração. Usamos  $z$  para denotar o nó  $z$  na iteração atual, e  $z' = p[p[z]]$  para denotar o nó  $z$  no teste da linha 1 na iteração seguinte.

- a. Como essa iteração torna a cor de  $p[p[z]]$  vermelha, o nó  $z'$  é vermelho no início da próxima iteração.
- b. O nó  $p[z']$  é  $p[p[p[z]]]$  nessa iteração, e a cor desse nó não se altera. Se esse nó é a raiz, ele era preto antes dessa iteração, e permanece preto no início da próxima iteração.
- c. Já mostramos que o caso 1 mantém a propriedade 5, e é claro que ele não introduz uma violação da propriedade 1 ou 3.

Se o nó  $z'$  é a raiz no início da próxima iteração, então o caso 1 corrigiu a única violação da propriedade 4 nessa iteração. Como  $z'$  é vermelho e é a raiz, a propriedade 2 passa a ser a única violada, e essa violação se deve a  $z'$ .



**FIGURA 13.5** O caso 1 do procedimento RB-INSERT. A propriedade 4 é violada, pois  $z$  e seu pai  $p[z]$  são ambos vermelhos. A mesma ação é adotada se (a)  $z$  é um filho da direita ou (b)  $z$  é um filho da esquerda. Cada uma das subárvore,  $\alpha, \beta, \gamma, \delta$  e  $\epsilon$  tem uma raiz preta, e cada uma tem a mesma altura de preto. O código para o caso 1 altera as cores de alguns nós, preservando a propriedade 5: todos os caminhos descendentes desde um nó até uma folha têm o mesmo número de pretos. O loop `while` continua com o avô  $p[p[z]]$  do nó  $z$  como o novo  $z$ . Qualquer violação da propriedade 4 só pode ocorrer agora entre o novo  $z$ , que é vermelho, e seu pai, que também é vermelho

Se nó  $z'$  não é a raiz no início da próxima iteração, então o caso 1 não criou uma violação da propriedade 2. O caso 1 corrigiu a única violação da propriedade 4 que existia no início dessa iteração. Em seguida, ele tornou  $z'$  vermelho e deixou  $p[z']$  como estava. Se  $p[z']$  era preto, não há nenhuma violação da propriedade 4. Se  $p[z']$  era vermelho, colar  $z'$  de vermelho criou uma violação da propriedade 4 entre  $z'$  e  $p[z']$ .

**Caso 2: o tio  $y$  de  $z$  é preto e  $z$  é um filho da direita**

**Caso 3: o tio  $y$  de  $z$  é preto e  $z$  é um filho da esquerda**

Nos casos 2 e 3, a cor do tio  $y$  de  $z$  é preta. Os dois casos se distinguem pelo fato de  $z$  ser um filho da direita ou da esquerda de  $p[z]$ . As linhas 10 e 11 constituem o caso 2, que é mostrado na Figura 13.6, juntamente com o caso 3. No caso 2, o nó  $z$  é um filho da direita de seu pai. Usamos imediatamente uma rotação à esquerda para transformar a situação no caso 3 (linhas 12 a 14), na qual o nó  $z$  é um filho da esquerda. Como tanto  $z$  quanto  $p[z]$  são vermelhos, a rotação não afeta nem a altura de preto dos nós nem a propriedade 5. Quer entremos no caso 3 diretamente ou através do caso 2, o tio  $y$  de  $z$  será preto, pois, do contrário, teríamos executado o caso 1. Além disso, o nó  $p[p[z]]$  existe, pois demonstramos que esse nó existia no momento em que as linhas 2 e 3 foram executadas e, após  $z$  subir um nível na linha 10 e depois descer um nível na linha 11, a identidade de  $p[p[z]]$  permanece inalterada. No caso 3, executamos algumas mudanças de co-

res e uma rotação à direita, o que preserva a propriedade 5; em seguida, tendo em vista que não temos mais dois nós vermelhos em uma linha, encerramos. O corpo do loop **while** não é executado outra vez, pois agora  $p[z]$  é preto.

Agora, mostramos que os casos 2 e 3 mantêm o loop invariante. (Como acabamos de demonstrar,  $p[z]$  será preto no próximo teste da linha 1, e o corpo do loop não será executado novamente.)

- a.** O caso 2 faz  $z$  apontar para  $p[z]$ , que é vermelho. Nenhuma mudança adicional em  $z$  ou em sua cor ocorre nos casos 2 e 3.
- b.** O caso 3 torna  $p[z]$  preto, de forma que, se  $p[z]$  é a raiz no início da próxima iteração, ele é preto.
- c.** Como ocorre no caso de 1, as propriedades 1, 3 e 5 são mantidas nos casos 2 e 3.

Tendo em vista que o nó  $z$  não é a raiz nos casos 2 e 3, sabemos que não há nenhuma violação da propriedade 2. Os casos 2 e 3 não introduzem uma violação da propriedade 2, pois o único nó que se tornou vermelho passa a ser um filho de um nó preto pela rotação no caso 3.

Os casos 2 e 3 corrigem a única violação da propriedade 4, e eles não introduzem outra violação.

Tendo mostrado que cada iteração do loop mantém o invariante, mostramos que RB-INSERT-FIXUP restaura corretamente as propriedades vermelho-preto.

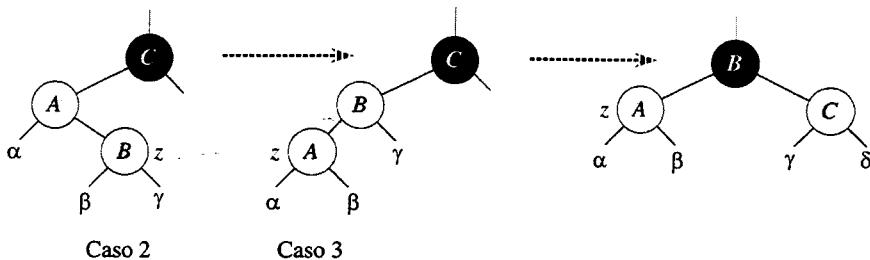


FIGURA 13.6 Os casos 2 e 3 do procedimento RB-INSERT. Como no caso 1, a propriedade 4 é violada no caso 2 ou no caso 3, porque  $z$  e seu pai  $p[z]$  são ambos vermelhos. Cada uma das subárvore,  $\alpha$ ,  $\beta$ ,  $\gamma$  e  $\delta$  tem uma raiz preta ( $\alpha$ ,  $\beta$  e  $\gamma$  pela propriedade 4, e  $\delta$  porque, em caso contrário, estariam no caso 1), e cada uma tem a mesma altura de preto. O caso 2 é transformado no caso 3 por uma rotação à esquerda, o que preserva a propriedade 5: todos os caminhos descendentes de um nó até uma folha têm o mesmo número de pretos. O caso 3 causa algumas mudanças de cores e uma rotação à direita, o que também preserva a propriedade 5. Em seguida, o loop **while** termina, porque a propriedade 4 é satisfeita: não há mais dois nós vermelhos em uma linha.

## Análise

Qual é o tempo de execução de RB-INSERT? Tendo em vista que a altura de uma árvore vermelho-preto sobre  $n$  nós é  $O(\lg n)$ , as linhas 1 a 16 de RB-INSERT levam o tempo  $O(\lg n)$ . Em RB-INSERT-FIXUP, o loop **while** só se repete se o caso 1 é executado, e então o ponteiro  $z$  sobe dois níveis na árvore. O número total de vezes que o loop **while** pode ser executado é portanto  $O(\lg n)$ . Desse modo, RB-INSERT demora um tempo total  $O(\lg n)$ . É interessante observar que ele nunca executa mais de duas rotações, pois o loop **while** termina se o caso 2 ou o caso 3 é executado.

## Exercícios

### 13.3-1

Na linha 16 de RB-INSERT, definimos a cor do nó recém-inserido como vermelho. Note que, se tivéssemos optado por definir a cor de  $z$  como preto, a propriedade 4 de uma árvore vermelho-preto não seria violada. Por que não optamos por definir a cor de  $z$  como preta?

### 13.3-2

Mostre as árvores vermelho-preto que resultam após a inserção bem-sucedida das chaves 41, 38, 31, 12, 19, 8 em uma árvore vermelho-preto inicialmente vazia.

### 13.3-3

Suponha que a altura de preto de cada uma das subárvores  $\alpha, \beta, \gamma, \delta, \varepsilon$  nas Figuras 13.5 e 13.6 seja  $k$ . Identifique cada nó em cada figura com sua altura de preto, a fim de verificar se a propriedade 5 é preservada pela transformação indicada.

### 13.3-4

O professor Teach está preocupado com o fato de RB-INSERT-FIXUP poder definir  $cor[nil[T]]$  como VERMELHO, em cujo caso o teste da linha 1 não faria o loop terminar quando  $z$  fosse a raiz. Mostre que a preocupação do professor é infundada, demonstrando que RB-INSERT-FIXUP nunca define  $cor[nil[T]]$  como VERMELHO.

### 13.3-5

Considere uma árvore vermelho-preto formada pela inserção de  $n$  nós com RB-INSERT. Mostre que, se  $n > 1$ , a árvore tem pelo menos um nó vermelho.

### 13.3-6

Sugira como implementar RB-INSERT de maneira eficiente, se a representação para árvores vermelho-preto não incluir nenhum espaço de armazenamento para ponteiros superiores.

## 13.4 Eliminação

Como as outras operações básicas em uma árvore vermelho-preto de  $n$  nós, a eliminação de um nó demora o tempo  $O(\lg n)$ . A eliminação de um nó de uma árvore vermelho-preto é apenas ligeiramente mais complicada que a inserção de um nó.

O procedimento RB-DELETE é uma modificação secundária do procedimento TREE-DELETE (Seção 12.3). Após extrair um nó, ele chama um procedimento auxiliar RB-DELETE-FIXUP que muda as cores e executa rotações para restaurar as propriedades vermelho-preto.

### RB-DELETE( $T, z$ )

```
1 if esquerda[z] = nil[T] or direita[z] = nil[T]
2   then y ← z
3   else y ← TREE-SUCCESSOR(z)
4 if esquerda[y] ≠ nil[T]
5   then x ← esquerda[y]
6   else x ← direita[y]
7 p[x] ← p[y]
8 if p[y] = nil[T]
9   then raiz[T] ← x
10  else if y = esquerda[p[y]]
11    then esquerda[p[y]] ← x
12    else direita[p[y]] ← x
13  if y ≠ z
14    then chave[z] ← chave[y]
```

```

15      copia dados satélite de  $y$  em  $z$ 
16  if  $cor[y] = \text{PRETO}$ 
17    then RB-DELETE-FIXUP( $T, x$ )
18  return  $y$ 

```

Existem três diferenças entre os procedimentos TREE-DELETE e RB-DELETE. Primeiro, todas as referências a NIL em TREE-DELETE foram substituídas por referências à sentinela  $nil[T]$  em RB-DELETE. Segundo, o teste para determinar se  $x$  é NIL na linha 7 de TREE-DELETE foi removido, e a atribuição  $p[x] \leftarrow p[y]$  é executada de modo incondicional na linha 7 de RB-DELETE. Desse modo, se  $x$  é a sentinela  $nil[T]$ , o ponteiro de seu pai aponta para o pai do nó  $y$  extraído. Terceiro, uma chamada a RB-DELETE-FIXUP é efetuada nas linhas 16 e 17 se  $y$  é preto. Se  $y$  é vermelho, as propriedades vermelho-preto ainda são válidas quando  $y$  é extraído, pelas seguintes razões:

- Nenhuma altura preta na árvore mudou.
- Nenhum nó vermelho se tornou adjacente.
- Como  $y$  não poderia ter sido a raiz se fosse vermelho, a raiz permanece preta.

O nó  $x$  repassado a RB-DELETE-FIXUP é um entre dois nós: ou o nó que era o único filho de  $y$  antes de  $y$  ser extraído, se  $y$  tinha um filho que não era a sentinela  $nil[T]$  ou, se  $y$  não tinha filhos,  $x$  é a sentinela  $nil[T]$ . Nesse último caso, a atribuição incondicional na linha 7 garante que o pai de  $x$  agora é o nó que anteriormente era o pai de  $y$ , quer seja  $x$  um nó interno de transporte de chave ou a sentinela  $nil[T]$ .

Agora podemos examinar o modo como o procedimento RB-DELETE-FIXUP restaura as propriedades vermelho-preto para a árvore de pesquisa.

**RB-DELETE-FIXUP( $T, x$ )**

```

1 while  $x \neq \text{raiz}[T]$  e  $cor[x] = \text{PRETO}$ 
2   do if  $x = \text{esquerda}[p[x]]$ 
3     then  $w \leftarrow \text{direita}[p[x]]$ 
4       if  $cor[w] = \text{VERMELHO}$ 
5         then  $cor[w] \leftarrow \text{PRETO}$            ▷ Caso 1
6            $cor[p[x]] \leftarrow \text{VERMELHO}$        ▷ Caso 1
7           LEFT-ROTATE( $T, p[x]$ )            ▷ Caso 1
8            $w \leftarrow \text{direita}[p[x]]$         ▷ Caso 1
9       if  $cor[\text{esquerda}[w]] = \text{PRETO}$  e  $cor[\text{direita}[w]] = \text{PRETO}$ 
10      then  $cor[w] \leftarrow \text{VERMELHO}$        ▷ Caso 2
11       $x \leftarrow p[x]$                       ▷ Caso 2
12    else if  $cor[\text{direita}[w]] = \text{PRETO}$ 
13      then  $cor[\text{esquerda}[w]] \leftarrow \text{PRETO}$  ▷ Caso 3
14       $cor[w] \leftarrow \text{VERMELHO}$            ▷ Caso 3
15      RIGHT-ROTATE( $T, w$ )                 ▷ Caso 3
16       $w \leftarrow \text{direita}[p[x]]$         ▷ Caso 3
17       $cor[w] \leftarrow cor[p[x]]$            ▷ Caso 4
18       $cor[p[x]] \leftarrow \text{PRETO}$         ▷ Caso 4
19       $cor[\text{direita}[w]] \leftarrow \text{PRETO}$  ▷ Caso 4
20      LEFT-ROTATE( $T, p[x]$ )            ▷ Caso 4
21       $x \leftarrow \text{raiz}[T]$               ▷ Caso 4
22    else (igual à cláusula then com "direita" e "esquerda" trocadas)
23   $cor[x] \leftarrow \text{PRETO}$ 

```

Se o nó  $y$  extraído em RB-DELETE é preto, três problemas podem surgir. Primeiro, se  $y$  era a raiz e um filho vermelho de  $y$  se torna a nova raiz, violamos a propriedade 2. Segundo, se tanto  $x$  quanto  $p[y]$  (que agora também é  $p[x]$ ) eram vermelhos, então violamos a propriedade 4. Terceiro, a remoção de  $y$  faz qualquer caminho que continha  $y$  anteriormente ter um nó preto a menos. Desse modo, a propriedade 5 agora é violada por qualquer ancestral de  $y$  na árvore. Podemos corrigir esse problema dizendo que o nó  $x$  tem um preto “extra”. Isto é, se adicionarmos 1 à contagem de nós pretos em qualquer caminho que contenha  $x$  então, sob essa interpretação, a propriedade 5 se mantém válida. Quando extraímos o nó preto  $y$ , “empurramos” sua característica de preto sobre seu filho. O problema é que agora o nó  $x$  não é vermelho nem preto, violando assim a propriedade 1. Em vez disso, o nó  $x$  é “duplamente preto” ou “vermelho e preto” e contribui com 2 ou 1, respectivamente, para a contagem de nós pretos em caminhos que contêm  $x$ . O atributo *cor* de  $x$  ainda será VERMELHO (se  $x$  é vermelho e preto) ou PRETO (se  $x$  é duplamente preto). Em outras palavras, o preto extra em um nó é refletido no fato de  $x$  apontar para o nó, e não no atributo *cor*.

O procedimento RB-DELETE-FIXUP restaura as propriedades 1, 2 e 4. Os Exercícios 13.4-1 e 13.4-2 lhe pedem para mostrar que o procedimento restaura as propriedades 2 e 4 e assim, no restante desta seção, nos concentraremos na propriedade 1. O objetivo do loop **while** nas linhas 1 a 22 é mover o preto extra para cima na árvore até

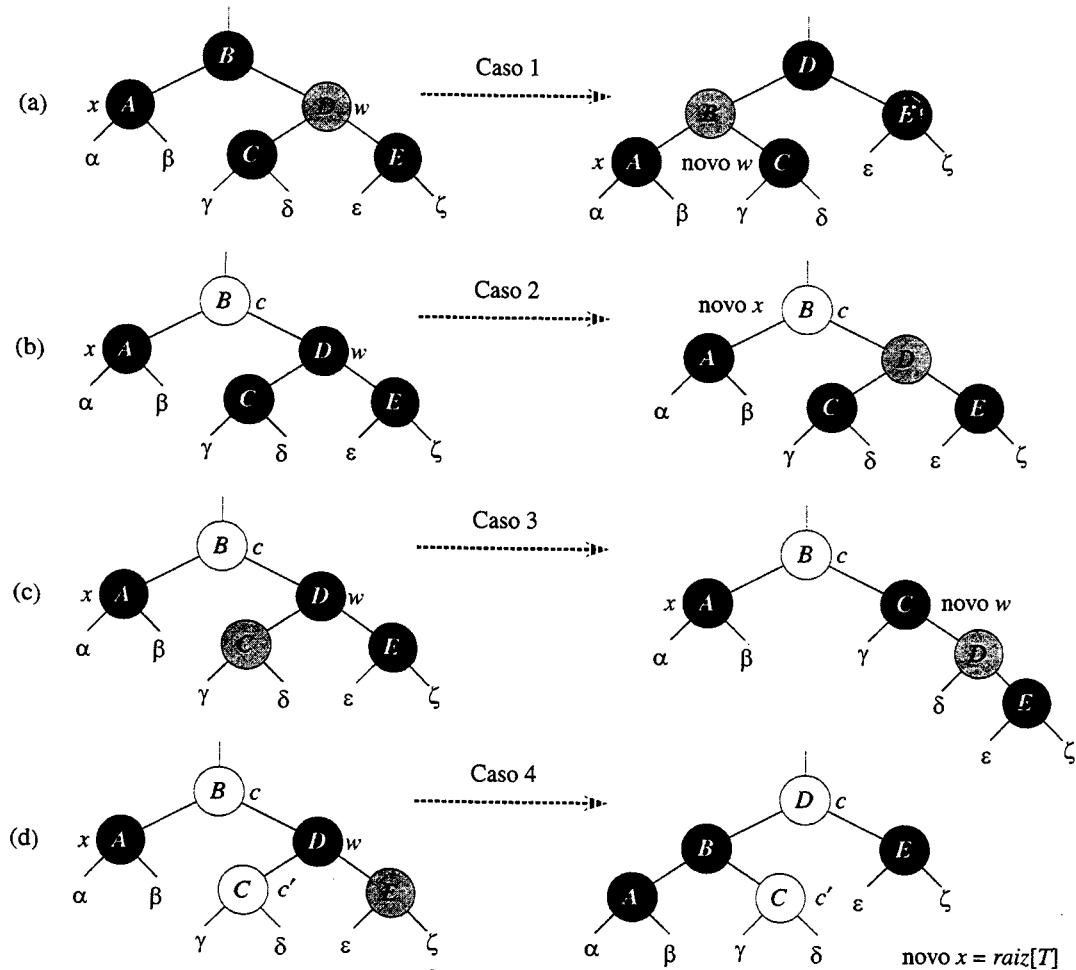
1.  $x$  apontar para um nó vermelho e preto, em cujo caso iremos colorir  $x$  (unicamente) de preto na linha 23,
2.  $x$  apontar para a raiz, e nesse caso o preto extra pode ser simplesmente “removido”, ou
3. podem ser executadas operações adequadas de rotação e de nova coloração.

Dentro do loop **while**,  $x$  sempre aponta para um nó duplamente preto não raiz. Determinamos na linha 2 se  $x$  é um filho da esquerda ou um filho da direita de seu pai  $p[x]$ . (Fornece-mos o código para a situação na qual  $x$  é um filho da esquerda; a situação na qual  $x$  é um filho da direita – linha 22 – é simétrica.) Mantemos um ponteiro  $w$  para o irmão de  $x$ . Como o nó  $x$  é duplamente preto, o nó  $w$  não pode ser *nil*[ $T$ ]; caso contrário, o número de pretos no caminho desde  $p[x]$  até a folha  $w$  (unicamente preta) seria menor que o número no caminho de  $p[x]$  até  $x$ .

Os quatro casos<sup>2</sup> no código estão ilustrados na Figura 13.7. Antes de examinar cada caso em detalhes, vamos dar uma olhada mais geral na maneira como podemos verificar se a transformação em cada um dos casos preserva a propriedade 5. A idéia chave é que em cada caso o número de nós pretos (incluindo o preto extra de  $x$ ) desde (e inclusive) a raiz da subárvore mostrada até cada uma das subárvores  $\alpha, \beta, \dots, \zeta$  é preservado pela transformação. Desse modo, se a propriedade 5 é válida antes da transformação, ela continua a ser válida daí em diante. Por exemplo, na Figura 13.7(a), que ilustra o caso 1, o número de nós pretos desde a raiz até a subárvore  $\alpha$  ou  $\beta$  é 3, tanto antes quanto após a transformação. (Mais uma vez, lembre-se de que o nó  $x$  adiciona um preto extra.) De modo semelhante, o número de nós pretos desde a raiz até qualquer das subárvores  $\gamma, \delta, \varepsilon$  e  $\zeta$  é 2, tanto antes quanto depois da transformação. Na Figura 13.7(b), a contagem deve envolver o valor  $c$  do atributo *cor* da raiz da subárvore mostrada, que pode ser VERMELHO ou PRETO. Se definirmos contador(VERMELHO) = 0 e contador(PRETO) = 1, então o número de nós pretos desde a raiz até  $\alpha$  é  $2 + \text{contador}(c)$ , tanto antes quanto após a transformação. Nesse caso, depois da transformação, o novo nó  $x$  tem o atributo *cor* igual a  $c$ , mas esse nó é realmente vermelho e preto (se  $c$  = VERMELHO) ou duplamente preto (se  $c$  = PRETO). Os outros casos podem ser verificados de maneira semelhante (ver Exercício 13.4-5.)

---

<sup>2</sup> Como em RB-INSERT-FIXUP, os casos em RB-DELETE-FIXUP não são mutuamente exclusivos.



**FIGURA 13.7** Os casos no loop `while` do procedimento RB-DELETE-FIXUP. Os nós escurecidos têm atributos *cor* PRETO, nós fortemente sombreados têm atributos *cor* VERMELHO e nós ligeiramente sombreados têm atributos *cor* representados por *c* e *c'*, que podem ser VERMELHO ou PRETO. As letras  $\alpha, \beta, \dots, \zeta$  representam subárvore arbitrárias. Em cada caso, a configuração da esquerda é transformada na configuração da direita pela mudança de algumas cores e/ou execução de uma rotação. Qualquer nó indicado por *x* tem um preto extra e é duplamente preto ou vermelho e preto. O único caso que faz o loop se repetir é o caso 2. (a) O caso 1 é transformado no caso 2, 3 ou 4 pela troca das cores dos nós *B* e *D* e pela execução de uma rotação à esquerda. (b) No caso 2, o preto extra representado pelo ponteiro *x* é movido para cima na árvore, colorindo-se o nó *D* de vermelho e definindo-se *x* de modo a apontar para o nó *B*. Se entrarmos no caso 2 através do caso 1, o loop `while` terminará, pois o novo nó *x* é vermelho e preto, e portanto o valor *c* de seu atributo *cor* é VERMELHO. (c) O caso 3 é transformado no caso 4 pela troca das cores dos nós *C* e *D* e pela execução de uma rotação à direita. (d) No caso 4, o preto extra representado por *x* pode ser removido mudando-se algumas cores e executando-se uma rotação à esquerda (sem violar as propriedades vermelho-preto), e o loop se encerra.

### Caso 1: o irmão *w* de *x* é vermelho

O caso 1 (linhas 5 a 8 de RB-DELETE-FIXUP e Figura 13.7(a)) ocorre quando o nó *w*, o irmão do nó *x*, é vermelho. Tendo em vista que *w* deve ter filhos pretos, podemos trocar as cores de *w* e *p[x]*, e depois executar uma rotação à esquerda sobre *p[x]* sem violar qualquer das propriedades vermelho-preto. O novo irmão de *x*, um dos filhos de *w* antes da rotação, agora é preto e, desse modo, convertemos o caso 1 no caso 2, 3 ou 4.

Os casos 2, 3 e 4 ocorrem quando o nó *w* é preto; eles se distinguem pelas cores dos filhos de *w*.

### Caso 2: o irmão $w$ de $x$ é preto, e ambos os filhos de $w$ são pretos

No caso 2 (linhas 10 e 11 de RB-DELETE-FIXUP e Figura 13.7(b)), ambos os filhos de  $w$  são pretos. Tendo em vista que  $w$  também é preto, retiramos um preto de  $x$  e de  $w$ , deixando  $x$  com apenas um preto e deixando  $w$  vermelho. Para compensar, a remoção de um preto de  $x$  e de  $w$ , seria interessante adicionar um preto extra a  $p[x]$ , que era originalmente vermelho ou preto. Fazemos isso repetindo o loop `while` com  $p[x]$  como o novo nó  $x$ . Observe que, se entrarmos no caso 2 através do caso 1, o novo nó  $x$  será vermelho e preto, pois o  $p[x]$  original era vermelho. Consequentemente, o valor  $c$  do atributo `cor` do novo nó  $x$  é VERMELHO, e o loop termina quando testa a condição de loop. O novo nó  $x$  é então colorido (unicamente) de preto na linha 23.

### Caso 3: o irmão $w$ de $x$ é preto, o filho da esquerda de $w$ é vermelho e o filho da direita de $w$ é preto

O caso 3 (linhas 13 a 16 e Figura 13.7(c)) ocorre quando  $w$  é preto, seu filho da esquerda é vermelho e seu filho da direita é preto. Podemos alternar as cores de  $w$  e de seu filho da esquerda  $esquerda[w]$ , e depois executar uma rotação à direita sobre  $w$  sem violar qualquer das propriedades vermelho-preto. O novo irmão  $w$  de  $x$  é agora um nó preto com um filho da direita vermelho, e desse modo transformamos o caso 3 no caso 4.

### Caso 4: o irmão $w$ de $x$ é preto, e o filho da direita de $w$ é vermelho

O caso 4 (linhas 17 a 21 e Figura 13.7(d)) ocorre quando o irmão  $w$  do nó  $x$  é preto, e o filho da direita de  $w$  é vermelho. Fazendo algumas mudanças de cores e executando uma rotação à esquerda sobre  $p[x]$ , podemos remover o preto extra em  $x$ , tornando-o unicamente preto, sem violar qualquer das propriedades vermelho-preto. A definição de  $x$  como a raiz faz o loop `while` se encerrar ao testar a condição de loop .

## Análise

Qual é o tempo de execução de RB-DELETE? Tendo em vista que a altura de uma árvore vermelho-preto de  $n$  nós é  $O(\lg n)$ , o custo total do procedimento sem a chamada a RB-DELETE-FIXUP demora o tempo  $O(\lg n)$ . Dentro de RB-DELETE-FIXUP, cada um dos casos 1, 3 e 4 termina depois de executar um número constante de mudanças de cores e no máximo três rotações. O caso 2 é o único caso em que o loop `while` pode ser repetido, e então o ponteiro  $x$  se move para cima na árvore no máximo  $O(\lg n)$  vezes e nenhuma rotação é executada. Desse modo, o procedimento RB-DELETE-FIXUP demora o tempo  $O(\lg n)$  e executa no máximo três rotações, e assim o tempo global para RB-DELETE também é  $O(\lg n)$ .

## Exercícios

### 13.4-1

Mostre que, após a execução de RB-DELETE-FIXUP, a raiz da árvore tem de ser preta.

### 13.4-2

Mostre que, se  $x$  e  $p[y]$  são ambos vermelhos em RB-DELETE, então a propriedade 4 é restabelecida pela chamada RB-DELETE-FIXUP( $T, x$ ).

### 13.4-3

No Exercício 13.3-2, você encontrou a árvore vermelho-preto que resulta da inserção sucessiva das chaves 41, 38, 31, 12, 19, 8 em uma árvore inicialmente vazia. Agora, mostre as árvores vermelho-preto que resultam da eliminação sucessiva das chaves na ordem 8, 12, 19, 31, 38, 41.

### 13.4-4

Em quais linhas do código de RB-DELETE-FIXUP poderíamos examinar ou modificar a sentinela  $nil[T]$ ?

### 13.4-5

Em cada um dos casos da Figura 13.7, forneça a contagem de nós pretos desde a raiz da subárvore mostrada até cada uma das subárvores  $\alpha, \beta, \dots, \zeta$ , e confirme que cada contagem permanece a mesma depois da transformação. Quando um nó tiver um atributo  $cor\ c$  ou  $c'$ , use a notação contador( $c$ ) ou contador( $c'$ ) simbolicamente em sua contagem.

### 13.4-6

Os professores Skelton e Baron estão preocupados com o fato de que, no início do caso 1 de RB-DELETE-FIXUP, o nó  $p[x]$  não poderia ser preto. Se os professores estão corretos, então as linhas 5 e 6 estão erradas. Mostre que  $p[x]$  deve ser preto no início do caso 1, e portanto os professores não têm nenhuma razão para se preocupar.

### 13.4-7

Suponha que um nó  $x$  seja inserido em uma árvore vermelho-preto com RB-INSERT e depois imediatamente eliminado com RB-DELETE. A árvore vermelho-preto resultante é igual à árvore vermelho-preto inicial? Justifique sua resposta.

## Problemas

### 13.1 Conjuntos dinâmicos persistentes

Durante o curso de um algoritmo, às vezes achamos que precisamos manter versões anteriores de um conjunto dinâmico à medida que ele é atualizado. Tal conjunto é chamado **persistente**. Um modo de implementar um conjunto persistente é copiar o conjunto inteiro sempre que ele é modificado, mas essa abordagem pode reduzir a velocidade de um programa e também consumir muito espaço. Às vezes, podemos fazer muito melhor que isso.

Considere um conjunto persistente  $S$  com as operações INSERT, DELETE e SEARCH, que implementamos usando árvores de pesquisa binária como mostra a Figura 13.8(a). Mantemos uma raiz separada para cada versão do conjunto. Para inserir a chave 5 no conjunto, criamos um novo nó com chave 5. Esse nó se torna o filho da esquerda de um novo nó com chave 7, pois não podemos modificar o nó existente com chave 7. De modo semelhante, o novo nó com chave 7 se torna o filho da esquerda de um novo nó com chave 8, cujo filho da direita é o nó existente com chave 10. O novo nó com chave 8 se torna, por sua vez, o filho da direita de uma nova raiz  $r'$  com chave 4, cujo filho da esquerda é o nó existente com chave 3. Desse modo, copiamos apenas parte da árvore e compartilhamos alguns dos nós com a árvore original, como mostra a Figura 13.8(b).

Suponha que cada nó da árvore tenha os campos *chave*, *esquerda* e *direita*, mas nenhum campo pai. (Consulte também o Exercício 13.3-6.)

- a. No caso geral de uma árvore de pesquisa binária persistente, identifique os nós que precisam ser alterados para se inserir uma chave  $k$  ou eliminar um nó  $y$ .
- b. Escreva um procedimento PERSISTENT-TREE-INSERT que, dada uma árvore persistente  $T$  e uma chave  $k$  a ser inserida, retorne uma nova árvore persistente  $T'$  que seja o resultado da inserção de  $k$  em  $T$ .
- c. Se a altura da árvore de pesquisa binária persistente  $T$  é  $b$ , quais são os requisitos de tempo e espaço da sua implementação de PERSISTENT-TREE-INSERT? (O requisito de espaço é proporcional ao número de novos nós alocados.)
- d. Suponha que tivéssemos incluído o campo pai em cada nó. Nesse caso, PERSISTENT-TREE-INSERT precisaria executar uma operação de cópia adicional. Prove que PERSISTENT-TREE-INSERT exigiria então o tempo e o espaço  $\Omega(n)$ , onde  $n$  é o número de nós na árvore.
- e. Mostre como usar árvores vermelho-preto para garantir que o tempo de execução do pior caso e o espaço são  $O(\lg n)$  por inserção ou eliminação.

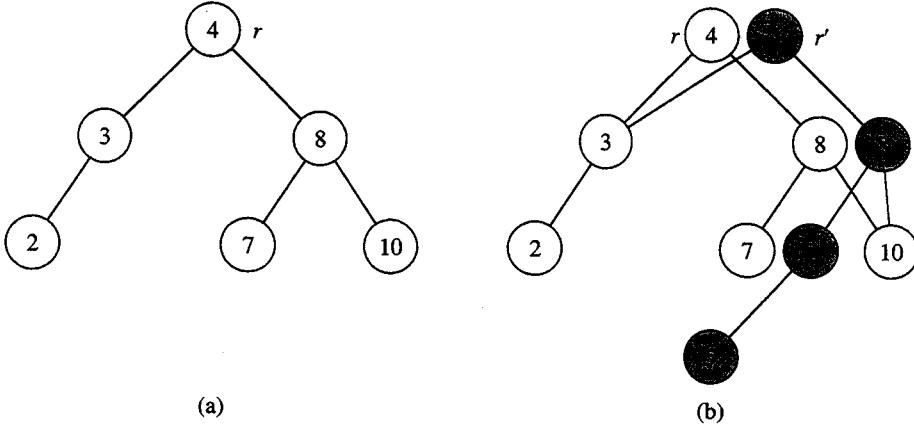


FIGURA 13.8 (a) Uma árvore de pesquisa binária com chaves 2, 3, 4, 7, 8, 10. (b) A árvore de pesquisa binária persistente que resulta da inserção da chave 5. A versão mais recente do conjunto consiste nos nós acessíveis a partir da raiz  $r'$ , e a versão anterior consiste nos nós acessíveis a partir de  $r$ . Os nós fortemente sombreados são adicionados quando a chave 5 é inserida

### 13-2 Operação de junção sobre árvores vermelho-preto

A operação de **junção** utiliza dois conjuntos dinâmicos  $S_1$  e  $S_2$  e um elemento  $x$  tal que, para qualquer  $x_1 \in S_1$  e  $x_2 \in S_2$ , temos  $\text{chave}[x_1] \leq \text{chave}[x] \leq \text{chave}[x_2]$ . Ela retorna um conjunto  $S = S_1 \cup \{x\} \cup S_2$ . Neste problema, investigamos como implementar a operação de junção sobre árvores vermelho-preto.

- Dada uma árvore vermelho-preto  $T$ , armazenamos sua altura de preto como o campo  $bb[T]$ . Mostre que esse campo pode ser mantido por RB-INSERT e RB-DELETE sem exigir espaço de armazenamento extra na árvore e sem aumentar os tempos de execução assintóticos. Mostre que, enquanto descemos em  $T$ , podemos determinar a altura de preto de cada nó que visitamos no tempo  $O(1)$  por nó visitado.  
Desejamos implementar a operação RB-JOIN( $T_1, x, T_2$ ), o que destrói  $T_1$  e  $T_2$  e retorna uma árvore vermelho-preto  $T = T_1 \cup \{x\} \cup T_2$ . Seja  $n$  o número total de nós em  $T_1$  e  $T_2$ .
  - Suponha que  $bb[T_1] \geq bb[T_2]$ . Descreva um algoritmo de tempo  $O(\lg n)$  que encontre um nó preto  $y$  em  $T_1$  com a maior chave entre os nós cuja altura de preto é  $bb[T_2]$ .
  - Seja  $T_y$  a subárvore com raiz em  $y$ . Descreva de que modo  $T_y$  pode ser substituído por  $T_y \cup \{x\} \cup T_2$  no tempo  $O(1)$  sem destruir a propriedade de árvore de pesquisa binária.
  - Que cor devemos usar em  $x$  para que as propriedades vermelho-preto 1, 3 e 5 sejam mantidas? Descreva de que maneira as propriedades 2 e 4 podem ser impostas no tempo  $O(\lg n)$ .
  - Demonstre que nenhuma generalidade é perdida tomando-se a hipótese da parte (b). Descreva a situação simétrica que surge quando  $bb[T_1] \leq bb[T_2]$ .
  - Mostre que o tempo de execução de RB-JOIN é  $O(\lg n)$ .

### 13-3 Árvores AVL

Uma **árvore AVL** é uma árvore de pesquisa binária que é de **altura balanceada**: Para cada nó  $x$ , as alturas das subárvores esquerda e direita de  $x$  diferem por no máximo 1. Para implementar uma árvore AVL, mantemos um campo extra em cada nó:  $b[x]$  é a altura de nó  $x$ . Como ocorre no caso de qualquer outra árvore de pesquisa binária  $T$ , supomos que  $raiz[T]$  aponta para o nó raiz.

- Prove que uma árvore AVL com  $n$  nós tem altura  $O(\lg n)$ . (Sugestão: Prove que, em uma árvore AVL de altura  $b$ , existem pelo menos  $F_b$  nós, onde  $F_b$  é o  $b$ -ésimo número de Fibonacci.)

- b.** No caso da inserção em uma árvore AVL, primeiro um nó é inserido no lugar apropriado na ordem da árvore de pesquisa binária. Depois dessa inserção, a árvore pode não ser mais de altura balanceada. Especificamente, as alturas dos filhos da esquerda e da direita de algum nó podem diferir por 2. Descreva um procedimento  $BALANCE(x)$ , que toma uma subárvore com raiz em  $x$  cujos filhos da esquerda e da direita são de altura balanceada e têm alturas que diferem por no máximo 2, isto é,  $|b[direita[x]] - b[esquerda[x]]| \leq 2$ , e altera a subárvore com raiz em  $x$  para ser de altura balanceada. (Sugestão: Use rotações.)
- c.** Usando a parte (b), descreva um procedimento recursivo  $AVL-INSERT(x, z)$ , que toma um nó  $x$  dentro de uma árvore AVL e um nó  $z$  recentemente criado (cuja chave já foi preenchida) e adiciona  $z$  à subárvore com raiz em  $x$ , mantendo a propriedade de que  $x$  é a raiz de uma árvore AVL. Como no procedimento  $TREE-INSERT$  da Seção 12.3, suponha que  $chave[z]$  já foi preenchida e que  $esquerda[z] = NIL$  e  $direita[z] = NIL$ ; suponha também que  $b[z] = 0$ . Desse modo, para inserir o nó  $z$  na árvore AVL  $T$ , chamamos  $AVL-INSERT(raiz[T], z)$ .
- d.** Dê um exemplo de uma árvore AVL de  $n$  nós, na qual uma operação  $AVL-INSERT$  resulta na execução de  $\Omega(\lg n)$  rotações.

#### 13-4 Treaps

Se inserirmos um conjunto de  $n$  itens em uma árvore de pesquisa binária, a árvore resultante pode ficar terrivelmente desbalanceada, levando a tempos de pesquisa longos. Porém, como vimos na Seção 12.4, árvores de pesquisa binária construídas aleatoriamente tendem a ser平衡adas. Portanto, uma estratégia que em média constrói uma árvore balanceada para um conjunto fixo de itens é permitir os itens ao acaso e depois inseri-los nessa ordem na árvore.

E se não tivermos todos os itens ao mesmo tempo? Se recebermos os itens um de cada vez, ainda poderemos construir aleatoriamente uma árvore de pesquisa binária a partir deles? Examinaremos uma estrutura de dados que responde de forma afirmativa a essa pergunta. Um *treap* é uma árvore de pesquisa binária com uma forma modificada de ordenar os nós. A Figura 13.9 mostra um exemplo. Como de hábito, cada nó  $x$  na árvore tem um valor de chave  $chave[x]$ . Além disso, atribuímos  $prioridade[x]$ , que é um número aleatório escolhido independentemente para cada nó. Supomos que todas as prioridades são distintas e também que todas as chaves são distintas. Os nós do treap são ordenados de forma que as chaves obedecem à propriedade de árvore de pesquisa binária, e que as prioridades obedecem à propriedade de ordem de heap mínimo:

- Se  $v$  é um filho da esquerda de  $u$ , então  $chave[v] < chave[u]$ .
- Se  $v$  é um filho da direita de  $u$ , então  $chave[v] > chave[u]$ .
- Se  $v$  é um filho de  $u$ , então  $prioridade[v] > prioridade[u]$ .

(Essa combinação de propriedades é o motivo pelo qual a árvore é chamada um “treap”; ela tem características de uma árvore – tree, em inglês – de pesquisa binária e de um heap.)

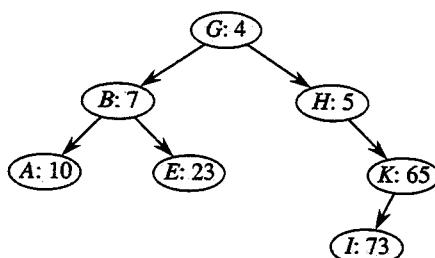


FIGURA 13.9 Um treap. Cada nó  $x$  é identificado com  $chave[x]:prioridade[x]$ . Por exemplo, a raiz tem chave  $G$  e prioridade 4

É conveniente pensar em treaps do modo a seguir. Suponha que inserimos nós  $x_1, x_2, \dots, x_n$ , com chaves associadas, em um treap. Então, o treap resultante é a árvore que teria sido formada se os nós fossem inseridos em uma árvore de pesquisa binária normal na ordem dada por suas prioridades (escolhidas ao acaso), isto é,  $\text{prioridade}[x_i] < \text{prioridade}[x_j]$  significa que  $x_i$  foi inserido antes de  $x_j$ .

- Mostre que, dado um conjunto de nós  $x_1, x_2, \dots, x_n$ , com chaves e prioridades associadas (todas distintas), existe um único treap associado a esses nós.
- Mostre que a altura esperada de um treap é  $\Theta(\lg n)$  e, consequentemente, o tempo para procurar um valor no treap é  $\Theta(\lg n)$ .
- Explique como TREAP-INSERT funciona. Explique a idéia em linguagem comum e forneça o pseudocódigo. (*Sugestão:* Execute o procedimento habitual de inserção em árvore de pesquisa binária e depois execute rotações para restaurar a propriedade de ordem de heap mínimo.)
- Mostre que o tempo de execução esperado de TREAP-INSERT é  $\Theta(\lg n)$ .

TREAP-INSERT executa uma pesquisa e depois uma seqüência de rotações. Embora essas duas operações tenham o mesmo tempo de execução esperado, elas têm custos diferentes na prática. Uma pesquisa lê informações do treap sem modificá-lo. Em contraste, uma rotação altera os ponteiros pai e filho dentro do treap. Na maioria dos computadores, as operações de leitura são muito mais rápidas que as operações de gravação. Desse modo, seria interessante que TREAP-INSERT executasse poucas rotações. Mostraremos que o número esperado de rotações executadas é limitado por uma constante.

Para fazê-lo, precisaremos de algumas definições, que estão ilustradas na Figura 13.11. A **espinha esquerda** de uma árvore de pesquisa binária  $T$  é o caminho desde a raiz até o nó com a menor chave. Em outras palavras, a espinha esquerda é o caminho desde a raiz que consiste apenas em arestas da esquerda. Simetricamente, a **espinha direita** de  $T$  é o caminho desde a raiz que consiste somente em arestas da direita. O **comprimento** de uma espinha é o número de nós que ela contém.

- Considere que o treap  $T$  imediatamente após  $x$  é inserido com o uso de TREAP-INSERT. Seja  $C$  o comprimento da espinha direita da subárvore esquerda de  $x$ . Seja  $D$  o comprimento da espinha esquerda da subárvore direita de  $x$ . Prove que o número total de rotações que foram executadas durante a inserção de  $x$  é igual a  $C + D$ .

Agora calcularemos os valores esperados de  $C$  e  $D$ . Sem perda de generalidade, supomos que as chaves são  $1, 2, \dots, n$ , pois estamos comparando essas chaves apenas entre si.

Para os nós  $x$  e  $y$ , onde  $y \neq x$ , seja  $k = \text{chave}[x]$  e  $i = \text{chave}[y]$ . Definimos variáveis indicadoras aleatórias

$$X_{i,k} = I \{y \text{ está na espinha direita da subárvore esquerda de } x \text{ (em } T\}$$

- Mostre que  $X_{i,k} = 1$  se e somente se  $\text{prioridade}[y] > \text{prioridade}[x]$ ,  $\text{chave}[y] < \text{chave}[x]$  e, para todo  $z$  tal que  $\text{chave}[y] < \text{chave}[z] < \text{chave}[x]$ , temos  $\text{prioridade}[y] < \text{prioridade}[z]$ .
- Mostre que

$$\Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

- Mostre que

$$E[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}.$$

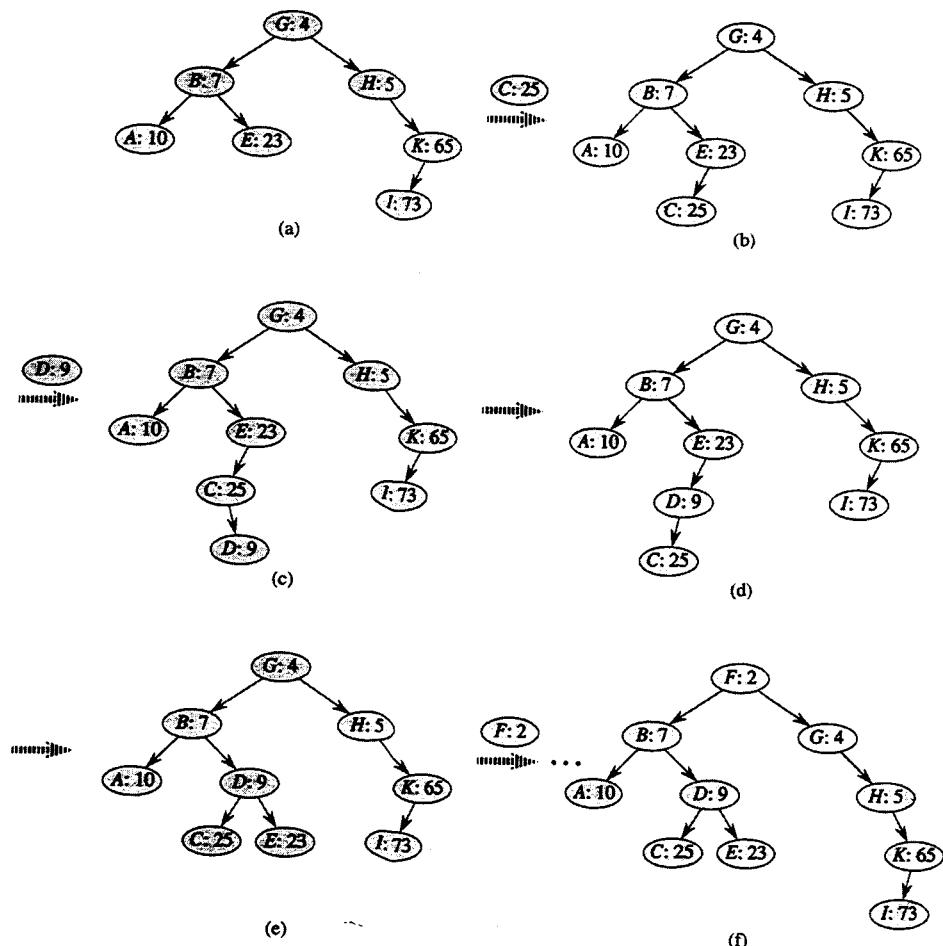


FIGURA 13.10 A operação de TREAP-INSERT. (a) O treap original, antes da inserção. (b) O treap depois da inserção de um nó com chave  $C$  e prioridade 25. (c)–(d) As fases intermediárias quando se insere um nó com chave  $D$  e prioridade 9. (e) O treap depois de terminada a inserção das partes (c) e (d). (f) O treap depois da inserção de um nó com chave  $F$  e prioridade 2

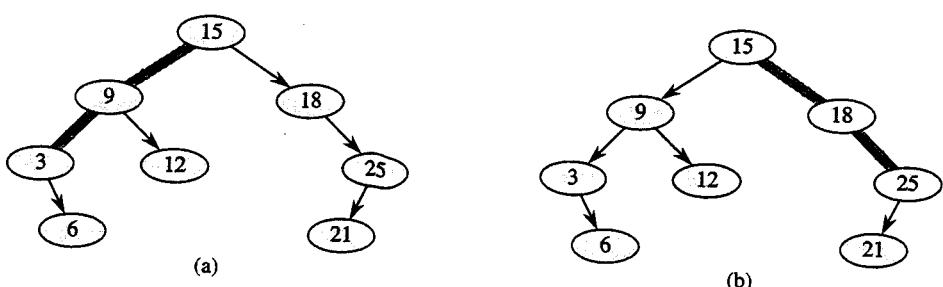


FIGURA 13.11 Espinhas de uma árvore de pesquisa binária. A espinha esquerda está sombreada em (a), e a espinha direita está sombreada em (b)

- i. Use um argumento de simetria para mostrar que

$$E[D] = 1 - \frac{1}{n-k+1}$$

- j. Conclua que o número esperado de rotações executadas quando se insere um nó em um treap é menor que 2.

## Notas do capítulo

A idéia de balancear uma árvore de pesquisa se deve a Adel'son-Vel'skii e Landis [2], que apresentaram em 1962 uma classe de árvores de pesquisa balanceadas chamada “árvores AVL”, descritas no Problema 13-3. Outra classe de árvores de pesquisa, chamadas “árvores 2-3”, foi apresentada por J. E. Hopcroft em 1970. Em uma árvore 2-3, o equilíbrio é mantido pela manipulação dos graus de nós na árvore. Uma generalização de árvores 2-3 apresentada por Bayer e McCreight [32], chamadas árvores B, é o tópico do Capítulo 18.

As árvores vermelho-preto foram criadas por Bayer [31] sob o nome “árvores B binárias simétricas”. Guibas e Sedgewick [135] estudaram profundamente suas propriedades e introduziram a convenção de cores vermelho/preto. Andersson [15] fornece uma variação de árvores vermelho-preto mais simples de codificar. Weiss [311] chama essa variação de árvores AA. Uma árvore AA é semelhante a uma árvore vermelho-preto, exceto pelo fato de que os filhos da esquerda nunca podem ser vermelhos.

Os treaps foram propostos por Seidel e Aragon [271]. Eles são a implementação padrão de um dicionário em Leda, uma coleção bem implementada de estruturas de dados e algoritmos.

Existem muitas outras variações sobre árvores binárias平衡adas, inclusive as árvores de peso balanceado [230], as árvores de  $k$  vizinhos [213] e as árvores de bode expiatório [108]. Talvez as mais curiosas sejam as “árvores oblíquas” introduzidas por Sleator e Tarjan [281], que são “auto-ajustáveis”. (Uma boa descrição de árvores oblíquas é dada por Tarjan [292].) As árvores oblíquas mantêm o equilíbrio sem qualquer condição explícita de equilíbrio, como cor. Em vez disso, “operações oblíquas” (que envolvem rotações) são executadas dentro da árvore toda vez que se efetua um acesso. O custo amortizado (consulte o Capítulo 17) de cada operação sobre uma árvore de  $n$  nós é  $O(\lg n)$ .

As listas de saltos [251] constituem uma alternativa às árvores binárias平衡adas. Uma lista de saltos é uma lista ligada que é ampliada com uma série de ponteiros adicionais. Cada operação de dicionário é executada no tempo esperado  $O(\lg n)$  em uma lista de saltos de  $n$  itens.

---

## *Capítulo 14*

# *Ampliando estruturas de dados*

Existem algumas situações de engenharia que não exigem mais que uma estrutura de dados “de livro didático” – como uma lista duplamente ligada, uma tabela hash ou uma árvore de pesquisa binária – mas muitas outras exigem uma grande dose de criatividade. Entretanto, apenas em raros momentos você precisará criar um tipo completamente novo de estrutura de dados. Com maior frequência, será suficiente ampliar uma estrutura de dados comum, armazenando nela informações adicionais. Você poderá então programar novas operações para que a estrutura de dados forneça suporte à aplicação desejada. Porém, ampliar uma estrutura de dados nem sempre é uma operação direta, pois as informações adicionadas devem ser atualizadas e mantidas pelas operações comuns sobre a estrutura de dados.

Este capítulo discute duas estruturas de dados que são construídas ampliando-se as árvores vermelho-preto. A Seção 14.1 descreve uma estrutura de dados que aceita operações gerais de estatísticas de ordem sobre um conjunto dinâmico. Então, poderemos encontrar rapidamente o  $i$ -ésimo menor número em um conjunto ou a ordem de um dado elemento na ordenação total do conjunto. A Seção 14.2 resume o processo de ampliar uma estrutura de dados e fornece um teorema que pode simplificar a ampliação de árvores vermelho-preto. A Seção 14.3 utiliza esse teorema para ajudar a projetar uma estrutura de dados com o objetivo de manter um conjunto dinâmico de intervalos, como intervalos de tempo. Dado um intervalo de consultas, poderemos então encontrar com rapidez um intervalo no conjunto que se sobreponha a ele.

### **14.1 Estatísticas de ordem dinâmicas**

O Capítulo 9 introduziu a noção de estatística de ordem. Especificamente, a  $i$ -ésima estatística de ordem de um conjunto de  $n$  elementos, onde  $i \in \{1, 2, \dots, n\}$ , é simplesmente o elemento no conjunto com a  $i$ -ésima menor chave. Vimos que qualquer estatística de ordem podia ser recuperada no tempo  $O(n)$  a partir de um conjunto não ordenado. Nesta seção, veremos como as árvores vermelho-preto podem ser modificadas de tal forma que qualquer estatística de ordem possa ser determinada no tempo  $O(\lg n)$ . Veremos também como a **ordem** de um elemento – sua posição na ordem linear do conjunto – pode ser igualmente determinada no tempo  $O(\lg n)$ .

Uma estrutura de dados que pode fornecer suporte a operações rápidas de estatísticas de ordem é mostrada na Figura 14.1. Uma **árvore de estatísticas de ordem**  $T$  é simplesmente uma árvore vermelho-preto com informações adicionais armazenadas em cada nó. Além dos campos habituais da árvore vermelho-preto,  $chave[x]$ ,  $cor[x]$ ,  $p[x]$ ,  $esquerda[x]$  e  $direita[x]$  em um nó  $x$ , temos outro campo  $tamanho[x]$ . Esse campo contém o número de nós (inter-

nos) na subárvore com raiz em  $x$  (inclusive o próprio  $x$ ), isto é, o tamanho da subárvore. Se definirmos o tamanho da sentinela como 0, isto é, se definirmos  $tamanho[nil[T]]$  como 0, então teremos a identidade

$$tamanho[x] = tamanho[esquerda[x]] + tamanho[direita[x]] + 1.$$

Não exigimos que as chaves sejam distintas em uma árvore de estatísticas de ordem. (Por exemplo, a árvore da Figura 14.1 tem duas chaves com valor 14 e duas chaves com valor 21.) Na presença de chaves iguais, a noção anterior de ordem não é bem definida. Removemos essa ambigüidade para uma árvore de estatísticas de ordem definindo a ordem de um elemento como a posição na qual ele seria impresso em um percurso em ordem da árvore. Por exemplo, na Figura 14.1, a chave 14 armazenada em um nó preto tem ordem 5, e a chave 14 armazenada em um nó vermelho tem ordem 6.

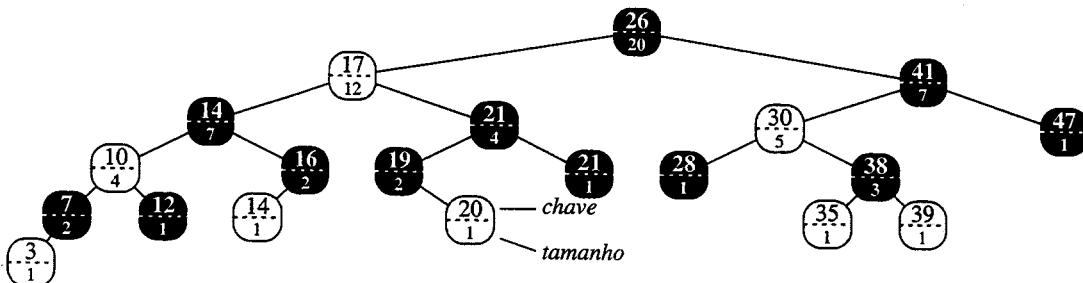


FIGURA 14.1 Uma árvore de estatísticas de ordem, que é uma árvore vermelho-preto ampliada. Os nós sombreados são vermelhos, e os nós escurecidos são pretos. Além de seus campos habituais, cada nó  $x$  tem um campo  $tamanho[x]$ , que é o número de nós na subárvore com raiz em  $x$ .

## Recuperação de um elemento com uma determinada ordem

Antes de mostrar como manter as informações de tamanho durante a inserção e eliminação, vamos examinar a implementação de duas consultas de estatísticas de ordem que utilizam essas informações adicionais. Começamos com uma operação que recupera um elemento com uma determinada ordem. O procedimento OS-SELECT( $x, i$ ) retorna um ponteiro para o nó contendo a  $i$ -ésima menor chave na subárvore com raiz em  $x$ . Para encontrar a  $i$ -ésima menor chave em uma árvore de estatísticas de ordem  $T$ , chamamos OS-SELECT( $raiz[T], i$ ).

```
OS-SELECT( $x, i$ )
1  $r \leftarrow tamanho[esquerda[x]] + 1$ 
2 if  $i = r$ 
3   then return  $x$ 
4 elseif  $i < r$ 
5   then return OS-SELECT( $esquerda[x], i$ )
6 else return OS-SELECT( $direita[x], i - r$ )
```

A idéia por trás de OS-SELECT é semelhante à dos algoritmos de seleção do Capítulo 9. O valor de  $tamanho[esquerda[x]]$  é o número de nós que vêm antes de  $x$  em um percurso de árvore em ordem da subárvore com raiz em  $x$ . Assim,  $tamanho[esquerda[x]] + 1$  é a ordem de  $x$  dentro da subárvore com raiz em  $x$ .

Na linha 1 de OS-SELECT, calculamos  $r$ , a ordem do nó  $x$  dentro da subárvore com raiz em  $x$ . Se  $i = r$ , então o nó  $x$  será o  $i$ -ésimo menor elemento, e assim retornaremos  $x$  na linha 3. Se  $i < r$ , então o  $i$ -ésimo menor elemento estará na subárvore esquerda de  $x$  e, portanto, faremos a recur-

são sobre  $esquerda[x]$  na linha 5. Se  $i > r$ , então o  $i$ -ésimo menor elemento estará na subárvore direita de  $x$ . Tendo em vista que existem  $r$  elementos na subárvore com raiz em  $x$  que vêm antes da subárvore direita de  $x$  em um percurso de árvore em ordem, o  $i$ -ésimo menor elemento na subárvore com raiz em  $x$  será o  $(i - r)$ -ésimo menor elemento na subárvore com raiz em  $direita[x]$ . Esse elemento é determinado recursivamente na linha 6.

Para ver como OS-SELECT opera, considere uma procura pelo 17º menor elemento na árvore de estatísticas de ordem da Figura 14.1. Começamos com  $x$  como a raiz, cuja chave é 26, e com  $i = 17$ . Como o tamanho da subárvore esquerda de 26 é 12, sua ordem é 13. Desse modo, sabemos que o nó com ordem 17 é o  $17 - 13 = 4$ º menor elemento na subárvore direita de 26. Após a chamada recursiva,  $x$  será o nó com chave 41, e  $i = 4$ . Tendo em vista que o tamanho da subárvore esquerda de 41 é 5, sua ordem dentro da subárvore é 6. Portanto, sabemos que o nó com ordem 4 está no 4º menor elemento da subárvore esquerda de 41. Após a chamada recursiva,  $x$  será o nó com chave 30, e sua ordem dentro de sua subárvore será 2. Assim, faremos mais uma vez uma recursão para encontrar o  $4 - 2 = 2$ º menor elemento na subárvore com raiz no nó com chave 38. Agora, descobrimos que sua subárvore esquerda tem tamanho 1, o que significa que ele é o segundo menor elemento. Desse modo, um ponteiro para o nó com chave 38 é retornado pelo procedimento.

Pelo fato de cada chamada recursiva descer um nível na árvore de estatísticas de ordem, o tempo total para OS-SELECT será, na pior das hipóteses, proporcional à altura da árvore. Como se trata de uma árvore vermelho-preto, sua altura é  $O(\lg n)$ , onde  $n$  é o número de nós. Desse modo, o tempo de execução de OS-SELECT é  $O(\lg n)$  para um conjunto dinâmico de  $n$  elementos.

## Determinação da ordem de um elemento

Dado um ponteiro para um nó  $x$  em uma árvore de estatísticas de ordem  $T$ , o procedimento OS-RANK retorna a posição de  $x$  na ordem linear determinada por um percurso de árvore em ordem de  $T$ .

```
OS-RANK( $T, x$ )
1  $r \leftarrow tamanbo[esquerda[x]] + 1$ 
2  $y \leftarrow x$ 
3 while  $y \neq raiz[T]$ 
4   do if  $y = direita[p[y]]$ 
5     then  $r \leftarrow r + tamanbo[esquerda[p[y]]] + 1$ 
6      $y \leftarrow p[y]$ 
7 return  $r$ 
```

O procedimento funciona da maneira descrita a seguir. A ordem de  $x$  pode ser vista como o número de nós que precedem  $x$  em um percurso de árvore em ordem, mais 1 para o próprio  $x$ . OS-RANK mantém o seguinte loop invariante:

No início de cada iteração do loop **while** das linhas 3 a 6,  $r$  é a ordem de  $chave[x]$  na subárvore com raiz no nó  $y$ .

Usamos esse invariante loop para mostrar que OS-RANK funciona corretamente como a seguir:

**Inicialização:** Antes da primeira iteração, a linha 1 define  $r$  como a ordem de  $chave[x]$  dentro da subárvore com raiz em  $x$ . A definição de  $y \geq x$  na linha 2 torna o invariante verdadeiro na primeira vez que o teste na linha 3 é executado.

**Manutenção:** No fim de cada iteração do loop **while**, definimos  $y \geq p[y]$ . Desse modo, devemos mostrar que, se  $r$  é a ordem de  $chave[x]$  na subárvore com raiz em  $y$  no início do corpo do loop, então  $r$  é a ordem de  $chave[x]$  na subárvore com raiz em  $p[y]$  no fim do corpo do loop.

Em cada iteração do loop **while**, consideramos a subárvore com raiz em  $p[y]$ . Já contamos o número de nós na subárvore com raiz no nó  $y$  que precedem  $x$  em um percurso em ordem; assim, devemos adicionar os nós na subárvore com raiz no irmão de  $y$  que precedem  $x$  em um percurso em ordem, mais 1 para  $p[y]$ , se ele também precede  $x$ . Se  $y$  é um filho da esquerda, então nem  $p[y]$  nem qualquer nó na subárvore direita de  $p[y]$  precede  $x$ ; portanto, deixamos  $r$  como está. Caso contrário,  $y$  é um filho da direita e todos os nós na subárvore esquerda de  $p[y]$  precedem  $x$ , como o próprio  $p[y]$ . Desse modo, na linha 5, adicionamos  $tamanho[esquerda[p[y]]] + 1$  ao valor atual de  $r$ .

**Término:** O loop termina quando  $y = raiz[T]$ , de forma que a subárvore com raiz em  $y$  é a árvore inteira. Desse modo, o valor de  $r$  é o grau de  $chave[x]$  na árvore toda.

Como exemplo, quando executamos OS-RANK na árvore de estatísticas de ordem da Figura 14.1 para encontrar a ordem do nó com chave 38, obtemos a seqüência de valores de  $chave[y]$  e  $r$  a seguir no início do loop **while**:

iteração	$chave[y]$	$r$
1	38	2
2	30	4
3	41	4
4	26	17

A ordem 17 é retornada.

Tendo em vista que cada iteração do loop **while** leva o tempo  $O(1)$  e como  $y$  sobe um nível na árvore com cada iteração, o tempo de execução de OS-RANK é, na pior das hipóteses, proporcional à altura da árvore:  $O(\lg n)$  em uma árvore de estatísticas de ordem de  $n$  nós.

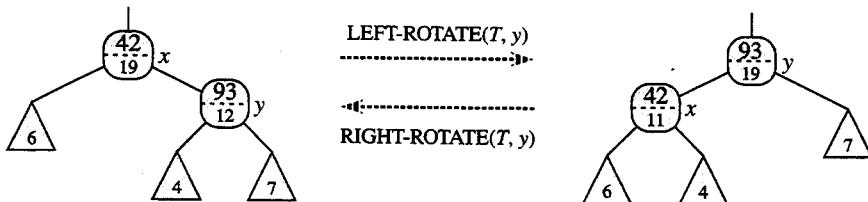


FIGURA 14.2 Atualização de tamanhos de subárvores durante rotações. A ligação em torno da qual a rotação é executada é incidente nos dois nós cujos campos *tamanho* precisam ser atualizados. As atualizações são locais, exigindo apenas as informações de *tamanho* armazenadas em  $x$ ,  $y$  e nas raízes das subárvores mostradas como triângulos

## Manutenção de tamanhos de subárvores

Dado o campo *tamanho* em cada nó, OS-SELECT e OS-RANK podem calcular com rapidez informações de estatísticas de ordem. Porém, a menos que esses campos possam ser mantidos de forma eficiente pelas operações básicas de modificação sobre árvores vermelho-preto, nosso trabalho terá sido em vão. Agora, mostraremos que os tamanhos de subárvores podem ser mantidos, tanto para inserção quanto para eliminação, sem afetar os tempos de execução assintóticos de uma ou de outra operação.

Observamos na Seção 13.3 que a inserção em uma árvore vermelho-preto consiste em duas fases. A primeira fase percorre a árvore de cima para baixo a partir da raiz, inserindo o novo nó como um filho de um nó existente. A segunda fase sobe a árvore, alterando cores e, em última análise, executando rotações para manter as propriedades vermelho-preto.

Para manter os tamanhos das subárvores na primeira fase, simplesmente incrementamos *tamanho*[*x*] para cada nó *x* no caminho percorrido para baixo desde a raiz até às folhas. O novo nó adicionado obtém um *tamanho* igual a 1. Tendo em vista que existem  $O(\lg n)$  nós no caminho percorrido, o custo adicional de manter os campos *tamanho* é  $O(\lg n)$ .

Na segunda fase, as únicas mudanças estruturais na árvore vermelho-preto subjacente são causadas por rotações, das quais existem no máximo duas. Além disso, uma rotação é uma operação local: somente dois nós têm seus campos *tamanho* invalidados. A ligação em torno da qual a rotação é executada incide sobre esses dois nós. Retornando ao código de LEFT-ROTATE(*T*, *x*) na Seção 13.2, adicionamos as seguintes linhas:

```
12 tamanho[y] ← tamanho[x]
13 tamanho[x] ← tamanho[esquerda[x]] + tamanho[direita[x]] + 1
```

A Figura 14.2 ilustra como os campos são atualizados. A mudança em RIGHT-ROTATE é simétrica.

Tendo em vista que são executadas no máximo duas rotações durante a inserção em uma árvore vermelho-preto, somente o tempo adicional  $O(1)$  é despendido na atualização de campos *tamanho* na segunda fase. Portanto, o tempo total para inserção em uma árvore de estatísticas de ordem de  $n$  nós é  $O(\lg n)$  – assintoticamente igual ao de uma árvore vermelho-preto comum.

A eliminação de uma árvore vermelho-preto também consiste em duas fases: a primeira opera sobre a árvore de pesquisa subjacente, e a segunda provoca no máximo três rotações e, além disso, não executa nenhuma mudança estrutural. (Consulte a Seção 13.4.) A primeira fase extrai um nó *y*. Para atualizar os tamanhos das subárvores, simplesmente percorremos um caminho desde o nó *y* até a raiz, decrementando o campo *tamanho* de cada nó no caminho. Tendo em vista que esse caminho tem o comprimento  $O(\lg n)$  em uma árvore vermelho-preto de  $n$  nós, o tempo adicional despendido na manutenção de campos *tamanho* na primeira fase é  $O(\lg n)$ . As  $O(1)$  rotações na segunda fase de eliminação podem ser tratadas da mesma maneira que no caso da inserção. Desse modo, tanto a inserção quanto a eliminação, incluindo a manutenção dos campos *tamanho*, demoram o tempo  $O(\lg n)$  para uma árvore de estatísticas de ordem de  $n$  nós.

## Exercícios

### 14.1-1

Mostre como OS-SELECT(*T*, 10) opera sobre a árvore vermelho-preto *T* da Figura 14.1.

### 14.1-2

Mostre como OS-RANK(*T*, *x*) opera sobre a árvore vermelho-preto *T* da Figura 14.1 e sobre o nó *x* com *chave*[*x*] = 35.

### 14.1-3

Escreva uma versão não recursiva de OS-SELECT.

### 14.1-4

Escreva um procedimento recursivo OS-KEY-RANK(*T*, *k*) que tome como entrada uma árvore de estatísticas de ordem *T* e uma chave *k* e retorne a ordem de *k* no conjunto dinâmico representado por *T*. Suponha que as chaves de *T* sejam distintas.

### 14.1-5

Dado um elemento *x* em uma árvore de estatísticas de ordem de  $n$  nós e um número natural *i*, de que modo o *i*-ésimo sucessor de *x* na ordem linear da árvore pode ser determinado no tempo  $O(\lg n)$ ?

### 14.1-6

Observe que, sempre que o campo *tamanho* de um nó é referenciado em OS-SELECT ou OS-RANK, ele só é usado para calcular a ordem do nó na subárvore com raiz nesse nó. De acordo com isso, suponha que armazenamos em cada nó sua ordem na subárvore da qual ele é a raiz. Mostre como essas informações podem ser mantidas durante a inserção e a eliminação. (Lembre-se de que essas duas operações podem provocar rotações.)

### 14.1-7

Mostre como usar uma árvore de estatísticas de ordem para contar o número de inversões (ver Problema 2-4) em um arranjo de tamanho  $n$  no tempo  $O(n \lg n)$ .

### 14.1-8 \*

Considere  $n$  cordas em um círculo, cada uma definida por seus pontos extremos. Descreva um algoritmo de tempo  $O(n \lg n)$  para determinar o número de pares de cordas que se interceptam no interior do círculo. (Por exemplo, se as  $n$  cordas são todas diâmetros que se encontram no centro, então a resposta correta é  $\binom{n}{2}$ ) Suponha que nenhum par de cordas compartilha um ponto extremo.

## 14.2 Como ampliar uma estrutura de dados

O processo de ampliar uma estrutura de dados básica para fornecer suporte a funções adicionais ocorre com bastante freqüência no projeto de algoritmos. Ele será usado novamente na próxima seção para projetar uma estrutura de dados que fornece suporte a operações sobre intervalos. Nesta seção, examinaremos as etapas envolvidas em tal ampliação. Também provaremos um teorema que nos permitirá ampliar árvores vermelho-preto com facilidade em muitos casos.

O processo de ampliar uma estrutura de dados pode ser dividido em quatro etapas:

1. Escolher uma estrutura de dados subjacente.
2. Descobrir informações adicionais a serem mantidas na estrutura de dados subjacente.
3. Verificar se as informações adicionais podem ser mantidas para as operações básicas de modificação sobre a estrutura de dados subjacentes.
4. Desenvolver novas operações.

Como ocorre com qualquer método de projeto prescritivo, você não deve seguir cegamente as etapas na ordem dada. A maior parte do trabalho de projeto contém um elemento de tentativa e erro, e o progresso em todas as etapas geralmente se dá em paralelo. Por exemplo, não existe nenhuma utilidade em determinar informações adicionais e desenvolver novas operações (Etapas 2 e 4) se não formos capazes de manter as informações adicionais de modo eficiente. Apesar disso, esse método de quatro etapas fornece um bom enfoque para nossos esforços de ampliar uma estrutura de dados, e é também um bom modo de organizar a documentação de uma estrutura de dados ampliada.

Seguimos essas etapas na Seção 14.1 para projetar nossas árvores de estatísticas de ordem. Para a Etapa 1, escolhemos as árvores vermelho-preto como a estrutura de dados subjacentes. Uma pista para determinar a adequação de árvores vermelho-preto vem de seu suporte eficiente para outras operações de conjuntos dinâmicos em uma ordem total, como MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR.

Para a Etapa 2, fornecemos o campo *tamanho* que, em cada nó  $x$ , armazena o tamanho da subárvore com raiz em  $x$ . Em geral, as informações adicionais tornam as operações mais eficientes. Por exemplo, poderíamos ter implementado OS-SELECT e OS-RANK usando apenas as chaves armazenadas na árvore, mas eles não teriam sido executados em tempo  $O(\lg n)$ . Algumas vezes, as informações adicionais são informações de ponteiros em lugar de dados, como no Exercício 14.2-1.

Para a Etapa 3, asseguramos que a inserção e a eliminação poderiam manter os campos *tamanho* enquanto ainda fossem executadas no tempo  $O(\lg n)$ . No caso ideal, um pequeno número de mudanças na estrutura de dados deve ser suficiente para manter as informações adicionais. Por exemplo, se simplesmente armazenássemos em cada nó sua ordem na árvore, os procedimentos OS-SELECT e OS-RANK seriam executados rapidamente, mas a inserção de um novo elemento mínimo causaria uma mudança nessas informações em cada nó da árvore. Em vez disso, quando armazenamos tamanhos de subárvore, a inserção de um novo elemento faz as informações mudarem apenas em  $O(\lg n)$  nós.

Para a Etapa 4, desenvolvemos as operações OS-SELECT e OS-RANK. Afinal, a necessidade de novas operações é o motivo pelo qual nos preocupamos em ampliar uma estrutura de dados. Ocasionalmente, em vez de desenvolver novas operações, usamos as informações adicionais para acelerar operações existentes, como no Exercício 14.2-1.

## Como ampliar árvores vermelho-preto

Quando árvores vermelho-preto formam a base de uma estrutura de dados ampliada, podemos provar que certos tipos de informações adicionais sempre podem ser mantidos de forma eficiente por inserção e eliminação, tornando assim a Etapa 3 muito fácil. A prova do teorema a seguir é semelhante ao argumento da Seção 14.1 de que o campo *tamanho* pode ser mantido no caso de árvores de estatísticas de ordem.

### **Teorema 14.1 (Como ampliar uma árvore vermelho-preto)**

Seja  $f$  um campo que amplia uma árvore vermelho-preto  $T$  de  $n$  nós, e suponha que o conteúdo de  $f$  para um nó  $x$  possa ser calculado usando-se apenas as informações contidas nos nós  $x$ , *esquerda*[ $x$ ] e *direita*[ $x$ ], inclusive  $f[\text{esquerda}[x]]$  e  $f[\text{direita}[x]]$ . Então, podemos manter os valores de  $f$  em todos os nós de  $T$  durante a inserção e a eliminação, sem afetar assintoticamente o desempenho  $O(\lg n)$  dessas operações.

**Prova** A principal idéia da prova é que uma mudança em um campo  $f$  em um nó  $x$  se propaga apenas até os ancestrais de  $x$  na árvore. Isto é, a mudança de  $f[x]$  pode exigir que  $f[p[x]]$  seja atualizado, mas nada além disso; a atualização de  $f[p[x]]$  pode exigir que  $f[p[p[x]]]$  seja atualizado, mas nada além disso; e assim por diante para cima na árvore. Quando  $f[\text{raiz}[T]]$  é atualizado, nenhum outro nó depende do novo valor e assim o processo termina. Como a altura de uma árvore vermelho-preto é  $O(\lg n)$ , a mudança em um campo  $f$  em um nó custa o tempo  $O(\lg n)$  na atualização de nós que dependem da mudança.

A inserção de um nó  $x$  em  $T$  consiste em duas fases. (Consulte a Seção 13.3.) Durante a primeira fase,  $x$  é inserido como um filho de um nó existente  $p[x]$ . O valor para  $f[x]$  pode ser calculado no tempo  $O(1)$  pois, por hipótese, ele depende apenas das informações nos outros campos do próprio  $x$  e das informações nos filhos de  $x$ , mas os filhos de  $x$  são ambos a sentinela  $\text{nil}[T]$ . Depois que  $f[x]$  é calculado, a mudança se propaga para cima na árvore. Desse modo, o tempo total para a primeira fase de inserção é  $O(\lg n)$ . Durante a segunda fase, as únicas mudanças estruturais na árvore vêm de rotações. Tendo em vista que apenas dois nós mudam em uma rotação, o tempo total para atualizar os campos  $f$  é  $O(\lg n)$  por rotação. Como o número de rotações durante a inserção é no máximo dois, o tempo total para a inserção é  $O(\lg n)$ .

Como a inserção, a eliminação tem duas fases. (Consulte a Seção 13.4.) Na primeira fase, as mudanças na árvore ocorrem se o nó eliminado é substituído por seu sucessor, e depois novamente quando o nó eliminado ou seu sucessor é extraído. A propagação das atualizações até  $f$  causada por essas mudanças custa no máximo  $O(\lg n)$ , pois as mudanças na árvore são locais. A correção da árvore vermelho-preto durante a segunda fase exige no máximo três rotações, e cada rotação exige no máximo o tempo  $O(\lg n)$  para propagar as atualizações até  $f$ . Desse modo, como a inserção, o tempo total para eliminação é  $O(\lg n)$ . ■

Em muitos casos, como a manutenção dos campos *tamanho* em árvores de estatísticas de ordem, o custo da atualização após uma rotação é  $O(1)$ , em vez do custo  $O(\lg n)$  derivado na prova do Teorema 14.1. O Exercício 14.2-4 fornece um exemplo.

## Exercícios

### 14.2-1

Mostre como cada uma das consultas de conjuntos dinâmicos MINIMUM, MAXIMUM, SUCCESSION e PREDECESSOR pode ser admitida no tempo de pior caso  $O(1)$  em uma árvore de estatísticas de ordem ampliada. O desempenho assintótico de outras operações sobre árvores de estatísticas de ordem não deve ser afetado. (Sugestão: Adicione ponteiros a nós.)

### 14.2-2

As alturas de preto dos nós de uma árvore vermelho-preto podem ser mantidas como campos nos nós da árvore sem afetar o desempenho assintótico de qualquer das operações de árvores vermelho-preto? Mostre como, ou explique por que não.

### 14.2-3

As profundidades de nós em uma árvore vermelho-preto podem ser mantidas de modo eficiente como campos nos nós da árvore? Mostre como, ou explique por que não.

### 14.2-4 \*

Seja  $\otimes$  um operador binário associativo, e seja  $\alpha$  um campo mantido em cada nó de uma árvore vermelho-preto. Suponha que quiséssemos incluir em cada nó  $x$  um campo adicional  $f$  tal que  $f[x] = \alpha[x_1] \otimes \alpha[x_2] \otimes \dots \otimes \alpha[x_m]$ , onde  $x_1, x_2, \dots, x_m$  é a listagem em ordem de nós da subárvore com raiz em  $x$ . Mostre que os  $f$  campos podem ser atualizados corretamente em tempo  $O(1)$  depois de uma rotação. Modifique ligeiramente seu argumento para mostrar que os campos *tamanho* em árvores de estatísticas de ordem podem ser mantidos no tempo  $O(1)$  por rotação.

### 14.2-5 \*

Desejamos ampliar as árvores vermelho-preto com uma operação RB ENUMERATE( $x, a, b$ ) que dê saída a todas as chaves  $k$  tais que  $a \leq k \leq b$  em uma árvore vermelho-preto com raiz em  $x$ . Descreva como RB ENUMERATE pode ser implementada em tempo  $\Theta(m + \lg n)$ , onde  $m$  é o número de chaves na saída e  $n$  é o número de nós internos na árvore. (Sugestão: Não há necessidade de adicionar novos campos à árvore vermelho-preto.)

## 14.3 Árvores de intervalos

Nesta seção, ampliaremos as árvores vermelho-preto para dar suporte a operações sobre conjuntos dinâmicos de intervalos. Um *intervalo fechado* é um par ordenado de números reais  $[t_1, t_2]$ , com  $t_1 \leq t_2$ . O intervalo  $[t_1, t_2]$  representa o conjunto  $\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$ . Intervalos *abertos* e *meio abertos* excluem ambos ou um dos pontos extremos do conjunto, respectivamente. Nesta seção, consideraremos que os intervalos são fechados; a extensão dos resultados a intervalos abertos e meio abertos é conceitualmente direta.

Os intervalos são convenientes para representar eventos tais que cada um ocupa um período contínuo de tempo. Por exemplo, talvez quiséssemos consultar um banco de dados de intervalos de tempo para descobrir quais eventos ocorreram durante um determinado intervalo. A estrutura de dados nesta seção fornece um meio eficiente para manter esse banco de dados de intervalos.

Podemos representar um intervalo  $[t_1, t_2]$  como um objeto  $i$ , com campos  $baixo[i] = t_1$  (o *ponto extremo baixo* ou *inferior*) e  $alto[i] = t_2$  (o *ponto extremo alto* ou *superior*). Dizemos que os intervalos  $i$  e  $i'$  se *superpõem* se  $i \cap i' \neq \emptyset$ , ou seja, se  $baixo[i] = alto[i']$  e  $baixo[i'] = alto[i]$

$= \text{alto}[i]$ . Dois intervalos quaisquer  $i$  e  $i'$  satisfazem à **tricotomia de intervalos**; isto é, exatamente uma das três propriedades a seguir é válida:

- a.  $i$  e  $i'$  se superpõem.
- b.  $i$  está à esquerda de  $i'$  (isto é,  $\text{alto}[i] < \text{baixo}[i']$ ).
- c.  $i$  está à direita de  $i'$  (isto é,  $\text{alto}[i'] < \text{baixo}[i]$ ).

A Figura 14.3 mostra as três possibilidades.

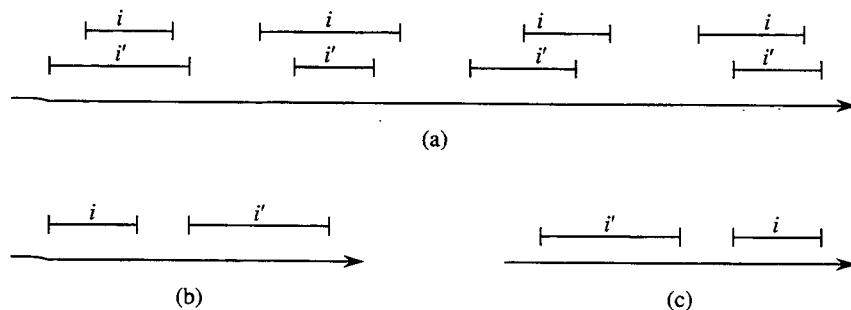


FIGURA 14.3 A tricotomia de intervalos para dois intervalos fechados  $i$  e  $i'$ . (a) Se  $i$  e  $i'$  se superpõem, há quatro situações; em cada uma,  $\text{baixo}[i] \leq \text{alto}[i'] \leq \text{alto}[i]$ . (b) Os intervalos não se superpõem, e  $\text{alto}[i] < \text{baixo}[i']$ . (c) Os intervalos não se superpõem, e  $\text{alto}[i'] < \text{baixo}[i]$

Uma **árvore de intervalos** é uma árvore vermelho-preto que mantém um conjunto dinâmico de elementos, com cada elemento  $x$  contendo um intervalo  $\text{int}[x]$ . As árvores de intervalos oferecem suporte para as operações a seguir.

**INTERVAL-INSERT( $T, x$ )** adiciona o elemento  $x$ , cujo campo  $\text{int}$  deve conter um intervalo, à árvore de intervalos  $T$ .

**INTERVAL-DELETE( $T, x$ )** remove o elemento  $x$  da árvore de intervalos  $T$ .

**INTERVAL-SEARCH( $T, i$ )** retorna um ponteiro para um elemento  $x$  na árvore de intervalos  $T$  tal que  $\text{int}[x]$  se superpõe ao intervalo  $i$ , ou a sentinela  $\text{nil}[T]$ , se não existe tal elemento no conjunto.

A Figura 14.4 mostra como uma árvore de intervalos representa um conjunto de intervalos. Acompanharemos o método de quatro etapas da Seção 14.2, à medida que revisarmos o projeto de uma árvore de intervalos e as operações que são executadas sobre ela.

## Etapa 1: Estrutura de dados subjacente

Escolhemos uma árvore vermelho-preto em que cada nó  $x$  contém um intervalo  $\text{int}[x]$  e a chave de  $x$  é o ponto extremo baixo,  $\text{baixo}[\text{int}[x]]$ , do intervalo. Desse modo, um percurso de árvore em ordem da estrutura de dados lista os intervalos em seqüência ordenada por ponto extremo baixo.

## Etapa 2: Informações adicionais

Além dos próprios intervalos, cada nó  $x$  contém um valor  $\text{max}[x]$ , que é o valor máximo de qualquer ponto extremo de intervalo armazenado na subárvore com raiz em  $x$ .

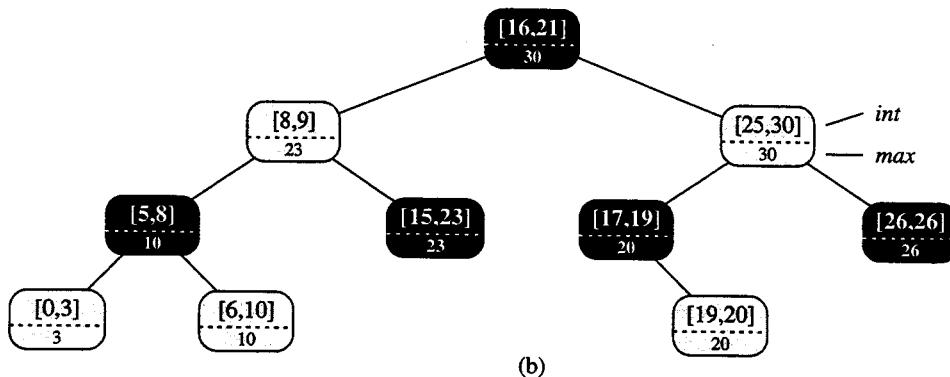
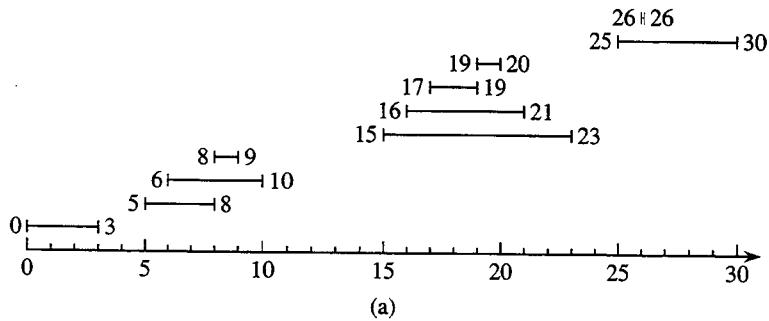


FIGURA 14.4 Uma árvore de intervalos. (a) Um conjunto de 10 intervalos mostrados em seqüência ordenada de baixo para cima por ponto extremo esquerdo. (b) A árvore de intervalos que os representa. Um percurso de árvore em ordem da árvore lista os nós em seqüência ordenada por ponto extremo esquerdo

### Etapa 3: Manutenção das informações

Devemos verificar que a inserção e a eliminação podem ser executadas no tempo  $O(\lg n)$  em uma árvore de intervalos de  $n$  nós. Podemos determinar  $\max[x]$  dado o intervalo  $\text{int}[x]$  e os valores  $\max$  dos filhos do nó  $x$ :

$$\max[x] = \max(\text{alto}[\text{int}[x]], \max[\text{esquerda}[x]], \max[\text{direita}[x]]).$$

Desse modo, pelo Teorema 14.1, a inserção e a eliminação são executadas no tempo  $O(\lg n)$ . De fato, a atualização dos campos  $\max$  depois de uma rotação pode ser realizada no tempo  $O(1)$ , como mostram os Exercícios 14.2-4 e 14.3-1.

### Etapa 4: Desenvolvimento de novas operações

A única operação nova de que necessitamos é INTERVAL-SEARCH( $T, i$ ), que encontra um nó na árvore  $T$  cujo intervalo se superpõe ao intervalo  $i$ . Se não existir nenhum intervalo que se superponha a  $i$  na árvore, um ponteiro para a sentinela  $\text{nil}[T]$  será retornado.

INTERVAL-SEARCH( $T, i$ )

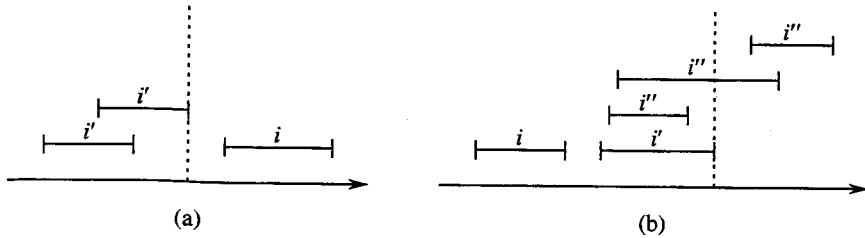
- 1  $x \leftarrow \text{raiz}[T]$
- 2 **while**  $x \neq \text{nil}[T]$  e  $i$  não se superpõe a  $\text{int}[x]$
- 3   **do if**  $\text{esquerda}[x] \neq \text{nil}[T]$  e  $\max[\text{esquerda}[x]] \geq \text{baixo}[i]$
- 4     **then**  $x \leftarrow \text{esquerda}[x]$
- 5     **else**  $x \leftarrow \text{direita}[x]$
- 6 **return**  $x$

A procura por um intervalo que se superponha a  $i$  começa com  $x$  na raiz da árvore e prossegue no sentido descendente. Ela termina quando é encontrado um intervalo de superposição ou quando  $x$  aponta para a sentinela  $\text{nil}[T]$ . Como cada iteração do loop básico demora o tempo  $O(1)$ , e como a altura de uma árvore vermelho-preto de  $n$  nós é  $O(\lg n)$ , o procedimento INTERVAL-SEARCH demora o tempo  $O(\lg n)$ .

Antes de verificarmos o motivo pelo qual INTERVAL-SEARCH é correto, vamos examinar como ele funciona na árvore de intervalos da Figura 14.4. Vamos supor que desejamos encontrar um intervalo que se superponha ao intervalo  $i = [22, 25]$ . Começamos com o nó  $x$  como raiz, o qual contém  $[16, 21]$  e não se superpõe a  $i$ . Tendo em vista que  $\text{max}[\text{esquerda}[x]] = 23$  é maior que  $\text{baixo}[i] = 22$ , o loop continua com  $x$  como o filho esquerdo da raiz – o nó contendo  $[8, 9]$ , que também não se superpõe a  $i$ . Dessa vez,  $\text{max}[\text{esquerda}[x]] = 10$  é menor que  $\text{baixo}[i] = 22$ , e assim o loop continua com o filho da direita de  $x$  como o novo  $x$ . O intervalo  $[15, 23]$  armazenado nesse nó se superpõe a  $i$ , e então o procedimento retorna esse nó.

Como exemplo de uma procura malsucedida, vamos supor que desejamos encontrar um intervalo que se superponha a  $i = [11, 14]$  na árvore de intervalos da Figura 14.4. Mais uma vez, começamos com  $x$  como raiz. Como o intervalo  $[16, 21]$  da raiz não se superpõe a  $i$ , e como  $\text{max}[\text{esquerda}[x]] = 23$  é maior que  $\text{baixo}[i] = 11$ , vamos para a esquerda até o nó que contém  $[8, 9]$ . (Observe que nenhum intervalo na subárvore da direita se superpõe a  $i$  – veremos mais adiante por que isso acontece.) O intervalo  $[8, 9]$  não se superpõe a  $i$ , e  $\text{max}[\text{esquerda}[x]] = 10$  é menor que  $\text{baixo}[i] = 11$  e assim vamos para a direita. (Observe que nenhum intervalo na subárvore da esquerda se superpõe a  $i$ .) O intervalo  $[15, 23]$  não se superpõe a  $i$ , e seu filho esquerdo é  $\text{nil}[T]$ ; assim, vamos para a direita, o loop termina e a sentinela  $\text{nil}[T]$  é retornada.

Para ver por que INTERVAL-SEARCH é correto, devemos entender por que basta examinar um único caminho a partir da raiz. A idéia básica é que em qualquer nó  $x$ , se  $\text{int}[x]$  não se superpõe a  $i$ , a procura sempre prossegue em uma direção segura: o intervalo de superposição será definitivamente encontrado se existir na árvore. O teorema a seguir enuncia essa propriedade de forma mais precisa.



**FIGURA 14.5** Intervalos na prova do Teorema 14.2. O valor de  $\text{max}[\text{esquerda}[x]]$  é mostrado em cada caso como uma linha tracejada. (a) A pesquisa prossegue para a direita. Nenhum intervalo  $i'$  na subárvore esquerda de  $x$  pode se superpor a  $i$ . (b) A pesquisa segue para a esquerda. A subárvore esquerda de  $x$  contém um intervalo que se superpõe a  $i$  (situação não mostrada), ou então existe um intervalo  $i'$  na subárvore esquerda de  $x$  tal que  $\text{alto}[i'] = \text{max}[\text{esquerda}[x]]$ . Tendo em vista que  $i$  não se superpõe a  $i'$ , ele também não se superpõe a qualquer intervalo  $i''$  na subárvore direita de  $x$ , pois  $\text{baixo}[i'] \leq \text{baixo}[i'']$

### Teorema 14.2

Qualquer execução de  $\text{INTERVAL-SEARCH}(T, i)$  retorna um nó cujo intervalo se superpõe a  $i$ , ou retorna  $\text{nil}[T]$  e a árvore  $T$  não contém nenhum nó cujo intervalo se superpõe a  $i$ .

**Prova** O loop while das linhas 2 a 5 termina quando  $x = \text{nil}[T]$  ou  $i$  se superpõe a  $\text{int}[x]$ . Nesse último caso, certamente é correto retornar  $x$ . Assim, vamos nos concentrar no primeiro caso, no qual o loop while termina porque  $x = \text{nil}[T]$ .

Usamos o seguinte invariante para o loop while das linhas 2 a 5:

Se a árvore  $T$  contém um intervalo que se superpõe a  $i$ , então existe tal intervalo na subárvore com raiz em  $x$ .

Usamos esse loop invariante da maneira descrita a seguir:

**Inicialização:** Antes da primeira iteração, a linha 1 define  $x$  como a raiz de  $T$ , de modo que o invariante é válido.

**Manutenção:** Em cada iteração do loop while, a linha 4 ou a linha 5 é executada. Mostraremos que o loop invariante é mantido em um ou outro caso.

Se a linha 5 é executada, então – por causa da condição de desvio na linha 3 – temos  $esquerda[x] = nil[T]$ , ou  $\max[esquerda[x]] < baixo[i]$ . Se  $esquerda[x] = nil[T]$ , a subárvore com raiz em  $esquerda[x]$  claramente não contém nenhum intervalo que se superponha a  $i$ , e assim a definição de  $x$  como  $direita[x]$  mantém o invariante. Portanto, suponha que  $esquerda[x] \neq nil[T]$  e  $\max[esquerda[x]] < baixo[i]$ . Como mostra a Figura 14.5, para cada intervalo  $i'$  na subárvore esquerda de  $x$ , temos

$$\begin{aligned} alto[i'] &\leq \max[esquerda[x]] \\ &< baixo[i] \end{aligned}$$

Portanto, pela tricotomia de intervalos,  $i'$  e  $i$  não se superpõem. Desse modo, a subárvore esquerda de  $x$  não contém nenhum intervalo que se superponha a  $i$ , de forma que a definição de  $x$  como  $direita[x]$  mantém o invariante.

Se, por outro lado, a linha 4 for executada, mostraremos que a antítese do loop invariante é válida. Isto é, se não existe nenhum intervalo que se superponha a  $i$  na subárvore com raiz em  $esquerda[x]$ , então não há nenhum intervalo que se superponha a  $i$  em nenhum lugar na árvore. Tendo em vista que a linha 4 é executada, então – por causa da condição de desvio na linha 3 – temos  $\max[esquerda[x]] \geq baixo[i]$ . Além disso, pela definição do campo  $max$ , deve haver algum intervalo  $i'$  na subárvore esquerda de  $x$  tal que

$$\begin{aligned} alto[i'] &= \max[esquerda[x]] \\ &\geq baixo[i]. \end{aligned}$$

(A Figura 14.5(b) ilustra a situação.) Tendo em vista que  $i$  e  $i'$  não se superpõem, e como não é verdade que  $alto[i'] < baixo[i]$ , segue-se pela tricotomia de intervalos que  $alto[i] < baixo[i']$ . As árvores de intervalos são chaveadas nos pontos extremos baixos de intervalos, e desse modo a propriedade de árvore de pesquisa implica que, para qualquer intervalo  $i''$  na subárvore direita de  $x$ ,

$$\begin{aligned} alto[i] &< baixo[i'] \\ &\leq baixo[i'']. \end{aligned}$$

Pela tricotomia de intervalos,  $i$  e  $i''$  não se superpõem. Concluímos que, independente de qualquer intervalo na subárvore esquerda de  $x$  se superpor a  $i$ , a definição de  $x$  como  $esquerda[x]$  mantém o invariante.

**Término:** Se o loop termina quando  $x = \text{nil}[T]$ , não existe nenhum intervalo que se superponha a  $i$  na subárvore com raiz em  $x$ . A antítese do loop invariante implica que  $T$  não contém nenhum intervalo que se superponha a  $i$ . Conseqüentemente, é correto retornar  $x = \text{nil}[T]$ . ■

Portanto, o procedimento INTERVAL-SEARCH funciona corretamente.

## Exercícios

### 14.3-1

Escreva pseudocódigo para LEFT-ROTATE que opere sobre os nós em uma árvore de intervalos e atualize os campos  $\max$  no tempo  $O(1)$ .

### 14.3-2

Reescreva o código para INTERVAL-SEARCH de tal forma que ele funcione de modo correto quando todos os intervalos estiverem supostamente abertos.

### 14.3-3

Descreva um algoritmo eficiente que, dado um intervalo  $i$ , retorne uma superposição de intervalos  $i$  que tenha o ponto extremo baixo mínimo, ou então  $\text{nil}[T]$  se não existir nenhum intervalo desse tipo.

### 14.3-4

Dada uma árvore de intervalos  $T$  e um intervalo  $i$ , descreva como todos os intervalos em  $T$  que se superpõem a  $i$  podem ser listados no tempo  $O(\min(n, k \lg n))$ , onde  $k$  é o número de intervalos na lista de saída. (Opcional: Encontre uma solução que não modifique a árvore.)

### 14.3-5

Sugira modificações para os procedimentos de árvores de intervalos com o objetivo de fornecer suporte para a operação INTERVAL-SEARCH-EXACTLY( $T, i$ ), que retorna um ponteiro para um nó  $x$  na árvore de intervalos  $T$  tal que  $\text{baixo[int[x]]} = \text{baixo}[i]$  e  $\text{alto[int[x]]} = \text{alto}[i]$ , ou  $\text{nil}[T]$  se  $T$  não contém nenhum nó desse tipo. Todas as operações, inclusive INTERVAL-SEARCH-EXACTLY, devem ser executadas no tempo  $O(\lg n)$  em uma árvore de  $n$  nós.

### 14.3-6

Mostre como manter um conjunto dinâmico  $Q$  de números que oferece suporte para a operação MIN-GAP, que fornece a magnitude da diferença dos dois números mais próximos em  $Q$ . Por exemplo, se  $Q = \{1, 5, 9, 15, 18, 22\}$ , então  $\text{MIN-GAP}(Q)$  retorna  $18 - 15 = 3$ , pois 15 e 18 são os dois números mais próximos em  $Q$ . Torne as operações INSERT, DELETE, SEARCH e MIN-GAP tão eficientes quanto possível, e analise seus tempos de execução.

### 14.3-7 ★

Os bancos de dados VLSI representam comumente um circuito integrado como uma lista de retângulos. Suponha que cada retângulo esteja orientado de forma retilínea (lados paralelos aos eixos  $x$  e  $y$ ), de tal modo que a representação de um retângulo seja formada por suas coordenadas  $x$  e  $y$  mínima e máxima. Forneça um algoritmo de tempo  $O(n \lg n)$  para descobrir se um conjunto de retângulos assim representados contém ou não dois retângulos que se superpõem. Seu algoritmo não precisa relatar todos os pares que se interceptam, mas deve informar que existe uma superposição se um retângulo cobrir inteiramente outro retângulo, ainda que as linhas limites não se interceptem. (Sugestão: Mova uma linha “de varredura” pelo conjunto de retângulos.)

## Problemas

### 14-1 Ponto de superposição máxima

Suponha que desejamos manter o controle de um ponto de superposição máxima em um conjunto de intervalos – um ponto que tenha o maior número de intervalos no banco de dados que se superponham a ele.

- a. Mostre que sempre existirá um ponto de superposição máxima que seja uma extremidade de um dos segmentos.
- b. Projete uma estrutura de dados que forneça suporte eficiente para as operações INTERVAL-INSERT, INTERVAL-DELETE e FIND-POM, que retorna um ponto de superposição máxima. (*Sugestão:* Mantenha uma árvore vermelho-preto de todas as extremidades. Associe o valor +1 a cada extremidade esquerda, e associe o valor -1 a cada extremidade direita. Amplie cada nó da árvore com algumas informações extras para manter o ponto de superposição máxima.)

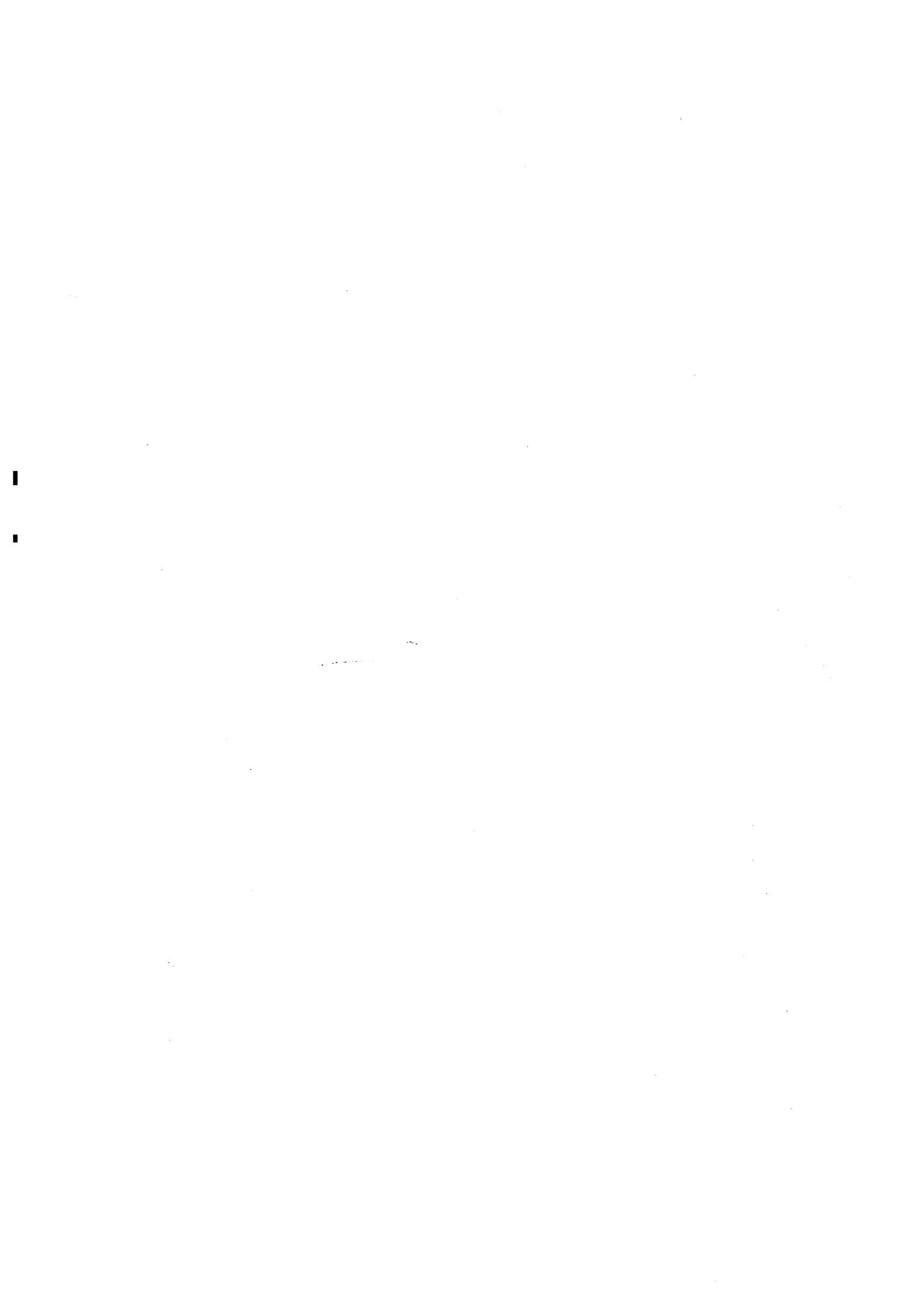
### 14-2 Permutação de Josephus

O problema de Josephus é definido da maneira mostrada a seguir. Vamos supor que  $n$  pessoas estão organizadas em um círculo e que temos um inteiro positivo  $m \leq n$ . Começando com uma primeira pessoa designada, prosseguimos em torno do círculo, removendo cada  $m$ -ésima pessoa. Depois que cada pessoa é removida, a contagem prossegue em torno do círculo restante. Esse processo continua até todas as  $n$  pessoas terem sido removidas. A ordem em que as pessoas são removidas do círculo define a **permutação de Josephus de  $(n, m)$**  dos inteiros  $1, 2, \dots, n$ . Por exemplo, a permutação de Josephus de  $(7, 3)$  é  $3, 6, 2, 7, 5, 1, 4$ .

- a. Suponha que  $m$  seja uma constante. Descreva um algoritmo de tempo  $O(n)$  que, dado um inteiro  $n$ , forneça como saída a permutação de Josephus de  $(n, m)$ .
- b. Suponha que  $m$  não seja uma constante. Descreva um algoritmo de tempo  $O(n \lg n)$  que, dados os inteiros  $n$  e  $m$ , forneça como saída a permutação de Josephus de  $(n, m)$ .

## Notas do capítulo

Em seu livro, Preparata e Shamos [160] descrevem diversas árvores de intervalos que aparecem na literatura, citando o trabalho de H. Edelsbrunner (1980) e E. M. McCreight (1981). O livro mostra em detalhes uma árvore de intervalos para a qual, dado um banco de dados de  $n$  intervalos, todos os  $k$  intervalos que se superpõem a um determinado intervalo de consulta podem ser enumerados no tempo  $O(k + \lg n)$ .



---

## *Parte IV*

# *Técnicas avançadas de projeto e análise*

### **Introdução**

Esta parte focaliza três técnicas importantes para o projeto e a análise eficiente de algoritmos: a programação dinâmica (Capítulo 15), os algoritmos gulosos (Capítulo 16) e a análise amortizada (Capítulo 17). Partes anteriores apresentaram outras técnicas extensamente aplicáveis, como a de dividir e conquistar, a de aleatoriedade e a solução de recorrências. As novas técnicas são um pouco mais sofisticadas, mas são úteis para se abordar de forma eficaz muitos problemas de computação. Os temas introduzidos nesta parte aparecerão periodicamente mais adiante no livro.

A programação dinâmica se aplica tipicamente a problemas de otimização em que uma série de escolhas deve ser feita, a fim de se alcançar uma solução ótima. À medida que as escolhas são feitas, freqüentemente surgem subproblemas da mesma forma. A programação dinâmica é eficaz quando um dado subproblema pode surgir a partir de mais de um conjunto parcial de escolhas; a técnica chave é armazenar a solução para cada um desses subproblemas, prevendo-se a hipótese de ele reaparecer. O Capítulo 15 mostra como essa idéia simples pode às vezes transformar facilmente algoritmos de tempo exponencial em algoritmos de tempo polinomial.

Como os algoritmos de programação dinâmica, os algoritmos gulosos em geral se aplicam a problemas de otimização em que diversas escolhas devem ser feitas, a fim de se chegar a uma solução ótima. A idéia de um algoritmo guloso é fazer cada escolha de uma maneira ótima para as condições locais. Um exemplo simples é a troca de moedas: para minimizar o número de moedas necessárias para trocar uma quantia dada, basta selecionar repetidamente a moeda de maior denominação (valor de face) que não seja maior que a quantia ainda devida. Existem muitos problemas desse tipo para os quais uma abordagem gulosa fornece uma solução ótima muito mais depressa que uma abordagem de programação dinâmica. Porém, nem sempre é fácil dizer se uma abordagem gulosa será efetiva. O Capítulo 16 examina a teoria dos matróides, que pode com freqüência ser útil para se fazer tal determinação.

A análise amortizada é uma ferramenta para análise de algoritmos que executam uma seqüência de operações semelhantes. Em vez de limitar o custo da seqüência de operações limitando o custo real de cada operação separadamente, uma análise amortizada pode ser usada

para fornecer um limite sobre o custo real da seqüência inteira. Uma razão pela qual essa idéia talvez seja efetiva é que pode ser impossível, em uma seqüência de operações, executar todas as operações individuais em seus limites de tempo do pior caso conhecido. Enquanto algumas operações são dispendiosas, muitas outras poderiam ser econômicas. Porém, a análise amortizada não é apenas uma ferramenta de análise; ela também é um modo de pensar sobre o projeto de algoritmos, pois o projeto de um algoritmo e a análise de seu tempo de execução muitas vezes estão intimamente relacionados. O Capítulo 17 apresenta três caminhos equivalentes para se realizar uma análise amortizada de um algoritmo.

---

## *Capítulo 15*

# *Programação dinâmica*

A programação dinâmica, como o método de dividir e conquistar, resolve problemas combinando as soluções para subproblemas. (Nesse contexto, o termo “programação” se refere a um método tabular, não ao processo de escrita de código de computador.) Como vimos no Capítulo 2, os algoritmos de dividir e conquistar particionam o problema em subproblemas independentes, resolvem os subproblemas recursivamente, e então combinam suas soluções para resolver o problema original. Em contraste, a programação dinâmica é aplicável quando os subproblemas não são independentes, isto é, quando os subproblemas compartilham subsubproblemas. Nesse contexto, um algoritmo de dividir e conquistar trabalha mais que o necessário, resolvendo repetidamente os subsubproblemas comuns. Um algoritmo de programação dinâmica resolve cada subsubproblema uma vez só e então grava sua resposta em uma tabela, evitando assim o trabalho de recalcular a resposta toda vez que o subsubproblema é encontrado.

A programação dinâmica em geral é aplicada a *problemas de otimização*. Em tais problemas, pode haver muitas soluções possíveis. Cada solução tem um valor, e desejamos encontrar uma solução com um valor ótimo (mínimo ou máximo). Chamamos tal solução *uma* solução ótima para o problema, em lugar de a chamarmos *a* solução ótima, pois podem existir várias soluções que alcançam o valor ótimo.

O desenvolvimento de um algoritmo de programação dinâmica pode ser desmembrado em uma seqüência de quatro etapas.

1. Caracterizar a estrutura de uma solução ótima.
2. Definir recursivamente o valor de uma solução ótima.
3. Calcular o valor de uma solução ótima em um processo de baixo para cima (bottom-up).
4. Construir uma solução ótima a partir de informações calculadas.

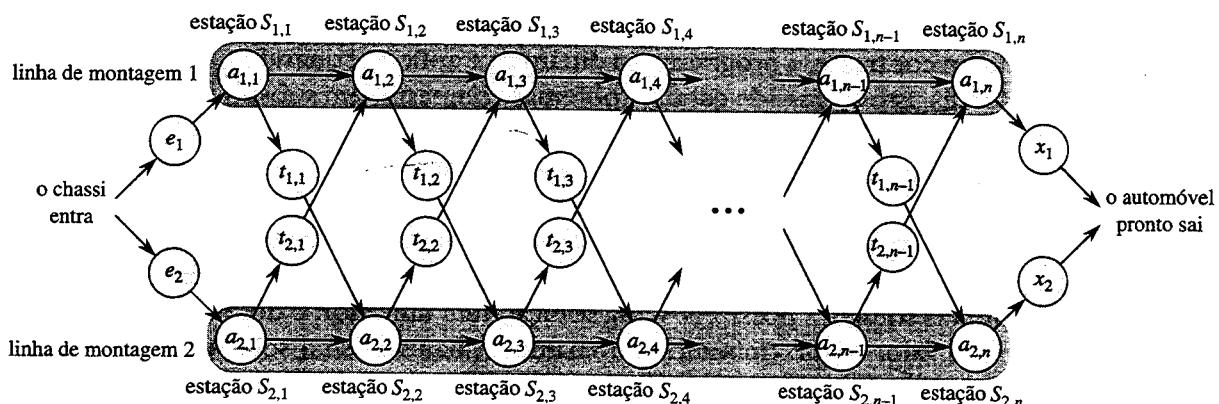
As Etapas 1 a 3 formam a base de uma solução de programação dinâmica para um problema. A Etapa 4 pode ser omitida se apenas o valor de uma solução ótima é exigido. Quando executamos a Etapa 4, às vezes mantemos informações adicionais durante a computação na Etapa 3, a fim de facilitar a construção de uma solução ótima.

As seções a seguir usam o método de programação dinâmica para resolver alguns problemas de otimização. A Seção 15.1 examina um problema na programação de duas linhas de montagem de automóveis onde, após cada estação, o automóvel em construção pode ficar na mesma linha ou passar para a outra. A Seção 15.2 pergunta como podemos multiplicar uma cadeia de matrizes de forma que seja executado o menor número total de multiplicações escalares. Dados esses exem-

ulos de programação dinâmica, a Seção 15.3 descreve duas características fundamentais que um problema deve ter para que a programação dinâmica seja uma técnica de solução viável. A Seção 15.4 mostra então como encontrar a subseqüência comum mais longa de duas seqüências. Finalmente, a Seção 15.5 utiliza a programação dinâmica para construir árvores de pesquisa binária que são ótimas, dada uma distribuição conhecida de chaves a serem examinadas.

## 15.1 Programação de linha de montagem

Nosso primeiro exemplo de programação dinâmica resolve um problema industrial. A Colonel Motors Corporation produz automóveis em uma fábrica que tem duas linhas de montagem, mostradas na Figura 15.1. Um chassi de automóvel entra em cada linha de montagem, tem as peças adicionadas a ele em uma série de estações, e um automóvel pronto sai no final da linha. Cada linha de montagem tem  $n$  estações, numeradas com  $j = 1, 2, \dots, n$ . Denotamos o  $j$ -ésima estação na linha  $i$  (onde  $i$  é 1 ou 2) por  $S_{i,j}$ . A  $j$ -ésima estação na linha 1 ( $S_{1,j}$ ) executa a mesma função que a  $j$ -ésima estação na linha 2 ( $S_{2,j}$ ). Porém, as estações foram construídas em épocas diferentes e com tecnologias diferentes, de forma que o tempo exigido em cada estação varia, até mesmo entre as estações na mesma posição nas duas linhas. Denotamos o tempo de montagem exigido na estação  $S_{i,j}$  por  $a_{i,j}$ . Conforme mostra a Figura 15.1, um chassi entra na estação 1 de uma das linhas de montagem e avança de cada estação para a seguinte. Também há um tempo de entrada  $e_i$  para o chassi entrar na linha de montagem  $i$ , e um tempo de saída  $x_i$  para o automóvel concluído sair da linha de montagem  $i$ .



**FIGURA 15.1** O problema industrial de encontrar o caminho mais rápido em uma fábrica. Há duas linhas de montagem, cada uma com  $n$  estações; a  $j$ -ésima estação na linha  $i$  é denotada por  $S_{i,j}$  e o tempo de montagem nessa estação é  $a_{i,j}$ . Um chassi de automóvel entra na fábrica e vai para a linha  $i$  (onde  $i = 1$  ou  $2$ ), demorando o tempo  $e_i$ . Depois de passar pela  $j$ -ésima estação em uma linha, o chassi vai para a  $(j + 1)$ -ésima estação de uma das linhas. Não há nenhum custo de transferência se ele ficar na mesma linha, mas demora o tempo  $t_{i,j}$  transferir o automóvel para a outra linha após a estação  $S_{i,j}$ . Depois de sair da  $n$ -ésima estação em uma linha, decorre o tempo  $x_i$  para o automóvel concluído sair da fábrica. O problema é determinar que estações escolher na linha 1 e quais escolher na linha 2 para minimizar o tempo total de passagem de um automóvel pela fábrica.

Normalmente, uma vez que um chassi entra em uma linha de montagem, ele percorre apenas essa linha. O tempo para ir de uma estação à seguinte dentro da mesma linha de montagem é desprezível. Às vezes, chega um pedido especial de urgência, e o cliente quer que o automóvel seja fabricado o mais rápido possível. Para pedidos de urgência, o chassi passa ainda pelas  $n$  estações em ordem, mas o gerente da fábrica pode passar o automóvel parcialmente concluído de uma linha de montagem para a outra após qualquer estação. O tempo para transferir um chassi da linha de montagem  $i$  depois da passagem pela estação  $S_{i,j}$  é  $t_{i,j}$ , onde  $i = 1, 2$  e  $j = 1, 2, \dots, n - 1$ .

(pois, após a  $n$ -ésima estação, a montagem se completa). O problema é determinar que estações escolher na linha 1 e quais escolher na linha 2 para minimizar o tempo total de passagem de um único automóvel pela fábrica. No exemplo da Figura 15.2(a), o tempo total mais rápido resulta da escolha das estações 1, 3 e 6 da linha 1 e das estações 2, 4 e 5 da linha 2.

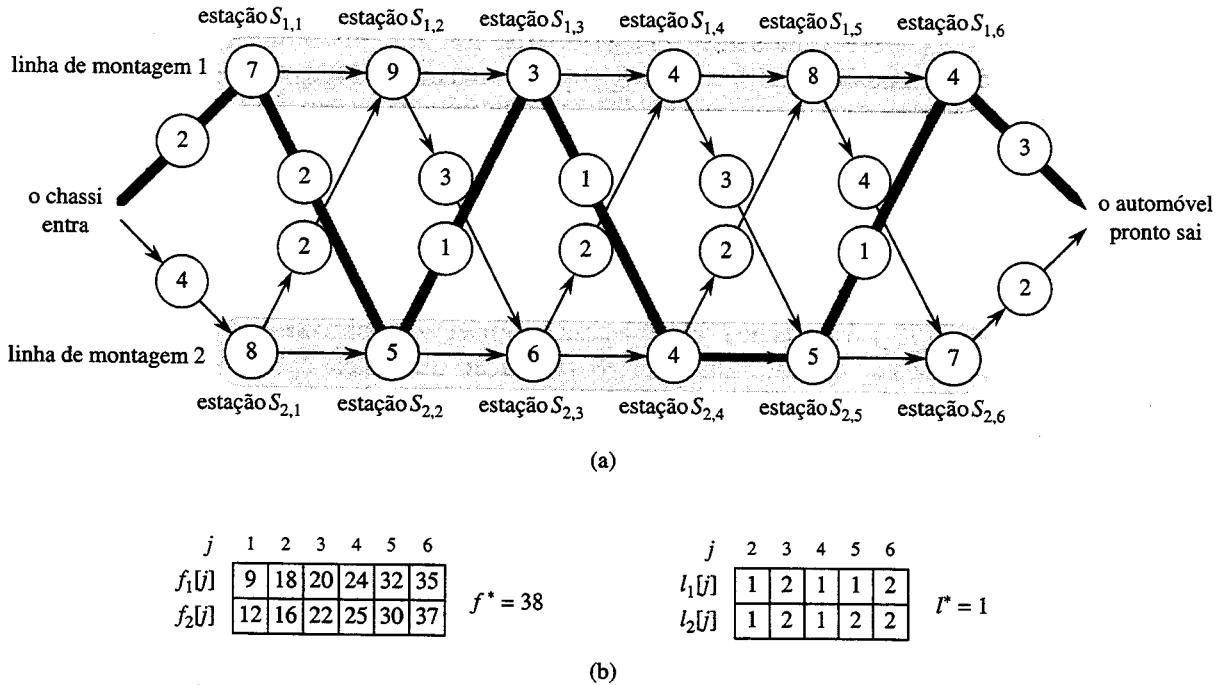


FIGURA 15.2 (a) Uma instância do problema de linha de montagem com custos  $e_i$ ,  $a_{i,j}$ ,  $t_{i,j}$  e  $x_i$  indicados. O caminho fortemente sombreado indica a passagem mais rápida pela fábrica. (b) Os valores de  $f_i[j]$ ,  $f^*$ ,  $l_i[j]$  e  $l^*$  para a instância da parte (a)

A maneira óbvia, de “força bruta”, de minimizar o tempo de passagem pela fábrica é inviável quando existem muitas estações. Se recebemos uma lista de quais estações usar na linha 1 e quais usar na linha 2, é fácil calcular no tempo  $\Theta(n)$  quanto tempo um chassi demora para passar pela fábrica. Infelizmente, há  $2^n$  maneiras possíveis de escolher estações, o que observamos visualizando o conjunto de estações usadas na linha 1 como um subconjunto de  $\{1, 2, \dots, n\}$  e notando que existem  $2^n$  subconjuntos. Desse modo, a determinação do caminho mais rápido pela fábrica enumerando todos os modos possíveis e calculando quanto tempo cada um deles demora exigiria o tempo  $\Omega(2^n)$ , que é inviável quando  $n$  é grande.

### Etapa 1: A estrutura do caminho mais rápido pela fábrica

A primeira etapa do paradigma de programação dinâmica é caracterizar a estrutura de uma solução ótima. Para o problema de programação de linha de montagem, podemos executar essa etapa como a seguir. Vamos considerar o modo mais rápido possível para um chassi seguir desde o ponto de partida passando pela estação  $S_{1,j}$ . Se  $j = 1$ , só existe um caminho que o chassi poderia ter seguido, e é fácil descobrir quanto tempo ele demora para passar pela estação  $S_{1,j}$ . Contudo, para  $j = 2, 3, \dots, n$ , há duas opções: o chassi poderia vir da estação  $S_{1,j-1}$  e depois seguir diretamente para a estação  $S_{1,j}$ , sendo desprezível o tempo para ir da estação  $j - 1$  para a estação  $j$  na mesma linha. Como alternativa, o chassi poderia vir da estação  $S_{2,j-1}$ , e depois ser transferido para a estação  $S_{1,j}$ , sendo o tempo de transferência  $t_{2,j-1}$ . Vamos considerar essas duas possibilidades separadamente, mas veremos que elas têm muito em comum.

Primeiro, vamos supor que o caminho mais rápido passando pela estação  $S_{1,j}$  seja pela estação  $S_{1,j-1}$ . A observação fundamental é que o chassi deve ter tomado um caminho mais rápido desde o ponto de partida e através da estação  $S_{1,j-1}$ . Por quê? Se houvesse um caminho mais rápido através da estação  $S_{1,j-1}$ , poderíamos utilizar esse caminho mais rápido como substituto para produzir um caminho mais rápido pela estação  $S_{1,j}$ : uma contradição.

De modo semelhante, vamos supor agora que a passagem mais rápida pela estação  $S_{1,j}$  seja feita pela estação  $S_{2,j-1}$ . Observamos que o chassi deve ter tomado um caminho mais rápido desde o ponto de partida, passando pela estação  $S_{2,j-1}$ . O raciocínio é o mesmo: se existisse um caminho mais rápido passando pela estação  $S_{2,j-1}$ , poderíamos utilizar esse caminho mais rápido como substituto para produzir uma passagem mais rápida através da estação  $S_{1,j}$ , o que seria uma contradição.

Em termos mais gerais, podemos dizer que, no caso da programação da linha de montagem, uma solução ótima para um problema (encontrar o caminho mais rápido passando pela estação  $S_{i,j}$ ) contém em seu interior uma solução ótima para subproblemas (encontrar a passagem mais rápida por  $S_{1,j-1}$  ou  $S_{2,j-1}$ ). Vamos nos referir a essa propriedade como **subestrutura ótima**, e ela é uma das indicações da aplicabilidade da programação dinâmica, como veremos na Seção 15.3.

Usamos a subestrutura ótima para mostrar que podemos construir uma solução ótima para um problema a partir de soluções ótimas para subproblemas. No caso da programação da linha de montagem, o raciocínio é dado a seguir. Se examinarmos um caminho mais rápido através da estação  $S_{1,j}$ , ele tem de passar pela estação  $j-1$  na linha 1 ou na linha 2. Desse modo, o caminho mais rápido pela estação  $S_{1,j}$  é:

- O caminho mais rápido pela estação  $S_{1,j-1}$ , e depois diretamente pela estação  $S_{1,j}$ .
- O caminho mais rápido pela estação  $S_{2,j-1}$ , uma transferência da linha 2 para a linha 1, e depois através da estação  $S_{1,j}$ .

Usando o raciocínio simétrico, o caminho mais rápido através da estação  $S_{2,j}$  é:

- O caminho mais rápido pela estação  $S_{2,j-1}$ , e depois diretamente pela estação  $S_{2,j}$ .
- O caminho mais rápido pela estação  $S_{1,j-1}$ , uma transferência da linha 1 para a linha 2, e depois através da estação  $S_{2,j}$ .

Para resolver o problema de encontrar o caminho mais rápido através da estação  $j$  de uma das linhas, resolvemos os subproblemas de encontrar os caminhos mais rápidos pela estação  $j-1$  em ambas as linhas.

Desse modo, podemos elaborar uma solução ótima para uma instância do problema de programação da linha de montagem pela elaboração de soluções ótimas para subproblemas.

## Etapa 2: Uma solução recursiva

A segunda etapa do paradigma de programação dinâmica é definir recursivamente o valor de uma solução ótima em termos das soluções ótimas para subproblemas. No caso do problema de programação da linha de montagem, escolhemos como nossos subproblemas os problemas de encontrar o caminho mais rápido pela estação  $j$  em ambas as linhas, para  $j = 1, 2, \dots, n$ . Seja  $f_i[j]$  o tempo mais rápido possível para levar um chassi desde o ponto de partida até a estação  $S_{i,j}$ .

Nosso objetivo final é descobrir o tempo mais rápido para levar um chassi por todo o percurso na fábrica, que denotamos por  $f^*$ . O chassi tem de passar por todo o caminho até a estação  $n$  na linha 1 ou na linha 2, e depois até a saída da fábrica. Tendo em vista que o mais rápido desses caminhos é o caminho mais rápido através da fábrica inteira, temos

$$262 \quad | \quad f^* = \min(f_1[n] + x_1, f_2[n] + x_2). \quad (15.1)$$

Também é fácil raciocinar sobre  $f_1[1]$  e  $f_2[1]$ . Para chegar até a estação 1 em qualquer linha, basta um chassi ir diretamente até essa estação. Assim,

$$f_1[1] = e_1 + \alpha_{1,1}, \quad (15.2)$$

$$f_2[1] = e_2 + \alpha_{2,1}. \quad (15.3)$$

Agora, vamos considerar como calcular  $f_i[j]$  para  $j = 2, 3, \dots, n$  (e  $i = 1, 2$ ). Concentrando nossa atenção em  $f_1[j]$ , lembramos que o caminho mais rápido pela estação  $S_{1,j}$  é o caminho mais rápido pela estação  $S_{1,j-1}$ , e depois diretamente pela estação  $S_{1,j}$ , ou o caminho mais rápido pela estação  $S_{2,j-1}$ , uma transferência da linha 2 para a linha 1, e depois pela estação  $S_{1,j}$ . No primeiro caso, temos  $f_1[j] = f_1[j-1] + \alpha_{1,j}$  e, no último caso,  $f_1[j] = f_2[j-1] + t_{2,j-1} + \alpha_{1,j}$ . Desse modo,

$$f_1[j] = \min(f_1[j-1] + \alpha_{1,j}, f_2[j-1] + t_{2,j-1} + \alpha_{1,j}). \quad (15.4)$$

para  $j = 2, 3, \dots, n$ . Simetricamente, temos

$$f_2[j] = \min(f_2[j-1] + \alpha_{2,j}, f_1[j-1] + t_{1,j-1} + \alpha_{2,j}) \quad (15.5)$$

para  $j = 2, 3, \dots, n$ . Combinando as equações (15.2) a (15.5), obtemos as equações recursivas

$$f_1[j] = \begin{cases} e_1 + \alpha_{1,1} & \text{se } j = 1, \\ \min(f_1[j-1] + \alpha_{1,j}, f_2[j-1] + t_{2,j-1} + \alpha_{1,j}) & \text{se } j \geq 2 \end{cases} \quad (15.6)$$

$$f_2[j] = \begin{cases} e_2 + \alpha_{2,1} & \text{se } j = 1, \\ \min(f_2[j-1] + \alpha_{2,j}, f_1[j-1] + t_{1,j-1} + \alpha_{2,j}) & \text{se } j \geq 2 \end{cases} \quad (15.7)$$

A Figura 15.2(b) mostra os  $f_i[j]$  valores para o exemplo da parte (a), calculados pelas equações (15.6) e (15.7), juntamente com o valor de  $f^*$ .

Os  $f_i[j]$  valores fornecem os valores de soluções ótimas para subproblemas. Para nos ajudar a acompanhar a construção de uma solução ótima, vamos definir  $l_i[j]$  como o número da linha, 1 ou 2, cuja estação  $j-1$  é usada em um caminho mais rápido pela estação  $S_{i,j}$ . Aqui,  $i = 1, 2$  e  $j = 2, 3, \dots, n$ . (Evitamos definir  $l_i[1]$  porque nenhuma estação precede a estação 1 em qualquer linha.) Também definimos  $l^*$  como a linha cuja estação  $n$  é usada em um caminho mais rápido pela fábrica inteira. Os  $l_i[j]$  valores nos ajudam a acompanhar um caminho mais rápido. Usando os valores de  $l^*$  e  $l_i[j]$  mostrados na Figura 15.2(b), seria possível acompanhar um caminho mais rápido pela fábrica mostrada na parte (a) como a seguir. Começando com  $l^* = 1$ , usamos a estação  $S_{1,6}$ . Agora, examinamos  $l_1[6]$ , que é 2, e então usamos a estação  $S_{2,5}$ . Continuando, examinamos  $l_2[5] = 2$  (usamos a estação  $S_{2,4}$ ),  $l_2[4] = 1$  (estação  $S_{1,3}$ ),  $l_1[3] = 2$  (estação  $S_{2,2}$ ) e  $l_2[2] = 1$  (estação  $S_{1,1}$ ).

### Etapa 3: Cálculo dos tempos mais rápidos

Nesse ponto, seria uma simples questão de escrever um algoritmo recursivo baseado na equação (15.1) e nas recorrências (15.6) e (15.7) para calcular o caminho mais rápido pela fábrica. Existe um problema com tal algoritmo recursivo: seu tempo de execução é exponencial em  $n$ . Para ver por que, seja  $r_i(j)$  o número de referências feitas a  $f_i[j]$  em um algoritmo recursivo. A partir da equação (15.1), temos

$$r_1(n) = r_2(n) = 1 . \quad (15.8)$$

Pelas recorrências (15.6) e (15.7), temos

$$r_1(j) = r_2(j) = r_1(j + 1) + r_2(j + 1) \quad (15.9)$$

para  $j = 1, 2, \dots, n - 1$ . Como o Exercício 15.1-2 lhe pede para mostrar,  $r_i(j) = 2^{n-j}$ . Portanto,  $f_1[1]$  sozinho é referenciado  $2^{n-1}$  vezes! Conforme o Exercício 15.1-3 lhe pede para mostrar, o número total de referências a todos os  $f_i[j]$  valores é  $\Theta(2^n)$ .

Podemos fazer muito melhor se calcularmos os  $f_i[j]$  valores em uma ordem diferente do modo recursivo. Observe que, para  $j \geq 2$ , cada valor de  $f_i[j]$  depende apenas dos valores de  $f_1[j-1]$  e  $f_2[j-1]$ . Calculando os  $f_i[j]$  valores na ordem de números de estação  $j$  crescentes – da esquerda para a direita na Figura 15.2(b) – podemos calcular o caminho mais rápido pela fábrica e o tempo que ele demora, no tempo  $\Theta(n)$ . O procedimento FASTEST-WAY toma como entrada os valores  $a_{i,j}$ ,  $t_{i,j}$ ,  $e_i$  e  $x_i$ , bem como  $n$ , o número de estações em cada linha de montagem.

**FASTEST-WAY( $a, t, e, x, n$ )**

```

1   $f_1[1] \leftarrow e_1 + a_{1,1}$ 
2   $f_2[1] \leftarrow e_2 + a_{2,1}$ 
3  for  $j \leftarrow 2$  to  $n$ 
4    do if  $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
5      then  $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$ 
6       $l_1[j] \leftarrow 1$ 
7      else  $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
8       $l_1[j] \leftarrow 2$ 
9      if  $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
10     then  $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$ 
11      $l_2[j] \leftarrow 2$ 
12     else  $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
13      $l_2[j] \leftarrow 1$ 
14  if  $f_1[n] + x_1 \leq f_2[n] + x_2$ 
15    then  $f^* = f_1[n] + x_1$ 
16     $l^* = 1$ 
17  else  $f^* = f_2[n] + x_2$ 
18     $l^* = 2$ 

```

FASTEST-WAY funciona como a seguir. As linhas 1 e 2 calculam  $f_1[1]$  e  $f_2[1]$  usando as equações (15.2) e (15.3). Em seguida, o loop **for** das linhas 3 a 13 calcula  $f_i[j]$  e  $l_i[j]$  para  $i = 1, 2$  e  $j = 2, 3, \dots, n$ . As linhas 4 a 8 calculam  $f_1[j]$  e  $l_1[j]$  usando a equação (15.4), e as linhas 9 a 13 calculam  $f_2[j]$  e  $l_2[j]$  usando a equação (15.5). Finalmente, as linhas 14 a 18 calculam  $f^*$  e  $l^*$  usando a equação (15.1). Como as linhas 1 e 2 e 4 a 18 demoram um tempo constante, e como cada uma das  $n - 1$  iterações do loop **for** das linhas 3 a 13 demora tempo constante, o procedimento inteiro demora o tempo  $\Theta(n)$ .

Um modo de visualizar o processo de computação dos valores de  $f_i[j]$  e  $l_i[j]$  é como se estivéssemos preenchendo entradas de tabelas. Consultando a Figura 15.2(b), preenchemos tabelas contendo valores  $f_i[j]$  e  $l_i[j]$  da esquerda para a direita (e de cima para baixo dentro de cada coluna). Para preencher uma entrada  $f_i[j]$ , precisamos dos valores de  $f_1[j-1]$  e  $f_2[j-1]$  e, sabendo que já calculamos e armazenamos esses valores, nós os descobrimos simplesmente observando a tabela.

## Etapa 4: Construção do caminho mais rápida pela fábrica

Tendo calculado os valores de  $f_i[j]$ ,  $f^*$ ,  $l_i[j]$  e  $l^*$ , precisamos construir a seqüência de estações usadas no caminho mais rápido pela fábrica. Já descrevemos como fazê-lo no exemplo da Figura 15.2.

O procedimento a seguir imprime as estações usadas, em ordem decrescente de número de estação. O Exercício 15.1-1 lhe pede para modificá-lo, de modo a imprimi-las em ordem crescente de número de estação.

```
PRINT-STATIONS( $l$ ,  $n$ )
1  $i \leftarrow l^*$ 
2 imprimir "linha "  $i$  ", estação "  $n$ 
3 for  $j \leftarrow n$  downto 2
4   do  $i \leftarrow l_i[j]$ 
5 imprimir "linha "  $i$  ", estação "  $j - 1$ 
```

No exemplo da Figura 15.2, PRINT-STATIONS produziria a saída

linha 1, estação 6  
linha 2, estação 5  
linha 2, estação 4  
linha 1, estação 3  
linha 2, estação 2  
linha 1, estação 1

## Exercícios

### 15.1-1

Mostre como modificar o procedimento PRINT-STATIONS para imprimir as estações em ordem crescente de número da estação. (Sugestão: Use recursão.)

### 15.1-2

Use as equações (15.8) e (15.9) e o método de substituição para mostrar que  $r_i(j)$ , o número de referências feitas a  $f_i[j]$  em um algoritmo recursivo, é igual a  $2^{n-j}$ .

### 15.1-3

Usando o resultado do Exercício 15.1-2, mostre que o número total de referências a todos os  $f_i[j]$  valores, ou  $\sum_{i=1}^2 \sum_{j=1}^n r_i(j)$ , é exatamente  $2^{n+1} - 2$ .

### 15.1-4

Juntas, as tabelas contendo  $f_i[j]$  e  $l_i[j]$  valores têm ao todo  $4n - 2$  entradas. Mostre como reduzir os requisitos de espaço para um total de  $2n + 2$  entradas, ao mesmo tempo que ainda calcula  $f^*$  e ainda é possível imprimir todas as estações em um caminho mais rápido pela fábrica.

### 15.1-5

O professor Canty supõe que possam existir alguns valores  $e_i$ ,  $a_{i,j}$  e  $t_{i,j}$  para os quais FASTEST-WAY produz  $l_i[j]$  valores tais que  $l_1[j] = 2$  e  $l_2[j] = 1$  para algum número de estação  $j$ . Supondo-se que todos os custos de transferência  $t_{i,j}$  sejam não negativos, mostre que o professor está errado.

## 15.2 Multiplicação de cadeias de matrizes

Nosso próximo exemplo de programação dinâmica é um algoritmo que resolve o problema de multiplicação de cadeias de matrizes. Recebemos uma seqüência (cadeia)  $\langle A_1, A_2, \dots, A_n \rangle$  de  $n$  matrizes a serem multiplicadas e desejamos calcular o produto

$$A_1 A_2 \dots A_n . \quad (15.10)$$

Podemos avaliar a expressão (15.10) usando o algoritmo padrão para multiplicação de pares de matrizes como uma sub-rotina, uma vez que o tenhamos colocado entre parênteses para solucionar todas as ambigüidades no modo como as matrizes são multiplicadas entre si. Um produto de matrizes é **completamente colocado entre parênteses** se ele for uma única matriz ou o produto de dois produtos de matrizes completamente colocados entre parênteses, cercado por parênteses. A multiplicação de matrizes é associativa, e assim todas as colocações entre parênteses resultam no mesmo produto. Por exemplo, se a cadeia de matrizes é  $\langle A_1, A_2, A_3, A_4 \rangle$ , o produto  $A_1 A_2 A_3 A_4$  pode ser completamente colocado entre parênteses de cinco modos distintos:

$$\begin{aligned} & (A_1(A_2(A_3A_4))) , \\ & (A_1((A_2A_3)A_4)) , \\ & ((A_1A_2)(A_3A_4)) , \\ & ((A_1(A_2A_3))A_4) , \\ & (((A_1A_2)A_3)A_4) . \end{aligned}$$

O modo como uma cadeia de matrizes é colocada entre parênteses pode ter um impacto dramático sobre o custo de avaliação do produto. Considere primeiro o custo de multiplicar duas matrizes. O algoritmo padrão é dado pelo pseudocódigo a seguir. Os atributos *linhas* e *colunas* são os números de linhas e colunas em uma matriz.

```
MATRIX-MULTIPLY(A, B)
1 if colunas[A] ≠ linhas[B]
2   then error “dimensões incompatíveis”
3 else for i ← 1 to linhas[A]
4   do for j ← 1 to colunas[B]
5     do C[i, j] ← 0
6     for k ← 1 to colunas[A]
7       do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8   return C
```

Podemos multiplicar duas matrizes *A* e *B* somente se elas forem **compatíveis**: o número de colunas de *A* deve ser igual ao número de linhas de *B*. Se *A* é uma matriz  $p \times q$  e *B* é uma matriz  $q \times r$ , a matriz resultante *C* é uma matriz  $p \times r$ . O tempo para calcular *C* é dominado pelo número de multiplicações escalares na linha 7, que é  $pqr$ . No texto a seguir, iremos expressar os custos em termos do número de multiplicações escalares.

Para ilustrar os diferentes custos resultantes da colocação dos parênteses de um produto de matrizes de maneiras distintas, considere o problema de uma cadeia  $\langle A_1, A_2, A_3 \rangle$  de três matrizes. Suponha que as dimensões das matrizes sejam  $10 \times 100$ ,  $100 \times 5$  e  $5 \times 50$ , respectivamente. Se multiplicarmos as matrizes de acordo com a colocação dos parênteses  $((A_1A_2)A_3)$ , executaremos  $10 \cdot 100 \cdot 5 = 5.000$  multiplicações escalares para calcular o produto de matrizes  $A_1A_2$   $10 \times 5$ , mais outras  $10 \cdot 5 \cdot 50 = 2.500$  multiplicações escalares para multiplicar essa matriz por  $A_3$ , pro-

duzindo um total de 7.500 multiplicações escalares. Se, em vez disso, multiplicarmos de acordo com a colocação dos parênteses ( $A_1(A_2A_3)$ ), executamos  $100 \cdot 5 \cdot 50 = 25.000$  multiplicações escalares para calcular o produto de matrizes  $A_2A_3$ ,  $100 \times 50$ , mais outras  $10 \cdot 100 \cdot 50 = 50.000$  multiplicações escalares para multiplicar  $A_1$  por essa matriz, dando um total de 75.000 multiplicações escalares. Desse modo, o cálculo do produto de acordo com a primeira colocação dos parênteses é 10 vezes mais rápido.

O problema de multiplicação de cadeias de matrizes pode ser enunciado da forma a seguir: dada uma cadeia  $\langle A_1, A_2, \dots, A_n \rangle$  de  $n$  matrizes na qual, para  $i = 1, 2, \dots, n$ , a matriz  $A_i$  tem dimensão  $p_{i-1} \times p_i$ , coloque completamente entre parênteses o produto  $A_1A_2 \dots A_n$  de um modo que minimize o número de multiplicações escalares.

Observe que, no problema de multiplicação de cadeias de matrizes, não estamos realmente multiplicando matrizes. Nossa meta é apenas determinar uma ordem para multiplicar matrizes que tenha o custo mais baixo. Em geral, o tempo investido para determinar essa ordem ótima é mais que compensado pelo tempo economizado mais tarde, quando as multiplicações de matrizes são de fato executadas (por exemplo, executando apenas 7.500 multiplicações escalares em vez de 75.000).

### Contagem do número de colocações entre parênteses

Antes de resolver o problema de multiplicação de cadeias de matrizes por programação dinâmica, devemos nos convencer de que a verificação exaustiva de todas as possíveis colocações entre parênteses não resulta em um algoritmo eficiente. Vamos representar por  $P(n)$  o número de alternativas para colocação dos parênteses de uma seqüência de  $n$  matrizes. Quando  $n = 1$ , há apenas uma matriz e portanto somente um modo de colocar totalmente entre parênteses o produto de matrizes. Quando  $n \geq 2$ , um produto de matrizes totalmente entre parênteses é o produto de dois subprodutos de matrizes totalmente entre parênteses, e a divisão entre os dois subprodutos pode ocorrer entre a  $k$ -ésima e a  $(k + 1)$ -ésima matrizes para qualquer  $k = 1, 2, \dots, n - 1$ . Desse modo, obtemos a recorrência

$$P(n) = \begin{cases} 1 & \text{se } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{se } n \geq 2. \end{cases} \quad (15.11)$$

O Problema 12-4 lhe pediu para mostrar que a solução para uma recorrência semelhante é a seqüência de **números catalães**, que cresce como  $\Omega(4^n/n^{3/2})$ . Um exercício mais simples (consulte o Exercício 15.2-3) é mostrar que a solução para a recorrência (15.11) é  $\Omega(2^n)$ . O número de soluções é, portanto, exponencial em  $n$  e, consequentemente, o método de força bruta para pesquisa exaustiva é uma estratégia pobre para se determinar a colocação ótima dos parênteses de uma cadeia de matrizes.

### Etapa 1: A estrutura de uma colocação ótima dos parênteses

A primeira etapa do paradigma de programação dinâmica é encontrar a subestrutura ótima, e depois usá-la para construir uma solução ótima para o problema a partir de soluções ótimas para subproblemas. Para o problema de multiplicação de cadeias de matrizes, podemos executar essa etapa como a seguir. Por conveniência, vamos adotar a notação  $A_{i..j}$ , onde  $i \leq j$  para a matriz que resulta da avaliação do produto  $A_iA_{i+1} \dots A_j$ . Observe que, se o problema é não trivial, isto é, se  $i < j$ , qualquer colocação ótima dos parênteses do produto  $A_iA_{i+1} \dots A_j$  deve dividir o produto entre  $A_k$  e  $A_{k+1}$  para algum inteiro  $k$  no intervalo  $i \leq k < j$ . Ou seja, para algum valor de  $k$ , primeiro calculamos as matrizes  $A_{i..k}$  e  $A_{k+1..j}$ , e depois multiplicamos os dois para gerar o produto final  $A_{i..j}$ . O custo dessa colocação dos parênteses é portanto o custo de calcular a matriz  $A_{i..k}$ , mais o custo de calcular  $A_{k+1..j}$ , mais o custo de multiplicá-las uma pela outra.

A subestrutura ótima desse problema é dada a seguir. Suponha que uma colocação dos parênteses de  $A_i A_{i+1} \dots A_j$  divida o produto entre  $A_k$  e  $A_{k+1}$ . Então, a colocação dos parênteses da subcadeia “prefixo”  $A_i A_{i+1} \dots A_k$  dentro dessa colocação ótima dos parênteses de  $A_i A_{i+1} \dots A_j$  deve ser uma colocação ótima dos parênteses de  $A_i A_{i+1} \dots A_k$ . Por quê? Se existisse um modo menos dispendioso de colocar entre parênteses  $A_i A_{i+1} \dots A_k$ , a substituição dessa colocação dos parênteses na colocação ótima dos parênteses de  $A_i A_{i+1} \dots A_j$  produziria outra colocação dos parênteses de  $A_i A_{i+1} \dots A_j$  cujo custo seria mais baixo que o custo ótimo: uma contradição. Uma observação semelhante é válida para a colocação dos parênteses da subcadeia  $A_{k+1} A_{k+2} \dots A_j$  na colocação ótima dos parênteses de  $A_i A_{i+1} \dots A_j$ : ela deve ser uma colocação ótima dos parênteses de  $A_{k+1} A_{k+2} \dots A_j$ .

Agora usamos nossa subestrutura ótima para mostrar que podemos construir uma solução ótima para o problema a partir de soluções ótimas para subproblemas. Vimos que qualquer solução para uma instância não trivial do problema de multiplicação de cadeias de matrizes nos obriga a dividir o produto, e que qualquer solução ótima contém dentro dela soluções ótimas para instâncias de subproblemas. Desse modo, podemos construir uma solução ótima para uma instância do problema de multiplicação de cadeias de matrizes dividindo o problema em dois subproblemas (por meio da colocação ótima dos parênteses de  $A_i A_{i+1} \dots A_k$  e  $A_{k+1} A_{k+2} \dots A_j$ ), encontrando soluções ótimas para instâncias de subproblemas, e depois combinando essas soluções ótimas de subproblemas. Devemos assegurar que, quando procurarmos pelo lugar correto para dividir o produto, consideraremos todos os lugares possíveis, de forma a termos a garantia de ter examinado a opção ótima.

## Etapa 2: Uma solução recursiva

Em seguida, definimos recursivamente o custo de uma solução ótima em termos das soluções ótimas para subproblemas. No caso do problema de multiplicação de cadeias de matrizes, escolhemos como nossos subproblemas os problemas de determinar o custo mínimo de uma colocação dos parênteses de  $A_i A_{i+1} \dots A_j$  para  $1 \leq i \leq j \leq n$ . Seja  $m[i, j]$  o número mínimo de multiplicações escalares necessárias para calcular a matriz  $A_{i \dots j}$ ; para o problema completo, o custo de um caminho mais econômico para calcular  $A_{1 \dots n}$  seria portanto  $m[1, n]$ .

Podemos definir  $m[i, j]$  recursivamente como a seguir. Se  $i = j$ , o problema é trivial; a cadeia consiste em apenas uma matriz  $A_{i \dots i} = A_i$ , e assim nenhuma multiplicação escalar é necessária para calcular o produto. Desse modo,  $m[i, i] = 0$  para  $i = 1, 2, \dots, n$ . Para calcular  $m[i, j]$  quando  $i < j$ , tiramos proveito da estrutura de uma solução ótima da Etapa 1. Vamos supor que a colocação ótima dos parênteses divide o produto  $A_i A_{i+1} \dots A_j$  entre  $A_k$  e  $A_{k+1}$ , onde  $i \leq k < j$ . Então,  $m[i, j]$  é igual ao custo mínimo para calcular os subprodutos  $A_{i \dots k}$  e  $A_{k+1 \dots j}$ , mais o custo de multiplicar essas duas matrizes. Recordando que cada matriz  $A_i$  é  $p_{i-1} p_i$ , vemos que o cálculo do produto de matrizes  $A_{i \dots k} A_{k+1 \dots j}$  exige  $p_{i-1} p_k p_j$  multiplicações escalares. Desse modo, obtemos

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j.$$

Essa equação recursiva pressupõe que conhecemos o valor de  $k$ , o que não ocorre. Porém, existem apenas  $j - i$  valores possíveis para  $k$ , isto é,  $k = i, i+1, \dots, j-1$ . Como a colocação ótima dos parênteses deve usar um desses valores para  $k$ , precisamos apenas verificar todos eles para encontrar o melhor. Desse modo, nossa definição recursiva para o custo mínimo de colocar entre parênteses o produto  $A_i A_{i+1} \dots A_j$  se torna

$$m[i, j] = \begin{cases} 0 & \text{se } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{se } i < j. \end{cases} \quad (15.12)$$

Os valores de  $m[i, j]$  fornecem os custos de soluções ótimas para subproblemas. Para nos ajudar a controlar o modo de construir uma solução ótima, vamos definir  $s[i, j]$  como um valor de  $k$  no qual podemos dividir o produto  $A_i A_{i+1} \dots A_j$  para obter uma colocação ótima dos parênteses. Ou seja,  $s[i, j]$  é igual a um valor  $k$  tal que  $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$ .

### Etapa 3: Como calcular os custos ótimos

Neste ponto, é uma questão simples escrever um algoritmo recursivo baseado na recorrência (15.12) para calcular o custo mínimo  $m[1, n]$  para multiplicar  $A_1 A_2 \dots A_n$ . Entretanto, como vemos na Seção 15.3, esse algoritmo demora um tempo exponencial – ele não é nada melhor que o método da força bruta de verificar cada maneira de colocar o produto entre parênteses.

A observação importante que podemos fazer a essa altura é que temos relativamente poucos subproblemas: um problema para cada opção de  $i$  e  $j$  que satisfaz a  $1 \leq i \leq j \leq n$ , ou  $\binom{n}{2} + n = \Theta(n^2)$  no total. Um algoritmo recursivo pode encontrar cada subproblema muitas vezes em diferentes ramificações dessa árvore de recursão. Essa propriedade de superpor subproblemas é a segunda indicação da aplicabilidade da programação dinâmica (sendo a primeira indicação a subestrutura ótima).

Em vez de calcular recursivamente a solução para a recorrência (15.12), executamos a terceira etapa do paradigma de programação dinâmica e calculamos o custo ótimo usando uma abordagem tabular de baixo para cima. O pseudocódigo a seguir pressupõe que a matriz  $A_i$  tem dimensões  $p_{i-1} \times p_i$  para  $i = 1, 2, \dots, n$ . A entrada é uma seqüência  $p = \langle p_0, p_1, \dots, p_n \rangle$ , onde  $comprimento[p] = n + 1$ . O procedimento utiliza uma tabela auxiliar  $m[1..n, 1..n]$  para armazenar os custos de  $m[i, j]$  e uma tabela auxiliar  $s[1..n, 1..n]$  que registra qual índice de  $k$  alcançou o custo ótimo no cálculo de  $m[i, j]$ . Usaremos a tabela  $s$  para construir uma solução ótima.

Para implementar corretamente a abordagem de baixo para cima, devemos determinar que entradas da tabela são usadas para se calcular  $m[i, j]$ . A equação (15.12) mostra que o custo  $m[i, j]$  de calcular um produto de cadeias de matrizes para  $j - i + 1$  matrizes só depende dos custos de calcular os produtos de cadeias de matrizes de menos de  $j - i + 1$  matrizes. Isto é, para  $k = i, i + 1, \dots, j - 1$ , a matriz  $A_{i..k}$  é um produto de  $k - i + 1 < j - i + 1$  matrizes, e a matriz  $A_{k+1..j}$  é um produto de  $j - k < j - i + 1$  matrizes. Desse modo, o algoritmo deve preencher a tabela  $m$  de uma forma que corresponda a resolver o problema de colocação dos parênteses em cadeias de matrizes de comprimento crescente.

```

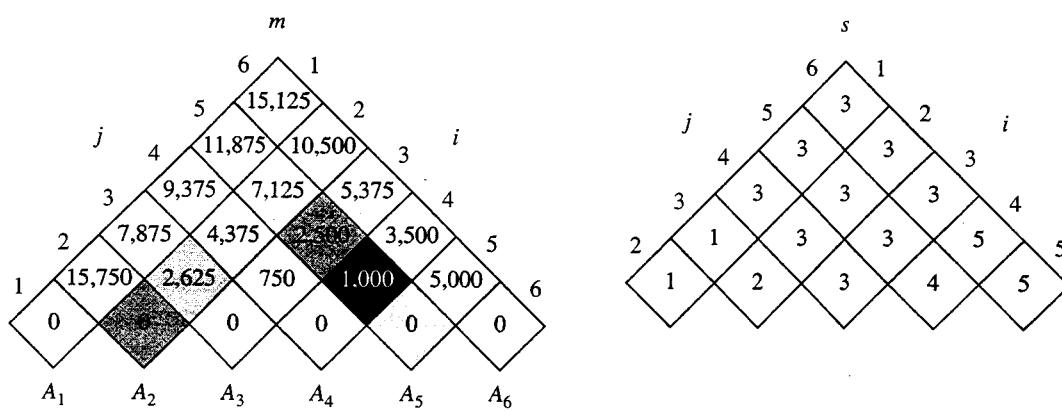
MATRIX-CHAIN-ORDER( $p$ )
1  $n \leftarrow comprimento[p] - 1$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   do  $m[i, i] \leftarrow 0$ 
4 for  $l \leftarrow 2$  to  $n$        $\triangleright l$  é o comprimento da cadeia
5   do for  $i \leftarrow 1$  to  $n - l + 1$ 
6     do  $j \leftarrow i + l - 1$ 
7        $m[i, j] \leftarrow \infty$ 
8       for  $k \leftarrow i$  to  $j - 1$ 
9         do  $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$ 
10        if  $q < m[i, j]$ 
11          then  $m[i, j] \leftarrow q$ 
12           $s[i, j] \leftarrow k$ 
13 return  $m$  e  $s$ 
```

O algoritmo calcula primeiro  $m[i, i] \leftarrow 0$  para  $i = 1, 2, \dots, n$  (os custos mínimos para cadeias de comprimento 1) nas linhas 2 e 3. Em seguida, ele usa a recorrência (15.12) para calcular  $m[i, i + 1]$  para  $i = 1, 2, \dots, n - 1$  (os custos mínimos para cadeias de comprimento  $l = 2$ ) durante a primeira execução do loop nas linhas 4 a 12. Na segunda passagem através do loop, ele calcula

$m[i, i + 2]$  para  $i = 1, 2, \dots, n - 2$  (os custos mínimos para cadeias de comprimento  $l = 3$ ) e assim por diante. Em cada etapa, o custo  $m[i, j]$  calculado nas linhas 9 a 12 depende apenas das entradas de tabela  $m[i, k]$  e  $m[k + 1, j]$  já calculadas.

A Figura 15.3 ilustra esse procedimento em uma cadeia de  $n = 6$  matrizes. Tendo em vista que definimos  $m[i, j]$  somente para  $i = j$ , apenas a porção da tabela  $m$  estritamente acima da diagonal principal é usada. A figura mostra a tabela girada para fazer a diagonal principal ficar disposta horizontalmente. A cadeia de matrizes é listada ao longo da parte inferior. Usando-se esse layout, o custo mínimo  $m[i, j]$  para multiplicar uma subcadeia de matrizes  $A_i A_{i+1} \dots A_j$  pode ser encontrado na interseção de linhas dispostas a nordeste de  $A_i$  e a noroeste de  $A_j$ . Cada linha horizontal na tabela contém as entradas para cadeias de matrizes do mesmo comprimento. MATRIX-CHAIN-ORDER calcula as linhas desde a parte inferior até a parte superior e da esquerda para a direita dentro de cada linha. Uma entrada  $m[i, j]$  é calculada usando-se os produtos  $p_{i-1} p_k p_j$  para  $k = i, i + 1, \dots, j - 1$  e todas as entradas a sudoeste e a sudeste de  $m[i, j]$ .

Uma inspeção simples da estrutura de loops aninhados de MATRIX-CHAIN-ORDER produz um tempo de execução igual a  $O(n^3)$  para o algoritmo. Os loops estão aninhados com profundidade três, e cada índice de loop ( $l$ ,  $i$  e  $k$ ) toma no máximo  $n$  valores. O Exercício 15.2-4 lhe pede para mostrar que o tempo de execução desse algoritmo também é de fato  $\Omega(n^3)$ . O algoritmo exige o espaço  $\Theta(n^2)$  para armazenar as tabelas  $m$  e  $s$ . Desse modo, MATRIX-CHAIN-ORDER é muito mais eficiente que o método de tempo exponencial de enumerar todas as possíveis colocações entre parênteses e verificar cada uma delas.



**FIGURA 15.3** As tabelas  $m$  e  $s$  calculadas por MATRIX-CHAIN-ORDER para  $n = 6$  e as dimensões de matrizes a seguir:

<b>matriz</b>	<b>dimensão</b>
$A_1$	$30 \times 35$
$A_2$	$35 \times 15$
$A_3$	$15 \times 5$
$A_4$	$5 \times 10$
$A_5$	$10 \times 20$
$A_6$	$20 \times 25$

As tabelas estão giradas de modo que a diagonal principal fique disposta horizontalmente. Somente a diagonal principal e o triângulo superior são usados na tabela  $m$ , e apenas o triângulo superior é usado na tabela  $s$ . O número mínimo de multiplicações escalares para multiplicar as 6 matrizes é  $m[1, 6] = 15.125$ . Das entradas mais escuras, os pares que têm o mesmo sombreado são tomados juntos na linha 9 quando se calcula.

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000, \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125, \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$

= 7125 .

#### Etapa 4: A construção de uma solução ótima

Embora MATRIX-CHAIN-ORDER determine o número ótimo de multiplicações escalares necessárias para calcular um produto de cadeias de matrizes, ele não mostra diretamente como multiplicar as matrizes. Não é difícil construir uma solução ótima a partir das informações calculadas armazenadas na tabela  $s[1..n, 1..n]$ . Cada entrada  $s[i, j]$  registra o valor de  $k$  tal que a colocação ótima dos parênteses de  $A_i A_{i+1} \dots A_j$  divide o produto entre  $A_k$  e  $A_{k+1}$ . Desse modo, sabemos que a multiplicação de matrizes final no cálculo ótimo de  $A_{1..n}$ , é  $A_{1..s[1, n]} A_{s[1, n]+1..n}$ . As multiplicações de matrizes anteriores podem ser calculadas recursivamente, pois  $s[1, s[1, n]]$  determina a última multiplicação de matrizes no cálculo de  $A_{1..s[1, n]}$ , e  $s[s[1, n] + 1, n]$  determina a última multiplicação de matrizes no cálculo de  $A_{s[1, n]+1..n}$ . O procedimento recursivo a seguir imprime uma colocação ótima dos parênteses de  $\langle A_i, A_{i+1}, \dots, A_j \rangle$ , dada a tabela  $s$  calculada por MATRIX-CHAIN-ORDER e os índices  $i$  e  $j$ . A chamada inicial PRINT-OPTIMAL-PARENS( $s, 1, n$ ) imprime uma colocação ótima dos parênteses de  $\langle A_1, A_2, \dots, A_n \rangle$ .

```
PRINT-OPTIMAL-PARENS( $s, i, j$ )
1 if  $i = j$ 
2 then print " $A$ " $i$ 
3 else print "("
4   PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )
5   PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )
6   print ")"
```

No exemplo da Figura 15.3, a chamada PRINT-OPTIMAL-PARENS( $s, 1, 6$ ) imprime a colocação dos parênteses  $((A_1 (A_2 A_3)) ((A_4 A_5) A_6))$ .

### Exercícios

#### 15.2-1

Encontre uma colocação ótima dos parênteses de um produto de cadeias de matrizes cuja seqüência de dimensões é  $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$ .

#### 15.2-2

Forneça um algoritmo recursivo MATRIX-CHAIN-MULTIPLY( $A, s, i, j$ ) que realmente execute a multiplicação ótima de cadeias de matrizes, dada a seqüência de matrizes  $\langle A_1, A_2, \dots, A_n \rangle$ , a tabela  $s$  calculada por MATRIX-CHAIN-ORDER e os índices  $i$  e  $j$ . (A chamada inicial seria MATRIX-CHAIN-MULTIPLY( $A, s, 1, n$ ).)

#### 15.2-3

Use o método de substituição para mostrar que a solução para a recorrência (15.11) é  $\Omega(2^n)$ .

#### 15.2-4

Seja  $R(i, j)$  o número de vezes que a entrada de tabela  $m[i, j]$  é referenciada durante o cálculo de outras entradas de tabela em uma chamada de MATRIX-CHAIN-ORDER. Mostre que o número total de referências para a tabela inteira é

$$\sum_{i=1}^n \sum_{j=i}^n R(i, j) = \frac{n^3 - n}{3}.$$

(*Sugestão:* Talvez você considere útil a equação (A.3).)

### 15.2-5

Mostre que uma colocação dos parênteses completa de uma expressão de  $n$  elementos tem exatamente  $n - 1$  pares de parênteses.

## 15.3 Elementos de programação dinâmica

Embora tenhamos acabado de analisar dois exemplos do método de programação dinâmica, talvez você ainda esteja tentando simplesmente imaginar quando o método deve se aplicar. Sob uma perspectiva de engenharia, quando devemos procurar uma solução de programação dinâmica para um problema? Nesta seção, examinamos os dois ingredientes fundamentais que um problema de otimização deve ter para que a programação dinâmica seja aplicável: subestrutura ótima e superposição de subproblemas. Também examinaremos uma variante do método, chamada memoização,<sup>1</sup> para aproveitarmos a propriedade de superposição de subproblemas.

### Subestrutura ótima

O primeiro passo para se resolver um problema de otimização por programação dinâmica é caracterizar a estrutura de uma solução ótima. Lembramos que um problema apresenta uma **subestrutura ótima** se uma solução ótima para o problema contém em seu interior soluções ótimas para subproblemas. Sempre que um problema apresenta uma subestrutura ótima, esse fato é uma boa indicação de que a programação dinâmica poderia se aplicar. (Porém, esse fato também poderia significar que uma estratégia gulosa seria aplicável. Consulte o Capítulo 16.) Em programação dinâmica, construímos uma solução ótima para o problema a partir de soluções ótimas para subproblemas. Conseqüentemente, devemos ter o cuidado de assegurar que o intervalo de subproblemas que consideramos inclui aqueles que são usados em uma solução ótima.

Descobrimos uma subestrutura ótima em ambos os problemas examinados neste capítulo até agora. Na Seção 15.1, observamos que o caminho mais rápido pela estação  $j$  de uma ou outra linha continha dentro dele o caminho mais rápido pela estação  $j - 1$  em uma linha. Na Seção 15.2, observamos que uma colocação ótima dos parênteses de  $A_i A_{i+1} \dots A_j$ , que divide o produto entre  $A_k$  e  $A_{k+1}$  contém dentro dela soluções ótimas para os problemas de colocação dos parênteses de  $A_i A_{i+1} \dots A_k$  e  $A_{k+1} A_{k+2} \dots A_j$ .

Você deverá seguir um padrão comum na descoberta de uma subestrutura ótima:

1. Mostrar que uma solução para o problema consiste em fazer uma escolha, como a escolha de uma estação precedente na linha de montagem ou a escolha de um índice no qual será dividida a cadeia de matrizes. Essa escolha deixa um ou mais subproblemas a serem resolvidos.
2. Supor que, para um dado problema, você tem a escolha que conduz a uma solução ótima. Não se preocupe ainda com a maneira de determinar essa escolha; simplesmente suponha que ela lhe foi dada.
3. Dada essa escolha, determinar quais subproblemas resultam dela e como caracterizar melhor o espaço de subproblemas resultante.

<sup>1</sup> Isso não é um erro de gráfia. A palavra realmente é *memoização*, não *memorização*. *Memoização* vem de *memo*, pois a técnica consiste em registrar um valor de tal forma que seja possível procurá-lo mais tarde.

\* Foi mantido nesta tradução o neologismo criado pelos autores na edição original em inglês; assim, também fizemos distinção entre “*memoização*” e *memorização*. (N.T.)

- Mostrar que as soluções para os subproblemas usados dentro da solução ótima para o problema devem elas próprias ser ótimas, usando uma técnica de “recortar e colar”. Isso é feito supondo-se que cada uma das soluções de subproblemas não é ótima, e então derivando uma contradição. Em particular, “recortando” a solução não ótima para o subproblema e “colando” a solução ótima, você mostra que pode conseguir uma solução melhor para o problema original, contradizendo assim sua hipótese de que você já tinha uma solução ótima. Se houver mais de um subproblema, em geral eles são tão semelhantes que o argumento de recortar e colar usado para um deles pode se ser modificado para os outros com pouco esforço.

Para caracterizar o espaço de subproblemas, uma boa regra prática é tentar manter o espaço tão simples quanto possível, e depois expandi-lo conforme necessário. Por exemplo, o espaço de subproblemas que consideramos para a programação da linha de montagem foi o caminho mais rápido a partir da entrada na fábrica pelas estações  $S_{1,j}$  e  $S_{2,j}$ . Esse espaço de subproblemas funcionou bem, e não havia necessidade de procurar um espaço de subproblemas mais geral.

Reciprocamente, vamos supor que tentássemos restringir nosso espaço de subproblemas para a multiplicação de cadeias de matrizes a produtos de matrizes da forma  $A_1 A_2 \dots A_j$ . Como antes, uma colocação ótima dos parênteses deve dividir esse produto entre  $A_k$  e  $A_{k+1}$  para algum  $1 \leq k \leq j$ . A menos que pudéssemos garantir que  $k$  sempre é igual a  $j - 1$ , descobriríamos que tínhamos subproblemas da forma  $A_1 A_2 \dots A_k$  e  $A_{k+1} A_{k+2} \dots A_j$ , e que o último subproblema não tem a forma  $A_1 A_2 \dots A_j$ . Para esse problema, era necessário permitir que nossos subproblemas variassem em “ambas as extremidades”, isto é, permitir que tanto  $i$  quanto  $j$  variassem no subproblema  $A_i A_{i+1} \dots A_j$ .

A subestrutura ótima varia nos domínios de problemas de duas maneiras:

- O número de subproblemas que são usados em uma solução ótima para o problema original.
- O número de opções que temos na determinação de qual(is) subproblema(s) usar em uma solução ótima.

Na programação da linha de montagem, uma solução ótima só utiliza um subproblema, mas devemos considerar duas opções para determinar uma solução ótima. Para encontrar o caminho mais rápido pela estação  $S_{i,j}$ , usamos o caminho mais rápido por  $S_{1,j-1}$  ou o caminho mais rápido por  $S_{2,j-1}$ ; qualquer das opções que usarmos representará o único subproblema que temos de resolver de maneira ótima. A multiplicação de cadeias de matrizes para a subcadeia  $A_i A_{i+1} \dots A_j$  serve como um exemplo com dois subproblemas e  $j - i$  opções. Para uma dada matriz  $A_k$  em que dividimos o produto, temos dois subproblemas – a colocação dos parênteses de  $A_i A_{i+1} \dots A_k$  e a colocação dos parênteses de  $A_{k+1} A_{k+2} \dots A_j$  – e devemos resolver *ambos* de maneira ótima. Uma vez que determinamos as soluções ótimas para subproblemas, escolhemos entre  $j - i$  candidatos para o índice  $k$ .

Informalmente, o tempo de execução de um algoritmo de programação dinâmica depende do produto de dois fatores: o número de subproblemas globais e quantas escolhas observamos para cada subproblema. Na programação de linha de montagem, tivemos  $\Theta(n)$  subproblemas globais e só duas escolhas a examinar para cada um, produzindo um tempo de execução  $\Theta(n)$ . No caso da multiplicação de cadeias de matrizes, havia  $\Theta(n^2)$  subproblemas globais e, em cada um deles, tivemos no máximo  $n - 1$  escolhas, dando um tempo de execução  $O(n^3)$ .

A programação dinâmica emprega a subestrutura ótima de baixo para cima. Isto é, primeiro encontramos soluções ótimas para subproblemas e, tendo resolvido os subproblemas, encontramos uma solução ótima para o problema. A descoberta de uma solução ótima para o problema significa fazer uma escolha entre os subproblemas que usaremos na resolução do problema. Normalmente, o custo da solução do problema é igual aos custos de subproblemas, mais um custo que pode ser diretamente atribuído à escolha em si. Por exemplo, na programação da linha de montagem, primeiro resolvemos os subproblemas de encontrar o caminho mais rápido

pelas estações  $S_{1,j-1}$  e  $S_{2,j-1}$ , e depois escolhemos uma dessas estações como a estação precedente  $S_{i,j}$ . O custo que pode ser atribuído à própria escolha depende do fato de passarmos ou não de uma linha para a outra entre as estações  $j-1$  e  $j$ ; esse custo é  $a_{i,j}$  se ficarmos na mesma linha, e é  $t_{i',j-1} + a_{i,j}$ , onde  $i' \neq i$ , se trocarmos de linha de montagem. Na multiplicação de cadeias de matrizes, determinamos colocações ótimas dos parênteses de subcadeias de  $A_i A_{i+1} \dots A_j$ , e depois escolhemos a matriz  $A_k$  na qual dividir o produto. O custo que pode ser atribuído à escolha propriamente dita é a expressão  $p_{i-1} p_k p_j$ .

No Capítulo 16, examinaremos os “algoritmos gulosos”, que guardam muitas semelhanças com a programação dinâmica. Em particular, os problemas aos quais os algoritmos gulosos se aplicam têm subestrutura ótima. Uma diferença notável entre algoritmos gulosos e programação dinâmica é que, nos algoritmos gulosos, usamos a subestrutura ótima de cima para baixo (top-down). Em vez de primeiro encontrar soluções ótimas para subproblemas e depois fazer uma escolha, os algoritmos gulosos primeiro fazem uma escolha – a escolha que pareça melhor no momento – e depois resolvem um subproblema resultante dessa escolha.

## Sutilezas

Devemos ter cuidado para não presumir que a subestrutura ótima é aplicável quando ela não o é. Considere os dois problemas a seguir, nos quais temos um grafo orientado  $G = (V, E)$  e vértices  $u, v \in V$ .

**Caminho mais curto não-ponderado:**<sup>2</sup> Encontrar um caminho de  $u$  para  $v$  consistindo no menor número de arestas. Tal caminho deve ser simples, pois a remoção de um ciclo de um caminho produz um caminho com menos arestas.

**Caminho simples mais longo não-ponderado:** Encontrar um caminho simples de  $u$  para  $v$  consistindo no maior número de arestas. Precisamos incluir o requisito de simplicidade porque, do contrário, podemos percorrer um ciclo tantas vezes quantas quisermos para criar caminhos com um número arbitrariamente grande de arestas.

O problema do caminho mais curto não-ponderado apresenta subestrutura ótima, como mostramos a seguir. Suponha que  $u \neq v$ , e assim o problema é não trivial. Então, qualquer caminho  $p$  de  $u$  para  $v$  deve conter um vértice intermediário, digamos  $w$ . (Observe que  $w$  pode ser  $u$  ou  $v$ .) Desse modo, podemos decompor o caminho  $u \xrightarrow{p} v$  em subcaminhos  $u \xrightarrow{p_1} w \xrightarrow{p_2} v$ . É claro que o número de arestas em  $p$  é igual à soma do número de arestas em  $p_1$  e o número de arestas em  $p_2$ . Afirmamos que, se  $p$  é um caminho ótimo (isto é, o caminho mais curto) de  $u$  para  $v$ , então  $p_1$  deve ser um caminho mais curto de  $u$  para  $w$ . Por quê? Usamos um argumento de “recortar e colar”: se existisse outro caminho, digamos  $p'_1$ , de  $u$  para  $w$  com menos arestas que  $p_1$ , então poderíamos recortar  $p_1$  e colar em  $p'_1$  para produzir um caminho  $u \xrightarrow{p'_1} w \xrightarrow{p_2} v$  com menos arestas que  $p$ , contradizendo desse modo o caráter ótimo de  $p$ . Simetricamente,  $p_2$  deve ser um caminho mais curto de  $w$  para  $v$ . Desse modo, podemos encontrar um caminho mais curto de  $u$  para  $v$  considerando todos os vértices intermediários  $w$ , encontrando um caminho mais curto de  $u$  para  $w$  e um caminho mais curto de  $w$  para  $v$ , e escolhendo um vértice intermediário  $w$  que produza o caminho mais curto global. Na Seção 25.2, usamos uma variante dessa observação de subestrutura ótima para encontrar um caminho mais curto entre todos os pares de vértices em um grafo ponderado, orientado.

É tentador presumir que o problema de encontrar um caminho simples mais longo não-ponderado também apresenta subestrutura ótima. Afinal, se fizermos a decomposição de um caminho simples mais longo  $u \xrightarrow{p} v$  em subcaminhos  $u \xrightarrow{p_1} w \xrightarrow{p_2} v$ , então  $p_1$  não deve ser um caminho

---

<sup>2</sup> Usamos a expressão “não-ponderado” para distinguir esse problema do problema de encontrar menores caminhos com arestas ponderadas, que veremos nos Capítulos 24 e 25. Podemos usar a técnica de pesquisa primeiro na extensão do Capítulo 22 para resolver o problema não-ponderado.

simples mais longo de  $u$  para  $w$ , e  $p_2$  não deve ser um caminho simples mais longo de  $w$  para  $v$ ? A resposta é não! A Figura 15.4 fornece um exemplo. Considere o caminho  $q \rightarrow r \rightarrow t$ , que é um caminho simples mais longo de  $q$  para  $t$ . O caminho  $q \rightarrow r$  é um caminho simples mais longo de  $q$  para  $r$ ? Não, pois o caminho  $q \rightarrow s \rightarrow t \rightarrow r$  é um caminho simples mais longo.  $r \rightarrow t$  é um caminho simples mais longo de  $r$  para  $t$ ? Não novamente, pois o caminho  $r \rightarrow q \rightarrow s \rightarrow t$  é um caminho simples mais longo.

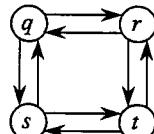


FIGURA 15.4 Um grafo orientado mostrando que o problema de encontrar um caminho simples mais longo em um grafo orientado não-ponderado não tem subestrutura ótima. O caminho  $q \rightarrow r \rightarrow t$  é um caminho simples mais longo de  $q$  para  $t$ , mas o subcaminho  $q \rightarrow r$  não é um caminho simples mais longo de  $q$  para  $r$ , nem o subcaminho  $r \rightarrow t$  é um caminho simples mais longo de  $r$  para  $t$

Esse exemplo mostra que, para caminhos simples mais longos, não apenas falta uma subestrutura ótima, mas não podemos montar necessariamente uma solução “válida” para o problema a partir de soluções para subproblemas. Se combinarmos os caminhos simples mais longos  $q \rightarrow s \rightarrow t \rightarrow r$  e  $r \rightarrow q \rightarrow s \rightarrow t$ , obteremos o caminho  $q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$ , que não é simples. Na realidade, o problema de encontrar um caminho simples mais longo não-ponderado não parece ter qualquer classificação de subestrutura ótima. Nenhum algoritmo eficiente de programação dinâmica já foi encontrado para esse problema. De fato, esse problema é NP-completo, que – como veremos no Capítulo 34 – significa que é improvável que ele possa ser resolvido em tempo polinomial.

E o caso da subestrutura de um caminho simples mais longo, tão diferente da subestrutura de um caminho mais curto? Embora sejam usados dois subproblemas em uma solução para um problema tanto de caminhos mais longos quanto de caminhos mais curtos, os subproblemas na localização do caminho simples mais longo não são *independentes*, enquanto para caminhos mais curtos eles o são. O que significa o fato de subproblemas serem independentes? Significa que a solução para um subproblema não afeta a solução para outro subproblema do mesmo problema. No exemplo da Figura 15.4, temos o problema de encontrar um caminho simples mais longo de  $q$  para  $t$  com dois subproblemas: encontrar caminhos simples mais longos de  $q$  para  $r$  e de  $r$  para  $t$ . Para o primeiro desses subproblemas, escolhemos o caminho  $q \rightarrow s \rightarrow t \rightarrow r$ , e assim também usamos os vértices  $s$  e  $t$ . Não podemos mais usar esses vértices no segundo subproblema, pois a combinação das duas soluções para subproblemas produziria um caminho que não é simples. Se não podemos usar o vértice  $t$  no segundo problema, então não podemos resolvê-lo, pois  $t$  tem de estar no caminho que encontrarmos, e ele não é o vértice em que estamos “reunindo” as soluções de subproblemas (esse vértice é  $r$ ). Nossa uso dos vértices  $s$  e  $t$  em uma solução de subproblema impede que eles sejam usados na solução do outro subproblema. Porém, devemos usar pelo menos um deles para resolver o outro subproblema, e temos de usar ambos para resolvê-lo de forma ótima. Assim, dizemos que esses subproblemas não são independentes. Visto de outro modo, nosso uso de recursos na resolução de um subproblema (sendo esses recursos os vértices) tornou-os indisponíveis para o outro subproblema.

Por que então os subproblemas são independentes na localização de um caminho mais curto? A resposta é que, por natureza, os subproblemas não compartilham recursos. Afirmamos que, se um vértice  $w$  está em um caminho mais curto  $p$  de  $u$  para  $v$ , então podemos reunir *qualquer* caminho mais curto  $u \xrightarrow{p_1} w$  e *qualquer* caminho mais curto  $w \xrightarrow{p_2} v$  para produzir um caminho mais curto de  $u$  para  $v$ . Estamos seguros de que, além de  $w$ , nenhum vértice pode aparecer em ambos os caminhos  $p_1$  e  $p_2$ . Por quê? Suponha que algum vértice  $x \neq w$  apareça tanto em  $p_1$  quanto em  $p_2$ , de forma que possamos decompor  $p_1$  como  $u \xrightarrow{p_{1x}} x \xrightarrow{p_{xy}} w$  e  $p_2$  como  $w \xrightarrow{p_{yx}} x \xrightarrow{p_{xy}} v$ .

Pela subestrutura ótima desse problema, o caminho  $p$  tem tantas arestas quanto  $p_1$  e  $p_2$  juntos; vamos dizer que  $p$  tem  $e$  arestas. Agora, vamos construir um caminho  $u \xrightarrow{p_u} x \xrightarrow{p_{xy}} v$  de  $u$  para  $v$ . Esse caminho tem no máximo  $e - 2$  arestas, o que contradiz a hipótese de que  $p$  é um caminho mais curto. Desse modo, estamos seguros de que os subproblemas para o problema do caminho mais curto são independentes.

Ambos os problemas examinados nas Seções 15.1 e 15.2 têm subproblemas independentes. Na multiplicação de cadeias de matrizes, os subproblemas são multiplicar subcadeias  $A_i A_{i+1} \dots A_k$  e  $A_{k+1} A_{k+2} \dots A_j$ . Essas subcadeias são disjuntas, de forma que nenhuma matriz teria possibilidade de ser incluída em ambas. Na programação da linha de montagem, para descobrir o caminho mais rápido pela estação  $S_{i,j}$ , examinamos os caminhos mais rápidos pelas estações  $S_{1,j-1}$  e  $S_{2,j-1}$ . Como nossa solução para o caminho mais rápido pela estação  $S_{i,j}$  incluirá apenas uma dessas soluções de subproblemas, esse subproblema é automaticamente independente de todos os outros usados na solução.

## Subproblemas superpostos

O segundo ingrediente que um problema de otimização deve ter para a programação dinâmica ser aplicável é que o espaço de subproblemas deve ser “pequeno”, no sentido de que um algoritmo recursivo para o problema resolve os mesmos subproblemas repetidas vezes, em lugar de sempre gerar novos subproblemas. Em geral, o número total de subproblemas distintos é um polinômio no tamanho de entrada. Quando um algoritmo recursivo reexamina o mesmo problema inúmeras vezes, dizemos que o problema de otimização tem **subproblemas superpostos**.<sup>3</sup> Em contraste, um problema para o qual uma abordagem de dividir e conquistar é satisfatória quase sempre gera problemas absolutamente novos em cada passo da recursão. Os algoritmos de programação dinâmica costumam tirar proveito de subproblemas superpostos resolvendo cada subproblema uma vez, e depois armazenando a solução em uma tabela, onde ela possa ser examinada quando necessário, usando-se um tempo constante por pesquisa.

Na Seção 15.1, examinamos brevemente como uma solução recursiva para o problema da programação da linha de montagem faz  $2^{n-j}$  referências a  $f_i[j]$  para  $j = 1, 2, \dots, n$ . Nossa solução tabular reduz um algoritmo recursivo de tempo exponencial a um algoritmo de tempo linear.

Para ilustrar a propriedade de subproblemas superpostos com mais detalhes, vamos examinar novamente o problema de multiplicação de cadeias de matrizes. Consultando mais uma vez a Figura 15.3, observe que MATRIX-CHAIN-ORDER procura repetidamente a solução para subproblemas em linhas inferiores quando resolve subproblemas em linhas superiores. Por exemplo, a entrada  $m[3, 4]$  é referenciada 4 vezes: durante os cálculos de  $m[2, 4]$ ,  $m[1, 4]$ ,  $m[3, 5]$  e  $m[3, 6]$ . Se  $m[3, 4]$  fosse recalculado a cada vez, em vez de ser apenas pesquisado, o aumento no tempo de execução seria drástico. Para verificar isso, considere o seguinte procedimento recursivo (ineficiente) que determina  $m[i, j]$ , o número mínimo de multiplicações escalares necessárias para calcular o produto de cadeias de matrizes  $A_{i..j} = A_i A_{i+1} \dots A_j$ . O procedimento é diretamente baseado na recorrência (15.12).

---

<sup>3</sup> Pode parecer estranho que a programação dinâmica se baseie no fato de os subproblemas serem ao mesmo tempo independentes e superpostos. Embora esses requisitos pareçam contraditórios, eles descrevem duas noções diferentes, em lugar de dois pontos no mesmo eixo. Dois subproblemas do mesmo problema são independentes se eles não compartilham recursos. Dois subproblemas estão superpostos se eles são realmente o mesmo subproblema que ocorre como um subproblema de problemas diferentes.

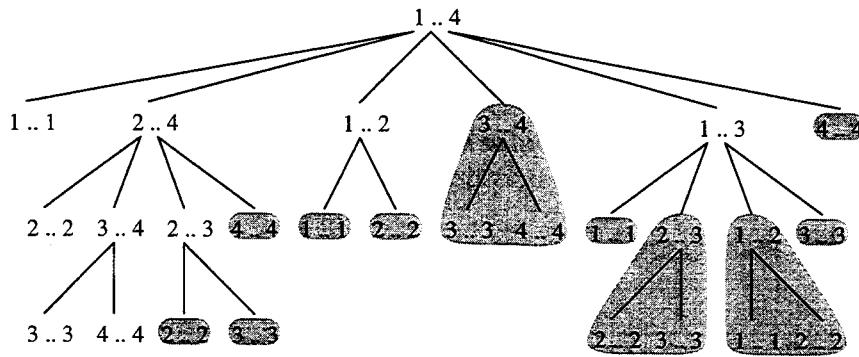


FIGURA 15.5 A árvore de recursão para a computação de RECURSIVE-MATRIX-CHAIN( $p, 1, 4$ ). Cada nó contém os parâmetros  $i$  e  $j$ . Os cálculos executados em uma subárvore sombreada são substituídos por uma única pesquisa de tabela em MEMOIZED-MATRIX-CHAIN( $p, 1, 4$ )

#### RECURSIVE-MATRIX-CHAIN( $p, i, j$ )

```

1 if  $i = j$ 
2 then return 0
3  $m[i, j] \leftarrow \infty$ 
4 for  $k \leftarrow i$  to  $j - 1$ 
5   do  $q \leftarrow \text{RECURSIVE-MATRIX-CHAIN}(p, i, k)$ 
     +  $\text{RECURSIVE-MATRIX-CHAIN}(p, k + 1, j)$ 
     +  $p_{i-1} p_k p_j$ 
6   if  $q < m[i, j]$ 
7     then  $m[i, j] \leftarrow q$ 
8 return  $m[i, j]$ 
```

A Figura 15.5 mostra a árvore de recursão produzida pela chamada RECURSIVE-MATRIX-CHAIN( $p, 1, 4$ ). Cada nó é identificado pelos valores dos parâmetros  $i$  e  $j$ . Observe que alguns pares de valores ocorrem muitas vezes.

De fato, podemos mostrar que o tempo de execução  $T(n)$  para calcular  $m[1, n]$  por esse procedimento recursivo é pelo menos exponencial em  $n$ . Seja  $T(n)$  o tempo tomado por RECURSIVE-MATRIX-CHAIN para calcular uma colocação ótima dos parênteses de uma cadeia de  $n$  matrizes. Vamos supor que a execução das linhas 1 e 2 e das linhas 6 e 7 demore cada uma pelo menos o tempo unitário. A inspeção do procedimento produz a recorrência

$$T(1) \geq 1, \\ T(n) \geq 1 + \sum_{k=1}^{n-1} T(k) + T(n-k) + 1 \quad \text{para } n > 1.$$

Observando que, para  $i = 1, 2, \dots, n-1$ , cada termo  $T(i)$  aparece uma vez como  $T(k)$  e uma vez como  $T(n-k)$ , e reunindo os  $n-1$  valores 1 no somatório com o valor 1 extraído, podemos reescrever a recorrência como

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n. \tag{15.13}$$

Provaremos que  $T(n) = \Omega(2^n)$  usando o método de substituição. Especificamente, mostraremos que  $T(n) \geq 2^{n-1}$  para todo  $n \geq 1$ . A base é fácil, pois  $T(1) \geq 1 = 2^0$ . Por indução, para  $n \geq 2$ , temos

$$T(n) \geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n$$

$$= 2 \sum_{i=0}^{n-2} 2^i + n$$

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

$$= 2(2^n - 2) + n$$

$$\geq 2^{n-1},$$

O que completa a prova. Desse modo, a quantidade total de trabalho executado pela chamada `RECURSIVE-MATRIX-CHAIN(p, 1, n)` é pelo menos exponencial em  $n$ .

Compare esse algoritmo recursivo top-down com o algoritmo de programação dinâmica bottom-up. Esse último é mais eficiente porque tira proveito da propriedade de subproblemas superpostos. Existem apenas  $O(n^2)$  subproblemas diferentes, e o algoritmo de programação dinâmica resolve cada um deles exatamente uma vez. Por outro lado, o algoritmo recursivo deve solucionar repetidamente cada subproblema toda vez que ele reaparece na árvore de recursão. Sempre que uma árvore de recursão para a solução recursiva natural para um problema contém o mesmo subproblema repetidamente, e o número total de subproblemas diferentes é pequeno, é uma boa idéia ver se a programação dinâmica pode ser usada no trabalho.

## Reconstrução de uma solução ótima

Como regra prática, com freqüência armazenamos em uma tabela a opção que escolhemos em cada subproblema, de forma que não tenhamos de reconstruir essa informação a partir dos custos que armazenamos. Na programação da linha de montagem, armazenamos em  $l_i[j]$  a estação que precede  $S_{i,j}$  em um caminho mais rápido por  $S_{i,j}$ . Como outra alternativa, tendo preenchido a tabela  $f_i[j]$  inteira, poderíamos determinar que estação precede  $S_{1,j}$  em um caminho mais rápido por  $S_{i,j}$ , com um pouco de trabalho extra. Se  $f_1[j] = f_1[j-1] + a_{1,j}$ , então a estação  $S_{1,j-1}$  precede  $S_{1,j}$  em um caminho mais rápido por  $S_{1,j}$ . Caso contrário, teria de ocorrer  $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$ , e então  $S_{2,j-1}$  precederia  $S_{1,j}$ . No caso da programação da linha de montagem, a reconstrução das estações predecessoras demora apenas o tempo  $O(1)$  por estação, mesmo sem a tabela  $l_i[j]$ .

Porém, no caso da multiplicação de cadeias de matrizes, a tabela  $s[i,j]$  nos poupa uma quantidade significativa de trabalho durante a reconstrução de uma solução ótima. Vamos supor que não mantivéssemos a tabela  $s[i,j]$ , tendo preenchido apenas a tabela  $m[i,j]$  que contém custos de subproblemas ótimos. Há  $j-i$  escolhas na determinação de quais subproblemas usar em uma solução ótima para a colocação dos parênteses de  $A_i A_{i+1} \dots A_j$ , e  $j-i$  não é uma constante. Então, demoraria o tempo  $\Theta(j-i) = \Omega(1)$  para reconstruir os subproblemas que escolhemos para uma solução de um problema dado. Armazenando em  $s[i,j]$  o índice da matriz em que dividimos o produto  $A_i A_{i+1} \dots A_j$ , podemos reconstruir cada escolha no tempo  $O(1)$ .

## Memoização

Existe uma variação de programação dinâmica que freqüentemente oferece a eficiência da abordagem de programação dinâmica habitual enquanto mantém uma estratégia top-down. A idéia, embora ineficiente, é **memoizar** o algoritmo recursivo natural. Como na programação dinâmica comum, mantemos uma tabela com soluções de subproblemas, mas a estrutura de controle para preencher a tabela é mais semelhante ao algoritmo recursivo.

Um algoritmo recursivo memoizado mantém uma entrada em uma tabela para a solução de cada subproblema. Cada entrada da tabela contém inicialmente um valor especial para indicar que a entrada ainda tem de ser preenchida. Quando o subproblema é encontrado pela primeira vez durante a execução do algoritmo recursivo, sua solução é calculada e depois armazenada na tabela. Em cada momento subsequente que o subproblema é encontrado, o valor armazenado na tabela é apenas pesquisado e retornado.<sup>4</sup>

O procedimento a seguir é uma versão memoizada de RECURSIVE-MATRIX-CHAIN.

**MEMOIZED-MATRIX-CHAIN( $p$ )**

```

1 n comprimento[ $p$ ] – 1
2 for  $i \leftarrow 1$  to  $n$ 
3   do for  $j \leftarrow i$  to  $n$ 
4     do  $m[i, j] \leftarrow \infty$ 
5 return LOOKUP-CHAIN( $p$ , 1,  $n$ )

```

**LOOKUP-CHAIN( $p$ ,  $i$ ,  $j$ )**

```

1 if  $m[i, j] < \infty$ 
2   then return  $m[i, j]$ 
3 if  $i = j$ 
4   then  $m[i, i] \leftarrow 0$ 
5 else for  $k \leftarrow i$  to  $j - 1$ 
6   do  $q \leftarrow$  LOOKUP-CHAIN( $p$ ,  $i$ ,  $k$ )
      + LOOKUP-CHAIN( $p$ ,  $k + 1$ ,  $j$ ) +  $p_{i-1} p_k p_j$ 
7   if  $q < m[i, j]$ 
8     then  $m[i, j] \leftarrow q$ 
9 return  $m[i, j]$ 

```

MEMOIZED-MATRIX-CHAIN, como MATRIX-CHAIN-ORDER, mantém uma tabela  $m[1..n, 1..n]$  de valores calculados de  $m[i, j]$ , o número mínimo de multiplicações escalares necessárias para calcular a matriz  $A_{i..j}$ . Cada entrada de tabela contém inicialmente o valor  $\infty$  para indicar que a entrada ainda tem de ser preenchida. Quando a chamada LOOKUP-CHAIN( $p$ ,  $i$ ,  $j$ ) é executada, se  $m[i, j] < \infty$  na linha 1, o procedimento simplesmente retorna o custo anteriormente calculado  $m[i, j]$  (linha 2). Caso contrário, o custo é calculado como em RECURSIVE-MATRIX-CHAIN, armazenado em  $m[i, j]$  e retornado. (O valor  $\infty$  é conveniente para uso no caso de uma entrada de tabela não preenchida, pois é o valor usado para inicializar  $m[i, j]$  na linha 3 de RECURSIVE-MATRIX-CHAIN.) Desse modo, LOOKUP-CHAIN( $p$ ,  $i$ ,  $j$ ) sempre retorna o valor de  $m[i, j]$ , mas ele só o calcula se essa é a primeira vez que LOOKUP-CHAIN é chamado com os parâmetros  $i$  e  $j$ .

A Figura 15.5 ilustra o modo como MEMOIZED-MATRIX-CHAIN poupa tempo em comparação com RECURSIVE-MATRIX-CHAIN. As subárvores sombreadas representam valores que são pesquisados em vez de serem calculados.

De modo semelhante ao algoritmo de programação dinâmica MATRIX-CHAIN-ORDER, o procedimento MEMOIZED-MATRIX-CHAIN é executado em tempo  $O(n^3)$ . Cada uma das  $\Theta(n^2)$  entradas da tabela é inicializada uma vez na linha 4 de MEMOIZED-MATRIX-CHAIN. Podemos dividir as chamadas de LOOKUP-CHAIN em dois tipos:

1. Chamadas em que  $m[i, j] = \infty$ , de forma que as linhas 3 a 9 são executadas.
2. Chamadas em que  $m[i, j] < \infty$ , de forma que LOOKUP-CHAIN simplesmente retorna na linha 2.

---

<sup>4</sup> Essa abordagem pressupõe que o conjunto de todos os parâmetros de subproblemas possíveis é conhecido e que a relação entre posições de tabela e subproblemas está estabelecida. Outra abordagem é memoizar usando o hash com os parâmetros do subproblema como chaves.

Há  $\Theta(n^2)$  chamadas do primeiro tipo, uma por entrada de tabela. Todas as chamadas do segundo tipo são feitas como chamadas recursivas por chamadas do primeiro tipo. Sempre que uma dada chamada de LOOKUP-CHAIN efetua chamadas recursivas, ela faz  $O(n)$  delas. Assim, existem ao todo  $O(n^3)$  chamadas do segundo tipo. Cada chamada do segundo tipo demora o tempo  $O(1)$ , e cada chamada do primeiro tipo demora o tempo  $O(n)$  mais o tempo gasto em suas chamadas recursivas. Por conseguinte, o tempo total é  $O(n^3)$ . Desse modo, a memoização transforma um algoritmo de tempo  $\Omega(2^n)$  em um algoritmo de tempo  $O(n^3)$ .

Em resumo, o problema de multiplicação de cadeias de matrizes pode ser resolvido no tempo  $O(n^3)$  por um algoritmo top-down memoizado ou por um algoritmo bottom-up de programação dinâmica. Ambos os métodos tiram proveito da propriedade de subproblemas superpostos. Existem apenas  $\Theta(n^2)$  subproblemas diferentes no total, e qualquer um desses métodos calcula a solução para cada subproblema uma vez. Sem a memoização, o algoritmo recursivo natural é executado em tempo exponencial, pois subproblemas resolvidos são repetidamente resolvidos.

Na prática geral, se todos os subproblemas devem ser resolvidos pelo menos uma vez, um algoritmo bottom-up de programação dinâmica normalmente supera um algoritmo top-down memoizado por um fator constante, porque não há nenhuma sobrecarga para recursão e menor sobrecarga para manutenção da tabela. Além disso, existem alguns problemas para os quais o padrão normal de acessos a tabelas no algoritmo de programação dinâmica pode ser explorado para reduzir ainda mais requisitos de tempo ou espaço. De modo alternativo, se alguns subproblemas no espaço de subproblemas não precisarem ser resolvidos de modo algum, a solução memoizada terá a vantagem de só resolver os subproblemas que são definitivamente necessários.

## Exercícios

### 15.3-1

Qual é a maneira mais eficiente de determinar o número ótimo de multiplicações em um problema de multiplicação de cadeias de matrizes: enumerar todos os modos de colocar o produto entre parênteses e calcular o número de multiplicações para cada um, ou executar RECURSIVE-MATRIX-CHAIN? Justifique sua resposta.

### 15.3-2

Desenhe a árvore de recursão para o procedimento MERGE-SORT da Seção 2.3.1 em um arranjo de 16 elementos. Explique por que a memoização é ineficaz na aceleração de um bom algoritmo de dividir e conquistar como MERGE-SORT.

### 15.3-3

Considere uma variante do problema da multiplicação de cadeias de matrizes na qual a meta é colocar entre parênteses a seqüência de matrizes de forma a maximizar, em lugar de minimizar, o número de multiplicações escalares. Esse problema apresenta subestrutura ótima?

### 15.3-4

Descreva como a programação da linha de montagem tem subproblemas superpostos.

### 15.3-5

Como está enunciado, em programação dinâmica primeiro resolvemos os subproblemas e depois escolhemos qual deles utilizar em uma solução ótima para o problema. A professora Capulet afirma que nem sempre é necessário resolver todos os subproblemas a fim de encontrar uma solução ótima. Ela sugere que uma solução ótima para o problema de multiplicação de cadeias de matrizes pode ser encontrada escolhendo-se sempre a matriz  $A_k$  na qual será dividido o subproduto  $A_i A_{i+1} \dots A_j$  (selecionando-se  $k$  para minimizar a quantidade  $p_{i-1} p_k p_j$ ) antes de resolver os subproblemas. Encontre uma instância do problema de multiplicação de cadeias de matrizes para a qual essa abordagem gulosa produz uma solução não ótima.

## 15.4 Subseqüência comum mais longa

Em aplicações biológicas, freqüentemente queremos comparar o DNA de dois (ou mais) organismos diferentes. Uma cadeia de DNA consiste em uma cadeia de moléculas chamadas **bases**, na qual as bases possíveis são adenina, guanina, citosina e timina. Representando-se cada uma dessas bases por suas letras iniciais, uma cadeia de DNA pode ser expressa como uma cadeia sobre o conjunto finito  $\{A, C, G, T\}$ . (Consulte o Apêndice C para ver uma definição de cadeia.) Por exemplo, o DNA de um organismo pode ser  $S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA$ , enquanto o DNA de outro organismo pode ser  $S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA$ . Um objetivo da comparação de duas cadeias de DNA é determinar o quanto as duas cadeias são “semelhantes” como alguma medida de quanto os dois organismos estão intimamente relacionados. A semelhança pode ser e é definida de muitas maneiras diferentes. Por exemplo, podemos dizer que duas cadeias de DNA são semelhantes se uma delas é uma subcadeia da outra. (O Capítulo 32 explora algoritmos para resolver esse problema.) Em nosso exemplo, nem  $S_1$  nem  $S_2$  é uma subcadeia da outra. Como alternativa, poderíamos dizer que duas cadeias são semelhantes se o número de alterações necessárias para transformar uma na outra é pequeno. (O Problema 15-3 examina essa noção.) Ainda outra maneira de medir a semelhança de cadeias  $S_1$  e  $S_2$  é encontrar uma terceira cadeia  $S_3$  na qual as bases de  $S_3$  aparecem em cada uma das cadeias  $S_1$  e  $S_2$ ; essas bases devem aparecer na mesma ordem, mas não necessariamente consecutivas. Quanto mais longa a cadeia  $S_3$  que pudermos encontrar, maior será a semelhança entre  $S_1$  e  $S_2$ . Em nosso exemplo, a cadeia  $S_3$  mais longa é  $GTCGTCGGAAGCCGGCCGAA$ .

Formalizamos essa última noção de semelhança como o problema da subseqüência comum mais longa. Uma subseqüência de uma determinada seqüência é apenas a seqüência dada com zero ou mais elementos omitidos. Em termos formais, dada uma seqüência  $X = \langle x_1, x_2, \dots, x_m \rangle$ , outra seqüência  $Z = \langle z_1, z_2, \dots, z_k \rangle$  é uma **subseqüência** de  $X$  se existe uma seqüência estritamente crescente  $\langle i_1, i_2, \dots, i_k \rangle$  de índices de  $X$  tais que, para todo  $j = 1, 2, \dots, k$ , temos  $x_{i_j} = z_j$ . Por exemplo,  $Z = \langle B, C, D, B \rangle$  é uma subseqüência de  $X = \langle A, B, C, B, D, A, B \rangle$  com seqüência de índice correspondente  $\langle 2, 3, 5, 7 \rangle$ .

Dadas duas seqüências  $X$  e  $Y$ , dizemos que uma seqüência  $Z$  é uma **subseqüência comum** de  $X$  e  $Y$  se  $Z$  é uma subseqüência de  $X$  e de  $Y$  ao mesmo tempo. Por exemplo, se  $X = \langle A, B, C, B, D, A, B \rangle$  e  $Y = \langle B, D, C, A, B, A \rangle$ , a seqüência  $\langle B, C, A \rangle$  é uma subseqüência comum das seqüências  $X$  e  $Y$ . Porém, a seqüência  $\langle B, C, A \rangle$  não é uma subseqüência comum *mais longa* (LCS – longest common subsequence) de  $X$  e  $Y$ , pois tem comprimento 3, e a seqüência  $\langle B, C, B, A \rangle$ , que também é comum a  $X$  e  $Y$ , tem comprimento 4. A seqüência  $\langle B, C, B, A \rangle$  é uma LCS de  $X$  e  $Y$ , da mesma forma que a seqüência  $\langle B, D, A, B \rangle$ , porque não existe nenhuma subseqüência comum de comprimento 5 ou maior.

No **problema da subseqüência comum mais longa**, temos duas seqüências  $X = \langle x_1, x_2, \dots, x_m \rangle$  e  $Y = \langle y_1, y_2, \dots, y_n \rangle$ , e desejamos encontrar uma subseqüência comum de comprimento máximo de  $X$  e  $Y$ . Esta seção mostra que o problema da LCS pode ser resolvido de forma eficiente usando-se a programação dinâmica.

### Etapa 1: Caracterização de uma subseqüência comum mais longa

Uma abordagem de força bruta para resolver o problema da LCS é enumerar todas as subseqüências de  $X$  e conferir cada subseqüência para ver se ela também é uma subseqüência de  $Y$ , controlando a subseqüência mais longa encontrada. Cada subseqüência de  $X$  corresponde a um subconjunto dos índices  $\{1, 2, \dots, m\}$  de  $X$ . Existem  $2^m$  subseqüências de  $X$ ; assim, essa abordagem exige tempo exponencial, o que a torna impraticável para longas seqüências.

Porém, o problema da LCS tem uma propriedade de subestrutura ótima, como mostra o teorema a seguir. Como veremos, as classes naturais de subproblemas correspondem a pares de “prefixos” das duas seqüências. Para ser preciso, dada uma seqüência  $X = \langle x_1, x_2, \dots, x_m \rangle$ , definimos o  $i$ -ésimo **prefixo** de  $X$ , para  $i = 0, 1, \dots, m$ , como  $X_i = \langle x_1, x_2, \dots, x_i \rangle$ . Por exemplo, se  $X = \langle A, B, C, B, D, A, B \rangle$ , então  $X_4 = \langle A, B, C, B \rangle$  e  $X_0$  é a seqüência vazia.

### Teorema 15.1 (Subestrutura ótima de uma LCS)

Sejam as seqüências  $X = \langle x_1, x_2, \dots, x_m \rangle$  e  $Y = \langle y_1, y_2, \dots, y_n \rangle$ , e seja  $Z = \langle z_1, z_2, \dots, z_l \rangle$  qualquer LCS de  $X$  e  $Y$ .

1. Se  $x_m = y_n$ , então  $z_k = x_m = y_n$  e  $Z_{k-1}$  é uma LCS de  $X_{m-1}$  e  $Y_{n-1}$ .
2. Se  $x_m \neq y_n$ , então  $z_k \neq x_m$  implica que  $Z$  é uma LCS de  $X_{m-1}$  e  $Y$ .
3. Se  $x_m \neq y_n$ , então  $z_k \neq y_m$  implica que  $Z$  é uma LCS de  $X$  e  $Y_{n-1}$ .

**Prova** (1) Se  $z_k \neq x_m$ , então poderíamos anexar  $x_m = y_n$  a  $Z$  para obter uma subseqüência comum de  $X$  e  $Y$  de comprimento  $k+1$ , contradizendo a hipótese de que  $Z$  é uma subseqüência comum *mais longa* de  $X$  e  $Y$ . Desse modo, devemos ter  $z_k = x_m = y_n$ . Agora, o prefixo  $Z_{k-1}$  é uma subseqüência comum de comprimento  $(k-1)$  de  $X_{m-1}$  e  $Y_{n-1}$ . Desejamos mostrar que ela é uma LCS. Suponha para fins de contradição que exista uma subseqüência comum  $W$  de  $X_{m-1}$  e  $Y_{n-1}$  com comprimento maior que  $k-1$ . Então, a anexação de  $x_m = y_n$  a  $W$  produz uma subseqüência comum de  $X$  e  $Y$  cujo comprimento é maior que  $k$ , o que é uma contradição.

(2) Se  $z_k \neq x_m$ , então  $Z$  é uma subseqüência comum de  $X_{m-1}$  e  $Y$ . Se existisse uma subseqüência comum  $W$  de  $X_{m-1}$  e  $Y$  com comprimento maior que  $k$ , então  $W$  também seria uma subseqüência comum de  $X_m$  e  $Y$ , contradizendo a hipótese de que  $Z$  é uma LCS de  $X$  e  $Y$ .

(3) A prova é simétrica a (2). ■

A caracterização do Teorema 15.1 mostra que uma LCS de duas seqüências contém dentro dela uma LCS de prefixos das duas seqüências. Desse modo, o problema de LCS tem uma propriedade de subestrutura ótima. Uma solução recursiva também tem a propriedade de subproblemas superpostos, como veremos em breve.

### Etapa 2: Uma solução recursiva

O Teorema 15.1 implica que existem um ou dois subproblemas a examinar quando se encontra uma LCS de  $X = \langle x_1, x_2, \dots, x_m \rangle$  e  $Y = \langle y_1, y_2, \dots, y_n \rangle$ . Se  $x_m = y_n$ , devemos encontrar uma LCS de  $X_{m-1}$  e  $Y_{n-1}$ . A anexação de  $x_m = y_n$  a essa LCS produz uma LCS de  $X$  e  $Y$ . Se  $x_m \neq y_n$ , então devemos resolver dois subproblemas: encontrar uma LCS de  $X_{m-1}$  e  $Y$  e encontrar uma LCS de  $X$  e  $Y_{n-1}$ . Qualquer dessas duas LCSs que seja a mais longa é uma LCS de  $X$  e  $Y$ . Como esses casos esgotam todas as possibilidades, sabemos que uma das soluções ótimas de subproblemas tem de ser usada dentro de um LCS de  $X$  e  $Y$ .

Podemos ver de imediato a propriedade de subproblemas superpostos no problema da LCS. Para encontrar uma LCS de  $X$  e  $Y$ , podemos precisar encontrar as LCSs de  $X$  e  $Y_{n-1}$  e de  $X_{m-1}$  e  $Y$ . Porém, cada um desses subproblemas tem o subsubproblema de encontrar a LCS de  $X_{m-1}$  e  $Y_{n-1}$ . Muitos outros subproblemas compartilham subsubproblemas.

Da mesma forma que o problema de multiplicação de cadeias de matrizes, nossa solução recursiva para o problema da LCS envolve estabelecer uma recorrência para o valor de uma solução ótima. Vamos definir  $c[i, j]$  como o comprimento de uma LCS das seqüências  $X_i$  e  $Y_j$ . Se  $i = 0$  ou  $j = 0$ , uma das seqüências tem comprimento 0; assim, a LCS tem comprimento 0. A subestrutura ótima do problema da LCS fornece a fórmula recursiva

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j. \end{cases} \quad (15.14)$$

Observe que, nessa formulação recursiva, uma condição no problema restringe os subproblemas que podemos considerar. Quando  $x_i = y_j$ , podemos e devemos considerar o subproblema de encontrar a LCS de  $X_{i-1}$  e  $Y_{j-1}$ . Caso contrário, consideraremos os dois subproblemas de

encontrar a LCS de  $X_i$  e  $Y_{j-1}$  e de  $X_{i-1}$  e  $Y_j$ . Nos algoritmos de programação dinâmica anteriores que examinamos – para a programação de linha de montagem e multiplicação de cadeias de matrizes –, nenhum subproblema foi excluído devido às condições do problema. A localização da LCS não é o único algoritmo de programação dinâmica que elimina subproblemas com base em condições do problema. Por exemplo, o problema da distância de edição (ver Problema 15-3) tem essa característica.

### Etapa 3: Como calcular o comprimento de uma LCS

Com base na equação (15.14), poderíamos escrever facilmente um algoritmo recursivo de tempo exponencial para calcular o comprimento de uma LCS de duas seqüências. Contudo, tendo em vista que existem apenas  $\Theta(mn)$  subproblemas distintos, podemos usar a programação dinâmica para calcular as soluções de baixo para cima.

O procedimento LCS-LENGTH toma duas seqüências  $X = \langle x_1, x_2, \dots, x_m \rangle$   $Y = \langle y_1, y_2, \dots, y_n \rangle$  como entradas. Ele armazena os valores  $c[i, j]$  em uma tabela  $c[0..m, 0..n]$  cujas entradas são calculadas em ordem de linha principal. (Isto é, a primeira linha de  $c$  é preenchida da esquerda para a direita, depois a segunda linha e assim por diante.) Ele também mantém a tabela  $b[1..m, 1..n]$  para simplificar a construção de uma solução ótima. Intuitivamente,  $b[i, j]$  aponta para a entrada da tabela correspondente à solução ótima de subproblema escolhida ao se calcular  $c[i, j]$ . O procedimento retorna as tabelas  $b$  e  $c$ ;  $c[m, n]$  contém o comprimento de uma LCS de  $X$  e  $Y$ .

```

LCS-LENGTH( $X, Y$ )
1  $m \leftarrow \text{comprimento}[X]$ 
2  $n \leftarrow \text{comprimento}[Y]$ 
3 for  $i \leftarrow 1$  to  $m$ 
4   do  $c[i, 0] \leftarrow 0$ 
5 for  $j \leftarrow 0$  to  $n$ 
6   do  $c[0, j] \leftarrow 0$ 
7 for  $i \leftarrow 1$  to  $m$ 
8   do for  $j \leftarrow 1$  to  $n$ 
9     do if  $x_i = y_j$ 
10       then  $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
11        $b[i, j] \leftarrow \text{"↖"}$ 
12     else if  $c[i - 1, j] \geq c[i, j - 1]$ 
13       then  $c[i, j] \leftarrow c[i - 1, j]$ 
14        $b[i, j] \leftarrow \text{"↑"}$ 
15     else  $c[i, j] \leftarrow c[i, j - 1]$ 
16        $b[i, j] \leftarrow \text{"←"}$ 
17 return  $c$  e  $b$ 
```

A Figura 15.6 mostra as tabelas produzidas por LCS-LENGTH nas seqüências  $X = \langle A, B, C, B, D, A, B \rangle$  e  $Y = \langle B, D, C, A, B, A \rangle$ . O tempo de execução do procedimento é  $O(mn)$ , pois cada entrada de tabela demora o tempo  $O(1)$  para ser calculada.

### Etapa 4: A construção de uma LCS

A tabela  $b$  retornada por LCS-LENGTH pode ser usada para construir rapidamente uma LCS de  $X = \langle x_1, x_2, \dots, x_m \rangle$  e  $Y = \langle y_1, y_2, \dots, y_n \rangle$ . Simplesmente começamos em  $b[m, n]$  e percorremos a tabela seguindo as setas. Sempre que encontramos uma “↖” na entrada  $b[i, j]$ , ela implica que  $x_i = y_j$  é um elemento da LCS. Os elementos da LCS são encontrados em ordem inversa por esse método. O procedimento recursivo a seguir imprime uma LCS de  $X$  e  $Y$  na ordem direta apropriada. A invocação inicial é PRINT-LCS( $b, X, \text{comprimento}[X], \text{comprimento}[Y]$ ).

	$j$	0	1	2	3	4	5	6
$i$	$y_j$	D	C	A	B	B	B	
0	$x_i$	0	0	0	0	0	0	0
1	A	0	↑	↑	↑	↖	↖	↖
2	B	0	1	↖	↖	1	2	↖
3	C	0	1	1	2	↖	2	↑
4	D	0	1	1	2	2	3	↖
5	D	0	1	2	2	2	3	↑
6	E	0	1	2	3	3	3	4
7	B	0	1	2	2	3	4	4

FIGURA 15.6 As tabelas  $c$  e  $b$  calculadas por LCS-LENGTH sobre as sequências  $X = \langle A, B, C, B, D, A, B \rangle$  e  $Y = \langle B, D, C, A, B, A \rangle$ . O quadrado na linha  $i$  e na coluna  $j$  contém o valor de  $c[i, j]$  e a seta apropriada para o valor de  $b[i, j]$ . A entrada 4 em  $c[7, 6]$  – o canto inferior direito da tabela – é o comprimento de uma LCS  $\langle B, C, B, A \rangle$  de  $X$  e  $Y$ . Para  $i, j > 0$ , a entrada  $c[i, j]$  depende apenas de  $x_i = y_j$  e dos valores nas entradas  $c[i - 1, j], c[i, j - 1]$  e  $c[i - 1, j - 1]$ , que são calculados antes de  $c[i, j]$ . Para reconstruir os elementos de uma LCS, siga as setas  $b[i, j]$  desde o canto inferior direito; o caminho está sombreado. Cada “↖” no caminho corresponde a uma entrada (destacada) para a qual  $x_i = y_j$  é membro de uma LCS

PRINT-LCS( $b, X, i, j$ )

```

1 if  $i = 0$  or  $j = 0$ 
2 then return
3 if  $b[i, j] = "↖"$ 
4 then PRINT-LCS( $b, X, i - 1, j - 1$ )
5     print  $x_i$ 
6 elseif  $b[i, j] = "↑"$ 
7 then PRINT-LCS( $b, X, i - 1, j$ )
8 else PRINT-LCS( $b, X, i, j - 1$ )

```

Para a tabela  $b$  na Figura 15.6, esse procedimento imprime “B CBA”. O procedimento demora o tempo  $O(m + n)$ , pois pelo menos um dentre  $i$  e  $j$  é decrementado em cada fase da recursão.

## Melhorando o código

Depois de desenvolver um algoritmo, com freqüência você descobrirá que é possível melhorar o tempo ou o espaço que ele utiliza. Isso é especialmente verdadeiro no caso de algoritmos diretos de programação dinâmica. Algumas mudanças podem simplificar o código e melhorar fatores constantes mas, por outro lado, não produzir nenhuma melhora assintótica no desempenho. Outras podem resultar em economias assintóticas significativas de tempo e de espaço.

Por exemplo, podemos eliminar totalmente a tabela  $b$ . Cada entrada  $c[i, j]$  depende apenas de três outras entradas na tabela  $c$ :  $c[i - 1, j - 1]$ ,  $c[i - 1, j]$  e  $c[i, j - 1]$ . Dado o valor de  $c[i, j]$ , podemos determinar em tempo  $O(1)$  qual desses três valores foi usado para calcular  $c[i, j]$ , sem inspecionar a tabela  $b$ . Desse modo, podemos reconstruir uma LCS em tempo  $O(m + n)$  usando um procedimento semelhante a PRINT-LCS. (O Exercício 15.4-2 lhe pede para fornecer o pseudocódigo.) Embora façamos economia de espaço  $\Theta(mn)$  por esse método, o requisito de espaço auxiliar para calcular uma LCS não diminui assintoticamente, pois de qualquer forma precisamos do espaço  $\Theta(mn)$  para a tabela  $c$ .

Entretanto, podemos reduzir os requisitos de espaço assintótico para LCS-LENGTH, pois ele

284 só precisa de duas linhas da tabela  $c$  de cada vez: a linha que está sendo calculada e a linha ante-

rior. (De fato, podemos usar apenas um pouco mais que o espaço para uma linha de  $c$  com o objetivo de calcular o comprimento de uma LCS. Consulte o Exercício 15.4-4.) Esse aperfeiçoamento funciona se necessitamos apenas do comprimento de uma LCS; se precisarmos reconstruir os elementos de uma LCS, a tabela menor não guardará informações suficientes para reconstituir nossas etapas no tempo  $O(m + n)$ .

## Exercícios

### 15.4-1

Determine uma LCS de  $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$  e  $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$ .

### 15.4-2

Mostre como reconstruir uma LCS a partir da tabela  $c$  concluída e das seqüências originais  $X = \langle x_1, x_2, \dots, x_m \rangle$  e  $Y = \langle y_1, y_2, \dots, y_n \rangle$  em tempo  $O(m + n)$ , sem usar a tabela  $b$ .

### 15.4-3

Forneça uma versão memoizada de LCS-LENGTH que seja executada no tempo  $O(mn)$ .

### 15.4-4

Mostre como calcular o comprimento de uma LCS usando apenas  $2 \cdot \min(m, n)$  entradas na tabela  $c$  e mais o espaço adicional  $O(1)$ . Em seguida, mostre como fazer isso usando  $\min(m, n)$  entradas e mais o espaço adicional  $O(1)$ .

### 15.4-5

Forneça um algoritmo de tempo  $O(n^2)$  para encontrar a subseqüência monotonicamente crescente mais longa de uma seqüência de  $n$  números.

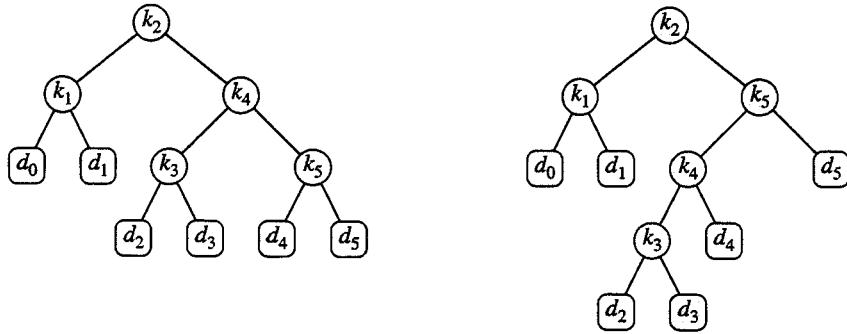
### 15.4-6 \*

Forneça um algoritmo de tempo  $O(n \lg n)$  para encontrar a subseqüência monotonicamente crescente mais longa de uma seqüência de  $n$  números. (Sugestão: Observe que o último elemento de uma subseqüência candidata de comprimento  $i$  é pelo menos tão grande quanto o último elemento de uma subseqüência candidata de comprimento  $i - 1$ . Mantenha as subseqüências candidatas unindo-as através da seqüência de entrada.)

## 15.5 Árvores de pesquisa binária ótimas

Suponha que estamos projetando um programa para traduzir texto de inglês para francês. Para cada ocorrência de cada palavra inglesa no texto, precisamos procurar seu equivalente em francês. Um modo de executar essas operações de pesquisa é construir uma árvore de pesquisa binária com  $n$  palavras inglesas como chaves e equivalentes franceses como dados satélite. Tendo em vista que pesquisaremos a árvore para cada palavra individual no texto, queremos que o tempo total gasto na pesquisa seja tão baixo quanto possível. Poderíamos assegurar um tempo de pesquisa  $O(\lg n)$  por ocorrência usando uma árvore vermelho-preto ou qualquer outra árvore de pesquisa binária平衡ada. Porém, as palavras aparecem com freqüências diferentes, e pode ocorrer o caso de uma palavra usada freqüentemente como “the” aparecer longe da raiz, enquanto uma palavra raramente usada como “mycophagist” apareceria perto da raiz. Tal organização diminuiria a velocidade da tradução, pois o número de nós visitados durante a pesquisa de uma chave em uma árvore de pesquisa binária é uma unidade maior que a profundidade do nó que contém a chave. Queremos que palavras que ocorrem com freqüência no texto sejam colocadas mais próximas à raiz.<sup>5</sup> Além disso, podem ocorrer palavras no texto para as quais não existe nenhuma tradução francesa, e tais palavras talvez não apareçam em nenhum lugar na árvore de pesquisa binária. De que modo organizariamos uma árvore de pesquisa binária de forma a minimizar o número de nós visitados em todas as pesquisas, considerando que sabemos com que freqüência cada palavra ocorre?

<sup>5</sup> Se o assunto do texto fosse cogumelos comestíveis, talvez quiséssemos que “mycophagist” aparecesse perto da raiz.



(a)

(b)

FIGURA 15.7 Duas árvores de pesquisa binária para um conjunto de  $n = 5$  chaves com as seguintes probabilidades:

$i$	0	1	2	3	4	5
$p_i$	0,15	0,10	0,05	0,10	0,20	
$q_i$	0,05	0,10	0,05	0,05	0,05	0,10

(a) Uma árvore de pesquisa binária com custo de pesquisa esperado 2,80. (b) Uma árvore de pesquisa binária com custo de pesquisa esperado 2,75. Essa árvore é ótima

O que precisamos é conhecido como uma **árvore de pesquisa binária ótima**. Formalmente, temos uma seqüência  $K = \langle k_1, k_2, \dots, k_n \rangle$  de  $n$  chaves distintas em seqüência ordenada (de forma que  $k_1 < k_2 < \dots < k_n$ ), e desejamos construir uma árvore de pesquisa binária dessas chaves. Para cada chave  $k_i$ , temos uma probabilidade  $p_i$  de que uma pesquisa seja para  $k_i$ . Algumas pesquisas podem ser para valores não pertencentes a  $K$ , e então também temos  $n + 1$  “chaves fictícias”  $d_0, d_1, d_2, \dots, d_n$  representando valores não pertencentes a  $K$ . Em particular,  $d_0$  representa todos os valores menores que  $k_1$ ,  $d_n$  representa todos os valores maiores que  $k_n$  e, para  $i = 1, 2, \dots, n - 1$ , a chave fictícia  $d_i$  representa todos os valores entre  $k_i$  e  $k_{i+1}$ . Para cada chave fictícia  $d_i$ , temos uma probabilidade  $q_i$  de que uma pesquisa corresponda a  $d_i$ . A Figura 15.7 mostra duas árvores de pesquisa binária para um conjunto de  $n = 5$  chaves. Cada chave  $k_i$  é um nó interno, e cada chave fictícia  $d_i$  é uma folha. Toda pesquisa é bem-sucedida (encontrando alguma chave  $k_i$ ) ou malsucedida (encontrando alguma chave fictícia  $d_i$ ), e então temos

$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1. \quad (15.15)$$

Pelo fato de termos probabilidades de pesquisas para cada chave e cada chave fictícia, podemos determinar o custo esperado de uma pesquisa em uma árvore de pesquisa binária dada  $T$ . Vamos supor que o custo real de uma pesquisa é o número de nós examinados, isto é, a profundidade do nó encontrado pela pesquisa em  $T$ , mais 1. Então, o custo esperado de uma pesquisa em  $T$  é

$$\begin{aligned} E[\text{custo da pesquisa em } T] &= \sum_{i=1}^n (\text{profundidade}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{profundidade}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^n \text{profundidade}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{profundidade}_T(d_i) \cdot q_i, \end{aligned} \quad (15.16)$$

onde profundidade<sub>T</sub> denota a profundidade de um nó na árvore  $T$ . A última igualdade decorre da equação (15.15). Na Figura 15.7(a), podemos calcular o custo esperado da pesquisa por nó:

nó	profundidade	probabilidade	contribuição
$k_1$	1	0,15	0,30
$k_2$	0	0,10	0,10
$k_3$	2	0,05	0,15
$k_4$	1	0,10	0,20
$k_5$	2	0,20	0,60
$d_0$	2	0,05	0,15
$d_1$	2	0,10	0,30
$d_2$	3	0,05	0,20
$d_3$	3	0,05	0,20
$d_4$	3	0,05	0,20
$d_5$	3	0,10	0,40
Total			2,80

Para um dado conjunto de probabilidades, nossa meta é construir uma árvore de pesquisa binária cujo custo de pesquisa esperado seja mínimo. Chamamos tal árvore uma **árvore de pesquisa binária ótima**. A Figura 15.7(b) mostra uma árvore de pesquisa binária ótima para as probabilidades dadas na legenda da figura; seu custo esperado é 2,75. Esse exemplo mostra que uma árvore de pesquisa binária ótima não é necessariamente uma árvore cuja altura global é menor. Nem podemos necessariamente construir uma árvore de pesquisa binária ótima inserindo sempre a chave com maior probabilidade na raiz. Aqui, a chave  $k_5$  tem a maior probabilidade de pesquisa de qualquer chave, ainda que a raiz da árvore de pesquisa binária ótima mostrada seja  $k_2$ . (O custo esperado mais baixo de qualquer árvore de pesquisa binária com  $k_5$  na raiz é 2,85.)

Como ocorre com a multiplicação de cadeias de matrizes, a verificação exaustiva de todas as possibilidades deixa de produzir um algoritmo eficiente. Podemos identificar os nós de qualquer árvore binária de  $n$  nós com as chaves  $k_1, k_2, \dots, k_n$  para construir uma árvore de pesquisa binária, e depois adicionar as chaves fictícias como folhas. No Problema 12-4, vimos que o número de árvores binárias com  $n$  nós é  $\Omega(4^n/n^{3/2})$ , e portanto existe um número exponencial de árvores de pesquisa binária que teríamos de examinar em uma pesquisa exaustiva. Não surpreende o fato de resolvermos esse problema com programação dinâmica.

### Etapa 1: A estrutura de uma árvore de pesquisa binária ótima

Para caracterizar a subestrutura ótima de árvores de pesquisa binária ótima, começamos com uma observação sobre subárvores. Considere qualquer subárvore de uma árvore de pesquisa binária. Ela deve conter chaves em um intervalo contíguo  $k_i, \dots, k_j$ , para algum  $1 \leq i \leq j \leq n$ . Além disso, uma subárvore que contém chaves  $k_i, \dots, k_j$  também deve ter como suas folhas as chaves fictícias  $d_{i-1}, \dots, d_j$ .

Agora podemos enunciar a subestrutura ótima: se uma árvore de pesquisa binária ótima  $T$  tem uma subárvore  $T'$  contendo chaves  $k_i, \dots, k_j$ , então essa subárvore  $T'$  também deve ser ótima para o subproblema com chaves  $k_i, \dots, k_j$  e chaves fictícias  $d_{i-1}, \dots, d_j$ . O argumento habitual de recortar e colar é aplicável. Se houvesse uma subárvore  $T''$  cujo custo esperado fosse mais baixo que o de  $T'$ , poderíamos recortar  $TY$  de  $T$  e colar em  $T''$ , resultando em uma árvore de pesquisa binária de custo esperado mais baixo que  $T$ , contradizendo assim o caráter ótimo de  $T$ .

Precisamos usar a subestrutura ótima para mostrar que podemos construir uma solução ótima para o problema a partir de soluções ótimas para subproblemas. Dadas as chaves  $k_i, \dots, k_j$ , uma dessas chaves, digamos  $k_r$  ( $i \leq r \leq j$ ), será a raiz de uma subárvore ótima contendo essas chaves. A subárvore esquerda da raiz  $k_r$  conterá as chaves  $k_i, \dots, k_{r-1}$  (e chaves fictícias  $d_{i-1}, \dots, d_{r-1}$ ),

e a subárvore direita conterá as chaves  $k_{r+1}, \dots, k_j$  (e chaves fictícias  $d_r, \dots, d_j$ ). Desde que examinemos todas as raízes candidatas  $k_r$ , onde  $i \leq r \leq j$ , e determinemos todas as árvores de pesquisa binária ótimas contendo  $k_i, \dots, k_{r-1}$  e as que contêm  $k_{r+1}, \dots, k_j$ , teremos a garantia de que encontraremos uma árvore de pesquisa binária ótima.

Há um detalhe que vale a pena observar sobre subárvores “vazias”. Suponha que, em uma subárvore com chaves  $k_i, \dots, k_j$ , selecionamos  $k_i$  como a raiz. Pelo argumento anterior, a subárvore esquerda de  $k_i$  contém as chaves  $k_i, \dots, k_{i-1}$ . É natural interpretar essa seqüência como não contendo nenhuma chave. Contudo, lembre-se de que as subárvores também contêm chaves fictícias. Adotamos a convenção de que uma subárvore contendo chaves  $k_i, \dots, k_{i-1}$  não tem nenhuma chave real, mas contém a única chave fictícia  $d_{i-1}$ . Simetricamente, se selecionarmos  $k_j$  como a raiz, então a subárvore direita de  $k_j$  contém as chaves  $k_{j+1}, \dots, k_j$ ; essa subárvore direita não contém nenhuma chave real, mas contém a chave fictícia  $d_j$ .

## Etapa 2: Uma solução recursiva

Estamos prontos para definir o valor de uma solução ótima recursivamente. Escolhemos o domínio de nosso subproblema como encontrar uma árvore de pesquisa binária ótima contendo as chaves  $k_i, \dots, k_j$ , onde  $i \geq 1, j \leq n$  e  $j \geq i - 1$ . (Quando  $j = i - 1$  não existe nenhuma chave real; temos apenas a chave fictícia  $d_{i-1}$ .) Vamos definir  $e[i, j]$  como o custo esperado de pesquisar uma árvore de pesquisa binária ótima contendo as chaves  $k_i, \dots, k_j$ . Em última análise, desejamos calcular  $e[1, n]$ .

O caso fácil ocorre quando  $j = i - 1$ . Então, temos apenas a chave fictícia  $d_{i-1}$ . O custo esperado de pesquisa é  $e[i, i - 1] = q_{i-1}$ .

Quando  $j \geq i$ , precisamos selecionar uma raiz  $k_r$  entre  $k_i, \dots, k_j$ , e depois tornar uma árvore de pesquisa binária ótima com chaves  $k_i, \dots, k_{r-1}$  sua subárvore esquerda e uma árvore de pesquisa binária ótima com chaves  $k_{r+1}, \dots, k_j$  sua subárvore direita. O que acontece ao custo esperado de pesquisa de uma subárvore quando ela se torna uma subárvore de um nó? A profundidade de cada nó na subárvore aumenta em 1. Pela equação (15.16), o custo esperado de pesquisa dessa subárvore aumenta de acordo com a soma de todas as probabilidades na subárvore. Para uma subárvore com chaves  $k_i, \dots, k_j$ , vamos denotar essa soma de probabilidades como

$$w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l . \quad (15.17)$$

Desse modo, se  $k_r$  é a raiz de uma subárvore ótima contendo chaves  $k_i, \dots, k_j$ , temos

$$e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j)) .$$

Observando que

$$w(i, j) = w(i, r - 1) + p_r + w(r + 1, j) ,$$

Reescrevemos  $e[i, j]$  como

$$e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j) . \quad (15.18)$$

A equação recursiva (15.18) pressupõe que sabemos qual nó  $k_r$  usar como raiz. Escolhemos a raiz que fornece o custo esperado de pesquisa mais baixo, dando nossa formulação recursiva final:

$$e[i, j] = \begin{cases} q_{i-1} & \text{se } j = i - 1, \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{se } i \geq j. \end{cases} \quad (15.19)$$

Os  $e[i, j]$  valores dão os custos esperados de pesquisa em árvores de pesquisa binária ótimas. Para ajudar a controlar a estrutura de árvores de pesquisa binária ótimas, definimos  $raiz[i, j]$ , para  $1 \leq i \leq j \leq n$ , como o índice  $r$  para o qual  $k_r$  é a raiz de uma árvore de pesquisa binária ótima contendo chaves  $k_i, \dots, k_j$ . Veremos como calcular os valores de  $raiz[i, j]$ , mas vamos deixar a construção da árvore de pesquisa binária ótima a partir desses valores para o Exercício 15.5-1.

### Etapa 3: Cálculo do custo esperado de pesquisa de uma árvore de pesquisa binária ótima

Até aqui, você deve ter notado algumas semelhanças entre nossas caracterizações de árvores de pesquisa binária ótimas e multiplicação de cadeias de matrizes. Para ambos os domínios de problemas, nossos subproblemas consistem em subintervalos de índices contíguos. Uma implementação recursiva direta da equação (15.19) seria tão ineficiente quanto um algoritmo recursivo direto de multiplicação de cadeias de matrizes. Em vez disso, armazenamos os  $e[i, j]$  valores em uma tabela  $e[1..n+1, 0..n]$ . O primeiro índice precisa ir até  $n+1$  em vez de  $n$  porque, para ter uma subárvore contendo apenas a chave fictícia  $d_n$ , precisaremos calcular e armazenar  $e[n+1, n]$ . O segundo índice tem de começar a partir de 0 porque, para ter uma subárvore contendo apenas a chave fictícia  $d_0$ , precisaremos calcular e armazenar  $e[1, 0]$ . Usaremos somente as entradas  $e[i, j]$  para as quais  $j \geq i-1$ . Também empregaremos uma tabela  $raiz[i, j]$  para registrar a raiz da subárvore que contém as chaves  $k_i, \dots, k_j$ . Essa tabela só utiliza as entradas para as quais  $1 \leq i \leq j \leq n$ .

Vamos precisar de outra tabela para manter a eficiência. Em lugar de calcular o valor de  $w(i, j)$  a partir do início toda vez que estamos calculando  $e[i, j]$  – o que exigiria  $\Theta(j-i)$  adições – armazenamos esses valores em uma tabela  $w[1..n+1, 0..n]$ . No caso básico, calculamos  $w[i, i-1] = q_{i-1}$  para  $1 \leq i \leq n$ . Para  $j \geq i$ , calculamos

$$w[i, j] = w[i, j-1] + p_j + q_j. \quad (15.20)$$

Desse modo, podemos calcular os  $\Theta(n^2)$  valores de  $w[i, j]$  no tempo  $\Theta(1)$  para cada um deles.

O pseudocódigo a seguir toma como entradas as probabilidades  $p_1, \dots, p_n$  e  $q_0, \dots, q_n$  e o tamanho  $n$ , e retorna as tabelas  $e$  e  $raiz$ .

```

OPTIMAL-BST( $p, q, n$ )
1 for  $i \leftarrow 1$  to  $n+1$ 
2   do  $e[i, i-1] \leftarrow q_{i-1}$ 
3    $w[i, i-1] \leftarrow q_{i-1}$ 
4 for  $l \leftarrow 1$  to  $n$ 
5   do for  $i \leftarrow 1$  to  $n-l+1$ 
6     do  $j \leftarrow i+l-1$ 
7        $e[i, j] \leftarrow \infty$ 
8        $w[i, j] \leftarrow w[i, j-1] + p_j + q_j$ 
9       for  $r \leftarrow i$  to  $j$ 
10      do  $t \leftarrow e[i, r-1] + e[r+1, j] + w[i, j]$ 
11        if  $t < e[i, j]$ 
12          then  $e[i, j] \leftarrow t$ 
13           $raiz[i, j] \leftarrow r$ 
14 retornar  $e$  e  $raiz$ 
```

Pela descrição anterior e pela semelhança em relação ao procedimento MATRIX-CHAIN-ORDER da Seção 15.2, a operação desse procedimento deve ser bastante direta. O loop **for** das linhas 1 a 3 inicializa os valores de  $e[i, i - 1]$  e  $w[i, i - 1]$ . O loop **for** das linhas 4 a 13 usa então as recorrências (15.19) e (15.20) para calcular  $e[i, j]$  e  $w[i, j]$  para todo  $1 \leq i \leq j \leq n$ . Na primeira iteração, quando  $l = 1$ , o loop calcula  $e[i, i]$  e  $w[i, i]$  para  $i = 1, 2, \dots, n$ . A segunda iteração, com  $l = 2$ , calcula  $e[i, i + 1]$  e  $w[i, i + 1]$  para  $i = 1, 2, \dots, n - 1$  e assim por diante. O loop **for** mais interno, nas linhas 9 a 13, experimenta cada índice candidato  $r$  para determinar que chave  $k_r$  usar como a raiz de uma árvore de pesquisa binária ótima contendo chaves  $k_i, \dots, k_j$ . Esse loop **for** salva o valor atual do índice  $r$  em  $raiz[i, j]$  sempre que encontra uma chave melhor para usar como raiz.

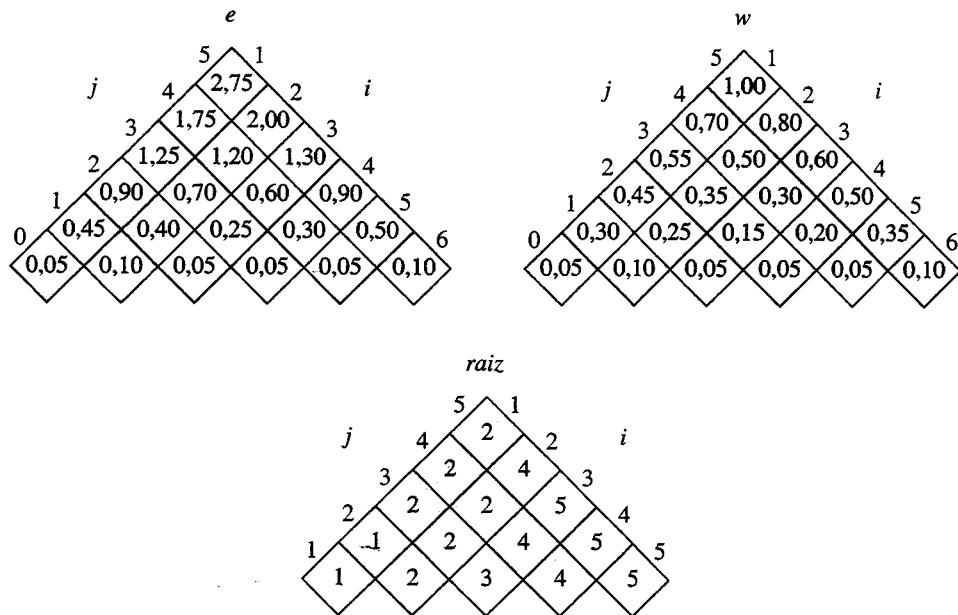


FIGURA 15.8 As tabelas  $e[i, j]$ ,  $w[i, j]$  e  $raiz[i, j]$  calculadas por OPTIMAL-BST sobre a distribuição de chaves mostrada na Figura 15.7. As tabelas foram giradas de forma que as diagonais fiquem dispostas horizontalmente

A Figura 15.8 mostra as tabelas  $e[i, j]$ ,  $w[i, j]$  e  $raiz[i, j]$  calculadas pelo procedimento OPTIMAL-BST sobre a distribuição de chaves mostrada na Figura 15.7. Como no exemplo de multiplicação de cadeias de matrizes, as tabelas foram giradas para dispor as diagonais horizontalmente. OPTIMAL-BST calcula as linhas de baixo para cima e da esquerda para a direita dentro de cada linha.

O procedimento OPTIMAL-BST demora o tempo  $\Theta(n^3)$ , exatamente como MATRIX-CHAIN-ORDER. É fácil verificar que o tempo de execução é  $O(n^3)$ , pois seus loops **for** estão aninhados na profundidade três e cada índice de loop exige no máximo  $n$  valores. Os índices de loops em OPTIMAL-BST não têm exatamente os mesmos limites que os de MATRIX-CHAIN-ORDER, mas eles estão dentro de no máximo 1 em todas as direções. Desse modo, como MATRIX-CHAIN-ORDER, o procedimento OPTIMAL-BST demora o tempo  $\Omega(n^3)$ .

## Exercícios

### 15.5-1

Escreva pseudocódigo para o procedimento CONSTRUCT-OPTIMAL-BST( $raiz$ ) que, dada a tabela  $raiz$ , forneça como saída a estrutura de uma árvore de pesquisa binária ótima. No exemplo da Figura 15.8, seu procedimento deve imprimir a estrutura

$k_2$  é a raiz  
 $k_1$  é o filho da esquerda de  $k_2$   
 $d_0$  é o filho da esquerda de  $k_1$   
 $d_1$  é o filho da direita de  $k_1$   
 $k_5$  é o filho da direita de  $k_2$   
 $k_4$  é o filho da esquerda de  $k_5$   
 $k_3$  é o filho da esquerda de  $k_4$   
 $d_2$  é o filho da esquerda de  $k_3$   
 $d_3$  é o filho da direita de  $k_3$   
 $d_4$  é o filho da direita de  $k_4$   
 $d_5$  é o filho da direita de  $k_5$

correspondente à árvore de pesquisa binária ótima mostrada na Figura 15.7(b).

### 15.5-2

Determine o custo e a estrutura de uma árvore de pesquisa binária ótima para um conjunto de  $n = 7$  chaves com as seguintes probabilidades:

$i$	0	1	2	3	4	5	6	7
$p_i$	0,04	0,06	0,08	0,02	0,10	0,12	0,14	
$q_i$	0,06	0,06	0,06	0,06	0,05	0,05	0,05	0,05

### 15.5-3

Suponha que, em vez de manter a tabela  $w[i, j]$ , calculássemos o valor de  $w(i, j)$  diretamente da equação (15.17) na linha 8 de OPTIMAL-BST e utilizássemos esse valor calculado na linha 10. Como essa mudança afetaria o tempo de execução assintótico de OPTIMAL-BST?

### 15.5-4 \*

Knuth [184] mostrou que sempre existem raízes de subárvores ótimas tais que  $raiz[i, j - 1] \leq raiz[i, j] \leq raiz[i + 1, j]$  para todo  $1 \leq i < j \leq n$ . Use esse fato para modificar o procedimento OPTIMAL-BST de forma que ele seja executado no tempo  $\Theta(n^2)$ .

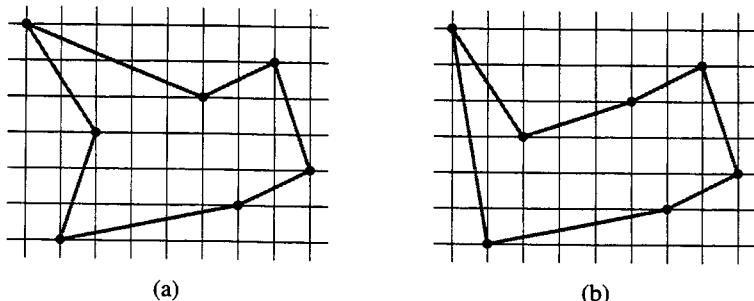
## Problemas

### 15-1 Problema euclidiano bitônico do caixeiro-viajante

O problema euclidiano do caixeiro-viajante é o problema de determinar o percurso fechado mais curto que conecta um dado conjunto de  $n$  pontos no plano. A Figura 15.9(a) mostra a solução para um problema de 7 pontos. O problema geral é NP-completo, e então se considera que sua solução exige mais que o tempo polinomial (ver Capítulo 34).

J. L. Bentley sugeriu que simplificássemos o problema restringindo nossa atenção a **percursos bitônicos**, isto é, percursos que começam no ponto mais à esquerda, seguem estritamente da esquerda para a direita até o ponto da extremidade direita, e depois seguem estritamente da direita para a esquerda, voltando ao ponto de partida. A Figura 15.9(b) mostra o percurso bitônico mais curto dos mesmos 7 pontos. Nesse caso, é possível um algoritmo de tempo polinomial.

Descreva um algoritmo de tempo  $O(n^2)$  para determinar um percurso bitônico ótimo. Considere por hipótese que não existem dois pontos com a mesma coordenada  $x$ . (Sugestão: Desloque-se da esquerda para a direita, mantendo possibilidades ótimas para as duas partes do percurso.)



**FIGURA 15.9** Sete pontos no plano, mostrados sobre uma grade unitária. (a) O percurso fechado mais curto, com comprimento aproximadamente igual a 24,89. Esse percurso não é bitônico. (b) O percurso bitônico mais curto para o mesmo conjunto de pontos. Seu comprimento é aproximadamente igual a 25,58.

## **15-2 Como imprimir nitidamente**

Considere o problema de imprimir nitidamente um parágrafo em uma impressora. O texto de entrada é uma seqüência de  $n$  palavras de comprimentos  $l_1, l_2, \dots, l_n$ , medidos em caracteres. Queremos imprimir esse parágrafo com nitidez em uma série de linhas que contêm no máximo  $M$  caracteres cada uma. Nossa critério de “nitidez” é dado a seguir. Se uma determinada linha contém palavras de  $i$  até  $j$ , onde  $i \leq j$ , e deixamos exatamente um espaço entre palavras, o número de caracteres de espaço extras no final da linha é  $M - j + i - \sum_{k=i}^j l_k$ , que deve ser não negativo para que as palavras caibam na linha. Desejamos minimizar a soma, sobre todas as linhas exceto a última, dos cubos dos números de caracteres de espaço extras nas extremidades das linhas. Forneça um algoritmo de programação dinâmica para imprimir um parágrafo de  $n$  palavras nitidamente em uma impressora. Analise o tempo de execução e os requisitos de espaço do seu algoritmo.

### **15-3 Distância de edição**

Para transformar uma cadeia de “origem”  $x[1..m]$  em uma nova cadeia de “destino”  $y[1..n]$ , podemos executar várias operações de transformação. Nossa meta é, dados  $x$  e  $y$ , produzir uma série de transformações que mudam  $x$  para  $y$ . Usamos um arranjo  $z$  – que supomos ser grande o suficiente para conter todos os caracteres de que precisará – para conter os resultados intermediários. Inicialmente,  $z$  está vazio e, em seu final, devemos ter  $z[j] = y[j]$  para  $j = 1, 2, \dots, n$ . Mantemos índices atuais  $i$  em  $x$  e  $j$  em  $z$ , e as operações têm permissão para alterar  $z$  e esses índices. Inicialmente,  $i = j = 1$ . Somos obrigados a examinar todo caractere em  $x$  durante a transformação, o que significa que, no fim da seqüência de operações de transformação, devemos ter  $i = m + 1$ .

Há seis operações de transformação:

**Copiar** um caractere de  $x$  para  $z$  definindo  $z[j] \leftarrow x[i]$  e, em seguida, incrementando  $i$  e  $j$ . Essa operação examina  $x[i]$ .

**Substituir** um caractere de  $x$  por outro caractere  $c$  definindo  $z[j] \leftarrow c$  e, em seguida, incrementando  $i$  e  $j$ . Essa operação examina  $x[i]$ .

**Excluir um caractere de  $x$  incrementando  $i$ , mas deixando  $i$  inalterado.** Essa operação examina  $x[i]$ .

Inserir o caractere  $c$  em  $z$  definindo  $z[j] \leftarrow c$ , e depois incrementando  $j$ , mas deixando  $i$  inalterado. Essa operação não examina nenhum caractere de  $x$ .

**Converter** (isto é, trocar) os dois caracteres seguintes, copiando-os de  $x$  para  $z$  mas na ordem oposta; fazemos isso definindo  $z[j] \leftarrow x[i + 1]$  e  $z[j + 1] \leftarrow x[i]$ , e depois definindo  $i \leftarrow i + 2$  e  $j \leftarrow j + 2$ . Essa operação examina  $x[i]$  e  $x[i + 1]$ .

**Eliminar o restante de  $x$  definindo  $i \leftarrow m + 1$ .** Essa operação examina todos os caracteres em  $x$  que ainda não foram examinados. Se essa operação for executada, ela deverá ser a última operação.

Como exemplo, um modo de transformar a cadeia de origem `algorithm` na cadeia de destino `altruistic` é usar a seqüência de operações a seguir, onde os caracteres sublinhados são  $x[i]$  e  $z[j]$  após a operação:

Operação	$x$	$z$
<i>cadeias iniciais</i>	<u>a</u> lgorithm	
copiar	algorith <u>m</u>	<u>a</u>
copiar	algorith <u>m</u>	<u>al</u>
substituir por t	algorith <u>m</u>	<u>alt</u>
excluir	algorith <u>m</u>	<u>alt</u>
copiar	algorith <u>m</u>	<u>altr</u>
inserir u	algorith <u>m</u>	<u>altru</u>
inserir i	algorith <u>m</u>	<u>altrui</u>
inserir s	algorith <u>m</u>	<u>altruis</u>
converter	algorith <u>m</u>	<u>altruisti</u>
inserir c	algorith <u>m</u>	<u>altruistic</u>
eliminar	algorith <u>m</u>	<u>altruistic</u>

Observe que há várias outras seqüências de operações de transformação que convertem `algorithm` em `altruistic`.

Cada uma das operações de transformação tem um custo associado. O custo de uma operação depende da aplicação específica, mas supomos que o custo de cada operação é uma constante conhecida para nós. Também supomos que os custos individuais das operações de copiar e substituir são menores que os custos combinados das operações de incluir e inserir; caso contrário, as operações de copiar e substituir não seriam usadas. O custo de uma dada seqüência de operações de transformação é a soma dos custos das operações individuais na seqüência. Para a seqüência anterior, o custo de transformar `algorithm` em `altruistic` é

$$(3 \cdot \text{custo(copiar)}) + \text{custo(substituir)} + \text{custo(excluir)} + (4 \cdot \text{custo(inserir)}) \\ + \text{custo(converter)} + \text{custo(eliminar)}$$

- a. Dadas duas seqüências  $x[1..m]$  e  $y[1..n]$  e um determinado conjunto de custos de operação, a **distância de edição** desde  $x$  até  $y$  é o custo da seqüência de transformação menos dispendiosa que converte  $x$  em  $y$ . Descreva um algoritmo de programação dinâmica para encontrar a distância de edição de  $x[1..m]$  até  $y[1..n]$  e imprima uma seqüência de transformação ótima. Analise o tempo de execução e os requisitos de espaço de seu algoritmo.

O problema da distância de edição é uma generalização do problema de alinhar duas seqüências de DNA (veja, por exemplo, Setubal e Meidanis [272, Seção 3.2]). Existem vários métodos para medir a semelhança de duas seqüências de DNA alinhando-as. Um dos métodos para alinhar duas seqüências  $x$  e  $y$  consiste em inserir espaços em posições arbitrárias nas duas seqüências (inclusive em qualquer extremidade) de forma que as seqüências resultantes  $x'$  e  $y'$  tenham o mesmo comprimento mas não apresentem um espaço na mesma posição (isto é, para nenhuma posição  $j$  tanto  $x'[j]$  quanto  $y'[j]$  são espaços.) Então, atribuímos uma “pontuação” a cada posição. A posição  $j$  recebe uma pontuação da seguinte maneira:

- $+1$  se  $x'[j] = y'[j]$  e nenhum deles é um espaço.
- $-1$  se  $x'[j] \neq y'[j]$  e nenhum deles é um espaço.
- $-2$  se  $x'[j]$  ou  $y'[j]$  é um espaço.

A pontuação do alinhamento é a soma das pontuações das posições individuais. Por exemplo, dadas as seqüências  $x = \text{GATCGGCAT}$  e  $y = \text{CAATGTGAATC}$ , um alinhamento é

G ATCG GCAT  
 CAAT GTGAATC  
 -\*++\*+\*\*+-+\*+

Um + em uma posição indica uma pontuação +1 para aquela posição, um - indica a pontuação -1 e um \* indica a pontuação -2, de forma que esse alinhamento tenha uma pontuação total igual a  $6 + 1 - 2 + 1 - 4 + 2 = -4$ .

- b.** Explique como lançar o problema de encontrar um alinhamento ótimo como problema de distância de edição usando um subconjunto das operações de transformação, copiar, substituir, excluir, inserir, converter e eliminar.

#### 15-4 Planejando uma festa da empresa

O professor Stewart presta consultoria ao presidente de uma corporação, que está planejando uma festa da empresa. A empresa tem uma estrutura hierárquica; isto é, a relação de supervisores forma uma árvore com raiz no presidente. O pessoal do escritório classificou cada funcionário com uma avaliação de sociabilidade, que é um número real. Para tornar a festa divertida para todos os participantes, o presidente não deseja que um funcionário e seu supervisor imediato participem.

O professor Stewart recebe a árvore que descreve a estrutura da corporação, usando a representação de filho da esquerda, irmão da direita descrita na Seção 10.4. Cada nó da árvore contém, além dos ponteiros, o nome de um funcionário e a ordem de sociabilidade desse funcionário. Descreva um algoritmo para compor uma lista de convidados que maximize a soma das avaliações de sociabilidade dos convidados. Analise o tempo de execução do seu algoritmo.

#### 15-5 Algoritmo de Viterbi

Podemos usar a programação dinâmica em um grafo orientado  $G = (V, E)$  para reconhecimento de voz. Cada aresta  $(u, v) \in E$  é identificada por um som  $\sigma(u, v)$  de um conjunto finito  $\Sigma$  de sons. O grafo identificado é um modelo formal de uma pessoa falando uma linguagem restrita. Cada caminho no grafo a partir de um vértice distinto  $v_0 \in V$  corresponde a uma seqüência possível de sons produzidos pelo modelo. A identificação de um caminho orientado é definida como a concatenação das identificações das arestas nesse caminho.

- a.** Descreva um algoritmo eficiente que, dado um grafo com arestas identificadas  $G$  contendo um vértice distinto  $v_0$  e uma seqüência  $s = \langle \sigma_1, \sigma_2, \dots, \sigma_k \rangle$  de caracteres a partir de  $\Sigma$ , retorne um caminho em  $G$  que comece em  $v_0$  e tenha  $s$  como sua identificação, se existir tal caminho. Caso contrário, o algoritmo deve retornar NO-SUCH-PATH. Analise o tempo de execução do seu algoritmo. (Sugestão: Talvez você considere úteis os conceitos do Capítulo 22.)

Agora, vamos supor que toda aresta  $(u, v) \in E$  também tenha associada a ela uma probabilidade não negativa  $p(u, v)$  de percorrer a aresta  $(u, v)$  desde o vértice  $u$ , e assim produzir o som correspondente. A soma das probabilidades das arestas que saem de cada vértice é igual a 1. A probabilidade de um caminho é definida como o produto das probabilidades de suas arestas. Podemos ver a probabilidade de um caminho que começa em  $v_0$  como a probabilidade de um “percurso aleatório” começando em  $v_0$  seguir o caminho especificado, onde a escolha de qual aresta tomar em um vértice  $u$  é feita de modo probabilístico, de acordo com as probabilidades das arestas disponíveis saindo de  $u$ .

- b.** Amplie sua resposta à parte (a), de modo que, se um caminho for retornado, ele seja um *caminho mais provável* começando em  $v_0$  e tendo a identificação  $s$ . Analise o tempo de execução do seu algoritmo.

#### 15-6 Movimentação em um tabuleiro de damas

Suponha que você recebe um tabuleiro de damas de  $n \times n$  e uma pedra de jogo de damas. Você deve mover a pedra de jogo de damas desde a borda inferior do tabuleiro até a borda superior, de acordo com a regra a seguir. Em cada etapa, a pedra de jogo de damas pode se deslocar até um destes três quadrados:

1. O quadrado imediatamente acima.
2. O quadrado uma posição acima e à esquerda (mas apenas se a pedra de jogo de damas ainda não estiver na coluna mais à esquerda).
3. O quadrado uma posição acima e à direita (mas apenas se a pedra de jogo de damas ainda não estiver na coluna mais à direita).

Toda vez que se desloca do quadrado  $x$  para o quadrado  $y$ , você recebe  $p(x, y)$  reais. Você recebe  $p(x, y)$  para todos os pares  $(x, y)$  para os quais um movimento de  $x$  até  $y$  é válido. Não pressuponha que  $p(x, y)$  é positivo.

Forneça um algoritmo que descubra o conjunto de movimentos que deslocarão a pedra de jogo de damas de algum lugar ao longo da borda inferior até algum lugar ao longo da borda superior, ao mesmo tempo que ganha a maior quantidade possível de reais. Seu algoritmo é livre para escolher qualquer quadrado ao longo da borda inferior como ponto de partida e qualquer quadrado ao longo da borda superior como destino, a fim de maximizar o número de reais ganhos ao longo do caminho. Qual é o tempo de execução de seu algoritmo?

### **15-7 Programando para maximizar o lucro**

Suponha que você tem uma máquina e um conjunto de  $n$  trabalhos  $a_1, a_2, \dots, a_n$  para processar nessa máquina. Cada trabalho  $a_j$  tem um tempo de processamento  $t_j$ , um lucro  $p_j$  e um prazo final  $d_j$ . A máquina só pode processar um trabalho de cada vez, e o trabalho  $a_j$  deve ser executado ininterruptamente para  $t_j$  unidades de tempo consecutivas. Se o trabalho  $a_j$  for concluído em seu prazo  $d_j$ , você recebe um lucro  $p_j$ , mas, se ele for completado depois de seu prazo final, você recebe um lucro 0. Forneça um algoritmo para encontrar a programação que obtenha a quantidade máxima de lucro, supondo que todos os tempos de processamento são inteiros entre 1 e  $n$ . Qual é o tempo de execução de seu algoritmo?

## **Notas do capítulo**

R. Bellman começou o estudo sistemático de programação dinâmica em 1955. A palavra “programação”, tanto aqui quanto em programação linear, se refere ao uso de um método de solução tabular. Embora as técnicas de otimização incorporando elementos de programação dinâmica fossem conhecidas antes, Bellman proporcionou à área uma sólida base matemática [34].

Hu e Shing [159, 160] apresentam um algoritmo de tempo  $O(n \lg n)$  para o problema de multiplicação de cadeias de matrizes.

O algoritmo de tempo  $O(mn)$  para o problema da subseqüência comum mais longa parece ser um algoritmo popular. Knuth [463] levantou a questão da existência ou não de algoritmos subquadráticos para o problema da LCS. Masek e Paterson [212] responderam afirmativamente a essa pergunta, fornecendo um algoritmo que é executado no tempo  $O(mn/\lg n)$ , onde  $n \leq m$  e as seqüências são extraídas de um conjunto de tamanho limitado. Para o caso especial no qual nenhum elemento aparece mais de uma vez em uma seqüência de entrada, Szymanski [288] mostra que o problema pode ser resolvido no tempo  $O((n+m)\lg(n+m))$ . Muitos desses resultados se estendem ao problema de calcular distâncias de edição de cadeias (Problema 15-3).

Um trabalho anterior sobre codificações binárias de comprimento variável apresentado por Gilbert e Moore [114] teve aplicações na construção de árvores de pesquisa binária ótimas para o caso em que todas as probabilidades  $p_i$  são 0; esse trabalho contém um algoritmo de tempo  $O(n^3)$ . Aho, Hopcroft e Ullman [5] apresentam o algoritmo da Seção 15.5. O Exercício 15.5-4 se deve a Knuth [184]. Hu e Tucker [161] criaram um algoritmo para o caso em que todas as probabilidades  $p_i$  são 0 e que utiliza o tempo  $O(n^2)$  e o espaço  $O(n)$ ; mais tarde, Knuth [185] reduziu o tempo para  $O(n \lg n)$ .

---

## *Capítulo 16*

### *Algoritmos gulosos*

Em geral, os algoritmos relacionados a problemas de otimização funcionam através de uma sequência de passos, com um conjunto de opções (escolhas) em cada passo. Para muitos problemas de otimização, é um exagero utilizar a programação dinâmica para descobrir as melhores escolhas: algoritmos mais simples e mais eficientes darão conta da mesma tarefa. Um **algoritmo guloso** sempre faz a escolha que parece ser a melhor no momento. Isto é, ele faz uma escolha ótima para as condições locais, na esperança de que essa escolha leve a uma solução ótima para a situação global. Este capítulo explora problemas de otimização que podem ser solucionados por meio de algoritmos gulosos. Antes de ler este capítulo, você deve ler sobre programação dinâmica no Capítulo 15, em particular na Seção 15.3.

Os algoritmos gulosos nem sempre produzem soluções ótimas, mas para muitos problemas eles são úteis. Primeiro, examinaremos na Seção 16.1 um problema simples mas não trivial, o problema de seleção de atividade, para o qual um algoritmo guloso calcula de modo eficiente uma solução. Chegaremos ao algoritmo guloso considerando primeiro uma solução de programação dinâmica, e depois mostrando que sempre podemos fazer escolhas gulosas para chegar a uma solução ótima. A Seção 16.2 revê os elementos básicos da abordagem gulosa, dando um enfoque mais direto ao fornecimento de algoritmos gulosos corretos que ao processo de programação dinâmica da Seção 16.1. A Seção 16.3 apresenta uma aplicação importante das técnicas gulosas: o projeto de códigos de compressão de dados (Huffman). Na Seção 16.4, investigamos uma parte da teoria subjacente às estruturas combinatórias chamadas “matróides”, para as quais os algoritmos gulosos sempre produzem uma solução ótima. Finalmente, a Seção 16.5 ilustra a aplicação de matróides usando o problema da programação de tarefas por unidade de tempo com prazos finais e penalidades.

O método guloso é bastante eficiente e funciona bem para uma ampla variedade de problemas. Capítulos posteriores apresentarão muitos algoritmos que podem ser vistos como aplicações do método guloso, inclusive algoritmos de árvore espalhada mínima (Capítulo 23), o algoritmo de Dijkstra para caminhos mais curtos a partir de uma única origem (Capítulo 24) e a heurística gulosa de cobertura de conjuntos de Chvátal (Capítulo 35). Os algoritmos de árvores espalhadas mínimas são um exemplo clássico do método guloso. Embora este capítulo e o Capítulo 23 possam ser lidos independentemente um do outro, você poderá achar útil ler os dois em conjunto.

## 16.1 Um problema de seleção de atividade

Nosso primeiro exemplo é o problema de programar um recurso entre diversas atividades concorrentes. Descobriremos que um algoritmo guloso fornece um método elegante e simples para selecionar um conjunto de tamanho máximo de atividades mutuamente compatíveis. Vamos supor que temos um conjunto  $S = \{a_1, a_2, \dots, a_n\}$  de  $n$  *atividades* propostas que desejam usar um recurso, como uma sala de conferências, o qual só pode ser utilizado por uma única atividade de cada vez. Cada atividade  $a_i$  tem um *tempo de início*  $s_i$  e um *tempo de término*  $f_i$ , onde  $0 \leq s_i < f_i < \infty$ . Se selecionada, a atividade  $a_i$  ocorre durante o intervalo de tempo meio aberto  $[s_i, f_i)$ . As atividades  $a_i$  e  $a_j$  são *compatíveis* se os intervalos  $[s_i, f_i)$  e  $[s_j, f_j)$  não se superpõem (isto é,  $a_i$  e  $a_j$  são compatíveis se  $s_i \geq f_j$  ou  $s_j \geq f_i$ ). O *problema de seleção de atividade* consiste em selecionar um subconjunto de tamanho máximo de atividades mutuamente compatíveis. Por exemplo, considere o seguinte conjunto  $S$  de atividades, que colocamos em ordem monotonicamente crescente de tempo de término:

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	8	9	10	11	12	13	14

(Veremos em breve por que é vantajoso considerar atividades em seqüência ordenada.) Para este exemplo, o subconjunto  $\{a_3, a_9, a_{11}\}$  consiste em atividades mutuamente compatíveis. Porém, ele não é um subconjunto máximo, pois o subconjunto  $\{a_1, a_4, a_8, a_{11}\}$  é maior. De fato, o conjunto  $\{a_1, a_4, a_8, a_{11}\}$  é um subconjunto maior de atividades mutuamente compatíveis; outro subconjunto maior é  $\{a_2, a_4, a_9, a_{11}\}$ .

Resolveremos esse problema em várias etapas. Começamos formulando uma solução de programação dinâmica para esse problema, na qual combinamos soluções ótimas para dois subproblemas, a fim de formar uma solução ótima para o problema original. Consideraremos diversas escolhas ao determinar quais subproblemas usar em uma solução ótima. Então, observaremos que só precisamos considerar uma escolha – a escolha gúloso – e que, ao optarmos pela escolha gúloso, um dos subproblemas tem a garantia de ser vazio, de forma que só resta um subproblema não vazio. Com base nessas observações, desenvolveremos um algoritmo gúloso recursivo para resolver o problema de tempo de atividades. Completaremos o processo de desenvolver uma solução gúloso convertendo o algoritmo recursivo em um algoritmo interativo. Embora as etapas que percorreremos nesta seção sejam mais complicadas do que é comum no desenvolvimento de um algoritmo gúloso, elas ilustram o relacionamento entre algoritmos gúlosos e programação dinâmica.

### A subestrutura ótima do problema de seleção de atividades

Conforme mencionamos antes, começamos desenvolvendo uma solução de programação dinâmica para o problema de seleção de atividades. Como no Capítulo 15, nosso primeiro passo é encontrar a subestrutura ótima, e depois usá-la para construir uma solução ótima para o problema a partir de soluções ótimas para subproblemas.

Vimos no Capítulo 15 que precisamos definir um espaço de subproblemas apropriado. Vamos começar definindo conjuntos

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\},$$

de forma que  $S_{ij}$  seja o subconjunto de atividades em  $S$  que podem começar após a atividade  $a_i$  terminar, e que terminam antes da atividade  $a_j$  começar. De fato,  $S_{ij}$  consiste em todas as ativi-

des compatíveis com  $a_i$  e  $a_j$ , e que também são compatíveis com todas as atividades que não terminam depois de  $a_i$  terminar e com todas as atividades que não começam antes de  $a_j$  começar. Para representar o problema inteiro, adicionamos atividades fictícias  $a_0$  e  $a_{n+1}$ , e adotamos as convenções de que  $f_0 = 0$  e  $s_{n+1} = \infty$ . Então  $S = S_{0, n+1}$ , e os intervalos para  $i$  e  $j$  são dados por  $0 \leq i, j \leq n + 1$ .

Podemos restringir ainda mais os intervalos de  $i$  e  $j$  como a seguir. Vamos supor que as atividades estejam dispostas em ordem monotonicamente crescente de tempo de término:

$$f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n < f_{n+1} \quad (16.1)$$

Afirmamos que  $S_{ij} = \emptyset$  sempre que  $i \geq j$ . Por quê? Suponha que exista uma atividade  $a_k \in S_{ij}$  para algum  $i \geq j$ , de forma que  $a_i$  segue  $a_j$  na seqüência ordenada. Então teríamos  $f_i \leq s_k < f_k \leq s_j < f_j$ . Desse modo,  $f_i < f_j$ , o que contradiz nossa hipótese de que  $a_i$  segue  $a_j$  na seqüência ordenada. Podemos concluir que, supondo que ordenamos as atividades em ordem monotonicamente crescente de tempo de término, nosso espaço de subproblemas é selecionar um subconjunto de tamanho máximo de atividades mutuamente compatíveis de  $S_{ij}$ , para  $0 \leq i < j \leq n + 1$ , sabendo que todos os outros  $S_{ij}$  são vazios.

Para ver a subestrutura do problema de seleção de atividades, considere algum subproblema não vazio  $S_{ij}$ ,<sup>1</sup> e suponha que uma solução para  $S_{ij}$  inclua alguma atividade  $a_k$ , de forma que  $f_i \leq s_k < f_k \leq s_j$ . O uso da atividade  $a_k$  gera dois subproblemas,  $S_{ik}$  (atividades que começam depois de  $a_i$  terminar e terminam antes de  $a_k$  começar) e  $S_{kj}$  (atividades que começam depois de  $a_k$  terminar e terminam antes de  $a_j$  começar), cada uma das quais consiste em um subconjunto das atividades em  $S_{ij}$ . Nossa solução para  $S_{ij}$  é a união das soluções para  $S_{ik}$  e  $S_{kj}$ , juntamente com a atividade  $a_k$ . Desse modo, o número de atividades em nossa solução para  $S_{ij}$  é o tamanho de nossa solução para  $S_{ik}$ , mais o tamanho de nossa solução para  $S_{kj}$ , mais uma unidade (para  $a_k$ ).

A subestrutura ótima desse problema é dada a seguir. Suponha agora que uma solução ótima  $A_{ij}$  para  $S_{ij}$  inclua a atividade  $a_k$ . Então, as soluções  $A_{ik}$  para  $S_{ik}$  e  $A_{kj}$  para  $S_{kj}$  usadas dentro dessa solução ótima para  $S_{ij}$  também devem ser ótimas. O argumento habitual de recortar e colar se aplica. Se tivéssemos uma solução  $A'_{ik}$  para  $S_{ik}$  que incluisse mais atividades que  $A_{ik}$ , poderíamos recortar  $A_{ik}$  de  $A_{ij}$  e colar em  $A'_{ik}$ , produzindo assim outra solução para  $S_{ij}$  com mais atividades que  $A_{ij}$ . Como supomos que  $A_{ij}$  é uma solução ótima, derivamos uma contradição. De modo semelhante, se tivéssemos uma solução  $A'_{kj}$  para  $S_{kj}$  com mais atividades que  $A_{kj}$ , poderíamos substituir  $A_{kj}$  por  $A'_{kj}$  para produzir uma solução para  $S_{ij}$  com mais atividades que  $A_{ij}$ .

Agora, usamos nossa subestrutura ótima para mostrar que podemos construir uma solução ótima para o problema a partir de soluções ótimas para subproblemas. Vimos que qualquer solução para um subproblema não vazio  $S_{ij}$  inclui alguma atividade  $a_k$ , e que qualquer solução ótima contém dentro dela soluções ótimas para instâncias de subproblemas  $S_{ik}$  e  $S_{kj}$ . Desse modo, podemos construir um subconjunto de tamanho máximo de atividades mutuamente compatíveis em  $S_{ij}$  dividindo o problema em dois subproblemas (encontrando subconjuntos de tamanho máximo de atividades mutuamente compatíveis em  $S_{ik}$  e  $S_{kj}$ ), encontrando subconjuntos de tamanho máximo  $A_{ik}$  e  $A_{kj}$  de atividades mutuamente compatíveis para esses subproblemas, e formando nosso subconjunto de tamanho máximo  $A_{ij}$  de atividades mutuamente compatíveis como

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}. \quad (16.2)$$

Uma solução ótima para o problema inteiro é uma solução para  $S_{0, n+1}$ .

---

<sup>1</sup> Às vezes, mencionaremos os conjuntos  $S_{ij}$  como subproblemas, em vez de apenas conjuntos de atividades. Sempre ficará claro a partir do contexto se estamos nos referindo a  $S_{ij}$  como um conjunto de atividades ou como o subproblema cuja entrada é esse conjunto.

## Uma solução recursiva

O segundo passo no desenvolvimento de uma solução de programação dinâmica é definir recursivamente o valor de uma solução ótima. Para o problema de seleção de atividades, seja  $c[i, j]$  o número de atividades em um subconjunto de tamanho máximo de atividades mutuamente compatíveis em  $S_{ij}$ . Temos  $c[i, j] = 0$  sempre que  $S_{ij} = \emptyset$ ; em particular,  $c[i, j] = 0$  para  $i \geq j$ .

Agora, considere um subconjunto não vazio  $S_{ij}$ . Como vimos, se  $\alpha_k$  é usado em um subconjunto de tamanho máximo de atividades mutuamente compatíveis de  $S_{ij}$ , também usamos subconjuntos de tamanho máximo de atividades mutuamente compatíveis para os subproblemas  $S_{ik}$  e  $S_{kj}$ . Usando a equação (16.2), temos a recorrência

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

Essa equação recursiva pressupõe que conhecemos o valor de  $k$ , o que não é verdade. Há  $j - i - 1$  valores possíveis para  $k$ , isto é,  $k = i + 1, \dots, j - 1$ . Tendo em vista que o subconjunto de tamanho máximo de  $S_{ij}$  deve usar um desses valores para  $k$ , conferimos todos eles para encontrar o melhor. Desse modo, nossa definição recursiva completa de  $c[i, j]$  se torna

$$c[i, j] = \begin{cases} 0 & \text{se } S_{ij} = \emptyset, \\ \max_{i \leq k \leq j} \{c[i, k] + c[k, j] + 1\} & \text{se } S_{ij} \neq \emptyset. \end{cases} \quad (16.3)$$

## Convertendo uma solução de programação dinâmica em uma solução gulosa

Neste momento, seria um exercício direto escrever um algoritmo de programação dinâmica tabular, bottom-up, baseado na recorrência (16.3). De fato, o Exercício 16.1-1 lhe pede para fazer exatamente isso. Entretanto, existem duas observações fundamentais que nos permitem simplificar nossa solução.

### **Teorema 16.1**

Considere qualquer subproblema não vazio  $S_{ij}$ , e seja  $\alpha_m$  a atividade em  $S_{ij}$  com o tempo de término mais antigo:

$$f_m = \min \{f_k : \alpha_k \in S_{ij}\}.$$

Então

1. A atividade  $\alpha_m$  é usada em algum subconjunto de tamanho máximo de atividades mutuamente compatíveis de  $S_{ij}$ .
2. O subproblema  $S_{im}$  é vazio, de forma que a escolha de  $\alpha_m$  deixa o subproblema  $S_{mj}$  como o único que pode ser não vazio.

**Prova** Provaremos a segunda parte primeiro, pois ela é um pouco mais simples. Suponha que  $S_{im}$  é não vazio, de forma que existe alguma atividade  $\alpha_k$  tal que  $f_i \leq s_k < f_k \leq s_m < f_m$ . Então,  $\alpha_k$  também está em  $S_{ij}$  e tem um tempo de término anterior à de  $\alpha_m$ , o que contradiz nossa escolha de  $\alpha_m$ . Concluímos que  $S_{im}$  é vazio.

Para provar a primeira parte, supomos que  $A_{ij}$  é um subconjunto de tamanho máximo de atividades mutuamente compatíveis de  $S_{ij}$ , e vamos ordenar as atividades em  $A_{ij}$  em ordem monotonicamente crescente de tempo de término. Seja  $\alpha_k$  a primeira atividade em  $A_{ij}$ . Se  $\alpha_k = \alpha_m$ , terminamos, pois mostramos que  $\alpha_m$  é usada em algum subconjunto de tamanho máximo de | 299

atividades mutuamente compatíveis de  $S_{ij}$ . Se  $\alpha_k \neq \alpha_m$ , construímos o subconjunto  $A'_{ij} = \{\alpha_k\} \cup \{\alpha_m\}$ . As atividades em  $A'_{ij}$  são disjuntas, pois as atividades em  $A_{ij}$  o são,  $\alpha_k$  é a primeira atividade em  $A_{ij}$  a terminar e  $f_m \leq f_k$ . Observando que  $A'_{ij}$  tem o mesmo número de atividades de  $A_{ij}$ , vemos que  $A'_{ij}$  é um subconjunto de tamanho máximo de atividades mutuamente compatíveis de  $S_{ij}$  que inclui  $\alpha_m$ . ■

Por que o Teorema 16.1 é tão valioso? Vimos na Seção 15.3 que a subestrutura ótima varia na quantidade de subproblemas usados em uma solução ótima para o problema original e em quantas escolhas temos na determinação de quais subproblemas usar. Em nossa solução de programação dinâmica, dois subproblemas são usados em uma solução ótima, e existem  $j - i - 1$  escolhas ao se resolver o subproblema  $S_{ij}$ . O Teorema 16.1 reduz ambas as quantidades de forma significativa: apenas um subproblema é usado em uma solução ótima (o outro subproblema tem a garantia de ser vazio) e, quando resolvemos o subproblema  $S_{ij}$ , só precisamos considerar uma escolha: a que tem o tempo de término mais antigo em  $S_{ij}$ . Felizmente, podemos determinar com facilidade que atividade é essa.

Além de reduzir o número de subproblemas e o número de escolhas, o Teorema 16.1 produz outro benefício: podemos resolver cada subproblema de cima para baixo, em vez de usar a solução de baixo para cima tipicamente empregada em programação dinâmica. Para resolver o subproblema  $S_{ij}$ , escolhemos a atividade  $\alpha_m$  de  $S_{ij}$  com o tempo de término mais antigo e adicionamos a essa solução o conjunto de atividades usadas em uma solução ótima para o subproblema  $S_{mj}$ . Como sabemos que, tendo escolhido  $\alpha_m$ , certamente usaremos uma solução para  $S_{mj}$  em nossa solução ótima para  $S_{ij}$ , não precisamos resolver  $S_{mj}$  antes de resolver  $S_{ij}$ . Para resolver  $S_{ij}$ , podemos *primeiro* escolher  $\alpha_m$  como a atividade em  $S_{ij}$  com o tempo de término mais antigo, e *depois* resolver  $S_{mj}$ .

Observe também que existe um padrão para os subproblemas que resolvemos. Nosso problema original é  $S = S_{0, n+1}$ . Suponha que escolhemos  $\alpha_{m_1}$  como a atividade em  $S_{0, n+1}$  com o tempo de término mais antigo. (Como ordenamos atividades por tempos de término monotonicamente crescentes e  $f_0 = 0$ , devemos ter  $m_1 = 1$ .) Nossa próximo subproblema é  $S_{m_1, n+1}$ . Agora, suponha que escolhemos  $\alpha_{m_2}$  como a atividade em  $S_{m_1, n+1}$  com o tempo de término mais antigo. (Não temos necessariamente  $m_2 = 2$ .) Nossa próximo subproblema é  $S_{m_2, n+1}$ . Continuando, vemos que cada subproblema terá a forma  $S_{m_1, n+1}$  para algum número de atividade  $m_i$ . Em outras palavras, cada subproblema consiste nas últimas atividades a terminar, e o número de tais atividades varia de um subproblema para outro.

Também há um padrão nas atividades que escolhemos. Como sempre escolhemos a atividade com o tempo de término mais antigo em  $S_{m_1, n+1}$ , os tempos de término das atividades escolhidas sobre todos os subproblemas serão estritamente crescentes ao longo do tempo. Além disso, podemos considerar cada atividade apenas uma vez de modo geral, na ordem monotonicamente crescente de tempos de término.

A atividade  $\alpha_m$  que escolhemos ao resolver um subproblema é sempre aquela com o tempo de término mais antigo que pode ser programado legalmente. A atividade escolhida é portanto uma escolha “gulosa” no sentido de que, intuitivamente, ela deixa tanta oportunidade quanto possível para que as atividades restantes sejam programadas. Isto é, a escolha gulosa é a que maximiza a quantidade de tempo restante não programado.

## Um algoritmo guloso recursivo

Agora que vimos como simplificar nossa solução de programação dinâmica e como tratá-la como um método top-down, estamos prontos para ver um algoritmo que funciona de modo puramente guloso, de cima para baixo. Daremos uma solução recursiva direta sob a forma do procedimento **RECURSIVE-ACTIVITY-SELECTOR**. Ele toma os tempos de início e término das atividades, representados como arranjos  $s$  e  $f$ , bem como os índices iniciais  $i$  e  $j$  do subproblema  $S_{i,j}$  que ele deve resolver. O procedimento retorna um conjunto de tamanho máximo de atividades

mutuamente compatíveis em  $S_{i,j}$ . Supomos que as  $n$  atividades de entrada estão ordenadas por tempo de término monotonicamente crescente, de acordo com a equação (16.1). Se não, podemos ordená-las nessa ordem no tempo  $O(n \lg n)$ , dividindo os intervalos arbitrariamente. A chamada inicial é RECUSIVE-ACTIVITY-SELECTOR( $s, f, 0, n + 1$ ).

```

RECUSIVE-ACTIVITY-SELECTOR( $s, f, i, j$ )
1  $m \leftarrow i + 1$ 
2 while  $m < j$  e  $s_m < f_i$             $\triangleright$  Encontrar a primeira atividade em  $S_{ij}$ .
3   do  $m \leftarrow m + 1$ 
4 if  $m < j$ 
5   then return  $\{a_m\} \leftarrow$  RECUSIVE-ACTIVITY-SELECTOR( $s, f, m, j$ )
6 else return  $\emptyset$ 
```

A Figura 16.1 mostra a operação do algoritmo. Em uma dada chamada recursiva RECUSIVE-ACTIVITY-SELECTOR( $s, f, i, j$ ), o loop **while** das linhas 2 e 3 procura pela primeira atividade em  $S_{ij}$ . O loop examina  $a_{i+1}, a_{i+2}, \dots, a_{j-1}$ , até encontrar a primeira atividade  $a_m$  que seja compatível com  $a_i$ ; tal atividade tem  $s_m \geq f_i$ . Se o loop terminar porque encontra tal atividade, o procedimento retorna na linha 5 a união de  $\{a_m\}$  com o subconjunto de tamanho máximo de  $S_{mj}$  retornado pela chamada recursiva RECUSIVE-ACTIVITY%SELECTOR( $s, f, m, j$ ). Como outra alternativa, o loop pode terminar porque  $m \geq j$ , e nesse caso examinamos todas as atividades cujos tempos de término são anteriores à de  $a_j$ , sem encontrar uma que seja compatível com  $a_i$ . Nesse caso,  $S_{ij} = \emptyset$ , e então o procedimento retorna  $\emptyset$  na linha 6.

Supondo que as atividades já foram ordenadas por tempos de término, o tempo de execução da chamada RECUSIVE-ACTIVITY-SELECTOR( $s, f, 0, n + 1$ ) é  $\Theta(n)$ , o que podemos verificar como a seguir. Em todas as chamadas recursivas, cada atividade é examinada exatamente uma vez no teste do loop **while** da linha 2. Em particular, a atividade  $a_k$  é examinada na última chamada feita em que  $i < k$ .

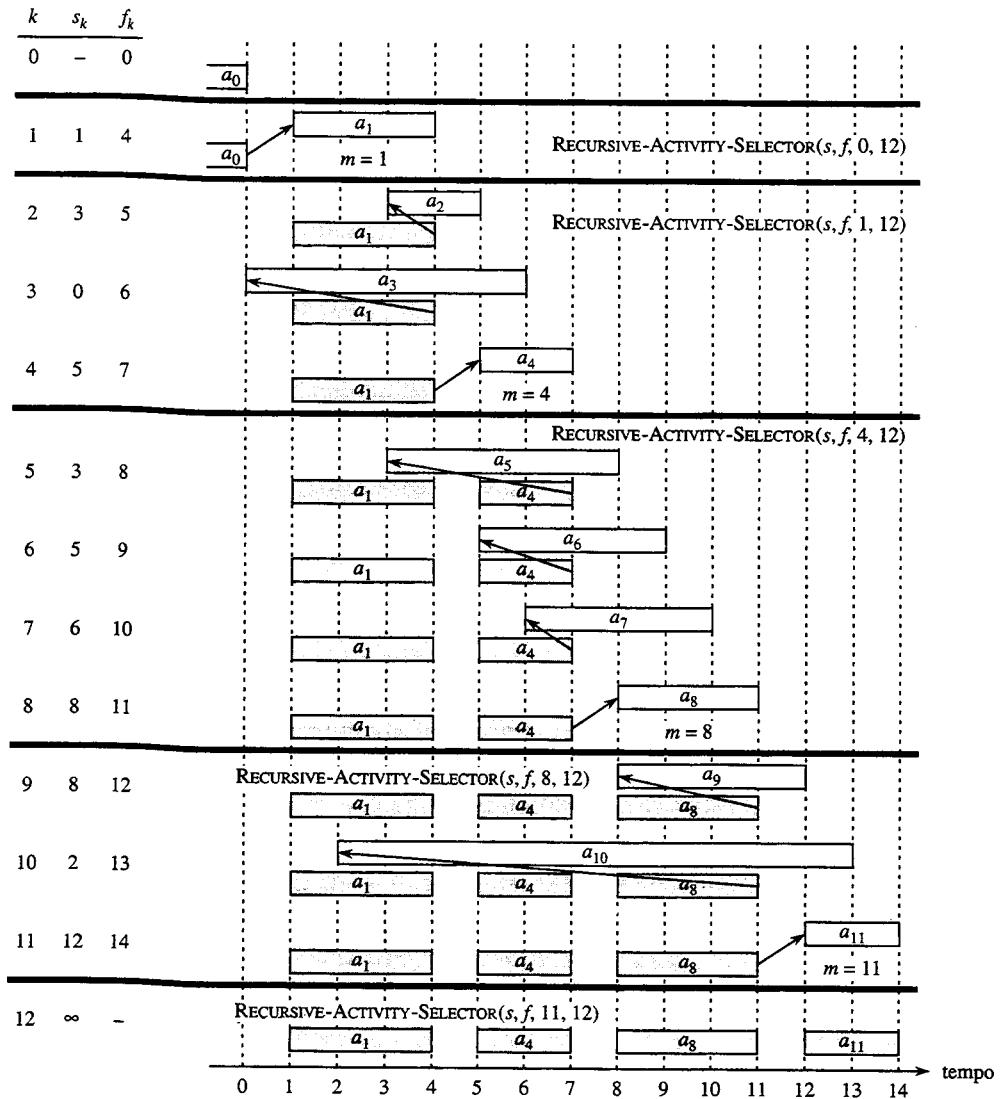
## Um algoritmo guloso iterativo

Podemos converter facilmente nosso procedimento recursivo em um iterativo. O procedimento RECUSIVE-ACTIVITY-SELECTOR é quase “recursivo de final” (consulte o Problema 7-4): ele termina com uma chamada recursiva a ele próprio, seguida por uma operação de união. Em geral, é uma tarefa direta transformar um procedimento recursivo de final para uma forma iterativa; de fato, alguns compiladores para certas linguagens de programação executam essa tarefa automaticamente. Como está escrito, RECUSIVE-ACTIVITY-SELECTOR funciona para qualquer subproblema  $S_{ij}$ , mas vimos que só é preciso considerar subproblemas para os quais  $j = n + 1$ , isto é, subproblemas que consistem nas últimas atividades a terminar.

O procedimento GREEDY-ACTIVITY-SELECTOR é uma versão iterativa do procedimento RECUSIVE-ACTIVITY-SELECTOR. Ele também pressupõe que as atividades de entrada estão ordenadas por tempo de término monotonicamente crescente. Ele reúne atividades selecionadas em um conjunto  $A$  e retorna esse conjunto quando termina.

```

GREEDY-ACTIVITY-SELECTOR( $s, f$ )
1  $n \leftarrow$  comprimento[ $s$ ]
2  $A \leftarrow \{1\}$ 
3  $i \leftarrow 1$ 
4 for  $m \leftarrow 2$  to  $n$ 
5   do if  $s_m \geq f_i$ 
6     then  $A \leftarrow A \cup \{a_m\}$ 
7      $i \leftarrow m$ 
8 return  $A$ 
```



**FIGURA 16.1** A operação de GREEDY-ACTIVITY-SELECTOR sobre 11 atividades dadas anteriormente. As atividades consideradas em cada chamada recursiva aparecem entre linhas horizontais. A atividade fictícia  $a_0$  termina no tempo 0 e, na chamada inicial,  $\text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, 0, 12)$ , a atividade  $a_1$  é selecionada. Em cada chamada recursiva, as atividades que já foram selecionadas estão sombreadas, e a atividade mostrada em branco está sendo considerada. Se o tempo de início de uma atividade ocorre antes do tempo de término da atividade mais recentemente adicionada (a seta entre elas aponta para a esquerda), ela é rejeitada. Caso contrário (a seta aponta diretamente para cima ou para a direita), ela é selecionada. A última chamada recursiva,  $\text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, 11, 12)$ , retorna  $\emptyset$ . O conjunto resultante de atividades selecionadas é  $\{a_1, a_4, a_8, a_{11}\}$

O procedimento funciona da maneira descrita a seguir. A variável  $i$  indexa a adição mais recente à  $A$ , correspondendo à atividade  $a_i$  na versão recursiva. Como as atividades são consideradas em ordem de tempo de término monotonicamente crescente,  $f_i$  é sempre o tempo de término máximo de qualquer atividade em  $A$ . Isto é,

$$f_i = \max \{f_k : a_k \in A\} . \quad (16.4)$$

As linhas 2 e 3 selecionam a atividade  $a_1$ , inicializam  $A$  para conter apenas essa atividade e iniciam  $i$  para indexar essa atividade. O loop **for** das linhas 4 a 7 encontra a atividade mais antiga a terminar em  $S_{i, n+1}$ . O loop considera cada atividade  $a_m$  por sua vez e adiciona  $a_m$  a  $A$  se ela é compatível com todas as atividades selecionadas anteriormente; tal atividade é a mais antiga a

terminar em  $S_{i, n+1}$ . Para ver se a atividade  $a_m$  é compatível com cada atividade presente no momento em  $A$ , é suficiente pela equação (16.4) verificar (linha 5) se seu tempo de início  $s_m$  não é anterior ao tempo de término  $f_i$  da atividade mais recentemente adicionada a  $A$ . Se a atividade  $a_m$  é compatível, as linhas 6 e 7 adicionam a atividade  $a_m$  a  $A$  e definem  $i$  como  $m$ . O conjunto  $A$  retornado pela chamada GREEDY-ACTIVITY-SELECTOR( $s, f$ ) é justamente o conjunto retornado pela chamada RECURSIVE-ACTIVITY-SELECTOR( $s, f, 0, n + 1$ ).

Como a versão recursiva, o procedimento GREEDY-ACTIVITY-SELECTOR programa um conjunto de  $n$  atividades no tempo  $\Theta(n)$ , supondo-se que as atividades já tenham sido ordenadas inicialmente por seus tempos de término.

## Exercícios

### 16.1-1

Forneça um algoritmo de programação dinâmica para o problema de seleção de atividades, baseado na recorrência (16.3). Faça seu algoritmo calcular os tamanhos  $c[i, j]$  definidos anteriormente e também produzir o subconjunto de tamanho máximo  $A$  das atividades. Suponha que as entradas tenham sido ordenadas como na equação (16.1). Compare o tempo de execução da sua solução com o tempo de execução de GREEDY-ACTIVITY-SELECTOR.

### 16.1-2

Suponha que, em vez de sempre selecionar a primeira atividade a terminar, selecionamos a última atividade a começar que seja compatível com todas as atividades selecionadas anteriormente. Descreva como essa abordagem é um algoritmo guloso e prove que ela produz uma solução ótima.

### 16.1-3

Vamos supor que temos um conjunto de atividades para programar entre um grande número de salas de conferências. Desejamos programar todas as atividades usando o mínimo possível de salas de conferências. Forneça um algoritmo guloso eficiente para determinar que atividade deve usar cada sala de conferências.

(Isso também é conhecido como o *problema de colorir grafos de intervalos*. Podemos criar um grafo de intervalos cujos vértices sejam as atividades dadas e cujas arestas conectem atividades incompatíveis. O menor número de cores necessárias para colorir cada vértice de tal modo que dois vértices adjacentes nunca recebam a mesma cor corresponde a encontrar o menor número de salas de conferências necessárias para programar todas as atividades dadas.)

### 16.1-4

Não é qualquer abordagem gulosa para o problema de seleção de atividades que produz um conjunto de tamanho máximo de atividades mutuamente compatíveis. Forneça um exemplo para mostrar que a abordagem de selecionar a atividade de menor duração entre aquelas que são compatíveis com atividades selecionadas anteriormente não funciona. Faça o mesmo no caso das abordagens de sempre selecionar a atividade que se superpõe ao menor número de outras atividades e de sempre selecionar a atividade restante compatível com o tempo de início mais antigo.

## 16.2 Elementos da estratégia gulosa

Um algoritmo guloso obtém uma solução ótima para um problema fazendo uma seqüência de escolhas. Para cada ponto de decisão no algoritmo, a opção que parece melhor no momento é escolhida. Essa estratégia de heurística nem sempre produz uma solução ótima mas, como vimos no problema de seleção de atividades, algumas vezes ela funciona. Esta seção discute algumas propriedades gerais de métodos gulosos.

O processo que seguimos na Seção 16.1 para desenvolver um algoritmo guloso foi um pouco mais complicado que o normal. Seguimos estas etapas:

1. Determinar a subestrutura ótima do problema.
2. Desenvolver uma solução recursiva.
3. Provar que, em qualquer fase da recursão, uma das escolhas ótimas é a escolha gulosa. Desse modo, sempre é seguro fazer a escolha gulosa.
4. Mostrar que todos os subproblemas induzidos por ter sido feita a escolha gulosa, exceto um, são vazios.
5. Desenvolver um algoritmo recursivo que implemente a estratégia gulosa.
6. Converter o algoritmo recursivo em um algoritmo iterativo.

Seguindo essas etapas, vimos em grandes detalhes as características de programação dinâmica de um algoritmo guloso. Contudo, na prática, normalmente simplificamos as etapas anteriores durante o projeto de um algoritmo guloso. Desenvolvemos nossa subestrutura com uma intenção de fazer uma escolha gulosa que deixe apenas um subproblema para resolver de forma ótima. Por exemplo, no problema de seleção de atividades, primeiro definimos os subproblemas  $S_{ij}$ , onde tanto  $i$  quanto  $j$  variavam. Em seguida, descobrimos que, se sempre fizéssemos a escolha gulosa, poderíamos restringir os subproblemas à forma  $S_{i, n+1}$ .

De forma alternativa, poderíamos ter adaptado nossa subestrutura ótima com uma escolha gulosa em mente. Isto é, poderíamos ter descartado o segundo subscrito e definido subproblemas da forma  $S_i = \{a_k \in S : f_i \leq s_k\}$ . Então, poderíamos ter provado que uma escolha gulosa (a primeira atividade  $a_m$  a terminar em  $S_i$ ), combinada com uma solução ótima para o conjunto restante  $S_m$  de atividades compatíveis, produz uma solução ótima para  $S_i$ . De modo mais geral, projetamos algoritmos gulosos de acordo com a seguinte seqüência de etapas:

1. Moldar o problema de otimização como um problema no qual fazemos uma escolha e ficamos com um único subproblema para resolver.
2. Provar que sempre existe uma solução ótima para o problema original que torna a escolha gulosa, de modo que a escolha gulosa é sempre segura.
3. Demonstrar que, tendo feito a escolha gulosa, o que resta é um subproblema com a propriedade de que, se combinarmos uma solução ótima para o subproblema com a escolha gulosa que fizemos, chegamos a uma solução ótima para o problema original.

Usaremos esse processo mais direto em seções posteriores deste capítulo. Apesar disso, embalio de todo algoritmo guloso, quase sempre existe uma solução de programação dinâmica mais incômoda.

Como saber se um algoritmo guloso resolverá um determinado problema de otimização? Em geral, não há como saber, mas existem dois ingredientes que são exibidos pela maioria dos problemas que se prestam a uma estratégia gulosa: a propriedade de escolha gulosa e a subestrutura ótima. Se pudermos demonstrar que o problema tem essas propriedades, estaremos no bom caminho para desenvolver um algoritmo guloso para ele.

## Propriedade de escolha gulosa

O primeiro ingrediente chave é a **propriedade de escolha gulosa**: uma solução globalmente ótima pode ser alcançada fazendo-se uma escolha localmente ótima (gulosa). Em outras palavras, quando estamos considerando que escolha fazer, efetuamos a escolha que parece melhor no problema atual, sem considerar resultados de subproblemas.

É nesse ponto que os algoritmos gulosos diferem da programação dinâmica. Na programação dinâmica, fazemos uma escolha em cada passo, mas a escolha pode depender das soluções

para subproblemas. Conseqüentemente, em geral resolvemos problemas de programação dinâmica de baixo para cima, progredindo de subproblemas menores para subproblemas maiores. Em um algoritmo guloso, fazemos qualquer escolha que pareça melhor no momento e depois resolvemos o subproblema que surge após a escolha ser feita. A escolha feita por um algoritmo guloso pode depender das escolhas até o momento, mas não pode depender de nenhuma escolha futura ou das soluções para subproblemas. Desse modo, diferente da programação dinâmica, que resolve os subproblemas de baixo para cima, uma estratégia gulosa em geral progride de cima para baixo, fazendo uma escolha gulosa após outra, reduzindo de modo iterativo cada instância do problema dado a um problema menor.

É claro que devemos provar que uma escolha gulosa em cada passo produz uma solução globalmente ótima, e é nesse ponto que pode ser necessária alguma inteligência. Normalmente, como no caso do Teorema 16.1, a prova examina uma solução globalmente ótima para algum subproblema. Em seguida, ela mostra que a solução pode ser modificada para usar a escolha gulosa, resultando em um subproblema similar, embora menor.

A propriedade de escolha gulosa freqüentemente nos oferece alguma eficiência ao fazermos nossa escolha em um subproblema. Por exemplo, no problema de seleção de atividades, supondo que já tenhamos ordenado as atividades em ordem monotonicamente crescente de tempos de término, precisamos examinar cada atividade apenas uma vez. Com freqüência, pelo pré-processamento da entrada ou usando uma estrutura de dados apropriada (muitas vezes, uma fila de prioridades), podemos fazer escolhas gulosas rapidamente, produzindo desse modo um algoritmo eficiente.

## Subestrutura ótima

Um problema exibe **subestrutura ótima** se uma solução ótima para o problema contém dentro dela soluções ótimas para subproblemas. Essa propriedade é um ingrediente chave para se avaliar a possibilidade de aplicação da programação dinâmica, como também de algoritmos gulosos. Como exemplo de subestrutura ótima, recordamos que, como demonstramos na Seção 16.1, se uma solução ótima para o subproblema  $S_{ij}$  inclui uma atividade  $a_k$ , então ela também deve conter soluções ótimas para os subproblemas  $S_{ik}$  e  $S_{kj}$ . Dada essa subestrutura ótima, demonstramos que, se soubéssemos que atividade usar como  $a_k$ , poderíamos construir uma solução ótima para  $S_{ij}$  selecionando  $a_k$  juntamente com todas as atividades em soluções ótimas para os subproblemas  $S_{ik}$  e  $S_{kj}$ . Com base nessa observação de subestrutura ótima, poderíamos criar a recorrência (16.3) que descreveu o valor de uma solução ótima.

Normalmente, usamos uma abordagem mais direta relativa à subestrutura ótima quando a aplicamos a algoritmos gulosos. Conforme mencionamos antes, nos demos o luxo de pressupor que chegamos a um subproblema tendo feito a escolha gulosa no problema original. Na realidade, tudo que precisamos fazer é demonstrar que uma solução ótima para o subproblema, combinada com a escolha gulosa já feita, produz uma solução ótima para o problema original. Esse esquema utiliza implicitamente a indução sobre os subproblemas para provar que fazer a escolha gulosa em cada etapa produz uma solução ótima.

## Estratégia gulosa *versus* programação dinâmica

Pelo fato da propriedade de subestrutura ótima ser explorada tanto por estratégias gulosas quanto por estratégias de programação dinâmica, alguém poderia ser tentado a gerar uma solução de programação dinâmica para um problema quando uma solução gulosa fosse suficiente, ou alguém poderia pensar erroneamente que uma solução gulosa funcionaria, quando de fato fosse necessária uma solução de programação dinâmica. Para ilustrar as sutilezas entre as duas técnicas, vamos investigar duas variantes de um problema clássico de otimização.

O **problema da mochila 0-1** é apresentado como a seguir. Um ladrão que rouba uma loja encontra  $n$  itens: o  $i$ -ésimo item vale  $v_i$  reais e pesa  $w_i$  quilos, onde  $v_i$  e  $w_i$  são inteiros. Ele deseja

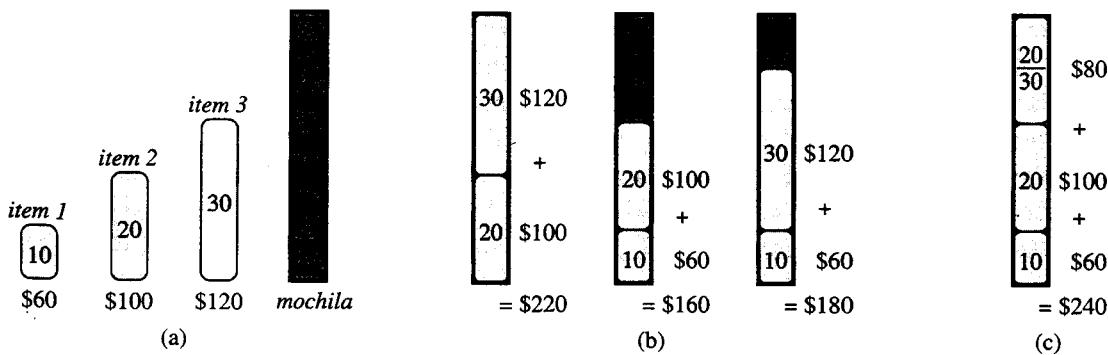
levar uma carga tão valiosa quanto possível, mas pode carregar no máximo  $W$  quilos em sua mochila, para algum inteiro  $W$ . Que itens ele deve levar? (Isso se chama problema da mochila 0-1 porque cada item deve ser levado ou deixado para trás; o ladrão não pode levar uma quantidade fracionária de um item ou levar um item mais de uma vez.)

No **problema da mochila fracionária**, a configuração é a mesma, mas o ladrão pode levar frações de itens, em vez de ter de fazer uma escolha binária (0-1) para cada item. Você pode imaginar um item no problema da mochila 0-1 como sendo algo semelhante a um lingote de ouro, enquanto um item no problema da mochila fracionária é semelhante ao ouro em pó.

Ambos os problemas de mochilas exibem a propriedade de subestrutura ótima. No caso do problema 0-1, considere a carga mais valiosa que pesa no máximo  $W$  quilos. Se removermos o item  $j$  dessa carga, a carga restante deve ser a carga mais valiosa que pese no máximo  $W - w_j$  que o ladrão pode levar dos  $n - 1$  itens originais, excluindo  $j$ . No caso do problema fracionário comparável, considere que, se removermos um peso  $w$  de um item  $j$  da carga ótima, a carga restante deve ser a carga mais valiosa que pese no máximo  $W - w$  que o ladrão pode levar dos  $n - 1$  itens originais, mais  $w_j - w$  que os do item  $j$ .

Embora os problemas sejam semelhantes, o problema da mochila fracionária pode ser resolvido por uma estratégia gulosa, enquanto o problema 0-1 não pode. Para resolver o problema fracionário, primeiro calculamos o valor por quilo  $v_i/w_i$  para cada item. Obedecendo a uma estratégia gulosa, o ladrão começa levando o máximo possível do item com o maior valor por quilo. Se o suprimento desse item se esgotar e ele ainda puder levar mais, o ladrão levará tanto quanto possível do item com o próximo maior valor por quilo e assim por diante até não poder levar mais nada. Desse modo, ordenando os itens pelo valor por quilo, o algoritmo guloso é executado no tempo  $O(n \lg n)$ . A prova de que o problema da mochila fracionária tem a propriedade de escolha gulosa fica para o Exercício 16.2-1.

Para ver que essa estratégia gulosa não funciona no caso do problema da mochila 0-1, considere a instância do problema ilustrada na Figura 16.2(a). Existem 3 itens, e a mochila pode conter 50 quilos. O item 1 pesa 10 quilos e vale 60 reais. O item 2 pesa 20 quilos e vale 100 reais. O item 3 pesa 30 quilos e vale 120 reais. Desse modo, o valor por quilo do item 1 é 6 reais por quilo, que é maior que o valor por quilo do item 2 (5 reais por quilo) ou do item 3 (4 reais por quilo). Assim, a estratégia gulosa levaria o item 1 primeiro. Porém, como podemos ver da análise de caso da Figura 16.2(b), a solução ótima leva os itens 2 e 3, deixando o item 1 para trás. As duas soluções possíveis que envolvem o item 1 não são ótimas.



**FIGURA 16.2** A estratégia gulosa não funciona para o problema da mochila 0-1. (a) O ladrão deve selecionar um subconjunto dos três itens mostrados cujo peso não deve exceder 50 quilos. (b) O subconjunto ótimo inclui os itens 2 e 3. Qualquer solução com o item 1 não é ótima, embora o item 1 tenha o maior valor por quilo. (c) Para o problema da mochila fracionária, tomar os itens em ordem de maior valor por quilo produz uma solução ótima

Contudo, para o problema fracionário comparável, a estratégia gulosa, que leva o item 1 primeiro, produz uma solução ótima, como mostra a Figura 16.2(c). Levar o item 1 não funciona no problema 0-1, porque o ladrão é incapaz de preencher plenamente sua mochila, e o espaço va-

zio diminui o valor efetivo por quilo de sua carga. No problema 0-1, quando consideramos um item para inclusão na mochila, devemos comparar a solução para o subproblema no qual o item é incluído com a solução para o subproblema no qual o item é excluído, antes de podermos fazer a escolha. O problema formulado desse modo ocasiona muitos subproblemas superpostos – um caso legítimo de programação dinâmica e, de fato, a programação dinâmica pode ser usada para resolver o problema 0-1. (Ver Exercício 16.2-2.)

## Exercícios

### 16.2-1

Prove que o problema da mochila fracionária tem a propriedade de escolha gulosa.

### 16.2-2

Forneça uma solução de programação dinâmica para o problema da mochila 0-1 que seja executado no tempo  $O(nW)$ , onde  $n$  é número de itens e  $W$  é o peso máximo dos itens que o ladrão pode pôr em sua mochila.

### 16.2-3

Suponha que, em um problema de mochila 0-1, a ordem dos itens ordenados por peso crescente seja igual à sua ordem quando eles são ordenados por valor decrescente. Forneça um algoritmo eficiente com o objetivo de encontrar uma solução ótima para essa variante do problema da mochila e mostre que seu algoritmo é correto.

### 16.2-4

O professor Midas dirige um automóvel do Rio para São Paulo pela Via Dutra. O tanque de combustível de seu carro, quando completo, contém gasolina suficiente para viajar  $n$  quilômetros, e o mapa do professor fornece as distâncias entre os postos de gasolina em sua rota. O professor deseja fazer o mínimo possível de paradas para abastecimento ao longo do caminho. Forneça um método eficiente pelo qual o professor Midas possa determinar em quais postos de gasolina deve parar, e mostre que sua estratégia produz uma solução ótima.

### 16.2-5

Descreva um algoritmo eficiente que, dado um conjunto  $\{x_1, x_2, \dots, x_n\}$  de pontos na reta real, determine o menor conjunto de intervalos fechados de comprimento unitário que contenha todos os pontos dados. Mostre que seu algoritmo é correto.

### 16.2-6 \*

Mostre como resolver o problema da mochila fracionária no tempo  $O(n)$ . Suponha que você tenha uma solução para o Problema 9-2.

### 16.2-7

Suponha que você tem dois conjuntos  $A$  e  $B$ , cada um contendo  $n$  inteiros positivos. Você tem a possibilidade de reordenar cada conjunto como preferir. Depois da reordenação, seja  $a_i$  o  $i$ -ésimo elemento do conjunto  $A$ , e seja  $b_i$  o  $i$ -ésimo elemento do conjunto  $B$ . Você recebe então uma recompensa  $\prod_{i=1}^n a_i^{b_i}$ . Forneça um algoritmo que maximize sua recompensa. Prove que seu algoritmo maximiza a recompensa e enuncie seu tempo de execução.

## 16.3 Códigos de Huffman

Os códigos de Huffman constituem uma técnica amplamente utilizada e muito eficiente para comprimir dados, guardando a relação típica de 20 a 90%, dependendo das características dos dados que estão sendo comprimidos. O algoritmo guloso de Huffman utiliza uma tabela das frequências de ocorrência dos caracteres para elaborar um modo ótimo de representar cada caractere como uma cadeia binária.

Vamos supor que temos um arquivo de dados de 100.000 caracteres que desejamos armazenar em forma compacta. Observamos que os caracteres no arquivo ocorrem com freqüências dadas pela Figura 16.3. Isto é, aparecem somente seis caracteres diferentes, e o caractere a ocorre 45.000 vezes.

Há muitas maneiras de representar um arquivo de informações como esse. Consideraremos o problema de projetar um *código de caracteres binários* (ou *código*, para abreviar) no qual cada caractere é representado por uma cadeia binária única. Se usarmos um *código de comprimento fixo*, precisaremos de 3 bits para representar seis caracteres: a = 000, b = 001, ..., f = 101. Esse método exige 300.000 bits para codificar o arquivo inteiro. Podemos fazer algo melhor?

Um *código de comprimento variável* pode funcionar consideravelmente melhor que um código de comprimento fixo, fornecendo palavras de código curtas a caracteres freqüentes e palavras de código longas a caracteres pouco freqüentes. A Figura 16.3 mostra um código desse tipo; aqui, a cadeia de 1 bit 0 representa a, e a cadeia de 4 bits 1100 representa f. Esse código exige

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1.000 = 224.000 \text{ bits}$$

para representar o arquivo, uma economia de aproximadamente 25%. De fato, esse é um código de caracteres ótimo para esse arquivo, como veremos.

## Códigos de prefixo

Consideramos aqui apenas códigos nos quais nenhuma palavra de código é também um prefixo de alguma outra palavra de código. Tais códigos são chamados *códigos de prefixo*.<sup>2</sup> É possível mostrar (embora não o façamos aqui) que a compressão de dados ótima que se pode obter por meio de um código de caracteres sempre pode ser alcançada com um código de prefixo, e assim não há perda de generalidade em restringirmos nossa atenção a códigos de prefixo.

A codificação é sempre simples para qualquer código de caracteres binários; simplesmente concatenamos as palavras de código que representam cada caractere do arquivo. Por exemplo, com o código de prefixo de comprimento variável da Figura 16.3, codificamos o arquivo de 3 caracteres abc como  $0 \cdot 101 \cdot 100 = 0101100$ , onde usamos “·” para denotar a concatenação.

	a	b	c	d	e	f
Freqüência (em milhares)	45	13	12	16	9	5
Palavra de código de comprimento fixo	000	001	010	011	100	101
Palavra de código de comprimento variável	0	101	100	111	1101	1100

FIGURA 16.3 Um problema de codificação de caracteres. Um arquivo de dados de 100.000 caracteres contém apenas os caracteres a-f, com as freqüências indicadas. Se for atribuída a cada caractere uma palavra de código de 3 bits, o arquivo poderá ser codificado em 300.000 bits. Usando-se o código de comprimento variável mostrado, o arquivo poderá ser codificado em 224.000 bits

Os códigos de prefixo são desejáveis porque simplificam a decodificação. Como nenhuma palavra de código é um prefixo de qualquer outra, a palavra de código que inicia um arquivo codificado não é ambígua. Podemos simplesmente identificar a palavra de código inicial, traduzi-la de volta para o caractere original, removê-la do arquivo codificado e repetir o processo de decodificação no restante do arquivo codificado. Em nosso exemplo, a cadeia 001011101 é analisada unicamente como 0 · 0 · 101 · 1101, que é decodificada como aabe.

O processo de decodificação precisa de uma representação conveniente para o código de prefixo, de forma que a palavra de código inicial possa ser extraída com facilidade. Uma árvore binária cujas folhas são os caracteres dados fornece tal representação. Interpretamos a palavra de código binária para um caractere como o caminho desde a raiz até esse caractere, onde 0 significa “vá para o filho da esquerda” e 1 significa “vá para o filho da direita”. A Figura 16.4 mostra as árvores para os dois códigos do nosso exemplo. Observe que elas não são árvores de pesquisa binária, pois as folhas não precisam aparecer em seqüência ordenada e os nós internos não contêm chaves de caracteres.

Um código ótimo para um arquivo é sempre representado por uma árvore binária *cheia*, na qual cada nó que não é uma folha tem dois filhos (ver Exercício 16.3-1). O código de comprimento fixo em nosso exemplo não é ótimo, pois sua árvore, mostrada na Figura 16.4(a), não é uma árvore binária cheia: existem palavras de código começando com 10..., mas nenhuma com 11.... Tendo em vista que agora podemos restringir nossa atenção a árvores binárias completas, dizemos que, se  $C$  é o alfabeto do qual os caracteres são obtidos e todas as freqüências de caracteres são positivas, então a árvore para um código de prefixo ótimo tem exatamente  $|C|$  folhas, uma para cada letra do alfabeto, e exatamente  $|C| - 1$  nós internos (veja o Exercício B.5-3).

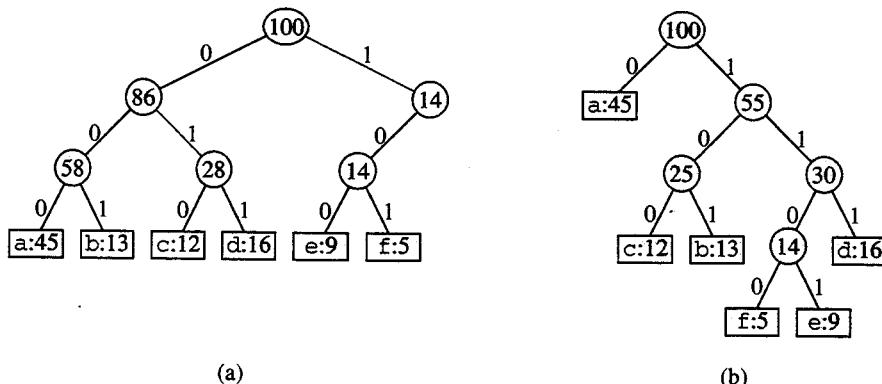


FIGURA 16.4 Árvores correspondentes aos esquemas de codificação na Figura 16.3. Cada folha é identificada com um caractere e sua freqüência de ocorrência. Cada nó interno é identificado com a soma das freqüências das folhas em sua subárvore. (a) A árvore correspondente ao código de comprimento fixo  $a = 000, \dots, f = 101$ . (b) A árvore correspondente ao código de prefixo ótimo  $a = 0, b = 101, \dots, f = 1100$

Dada uma árvore  $T$  correspondente a um código de prefixo, é uma questão simples calcular o número de bits exigidos para codificar um arquivo. Para cada caractere  $c$  no alfabeto  $C$ , seja  $f(c)$  a freqüência de  $c$  no arquivo e seja  $d_T(c)$  a profundidade de folha de  $c$  na árvore. Observe que  $d_T(c)$  é também o comprimento da palavra de código para o caractere  $c$ . Desse modo, o número de bits exigidos para codificar um arquivo é

$$B(T) = \sum_{c \in C} f(c)d_T(c), \quad (16.5)$$

que definimos como o *custo* da árvore  $T$ .

## A construção de um código de Huffman

Huffman criou um algoritmo guloso que produz um código de prefixo ótimo chamado *código de Huffman*. De acordo com nossas observações na Seção 16.2, a prova de sua correção se baseia na propriedade de escolha gulosa e subestrutura ótima. Em vez de demonstrar que essas propriedades são válidas e depois desenvolver pseudocódigo, apresentamos primeiro o pseudocódigo. Isso ajudará a esclarecer como o algoritmo faz escolhas gulosas.

No pseudocódigo a seguir, supomos que  $C$  é um conjunto de  $n$  caracteres e que cada caractere  $c \in C$  é um objeto com uma freqüência definida  $f[c]$ . O algoritmo constrói de baixo para cima a árvore  $T$  correspondente ao código ótimo. Ele começa com um conjunto de  $|C|$  folhas e executa uma seqüência de  $|C| - 1$  operações de “intercalação” para criar a árvore final. Uma fila de prioridade mínima  $Q$ , tendo  $f$  como chave, é usada para identificar os dois objetos menos freqüentes a serem intercalados. O resultado da intercalação de dois objetos é um novo objeto cuja freqüência é a soma das freqüências dos dois objetos que foram intercalados.

```
HUFFMAN( $C$ )
1  $n \leftarrow |C|$ 
2  $Q \leftarrow C$ 
3 for  $i \leftarrow 1$  to  $n - 1$ 
4   do alocar um novo nó  $z$ 
5      $esquerda[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$ 
6      $direita[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$ 
7      $f[z] \leftarrow f[x] + f[y]$ 
8      $\text{INSERT}(Q, z)$ 
9 return EXTRACT-MIN( $Q$ )       $\triangleright$  Retornar a raiz da árvore.
```

Em nosso exemplo, o algoritmo de Huffman procede como mostra a Figura 16.5. Tendo em vista que existem 6 letras no alfabeto, o tamanho da fila inicial é  $n = 6$ , e 5 passos de intercalação são exigidos para construir a árvore. A árvore final representa o código de prefixo ótimo. A palavra de código para uma letra é a seqüência de etiquetas de arestas no caminho da raiz até a letra.

A linha 2 inicializa a fila de prioridades  $Q$  com os caracteres em  $C$ . O loop **for** nas linhas 3 a 8 extrai repetidamente os dois nós  $x$  e  $y$  de freqüência mais baixa da fila e os substitui na fila por um novo nó  $z$  representando sua intercalação. A freqüência de  $z$  é calculada como a soma das freqüências de  $x$  e  $y$  na linha 7. O nó  $z$  tem  $x$  como seu filho da esquerda e  $y$  como seu filho da direita. (Essa ordem é arbitrária; a troca do filho da esquerda pelo da direita de qualquer nó produz um código diferente do mesmo custo.) Depois de  $n - 1$  intercalações, o único nó da esquerda na fila – a raiz da árvore de código – é retornado na linha 9.

A análise do tempo de execução do algoritmo de Huffman supõe que  $Q$  é implementada como um heap binário (ver Capítulo 6). Para um conjunto  $C$  de  $n$  caracteres, a inicialização de  $Q$  na linha 2 pode ser executada em tempo  $O(n)$  usando-se o procedimento BUILD-MIN-HEAP da Seção 6.3. O loop **for** nas linhas 3 a 8 é executado exatamente  $n - 1$  vezes e, como cada operação de heap exige o tempo  $O(\lg n)$ , o loop contribui com  $O(n \lg n)$  para o tempo de execução. Desse modo, o tempo de execução total de HUFFMAN em um conjunto de  $n$  caracteres é  $O(n \lg n)$ .

## Correção do algoritmo de Huffman

Para provar que o algoritmo guloso HUFFMAN é correto, mostramos que o problema de determinar um código de prefixo ótimo exibe as propriedades de escolha gulosa e de subestrutura ótima. O próximo lema mostra que a propriedade de escolha gulosa é válida.

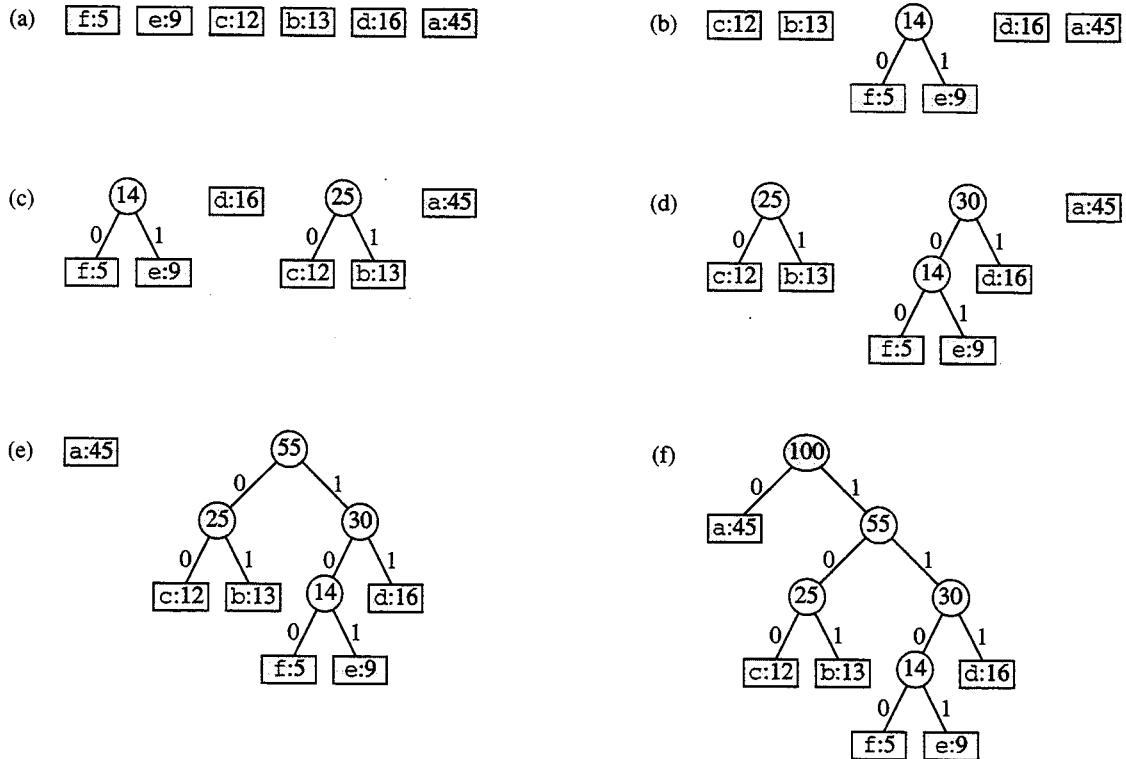


FIGURA 16.5 Os passos do algoritmo de Huffman para as freqüências dadas na Figura 16.3. Cada parte mostra o conteúdo da fila ordenado em ordem crescente por freqüência. Em cada passo, as duas árvores com freqüências mais baixas são intercaladas. As folhas são mostradas como retângulos contendo um caractere e sua freqüência. Nós internos são mostrados como círculos, contendo a soma das freqüências de seus filhos. Uma aresta conectando um nó interno a seus filhos é identificada por 0 se é uma aresta para um filho da esquerda e com 1, se é uma aresta para um filho da direita. A palavra de código para uma letra é a sequência de etiquetas nas arestas que conectam a raiz à folha correspondente a essa letra. (a) O conjunto inicial de  $n = 6$  nós, um para cada letra. (b)–(e) Estágios intermediários. (f) A árvore final

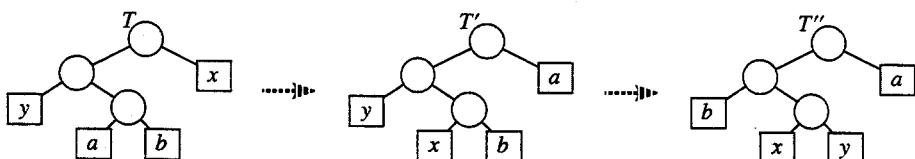


FIGURA 16.6 Uma ilustração do passo chave na prova do Lema 16.2. Na árvore ótima  $T$ , as folhas  $a$  e  $b$  são duas das folhas mais profundas e são irmãs. As folhas  $x$  e  $y$  são as duas folhas que o algoritmo de Huffman intercala primeiro; elas aparecem em posições arbitrárias em  $T$ . As folhas  $a$  e  $x$  são trocadas para se obter a árvore  $T'$ . Em seguida, as folhas  $b$  e  $y$  são trocadas para se obter a árvore  $T''$ . Como cada troca não aumenta o custo, a árvore resultante  $T''$  também é uma árvore ótima

### Lema 16.2

Seja  $C$  um alfabeto em que cada caractere  $c \in C$  tem freqüência  $f[c]$ . Sejam  $x$  e  $y$  dois caracteres em  $C$  que têm as freqüências mais baixas. Então, existe um código de prefixo ótimo para  $C$  no qual as palavras de código para  $x$  e  $y$  têm o mesmo comprimento e diferem apenas no último bit.

**Prova** A idéia da prova é tomar a árvore  $T$  representando um código de prefixo ótimo arbitrário e modificá-la para criar uma árvore representando outro código de prefixo ótimo, tal que os caracteres  $x$  e  $y$  apareçam como folhas irmãs de profundidade máxima na nova árvore. Se pudermos fazer isso, então suas palavras de código terão o mesmo comprimento e serão diferentes apenas no último bit.

Sejam  $a$  e  $b$  dois caracteres que representam folhas irmãs de profundidade máxima em  $T$ . Sem perda de generalidade, supomos  $f[a] \leq f[b]$  e  $f[x] \leq f[y]$ . Tendo em vista que  $f[x]$  e  $f[y]$  são as duas freqüências mais baixas de folhas, nessa ordem, e  $f[a]$  e  $f[B]$  são duas freqüências arbitrárias, nessa ordem, temos  $f[x] \leq f[a] \leq f[y] \leq f[b]$ . Como mostra a Figura 16.6, permutamos as posições em  $T$  de  $a$  e  $x$  para produzir uma árvore  $T'$ , e então permutamos as posições em  $T'$  de  $b$  e  $y$  para produzir uma árvore  $T''$ . Pela equação (16.5), a diferença de custo entre  $T$  e  $T'$  é

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c), \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \\ &\geq 0, \end{aligned}$$

porque tanto  $f[a] - f[x]$  quanto  $d_T(a) - d_T(x)$  são não negativas. Mais especificamente,  $f[a] - f[x]$  é não negativa porque  $x$  é uma folha de freqüência mínima, e  $d_T(a) - d_T(x)$  é não negativa porque  $a$  é uma folha de profundidade máxima em  $T$ . De modo semelhante, como a troca de  $y$  e  $b$  não aumenta o custo,  $B(T') - B(T'')$  é não negativa. Então,  $B(T') \leq B(T)$  e, como  $T$  é ótima,  $B(T) \leq B(T'')$ , que implica  $B(T'') = B(T)$ . Desse modo,  $T''$  é uma árvore ótima em que  $x$  e  $y$  aparecem como folhas irmãs de profundidade máxima, e disso decorre o lema. ■

O Lema 16.2 implica que o processo de construir uma árvore ótima por intercalações pode, sem perda de generalidade, começar com a escolha gulosa de intercalar esses dois caracteres de freqüência mais baixa. Por que essa é uma escolha gulosa? Podemos ver o custo de uma intercalação única como a soma das freqüências dos dois itens que estão sendo intercalados. O Exercício 16.3-3 mostra que o custo total da árvore construída é a soma dos custos de suas intercalações. De todas as intercalações possíveis em cada passo, HUFFMAN escolhe aquela que incorre no menor custo.

O próximo lema mostra que o problema de construir códigos de prefixo ótimos tem a propriedade de subestrutura ótima.

### Lema 16.3

Seja  $C$  um dado alfabeto com freqüência  $f[c]$  definida para cada caractere  $c \in C$ . Sejam dois caracteres  $x$  e  $y$  em  $C$  com freqüência mínima. Seja  $C'$  o alfabeto  $C$  com os caracteres  $x, y$  removidos e o (novo) caractere  $z$  adicionado, de forma que  $C' = C - \{x, y\} \cup \{z\}$ ; defina  $f$  para  $C'$  como para  $C$ , exceto pelo fato de que  $f[z] = f[x] + f[y]$ . Seja  $T'$  qualquer árvore representando um código de prefixo ótimo para o alfabeto  $C'$ . Então a árvore  $T$ , obtida a partir de  $T'$  pela substituição do nó de folha para  $z$  por um nó interno que tem  $x$  e  $y$  como filhos, representa um código de prefixo ótimo para o alfabeto  $C$ .

**Prova** Primeiro, mostramos que o custo  $B(T)$  da árvore  $T$  pode ser expresso em termos do custo  $B(T')$  da árvore  $T'$ , considerando-se os custos componentes na equação (16.5). Para cada  $c \in C - \{x, y\}$ , temos  $d_T(c) = d_{T'}(c)$ , e portanto  $f[c]d_T(c) = f[c]d_{T'}(c)$ . Como  $d_T(x) = d_T(y) = d_{T'}(z) + 1$ , temos

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= (f[z]d_{T'}(z) + f[x]) + f[y], \end{aligned}$$

$$B(T) = B(T') + f[x] + f[y]$$

ou, de modo equivalente,

$$B(T') = B(T) - f[x] - f[y].$$

Agora, provamos o lema por contradição. Suponha que  $T$  não representa um código de prefixo ótimo para  $C$ . Então, existe uma árvore  $T''$  tal que  $B(T') < B(T)$ . Sem perda de generalidade (pelo Lema 16.2),  $T''$  tem  $x$  e  $y$  como irmãos. Seja  $T'''$  a árvore  $T''$  com o pai comum de  $x$  e  $y$  substituído por uma folha  $z$  com freqüência  $f[z] = f[x] + f[y]$ . Assim

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T'), \end{aligned}$$

o que produz uma contradição para a hipótese de que  $T'$  representa um código de prefixo ótimo para  $C'$ . Desse modo,  $T$  deve representar um código de prefixo ótimo para o alfabeto  $C$ . ■

#### **Teorema 16.4**

O procedimento HUFFMAN produz um código de prefixo ótimo.

**Prova** Imediata, a partir dos Lemas 16.2 e 16.3. ■

### **Exercícios**

#### **16.3-1**

Prove que uma árvore binária que não é completa não pode corresponder a um código de prefixo ótimo.

#### **16.3-2**

O que é um código de Huffman ótimo para o conjunto de freqüências a seguir, baseado nos primeiros 8 números de Fibonacci?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Você pode generalizar sua resposta para encontrar o código ótimo quando as freqüências forem os primeiros  $n$  números de Fibonacci?

#### **16.3-3**

Prove que o custo total de uma árvore para um código também pode ser calculado como a soma sobre todos os nós internos das freqüências combinadas dos dois filhos do nó.

#### **16.3-4**

Prove que, se ordenarmos os caracteres em um alfabeto de modo que suas freqüências sejam monotonicamente decrescentes, então existe um código ótimo cujos comprimentos de palavras de código são monotonicamente crescentes.

#### **16.3-5**

Suponha que temos um código de prefixo ótimo em um conjunto de caracteres  $C = \{0, 1, \dots, n-1\}$  e desejamos transmitir esse código usando o menor número de bits possível. Mostre como representar qualquer código de prefixo ótimo em  $C$  usando apenas  $2n - 1 + n \lceil \lg n \rceil$  bits. (Sugestão: Use  $2n - 1$  bits para especificar a estrutura da árvore, como se descobre por um percurso da árvore.)

### 16.3-6

Generalize o algoritmo de Huffman para palavras de código ternárias (isto é, palavras de código que utilizam os símbolos 0, 1 e 2), e prove que ele produz códigos ternários ótimos.

### 16.3-7

Suponha que um arquivo de dados contém uma sequência de caracteres de 8 bits tais que todos os 256 caracteres são quase comuns: a freqüência máxima de caracteres é menor que duas vezes a freqüência mínima de caracteres. Prove que a codificação de Huffman nesse caso não é mais eficiente que o uso de um código de comprimento fixo de 8 bits comum.<sup>7</sup>

### 16.3-8

Mostre que nenhum esquema de compressão pode esperar comprimir um arquivo de caracteres de 8 bits escolhidos aleatoriamente por sequer um único bit. (*Sugestão:* Compare o número de arquivos com o número de arquivos codificados possíveis.)

## ★ 16.4 Fundamentos teóricos de métodos gulosos

Existe uma bela teoria sobre algoritmos gulosos, a qual descreveremos nesta seção. Essa teoria é útil para se determinar quando o método guloso produz soluções ótimas. Ela envolve estruturas combinatórias conhecidas como “matróides”. Embora essa teoria não cubra todos os casos aos quais um método guloso se aplica (por exemplo, ela não abrange o problema de seleção de atividades da Seção 16.1 ou o problema de codificação de Huffman da Seção 16.3), ela envolve muitos casos de interesse prático. Além disso, essa teoria está sendo rapidamente desenvolvida e estendida para cobrir muitas outras aplicações; consulte as notas no final deste capítulo para referência.

### Matróides

Um **matróide** é um par ordenado  $M = (S, \ell)$  que satisfaz às condições a seguir.

1.  $S$  é um conjunto finito não vazio.
2.  $\ell$  é uma família não vazia de subconjuntos de  $S$ , chamados subconjuntos **independentes** de  $S$ , tais que, se  $B \in \ell$  e  $A \subseteq B$ , então  $A \in \ell$ . Dizemos que  $\ell$  é **hereditário** se ele satisfaz a essa propriedade. Observe que o conjunto vazio  $\emptyset$  é necessariamente um membro de  $\ell$ .
3. Se  $A \in \ell$ ,  $B \in \ell$  e  $|A| < |B|$ , então existe algum elemento  $x \in B - A$  tal que  $A \cup \{x\} \in \ell$ . Dizemos que  $M$  satisfaz à **propriedade de troca**.

A palavra “matróide” se deve a Hassler Whitney. Ele estava estudando **matróides de matrícula**, nos quais os elementos de  $S$  são as linhas de uma dada matriz, e um conjunto de linhas é independente se elas são linearmente independentes no sentido usual. É fácil mostrar que essa estrutura define um matróide (ver Exercício 16.4-2).

Como outro exemplo de matróides, considere o **matróide gráfico**  $M_G = (S_G, \ell_G)$ , definido em termos de um dado grafo não orientado  $G = (V, E)$  como a seguir.

- O conjunto  $S_G$  é definido como  $E$ , o conjunto de arestas de  $G$ .
- Se  $A$  é um subconjunto de  $E$ , então  $A \in \ell_G$  se e somente se  $A$  é acíclico. Ou seja, um conjunto de arestas  $A$  é independente se e somente se o subgrafo  $G_A = (V, A)$  forma uma floresta.

O matróide gráfico  $M_G$  está intimamente relacionado com o problema da árvore de amplitude mínima, apresentado em detalhes no Capítulo 23.

### Teorema 16.5

Se  $G$  é um grafo não orientado, então  $M_G = (S_G, \ell_G)$  é um matróide.

**Prova** Claramente,  $S_G = E$  é um conjunto finito. Além disso,  $\ell_G$  é hereditário, pois um subconjunto de uma floresta é uma floresta. Em outros termos, a remoção de arestas de um conjunto acíclico de aresta não pode criar ciclos.

Desse modo, resta mostrar que  $M_G$  satisfaz à propriedade de troca. Suponha que  $G_A = (V, A)$  e  $G_B = (V, B)$  sejam florestas de  $G$  e que  $|B| > |A|$ . Isto é,  $A$  e  $B$  são conjuntos acíclicos de arestas, e  $B$  contém mais arestas que  $A$ .

Segue-se do Teorema B.2 que uma floresta que tem  $k$  arestas contém exatamente  $|V| - k$  árvores. (Para provar esse fato de outro modo, comece com  $|V|$  árvores, cada uma consistindo em um único vértice, e nenhuma aresta. Depois, cada aresta adicionada à floresta reduz o número de árvores em uma unidade.) Desse modo, a floresta  $G_A$  contém  $|V| - |A|$  árvores, e a floresta  $G_B$  contém  $|V| - |B|$  árvores.

Tendo em vista que a floresta  $G_B$  tem menos árvores que a floresta  $G_A$ , a floresta  $G_B$  deve conter alguma árvore  $T$  cujos vértices estejam em duas árvores diferentes na floresta  $G_A$ . Além disso, como  $T$  está conectada, ela deve conter uma aresta  $(u, v)$  tal que os vértices  $u$  e  $v$  estejam em árvores diferentes na floresta  $G_A$ . Como a aresta  $(u, v)$  conecta vértices em duas árvores diferentes na floresta  $G_A$ , a aresta  $(u, v)$  pode ser adicionada à floresta  $G_A$  sem criar um ciclo. Então,  $M_G$  satisfaz à propriedade de troca, completando a prova de que  $M_G$  é um matróide. ■

Dado um matróide  $M = (S, \ell)$ , chamamos um elemento  $x \notin A$  de **extensão** de  $A \in \ell$  se  $x$  pode ser adicionado a  $A$  enquanto preserva a independência; isto é,  $x$  é uma extensão de  $A$  se  $A \cup \{x\} \in \ell$ . Como exemplo, considere um matróide gráfico  $M_G$ . Se  $A$  é um conjunto independente de arestas, então a aresta  $e$  é uma extensão de  $A$  se  $e$  somente se  $e$  não está em  $A$  e a adição de  $x$  a  $A$  não cria um ciclo.

Se  $A$  é um subconjunto independente em um matróide  $M$ , dizemos que  $A$  é **máximo** se ele não tem nenhuma extensão. Isto é,  $A$  é máximo se ele não está contido em qualquer subconjunto independente maior de  $M$ . A propriedade a seguir freqüentemente é útil.

### Teorema 16.6

Todos os subconjuntos independentes máximos em um matróide têm o mesmo tamanho.

**Prova** Suponha a hipótese contrária de que  $A$  seja um subconjunto independente máximo de  $M$ , e que exista outro subconjunto independente máximo maior  $B$  de  $M$ . Então, a propriedade de troca implica que  $A$  é extensível até um conjunto independente maior  $A \cup \{x\}$  para algum  $x \in B - A$ , contradizendo a hipótese de que  $A$  é máximo. ■

Como ilustração desse teorema, considere um matróide gráfico  $M_G$  para um grafo conectado e não orientado  $G$ . Todo subconjunto independente máximo de  $M_G$  deve ser uma árvore livre com exatamente  $|V| - 1$  aresta que conecta todos os vértices de  $G$ . Tal árvore é chamada **árvore espalhada** de  $G$ .

Dizemos que um matróide  $M = (S, \ell)$  é **ponderado** se existe uma função peso associada  $w$  que atribui um peso estritamente positivo  $w(x)$  a cada elemento  $x \in S$ . A função peso  $w$  se estende a subconjuntos de  $S$  por somatório:

$$w(A) = \sum_{x \in A} w(x)$$

para qualquer  $A \subseteq S$ . Por exemplo, se  $w(e)$  denota o comprimento de uma aresta  $e$  em um matróide gráfico  $M_G$ , então  $w(A)$  é o comprimento total das arestas no conjunto de arestas  $A$ .

### Algoritmos gulosos em um matróide ponderado

Muitos problemas para os quais uma abordagem gulosa fornece soluções ótimas podem ser formulados em termos de encontrar um subconjunto independente de peso máximo em um ma-

tróide ponderado. Isto é, temos um matróide ponderado  $M = (S, \ell)$  e desejamos encontrar um conjunto independente  $A \in \ell$  tal que  $w(A)$  seja maximizado. Chamamos tal subconjunto que é independente e tem peso máximo possível de subconjunto **ótimo** do matróide. Pelo fato do peso  $w(x)$  de qualquer elemento  $x \in S$  ser positivo, um subconjunto ótimo é sempre um subconjunto independente máximo – ele sempre ajuda tornar  $A$  tão grande quanto possível.

Por exemplo, no **problema da árvore espalhada mínima**, temos um grafo conectado não orientado  $G = (V, E)$  e uma função comprimento  $w$  tal que  $w(e)$  é o comprimento (positivo) da aresta  $e$ . (Usamos aqui o termo “comprimento” para fazer referência aos pesos de arestas originais para o grafo, reservando o termo “peso” para fazer referência aos pesos no matróide associado.) Devemos encontrar um subconjunto das arestas que conecte todos os vértices e tenha comprimento total mínimo. Para visualizar esse problema como um problema de encontrar um subconjunto ótimo de um matróide, considere o matróide ponderado  $M_G$  com função peso  $w'$ , onde  $w'(e) = w_0 - w(e)$  e  $w_0$  é maior que o comprimento máximo de qualquer aresta. Nesse matróide ponderado, todos os pesos são positivos e um subconjunto ótimo é uma árvore espalhada de comprimento total mínimo no grafo original. Mais especificamente, cada subconjunto independente máximo  $A$  corresponde a uma árvore espalhada  $e$ , como

$$w'(A) = (|V| - 1)w_0 - w(A)$$

para qualquer subconjunto independente máximo  $A$ , o subconjunto independente que maximiza a quantidade  $w'(A)$  deve minimizar  $w(A)$ . Desse modo, qualquer algoritmo que possa encontrar um subconjunto ótimo  $A$  em um matróide arbitrário pode resolver o problema da árvore espalhada mínima.

O Capítulo 24 fornece algoritmos para o problema da árvore espalhada mínima, mas daremos aqui um algoritmo guloso que funciona para qualquer matróide ponderado. O algoritmo toma como entrada um matróide ponderado  $M = (S, \ell)$  com uma função de peso positivo associada  $w$  e retorna um subconjunto ótimo  $A$ . Em nosso pseudocódigo, denotamos os componentes de  $M$  por  $S[M]$  e  $\ell[M]$ , e a função peso por  $w$ . O algoritmo é guloso porque considera cada elemento  $x \in S$  por sua vez em ordem de peso monotonicamente decrescente e o adiciona de imediato ao conjunto  $A$  que está sendo acumulado, se  $A \cup \{x\}$  é independente.

#### GREEDY( $M, w$ )

```

1  $A \leftarrow 0$ 
2 ordenar  $S[M]$  em seqüência monotonicamente decrescente de peso  $w$ 
3 for cada  $x \in S[M]$ , tomado em ordem monotonicamente decrescente por peso  $w(x)$ 
4   do if  $A \cup \{x\} \in \ell[M]$ 
5     then  $A \leftarrow A \cup \{x\}$ 
6 return  $A$ 
```

Os elementos de  $S$  são considerados um por vez, em ordem de peso monotonicamente decrescente. Se o elemento  $x$  que estiver sendo considerado puder ser incluído em  $A$  enquanto se mantém a independência de  $A$ , ele será adicionado. Caso contrário,  $x$  será descartado. Como, pela definição de um matróide, o conjunto vazio é independente, e como  $x$  só será adicionado a  $A$  se  $A \cup \{x\}$  for independente, o subconjunto  $A$  sempre será independente, por indução. Então, GREEDY sempre retorna um subconjunto independente  $A$ . Veremos mais adiante que  $A$  é um subconjunto de peso máximo possível, e então  $A$  é um subconjunto ótimo.

O tempo de execução de GREEDY é fácil de analisar. Seja  $n$  um valor que representa  $|S|$ . A fase de ordenação de GREEDY demora o tempo  $O(n \lg n)$ . A linha 4 é executada exatamente  $n$  vezes, uma vez para cada elemento de  $S$ . Cada execução da linha 4 exige verificar se o conjunto  $A \cup \{x\}$  é ou não independente. Se cada verificação demorar o tempo  $O(f(n))$ , o algoritmo inteiro será executado no tempo  $O(n \lg n + nf(n))$ .

Agora, provaremos que GREEDY retorna um subconjunto ótimo.

### **Lema 16.7 (Matróides exibem a propriedade de escolha gulosa)**

Suponha que  $M = (S, \ell)$  seja um matróide ponderado com função peso  $w$  e que  $S$  esteja ordenado em ordem monotonicamente decrescente por peso. Seja  $x$  o primeiro elemento de  $S$  tal que  $\{x\}$  seja independente, se existir qualquer  $x$  desse tipo. Se  $x$  existe, então existe um subconjunto ótimo  $A$  de  $S$  que contém  $x$ .

**Prova** Se não existir tal  $x$ , então o único subconjunto independente será o conjunto vazio e terminamos. Caso contrário, seja  $B$  qualquer subconjunto ótimo não vazio. Suponha que  $x \notin B$ ; caso contrário, fazemos  $A = B$  e terminamos.

Nenhum elemento de  $B$  tem peso maior que  $w(x)$ . Para ver isso, observe que  $y \in B$  implica que  $\{y\}$  é independente, pois  $B \in \ell$ , e  $\ell$  é hereditário. Por essa razão, nossa escolha de  $x$  assegura que  $w(x) \geq w(y)$  para qualquer  $y \in B$ .

Construa o conjunto  $A$  como a seguir. Comece com  $A = \{x\}$ . Pela escolha de  $x$ ,  $A$  é independente. Usando a propriedade de troca, encontre repetidamente um novo elemento de  $B$  que possa ser adicionado a  $A$  até  $|A| = |B|$  enquanto se preserva a independência de  $A$ . Então,  $A = B - \{y\} \cup \{x\}$  para algum  $y \in B$ , e assim

$$\begin{aligned} w(A) &= w(B) - w(y) + w(x) \\ &\geq w(B). \end{aligned}$$

Como  $B$  é ótimo,  $A$  também deve ser ótimo e, como  $y \in A$ , o lema está provado. ■

Mostraremos em seguida que, se um elemento não é uma opção inicialmente, então ele não poderá ser uma opção mais tarde.

### **Lema 16.8**

Seja  $M = (S, \ell)$  qualquer matróide. Se  $x$  é um elemento de  $S$  que é uma extensão de algum subconjunto independente  $A$  de  $S$ , então  $x$  também é uma extensão de  $\emptyset$ .

**Prova** Como  $x$  é uma extensão de  $A$ , temos que  $A \cup \{x\}$  é independente. Como  $\ell$  é hereditário,  $\{x\}$  tem de ser independente. Desse modo,  $x$  é uma extensão de  $\emptyset$ .

### **Corolário 16.9**

Seja  $M = (S, \ell)$  qualquer matróide. Se  $x$  é um elemento de  $S$  tal que  $x$  não é uma extensão de  $\emptyset$ , então  $x$  não é uma extensão de qualquer subconjunto independente  $A$  de  $S$ .

**Prova** Este corolário é simplesmente o contrapositivo do Lema 16.8. ■

O Corolário 16.8 diz que qualquer elemento que não possa ser usado imediatamente nunca poderá ser usado. Então, GREEDY não pode gerar um erro ignorando quaisquer elementos iniciais em  $S$  que não sejam uma extensão de  $\emptyset$ , pois eles nunca poderão ser usados.

### **Lema 16.10 (Matróides exibem a propriedade de subestrutura ótima)**

Seja  $x$  o primeiro elemento de  $S$  escolhido por GREEDY para o matróide ponderado  $M = (S, \ell)$ . O problema restante de encontrar um subconjunto independente de peso máximo contendo  $x$  se reduz a encontrar um subconjunto independente de peso máximo do matróide ponderado  $M' = (S', \ell')$ , onde

$$\begin{aligned} S' &= \{y \in S : \{x, y\} \in \ell\}, \\ \ell' &= \{B \subseteq S - \{x\} : B \cup \{X\} \in \ell\}, \text{ e} \end{aligned}$$

a função peso para  $M'$  é a função peso para  $M$ , restrita a  $S'$ . (Chamamos  $M'$  a **contracção** de  $M$  pelo elemento  $x$ .)

**Prova** Se  $A$  é qualquer subconjunto independente de peso máximo de  $M$  contendo  $x$ , então  $A' = A - \{x\}$  é um subconjunto independente de  $M'$ . Reciprocamente, qualquer subconjunto independente  $A'$  de  $M'$  produz um subconjunto independente  $A = A' \cup \{x\}$  de  $M$ . Como temos em ambos os casos  $w(A) = w(A') + w(x)$ , uma solução de peso máximo em  $M$  contendo  $x$  produz uma solução de peso máximo em  $M'$ , e vice-versa. ■

### **Teorema 16.11 (Correção do algoritmo guloso sobre matróides)**

Se  $M = (S, \ell)$  é um matróide ponderado com função peso  $w$ , então  $\text{GREEDY}(M, w)$  retorna um subconjunto ótimo.

**Prova** Pelo Corolário 16.9, quaisquer elementos ignorados inicialmente porque não são extensões de  $\emptyset$  podem ser esquecidos, pois nunca serão úteis. Depois que o primeiro elemento  $x$  é selecionado, o Lema 16.7 implica que  $\text{GREEDY}$  não erra adicionando  $x$  a  $A$ , pois existe um subconjunto ótimo contendo  $x$ . Por fim, o Lema 16.10 implica que o problema restante é o de encontrar um subconjunto ótimo no matróide  $M'$  que seja a contração de  $M$  por  $x$ . Depois que o procedimento  $\text{GREEDY}$  define  $A$  como  $\{x\}$ , todos os seus passos restantes podem ser interpretados como se agissem no matróide  $M' = (S', \ell')$ , porque  $B$  é independente em  $M'$  se e somente se  $B \cup \{x\}$  é independente em  $M$ , para todos os conjuntos  $B \in \ell'$ . Desse modo, a operação subsequente de  $\text{GREEDY}$  encontrará um subconjunto independente de peso máximo para  $M'$ , e a operação global de  $\text{GREEDY}$  encontrará um subconjunto independente de peso máximo para  $M$ . ■

## **Exercícios**

### **16.4-1**

Mostre que  $(S, \ell_k)$  é um matróide, onde  $S$  é qualquer conjunto finito e  $\ell_k$  é o conjunto de todos os subconjuntos de  $S$  de tamanho no máximo  $k$ , onde  $k \leq |S|$ .

### **16.4-2** \*

Dada uma matriz  $T m \times n$  sobre algum campo (como os reais), mostre que  $(S, \ell)$  é um matróide, onde  $S$  é o conjunto de colunas de  $T$  e  $A \in \ell$  se e somente se as colunas em  $A$  são linearmente independentes.

### **16.4-3** \*

Mostre que, se  $(S, \ell)$  é um matróide, então  $(S, \ell')$  é um matróide, onde

$$\ell' = \{A' : S - A' \text{ contém algum } A \in \ell \text{ máximo}\}.$$

Isto é, os conjuntos independentes máximos de  $(S, \ell)$  são apenas os complementos dos conjuntos independentes máximos de  $(S, \ell)$ .

### **16.4-4** \*

Seja  $S$  um conjunto finito e seja  $S_1, S_2, \dots, S_k$  uma partição de  $S$  em subconjuntos disjuntos não vazios. Defina a estrutura  $(S, \ell)$  pela condição de que  $\ell = \{A : |A \cap S^i| \leq 1 \text{ para } i = 1, 2, \dots, k\}$ . Mostre que  $(S, \ell)$  é um matróide. Isto é, o conjunto de todos os conjuntos  $A$  que contêm no máximo um membro em cada bloco da partição determina os conjuntos independentes de um matróide.

### **16.4-5**

Mostre como transformar a função peso de um problema de matróide ponderado, onde a solução ótima desejada é um subconjunto independente máximo de peso mínimo, para torná-lo um problema padrão de matróide ponderado. Mostre cuidadosamente que sua transformação está correta.

## ★ 16.5 Um problema de programação de tarefas

Um problema interessante que pode ser resolvido com o uso de matróides é o problema de programar de modo ótimo tarefas de tempo unitário em um único processador, onde cada tarefa tem um prazo final e uma penalidade que deve ser paga, se o prazo final for perdido. O problema parece complicado, mas pode ser resolvido de uma forma surpreendentemente simples com a utilização de um algoritmo guloso.

Uma *tarefa de tempo unitário* é um trabalho, como um programa a ser executado em um computador, que exige exatamente uma unidade de tempo para se completar. Dado um conjunto finito  $S$  de tarefas de tempo unitário, uma *programação* para  $S$  é uma permutação de  $S$  especificando a ordem em que essas tarefas devem ser executadas. A primeira tarefa na programação começa no tempo 0 e termina no tempo 1; a segunda tarefa começa no tempo 1 e termina no tempo 2 e assim por diante.

O problema de *programar tarefas de tempo unitário com prazos finais e penalidades para um único processador* tem as seguintes entradas:

- Um conjunto  $S = \{a_1, a_2, \dots, a_n\}$  de  $n$  tarefas de tempo unitário.
- Um conjunto de  $n$  *prazos finais* inteiros  $d_1, d_2, \dots, d_n$ , tal que cada  $d_i$  satisfaz a  $1 \leq d_i \leq n$  e a tarefa  $i$  deve terminar no tempo  $d_i$ .
- Um conjunto de  $n$  pesos não negativos ou *penalidades*  $w_1, w_2, \dots, w_n$ , tal que uma penalidade  $w_i$  ocorre se tarefa  $i$  não é terminada no tempo  $d_i$  e nenhuma penalidade ocorre se uma tarefa termina em seu prazo final.

Devemos encontrar uma programação para  $S$  que minimize a penalidade total que ocorre para perda de prazos finais.

Considere uma dada programação. Dizemos que uma tarefa está *atrasada* nessa programação se ela termina depois seu prazo final. Caso contrário, a tarefa está *adiantada* na programação. Uma programação arbitrária sempre pode ser colocada na *forma de primeiro-adiantada*, na qual as tarefas adiantadas precedem as tarefas atrasadas. Para ver isso, observe que, se alguma tarefa adiantada  $a_i$  seguir alguma tarefa atrasada  $a_j$ , então poderemos trocar as posições de  $a_i$  e  $a_j$ , e  $a_i$  ainda estará adiantada e  $a_j$  ainda estará atrasada.

De modo semelhante, podemos afirmar que uma programação arbitrária sempre pode ser colocada em *forma canônica*, na qual as tarefas adiantadas precedem as tarefas atrasadas e as tarefas adiantadas são programadas em ordem de prazos finais monotonicamente crescentes. Para isso, colocamos a programação em forma de primeiro-adiantada. Em seguida, desde que existam duas tarefas adiantadas  $a_i$  e  $a_j$  terminando nos tempos respectivos  $k$  e  $k + 1$  na programação, tais que  $d_j < d_i$ , trocamos as posições de  $a_i$  e  $a_j$ . Como a tarefa  $a_j$  está adiantada antes da troca,  $k + 1 \leq d_j$ . Então,  $k + 1 < d_i$ , e assim a tarefa  $a_i$  ainda está adiantada após a troca. A tarefa  $a_j$  é movida para mais cedo na programação e assim ela também ainda está adiantada depois da troca.

A procura por uma programação ótima se reduz assim a encontrar um conjunto  $A$  de tarefas que estejam adiantadas na programação ótima. Depois que  $A$  é determinado, podemos criar a programação real listando os elementos de  $A$  em ordem de prazo final monotonicamente crescente, depois listando as tarefas atrasadas (isto é,  $S - A$ ) em qualquer ordem, produzindo uma ordenação canônica da programação ótima.

Dizemos que um conjunto  $A$  de tarefas é *independente* se existe uma programação para essas tarefas tal que nenhuma tarefa está atrasada. É claro que um conjunto de tarefas adiantadas para uma programação forma um conjunto independente de tarefas. Seja  $\ell$  o conjunto de todos os conjuntos independentes de tarefas.

Considere o problema de determinar se um dado conjunto  $A$  de tarefas é independente. Para  $t = 0, 1, 2, \dots, n$ , seja  $N_t(A)$  o número de tarefas em  $A$  cujo prazo final é  $t$  ou mais cedo. Observe que  $N_0(A) = 0$  para qualquer conjunto  $A$ .

### Lema 16.12

Para qualquer conjunto de tarefas  $A$ , as declarações a seguir são equivalentes.

1. O conjunto  $A$  é independente.
2. Para  $t = 0, 1, 2, \dots, n$ , temos  $N_t(A) \leq t$ .
3. Se as tarefas em  $A$  estão programadas em ordem de prazos finais monotonicamente crescentes, então nenhuma tarefa está atrasada.

**Prova** É claro que, se  $N_t(A) > t$  para algum  $t$ , então não existe nenhum modo de fazer uma programação sem nenhuma tarefa atrasada para o conjunto  $A$ , porque existem mais de  $t$  tarefas para terminar antes do tempo  $t$ . Por essa razão, (1) implica (2). Se (2) é válida, então (3) deve ser válida: não existe nenhum modo de “ficar confuso” quando se programam as tarefas em ordem de prazos finais monotonicamente crescentes, pois (2) implica que o  $i$ -ésimo maior prazo final é no máximo  $i$ . Finalmente, (3) implica (1) de forma trivial. ■

Usando a propriedade 2 do Lema 16.12, podemos calcular facilmente se um dado conjunto de tarefas é independente ou não (ver Exercício 16.5-2).

O problema de minimizar a soma das penalidades das tarefas atrasadas é igual ao problema de maximizar a soma das penalidades das tarefas adiantadas. Desse modo, o teorema a seguir assegura que podemos usar o algoritmo guloso para encontrar um conjunto independente  $A$  de tarefas com a penalidade total máxima.

### Teorema 16.13

Se  $S$  é um conjunto de tarefas de tempo unitário com prazos finais, e se  $\ell$  é o conjunto de todos os conjuntos de tarefas independentes, então o sistema correspondente  $(S, \ell)$  é um matróide.

**Prova** Todo subconjunto de um conjunto independente de tarefas certamente é independente. Para provar a propriedade de troca, suponha que  $B$  e  $A$  sejam conjuntos de tarefas independentes e que  $|B| > |A|$ . Seja  $k$  o maior  $t$  tal que  $N_t(B) \leq N_t(A)$ . (Tal valor de  $t$  existe, pois  $N_0(A) = N_0(B) = 0$ . Como  $N_n(B) = |B|$  e  $N_n(A) = |A|$ , mas  $|B| > |A|$ , devemos ter  $k < n$  e  $N_j(B) > N_j(A)$  para todo  $j$  no intervalo  $k + 1 \leq j \leq n$ . Por essa razão,  $B$  contém mais tarefas com prazo final  $k + 1$  do que  $A$ . Seja  $x$  uma tarefa em  $B - A$  com prazo final  $k + 1$ . Seja  $A' = A \cup \{x\}$ .

Agora, mostraremos que  $A'$  deve ser independente, usando a propriedade 2 do Lema 16.12. Para  $0 \leq t \leq k$ , temos  $N_t(A') = N_t(A) \leq t$ , pois  $A$  é independente. Para  $k < t \leq n$ , temos  $N_t(A') \leq N_t(B) \leq t$ , pois  $B$  é independente. Então,  $A'$  é independente, completando nossa prova de que  $(S, \ell)$  é um matróide. ■

Pelo Teorema 16.11, podemos usar um algoritmo guloso para encontrar um conjunto independente de peso máximo de tarefas  $A$ . Então, podemos criar uma programação ótima que tem as tarefas em  $A$  como suas tarefas adiantadas. Esse método é um algoritmo eficiente para programação de tarefas de tempo unitário com prazos finais e penalidades para um único processador. O tempo de execução é  $O(n^2)$  com o uso de GREEDY, pois cada uma das  $O(n)$  verificações de independência feitas por esse algoritmo demora o tempo  $O(n)$  (ver Exercício 16.5-2). Uma implementação mais rápida é dada no Problema 16-4.

	Tarefa						
$a_i$	1	2	3	4	5	6	7
$d_i$	4	2	4	3	1	4	6
$w_i$	70	60	50	40	30	20	10

FIGURA 16.7 Uma instância do problema de programar tarefas de tempo unitário com prazos finais e penalidades para um único processador

A Figura 16.7 fornece um exemplo de um problema de programação de tarefas de tempo unitário com prazos finais e penalidades para um único processador. Nesse exemplo, o algoritmo guloso seleciona as tarefas  $a_1, a_2, a_3$  e  $a_4$ , depois rejeita as tarefas  $a_5$  e  $a_6$ , e finalmente aceita a tarefa  $a_7$ . A programação final ótima é

$$\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle,$$

que incorre em uma penalidade total  $w_5 + w_6 = 50$ .

## Exercícios

### 16.5-1

Resolva a instância do problema de programação dado na Figura 16.7, mas com cada penalidade  $w_i$  substituída por  $80 - w_i$ .

### 16.5-2

Mostre como usar a propriedade 2 do Lema 16.12 para determinar o tempo  $O(|A|)$  se um dado conjunto  $A$  de tarefas é independente ou não.

## Problemas

### 16-1 Troca de moedas

Considere o problema de fazer a troca de  $n$  centavos usando o menor número de moedas. Suponha que o valor de cada moeda seja um inteiro.

- a. Descreva um algoritmo guloso para efetuar a troca consistindo em quatro valores de moedas diferentes. Prove que seu algoritmo produz uma solução ótima.
- b. Suponha que as moedas disponíveis tenham as denominações que são potências de  $c$ , isto é, as denominações são  $c^0, c^1, \dots, c^k$  para alguns inteiros  $c > 1$  e  $k \geq 1$ . Mostre que o algoritmo guloso sempre produz uma solução ótima.
- c. Forneça um conjunto de denominações de moedas para as quais o algoritmo guloso não produza uma solução ótima. Seu conjunto deve incluir um centavo de modo que exista uma solução para todo valor de  $n$ .
- d. Forneça um algoritmo de tempo  $O(nk)$  que faça a troca para qualquer conjunto de  $k$  denominações de moedas diferentes, supondo-se que uma das moedas seja um centavo.

### 16-2 Programação para minimizar o tempo médio de conclusão

Suponha que você tem um conjunto  $S = \{a_1, a_2, \dots, a_n\}$  de tarefas, onde a tarefa  $a_i$  exige  $p_i$  unidades de tempo de processamento para se completar, depois de ser iniciada. Você tem um computador no qual executar essas tarefas, e o computador só pode executar uma tarefa de cada vez. Seja  $c_i$  o **tempo de conclusão** da tarefa  $a_i$ , isto é, o tempo em que a tarefa  $a_i$  completa seu processamento. Sua meta é minimizar o tempo médio de conclusão, ou seja, minimizar  $(1/n) \sum_{i=1}^n c_i$ . Por exemplo, suponha que existam duas tarefas,  $a_1$  e  $a_2$ , com  $p_1 = 3$  e  $p_2 = 5$ , e considere a programação em que  $a_2$  é executada primeiro, seguida por  $a_1$ . Então  $c_2 = 5$ ,  $c_1 = 8$ , e o tempo médio de conclusão é  $(5 + 8)/2 = 6,5$ .

- a. Forneça um algoritmo que programe as tarefas de forma a minimizar o tempo médio de conclusão. Cada tarefa deve ser executada de modo não preemptivo, isto é, uma vez que a tarefa  $a_i$  seja iniciada, ela deve ser executada continuamente durante  $p_i$  unidades de tempo. Prove que seu algoritmo minimiza o tempo médio de conclusão e enuncie o tempo de execução do seu algoritmo.

- b.** Suponha agora que as tarefas não estejam todas disponíveis ao mesmo tempo. Ou seja, cada tarefa tem um **tempo de liberação**  $r_i$ , antes do qual ela não está disponível para ser processada. Suponha também que permitimos a **preempção**, de modo que uma tarefa pode ser suspensa e reiniciada mais tarde. Por exemplo, uma tarefa  $a_i$  com tempo de processamento  $p_i = 6$  pode iniciar sua execução no tempo 1 e ser suspensa no tempo 4. Ela pode então prosseguir no tempo 10, mas ser suspensa no tempo 11 e, por fim, prosseguir no tempo 13 e se completar no tempo 15. A tarefa  $a_i$  é executada durante um total de 6 unidades de tempo, mas seu tempo de execução é dividido em três frações. Dizemos que o tempo de conclusão de  $a_i$  é 15. Forneça um algoritmo que programe as tarefas para minimizar o tempo médio de conclusão nesse novo cenário. Prove que seu algoritmo minimiza o tempo médio de conclusão e enuncie o tempo de execução do algoritmo.

### 16.3 Subgrafos acíclicos

- a.** Seja  $G = (V, E)$  um grafo não orientado. Usando a definição de matróide, mostre que  $(E, \ell)$  é um matróide, onde  $A \in \ell$  se e somente se  $A$  é um subconjunto acíclico de  $E$ .
- b.** A **matriz de incidência** de um grafo não orientado  $G = (V, E)$  é uma matriz  $|V| \times |E| M$  tal que  $M_{ve} = 1$  se a aresta  $e$  é incidente sobre o vértice  $v$ , e  $M_{ve} = 0$  em caso contrário. Mostre que um conjunto de colunas de  $M$  é linearmente independente se e somente se o conjunto correspondente de arestas é acíclico. Em seguida, use o resultado do Exercício 16.4-2 para fornecer uma prova alternativa de que  $(E, \ell)$  da parte (a) é um matróide.
- c.** Suponha que um peso não negativo  $w(e)$  esteja associado a cada aresta em um grafo não orientado  $G = (V, E)$ . Forneça um algoritmo eficiente para encontrar um subconjunto acíclico de  $E$  de peso total máximo.
- d.** Seja  $G(V, E)$  um grafo orientado arbitrário, e seja  $(E, \ell)$  definido de tal modo que  $A \in \ell$  se e somente se  $A$  não contém nenhum ciclo orientado. Forneça um exemplo de um grafo orientado  $G$  tal que o sistema associado  $(E, \ell)$  não seja um matróide. Especifique qual condição de definição deixa de ser válida para um matróide.
- e.** A **matriz de incidência** para um grafo orientado  $G = (V, E)$  é uma matriz  $|V| \times |E| M$  tal que  $M_{ve} = -1$  se a aresta  $e$  sai do vértice  $v$ ,  $M_{ve} = 1$  se a aresta  $e$  entra no vértice  $v$  e  $M_{ve} = 0$  em qualquer outro caso. Mostre que, se um conjunto de arestas de  $G$  é linearmente independente, então o conjunto correspondente de arestas não contém um ciclo orientado.
- f.** O Exercício 16.4-2 nos informa que o conjunto de conjuntos de colunas linearmente independentes de qualquer matriz  $M$  forma um matróide. Explique cuidadosamente por que os resultados das partes (d) e (e) não são contraditórios. Como poderia deixar de haver uma perfeita correspondência entre a noção de um conjunto de arestas ser acíclico e a noção do conjunto associado de colunas da matriz de incidência ser linearmente independente?

### 16.3 Variações de programação

Considere o algoritmo a seguir para resolver o problema da Seção 16.5 de programar tarefas de tempo unitário com prazos finais e penalidades. Sejam todas as  $n$  aberturas de tempo inicialmente vazias, onde a abertura de tempo  $i$  é a abertura de tempo de comprimento unitário que termina no tempo  $i$ . Consideraremos os serviços em ordem de penalidade monotonicamente decrescente. Quando considerar a tarefa  $a_j$ , se existir uma abertura de tempo no prazo final  $d_j$  de  $a_j$  ou antes dele que ainda esteja vazia, atribua  $a_j$  à abertura mais recente, preenchendo-a. Se não houver tal abertura, atribua a tarefa  $a_j$  à mais recente das aberturas ainda não preenchidas.

- a.** Mostre que esse algoritmo sempre fornece uma resposta ótima.
- b.** Use a floresta de conjuntos disjuntos rápidos apresentada na Seção 21.3 para implementar o algoritmo de forma eficiente. Suponha que o conjunto de tarefas de entrada já esteja ordenado em ordem monotonicamente decrescente por penalidade. Analise o tempo de execução de sua implementação.

## Notas do capítulo

É possível encontrar muito mais material sobre algoritmos gulosos e matróides em Lawler [196] e em Papadimitriou e Steiglitz [237].

O algoritmo goso apareceu primeiro na literatura sobre otimização combinatória em um artigo de 1971 de Edmonds [85], embora a teoria de matróides date de um artigo de 1935 escrito por Whitney [314].

Nossa prova da correção do algoritmo goso para o problema de seleção de atividades se baseia na de Gavril [112]. O problema de programação de tarefas é estudado em Lawler [196], Horowitz e Sahni [157] e Brassard e Bratley [47].

Os códigos de Huffman foram criados em 1952 [162]; Lelewer e Hirschberg [200] pesquisaram técnicas de compressão de dados conhecidas desde 1987.

Uma extensão da teoria de matróides para a teoria de “gulóides” (*greedoids*, na edição original) teve como pioneiros Korte e Lovász [189, 190, 191, 192], que generalizam bastante a teoria apresentada aqui.

---

## *Capítulo 17*

### *Análise amortizada*

Em uma **análise amortizada**, o tempo exigido para executar uma seqüência de operações de estruturas de dados é calculado sobre a média de todas as operações executadas. A análise amortizada pode ser usada para mostrar que o custo médio de uma operação é pequeno, se for feita a média sobre uma seqüência de operações, embora uma operação única possa ser dispendiosa. A análise amortizada difere da análise do caso médio pelo fato de não haver nenhuma probabilidade envolvida; uma análise amortizada garante o *desempenho médio de cada operação no pior caso*.

As três primeiras seções deste capítulo abordam as três técnicas mais comuns usadas em análise amortizada. A Seção 17.1 começa com a análise agregada, na qual determinamos um limite superior  $T(n)$  sobre o custo total de uma seqüência de  $n$  operações. O custo amortizado por operação é então  $T(n)/n$ . Tomamos o custo médio como o custo amortizado de cada operação, de forma que todas as operações têm o mesmo custo amortizado.

A Seção 17.2 focaliza o método de contabilidade, no qual determinamos um custo amortizado de cada operação. Quando existe mais de um tipo de operação, cada tipo de operação pode ter um custo amortizado diferente. O método de contabilidade sobretaxa algumas operações no início da seqüência, armazenando a sobretaxa como um “crédito pré-pago” sobre objetos específicos na estrutura de dados. O crédito é usado mais tarde na seqüência para compensar operações que são debitadas com valor menor que seu custo real.

A Seção 17.3 discute o método potencial, semelhante ao método de contabilidade no sentido de que determinamos o custo amortizado de cada operação e podemos sobretaxar operações no início para compensar débitos reduzidos posteriores. O método potencial mantém o crédito como a “energia potencial” da estrutura de dados como um todo, em vez de associar o crédito a objetos individuais dentro da estrutura de dados.

Usaremos dois exemplos para examinar esses três modelos. Um deles é uma pilha com a operação adicional MULTIPOP, que desempilha vários objetos de uma vez. O outro é um contador binário que efetua a contagem a partir de 0 por meio da única operação INCREMENT.

Enquanto estiver lendo este capítulo, tenha em mente que os débitos atribuídos durante uma análise amortizada são válidos apenas para fins de análise. Eles não precisam e não devem aparecer no código. Por exemplo, se um crédito for atribuído a um objeto  $x$  quando se usar o método de contabilidade, não haverá necessidade de atribuir um valor apropriado a algum atributo  $crédito[x]$  no código.

As idéias sobre uma determinada estrutura de dados obtidas pela realização de uma análise amortizada podem ajudar na otimização do projeto. Por exemplo, na Seção 17.4, usaremos o método potencial para analisar uma tabela expandindo e contraindo dinamicamente a tabela.

## 17.1 A análise agregada

Na **análise agregada**, mostramos que, para todo  $n$ , uma seqüência de  $n$  operações demora o tempo de *pior caso*  $T(n)$  no total. No pior caso, o custo médio ou **custo amortizado**, por operação é portanto  $T(n)/n$ . Observe que esse custo amortizado se aplica a cada operação, mesmo quando existem diversos tipos de operações na seqüência. Os outros dois métodos que estudaremos neste capítulo, o método de contabilidade e o método potencial, podem atribuir diferentes custos amortizados a diferentes tipos de operações.

### Operações de pilhas

Em nosso primeiro exemplo de análise agregada, analisamos pilhas que foram aumentadas com uma nova operação. A Seção 10.1 apresentou as duas operações fundamentais de pilhas, cada uma das quais demora o tempo  $O(1)$ :

PUSH( $S, x$ ) empilha o objeto  $x$  sobre a pilha  $S$ .

POP( $S$ ) desempilha a parte superior da pilha  $S$  e retorna o objeto desempilhado.

Tendo em vista que cada uma dessas operações é executada no tempo  $O(1)$ , vamos considerar o custo de cada uma igual a 1. O custo total de uma seqüência de  $n$  operações PUSH e POP é então  $n$  e o tempo de execução real para  $n$  operações é consequentemente  $\Theta(n)$ .

Agora, adicionarmos a operação de pilha MULTIPOP( $S, k$ ), que remove os  $k$  objetos superiores da pilha  $S$  ou retira a pilha inteira se ela contiver menos de  $k$  objetos. No pseudocódigo a seguir, a operação STACK-EMPTY retorna TRUE se não há nenhum objeto atualmente na pilha e FALSE em caso contrário.

MULTIPOP( $S, k$ )

```
1 while not STACK-EMPTY( $S$ ) e  $k \dots 0$ 
2   do POP( $S$ )
3      $k \leftarrow k - 1$ 
```

A Figura 17.1 mostra um exemplo de MULTIPOP.

Qual é o tempo de execução de MULTIPOP( $S, k$ ) sobre uma pilha de  $s$  objetos? O tempo de execução real é linear no número de operações POP realmente executadas, e assim basta analisar MULTIPOP em termos dos custos abstratos iguais a 1 cada um para PUSH e POP. O número de iterações do loop **while** é o número  $\min(s, k)$  de objetos desempilhados da pilha. Para cada iteração do loop, é feita uma chamada a POP na linha 2. Desse modo, o custo total de MULTIPOP é  $\min(s, k)$  e o tempo de execução real é uma função linear desse custo.

topo → 23		
17		
6		
39		
10	topo → 10	
<u>47</u>	<u>47</u>	—
(a)	(b)	(c)

FIGURA 17.1 A ação de MULTIPOP sobre uma pilha  $S$ , mostrada inicialmente em (a). Os 4 objetos superiores são retirados por MULTIPOP( $S, 4$ ), cujo resultado é mostrado em (b). A próxima operação é MULTIPOP( $S, 7$ ), que esvazia a pilha – mostrado em (c) – porque existem menos de 7 objetos restantes

Vamos analisar uma seqüência de  $n$  operações PUSH, POP e MULTIPOP em uma pilha inicialmente vazia. O custo no pior caso de uma operação MULTIPOP na seqüência é  $O(n)$ , pois o tamanho da pilha é no máximo  $n$ . O tempo do pior caso de qualquer operação de pilha é então  $O(n)$  e consequentemente uma seqüência de  $n$  operações custa  $O(n^2)$ , pois podemos ter  $O(n)$  operações MULTIPOP custando  $O(n)$  cada uma. Embora essa análise esteja correta, o resultado  $O(n^2)$  obtido pela consideração do custo no pior caso de cada operação individual não é restrito.

Usando a análise agregada, podemos obter um limite superior melhor que considera a seqüência inteira de  $n$  operações. De fato, embora uma única operação MULTIPOP possa ser dispendiosa, qualquer seqüência de  $n$  operações PUSH, POP e MULTIPOP sobre uma pilha inicialmente vazia pode custar no máximo  $O(n)$ . Por quê? Cada objeto pode ser desempilhado no máximo uma vez para cada vez que é empilhado. Então, o número de vezes que POP pode ser chamada sobre uma pilha não vazia, inclusive as chamadas dentro de MULTIPOP, é no máximo igual ao número de operações PUSH, que é no máximo  $n$ . Para qualquer valor de  $n$ , qualquer seqüência de  $n$  operações PUSH, POP e MULTIPOP demora um tempo total  $O(n)$ . O custo médio de uma operação é  $O(n)/n = O(1)$ . Na análise agregada, designamos o custo amortizado de cada operação como o custo médio. Então, nesse exemplo todas as três operações de pilhas têm o custo amortizado  $O(1)$ .

Enfatizamos novamente que, embora tenhamos acabado de mostrar que o custo médio, e consequentemente o tempo de execução, de uma operação de pilha é  $O(1)$ , não estava envolvido nenhum raciocínio probabilístico. Na realidade, mostramos um limite do *pior caso*  $O(n)$  sobre uma seqüência de  $n$  operações. A divisão desse custo total por  $n$  produziu o custo médio por operação, ou o custo amortizado.

## Como incrementar um contador binário

Como outro exemplo de análise agregada, considere o problema de implementar um contador binário de  $k$  bits que efetua a contagem crescente a partir de 0. Usamos um arranjo  $A[0..k-1]$  de bits, onde  $\text{comprimento}[A] = k$ , como contador. Um número binário  $x$  armazenado no contador tem seu bit de mais baixa ordem em  $A[0]$  e seu bit de mais alta ordem em  $A[k-1]$ , de modo que  $x = \sum_{i=0}^{k-1} 2^i \cdot A[i]$ . Inicialmente,  $x = 0$ , e desse modo  $A[i] = 0$  para  $i = 0, 1, \dots, k-1$ . Para adicionar 1 (módulo  $2^k$ ) ao valor do contador, utilizamos o procedimento a seguir.

```

INCREMENT( $A$ )
1  $i \leftarrow 0$ 
2 while  $i < \text{comprimento}[A]$  e  $A[i] = 1$ 
3   do  $A[i] \leftarrow 0$ 
4    $i \leftarrow i + 1$ 
5 if  $i < \text{comprimento}[A]$ 
6   then  $A[i] \leftarrow 1$ 
```

A Figura 17.2 mostra o que acontece a um contador binário à medida que ele é incrementado 16 vezes, começando com o valor inicial 0 e terminando com o valor 16. No começo de cada iteração do loop **while** nas linhas 2 a 4, desejamos adicionar 1 na posição  $i$ . Se  $A[i] = 1$ , então a adição de 1 inverte o bit para 0 na posição  $i$  e produz um transporte igual a 1, a ser somado na posição  $i + 1$  na próxima iteração do loop. Caso contrário, o loop termina e então, se  $i < k$ , sabemos que  $A[i] = 0$ , de modo que a adição de 1 na posição  $i$ , invertendo o bit de 0 para 1, é realizada na linha 6. O custo de cada operação INCREMENT é linear no número de bits invertidos.

Como ocorre com o exemplo da pilha, uma análise superficial produz um limite que é correto mas não restrito. Uma única execução de INCREMENT demora o tempo  $\Theta(k)$  no pior caso, no qual o arranjo  $A$  contém somente valores 1. Portanto, uma seqüência de  $n$  operações INCREMENT sobre um contador inicialmente igual a zero demora o tempo  $O(nk)$  no pior caso.

Valor do contador	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Custo total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	0	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	0	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	0	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	0	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	0	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	0	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	0	0	1	1	26
16	0	0	0	1	0	0	0	1	31

FIGURA 17.2 Um contador binário de 8 bits à medida que seu valor vai de 0 a 16 por uma seqüência de 16 operações INCREMENT. Os bits invertidos para alcançar o próximo valor estão sombreados. O custo de execução para a inversão de bits é mostrado à direita. Observe que o custo total nunca é maior que duas vezes o número total de operações INCREMENT

Podemos restringir nossa análise para produzir um custo no pior caso  $O(n)$  para uma sequência de  $n$  operações INCREMENT, observando que nem todos os bits são invertidos toda vez que INCREMENT é chamada. Como mostra a Figura 17.2,  $A[0]$  é invertido toda vez que INCREMENT é chamada. O bit de mais alta ordem seguinte,  $A[1]$ , só é invertido em vezes alternadas: uma seqüência de  $n$  operações INCREMENT sobre um contador inicialmente zero faz  $A[1]$  inverter  $\lfloor n/2 \rfloor$  vezes. De modo semelhante, o bit  $A[2]$  é invertido de quatro em quatro vezes ou  $\lfloor n/4 \rfloor$  vezes em uma seqüência de  $n$  operações INCREMENT. Em geral, para  $i = 0, 1, \dots, \lfloor \lg n \rfloor$ , o bit  $A[i]$  é invertido  $\lfloor n/2^i \rfloor$  vezes em uma seqüência de  $n$  operações INCREMENT sobre um contador inicialmente igual a zero. Para  $i > \lfloor \lg n \rfloor$ , o bit  $A[i]$  nunca é invertido. O número total de inversões na seqüência é portanto

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$$= 2n,$$

pela equação (A.6). O tempo do pior caso para uma seqüência de  $n$  operações INCREMENT sobre um contador inicialmente igual a zero é então  $O(n)$ . O custo médio de cada operação, e portanto o custo amortizado por operação, é  $O(n)/n = O(1)$ .

## Exercícios

### 17.1-1

Se o conjunto de operações de pilhas incluísse uma operação MULTIPUSH, que coloca  $k$  itens na pilha, o limite  $O(1)$  sobre o custo amortizado de operações de pilhas continuaria a ser válido?

### 17.1-2

Mostre que, se uma operação DECREMENT fosse incluída no exemplo do contador de  $k$  bits,  $n$  operações poderiam custar até o tempo  $\Theta(nk)$ .

### 17.1-3

Uma seqüência de  $n$  operações é executada sobre uma estrutura de dados. A  $i$ -ésima operação custa  $i$  se  $i$  é uma potência exata de 2, e 1 em caso contrário. Utilize a análise agregada para determinar o custo amortizado por operação.

## 17.2 O método de contabilidade

No **método de contabilidade** de análise amortizada, atribuímos débitos diferenciados a operações diferentes, com algumas operações debitadas a mais ou a menos do que realmente custam. O valor que debitamos por uma operação é chamado seu **custo amortizado**. Quando o custo amortizado de uma operação excede seu custo real, a diferença é atribuída a objetos específicos na estrutura de dados sob a forma de **crédito**. O crédito pode ser usado mais tarde para ajudar a pagar operações cujo custo amortizado é menor que seu custo real. Desse modo, podemos considerar o custo amortizado de uma operação sendo dividido entre seu custo real e o crédito que é depositado ou consumido. Isso é muito diferente da análise agregada, em que todas as operações têm o mesmo custo amortizado.

Deve-se escolher com cuidado os custos amortizados de operações. Se quisermos que a análise com custos amortizados mostre que, no pior caso, o custo médio por operação é pequeno, o custo amortizado total de uma seqüência de operações deve ser um limite superior sobre o custo real total da seqüência. Além disso, como ocorre na análise agregada, esse relacionamento deve se manter válido para todas as seqüências de operações. Se denotarmos o custo real da  $i$ -ésima operação por  $c_i$  e o custo amortizado da  $i$ -ésima operação por  $\hat{c}_i$ , exigiremos

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i \quad (17.1)$$

para todas as seqüências de  $n$  operações. O crédito total armazenado na estrutura de dados é a diferença entre o custo amortizado total e o custo real total, ou  $\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i$ . Pela desigualdade (17.1), o crédito total associado com a estrutura de dados deve ser não negativo em todos os momentos. Se o crédito total pudesse se tornar negativo (o resultado da subtaxação de operações anteriores com a promessa de reembolsar a conta mais tarde), então os custos amortizados totais incidentes nesse momento estariam abaixo dos custos reais totais incidentes; para a seqüência de operações até esse instante, o custo amortizado total não seria um limite superior sobre o custo real total. Desse modo, devemos cuidar para que o crédito total na estrutura de dados nunca se torne negativo.

### Operações de pilhas

Para ilustrar o método de contabilidade de análise amortizada, vamos voltar ao exemplo da pilha. Lembre-se de que os custos reais das operações eram

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min(k, s)$ ,

onde  $k$  é o parâmetro fornecido para MULTIPOP e  $s$  é o tamanho da pilha quando ela é chamada. Vamos atribuir os custos amortizados a seguir:

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

Observe que o custo amortizado de MULTIPOP é uma constante (0), enquanto o custo real é variável. Aqui, todos os três custos amortizados são  $O(1)$ , embora em geral os custos amortizados das operações que estão sendo consideradas possam diferir assintoticamente.

Agora mostraremos que é possível pagar por qualquer seqüência de operações de pilhas debitando os custos amortizados. Suponha que seja usada uma nota de um real para representar cada unidade de custo. Começamos com uma pilha vazia. Lembre-se da analogia da Seção 10.1 entre a estrutura de dados de pilha e uma pilha de pratos em uma lanchonete. Quando colocamos um prato na pilha, usamos 1 real para pagar o custo do empilhamento e ficamos com um crédito de 1 real (além dos 2 reais cobrados), que colocamos sobre o prato. Em qualquer instante, todo prato na pilha tem 1 real de crédito sobre ele.

O real guardado no prato é o pagamento prévio pelo custo de retirá-lo da pilha. Quando executamos uma operação POP, não debitamos nada pela operação e pagamos seu custo real usando o crédito armazenado na pilha. Para desempilhar um prato, tomamos o real de crédito do prato e o utilizamos para pagar o custo real da operação. Desse modo, cobrando um pouco mais pela operação PUSH, não precisamos cobrar nada pela operação POP.

Além disso, também não precisamos cobrar nada por operações MULTIPOP. Para desempilhar o primeiro prato, tomamos o real de crédito do prato e o utilizamos para pagar o custo real de uma operação POP. Para desempilhar um segundo prato, temos novamente um real de crédito no prato para pagar pela operação POP e assim sucessivamente. Desse modo, sempre cobramos o bastante antecipadamente para pagar por operações MULTIPOP. Em outras palavras, como cada prato na pilha tem 1 real de crédito sobre ele e a pilha sempre tem um número não negativo de pratos, asseguramos que o valor do crédito é sempre não negativo. Desse modo, para *qualquer* seqüência de  $n$  operações PUSH, POP e MULTIPOP, o custo amortizado total é um limite superior sobre o custo real total. Tendo em vista que o custo amortizado total é  $O(n)$ , também o custo real total tem esse valor.

## Como incrementar um contador binário

Como outra ilustração do método de contabilidade, analisamos a operação INCREMENT sobre um contador binário que começa em zero. Conforme observamos antes, o tempo de execução dessa operação é proporcional ao número de bits invertidos, que usaremos como nosso custo para esse exemplo. Vamos utilizar uma vez mais uma nota de 1 real para representar cada unidade de custo (a inversão de um bit nesse exemplo).

No caso da análise amortizada, vamos cobrar um custo amortizado de 2 reais para definir um bit como 1. Quando um bit é definido, usamos 1 real (afora os 2 reais cobrados) para pagar pela configuração real do bit e colocamos o outro real no bit como crédito. Em qualquer instante dado, todo valor 1 no contador tem um real de crédito, e desse modo não precisamos cobrar nada para redefinir um bit como 0; apenas pagamos pela reinicialização com a nota de real no bit.

O custo amortizado de INCREMENT pode agora ser determinado. O custo de redefinir os bits dentro do loop **while** é pago pelos reais nos bits que são redefinidos. É definido no máximo um bit, na linha 6 de INCREMENT, e então o custo amortizado de uma operação INCREMENT é no máximo 2 reais. O número de valores 1 no contador nunca é negativo e, portanto, o valor do crédito é sempre não negativo. Desse modo, para  $n$  operações INCREMENT, o custo amortizado total é  $O(n)$ , o que limita o custo real total.

## Exercícios

### 17.2-1

Uma seqüência de operações de pilhas é executada sobre uma pilha cujo tamanho nunca excede  $k$ . Depois de efetuadas todas as  $k$  operações, é criada uma cópia da pilha inteira para fins de backup. Mostre que o custo de  $n$  operações de pilhas, inclusive copiar a pilha, é  $O(n)$ , atribuindo custos amortizados adequados às diversas operações de pilhas.

### 17.2-2

Faça novamente o Exercício 17.1-3, usando um método de análise de contabilidade.

### 17.2-3

Suponha que desejamos não apenas incrementar um contador, mas também reinicializá-lo com o valor zero (isto é, tornar todos os bits do número iguais a 0). Mostre como implementar um contador como um arranjo de bits de tal forma que qualquer seqüência de  $n$  operações INCREMENT e RESET demore o tempo  $O(n)$  em um contador inicialmente igual a zero. (Sugestão: Mantenha um ponteiro para o valor 1 de mais alta ordem.)

## 17.3 O método potencial

Em vez de representar o trabalho pago antecipadamente como crédito armazenado com objetos específicos na estrutura de dados, o **método potencial** de análise amortizada representa o trabalho pago antecipadamente como “energia potencial”, ou apenas “potencial”, que pode ser liberado para pagamento de operações futuras. O potencial está associado à estrutura de dados como um todo, em lugar de estar associado com objetos específicos dentro da estrutura de dados.

O método potencial funciona da maneira descrita a seguir. Começamos com uma estrutura de dados inicial  $D_0$  sobre a qual são executadas  $n$  operações. Para cada  $i = 1, 2, \dots, n$ , seja  $c_i$  o custo real da  $i$ -ésima operação e seja  $D_i$  a estrutura de dados que resulta depois da aplicação da  $i$ -ésima operação à estrutura de dados  $D_{i-1}$ . Uma **função potencial**  $\Phi$  mapeia cada estrutura de dados  $D_i$  como um número real  $\Phi(D_i)$ , que é o **potencial** associado com a estrutura de dados  $D_i$ . O **custo amortizado**  $\hat{c}_i$  da  $i$ -ésima operação com respeito à função potencial  $\Phi$  é definido por

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \quad (17.2)$$

O custo amortizado de cada operação é então seu custo real mais o aumento de potencial devido à operação. Pela equação (17.2), o custo amortizado total das  $n$  operações é

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned} \quad (17.3)$$

A segunda igualdade decorre da equação (A.9), pois os termos  $\Phi(D_i)$  se encaixam.

Se pudermos definir uma função potencial  $\Phi$  de tal forma que  $\Phi(D_n) \geq \Phi(D_0)$ , então o custo amortizado total  $\sum_{i=1}^n \hat{c}_i$  será um limite superior sobre o custo real total  $\sum_{i=1}^n c_i$ . Na prática, nem sempre sabemos quantas operações poderiam ser executadas. Assim, se exigirmos que  $\Phi(D_i) \geq \Phi(D_0)$  para todo  $i$ , então garantiremos, como no método de contabilidade, que pagaremos com antecedência. Muitas vezes é conveniente definir  $\Phi(D_0)$  como 0 e então mostrar que  $\Phi(D_i) \geq 0$  para todo  $i$ . (Veja no Exercício 17.3-1 um modo fácil para tratar os casos nos quais  $\Phi(D_0) \neq 0$ .)

Intuitivamente, se a diferença de potencial  $\Phi(D_i) - \Phi(D_{i-1})$  da  $i$ -ésima operação é positiva, então o custo amortizado  $\hat{c}_i$  representa uma sobretaxa para a  $i$ -ésima operação, e o potencial da estrutura de dados aumenta. Se a diferença de potencial é negativa, então o custo amortizado representa um desconto para a  $i$ -ésima operação, e o custo real da operação é pago pela diminuição do potencial.

Os custos amortizados definidos pelas equações (17.2) e (17.3) dependem da escolha da

tos, e ainda assim serem limites superiores sobre os custos reais. Com freqüência, existem compromissos que podem ser estabelecidos na escolha de uma função potencial; a melhor função potencial a utilizar depende dos limites de tempo desejados.

## Operações de pilhas

Para ilustrar o método potencial, voltamos uma vez mais ao exemplo das operações de pilhas PUSH, POP e MULTIPOP. Definimos a função potencial  $\Phi$  em uma pilha como o número de objetos na pilha. Para a pilha vazia  $D_0$  com que começamos, temos  $\Phi(D_0) = 0$ . Como o número de objetos na pilha nunca é negativo, a pilha  $D_i$  que resulta depois da  $i$ -ésima operação tem potencial não negativo e, desse modo,

$$\begin{aligned}\Phi(D_i) &\geq 0 \\ &= \Phi(D_0).\end{aligned}$$

O custo amortizado total de  $n$  operações com respeito a  $\Phi$  representa então um limite superior sobre o custo real.

Vamos calcular agora os custos amortizados das várias operações de pilhas. Se a  $i$ -ésima operação sobre uma pilha contendo  $s$  objetos é uma operação PUSH, então a diferença de potencial é

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s + 1) - s \\ &= 1.\end{aligned}$$

Pela equação (17.2), o custo amortizado dessa operação PUSH é

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2.\end{aligned}$$

Suponha que a  $i$ -ésima operação na pilha seja MULTIPOP( $S, k$ ) e que  $k' = \min(k, s)$  objetos sejam retirados da pilha. O custo real da operação é  $k'$ , e a diferença de potencial é

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$

Desse modo, o custo amortizado da operação MULTIPOP é

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0.\end{aligned}$$

De modo semelhante, o custo amortizado de uma operação POP comum é 0.

O custo amortizado de cada uma das três operações é  $O(1)$  e, desse modo, o custo amortizado total de uma seqüência de  $n$  operações é  $O(n)$ . Tendo em vista que já demonstramos que  $\Phi(D_i) \geq \Phi(D_0)$ , o custo amortizado total de  $n$  operações é um limite superior sobre o custo real total. O custo no pior caso de  $n$  operações é então  $O(n)$ .

## Como incrementar um contador binário

Como outro exemplo do método potencial, vamos examinar novamente como incrementar um contador binário. Dessa vez, definimos o potencial do contador depois da  $i$ -ésima operação INCREMENT como  $b_i$ , o número de valores 1 no contador depois da  $i$ -ésima operação.

Vamos calcular o custo amortizado de uma operação INCREMENT. Suponha que a  $i$ -ésima operação INCREMENT reinicialize  $t_i$  bits. O custo real da operação é então no máximo  $t_i + 1$ , pois, além de redefinir  $t_i$  bits, ela define no máximo um bit como 1. Se  $b_i = 0$ , então a  $i$ -ésima operação redefine todos os  $k$  bits, e então  $b_{i-1} = t_i = k$ . Se  $b_i > 0$ , então  $b_i = b_{i-1} - t_i + 1$ . Em qualquer caso,  $b_i \leq b_{i-1} - t_i + 1$ , e a diferença de potencial é

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\&= 1 - t_i.\end{aligned}$$

O custo amortizado é portanto

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\&\leq (t_{i+1}) + (1 - t_i) \\&= 2.\end{aligned}$$

Se o contador começa em zero, então  $\Phi(D_0) = 0$ . Tendo em vista que  $\Phi(D_i) \geq 0$  para todo  $i$ , o custo amortizado total de uma seqüência de  $n$  operações INCREMENT é um limite superior sobre o custo real total, e assim o custo no pior caso de  $n$  operações INCREMENT é  $O(n)$ .

O método potencial nos proporciona um caminho fácil para analisar o contador até mesmo quando ele não começa em zero. Existem inicialmente  $b_0$  valores 1 e, depois de  $n$  operações INCREMENT, há  $b_n$  valores 1, onde  $0 \leq b_0, b_n \leq k$ . (Lembre-se de que  $k$  é o número de bits no contador.) Podemos então reescrever a equação (17.3) como

$$\sum_{i=1}^n c_i = \sum_{i=1}^n \hat{c}_i - \Phi(D_n) + \Phi(D_0). \quad (17.4)$$

Temos  $\hat{c}_i \leq 2$  para todo  $1 \leq i \leq n$ . Tendo em vista que  $\Phi(D_0) = b_0$  e  $\Phi(D_n) = b_n$ , o custo real total de  $n$  operações INCREMENT é

$$\begin{aligned}\sum_{i=1}^n c_i &\leq \sum_{i=1}^n 2 - b_n + b_0 \\&= 2n - b_n + b_0\end{aligned}$$

Observe em particular que, sendo  $b_0 \leq k$ , desde que  $k = O(n)$ , o custo real total é  $O(n)$ . Em outras palavras, se executarmos pelo menos  $\Omega(k)$  operações INCREMENT, o custo real total será  $O(n)$ , não importando que valor inicial o contador contém.

## Exercícios

### 17.3-1

Vamos supor que temos uma função potencial  $\Phi$  tal que  $\Phi(D_i) \geq \Phi(D_0)$  para todo  $i$ , mas  $\Phi(D_i) \neq 0$ . Mostre que existe uma função potencial  $\Phi'$  tal que  $\Phi'(D_0) = 0$ ,  $\Phi'(D_i) \geq 0$  para todo  $i \geq 1$ , e os custos amortizados com o uso de  $\Phi'$  são iguais aos custos amortizados empregando-se  $\Phi$ .

### 17.3-2

Faça novamente o Exercício 17.1-3 usando um método potencial de análise.

### 17.3-3

Considere uma estrutura de dados de heap mínimo binário comum com  $n$  elementos que admite as instruções INSERT e EXTRACT-MIN no tempo do pior caso  $O(\lg n)$ . Forneça uma função potencial  $\Phi$  tal que o custo amortizado de INSERT seja  $O(\lg n)$  e o custo amortizado de EXTRACT-MIN seja  $O(1)$ , e mostre que ela funciona.

### 17.3-4

Qual é o custo total da execução de  $n$  operações de pilhas PUSH, POP e MULTIPOP, supondo-se que a pilha comece com  $s_0$  objetos e termine com  $s_n$  objetos?

### 17.3-5

Suponha que um contador comece em um número com  $b$  valores 1 em sua representação binária, em lugar de começar em 0. Mostre que o custo da execução de  $n$  operações INCREMENT é  $O(n)$  se  $n = \Omega(b)$ . (Não suponha que  $b$  seja constante.)

### 17.3-6

Mostre como implementar uma fila com duas pilhas comuns (Exercício 10.1-6) de tal forma que o custo amortizado de cada operação ENQUEUE e de cada operação DEQUEUE seja  $O(1)$ .

### 17.3-7

Projete uma estrutura de dados para suportar as duas operações a seguir para um conjunto  $S$  de inteiros:

INSERT( $S, x$ ) insere  $x$  no conjunto  $S$ .

DELETE-LARGER-HALF( $S$ ) elimina os  $\lceil S/2 \rceil$  maiores elementos de  $S$ .

Explique como implementar essa estrutura de dados de forma que qualquer seqüência de  $m$  operações seja executada no tempo  $O(m)$ .

## 17.4 Tabelas dinâmicas

Em algumas aplicações, não sabemos com antecedência quantos objetos serão armazenados em uma tabela. Podemos alocar espaço para uma tabela e somente mais tarde descobrir que ele não é suficiente. A tabela deverá então ser realocada com um tamanho maior, e todos os objetos armazenados na tabela original terão de ser copiados na tabela maior e mais nova. De modo semelhante, se muitos objetos tiverem sido eliminados da tabela, talvez compense realocar a tabela com um tamanho menor. Nesta seção, estudaremos esse problema de expandir e contrair dinamicamente uma tabela. Usando a análise amortizada, mostraremos que o custo amortizado das operações de inserção e eliminação é apenas  $O(1)$ , embora o custo real de uma operação seja grande quando ela ativa uma expansão ou uma contração. Além disso, veremos como garantir que o espaço não utilizado em uma tabela dinâmica nunca excederá uma fração constante do espaço total.

Supomos que a tabela dinâmica admite as operações TABLE-INSERT e TABLE-DELETE. TABLE-INSERT insere na tabela um item que ocupa uma única *posição*, isto é, um espaço para um só item. Do mesmo modo, podemos imaginar TABLE-DELETE como a remoção de um item da tabela, liberando assim uma posição. Os detalhes do método de estruturação de dados usado para organizar a tabela são de pouca importância; poderíamos usar uma pilha (Seção 10.1), um heap (Capítulo 6) ou uma tabela hash (Capítulo 11). Também seria possível empregar um arranjo ou uma coleção de arranjos para implementar o armazenamento de objetos, como fizemos na Seção 10.3.

Talvez seja conveniente utilizar um conceito introduzido em nossa análise de hash (Capítulo 11). Definimos o *fator de carga*  $\alpha(T)$  de uma tabela não vazia  $T$  como o número de itens armazenados na tabela dividido pelo tamanho (número de posições) da tabela. Atribuímos um tama-

nho 0 a uma tabela vazia (uma tabela sem itens) e definimos seu fator de carga como 1. Se o fator de carga de uma tabela dinâmica é limitado na parte inferior por uma constante, o espaço não utilizado na tabela nunca é maior que uma fração constante da quantidade total de espaço.

Começamos analisando uma tabela dinâmica na qual são executadas apenas inserções. Em seguida, consideraremos o caso mais geral em que são permitidas tanto inserções quanto eliminações.

### 17.4.1 Expansão de tabelas

Vamos supor que o espaço de armazenamento para uma tabela seja alocado como um arranjo de posições. Uma tabela é preenchida quando todas as posições são usadas ou, de modo equivalente, quando seu fator de carga é 1.<sup>1</sup> Em alguns ambientes de software, se é feita uma tentativa para inserir um item em uma tabela completa, não existe nenhuma alternativa além de abortar a operação com um erro. Porém, iremos supor que nosso ambiente de software, como muitos ambientes mais modernos, fornece um sistema de gerenciamento de memória que pode alocar e liberar blocos de armazenamento sob solicitação. Desse modo, quando um item for inserido em uma tabela completa, poderemos *expandir* a tabela, alocando uma nova tabela com mais posições que a tabela antiga. Como sempre precisamos que a tabela resida em memória contígua, devemos alocar um novo arranjo para a tabela maior, depois copiar itens da tabela antiga na nova tabela.

Uma heurística comum é alocar uma nova tabela que tenha duas vezes o número de posições da antiga. Se são executadas somente inserções, o fator de carga de uma tabela é sempre pelo menos  $1/2$  e, desse modo, a quantidade de espaço perdido nunca excede metade do espaço total na tabela.

No pseudocódigo a seguir, supomos que  $T$  é um objeto representando a tabela. O campo *tabela*[ $T$ ] contém um ponteiro para o bloco de armazenamento que representa a tabela. O campo *num*[ $T$ ] contém o número de itens na tabela e o campo *tamanho*[ $T$ ] é o número total de posições na tabela. Inicialmente, a tabela está vazia:  $\text{num}[T] = \text{tamanho}[T] = 0$ .

TABLE-INSERT( $T, x$ )

```

1 if tamanho[ $T$ ] = 0
2   then alocar tabela[ $T$ ] com 1 posição
3     tamanho[ $T$ ]  $\leftarrow$  1
4 if num[ $T$ ] = tamanho[ $T$ ]
5   then alocar nova-tabela com  $2 \cdot \text{tamanho}[T]$  posições
6     inserir todos os itens de tabela[ $T$ ] em nova-tabela
7     liberar tabela[ $T$ ]
8     tabela[ $T$ ]  $\leftarrow$  nova-tabela
9     tamanho[ $T$ ]  $\leftarrow 2 \cdot \text{tamanho}[T]$ 
10 inserir  $x$  em tabela[ $T$ ]
11 num[ $T$ ]  $\leftarrow \text{num}[T] + 1$ 
```

Note que temos dois procedimentos de “inserção” aqui: o procedimento TABLE-INSERT propriamente dito e a *inserção elementar* em uma tabela nas linhas 6 e 10. Podemos analisar o tempo de execução de TABLE-INSERT em termos do número de inserções elementares atribuindo um custo igual a 1 para cada inserção elementar. Supomos que o tempo de execução real de TABLE-INSERT é linear no momento de inserir itens individuais, de forma que a sobrecarga para alocação de uma tabela inicial na linha 2 é constante, e a sobrecarga para alocar e liberar espaço de armazenamento nas linhas 5 e 7 é dominado pelo custo de transferir itens na linha 6. Chamamos *expansão* o evento em que a cláusula *then* nas linhas 5 a 9 é executada.

---

<sup>1</sup>Em algumas situações, como no caso de uma tabela hash de endereço aberto, talvez seja desejável considerar uma tabela completa se seu fator de carga for igual a alguma constante estritamente menor que 1. (Ver Exercício 17.4-1.)

Vamos analisar uma sequência de  $n$  operações TABLE-INSERT em uma tabela inicialmente vazia. Qual é o custo  $c_i$  da  $i$ -ésima operação? Se existe espaço na tabela atual (ou se essa é a primeira operação), então  $c_i = 1$ , tendo em vista que precisamos executar apenas a única inserção elementar na linha 10. Entretanto, se a tabela atual está completa e ocorre uma expansão, então  $c_i = i$ : o custo é 1 para a inserção elementar na linha 10 mais  $i - 1$  para os itens que devem ser copiados da tabela antiga para a nova tabela na linha 6. Se forem executadas  $n$  operações, o custo no pior caso de uma operação será  $O(n)$ , o que leva a um limite superior  $O(n^2)$  sobre o tempo total de execução para  $n$  operações.

Esse limite não é restrito, porque o custo de expandir a tabela freqüentemente não é mantido no curso de  $n$  operações TABLE-INSERT. Especificamente, a  $i$ -ésima operação provoca uma expansão apenas quando  $i - 1$  é uma potência exata de 2. O custo amortizado de uma operação é de fato  $O(1)$ , como podemos mostrar usando a análise agregada. O custo da  $i$ -ésima operação é

$$c_i = \begin{cases} i & \text{se } i - 1 \text{ é uma potência exata de 2 ,} \\ 1 & \text{caso contrário .} \end{cases}$$

O custo total de  $n$  operações TABLE-INSERT é então

$$\sum_{i=1}^n c_i \leq \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j ,$$

$$< n + 2n$$

$$= 3n ,$$

pois existem no máximo  $n$  operações com custo 1 e os custos das operações restantes formam uma série geométrica. Tendo em vista que o custo total de  $n$  operações TABLE-INSERT é  $3n$ , o custo amortizado de uma única operação é 3.

Usando o método de contabilidade, podemos ter uma idéia do motivo pelo qual o custo amortizado de uma operação TABLE-INSERT deve ser 3. Intuitivamente, cada item paga por 3 inserções elementares: sua própria inserção na tabela atual, a mudança do próprio item quando a tabela é expandida e a mudança de outro item que já tinha sido movido uma vez quando a tabela foi expandida. Por exemplo, suponha que o tamanho da tabela seja  $m$  logo depois de uma expansão. Então, o número de itens na tabela é  $m/2$  e a tabela não contém nenhum crédito. Cobramos 3 reais por cada inserção. A inserção elementar que ocorre imediatamente custa 1 real. Outro real é colocado como crédito no item inserido. O terceiro real é colocado como crédito em um dos  $m/2$  itens que já estão na tabela. Preencher a tabela exige  $m/2 - 1$  inserções adicionais e, desse modo, quando a tabela contém  $m$  itens e está completa, cada item tem um real a pagar por sua reinserção durante a expansão.

O método potencial também pode ser usado para analisar uma seqüência de  $n$  operações TABLE-INSERT e nós o empregaremos na Seção 17.4.2 para projetar uma operação TABLE-DELETE, que também tem custo amortizado  $O(1)$ . Começamos definindo uma função potencial  $\Phi$  que é igual a 0 logo depois de uma expansão, mas aumenta até o tamanho da tabela quando a tabela está completa, de forma que a próxima expansão possa ser paga pelo potencial. A função

$$\Phi(T) = 2 \cdot \text{num}[T] - \text{tamanho}[T] \tag{17.5}$$

é uma possibilidade. Logo depois de uma expansão, temos  $\text{num}[T] = \text{tamanho}[T]/2$  e desse modo  $\Phi(T) = 0$ , como desejado. Imediatamente antes de uma expansão, temos  $\text{num}[T] - \text{tama}-$

$nbo[T]$  e, portanto,  $\Phi(T) = num[T]$ , conforme desejado. O valor inicial do potencial é 0 e, tendo em vista que a tabela está sempre pelo menos cheia até a metade,  $num[T] \geq tamanbo[T]/2$ , o que implica que  $\Phi(T)$  é sempre não negativo. Desse modo, a soma dos custos amortizados de  $n$  operações TABLE-INSERT é um limite superior sobre a soma dos custos reais.

Para analisar o custo amortizado da  $i$ -ésima operação TABLE-INSERT, sejam  $num_i$ , que representa o número de itens armazenados na tabela depois da  $i$ -ésima operação,  $tamanbo_i$ , que denota o tamanho total da tabela depois da  $i$ -ésima operação e  $\Phi_i$ , que representa o potencial depois da  $i$ -ésima operação. Inicialmente, temos  $num_0 = 0$ ,  $tamanbo_0 = 0$  e  $\Phi_0 = 0$ .

Se a  $i$ -ésima operação TABLE-INSERT não ativa uma expansão, então  $tamanbo_i = tamanbo_{i-1}$  e o custo amortizado da operação é

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - tamanbo_i) - (2 \cdot num_{i-1} - tamanbo_{i-1}) \\ &= 1 + (2 \cdot num_i - tamanbo_i) - (2(num_i - 1) - tamanbo_i) \\ &= 3.\end{aligned}$$

Se a  $i$ -ésima operação ativa uma expansão, então temos  $tamanbo_i = 2 \cdot tamanbo_{i-1}$  e  $tamanbo_{i-1} = num_{i-1} = num_i - 1$ , o que implica que  $tamanbo_i = 2 \cdot (num_i - 1)$ . Desse modo, o custo amortizado da operação é

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - tamanbo_i) - (2 \cdot num_{i-1} - tamanbo_{i-1}) \\ &= num_i + (2 \cdot num_i - 2 \cdot num_i - 1) - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) \\ &= 3.\end{aligned}$$

A Figura 17.3 mostra a plotagem dos valores de  $num_i$ ,  $tamanbo_i$  e  $\Phi_i$  em relação a  $i$ . Note como o potencial aumenta para compensar a expansão da tabela.

### 17.4.2 Expansão e contração de tabelas

Para implementar uma operação TABLE-DELETE, é bastante simples remover o item especificado da tabela. Porém, freqüentemente é desejável *contrair* a tabela quando o fator de carga da tabela se torna muito pequeno, de forma que o espaço perdido não seja exorbitante. A contração de tabela é análoga à expansão de tabela: quando o número de itens na tabela fica baixo demais, alocamos uma nova tabela menor e depois copiamos os itens da tabela antiga na tabela nova. O espaço de armazenamento para a tabela antiga pode então ser liberado e devolvido ao sistema de gerenciamento de memória. No caso ideal, gostaríamos de preservar duas propriedades:

- O fator de carga da tabela dinâmica é limitado na parte inferior por uma constante.
- O custo amortizado de uma operação de tabela é limitado na parte superior por uma constante.

Supomos que o custo pode ser medido em termos de inserções e eliminações elementares.

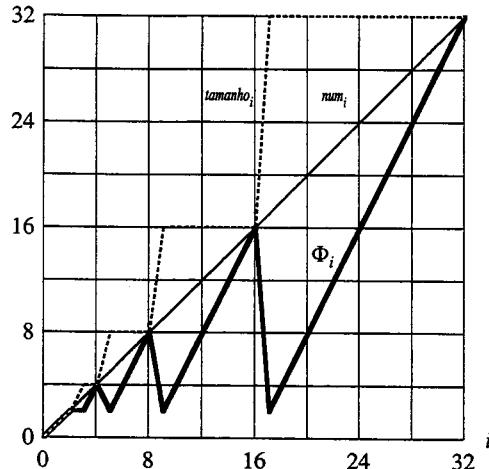


FIGURA 17.3 O efeito de uma seqüência de  $n$  operações TABLE-INSERT sobre o número  $num_i$  de itens na tabela, o número  $tamanho_i$  de posições na tabela e o potencial  $\Phi_i = 2 \cdot num_i - tamanho_i$ , cada um deles sendo medido depois da  $i$ -ésima operação. A linha fina mostra  $num_i$ , a linha tracejada mostra  $tamanho_i$ , e a linha grossa mostra  $\Phi_i$ . Note que, imediatamente antes de uma expansão, o potencial cresce até o número de itens na tabela, e assim ele pode pagar pela movimentação de todos os itens para a nova tabela. Depois disso, o potencial desce até 0, mas é imediatamente aumentado de 2 quando o item que causou a expansão é inserido

Uma estratégia natural para expansão e contração é dobrar o tamanho da tabela quando um item é inserido em uma tabela completa e reduzir à metade o tamanho quando uma eliminação puder fazer a tabela ficar preenchida com menos de metade de sua capacidade completa. Essa estratégia garante que o fator de carga da tabela nunca cairá abaixo de  $1/2$  mas, infelizmente, pode fazer o custo amortizado de uma operação ser bastante grande. Considere o cenário a seguir. Executamos  $n$  operações sobre uma tabela  $T$ , onde  $n$  é uma potência exata de 2. As  $n/2$  primeiras operações são inserções e, de acordo com nossa análise anterior, têm um custo total  $\Theta(n)$ . No fim dessa seqüência de inserções,  $num[T] = tamanho[T] = n/2$ . Para a segunda série de  $n/2$  operações, executamos a seqüência a seguir:

I, D, D, I, I, D, D, I, I, ...,

onde I significa uma inserção e D uma eliminação. A primeira inserção causa uma expansão da tabela até o tamanho  $n$ . As duas eliminações seguintes provocam uma contração da tabela de volta ao tamanho  $n/2$ . Duas inserções adicionais provocam outra expansão e assim por diante. O custo de cada expansão e contração é  $\Theta(n)$ , e existem  $\Theta(n)$  dessas operações. Desse modo, o custo total das  $n$  operações é  $\Theta(n^2)$  e o custo amortizado de uma operação é  $\Theta(n)$ .

A dificuldade com essa estratégia é óbvia: depois de uma expansão, não executamos eliminações suficientes para compensar uma contração. Do mesmo modo, depois de uma contração, não executamos inserções bastantes para compensar uma expansão.

Podemos aperfeiçoar essa estratégia, permitindo que o fator de carga da tabela caia abaixo de  $1/2$ . Especificamente, continuamos a duplicar o tamanho da tabela quando um item é inserido em uma tabela completa, mas reduzimos à metade o tamanho da tabela quando uma eliminação faz a tabela ficar menos de  $1/4$  completa, em lugar de ficar  $1/2$  completa como antes. O fator de carga da tabela tem então um limite inferior dado pela constante  $1/4$ . A idéia é que, depois de uma expansão, o fator de carga da tabela é  $1/2$ . Desse modo, metade dos itens na tabela devem ser eliminados antes de ser possível ocorrer uma contração, pois a contração não acontecerá a menos que o fator de carga caia abaixo de  $1/4$ . De maneira semelhante, depois de uma contração, o fator de carga da tabela também é  $1/2$ . Assim, o número de itens na tabela deve ser duplicado por inserções antes de poder ocorrer uma expansão, pois a expansão só aconteceria quando o fator de carga excedesse 1.

Omitimos o código correspondente a TABLE-DELETE, tendo em vista que ele é análogo a TABLE-INSERT. Contudo, é conveniente presumir para análise que, se o número de itens na tabela cair a 0, o espaço de armazenamento para a tabela será liberado. Isto é, se  $num[T] = 0$ , então  $tamanho[T] = 0$ .

Podemos agora usar o método potencial para analisar o custo de uma seqüência de  $n$  operações TABLE-INSERT e TABLE-DELETE. Começamos definindo uma função potencial  $\Phi$  que é 0 imediatamente após uma expansão ou contração e que aumenta à medida que o fator de carga aumenta até 1 ou diminui até 1/4. Vamos denotar o fator de carga de uma tabela não vazia  $T$  por  $\alpha(T) = num[T]/tamanho[T]$ . Tendo em vista que, para uma tabela vazia,  $num[T] = tamanho[T] = 0$  e  $\alpha[T] = 1$ , sempre temos  $num[T] = \alpha(T) \cdot tamanho[T]$ , quer a tabela esteja vazia ou não. Utilizaremos como nossa função potencial

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - tamanho[T] & \text{se } \alpha(T) \geq 1/2, \\ tamanho[T]/2 - num[T] & \text{se } \alpha(T) < 1/2. \end{cases} \quad (17.6)$$

Observe que o potencial de uma tabela vazia é 0 e que o potencial nunca é negativo. Portanto, o custo amortizado total de uma seqüência de operações com respeito a  $\Phi$  é um limite superior sobre seu custo real.

Antes de continuar com uma análise precisa, faremos uma pausa para examinar algumas propriedades da função potencial. Note que, quando o fator de carga é 1/2, o potencial é 0. Quando o fator de carga vale 1, temos  $tamanho[T] = num[T]$ , o que implica  $\Phi = num[T]$  e, portanto, o potencial pode compensar uma expansão se um item é inserido. Quando o fator de carga é 1/4, temos  $tamanho[T] = 4 \cdot num[T]$ , o que implica  $\Phi = num[T]$ , e assim o potencial pode compensar uma contração se um item é eliminado. A Figura 17.4 ilustra o modo como o potencial se comporta no caso de uma seqüência de operações.

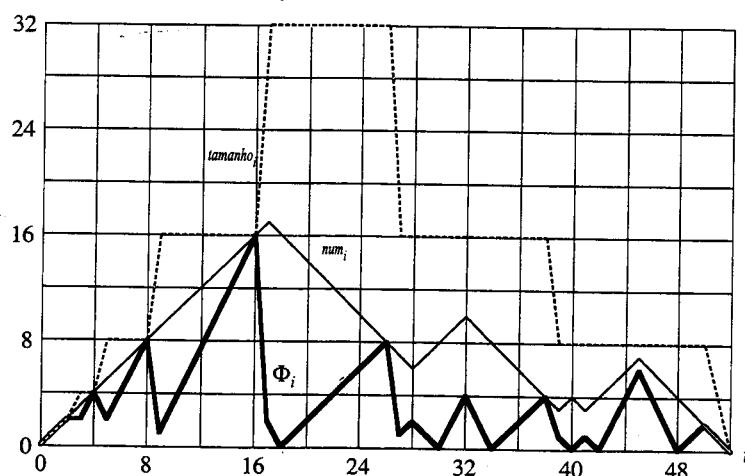


FIGURA 17.4 O efeito de uma seqüência de  $n$  operações TABLE-INSERT e TABLE-DELETE sobre o número  $num_i$  de itens na tabela, o número  $tamanho_i$  de posições na tabela e o potencial

$$\Phi_i = \begin{cases} 2 \cdot num_i - tamanho_i & \text{se } \alpha_i \geq 1/2, \\ tamanho_i / 2 - num_i & \text{se } \alpha_i < 1/2. \end{cases}$$

cada um deles medido depois da  $i$ -ésima operação. A linha fina mostra  $num_i$ , a linha tracejada mostra  $tamanho_i$ , e a linha grossa mostra  $\Phi_i$ . Note que, imediatamente antes de uma expansão, o potencial aumenta até o número de itens na tabela e, consequentemente, ele pode compensar a movimentação de todos os itens para a nova tabela. Da mesma forma, imediatamente antes de uma contração, o potencial aumenta até alcançar o número de itens na tabela

Para analisar uma seqüência de  $n$  operações TABLE-INSERT e TABLE-DELETE, seja  $c_i$  o custo real da  $i$ -ésima operação, seja  $\hat{c}_i$  seu custo amortizado com respeito a  $\Phi$ , seja  $num_i$  o número de itens armazenados na tabela depois da  $i$ -ésima operação, seja  $tamanho_i$  o tamanho total da tabela depois da  $i$ -ésima operação, seja  $\alpha_i$  o fator de carga da tabela depois da  $i$ -ésima operação e seja  $\Phi_i$  potencial depois da  $i$ -ésima operação. Inicialmente,  $num_0 = 0$ ,  $tamanho_0 = 0$ ,  $\alpha_0 = 1$  e  $\Phi_0 = 0$ .

Começamos com o caso em que a  $i$ -ésima operação é TABLE-INSERT. A análise é idêntica à da expansão de tabela da Seção 17.4.1 se  $\alpha_{i-1} \geq 1/2$ . Quer a tabela seja expandida ou não, o custo amortizado  $\alpha_{i-1} < 1/2$  da operação é no máximo 3. Se  $\alpha_{i-1} < 1/2$ , a tabela não pode se expandir como resultado da operação, pois a expansão acontece apenas quando  $\alpha_{i-1} = 1$ . Se também tivermos  $\alpha_i < 1/2$ , então o custo amortizado da  $i$ -ésima operação será

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (tamanho_i/2 - num_i) - (tamanho_{i-1}/2 - num_{i-1}) \\ &= 1 + (tamanho_i/2 - num_i) - (tamanho_i/2 - (num_i - 1)) \\ &= 0.\end{aligned}$$

Se  $\alpha_{i-1} < 1/2$ , mas  $\alpha_i \geq 1/2$ , então

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - tamanho_i) - (tamanho_{i-1}/2 - num_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - tamanho_{i-1}) - (tamanho_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_{i-1} - \frac{3}{2} tamanho_{i-1} + 3 \\ &= 3\alpha_{i-1} tamanho_{i-1} - \frac{3}{2} tamanho_{i-1} + 3 \\ &< \frac{3}{2} tamanho_{i-1} - \frac{3}{2} tamanho_{i-1} + 3 \\ &= 3.\end{aligned}$$

Desse modo, o custo amortizado de uma operação TABLE-INSERT é no máximo 3.

Examinaremos agora a situação na qual a  $i$ -ésima operação é TABLE-DELETE. Nesse caso,  $num_i = num_{i-1} - 1$ . Se  $\alpha_{i-1} < 1/2$ , então devemos considerar se a operação provoca uma contração. Se isso não ocorre, então  $tamanho_i = tamanho_{i-1}$ , e o custo amortizado da operação é

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (tamanho_i/2 - num_i) - (tamanho_{i-1}/2 - num_{i-1}) \\ &= 1 + (tamanho_i/2 - num_i) - (tamanho_i/2 - (num_i + 1)) \\ &= 2.\end{aligned}$$

Se  $\alpha_{i-1} < 1/2$  e a  $i$ -ésima operação ativa uma contração, então o custo real da operação é  $c_i = num_i + 1$ , tendo em vista que eliminamos um item e movemos  $num_i$  itens. Temos  $tamanho_i/2 = tamanho_{i-1}/4 = num_i + 1$  e o custo amortizado da operação é

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= (num_i + 1) + (tamanho_i/2 - num_i) - (tamanho_{i-1}/2 - num_{i-1}) \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\ &= 1.\end{aligned}$$

Quando a  $i$ -ésima operação é uma TABLE-DELETE e  $\alpha_{i-1} \geq 1/2$ , o custo amortizado também é limitado na parte superior por uma constante. A análise fica para o Exercício 17.4-2.

Em resumo, tendo em vista que o custo amortizado de cada operação é limitado na parte superior por uma constante, o tempo real para qualquer seqüência de  $n$  operações sobre uma tabela dinâmica é  $O(n)$ .

## Exercícios

### 17.4-1

Vamos supor que desejamos implementar uma tabela hash dinâmica de endereço aberto. Por que poderíamos considerar a tabela completa quando seu fator de carga alcançasse algum valor  $\alpha$  que fosse estritamente menor que 1? Descreva de forma resumida como fazer a inserção em uma tabela hash dinâmica de endereço aberto funcionar de tal maneira que o valor esperado do custo amortizado por inserção seja  $O(1)$ . Por que o valor esperado do custo real por inserção não é necessariamente  $O(1)$  para todas as inserções?

### 17.4-2

Mostre que, se a  $i$ -ésima operação em uma tabela dinâmica é TABLE-DELETE e  $\alpha_{i-1} \geq 1/2$ , então o custo amortizado da operação com respeito à função potencial (17.6) é limitado na parte superior por uma constante.

### 17.4-3

Suponha que, em vez de contrair uma tabela reduzindo à metade seu tamanho quando seu fator de carga cai abaixo de  $1/4$ , nós a contraímos multiplicando seu tamanho por  $2/3$  quando seu fator de carga cai abaixo de  $1/3$ . Usando a função potencial

$$\Phi(T) = |2 \cdot \text{num}[T] - \text{tamanho}[T]|,$$

mostre que o custo amortizado de uma operação TABLE-DELETE que utiliza essa estratégia é limitada na parte superior por uma constante.

## Problemas

### 17-1 Contador binário com inversão de bits

O Capítulo 30 examina um importante algoritmo chamado Transformação Rápida de Fourier (FFT – Fast Fourier Transform). O primeiro passo do algoritmo FFT executa uma **permutação com inversão de bits** sobre um arranjo de entrada  $A[0..n-1]$  cujo comprimento é  $n = 2^k$  para algum inteiro não negativo  $k$ . Essa permutação troca os elementos cujos índices têm representações binárias que são o inverso uma da outra.

Podemos expressar cada índice  $a$  como uma seqüência de  $k$  bits  $\langle a_{k-1}, a_{k-2}, \dots, a_0 \rangle$ , onde  $a = \sum_{i=0}^{k-1} a_i 2^i$ . Definimos

$$\text{rev}_k(\langle a_{k-1}, a_{k-2}, \dots, a_0 \rangle) = \langle a_{k-1}, a_{k-2}, \dots, a_0 \rangle;$$

desse modo,

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i.$$

Por exemplo, se  $n = 16$  (ou, de forma equivalente,  $k = 4$ ), então  $\text{rev}_k(3) = 12$ , pois a representação de 3 com 4 bits é 0011 que, ao ser invertido, fornece 1100, a representação com 4 bits de 12.

- a. Dada uma função  $\text{rev}_k$  que é executada no tempo  $\Theta(k)$ , escreva um algoritmo para executar a permutação com inversão de bits sobre um arranjo de comprimento  $n - 2^k$  no tempo  $O(nk)$ .

Podemos usar um algoritmo baseado em uma análise amortizada para melhorar o tempo de execução da permutação com inversão de bits. Mantemos um “contador de inversão de bits” e um procedimento BIT-REVERSED-INCREMENT que, dado um valor do contador com inversão de bits  $a$ , produza  $\text{rev}_k(\text{rev}_k(a) + 1)$ . Por exemplo, se  $k = 4$  e o contador com inversão de bits começar em 0, então chamadas sucessivas a BIT-REVERSED-INCREMENT produzirão a seqüência

$$0000, 1000, 0100, 1100, 0010, 1010, \dots = 0, 8, 4, 12, 2, 10, \dots .$$

- b. Suponha que as palavras em seu computador armazenem valores de  $k$  bits e que, no tempo unitário, o computador possa manipular os valores binários com operações tais como deslocamentos à esquerda ou à direita de valores arbitrários, AND no nível de bits, OR no nível de bits etc. Descreva uma implementação do procedimento BIT-REVERSED-INCREMENT que permita que a permutação com inversão de bits sobre um arranjo de  $n$  elementos seja executada no tempo total  $O(n)$ .
- c. Suponha que seja possível deslocar uma palavra à esquerda ou à direita por apenas um bit em tempo unitário. Ainda será possível implementar uma permutação com inversão de bits no tempo  $O(n)$ ?

### 17-2 Como tornar dinâmica a pesquisa binária

A pesquisa binária de um arranjo ordenado demora um tempo de pesquisa logarítmico, mas o tempo para inserir um novo elemento é linear no tamanho do arranjo. Podemos melhorar o tempo para inserção mantendo vários arranjos ordenados.

Especificamente, vamos supor que desejamos dar suporte a SEARCH e INSERT sobre um conjunto de  $n$  elementos. Seja  $k = \lceil \lg(n+1) \rceil$ , e seja a representação binária de  $n \langle n_{k-1}, n_{k-2}, \dots, n_0 \rangle$ . Temos  $k$  arranjos ordenados  $A_0, A_1, \dots, A_{k-1}$  onde, para  $i = 0, 1, \dots, k-1$ , o comprimento do arranjo  $A_i$  é  $2^i$ . Cada arranjo está completo ou vazio, dependendo de se ter  $n_i = 1$  ou  $n_i = 0$ , respectivamente. O número total de elementos contidos em todos os  $k$  arranjos é portanto  $\sum_{i=0}^{k-1} n_i 2^i = n$ . Embora cada arranjo individual seja ordenado, não existe nenhum relacionamento particular entre elementos de arranjos diferentes.

- a. Descreva como executar a operação SEARCH para essa estrutura de dados. Analise seu tempo de execução no pior caso.
- b. Descreva como inserir um novo elemento nessa estrutura de dados. Analise seu tempo de execução no pior caso e seu tempo de execução amortizado.
- c. Discuta como implementar DELETE.

### 17-3 Árvores de peso balanceado amortizadas

Considere uma árvore de pesquisa binária comum aumentada pela adição a cada nó  $x$  do campo  $tamanho[x]$  que fornece o número de chaves armazenadas na subárvore com raiz em  $x$ . Seja  $\alpha$  uma constante no intervalo  $1/2 \leq \alpha < 1$ . Dizemos que um dado nó  $x$  é  $\alpha$ -balanceado se

$$tamanho[\text{esquerda}[x]] \leq \alpha \cdot tamanho[x]$$

e

$$tamanho[\text{direita}[x]] \leq \alpha \cdot tamanho[x]$$

A árvore como um todo é  $\alpha$ -balanceada se todo nó na árvore é  $\alpha$ -balanceado. A abordagem amortizada a seguir para manter árvores de peso balanceado foi sugerida por G. Varghese.

- a. Uma árvore 1/2-balanceada é, em certo sentido, tão balanceada quanto possível. Dado um nó  $x$  em uma árvore de pesquisa binária arbitrária, mostre como reconstruir a subárvore com raiz em  $x$  de modo que ela se torne 1/2-balanceada. Seu algoritmo deve ser executado no tempo  $\Theta(\text{tamanho}[x])$ , e pode utilizar o espaço de armazenamento auxiliar  $O(\text{tamanho}[x])$ .
- b. Mostre que a execução de uma pesquisa em uma árvore de pesquisa binária  $\alpha$ -balanceada de  $n$  nós demora no pior caso o tempo  $O(\lg n)$ .

Para o restante deste problema, suponha que a constante  $\alpha$  seja estritamente maior que 1/2. Suponha que INSERT e DELETE sejam implementadas da maneira habitual para uma árvore de pesquisa binária de  $n$  nós, exceto pelo fato de que, após cada uma dessas operações, se qualquer nó na árvore não for mais  $\alpha$ -balanceado, então a subárvore com raiz no nó mais alto na árvore será “reconstruída” de tal modo que ela se torne 1/2-balanceada.

Analisaremos esse esquema de reconstrução usando o método potencial. Para um nó  $x$  em uma árvore de pesquisa binária  $T$ , definimos

$$\Delta(x) = |\text{tamanho}[\text{esquerda}[x]] - \text{tamanho}[\text{direita}[x]]|,$$

e definimos o potencial de  $T$  como

$$\Phi(T) = c \sum_{x \in T: \Delta(x) \geq 2} \Delta(x),$$

onde  $c$  é uma constante suficientemente grande que depende de  $\alpha$ .

- c. Demonstre que qualquer árvore de pesquisa binária tem potencial não negativo e que uma árvore 1/2-balanceada tem potencial 0.
- d. Suponha que  $m$  unidades de potencial podem compensar a reconstrução de uma subárvore de  $m$  nós. Que tamanho deve ter  $c$  em termos de  $\alpha$ , a fim de que o tempo amortizado para a reconstrução de uma subárvore que não é  $\alpha$ -balanceada seja  $O(1)$ ?
- e. Mostre que a inserção de um nó ou a eliminação de um nó de uma árvore  $\alpha$ -balanceada de  $n$  nós tem o custo de tempo amortizado  $O(\lg n)$ .

#### **17-4 O custo de reestruturação de árvores vermelho-preto**

Existem quatro operações básicas em árvores vermelho-preto que executam **modificações estruturais**: inserções de nós, eliminações de nós, rotações e modificações de cores. Vimos que RB-INSERT e RB-DELETE utilizam somente  $O(1)$  inserções de nós, rotações e eliminações de nós para manter as propriedades vermelho-preto, mas podem fazer muitas outras modificações de cores.

- a. Descreva uma árvore vermelho-preto válida com  $n$  nós tal que a chamada a RB-INSERT para adicionar o  $(n + 1)$ -ésimo nó cause  $S(\lg n)$  modificações de cores. Em seguida, descreva uma árvore vermelho-preto válida com  $n$  nós para a qual a chamada a RB-DELETE em um nó particular cause  $\Omega(\lg n)$  modificações de cores.

Embora o número de modificações de cores por operação possa ser logarítmico no pior caso, provaremos que qualquer seqüência de  $m$  operações RB-INSERT e RB-DELETE sobre uma árvore vermelho-preto inicialmente vazia causa  $O(m)$  modificações estruturais no pior caso.

- b. Alguns dos casos tratados pelo loop principal do código de RB-INSERT-FIXUP e RB-DELETE-FIXUP são **terminais**: uma vez encontrados, eles fazem o loop terminar após um número constante de operações adicionais. Para cada um dos casos de RB-INSERT-FIXUP e

**RB-DELETE-FIXUP**, especifique quais são terminais e quais não são. (*Sugestão:* Observe as Figuras 13.5, 13.6 e 13.7.)

Primeiro analisaremos as modificações estruturais quando somente inserções são executadas. Seja  $T$  uma árvore vermelho-preto e defina  $\Phi(T)$  como o número de nós vermelhos em  $T$ . Suponha que uma unidade de potencial possa compensar as modificações estruturais executadas por qualquer dos três casos de RB-INSERT-FIXUP.

- c. Seja  $T'$  o resultado da aplicação do Caso 1 de RB-INSERT-FIXUP a  $T$ . Mostre que  $\Phi(T') = \Phi(T) - 1$ .
- d. A inserção de nó em uma árvore vermelho-preto com o uso de RB-INSERT pode ser desmembrada em três partes. Liste as modificações estruturais e as mudanças potenciais resultantes das linhas 1 a 16 de RB-INSERT, de casos não terminais de RB-INSERT-FIXUP e de casos terminais de RB-INSERT-FIXUP.
- e. Usando a parte (d), demonstre que o número amortizado de modificações estruturais executadas por qualquer chamada de RB-INSERT é  $O(1)$ .

Agora desejamos provar que existem  $O(m)$  modificações estruturais quando há tanto inserções quanto eliminações. Vamos definir, para cada nó  $x$ ,

$$w(x) = \begin{cases} 0 & \text{se } x \text{ é vermelho,} \\ 1 & \text{se } x \text{ é preto e não tem nenhum filho vermelho,} \\ 0 & \text{se } x \text{ é preto e tem um filho vermelho,} \\ 2 & \text{se } x \text{ é preto e tem dois filhos vermelhos.} \end{cases}$$

Agora, redefinimos o potencial de uma árvore vermelho-preto  $T$  como

$$\Phi(T) = \sum_{x \in T} w(x),$$

e seja  $T'$  a árvore que resulta da aplicação de qualquer não terminal de RB-INSERT-FIXUP ou RB-DELETE-FIXUP a  $T$ .

- f. Mostre que  $\Phi(T') \leq \Phi(T) - 1$  para todos os casos não terminais de RB-INSERT-FIXUP. Demonstre que o número amortizado de modificações estruturais executadas por qualquer chamada de RB-INSERT-FIXUP é  $O(1)$ .
- g. Mostre que  $\Phi(T') \leq \Phi(T) - 1$  para todos os casos não terminais de RB-DELETE-FIXUP. Demonstre que o número amortizado de modificações estruturais executadas por qualquer chamada de RB-DELETE-FIXUP é  $O(1)$ .
- b. Complete a prova de que, no pior caso, qualquer seqüência de  $m$  operações RB-INSERT e RB-DELETE executa  $O(m)$  modificações estruturais.

## Notas do capítulo

A análise agregada foi utilizada por Aho, Hopcroft e Ullman [5]. Tarjan [293] examina os métodos de contabilidade e de potencial de análise amortizada e apresenta diversas aplicações. Ele atribui o método de contabilidade a vários autores, inclusive M. R. Brown, R. E. Tarjan, S. Huddleston e K. Mehlhorn. Ele atribui ainda o método potencial a D. D. Sleator. O termo “amortizado” se deve a D. D. Sleator e R. E. Tarjan.

As funções potenciais também são úteis para provar limites inferiores para certos tipos de problemas. Para cada configuração do problema, definimos uma função potencial que mapeia a configuração para um número real. Então, determinamos o potencial  $\Phi_{\text{inicial}}$  da configuração inicial, o potencial  $\Phi_{\text{final}}$  da configuração final e a mudança máxima no potencial  $\Delta\Phi_{\text{max}}$  causa-

da por qualquer etapa. O número de etapas deve portanto ser pelo menos  $|\Phi_{\text{final}} - \Phi_{\text{inicial}}| / |\Delta F_{\max}|$ . Os exemplos do uso de funções potenciais para provar limites inferiores de complexidade de E/S aparecem em trabalhos de Cormen [71], Floyd [91] e Aggarwal e Vitter [4]. Krumme, Cybenko e Venkataraman [194] aplicaram funções potenciais para provar limites inferiores sobre **broadcast**: comunicar um único item de cada vértice em um grafo a todos os outros vértices.

---

## *Parte V*

# *Estruturas de dados avançadas*

### **Introdução**

Esta parte retorna ao exame de estruturas de dados que fornecem suporte a operações sobre conjuntos dinâmicos, embora em um nível mais avançado que o da Parte III. Por exemplo, dois dos capítulos desta parte fazem uso extensivo das técnicas de análise amortizada que vimos no Capítulo 17.

O Capítulo 18 apresenta as árvores B, que são árvores de pesquisa balanceadas projetadas especificamente para serem armazenadas em discos magnéticos. Tendo em vista que os discos magnéticos operam muito mais lentamente que a memória de acesso aleatório, medimos o desempenho de árvores B não apenas pela quantidade de tempo de computação que as operações sobre conjuntos dinâmicos consomem, mas também pela quantidade de acessos a disco que são realizadas. Para cada operação de árvore B, o número de acessos a disco aumenta com a altura da árvore B, que é mantida baixa pelas operações sobre a árvore B.

Os Capítulos 19 e 20 apresentam implementações de heaps intercaláveis, os quais admitem as operações INSERT, MINIMUM, EXTRACT-MIN e UNION.<sup>1</sup> A operação UNION une, ou intercala, dois heaps. As estruturas de dados nesses capítulos também admitem as operações DELETE e DECREASE-KEY.

Os heaps binomiais, que aparecem no Capítulo 19, admitem cada uma dessas operações no tempo de pior caso  $O(\lg n)$ , onde  $n$  é o número total de elementos no heap de entrada (ou nos dois heaps de entrada juntos, no caso de UNION). Quando a operação UNION deve ser admitida, os heaps binomiais são superiores aos heaps binários introduzidos no Capítulo 6, porque ela demora o tempo  $1(n)$  para unir dois heaps binários no pior caso.

---

<sup>1</sup> Como no Problema 10.2, definimos um heap intercalável para suportar MINIMUM e EXTRACT-MIN, e assim também podemos nos referir a ele como um *heap mínimo intercalável*. Como alternativa, se ele desse suporte a MAXIMUM e EXTRACT-MAX, seria um *heap máximo intercalável*. A menos que especifiquemos em contrário, os heaps intercaláveis serão por padrão heaps mínimos intercaláveis.

Os heaps de Fibonacci, vistos no Capítulo 20, constituem um avanço em relação aos heaps binomiais, pelo menos em um sentido teórico. Usamos limites de tempo amortizados para medir o desempenho de heaps de Fibonacci. As operações INSERT, MINIMUM e UNION demoram apenas o tempo  $O(1)$  real e amortizado sobre heaps de Fibonacci, e as operações EXTRACT-MIN e DELETE demoram o tempo amortizado  $O(\lg n)$ . Porém, a vantagem mais significativa dos heaps de Fibonacci é que DECREASE-KEY demora apenas o tempo amortizado  $O(1)$ . O baixo tempo amortizado da operação DECREASE-KEY é o motivo pelo qual os heaps de Fibonacci estão no núcleo de alguns dos algoritmos assintoticamente mais rápidos vistos até hoje para problemas de grafos.

Finalmente, o Capítulo 21 apresenta estruturas de dados para conjuntos disjuntos. Temos um universo de  $n$  elementos que são agrupados em conjuntos dinâmicos. Inicialmente, cada elemento pertence a seu próprio conjunto unitário. A operação UNION une dois conjuntos, e a consulta FIND-SET identifica o conjunto em que um dado elemento se encontra no momento. Representando cada conjunto por uma árvore enraizada simples, obtemos operações surpreendentemente rápidas: uma sequência de  $m$  operações é executada no tempo  $O(m \alpha(n))$ , onde  $\alpha(n)$  é uma função que cresce de forma incrivelmente lenta –  $\alpha(n)$  é no máximo 4 em qualquer aplicação concebível. A análise amortizada que prova esse limite de tempo é tão complexa quanto a estrutura de dados é simples.

Os tópicos abordados nesta parte não são de modo algum os únicos exemplos de estruturas de dados “avançadas”. Outras estruturas de dados avançadas incluem as seguintes:

- As **árvores dinâmicas**, introduzidas por Sleator e Tarjan [281] e discutidas por Tarjan [292], mantêm uma floresta de árvores enraizadas disjuntas. Cada aresta em cada árvore tem um custo de valor real. As árvores dinâmicas admitem consultas para encontrar pais, raízes, custos de arestas e o custo mínimo de aresta sobre um caminho desde um nó até uma raiz. As árvores podem ser manipuladas cortando-se arestas, atualizando-se todos os custos de arestas sobre um caminho desde um nó até uma raiz, vinculando uma raiz em outra árvore e tornando um nó a raiz da árvore em que ele aparece. Uma implementação de árvores dinâmicas fornece um limite de tempo amortizado  $O(\lg n)$  para cada operação; uma implementação mais complicada produz limites de tempo  $O(\lg n)$  no pior caso. As árvores dinâmicas são usadas em alguns dos algoritmos de fluxo de rede assintoticamente mais rápidos.
- As **árvores espalhadas**, desenvolvidas por Sleator e Tarjan [282] e discutidas por Tarjan [292], são uma forma de árvore de pesquisa binária, na qual as operações padrão de árvores de pesquisa são executadas em tempo amortizado  $O(\lg n)$ . Uma aplicação de árvores espalhadas simplifica as árvores dinâmicas.
- Estruturas de dados **persistentes** permitem consultas, e às vezes também atualizações, sobre versões anteriores de uma estrutura de dados. Driscoll, Sarnak, Sleator e Tarjan [82] apresentam técnicas para tornar estruturas de dados encadeadas persistentes com apenas um pequeno custo de tempo e espaço. O Problema 13-1 fornece um exemplo simples de um conjunto dinâmico persistente.
- Várias estruturas de dados permitem uma implementação mais rápida de operações de dicionário (INSERT, DELETE e SEARCH) para um universo restrito de chaves. Tirando proveito dessas restrições, elas são capazes de alcançar melhores tempos de execução assintóticos no pior caso que estruturas de dados baseadas em comparação. Uma estrutura de dados criada por van Emde Boas [301] admite as operações MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT-MIN, EXTRACT-MAX, PREDECESSOR e SUCCESSOR no tempo de pior caso  $O(\lg \lg n)$ , sujeitas à restrição de que o universo de chaves é o conjunto  $\{1, 2, \dots, n\}$ . Fredman e Willard introduziram as **árvores de fusão** [99], a primeira estrutura de dados a permitir operações de dicionário mais rápidas quando o universo se limita a inteiros. Eles mostraram como implementar essas operações no tempo  $O(\lg n / \lg \lg n)$ . Várias

estruturas de dados subsequentes, inclusive *árvores de pesquisa exponenciais* [16] também forneceram limites melhorados sobre algumas ou todas as operações de dicionário e são mencionadas nas notas de capítulos de todo este livro.

- As *estruturas de dados de grafos dinâmicos* admitem diversas consultas, ao mesmo tempo que permitem que a estrutura de um grafo se altere por meio de operações que inserem ou eliminam vértices ou arestas. Os exemplos das consultas admitidas incluem conectividade de vértices [144], conectividade de arestas, árvores espalhadas mínimas [143], biconectividade e fecho transitivo [142].

Notas do capítulo em todo o livro mencionam estruturas de dados adicionais.



## Capítulo 18

# Árvores B

As árvores B são árvores de pesquisa balanceadas projetadas para funcionar bem em discos magnéticos ou outros dispositivos de armazenamento secundário de acesso direto. As árvores B são semelhantes às árvores vermelho-preto (Capítulo 13), mas são melhores para minimizar operações de E/S de disco. Muitos sistemas de bancos de dados usam árvores B ou variações de árvores B para armazenar informações.

As árvores B diferem significativamente das árvores vermelho-preto pelo fato de que os nós de árvores B podem ter muitos filhos, desde uma dezena até milhares. Isto é, o “fator de ramificação” de uma árvore B pode ser bastante grande, embora normalmente seja determinado por características da unidade de disco usado. As árvores B são semelhantes às árvores vermelho-preto no fato de que toda árvore B de  $n$  nós tem altura  $O(\lg n)$ , embora a altura de uma árvore B possa ser consideravelmente menor que a altura de uma árvore vermelho-preto, porque seu fator de ramificação pode ser muito maior. Então, as árvores B também podem ser usadas para implementar muitas operações sobre conjuntos dinâmicos no tempo  $O(\lg n)$ .

As árvores B generalizam árvores de pesquisa binária de maneira natural. A Figura 18.1 mostra uma árvore B simples. Se um nó interno  $x$  de uma árvore B contém  $n[x]$  chaves, então  $x$  tem  $n[x] + 1$  filhos. As chaves no nó  $x$  são usadas como pontos de divisão que separam o intervalo de chaves manipuladas por  $x$  em  $n[x] + 1$  subintervalos, cada qual manipulado por um filho de  $x$ . Quando procuramos por uma chave em uma árvore B, tomamos uma decisão de  $(n[x] + 1)$  modos, com base em comparações com as  $n[x]$  chaves armazenadas no nó  $x$ . A estrutura de nós de folhas difere da estrutura de nós internos; examinaremos essas diferenças na Seção 18.1.

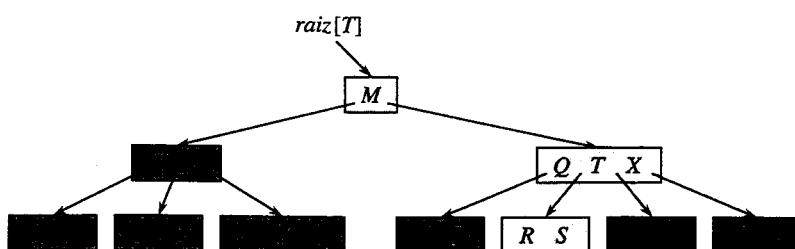


FIGURA 18.1 Uma árvore B cujas chaves são as consoantes do alfabeto. Um nó interno  $x$  contendo  $n[x]$  chaves tem  $n[x] + 1$  filhos. Todas as folhas estão na mesma profundidade na árvore. Os nós levemente sombreados são examinados em uma pesquisa que busca a letra R

A Seção 18.1 fornece uma definição precisa de árvores B e prova que a altura de uma árvore B cresce apenas de forma logarítmica com o número de nós que contém. A Seção 18.2 descreve como procurar por uma chave e inserir uma chave em uma árvore B, e a Seção 18.3 discute a eliminação. Porém, antes de prosseguirmos, precisamos indagar por que estruturas de dados projetadas para o trabalho em um disco magnético são avaliadas de maneira diferente das estruturas de dados projetadas para atuar na memória de acesso aleatório principal.

## Estruturas de dados no espaço de armazenamento secundário

Existem muitas tecnologias diferentes disponíveis para fornecer capacidade de memória em um sistema de computador. A **memória primária** (ou **memória principal**) de um sistema de computador consiste normalmente em chips de memória de silício. Essa tecnologia é em geral duas ordens de magnitude mais dispendiosa por bit armazenado que a tecnologia de armazenamento magnético, como fitas ou discos. Um sistema de computador típico tem **armazenamento secundário** baseado em discos magnéticos; a quantidade de tal armazenamento secundário com freqüência excede a quantidade de memória primária em pelo menos duas ordens de magnitude.

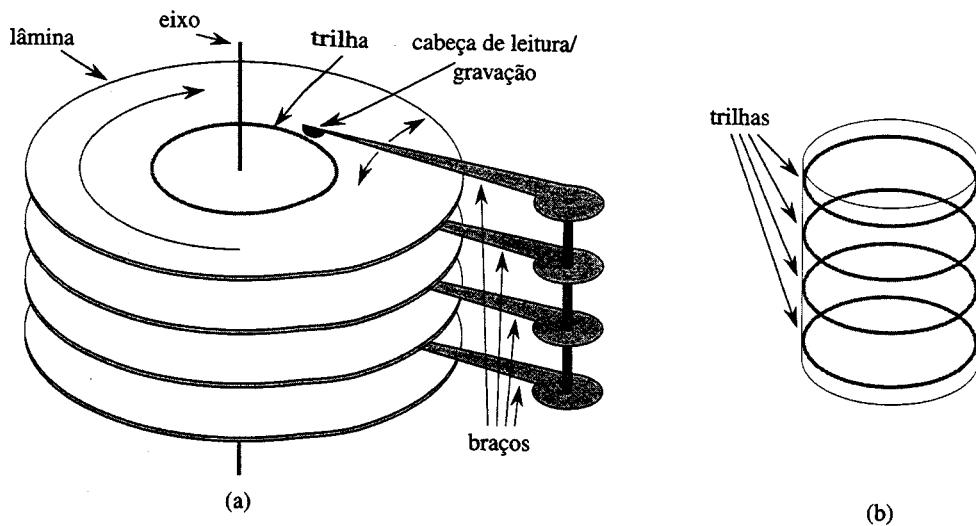
A Figura 18.2 mostra uma unidade de disco típica. A unidade consiste em várias *lâminas*, que giram a uma velocidade constante em torno de um *eixo* comum. A superfície de cada lâmina é coberta por um material magnetizável. O lâmina é lida ou gravada por uma *cabeça* na extremidade de um *braço*. Os braços estão fisicamente conectados, ou “associados”, e podem mover suas cabeças para dentro ou para fora na direção do eixo. Quando uma determinada cabeça está estacionária, a superfície que passa sob ela é chamada uma *trilha*. As cabeças de leitura/gravação estão alinhadas verticalmente o tempo todo e, portanto, o conjunto de trilhas sob elas é acessado simultaneamente. A Figura 18.2(b) mostra esse conjunto de trilhas, conhecido como *cilindro*.

Embora os discos sejam mais econômicos e tenham maior capacidade que a memória principal, eles são muito, muito mais lentos, porque têm peças móveis. Há dois componentes para o movimento mecânico: a rotação da lâmina e o movimento do braço. Na época em que esta edição foi escrita, discos comerciais giravam a velocidades de 5.400 a 15.000 revoluções por minuto (RPM), sendo 7.200 RPM a mais comum. Apesar de 7.200 RPM parecer rápido, uma rotação demora 8,33 milissegundos, quase cinco ordens de magnitude demorada que os tempos de acesso de 100 nanosegundos comumente encontrados na memória de silício. Em outras palavras, se tivéssemos de esperar uma rotação completa para um item específico cair sob a cabeça de leitura/gravação, poderíamos acessar a memória principal quase 100.000 vezes durante esse período! Em média, temos de esperar apenas por metade de uma rotação mas, ainda assim, a diferença em tempos de acesso para a memória de silício *versus* discos é enorme. A movimentação dos braços também demora algum tempo. Na época em que escrevemos, os tempos médios de acesso para discos comerciais estão no intervalo de 3 a 9 milissegundos.

Para amortizar o tempo gasto na espera por movimentos mecânicos, os discos acessam não apenas um item, mas vários de cada vez. As informações são divididas em diversas *páginas* de bits de igual tamanho que apareçam sucessivamente dentro de cilindros, e cada leitura ou gravação de disco inclui uma ou mais páginas inteiras. Para um disco típico, uma página pode ter  $2^{11}$  a  $2^{14}$  bytes de comprimento. Uma vez que a cabeça de leitura/gravação está posicionada corretamente e o disco gira até o início da página desejada, a leitura ou gravação de um disco magnético é inteiramente eletrônica (exceto pela rotação do disco), e grandes quantidades de dados podem ser lidas ou gravadas com rapidez.

Muitas vezes, demora mais tempo para se obter acesso a uma página de informações e fazer a leitura da página de um disco que o tempo necessário para o computador examinar todas as informações lidas. Por essa razão, examinaremos separadamente neste capítulo os dois componentes principais do tempo de execução:

- O número de acessos ao disco.
- O tempo de CPU (ou de computação).



**FIGURA 18.2** (a) Uma unidade de disco típica. Ela é composta por várias lâminas que giram em torno de um eixo. Cada lâmina é lida e gravada com uma cabeça na extremidade de um braço. Os braços são associados de modo a moverem suas cabeças em conjunto. Aqui, os braços giram em torno de um eixo pivô comum. Uma trilha é a superfície que passa sob a cabeça de leitura/gravação quando ela é estacionária. (b) Um cilindro consiste em um conjunto de trilhas concêntricas

O número de acessos ao disco é medido em termos do número de páginas de informações que precisam ser lidas do disco ou gravadas nele. Observamos que o tempo de acesso ao disco não é constante – ele depende da distância entre a trilha atual e a trilha desejada, e também do estado inicial de rotação do disco. Contudo, usaremos o número de páginas lidas ou gravadas como uma aproximação inicial bruta do tempo total gasto no acesso ao disco.

Em uma aplicação típica de árvore B, a quantidade de dados manipulados é tão grande que os dados não cabem todos na memória principal de uma só vez. Os algoritmos de árvores B copiam páginas selecionadas do disco para a memória principal conforme necessário e gravam novamente em disco as páginas que foram alteradas. Como os algoritmos de árvores B só precisam de um número constante de páginas na memória principal em qualquer instante, o tamanho da memória principal não limita o tamanho das árvores B que podem ser manipuladas.

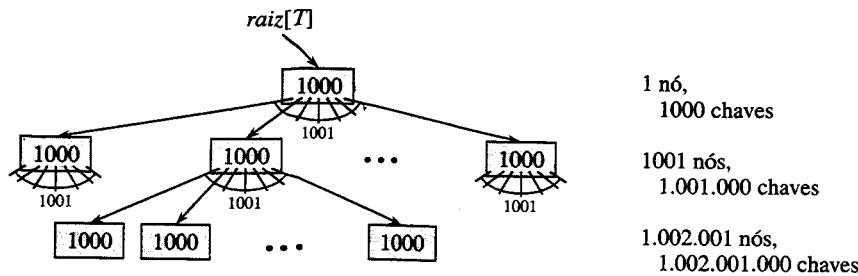
Modelamos operações de disco em nosso pseudocódigo da maneira ilustrada a seguir. Seja  $x$  um ponteiro para um objeto. Se o objeto estiver atualmente na memória principal do computador, então poderemos fazer referência aos campos do objeto do modo usual: por exemplo,  $chave[x]$ . Contudo, se o objeto referenciado por  $x$  residir no disco, então teremos de executar a operação  $\text{DISK-READ}(x)$  para ler o objeto  $x$  e inseri-lo na memória principal, antes que seus campos possam ser referenciados. (Supomos que, se  $x$  já estiver na memória principal, então  $\text{DISK-READ}(x)$  não exigirá nenhum acesso ao disco; ele será “não operacional”.) De modo semelhante, a operação  $\text{DISK-WRITE}(x)$  é usada para gravar quaisquer alterações que tenham sido efetuadas nos campos do objeto  $x$ . Ou seja, o padrão típico de trabalho com um objeto é dado a seguir.

```

 $x \leftarrow$  um ponteiro para algum objeto
 $\text{DISK-READ}(x)$ 
    operações que têm acesso e/ou modificam os campos de  $x$ 
 $\text{DISK-WRITE}(x)$       > Omitida se nenhum campo de  $x$  foi alterado.
    outras operações que têm acesso mas não modificam campos de  $x$ 

```

O sistema só pode manter um número limitado de páginas na memória principal em qualquer instante. Presumiremos que as páginas que não estão mais em uso são retiradas da memória principal pelo sistema; nossos algoritmos de árvores B ignorarão essa questão.



**FIGURA 18.3** Uma árvore B de altura 2 contendo mais de um bilhão de chaves. Cada nó interno e cada folha contém 1000 chaves. Existem 1001 nós na profundidade 1 e mais de um milhão de folhas na profundidade 2. Mostramos dentro de cada nó  $x$  o valor de  $n[x]$ , o número de chaves em  $x$

Tendo em vista que, na maioria dos sistemas, o tempo de execução de um algoritmo de árvore B é determinado principalmente pelo número de operações DISK-READ e DISK-WRITE que executa, é sensato usar essas operações de forma eficiente, fazendo-as ler ou gravar o máximo de informações possíveis. Desse modo, um nó de árvore B é normalmente tão grande quanto uma página de disco inteira. O número de filhos que um nó de árvore B pode ter é então limitado pelo tamanho de uma página de disco.

Para uma grande árvore B armazenada em um disco, fatores de ramificação entre 50 e 2000 são usados com freqüência, dependendo do tamanho de uma chave em relação ao tamanho de uma página. Um grande fator de ramificação reduz drasticamente tanto a altura da árvore quanto o número de acessos ao disco necessários para encontrar qualquer chave. A Figura 18.3 mostra uma árvore B com um fator de ramificação igual a 1001 e altura 2 que pode armazenar mais de um bilhão de chaves; não obstante, como o nó de raiz pode ser mantido permanentemente na memória principal, no máximo apenas *dois* acessos ao disco são exigidos para encontrar qualquer chave nessa árvore!

## 18.1 Definição de árvores B

Para manter tudo em termos simples, supomos – como fizemos no caso das árvores de pesquisa binária e no caso das árvores vermelho-preto – que quaisquer “informações satélite” associadas a uma chave estão armazenadas no mesmo nó em que está a chave. Na prática, realmente seria possível armazenar com cada chave apenas um ponteiro para outra página de disco contendo as informações satélite correspondentes a essa chave. O pseudocódigo deste capítulo pressupõe implicitamente que as informações satélite associadas a uma chave, ou o ponteiro para tais informações satélite, viajam com a chave sempre que a chave é deslocada de um nó até outro nó. Outra organização de árvore B comumente utilizada, conhecida como **árvore B<sup>+</sup>**, armazena todas as informações satélite nas folhas e só armazena ponteiros de chaves e filhos nos nós internos, maximizando assim o fator de ramificação dos nós internos.

Uma **árvore B T** é uma árvore enraizada (com raiz identificada por  $raiz[T]$ ) que tem as propriedades a seguir.

1. Todo nó  $x$  tem os seguintes campos:
  - a.  $n[x]$ , o número de chaves atualmente armazenadas no nó  $x$ ,
  - b. as próprias  $n[x]$  chaves, armazenadas em ordem não decrescente, de modo que  $chave_1[x] \leq chave_2[x] \leq \dots \leq chave_{n[x]}[x]$  e
  - c.  $folha[x]$ , um valor booleano que é TRUE se  $x$  é uma folha, e FALSE se  $x$  é um nó interno.
2. Cada nó interno  $x$  também contém  $n[x] + 1$  ponteiros  $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$  para seus filhos. Os nós de folhas não têm filhos, e assim seus campos  $c_i$  são indefinidos.

3. As chaves  $chave_i[x]$  separam os intervalos de chaves armazenadas em cada subárvore: se  $k_i$  é qualquer chave armazenada na subárvore com raiz  $c_i[x]$ , então

$$k_1 \leq chave_1[x] \leq k_2 \leq chave_2[x] \leq \dots \leq chave_{n[x]}[x] \leq k_{n[x]+1}.$$

4. Toda folha tem a mesma profundidade, que é a altura  $b$  da árvore.
5. Existem limites inferiores e superiores sobre o número de chaves que um nó pode conter. Esses limites podem ser expressos em termos de um inteiro fixo  $t \geq 2$  chamado **grau mínimo** da árvore B:
- Todo nó diferente da raiz deve ter pelo menos  $t - 1$  chaves. Desse modo, todo nó interno diferente da raiz tem pelo menos  $t$  filhos. Se a árvore é não vazia, a raiz deve ter pelo menos uma chave.
  - Todo nó pode conter no máximo  $2t - 1$  chaves. Então, um nó interno pode ter no máximo  $2t$  filhos. Dizemos que um nó é **completo** se ele contém exatamente  $2t - 1$  chaves.<sup>1</sup>

A árvore B mais simples ocorre quando  $t = 2$ . Todo nó interno tem então 2, 3 ou 4 filhos, e temos uma **árvore 2-3-4**. Na prática, porém, em geral são utilizados valores de  $t$  muito maiores.

## A altura de uma árvore B

O número de acessos ao disco exigidos para a maioria das operações em uma árvore B é proporcional à altura da árvore B. Agora, vamos analisar a altura de uma árvore B no pior caso.

### **Teorema 18.1**

Se  $n \geq 1$ , então, para qualquer árvore B  $T$  de  $n$  nós de altura  $b$  e grau mínimo  $t \geq 2$ ,

$$b \leq \log_t \frac{n+1}{2}.$$

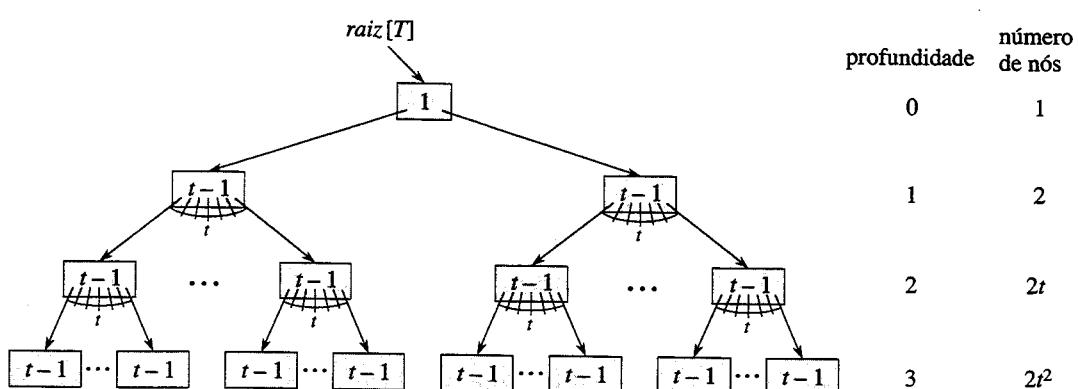


FIGURA 18.4 Uma árvore B de altura 3 contendo um número mínimo possível de chaves. Mostramos dentro de cada nó  $x$  o valor de  $n/x$

<sup>1</sup> Outra variante comum em uma árvore B, conhecida como uma **árvore B\***, exige que cada nó interno esteja pelo menos 2/3 completo, em vez de pelo menos metade completo, como uma árvore B exige.

**Prova** Se uma árvore B tem altura  $b$ , o número de seus nós é minimizado quando a raiz contém uma chave e todos os outros nós contêm  $t - 1$  chaves. Nesse caso, existem 2 nós na profundidade 1,  $2t$  nós na profundidade 2,  $2t^2$  nós na profundidade 3 e assim por diante, até a profundidade  $b$ , em que existem pelo menos  $2t^{b-1}$  nós. A Figura 18.4 ilustra tal árvore para  $b = 3$ . Desse modo, o número  $n$  de chaves satisfaz à desigualdade

$$\begin{aligned} n &\geq 1 + (t-1) \sum_{i=1}^b 2t^{i-1} \\ &= 1 + 2(t-1) \left( \frac{t^b - 1}{t-1} \right) \\ &= 2t^b - 1. \end{aligned}$$

Por álgebra simples, obtemos  $t^b \leq (n+1)/2$ . O uso de logaritmos de base  $t$  de ambos os lados prova o teorema. ■

Aqui vemos a capacidade das árvores B, quando comparadas a árvores vermelho-preto. Embora a altura da árvore cresça na proporção  $O(\lg n)$  em ambos os casos (lembre-se de que  $t$  é uma constante), para as árvores B, a base do logaritmo pode ser muitas vezes maior. Desse modo, as árvores B pouparam um fator de aproximadamente  $\lg t$  sobre as árvores vermelho-preto no número de nós examinados para a maioria das operações de árvores. Tendo em vista que o exame de um nó arbitrário em uma árvore normalmente exige um acesso ao disco, o número de acessos ao disco é substancialmente reduzido.

## Exercícios

### 18.1-1

Por que não permitimos um grau mínimo  $t = 1$ ?

### 18.1-2

Para que valores de  $t$  a árvore da Figura 18.1 é uma árvore B válida?

### 18.1-3

Mostre todas as árvores B válidas de grau mínimo 2 que representam  $\{1, 2, 3, 4, 5\}$ .

### 18.1-4

Como uma função do grau mínimo  $t$ , qual é o número máximo de chaves que podem ser armazenadas em uma árvore B de altura  $b$ ?

### 18.1-5

Descreva a estrutura de dados que resultaria se cada nó preto em uma árvore vermelho-preto absorvesse seus filhos vermelhos, incorporando os filhos de seus filhos a seus próprios filhos.

## 18.2 Operações básicas sobre árvores B

Nesta seção, apresentamos os detalhes das operações B-TREE-SEARCH, B-TREE-CREATE e B-TREE-INSERT. Nesses procedimentos, adotamos duas convenções:

- A raiz da árvore B está sempre na memória principal, de forma que uma operação DISK-READ na raiz nunca é exigida; porém, uma operação DISK-WRITE da raiz é exigida sempre que o nó de raiz é modificado.

- Quaisquer nós repassados como parâmetros já devem ter tido uma operação DISK-READ executada sobre eles.

Os procedimentos que apresentamos são todos algoritmos de “uma passagem” que prosseguem em sentido descendente a partir da raiz da árvore, sem terem de subir de volta.

## Pesquisa em uma árvore B

Pesquisar em uma árvore B é muito semelhante a pesquisar em uma árvore de pesquisa binária, exceto pelo fato de que, em vez de tomar uma decisão de ramificação binária ou de “duas vias” em cada nó, tomamos uma decisão de ramificação de várias vias, de acordo com o número de filhos do nó. Mais precisamente, em cada nó interno  $x$ , tomamos uma decisão de ramificação de  $(n[x] + 1)$  vias.

B-TREE-SEARCH é uma generalização direta do procedimento TREE-SEARCH definido para árvores de pesquisa binária. B-TREE-SEARCH toma como entrada um ponteiro para o nó de raiz  $x$  de uma subárvore e uma chave  $k$  a ser pesquisada nessa subárvore. A chamada de nível superior é portanto da forma B-TREE-SEARCH( $raiz[T], k$ ). Se  $k$  está na árvore B, B-TREE-SEARCH retorna o par ordenado  $(y, i)$  consistindo em um nó  $y$  e um índice  $i$  tal que  $chave_i[y] = k$ . Caso contrário, o valor NIL é retornado.

```

B-TREE-SEARCH( $x, k$ )
1  $i \leftarrow 1$ 
2 while  $i \leq n[x]$  e  $k > chave_i[x]$ 
3   do  $i \leftarrow i + 1$ 
4   if  $i \leq n[x]$  e  $k = chave_i[x]$ 
5     then return  $(x, i)$ 
6   if  $folha[x]$ 
7     then return NIL
8   else DISK-READ( $c_i[x]$ )
9     return B-TREE-SEARCH( $c_i[x], k$ )

```

Usando um procedimento de pesquisa linear, as linhas 1 a 3 encontram o menor  $i$  tal que  $k \leq chave_i[x]$ , ou então definem  $i$  como  $n[x] + 1$ . As linhas 4 e 5 verificam se agora descobrimos a chave, retornando se tivermos descoberto. As linhas 6 a 9 encerram uma pesquisa malsucedida (se  $x$  é uma folha) ou usam a recursão para pesquisar a subárvore apropriada de  $x$ , depois de executar a necessária operação DISK-READ sobre esse filho.

A Figura 18.1 ilustra a operação de B-TREE-SEARCH; os nós levemente sombreados são examinados durante uma pesquisa pela chave  $R$ .

Como no procedimento TREE-SEARCH para árvores de pesquisa binária, os nós encontrados durante a recursão formam um caminho descendente desde a raiz da árvore. O número de páginas de disco às quais B-TREE-SEARCH tem acesso é portanto  $\Theta(b) = \Theta(\log_b n)$ , onde  $b$  é a altura da árvore B e  $n$  é o número de chaves na árvore B. Tendo em vista que  $n[x] < 2t$ , o tempo tomado pelo loop while das linhas 2 e 3 dentro de cada nó é  $O(t)$ , e o tempo total da CPU é  $O(tb) = O(t \log_t n)$ .

## Como criar uma árvore B vazia

Para construir uma árvore B, primeiro utilizamos B-TREE-CREATE para criar um nó de raiz vazio, e depois chamamos B-TREE-INSERT para adicionar novas chaves. Esses dois procedimentos usam um procedimento auxiliar ALLOCATE-NODE, que aloca uma página de disco para ser usada como um novo nó no tempo  $O(1)$ . Podemos presumir que um nó criado por ALLOCATE-NODE não exige nenhuma operação DISK-READ, pois ainda não existe nenhuma informação útil armazenada no disco para esse nó.

```

B-TREE-CREATE( $T$ )
1  $x \leftarrow \text{ALLOCATE-NODE}()$ 
2  $\text{folha}[x] \leftarrow \text{TRUE}$ 
3  $n[x] \leftarrow 0$ 
4  $\text{DISK-WRITE}(x)$ 
5  $\text{raiz}[T] \leftarrow x$ 

```

B-TREE-CREATE exige  $O(1)$  operações de disco e o tempo de CPU  $O(1)$ .

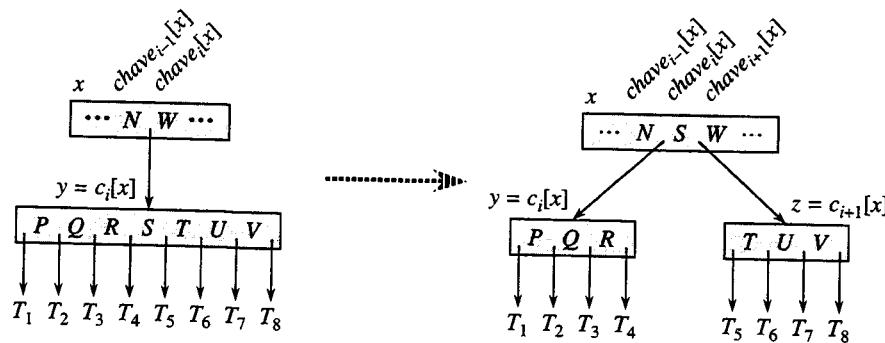


FIGURA 18.5 A divisão de um nó com  $t = 4$ . O nó  $y$  é dividido em dois nós,  $y$  e  $z$ , e a chave mediana  $S$  de  $y$  é movida para cima até o pai de  $y$

### A inserção de uma chave em uma árvore B

Inserir uma chave em uma árvore B é significativamente mais complicado que inserir uma chave em uma árvore de pesquisa binária. Como ocorre no caso das árvores de pesquisa binária, procuramos pela posição de folha em que devemos inserir a nova chave. Porém, como em uma árvore B, não podemos simplesmente criar um novo nó de folha e inseri-lo, pois a árvore resultante deixaria de ser uma árvore B válida. Em vez disso, inserimos a nova chave em um nó de folha existente. Tendo em vista que não podemos inserir uma chave em um nó de folha completo, introduzimos uma operação que *divide* um nó completo  $y$  (que tem  $2t - 1$  chaves) ao redor de sua **chave mediana**  $\text{chave}_t[y]$  em dois nós que têm  $t - 1$  chaves cada. A chave mediana se desloca para cima até o pai de  $y$  para identificar o ponto de divisão entre as duas novas árvores. Contudo, se o pai de  $y$  também está completo, ele deve ser dividido antes de ser possível inserir a nova chave e, portanto, essa necessidade de dividir nós completos pode se propagar para cima por toda a árvore.

Como no caso de uma árvore de pesquisa binária, podemos inserir uma chave em uma árvore B em uma única passagem para baixo na árvore, desde a raiz até uma folha. Para fazer isso, não esperamos descobrir se realmente precisaremos dividir um nó completo a fim de fazer a inserção. Em vez disso, à medida que descemos a árvore procurando pela posição à qual pertence a nova chave, dividimos cada nó completo que encontramos pelo caminho (inclusive a própria folha). Desse modo, sempre que queremos dividir um nó completo  $y$ , temos a certeza de que seu pai não é completo.

### A divisão de um nó em uma árvore B

O procedimento B-TREE-SPLIT-CHILD toma como entrada um nó interno *não completo*  $x$  (que se presume estar na memória principal), um índice  $i$  e um nó  $y$  (que também se presume estar na memória principal) tal que  $y = c_i[x]$  é um filho *completo* de  $x$ . Então, o procedimento divide esse filho em dois e ajusta  $x$  de modo que ele tenha agora um filho adicional. (Para dividir uma raiz completa, primeiro transformaremos a raiz em um filho de um novo nó raiz vazio, de modo a podermos usar B-TREE-SPLIT-CHILD. Assim, a árvore cresce uma unidade em altura; a divisão é o único meio pelo qual a árvore cresce.)

A Figura 18.5 ilustra esse processo. O nó completo  $y$  é dividido aproximadamente em sua chave mediana  $S$ , que é deslocada para cima até o nó  $x$  pai de  $y$ . As chaves em  $y$  que são maiores que a chave mediana são inseridas em um novo nó  $z$ , o qual se torna um novo filho de  $x$ .

```
B-TREE-SPLIT-CHILD( $x, i, y$ )
1  $z \leftarrow \text{ALLOCATE-NODE}()$ 
2  $\text{folba}[z] \leftarrow \text{folba}[y]$ 
3  $n[z] \leftarrow t - 1$ 
4 for  $j \leftarrow 1$  to  $t - 1$ 
5   do  $\text{chave}_j[z] \leftarrow \text{chave}_{j+t}[y]$ 
6 if not  $\text{folba}[y]$ 
7   then for  $j \leftarrow 1$  to  $t$ 
8     do  $c_j[z] \leftarrow c_{j+t}[y]$ 
9  $n[y] \leftarrow t - 1$ 
10 for  $j \leftarrow n[x] + 1$  downto  $i + 1$ 
11   do  $c_{j+1}[x] \leftarrow c_j[x]$ 
12    $c_{i+1}[x] \leftarrow z$ 
13 for  $j \leftarrow n[x]$  downto  $i$ 
14   do  $\text{chave}_{j+1}[x] \leftarrow \text{chave}_j[x]$ 
15    $\text{chave}_i[x] \leftarrow \text{chave}_t[y]$ 
16    $n[x] \leftarrow n[x] + 1$ 
17 DISK-WRITE( $y$ )
18 DISK-WRITE( $z$ )
19 DISK-WRITE( $x$ )
```

B-TREE-SPLIT-CHILD funciona pelo método direto de “recortar e colar”. Aqui,  $y$  é o  $i$ -ésimo filho de  $x$  e é o nó que está sendo dividido. O nó  $y$  tem originalmente  $2t$  filhos ( $2t - 1$  chaves), mas é reduzido a  $t$  filhos ( $t - 1$  chaves) por essa operação. O nó  $z$  “adota” os  $t$  maiores filhos ( $t - 1$  chaves) de  $y$ , e  $z$  se torna um novo filho de  $x$ , posicionado logo após  $y$  na tabela de filhos de  $x$ . A chave mediana de  $y$  se desloca para cima até se tornar a chave em  $x$  que separa  $y$  e  $z$ .

As linhas 1 a 8 criam o nó  $z$  e dão a ele as  $t - 1$  chaves maiores e os  $t$  filhos correspondentes de  $y$ . A linha 9 ajusta a contagem de chaves para  $y$ . Finalmente, as linhas 10 a 16 inserem  $z$  como um filho de  $x$ , movem a chave mediana de  $y$  para cima até  $x$ , a fim de separar  $y$  de  $z$ , e ajustam a contagem de chaves de  $x$ . As linhas 17 a 19 gravam todas as páginas de disco modificadas. O tempo de CPU usado por B-TREE-SPLIT-CHILD é  $O(t)$ , devido aos loops nas linhas 4-5 e 7-8. (Os outros loops são executados para  $O(t)$  iterações.) O procedimento executa  $O(1)$  operações de disco.

### Inserção de uma chave em uma árvore B em uma única passagem pela árvore

A inserção de uma chave  $k$  em uma árvore B denominada  $T$  de altura  $b$  é feita em uma única passagem descendente na árvore, exigindo  $O(b)$  acessos ao disco. O tempo de CPU exigido é  $O(tb) = O(t \log_t n)$ . O procedimento B-TREE-INSERT utiliza B-TREE-SPLIT-CHILD para garantir que a recursão nunca descerá até um nó completo.

```
B-TREE-INSERT( $T, k$ )
1  $r \leftarrow \text{raiz}[T]$ 
2 if  $n[r] = 2t - 1$ 
3   then  $s \leftarrow \text{ALLOCATE-NODE}()$ 
4      $\text{raiz}[T] \leftarrow s$ 
5      $\text{folba}[s] \leftarrow \text{FALSE}$ 
6      $n[s] \leftarrow 0$ 
```

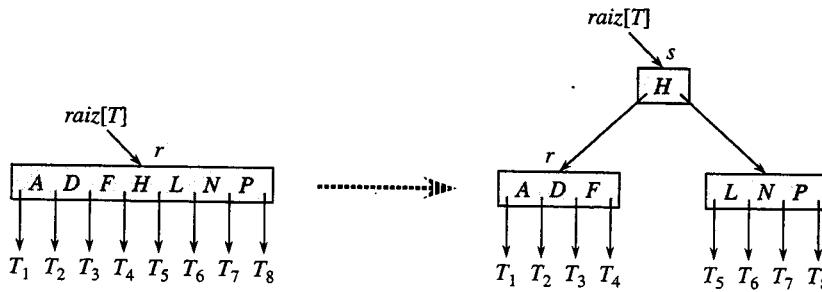


FIGURA 18.6 Divisão da raiz com  $t = 4$ . O nó de raiz  $r$  é dividido em dois, e é criado um novo nó de raiz  $s$ . A nova raiz contém a chave mediana de  $r$  e tem as duas metades de  $r$  como filhos. A árvore B cresce em altura uma unidade quando a raiz é dividida

```

7       $c_1[s] \leftarrow r$ 
8      B-TREE-SPLIT-CHILD( $s, 1, r$ )
9      B-TREE-INSERT-NONFULL( $s, k$ )
10     else B-TREE-INSERT-NONFULL( $r, k$ )

```

As linhas 3 a 9 tratam o caso no qual o nó de raiz  $r$  é completo: a raiz é dividida e um novo nó  $s$  (que tem dois filhos) se torna a raiz. A divisão da raiz é o único modo de aumentar a altura de uma árvore B. A Figura 18.6 ilustra esse caso. Diferente de uma árvore de pesquisa binária, uma árvore B aumenta em altura na parte superior, em vez de aumentar na parte inferior. O procedimento termina chamando B-TREE-INSERT-NONFULL para executar a inserção da chave  $k$  na árvore com raiz no nó de raiz não-completo. B-TREE-INSERT-NONFULL executa a recursão descendo a árvore conforme necessário, e garante em todos os momentos que o nó ao qual ele recorre não está completo, através de uma chamada a B-TREE-SPLIT-CHILD à medida que se torna necessário.

O procedimento recursivo auxiliar B-TREE-INSERT-NONFULL insere a chave  $k$  no nó  $x$ , que se presume ser não cheio quando o procedimento é chamado. A operação de B-TREE-INSERT e a operação recursiva de B-TREE-INSERT-NONFULL garantem que essa hipótese é verdadeira.

#### B-TREE-INSERT-NONFULL( $x, k$ )

```

1  $i \leftarrow n[x]$ 
2 if  $folha[x]$ 
3   then while  $i \geq 1$  e  $k < chave_i[x]$ 
4     do  $chave_{i+1}[x] \leftarrow chave_i[x]$ 
5      $i \leftarrow i - 1$ 
6    $chave_{i+1}[x] \leftarrow k$ 
7    $n[x] \leftarrow n[x] + 1$ 
8   DISK-WRITE( $x$ )
9 else while  $i \geq 1$  e  $k < chave_i[x]$ 
10   do  $i \leftarrow i - 1$ 
11    $i \leftarrow i + 1$ 
12   DISK-READ( $c_i[x]$ )
13   if  $n[c_i[x]] = 2t - 1$ 
14     then B-TREE-SPLIT-CHILD( $x, i, c_i[x]$ )
15     if  $k > chave_i[x]$ 
16       then  $i \leftarrow i + 1$ 
17   B-TREE-INSERT-NONFULL( $c_i[x], k$ )

```

O procedimento B-TREE-INSERT-NONFULL funciona da maneira descrita a seguir. As linhas 3 a 8 tratam o caso no qual  $x$  é um nó de folha, inserindo a chave  $k$  em  $x$ . Se  $x$  não é um nó de folha, então devemos inserir  $k$  no nó de folha apropriado na subárvore com raiz no nó interno  $x$ .

Nesse caso, as linhas 9 a 11 determinam o filho de  $x$  para o qual a recursão é descendente. A linha 13 detecta se a recursão desceria até um filho completo, e nesse caso a linha 14 utiliza B-TREE-SPLIT-CHILD para dividir esse filho em dois filhos não-completos, e as linhas 15 e 16 determinam qual dos dois filhos é agora o filho correto para o qual se deve descer. (Observe que não há necessidade de uma operação DISK-READ( $c_i[x]$ ) após a linha 16 incrementar  $i$ , pois a recursão descerá nesse caso até um filho que acaba de ser criado por B-TREE-SPLIT-CHILD.) Portanto, o efeito final das linhas 13 a 16 é garantir que o procedimento nunca recorrerá até um nó completo. Então, a linha 17 utiliza a recursão para inserir  $k$  na subárvore apropriada. A Figura 18.7 ilustra os diversos casos de inserção em uma árvore B.

O número de acessos ao disco executados por B-TREE-INSERT é  $O(b)$  para uma árvore B de altura  $b$ , pois somente  $O(1)$  operações DISK-READ e DISK-WRITE são executados entre chamadas a B-TREE-INSERT-NONFULL. O tempo total de CPU usado é  $O(tb) = O(t \log n)$ . Tendo em vista que B-TREE-INSERT-NONFULL é recursiva de final, ele pode ser implementado de modo alternativo como um loop while, demonstrando que o número de páginas que precisam estar na memória principal em qualquer instante é  $O(1)$ .

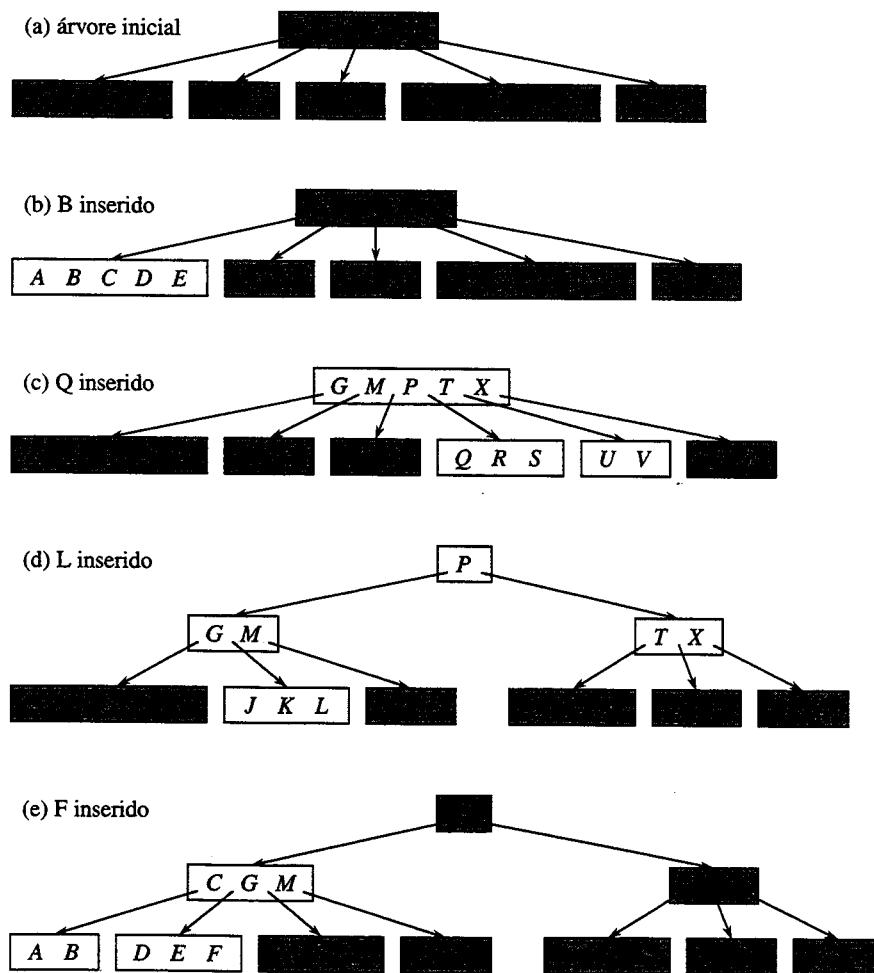


FIGURA 18.7 Inserção de chaves em uma árvore B. O grau mínimo  $t$  para essa árvore B é 3; assim, um nó pode conter no máximo 5 chaves. Nós que são modificados pelo processo de inserção estão levemente sombreados. (a) A árvore inicial para este exemplo. (b) O resultado da inserção de  $B$  na árvore inicial; essa é uma inserção simples em um nó de folha. (c) O resultado da inserção de  $Q$  na árvore anterior. O nó  $R S T UV$  é dividido em dois nós contendo  $R S$  e  $U V$ , a chave  $T$  é movida para cima até a raiz e  $Q$  é inserido na metade mais à esquerda das duas (o nó  $R S$ ). (d) O resultado da inserção de  $L$  na árvore anterior. A raiz é dividida de qualquer modo, pois ela é completa, e a árvore B cresce uma unidade em altura. Então,  $L$  é inserida na folha que contém  $J K$ . (e) O resultado da inserção de  $F$  na árvore anterior. O nó  $A B C D E$  é dividido antes de  $F$  ser inserido na metade mais à direita das duas (o nó  $D E$ )

## Exercícios

### 18.2-1

Mostre os resultados da inserção das chaves

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

nessa ordem em uma árvore B vazia com grau mínimo 2. Desenhe apenas as configurações da árvore imediatamente antes de ter de dividir algum nó, e desenhe também a configuração final.

### 18.2-2

Explique sob quais circunstâncias, se houver, operações DISK-READ ou DISK-WRITE redundantes são executadas durante o curso da execução de uma chamada a B-TREE-INSERT. (Uma operação DISK-READ redundante é uma operação DISK-READ para uma página que já está na memória. Uma operação DISK-WRITE redundante grava em disco uma página de informações idêntica à que já está armazenada lá.)

### 18.2-3

Explique como encontrar a chave mínima armazenada em uma árvore B e como encontrar o predecessor de uma dada chave armazenada em uma árvore B.

### 18.2-4 \*

Suponha que as chaves  $\{1, 2, \dots, n\}$  sejam inseridas em uma árvore B vazia com grau mínimo 2. Quantos nós tem a árvore B final?

### 18.2-5

Tendo em vista que nós de folhas não exigem nenhum ponteiro para filhos, eles poderiam usar um valor de  $t$  diferente (maior) de nós internos para o mesmo tamanho de página de disco. Mostre como modificar os procedimentos de criação e inserção em uma árvore B para tratar essa variação.

### 18.2-6

Suponha que B-TREE-SEARCH seja implementado para usar a pesquisa binária em lugar da pesquisa linear dentro de cada nó. Mostre que isso torna o tempo de CPU necessário igual a  $O(\lg n)$ , independentemente do modo como  $t$  poderia ser escolhido como uma função de  $n$ .

### 18.2-7

Suponha que o hardware de disco nos permite escolher arbitrariamente o tamanho de uma página de disco, mas que o tempo necessário para ler a página de disco seja  $a + bt$ , onde  $a$  e  $b$  são constantes especificadas, e  $t$  é o grau mínimo para uma árvore B que utiliza páginas do tamanho selecionado. Descreva como escolher  $t$  para minimizar (aproximadamente) o tempo de pesquisa na árvore B. Sugira um valor ótimo de  $t$  para o caso no qual  $a = 5$  milissegundos e  $b = 10$  microsegundos.

## 18.3 Eliminação de uma chave de uma árvore B

A eliminação de uma árvore B é análoga à inserção, embora um pouco mais complicada, porque uma chave pode ser eliminada de qualquer nó – não apenas de uma folha – e a eliminação de um nó interno exige que os filhos do nó sejam reorganizados. Como na inserção, devemos nos resguardar contra a possibilidade da eliminação produzir uma árvore cuja estrutura viole as propriedades de árvores B. Da mesma maneira que tivemos de assegurar que um nó não ficasse grande demais devido à inserção, devemos assegurar que um nó não ficará pequeno demais durante a eliminação (a não ser pelo fato de a raiz poder ter menos que o número mínimo  $t-1$  de chaves, embora ela não tenha permissão para ter mais que o número máximo  $2t - 1$  de chaves). Da mesma maneira que um algoritmo de inserção simples poderia ter de retornar se um nó no caminho onde a chave fosse inserida estivesse completo, uma abordagem simples para a eliminação po-

deria ter de retornar se um nó (diferente da raiz) ao longo do caminho no qual a chave tivesse de ser eliminada apresentasse o número mínimo de chaves.

Suponha que o procedimento B-TREE-DELETE tenha de eliminar a chave  $k$  da subárvore com raiz em  $x$ . Esse procedimento está estruturado para garantir que, sempre que B-TREE-DELETE for chamado recursivamente em um nó  $x$ , o número de chaves em  $x$  seja pelo menos o grau mínimo  $t$ . Observe que essa condição exige uma chave além do mínimo exigido pelas condições usuais de árvores B, de forma que às vezes uma chave talvez tenha de ser movida para dentro de um nó filho, antes da recursão descer até esse filho. Essa condição reforçada nos permite eliminar uma chave da árvore em uma única passagem descendente sem a necessidade de “voltar” (com uma única exceção, que explicaremos em breve). A especificação a seguir para eliminação de uma árvore B deve ser interpretada com a compreensão de que, se acontecer de o nó raiz  $x$  se tornar um nó interno sem nenhuma chave (essa situação pode ocorrer nos casos 2c e 3b a seguir), então  $x$  será eliminada e o único filho de  $x$ ,  $c_1[x]$ , se tornará a nova raiz da árvore, diminuindo a altura da árvore em uma unidade e preservando a propriedade que afirma que a raiz da árvore contém no mínimo uma chave (a menos que a árvore esteja vazia).

Descreveremos como a eliminação funciona, em vez de apresentarmos o pseudocódigo. A Figura 18.8 ilustra os diversos casos de eliminação de chaves de uma árvore B.

1. Se a chave  $k$  está no nó  $x$  e  $x$  é uma folha, eliminate a chave  $k$  de  $x$ .
2. Se a chave  $k$  está no nó  $x$  e  $x$  é um nó interno, faça o seguinte.
  - a. Se o filho  $y$  que precede  $k$  no nó  $x$  tem pelo menos  $t$  chaves, então encontre o predecessor  $k'$  de  $k$  na subárvore com raiz em  $y$ . Elimine recursivamente  $k'$ , e substitua  $k$  por  $k'$  em  $x$ . (Encontrar  $k'$  e eliminá-lo pode ser uma operação executada em uma única passagem descendente.)
  - b. Simetricamente, se o filho  $z$  que segue  $k$  no nó  $x$  tem pelo menos  $t$  chaves, então encontre o sucessor  $k'$  de  $k$  na subárvore com raiz em  $z$ . Elimine recursivamente  $k'$ , e substitua  $k$  por  $k'$  em  $x$ . (Encontrar  $k'$  e eliminá-lo pode ser uma operação executada em uma única passagem descendente.)
  - c. Caso contrário, se tanto  $y$  quanto  $z$  têm apenas  $t - 1$  chaves, faça a intercalação de  $k$  e todos os itens  $z$  em  $y$ , de modo que  $x$  perca tanto  $k$  quanto o ponteiro para  $z$ , e  $y$  conteña agora  $2t - 1$  chaves. Em seguida, libere  $z$  e eliminate recursivamente  $k$  de  $y$ .
3. Se a chave  $k$  não estiver presente no nó interno  $x$ , determine a raiz  $c_i[x]$  da subárvore apropriada que deve conter  $k$ , se  $k$  estiver absolutamente na árvore. Se  $c_i[x]$  tiver somente  $t - 1$  chaves, execute o passo 3a ou 3b conforme necessário para garantir que desceremos até um nó contendo pelo menos  $t$  chaves. Em seguida, encerre efetuando uma recursão sobre o filho apropriado de  $x$ .
  - a. Se  $c_i[x]$  tiver somente  $t - 1$  chaves, mas tiver um irmão com  $t$  chaves, forneça a  $c_i[x]$  uma chave extra, movendo uma chave de  $x$  para baixo até  $c_i[x]$ , movendo uma chave do irmão esquerdo ou direito imediato de  $c_i[x]$  para dentro de  $x$ , e movendo o ponteiro do filho apropriado do irmão para  $c_i[x]$ .
  - b. Se  $c_i[x]$  e todos os irmãos de  $c_i[x]$  têm  $t - 1$  chaves, faça a intercalação de  $c_i[x]$  com um único irmão, o que envolve mover uma chave de  $x$  para baixo até o novo nó intercalado, a fim de se tornar a chave mediana para esse nó.

Tendo em vista que a maioria das chaves em uma árvore B se encontra nas folhas, podemos esperar que, na prática, as operações de eliminação sejam usadas com maior freqüência para eliminar chaves de folhas. O procedimento B-TREE-DELETE atua então em uma passagem descendente pela árvore, sem ter de voltar a subir. Contudo, quando elimina uma chave em um nó interno, o procedimento efetua uma passagem descendente através da árvore, mas pode ter de retornar ao nó do qual a chave foi eliminada, a fim de substituir a chave por seu predecessor ou sucessor (casos 2a e 2b).

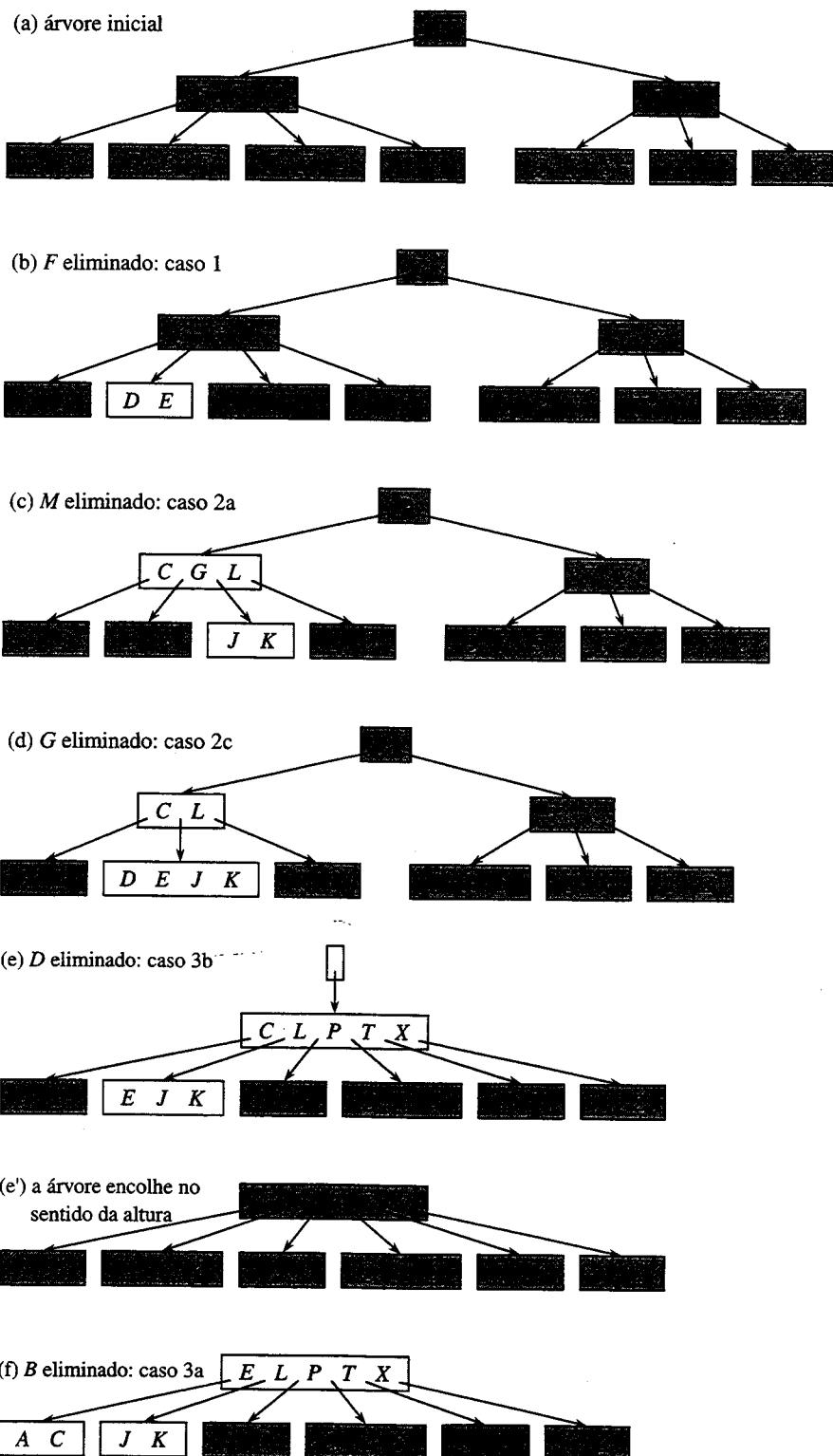


FIGURA 18.8 Eliminação de chaves de uma árvore B. O grau mínimo para essa árvore B é  $t = 3$ ; assim, um nó (diferente da raiz) não pode ter menos de 2 chaves. Nós que são modificados estão levemente sombreados. (a) A árvore B da Figura 18.7(e). (b) Eliminação de  $F$ . Esse é o caso 1: a simples eliminação de uma folha. (c) Eliminação de  $M$ . Esse é o caso 2a: o predecessor  $L$  de  $M$  é movido para cima, a fim de ocupar a posição de  $M$ . (d) Eliminação de  $G$ . Esse é caso 2c:  $G$  é empurrado para baixo até formar o nó  $DEGJK$ , e então  $G$  é eliminado dessa folha (caso 1). (e) Eliminação de  $D$ . Esse é o caso 3b: a recursão não pode descer até o nó  $CL$ , porque ele tem apenas 2 chaves; assim,  $P$  é empurrado para baixo e intercalado com  $CL$  e  $TX$  para formar  $CLPTX$ ; em seguida,  $D$  é eliminado de uma folha (caso 1). (e') Após (d), a raiz é eliminada e a árvore encolhe uma unidade em altura. (f) Eliminação de  $B$ . Esse é o caso 3a:  $C$  é movido para preencher a posição de  $B$ , e  $E$  é movido para preencher a posição de  $C$

Embora esse procedimento pareça complicado, ele envolve apenas  $O(b)$  operações de disco para uma árvore B de altura  $b$ , pois somente  $O(1)$  chamadas a DISK-READ e DISK-WRITE são efetuadas entre invocações recursivas do procedimento. O tempo de CPU necessário é  $O(tb) = O(t \log_t n)$ .

## Exercícios

### 18.3-1

Mostre os resultados da eliminação de  $C$ ,  $P$  e  $V$ , nessa ordem, da árvore da Figura 18.8(f).

### 18.3-2

Escreva o pseudocódigo para B-TREE-DELETE.

## Problemas

### 18-1 Pilhas no espaço de armazenamento secundário

Considere a implementação de uma pilha em um computador que tem uma quantidade relativamente pequena de memória primária rápida e uma quantidade relativamente grande de espaço de armazenamento em disco mais lento. As operações PUSH e POP são admitidas sobre valores de uma única palavra. A pilha a que desejamos dar suporte pode crescer até se tornar muito maior do que seria possível caber na memória, e assim a maior parte dela terá de ser armazenada em disco.

Uma implementação de pilha simples, embora ineficiente, mantém a pilha inteira no disco. Mantemos na memória um ponteiro de pilha, o qual é o endereço de disco do elemento do topo da pilha. Se o ponteiro tiver o valor  $p$ , o elemento superior será a  $(p \bmod m)$ -ésima palavra na página  $\lfloor p / m \rfloor$  do disco, onde  $m$  é o número de palavras por página.

Para implementar a operação PUSH, incrementamos o ponteiro da pilha, lemos a página apropriada no disco e a inserimos na memória, copiamos o elemento a ser empilhado para a palavra apropriada na página e gravamos a página de novo no disco. Uma operação POP é semelhante. Decrementamos o ponteiro da pilha, lemos a página apropriada no disco e retornamos à posição superior da pilha. Não precisamos gravar de novo a página, pois ela não foi modificada.

Pelo fato de operações de disco serem relativamente dispendiosas, usamos o número total de acessos ao disco como uma medida de mérito para qualquer implementação. Também levamos em conta o tempo da CPU, mas debitamos  $\Theta(m)$  por qualquer acesso de disco a uma página de  $m$  palavras.

- Assintoticamente, qual é o número de acessos ao disco no pior caso para  $n$  operações de pilhas com o uso dessa implementação simples? Qual é o tempo de CPU para  $n$  operações de pilhas? (Expresse sua resposta em termos de  $m$  e  $n$  para esta parte e para as partes subsequentes.)

Agora, considere uma implementação de pilha na qual mantemos na memória uma única página da pilha. (Também mantemos uma pequena quantidade de memória para controlar qual página está atualmente na memória.) Podemos executar uma operação de pilha somente se a página de disco relevante reside na memória. Se necessário, a página atualmente na memória pode ser gravada no disco e a nova página lida do disco para a memória. Se a página de disco relevante já estiver na memória, então não será necessário nenhum acesso ao disco.

- Qual é o número de acessos ao disco no pior caso exigido para  $n$  operações PUSH? Qual é o tempo de CPU?
- Qual é o número de acessos ao disco no pior caso exigido para  $n$  operações de pilhas? Qual é o tempo de CPU?

Agora, suponha que a pilha seja implementada mantendo-se duas páginas na memória (além de um número pequeno de palavras para contabilidade).

- d.** Descreva como gerenciar as páginas da pilha de modo que o número amortizado de acessos ao disco para qualquer operação de pilha seja  $O(1/m)$  e o tempo de CPU amortizado para qualquer operação de pilha seja  $O(1)$ .

### 18-2 Junção e divisão de árvores 2-3-4

A operação de *junção* toma dois conjuntos dinâmicos  $S'$  e  $S''$  e um elemento  $x$  tal que, para qualquer  $x' \in S'$  e  $x'' \in S''$ , temos  $chave[x'] < chave[x] < chave[x'']$ . Ela retorna um conjunto  $S = S' \cup \{x\} \cup S''$ . A operação de *divisão* é como uma junção “inversa”: dado um conjunto dinâmico  $S$  e um elemento  $x \in S$ , ela cria um conjunto  $S'$  que consiste em todos os elementos de  $S - \{x\}$  cujas chaves são menores que  $chave[x]$ , e um conjunto  $S''$  que consiste em todos os elementos em  $S - \{x\}$  cujas chaves são maiores que  $chave[x]$ . Neste problema, investigaremos como implementar essas operações sobre árvores 2-3-4. Supomos por conveniência que os elementos consistem apenas em chaves e que todos os valores de chaves são distintos.

- Mostre como manter, para todo nó  $x$  de uma árvore 2-3-4, a altura da subárvore com raiz em  $x$  como um campo  $altura[x]$ . Certifique-se de que sua implementação não afeta os tempos de execução assintóticos de pesquisa, inserção e eliminação.
- Mostre como implementar a operação de junção. Dadas duas árvores 2-3-4  $T'$  e  $T''$  e uma chave  $k$ , a junção deve ser executada no tempo  $O(1 + |b' - b''|)$ , onde  $b'$  e  $b''$  são as alturas de  $T'$  e  $T''$ , respectivamente.
- Considere o caminho  $p$  desde a raiz de uma árvore 2-3-4  $T$  até uma dada chave  $k$ , o conjunto  $S'$  de chaves em  $T$  que são menores que  $k$ , e o conjunto  $S''$  de chaves em  $T$  que são maiores que  $k$ . Mostre que  $p$  divide  $S'$  em um conjunto de árvores  $\{T'_0, T'_1, \dots, T'_m\}$  e um conjunto de chaves  $\{k'_1, k'_2, \dots, k'_m\}$ , onde para  $i = 1, 2, \dots, m$ , temos  $y < k'_i < z$  para quaisquer chaves  $y \in T'_{i-1}$  e  $z \in T'_i$ . Qual é o relacionamento entre as alturas de  $T'_{i-1}$  e  $T'_i$ ? Descreva o modo como  $p$  divide  $S''$  em conjuntos de árvores e chaves.
- Mostre como implementar a operação de divisão sobre  $T$ . Utilize a operação de junção para montar as chaves de  $S'$  em uma única árvore 2-3-4  $T'$  e as chaves de  $S''$  em uma única árvore 2-3-4  $T''$ . O tempo de execução da operação de divisão deve ser  $O(\lg n)$ , onde  $n$  é o número de chaves em  $T$ . (Sugestão: Os custos para as operações de junção devem se encaixar.)

## Notas do capítulo

Knuth [185], Aho, Hopcroft e Ullman [5] e ainda Sedgewick [269] apresentam discussões adicionais de esquemas de árvores balanceadas e árvores B. Comer [66] fornece uma pesquisa ampla de árvores B. Guibas e Sedgewick [135] discutem os relacionamentos entre vários tipos de esquemas de árvores balanceadas, inclusive árvores vermelho-preto e árvores 2-3-4.

Em 1970, J. E. Hopcroft criou as árvores 2-3, precursoras das árvores B e das árvores 2-3-4, nas quais todo nó interno tem dois ou três filhos. As árvores B foram introduzidas por Bayer e McCreight em 1972 [32]; eles não explicaram a escolha desse nome.

Bender, Demaine e Farach-Colton [37] estudaram como fazer árvores B funcionarem bem na presença de efeitos de hierarquia de memória. Seus algoritmos *sem memória do cache* funcionam de forma eficiente sem conhecer explicitamente os tamanhos de transferência de dados dentro da hierarquia de memória.

---

## *Capítulo 19*

### *Heaps binomiais*

Este capítulo e o Capítulo 20 apresentam estruturas de dados conhecidas como *heaps intercaláveis*, que admitem as cinco operações a seguir.

**MAKE-HEAP()** cria e retorna um novo heap que não contém nenhum elemento.

**INSERT( $H, x$ )** insere o nó  $x$ , cujo campo *chave* já foi preenchido, no heap  $H$ .

**MINIMUM( $H$ )** retorna um ponteiro para o nó do heap  $H$  cuja chave é mínima.

**EXTRACT-MIN( $H$ )** elimina o nó do heap  $H$  cuja chave é mínima, retornando um ponteiro para o nó.

**UNION( $H_1, H_2$ )** cria e retorna um novo heap que contém todos os nós dos heaps  $H_1$  e  $H_2$ . Os heaps  $H_1$  e  $H_2$  são “destruídos” por essa operação.

Além disso, as estruturas de dados nestes capítulos também admitem as duas operações a seguir.

**DECREASE-KEY( $H, x, k$ )** atribui ao nó  $x$  dentro do heap  $H$  o novo valor de chave  $k$ , que se supõe não ser maior que seu valor de chave atual.<sup>1</sup>

**DELETE( $H, x$ )** elimina o nó  $x$  do heap  $H$ .

Como mostra a tabela na Figura 19.1, se não precisamos da operação UNION, heaps binários comuns, como os que são utilizados em heapsort (Capítulo 6), funcionam bem. Outras operações além de UNION são executadas no tempo de pior caso  $O(\lg n)$  (ou melhor) sobre um heap binário. Porém, se a operação UNION tiver de ser admitida, os heaps binários terão um desempenho sofrível. Pela concatenação dos dois arranjos que contêm os heaps binários a serem intercalados e depois pela execução de MIN-HEAPIFY (consulte o Exercício 6.2-2), a operação UNION demora o tempo  $\Theta(n)$  no pior caso.

Neste capítulo, examinaremos os “heaps binomiais”, cujos limites de tempo no pior caso também são mostrados na Figura 19.1. Em particular, a operação UNION demora apenas o tempo  $O(\lg n)$  para intercalar dois heaps binomiais com um total de  $n$  elementos.

---

<sup>1</sup> Conforme mencionamos na introdução à Parte V, nossos heaps intercaláveis padrão são heaps mínimos intercaláveis, e assim as operações MINIMUM, EXTRACT-MIN e DECREASE-KEY se aplicam a eles. Como outra alternativa, poderíamos definir um *heap máximo intercalável* com as operações MAXIMUM, EXTRACT-MAX e INCREASE-KEY.

No Capítulo 20, exploraremos os heaps de Fibonacci, que têm limites de tempo ainda melhores para algumas operações. Entretanto, observe que os tempos de execução para heaps de Fibonacci na Figura 19.1 são limites de tempo amortizados, e não limites de tempo de pior caso por operação.

Procedimento	Heap binário (pior caso)	Heap binomial (pior caso)	Heap de Fibonacci (amortizado)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

FIGURA 19.1 Tempos de execução para operações sobre três implementações de heaps intercaláveis. O número de itens no(s) heap(s) no momento de uma operação é denotado por  $n$ .

Este capítulo ignora questões de alocação de nós antes da inserção e liberação de nós em seguida à eliminação. Supomos que o código que chama os procedimentos de heaps trata desses detalhes.

Heaps binários, heaps binomiais e heaps de Fibonacci são todos ineficientes em seu suporte da operação SEARCH; pode demorar um pouco até se encontrar um nó com uma determinada chave. Por essa razão, operações como DECREASE-KEY e DELETE, que se referem a um dado nó exigem um ponteiro para esse nó como parte de sua entrada. Como em nossa discussão de filas de prioridades na Seção 6.5, quando usarmos um heap intercalável em uma aplicação, freqüentemente armazenamos um descritor para o objeto da aplicação correspondente em cada elemento de heap intercalável, bem como um descritor para o elemento de heap intercalável correspondente em cada objeto da aplicação. A natureza exata desses descritores depende da aplicação e de sua implementação.

A Seção 19.1 define heaps binomiais depois de definir primeiro suas árvores binomiais constituintes. Ela também introduz uma representação particular de heaps binomiais. A Seção 19.2 mostra como podemos implementar operações sobre heaps binomiais nos limites de tempo dados na Figura 19.1.

## 19.1 Árvores binomiais e heaps binomiais

Um heap binomial é uma coleção de árvores binomiais, e assim esta seção começa definindo árvores binomiais e provando algumas propriedades fundamentais. Em seguida, definiremos heaps binomiais e mostraremos como eles podem ser representados.

### 19.1.1 Árvores binomiais

A **árvore binomial**  $B_k$  é uma árvore ordenada (ver Seção B.5.2) definida recursivamente. Conforme mostra a Figura 19.2(a), a árvore binomial  $B_0$  consiste em um único nó. A árvore binomial  $B_k$  consiste em duas árvores binomiais  $B_{k-1}$  que são **ligadas** uma à outra: a raiz de uma é o filho mais à esquerda da raiz da outra. A Figura 19.2(b) mostra as árvores binomiais de  $B_0$  até  $B_4$ .

366 | Algumas propriedades de árvores binomiais são dadas pelo lema a seguir.

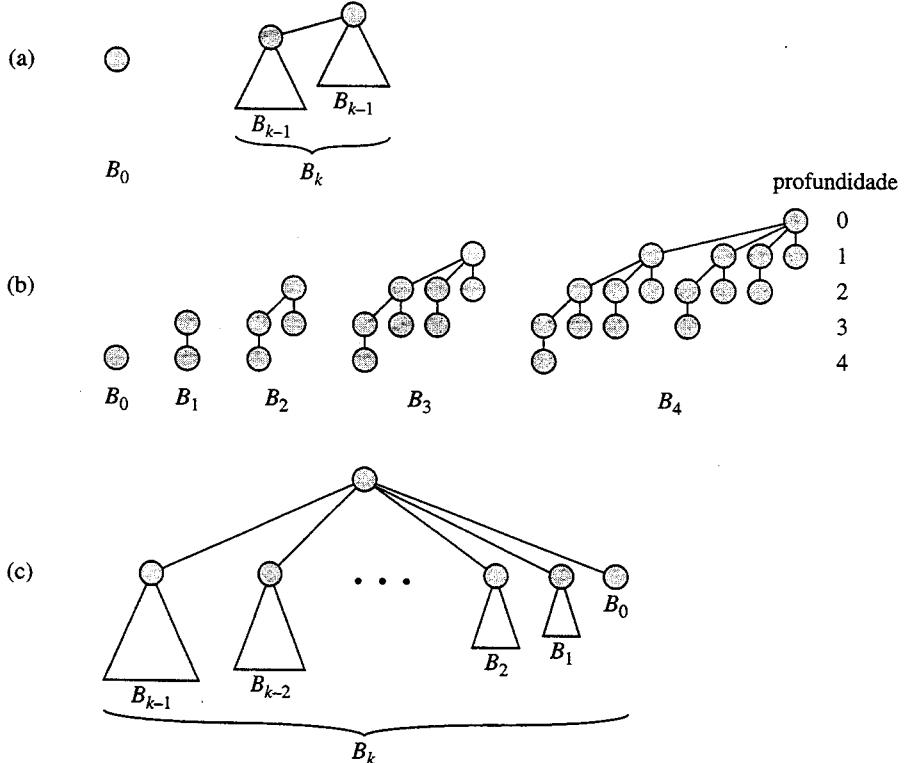


FIGURA 19.2 (a) A definição recursiva da árvore binomial  $B_k$ . Os triângulos representam subárvores enraizadas. (b) As árvores binomiais  $B_0$  até  $B_4$ . As profundidades dos nós em  $B_4$  são mostradas. (c) Outro modo de examinar a árvore binomial  $B_k$

### Lema 19.1 (Propriedades de árvores binomiais)

Para a árvore binomial  $B_k$ ,

1. existem  $2^k$  nós,
2. a altura da árvore é  $k$ ,
3. existem exatamente  $\binom{k}{i}$  nós na profundidade  $i$  para  $i = 0, 1, \dots, k$ , e
4. a raiz tem grau  $k$ , o qual é maior que o de qualquer outro nó; além disso, se os filhos da raiz são numerados da esquerda para a direita por  $k-1, k-2, \dots, 0$ , o filho  $i$  é a raiz de uma subárvore  $B_i$ .

**Prova** A prova é por indução sobre  $k$ . Para cada propriedade, a base é a árvore binomial  $B_0$ . A verificação de que cada propriedade se mantém válida para  $B_0$  é trivial.

Para a etapa indutiva, supomos que o lema se mantém válido para  $B_{k-1}$ .

1. A árvore binomial  $B_k$  consiste em duas cópias de  $B_{k-1}$ , e assim  $B_k$  tem  $2^{k-1} + 2^{k-1} = 2^k$  nós.
2. Devido ao modo como as duas cópias de  $B_{k-1}$  são ligadas para formar  $B_k$ , a profundidade máxima de um nó em  $B_k$  é uma unidade maior que a profundidade máxima em  $B_{k-1}$ . Pela hipótese indutiva, essa profundidade máxima é  $(k-1) + 1 = k$ .
3. Seja  $D(k, i)$  o número de nós na profundidade  $i$  da árvore binomial  $B_k$ . Tendo em vista que  $B_k$  é composta de duas cópias de  $B_{k-1}$  ligadas uma à outra, um nó na profundidade  $i$  em  $B_{k-1}$  aparece em  $B_k$  uma vez na profundidade  $i$  e uma vez na profundidade  $i+1$ . Em outras palavras, o número de nós na profundidade  $i$  em  $B_k$  é o número de nós na profundidade  $i$  em  $B_{k-1}$  mais o número de nós na profundidade  $i-1$  em  $B_{k-1}$ . Desse modo,

$$D(k, i) = D(k-1, i) + D(k-1, i-1) \text{ (pela hipótese indutiva)}$$

$$\begin{aligned} &= \binom{k-1}{i} + \binom{k-1}{i-1} \text{ (pelo Exercício C.1-7)} \\ &= \binom{k}{i}. \end{aligned}$$

4. O único nó com grau maior em  $B_k$  que em  $B_{k-1}$  é a raiz, que tem um filho a mais que a raiz de  $B_{k-1}$ . Como a raiz de  $B_{k-1}$  tem grau  $k-1$ , a raiz de  $B_k$  tem grau  $k$ . Agora, pela hipótese indutiva, e como mostra a Figura 19.2(c), da esquerda para a direita, os filhos da raiz de  $B_{k-1}$  são raízes de  $B_{k-2}, B_{k-3}, \dots, B_0$ . Conseqüentemente, quando  $B_{k-1}$  é ligado a  $B_{k-1}$ , os filhos da raiz resultante são raízes de  $B_{k-1}, B_{k-2}, \dots, B_0$ .

### **Corolário 19.2**

O grau máximo de qualquer nó em uma árvore binomial de  $n$  nós é  $\lg n$ .

**Prova** Imediata, a partir das propriedades 1 e 4 do Lema 19.1. ■

A expressão “árvore binomial” vem da propriedade 3 do Lema 19.1, pois os termos  $\binom{k}{i}$  são os coeficientes binomiais. O Exercício 19.1-3 nos fornece uma justificativa adicional para a expressão.

### **19.1.2 Heaps binomiais**

Um **heap binomial**  $H$  é um conjunto de árvores binomiais que satisfaz às **propriedades de heaps binomiais** a seguir.

1. Cada árvore binomial em  $H$  obedece à **propriedade de heap mínimo**: a chave de um nó é maior que ou igual à chave de seu pai. Dizemos que tal árvore é **ordenada como heap mínimo**.
2. Existe no máximo uma árvore binomial em  $H$  cuja raiz tem grau  $k$ , para qualquer inteiro não negativo  $k$ .

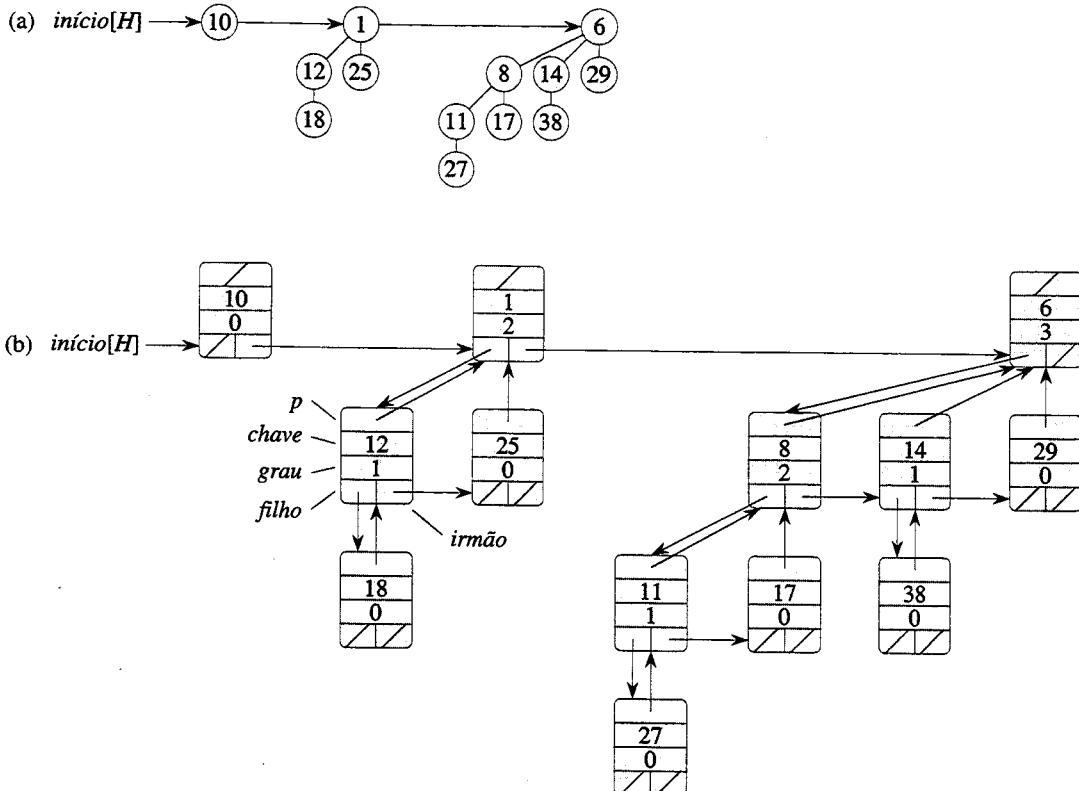
A primeira propriedade nos informa que a raiz de uma árvore ordenada como heap mínimo contém a menor chave na árvore.

A segunda propriedade implica que um heap binomial de  $n$  nós  $H$  consiste em no máximo  $\lfloor \lg n \rfloor + 1$  árvores binomiais. Para ver por que isso ocorre, observe que a representação binária de  $n$  tem  $\lfloor \lg n \rfloor + 1$  bits, digamos  $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \dots, b_0 \rangle$ , de forma que  $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$ . Então, pela propriedade 1 do Lema 19.1, a árvore binomial  $B_i$  aparece em  $H$  se e somente se o bit  $b_i = 1$ . Desse modo, o heap binomial  $H$  contém no máximo  $\lfloor \lg n \rfloor + 1$  árvores binomiais.

A Figura 19.3(a) mostra um heap binomial  $H$  com 13 nós. A representação binária de 13 é  $\langle 1101 \rangle$ , e  $H$  consiste nas árvores binomiais ordenadas como heaps mínimos  $B_3, B_2$  e  $B_0$ , que têm 8, 4 e 1 nós respectivamente, dando um total de 13 nós.

### **Representação de heaps binomiais**

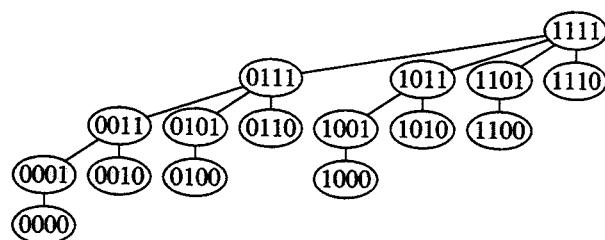
Como mostra a Figura 19.3(b), cada árvore binomial dentro de um heap binomial é armazenada na representação de filho da esquerda, irmão da direita da Seção 10.4. Cada nó tem um campo *chave* e quaisquer outras informações satélite exigidas pela aplicação. Além disso, cada nó  $x$  contém ponteiros  $p[x]$  para seu pai,  $filho[x]$  para seu filho mais à esquerda, e  $irmão[x]$  para o irmão de  $x$  imediatamente à sua direita. Se o nó  $x$  é uma raiz, então  $p[x] = \text{NIL}$ . Se o nó  $x$  não tem nenhum filho, então  $filho[x] = \text{NIL}$ , e se  $x$  é o filho mais à direita de seu pai, então  $irmão[x] = \text{NIL}$ . Cada nó  $x$  também contém o campo *grau[x]*, que é o número de filhos de  $x$ .



**FIGURA 19.3** Um heap binomial  $H$  com  $n = 13$  nós. (a) O heap consiste nas árvores binomiais  $B_0$ ,  $B_2$  e  $B_3$ , que têm 1, 4 e 8 nós respectivamente, totalizando  $n = 13$  nós. Tendo em vista que cada árvore binomial está ordenada como heap mínimo, a chave de qualquer nó não é menor que a chave de seu pai. Também é mostrada a lista de raízes, que é uma lista ligada de raízes na ordem de grau crescente. (b) Uma representação mais detalhada do heap binomial  $H$ . Cada árvore binomial é armazenada na representação de filho da esquerda, irmão da direita, e cada nó armazena seu grau

Conforme também mostra a Figura 19.3, as raízes das árvores binomiais dentro de um heap binomial estão organizadas em uma lista ligada, à qual nos referimos como a *lista de raízes*. Os graus das raízes aumentam estritamente à medida que atravessamos a lista de raízes. Pela segunda propriedade de heap binomial, em um heap binomial de  $n$  nós, os graus das raízes formam um subconjunto de  $\{0, 1, \dots, \lfloor \lg n \rfloor\}$ . O campo *irmão* tem um significado diferente para raízes em relação a não raízes. Se  $x$  é uma raiz, então  $irmão[x]$  aponta para a próxima raiz na lista de raízes. (Como de costume,  $irmão[x] = \text{NIL}$  se  $x$  é a última raiz na lista de raízes.)

O acesso a um dado heap binomial  $H$  é feito pelo campo *início*[ $H$ ], que é simplesmente um ponteiro para a primeira raiz na lista de raízes de  $H$ . Se o heap binomial  $H$  não tem nenhum elemento, então *início*[ $H$ ] = NIL.



**FIGURA 19.4** A árvore binomial  $B_4$  com nós identificados em binário por um percurso pós-ordem

## Exercícios

### 19.1-1

Suponha que  $x$  seja um nó em uma árvore binomial dentro de um heap binomial, e suponha que  $irmão[x] \neq \text{NIL}$ . Se  $x$  não é uma raiz, de que modo  $grau[irmão[x]]$  se compara a  $grau[x]$ ? E se  $x$  é uma raiz?

### 19.1-2

Se  $x$  é um nó não de raiz em uma árvore binomial dentro de um heap binomial, de que modo  $grau[p[x]]$  se compara a  $grau[x]$ ?

### 19.1-3

Suponha que identifiquemos os nós da árvore binomial  $B_k$  em binário por um percurso pós-ordem, como na Figura 19.4. Considere um nó  $x$  identificado como  $l$  na profundidade  $i$ , e seja  $j = k - i$ . Mostre que  $x$  tem  $j$  valores 1 em sua representação binária. Quantas cadeias binárias de  $k$  elementos existem que contêm exatamente  $j$  valores 1? Mostre que o grau de  $x$  é igual ao número de valores 1 à direita do 0 situado mais à direita na representação binária de  $l$ .

## 19.2 Operações sobre heaps binomiais

Nesta seção, mostramos como executar operações sobre heaps binomiais nos limites de tempo mostrados na Figura 19.1. Apresentaremos apenas os limites superiores; os limites inferiores ficam para o Exercício 19.2-10.

### Criação de um novo heap binomial

Para produzir um heap binomial vazio, o procedimento **MAKE-BINOMIAL-HEAP** simplesmente aloca e retorna um objeto  $H$ , onde  $início[H] = \text{NIL}$ . O tempo de execução é  $\Theta(1)$ .

### Como encontrar a chave mínima

O procedimento **BINOMIAL-HEAP-MINIMUM** retorna um ponteiro para o nó com a chave mínima em um heap binomial de  $n$  nós  $H$ . Essa implementação pressupõe que não existe nenhuma chave com o valor  $\infty$ . (Veja o Exercício 19.2-5.)

```
BINOMIAL-HEAP-MINIMUM( $H$ )
1  $y \leftarrow \text{NIL}$ 
2  $x \leftarrow início[H]$ 
3  $min \leftarrow \infty$ 
4 while  $x \neq \text{NIL}$ 
5   do if  $chave[x] < min$ 
6     then  $min \leftarrow chave[x]$ 
7      $y \leftarrow x$ 
8      $x \leftarrow irmão[x]$ 
9 return  $y$ 
```

Tendo em vista que um heap binomial é ordenado como heap mínimo, a chave mínima deve residir em um nó de raiz. O procedimento **BINOMIAL-HEAP-MINIMUM** verifica todas as raízes, cujo número é no máximo  $\lfloor \lg n \rfloor + 1$ , gravando o mínimo atual em  $min$  e um ponteiro para o mínimo atual em  $y$ . Ao ser chamado sobre o heap binomial da Figura 19.3, **BINOMIAL-HEAP-MINIMUM** retorna um ponteiro para o nó com chave 1.

Pelo fato de existirem no máximo  $\lfloor \lg n \rfloor + 1$  raízes para verificar, o tempo de execução de

## Como unir dois heaps binomiais

A operação de unir dois heaps binomiais é utilizada como uma sub-rotina pela maioria das operações restantes. O procedimento BINOMIAL-HEAP-UNION liga repetidamente entre si as árvores binomiais cujas raízes têm o mesmo grau. O procedimento a seguir liga a árvore  $B_{k-1}$  com raiz no nó  $y$  à árvore  $B_{k-1}$  com raiz no nó  $z$ ; isto é, ele faz de  $z$  o pai de  $y$ . Desse modo, o nó  $z$  se torna a raiz de uma árvore  $B_k$ .

### BINOMIAL-LINK( $y$ , $z$ )

- 1  $p[y] \leftarrow z$
  - 2  $irmão[y] \leftarrow filho[z]$
  - 3  $filho[z] \leftarrow y$
  - 4  $grau[z] \leftarrow grau[z] + 1$

O procedimento **BINOMIAL-LINK** torna o nó  $y$  o novo início da lista ligada de filhos do nó  $z$  no tempo  $O(1)$ . Ele funciona porque a representação de filho da esquerda, irmão da direita de cada árvore binomial corresponde à propriedade de ordenação da árvore: em uma árvore  $B_k$ , o filho mais à esquerda da raiz é a raiz de uma árvore  $B_{k-1}$ .

O procedimento a seguir une os heaps binomiais  $H_1$  e  $H_2$ , retornando o heap resultante. Ele destrói as representações de  $H_1$  e  $H_2$  no processo. Além de BINOMIAL-LINK, o procedimento emprega um procedimento auxiliar BINOMIAL-HEAP-MERGE que intercala as listas de raízes de  $H_1$  e  $H_2$  em uma única lista ligada que é ordenada por grau em ordem monotonicamente crescente. O procedimento BINOMIAL-HEAP-MERGE, cujo pseudocódigo deixamos para o Exercício 19.2-1, é semelhante ao procedimento MERGE da Seção 2.3.1.

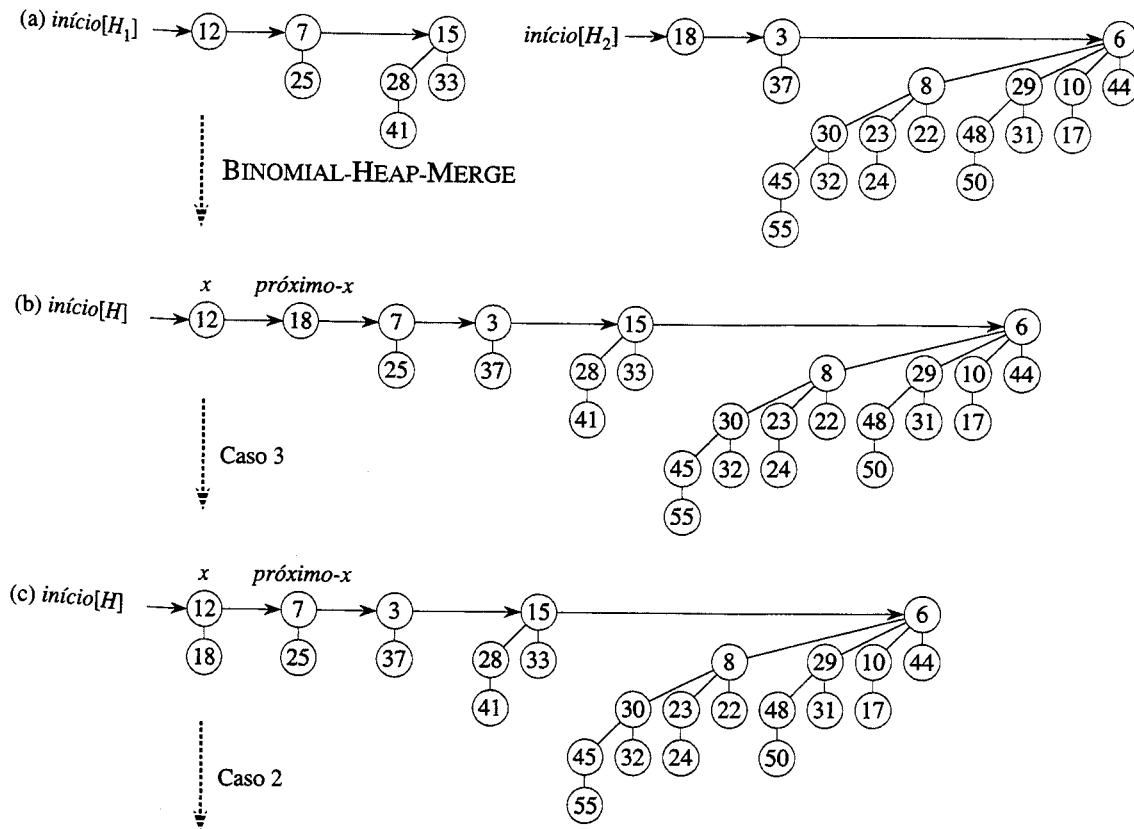
**BINOMIAL-HEAP-UNION( $H_1, H_2$ )**

- ```

1  $H \leftarrow \text{MAKE-BINOMIAL-HEAP}()$ 
2  $\text{início}[H] \leftarrow \text{BINOMIAL-HEAP-MERGE}(H_1, H_2)$ 
3 libera os objetos  $H_1$  e  $H_2$  mas não as listas para as quais eles apontam
4 if  $\text{início}[H] = \text{NIL}$ 
5 then return  $H$ 
6  $\text{anterior-}x \leftarrow \text{NIL}$ 
7  $x \leftarrow \text{início}[H]$ 
8  $\text{próximo-}x \leftarrow \text{irmão}[x]$ 
9 while  $\text{próximo-}x \neq \text{NIL}$ 
10 do if ( $\text{grau}[x] \neq \text{grau}[\text{próximo-}x]$ ) or
     ( $\text{irmão}[\text{próximo-}x] \neq \text{NIL}$  e  $\text{grau}[\text{irmão}[\text{próximo-}x]] = \text{grau}[x]$ )
    then  $\text{anterior-}x \leftarrow x$                                 ▷ Casos 1 e 2
          $x \leftarrow \text{próximo-}x$                           ▷ Casos 1 e 2
11 else if  $\text{chave}[x] \leq \text{chave}[\text{próximo-}x]$ 
12     then  $\text{irmão}[x] \leftarrow \text{irmão}[\text{próximo-}x]$           ▷ Caso 3
13         BINOMIAL-LINK( $\text{próximo-}x, x$ )
14     else if  $\text{anterior-}x = \text{NIL}$ 
15         then  $\text{início}[H] \leftarrow \text{próximo-}x$           ▷ Caso 3
16         else  $\text{irmão}[\text{anterior-}x] \leftarrow \text{próximo-}x$  ▷ Caso 4
17             BINOMIAL-LINK( $x, \text{próximo-}x$ )
18              $x \leftarrow \text{próximo-}x$                       ▷ Caso 4
19              $\text{próximo-}x \leftarrow \text{irmão}[x]$                 ▷ Caso 4
20
21 return  $H$ 

```

A Figura 19.5 mostra um exemplo de BINOMIAL-HEAP-UNION no qual ocorrem todos os quatro casos dados no pseudocódigo.

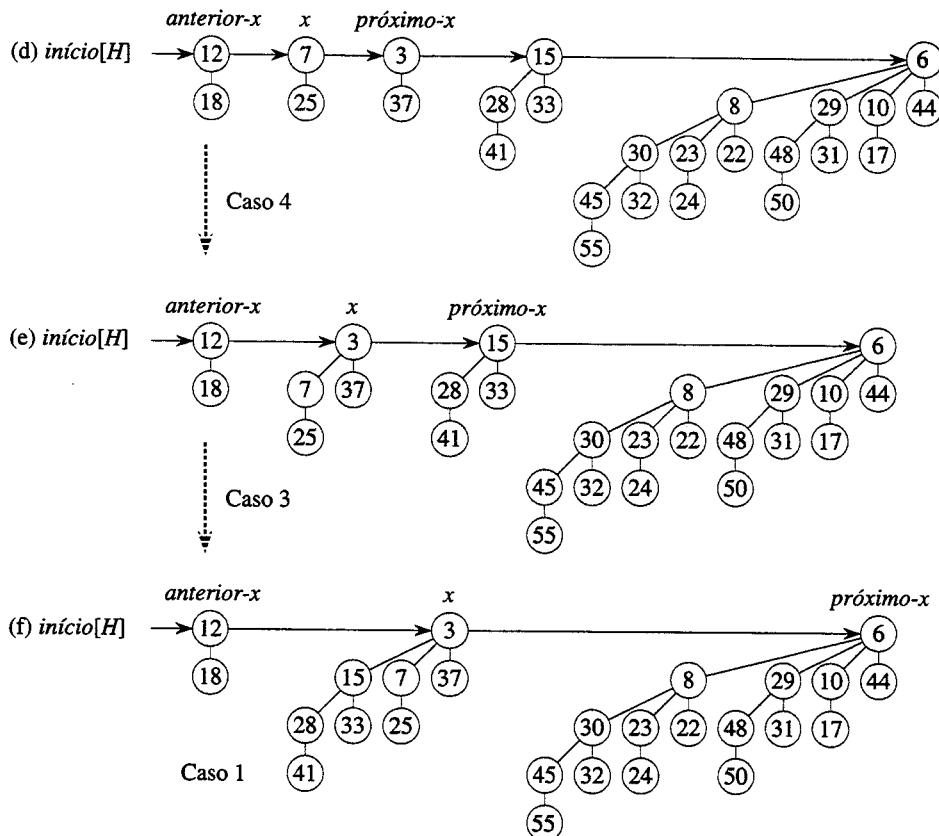


**FIGURA 19.5** A execução de BINOMIAL-HEAP-UNION. (a) Os heaps binomiais  $H_1$  e  $H_2$ . (b) O heap binomial  $H$  é a saída de BINOMIAL-HEAP-MERGE( $H_1, H_2$ ). Inicialmente,  $x$  é a primeira raiz na lista de raízes de  $H$ . Tendo em vista que tanto  $x$  quanto  $próximo-x$  têm grau 0 e  $chave[x] < chave[próximo-x]$ , aplica-se o caso 3. (c) Após a ligação,  $x$  é a primeira de três raízes com o mesmo grau, e assim o caso 2 se aplica. (d) Após todos os ponteiros se deslocarem uma posição para baixo na lista de raízes, aplica-se o caso 4, pois  $x$  é a primeira de duas raízes de mesmo grau. (e) Depois de ocorrer a ligação, aplica-se o caso 3. (f) Após outra ligação, o caso 1 se aplica, porque  $x$  tem grau 3 e  $próximo-x$  tem grau 4. Essa iteração do loop while é a última porque, depois que os ponteiros se deslocam uma posição para baixo na lista de raízes,  $próximo-x = NIL$ .

O procedimento BINOMIAL-HEAP-UNION tem duas fases. A primeira fase, executada pela chamada de BINOMIAL-HEAP-MERGE, intercala as listas de raízes de heaps binomiais  $H_1$  e  $H_2$  em uma única lista ligada  $H$  que é ordenada por grau em ordem monotonicamente crescente. Entretanto, podem existir até duas raízes (mas não mais) de cada grau; assim, a segunda fase liga raízes de mesmo grau até restar no máximo uma raiz de cada grau. Pelo fato da lista ligada  $H$  ser ordenada por grau, podemos executar todas as operações de ligação com rapidez.

Em detalhes, o procedimento funciona da maneira ilustrada a seguir. As linhas 1 a 3 começam intercalando as listas de raízes dos heaps binomiais  $H_1$  e  $H_2$  em uma única lista de raízes  $H$ . As listas de raízes de  $H_1$  e  $H_2$  são ordenadas por grau estritamente crescente, e BINOMIAL-HEAP-MERGE retorna uma lista de raízes  $H$  que é ordenada por grau monotonicamente crescente. Se as listas de raízes de  $H_1$  e  $H_2$  têm  $m$  raízes no total, BINOMIAL-HEAP-MERGE é executado no tempo  $O(m)$ , através do exame repetido das raízes no início das duas listas de raízes, anexando-se a raiz com o menor grau à lista de raízes de saída, removendo-a de sua lista de raízes de entrada no processo.

O procedimento BINOMIAL-HEAP-UNION inicializa em seguida alguns ponteiros na lista de raízes de  $H$ . Primeiro, ele simplesmente retorna nas linhas 4 e 5, se acontecer de estar unindo dois heaps binomiais vazios. Então, a partir da linha 6, sabemos que  $H$  tem pelo menos uma raiz. Por todo o procedimento, mantemos três ponteiros na lista de raízes:



- $x$  aponta para a raiz que está sendo examinada atualmente,
- $anterior-x$  aponta para a raiz que precede  $x$  na lista de raízes:  $irmão[anterior-x] = x$  (como inicialmente  $x$  não tem nenhum predecessor, começamos com  $anterior-x$  definido como NIL) e
- $próximo-x$  aponta para a raiz que segue  $x$  na lista de raízes:  $irmão[x] = próximo-x$ .

Inicialmente, existem no máximo duas raízes na lista de raízes  $H$  de um dado grau: como  $H_1$  e  $H_2$  eram heaps binomiais, cada um deles tinha no máximo uma raiz de um determinado grau. Além disso, BINOMIAL-HEAP-MERGE nos garante que, se duas raízes em  $H$  têm o mesmo grau, elas são adjacentes na lista de raízes.

De fato, durante a execução de BINOMIAL-HEAP-UNION, podem existir três raízes de um dado grau aparecendo na lista de raízes  $H$  em algum momento. Veremos em breve como essa situação poderia ocorrer. Então, em cada iteração do loop while das linhas 9 a 21, decidimos se iremos ligar  $x$  e  $próximo-x$  com base em seus graus e possivelmente no grau de  $irmão[próximo-x]$ . Um invariante do loop é que, toda vez que iniciamos o corpo do loop, tanto  $x$  quanto  $próximo-x$  são não NIL. (Veja no Exercício 19.2-4 um loop invariante preciso.)

O caso 1, mostrado na Figura 19.6(a), ocorre quando  $grau[x] \neq grau[próximo-x]$ , isto é, quando  $x$  é a raiz de uma árvore  $B_k$  e  $próximo-x$  é a raiz de uma árvore  $B_l$  para algum  $l > k$ . As linhas 11 e 12 tratam esse caso. Não ligamos  $x$  e  $próximo-x$ , e assim simplesmente movemos os ponteiros para uma posição abaixo na lista. A atualização de  $próximo-x$  de modo a apontar para o nó seguinte ao novo  $x$  é tratada na linha 21, que é comum a todos os casos.

O caso 2, mostrado na Figura 19.6(b), ocorre quando  $x$  é a primeira das três raízes de grau idêntico, ou seja, quando

$$grau[x] = grau[próximo-x] = grau[irmão[próximo-x]] .$$

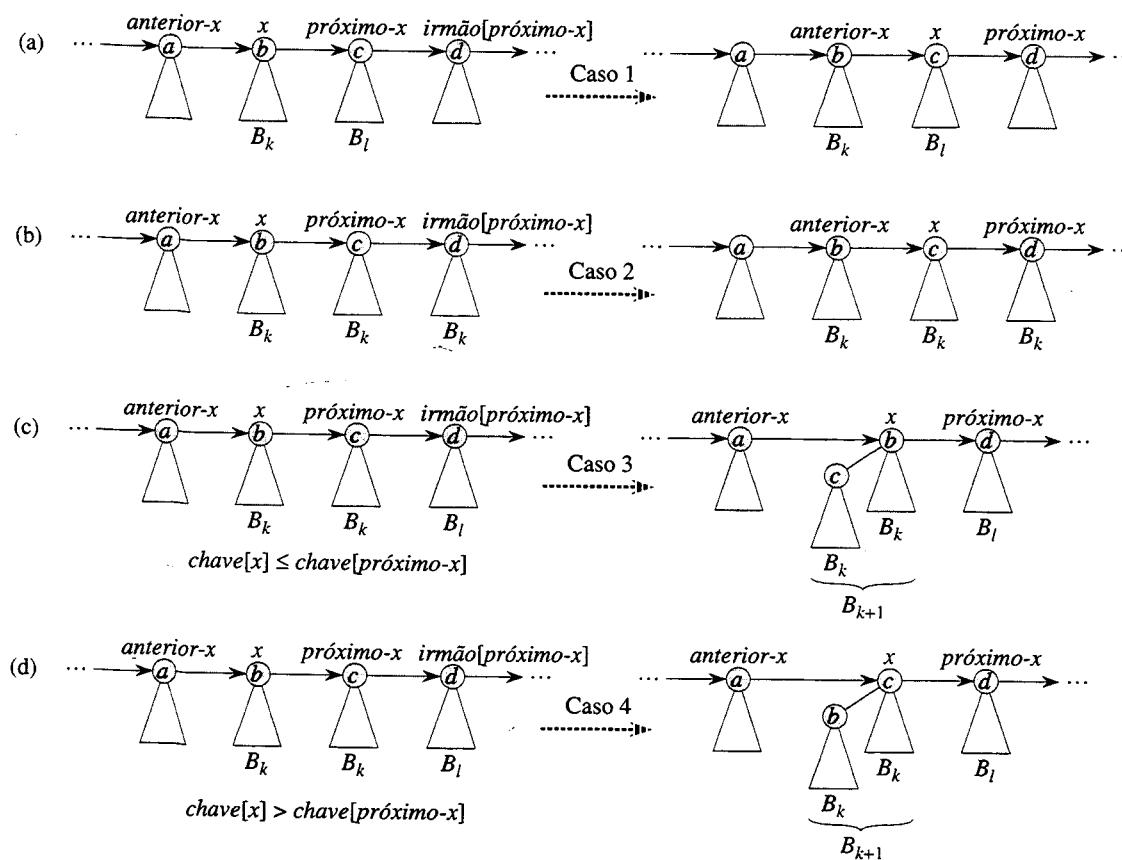
Tratamos esse caso da mesma maneira que o caso 1: simplesmente deslocamos os ponteiros para uma posição mais abaixo na lista. A próxima iteração executará o caso 3 ou 4 para combinar a segunda e a terceira das três raízes de grau idêntico. A linha 10 testa os casos 1 e 2, e as linhas 11 e 12 tratam ambos os casos.

Os casos 3 e 4 ocorrem quando  $x$  é a primeira das duas raízes de mesmo grau, ou seja, quando

$$grau[x] = grau[próximo-x] \neq grau[irmão[próximo-x]].$$

Esses casos podem ocorrer em qualquer iteração, mas um deles sempre ocorre imediatamente em seguida ao caso 2. Nos casos 3 e 4, ligamos  $x$  e  $próximo-x$ . Os dois casos se distinguem pelo fato de  $x$  ou  $próximo-x$  ter a menor chave, o que determina o nó que será a raiz após os dois serem ligados.

No caso 3, mostrado na Figura 19.6(c),  $chave[x] \leq chave[próximo-x]$ ; assim,  $próximo-x$  é ligado a  $x$ . A linha 14 remove  $próximo-x$  da lista de raízes, e a linha 15 torna  $próximo-x$  o filho mais à esquerda de  $x$ .



**FIGURA 19.6** Os quatro casos que ocorrem em BINOMIAL-HEAP-UNION. As etiquetas  $a$ ,  $b$ ,  $c$  e  $d$  servem apenas para identificar as raízes envolvidas; elas não indicam os graus ou as chaves dessas raízes. Em cada caso,  $x$  é a raiz de uma árvore  $B_k$  e  $l > k$ . (a) Caso 1:  $grau[x] \neq grau[próximo-x]$ . Os ponteiros se deslocam uma posição mais para baixo na lista de raízes. (b) Caso 2:  $grau[x] = grau[próximo-x] = grau[irmão[próximo-x]]$ . Novamente, os ponteiros se movem uma posição mais abaixo na lista, e a próxima iteração executa o caso 3 ou o caso 4. (c) Caso 3:  $grau[x] = grau[próximo-x] \neq grau[irmão[próximo-x]]$  e  $chave[x] \leq chave[próximo-x]$ . Removemos  $próximo-x$  da lista de raízes e a ligamos a  $x$ , criando uma árvore  $B_{k+1}$ . (d) Caso 4:  $grau[x] = grau[próximo-x] \neq grau[irmão[próximo-x]]$  e  $chave[x] > chave[próximo-x]$ . Removemos  $x$  da lista de raízes e a ligamos a  $próximo-x$ , criando mais uma vez uma árvore  $B_{k+1}$ .

No caso 4, mostrado na Figura 19.6(d), *próximo-x* tem a menor chave, e assim  $x$  é ligado a *próximo-x*. As linhas 16 a 18 removem  $x$  da lista de raízes, a qual tem dois casos dependendo de  $x$  ser a primeira raiz na lista (linha 17) ou não (linha 18). Então, a linha 19 faz de  $x$  o filho mais à esquerda de *próximo-x*, e a linha 20 atualiza  $x$  para a próxima iteração.

Em seguida ao caso 3 ou ao caso 4, a configuração para a próxima iteração do loop **while** é a mesma. Acabamos de ligar duas árvores  $B_k$  para formar uma árvore  $B_{k+1}$ , para a qual  $x$  aponta agora. Já existiam zero, uma ou duas outras árvores  $B_{k+1}$  na lista de raízes resultante de BINOMIAL-HEAP-MERGE, e assim  $x$  é agora a primeira de uma, duas ou três árvores  $B_{k+1}$  na lista de raízes. Se  $x$  é a única raiz, então entramos no caso 1 na próxima iteração:  $grau[x] \neq grau[próximo-x]$ . Se  $x$  é a primeira de duas, então entramos no caso 3 ou no caso 4 na próxima iteração. É quando  $x$  é a primeira de três raízes que entramos no caso 2 na próxima iteração.

O tempo de execução de BINOMIAL-HEAP-UNION é  $O(\lg n)$ , onde  $n$  é o número total de nós nos heaps binomiais  $H_1$  e  $H_2$ . Podemos ver isso como a seguir. Sejam os heaps  $H_1$  contendo  $n_1$  nós e  $H_2$  contendo  $n_2$  nós, de modo que  $n = n_1 + n_2$ . Então,  $H_1$  contém no máximo  $\lfloor \lg n_1 \rfloor + 1$  raízes e  $H_2$  contém no máximo  $\lfloor \lg n_2 \rfloor + 1$  raízes; assim,  $H$  contém no máximo  $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leq 2\lfloor \lg n \rfloor + 2 = O(\lg n)$  raízes imediatamente após a chamada de BINOMIAL-HEAP-MERGE. O tempo para executar BINOMIAL-HEAP-MERGE é portanto  $O(\lg n)$ . Cada iteração do loop **while** demora o tempo  $O(1)$ , e há no máximo  $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$  iterações, porque cada iteração avança os ponteiros uma posição para baixo na lista de raízes de  $H$  ou remove uma raiz da lista de raízes. Desse modo, o tempo total é  $O(\lg n)$ .

## Como inserir um nó

O procedimento a seguir insere o nó  $x$  no heap binomial  $H$ , supondo-se que o nó  $x$  já tenha sido alocado e que  $chave[x]$  já tenha sido preenchida.

```
BINOMIAL-HEAP-INSERT( $H, x$ )
1  $H' \leftarrow \text{MAKE-BINOMIAL-HEAP}()$ 
2  $p[x] \leftarrow \text{NIL}$ 
3  $filho[x] \leftarrow \text{NIL}$ 
4  $irmão[x] \leftarrow \text{NIL}$ 
5  $grau[x] \leftarrow 0$ 
6  $início[H'] \leftarrow x$ 
7  $H \leftarrow \text{BINOMIAL-HEAP-UNION}(H, H')$ 
```

O procedimento simplesmente cria um heap binomial de um nó  $H'$  no tempo  $O(1)$  e o une com o heap binomial de  $n$  nós  $H$  no tempo  $O(\lg n)$ . A chamada a BINOMIAL-HEAP-UNION cuida de liberar o heap binomial temporário  $H'$ . (Uma implementação direta que não chama BINOMIAL-HEAP-UNION é dada como o Exercício 19.2-8.)

## Como extraír o nó com chave mínima

O procedimento a seguir extraí o nó com a chave mínima do heap binomial  $H$  e retorna um ponteiro para o nó extraído.

```
BINOMIAL-HEAP-EXTRACT-MIN( $H$ )
1 encontrar a raiz  $x$  com a chave mínima na lista de raízes de  $H$ 
   e remover  $x$  da lista de raízes de  $H$ 
2  $H' \leftarrow \text{MAKE-BINOMIAL-HEAP}()$ 
3 inverter a ordem da lista ligada de filhos de  $x$ 
   e definir  $início[H']$  de modo a apontar para o início da lista resultante
4  $H \leftarrow \text{BINOMIAL-HEAP-UNION}(H, H')$ 
5 return  $x$ 
```

Esse procedimento funciona como ilustra a Figura 19.7. O heap binomial de entrada  $H$  é mostrado na Figura 19.7(a). A Figura 19.7(b) mostra a situação após a linha 1: a raiz  $x$  com a chave mínima foi removida da lista de raízes de  $H$ . Se  $x$  é a raiz de uma árvore  $B_k$ , então, pela propriedade 4 do Lema 19.1, os filhos de  $x$ , da esquerda para a direita, são raízes de árvores  $B_{k-1}, B_{k-2}, \dots, B_0$ . A Figura 19.7(c) mostra que, invertendo a lista de filhos de  $x$  na linha 3, temos um heap binomial  $H'$  que contém todo nó na árvore de  $x$ , exceto o próprio  $x$ . Pelo fato de a árvore de  $x$  ser removida de  $H$  na linha 1, o heap binomial que resulta da união de  $H$  e  $H'$  na linha 4, mostrado na Figura 19.7(d), contém todos os nós originalmente em  $H$ , com exceção de  $x$ . Finalmente, a linha 5 retorna  $x$ .

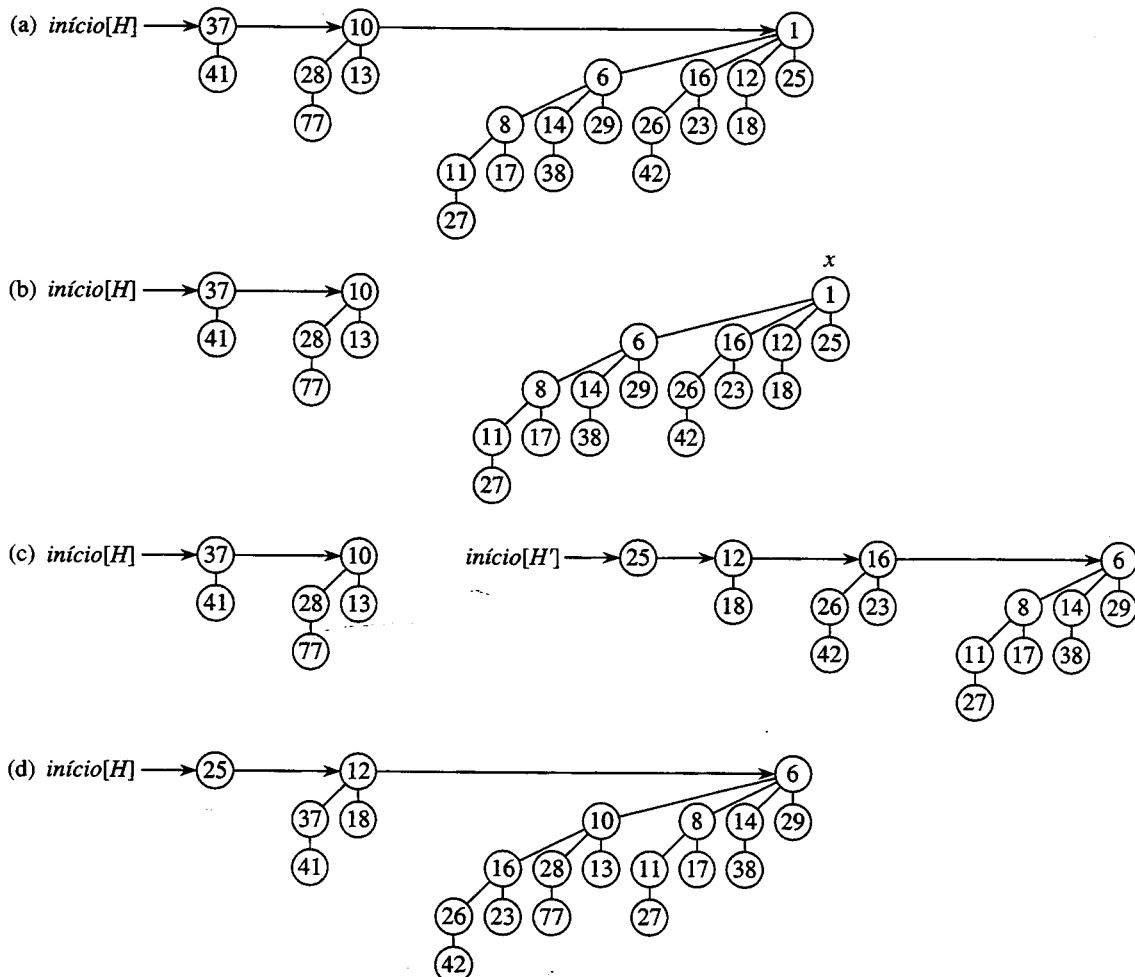


FIGURA 19.7 A ação de BINOMIAL-HEAP-EXTRACT-MIN. (a) Um heap binomial  $H$ . (b) A raiz  $x$  com chave mínima é removida da lista de raízes  $H$ . (c) A lista ligada de filhos de  $x$  é invertida, fornecendo outro heap binomial  $H'$ . (d) O resultado da união de  $H$  e  $H'$

Tendo em vista que cada uma das linhas 1 a 4 demora o tempo  $O(\lg n)$  se  $H$  tem  $n$  nós, BINOMIAL-HEAP-EXTRACT-MIN é executado no tempo  $O(\lg n)$ .

### Como diminuir uma chave

O procedimento a seguir diminui a chave de um nó  $x$  em um heap binomial  $H$  até um novo valor  $k$ . Ele assinala um erro se  $k$  é maior que a chave atual de  $x$ .

```

BINOMIAL-HEAP-DECREASE-KEY( $H$ ,  $x$ ,  $k$ )
1 if  $k > chave[x]$ 
2   then error "nova chave é maior que chave atual"
3    $chave[x] \leftarrow k$ 
4    $y \leftarrow x$ 
5    $z \leftarrow p[y]$ 
6   while  $z \neq \text{NIL}$  e  $chave[y] < chave[z]$ 
7     do trocar  $chave[y] \leftrightarrow chave[z]$ 
8        $\triangleright$  Se  $y$  e  $z$  têm campos satélite, trocá-los também.
9      $y \leftarrow z$ 
10     $z \leftarrow p[y]$ 

```

Como mostra a Figura 19.8, esse procedimento diminui uma chave da mesma maneira que em um heap binário: “borbulhando” a chave no heap. Depois de assegurar que a nova chave de fato não é maior que a chave atual, e então atribuindo a nova chave a  $x$ , o procedimento sobe na árvore, com  $y$  apontando inicialmente para o nó  $x$ . Em cada iteração do loop while das linhas 6 a 10,  $chave[y]$  é comparada à chave do pai  $z$  de  $y$ . Se  $y$  é a raiz ou  $chave[y] = chave[z]$ , a árvore binomial é agora ordenada como heap mínimo. Caso contrário, o nó  $y$  viola a ordenação de heap mínimo, e assim sua chave é trocada com a chave de seu pai  $z$ , juntamente com quaisquer outras informações satélite. Então, o procedimento define  $y$  como  $z$ , subindo um nível na árvore, e continua com a próxima iteração.

O procedimento BINOMIAL-HEAP-DECREASE-KEY demora o tempo  $O(\lg n)$ . Pela propriedade 2 do Lema 19.1, a profundidade máxima de  $x$  é  $\lfloor \lg n \rfloor$ , e assim o loop while das linhas 6 a 10 itera-se no máximo  $\lfloor \lg n \rfloor$  vezes.

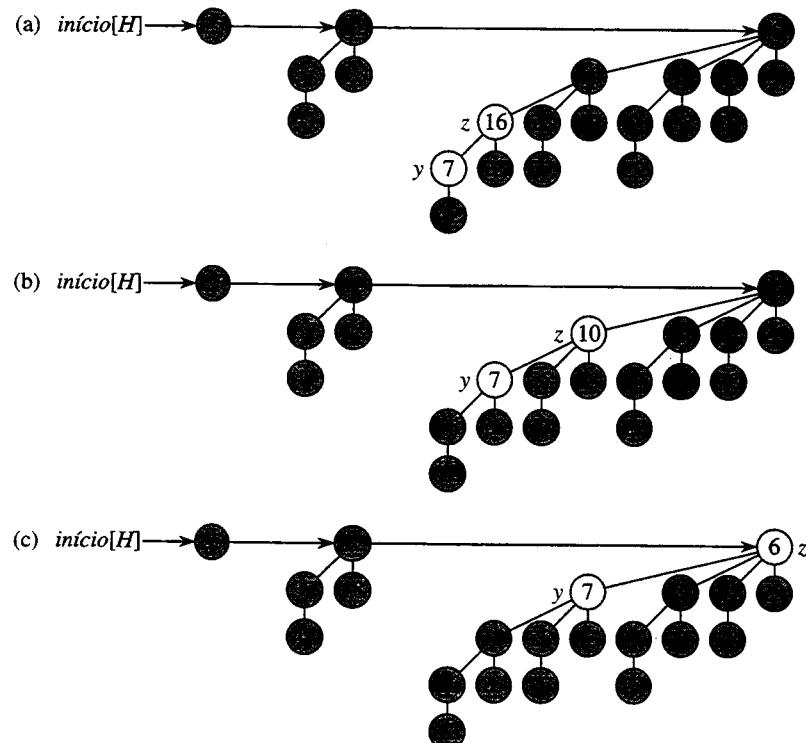


FIGURA 19.8 A ação de BINOMIAL-HEAP-DECREASE-KEY. (a) A situação imediatamente antes da linha 6 da primeira iteração do loop while. O nó  $y$  teve sua chave diminuída para 7, que é menor que a chave do pai  $z$  de  $y$ . (b) As chaves dos dois nós são trocadas, e a situação imediatamente antes da linha 6 da segunda iteração é mostrada. Os ponteiros  $y$  e  $z$  subiram um nível na árvore, mas a ordem de heap mínimo continua a ser violada. (c) Depois de outra troca e de mover os ponteiros  $y$  e  $z$  para cima mais um nível, finalmente descobrimos que a ordem de heap mínimo é satisfeita, e assim o loop while termina

## Como eliminar uma chave

É fácil eliminar a chave de um nó  $x$  e as informações satélite do heap binomial  $H$  no tempo  $O(\lg n)$ . A implementação a seguir pressupõe que nenhum nó atualmente no heap binomial tem uma chave  $-\infty$ .

**BINOMIAL-HEAP-DELETE( $H, x$ )**

- 1 **BINOMIAL-HEAP-DECREASE-KEY( $H, x, -\infty$ )**
- 2 **BINOMIAL-HEAP-EXTRACT-MIN( $H$ )**

O procedimento **BINOMIAL-HEAP-DELETE** faz o nó  $x$  ter a única chave mínima no heap binomial inteiro, fornecendo a ele uma chave  $-\infty$ . (O Exercício 19.2-6 lida com a situação na qual  $-\infty$  não pode aparecer como chave, nem mesmo temporariamente.) Então, ele borbulha essa chave e as informações satélite associadas para cima até uma raiz, chamando **BINOMIAL-HEAP-DECREASE-KEY**. Essa raiz é então removida de  $H$  por uma chamada de **BINOMIAL-HEAP-EXTRACT-MIN**.

O procedimento **BINOMIAL-HEAP-DELETE** demora o tempo  $O(\lg n)$ .

## Exercícios

### 19.2-1

Escreva pseudocódigo para **BINOMIAL-HEAP-MERGE**.

### 19.2-2

Mostre o heap binomial que resulta quando um nó com chave 24 é inserido no heap binomial mostrado na Figura 19.7(d).

### 19.2-3

Mostre o heap binomial que resulta quando o nó com chave 28 é eliminado do heap binomial mostrado na Figura 19.8(c).

### 19.2-4

Demonstre a correção de **BINOMIAL-HEAP-UNION**, usando o seguinte loop invariante:

No início de cada iteração do loop **while** das linhas 9 a 21,  $x$  aponta para uma raiz que é uma das seguintes:

- A única raiz de seu grau.
- A primeira das duas únicas raízes de seu grau.
- A primeira ou a segunda das três únicas raízes de seu grau.

Além disso, todas as raízes que precedem o predecessor de  $x$  na lista de raízes têm graus exclusivos na lista de raízes e, se o predecessor de  $x$  tem um grau diferente do grau de  $x$ , seu grau na lista de raízes também é exclusivo. Finalmente, os graus de nós crescem monotonicamente à medida que percorremos a lista de raízes.

### 19.2-5

Explique por que o procedimento **BINOMIAL-HEAP-MINIMUM** não poderia funcionar corretamente se as chaves pudessem ter o valor  $\infty$ . Reescreva o pseudocódigo para fazê-lo funcionar corretamente em tais casos.

### 19.2-6

Suponha que não exista nenhum modo de representar a chave  $-\infty$ . Reescreva o procedimento **BINOMIAL-HEAP-DELETE** para funcionar corretamente nessa situação. Ele ainda deve demorar o tempo  $O(\lg n)$ .

### 19.2-7

Descreva o relacionamento entre a inserção em um heap binomial e a ação de incrementar um número binário, e também o relacionamento entre unir dois heaps binomiais e somar dois números binários.

### 19.2-8

Com base no Exercício 19.2-7, reescreva BINOMIAL-HEAP-INSERT para inserir um nó diretamente em um heap binomial sem chamar BINOMIAL-HEAP-UNION.

### 19.2-9

Mostre que, se as listas de raízes forem mantidas em ordem estritamente decrescente por grau (em vez de ficarem em ordem estritamente crescente), cada uma das operações de heaps binomiais poderá ser implementada sem alterar seu tempo de execução assintótico.

### 19.2-10

Encontre entradas que façam BINOMIAL-HEAP-EXTRACT-MIN, BINOMIAL-HEAP-DECREASE-KEY e BINOMIAL-HEAP-DELETE serem executados no tempo  $\Omega(\lg n)$ . Explique por que os tempos de execução do pior caso de BINOMIAL-HEAP-INSERT, BINOMIAL-HEAP-MINIMUM e BINOMIAL-HEAP-UNION são  $\Omega(\lg n)$ , mas não  $\Omega(n)$ . (Ver Problema 3-5.)

## Problemas

### 19-1 Heaps 2-3-4

O Capítulo 18 introduziu a árvore 2-3-4, na qual todo nó interno (possivelmente, com exceção da raiz) tem dois, três ou quatro filhos, e todas as folhas têm a mesma profundidade. Neste problema, implementaremos heaps 2-3-4, que admitem as operações de heaps intercaláveis.

Os heaps 2-3-4 diferem de árvores 2-3-4 nos seguintes aspectos. Nos heaps 2-3-4, apenas as folhas armazenam chaves, e cada folha  $x$  armazena exatamente uma chave no campo  $chave[x]$ . Não existe nenhuma ordenação particular das chaves nas folhas; isto é, da esquerda para a direita, as chaves podem estar em qualquer ordem. Cada nó interno  $x$  contém um valor  $pequeno[x]$  que é igual à menor chave armazenada em qualquer folha na subárvore com raiz em  $x$ . A raiz  $r$  contém um campo  $altura[r]$  que é a altura da árvore. Finalmente, os heaps 2-3-4 foram planejados para serem mantidos na memória principal, e assim não são necessárias leituras e gravações em disco.

Implemente as seguintes operações de heaps 2-3-4. Cada uma das operações nas partes (a)–(e) deve ser executada no tempo  $O(\lg n)$  sobre um heap 2-3-4 com  $n$  elementos. A operação UNION na parte (f) deve ser executada no tempo  $O(\lg n)$ , onde  $n$  é o número de elementos nos dois heaps de entrada.

- a. MINIMUM, que retorna um ponteiro para a folha com a menor chave.
- b. DECREASE-KEY, que diminui a chave de uma dada folha  $x$  para um dado valor  $k \leq chave[x]$ .
- c. INSERT, que insere a folha  $x$  com a chave  $k$ .
- d. DELETE, que elimina uma dada folha  $x$ .
- e. EXTRACT-MIN, que extrai a folha com a menor chave.
- f. UNION, que une dois heaps 2-3-4, retornando um único heap 2-3-4 e destruindo os heaps de entrada.

### 19-2 Algoritmo de árvore espalhada mínima com o uso de heaps binomiais

O Capítulo 23 apresenta dois algoritmos para resolver o problema de encontrar uma árvore espalhada mínima de um grafo não orientado. Aqui, veremos como os heaps binomiais podem ser usados para criar um algoritmo de árvore espalhada mínima diferente.

Temos um grafo não orientado conectado  $G = (V, E)$  com uma função peso  $w: E \rightarrow \mathbf{R}$ . Chamamos  $w(u, v)$  o peso da aresta  $(u, v)$ . Desejamos encontrar uma árvore espalhada mínima para  $G$ : um subconjunto acíclico  $T \subseteq E$  que conecte todos os vértices em  $V$  e cujo peso total

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

seja minimizado.

O pseudocódigo a seguir, que podemos provar ser correto usando técnicas da Seção 23.1, constrói uma árvore espalhada mínima  $T$ . Ele mantém uma partição  $\{V_i\}$  dos vértices de  $V$ , com cada conjunto  $V_i$ , um conjunto

$$E_i \subseteq \{(u, v) : u \in V_i \text{ ou } v \in V_i\}$$

de arestas incidentes sobre vértices em  $V_i$ .

```

MST( $G$ )
1  $T \leftarrow \emptyset$ 
2 for cada vértice  $v_i \in V[G]$ 
3   do  $V_i \leftarrow \{v_i\}$ 
4    $E_i \leftarrow \{(v_i, v) \in E[G]\}$ 
5 while existir mais de um conjunto  $V_i$ 
6   do escolher qualquer conjunto  $V_i$ 
7     extrair a aresta de peso mínimo  $(u, v)$  de  $E_i$ 
8     supor sem perda de generalidade que  $u \in V_i$  e  $v \in V_j$ 
9     if  $i \neq j$ 
10      then  $T \leftarrow T \cup \{(u, v)\}$ 
11       $V_i \leftarrow V_i \cup V_j$ , destruindo  $V_j$ 
12       $E_i \leftarrow E_i \cup E_j$ 
```

Descreva como implementar esse algoritmo usando heaps binomiais para gerenciar os conjuntos de vértices e arestas. Você precisa mudar a representação de um heap binomial? Você precisa adicionar operações além das operações de heaps intercaláveis dadas na Figura 19.1? Forneça o tempo de execução da sua implementação.

## Notas do capítulo

Os heaps binomiais foram apresentados em 1978 por Vuillemin [307]. Brown [49, 50] estudou suas propriedades em detalhes.

---

## *Capítulo 20*

### *Heaps de Fibonacci*

No Capítulo 19, vimos como os heaps binomiais admitem no tempo de pior caso  $O(\lg n)$  as operações de heaps intercaláveis INSERT, MINIMUM, EXTRACT-MIN e UNION, além das operações DECREASE-KEY e DELETE. Neste capítulo, examinaremos os heaps de Fibonacci, que admitem as mesmas operações mas têm a vantagem de que as operações que não envolvem a eliminação de um elemento são executadas no tempo amortizado  $O(1)$ .

Do um ponto de vista teórico, os heaps de Fibonacci são especialmente desejáveis quando o número de operações EXTRACT-MIN e DELETE é pequeno em relação ao número de outras operações executadas. Essa situação surge em muitas aplicações. Por exemplo, alguns algoritmos para problemas de grafos podem chamar DECREASE-KEY uma vez por aresta. Para grafos densos, que têm muitas arestas, o tempo amortizado  $O(1)$  de cada chamada de DECREASE-KEY adiciona uma grande melhoria sobre o tempo do pior caso  $\Theta(\lg n)$  de heaps binários ou binomiais. Algoritmos rápidos para problemas como o cálculo de árvores espalhadas mínimas (Capítulo 23) e a localização de caminhos mais curtos de origem única (Capítulo 24) tornam essencial o uso de heaps de Fibonacci.

Porém, do ponto de vista prático, os fatores constantes e a complexidade de programação de heaps de Fibonacci os tornam menos desejáveis que heaps binários (ou  $k$ -ários) comuns para a maioria das aplicações. Desse modo, os heaps de Fibonacci são predominantemente de interesse teórico. Se uma estrutura de dados muito mais simples com os mesmos limites de tempo amortizado que os heaps de Fibonacci fosse desenvolvida, ela também seria de grande utilidade prática.

Como um heap binomial, um heap de Fibonacci é uma coleção de árvores. De fato, os heaps de Fibonacci se baseiam livremente em heaps binomiais. Se nem DECREASE-KEY nem DELETE for invocada sobre um heap de Fibonacci, cada árvore no heap será semelhante a uma árvore binomial. Contudo, os heaps de Fibonacci diferem de heaps binomiais pelo fato de terem uma estrutura mais relaxada, permitindo limites de tempo assintótico melhores. O trabalho que mantém a estrutura pode ser retardado até sua execução se tornar conveniente.

Como as tabelas dinâmicas da Seção 17.4, os heaps de Fibonacci oferecem um bom exemplo de uma estrutura de dados projetada tendo a análise amortizada em mente. A intuição e as análises de operações de heaps de Fibonacci no restante deste capítulo dependem fortemente do método potencial da Seção 17.3.

A exposição neste capítulo pressupõe que você leu o Capítulo 19 sobre heaps binomiais. As especificações para as operações aparecem naquele capítulo, bem como a tabela da Figura 19.1, que resume os limites de tempo para operações sobre heaps binários, heaps binomiais e heaps

de Fibonacci. Nossa apresentação da estrutura de heaps de Fibonacci se baseia na estrutura de heaps binomiais, e algumas das operações executadas sobre heaps de Fibonacci são semelhantes às que são executados sobre heaps binomiais.

Como os heaps binomiais, os heaps de Fibonacci não são projetados para dar suporte eficiente à operação SEARCH; portanto, operações que se referem a um dado nó exigem um ponteiro para esse nó como parte de sua entrada. Quando usarmos um heap de Fibonacci em uma aplicação, freqüentemente armazenamos um descritor para o objeto da aplicação correspondente em cada elemento do heap de Fibonacci, bem como um descritor para o elemento do heap de Fibonacci correspondente em cada objeto da aplicação.

A Seção 20.1 define heaps de Fibonacci, discute sua representação e apresenta a função potencial usada para sua análise amortizada. A Seção 20.2 mostra como implementar as operações de heaps intercaláveis e como obter os limites de tempo amortizado mostrados na Figura 19.1. As duas operações restantes, DECREASE-KEY e DELETE, são apresentados na Seção 20.3. Finalmente, a Seção 20.4 conclui uma parte fundamental da análise e também explica o curioso nome da estrutura de dados.

## 20.1 Estrutura de heaps de Fibonacci

Como um heap binomial, um **heap de Fibonacci** é uma coleção de árvores ordenadas como heaps mínimos. Porém, as árvores em um heap de Fibonacci não estão restritas a serem árvores binomiais. A Figura 20.1(a) mostra um exemplo de um heap de Fibonacci.

Diferentes das árvores dentro de heaps binomiais, que são ordenadas, as árvores dentro de heaps de Fibonacci são enraizadas, mas não ordenadas. Como mostra a Figura 20.1(b), cada nó  $x$  contém um ponteiro  $p[x]$  para seu pai e um ponteiro  $\text{filho}[x]$  para qualquer um de seus filhos. Os filhos de  $x$  estão reunidos em uma lista circular, duplamente ligada, que chamamos **lista de filhos** de  $x$ . Cada filho  $y$  em uma lista de filhos tem ponteiros  $\text{esquerdo}[y]$  e  $\text{direito}[y]$  que apontam para os irmãos esquerdo e direito de  $y$ , respectivamente. Se o nó  $y$  é um filho único, então  $\text{esquerdo}[y] = \text{direito}[y] = y$ . A ordem na qual os irmãos aparecem em uma lista de filhos é arbitrária.

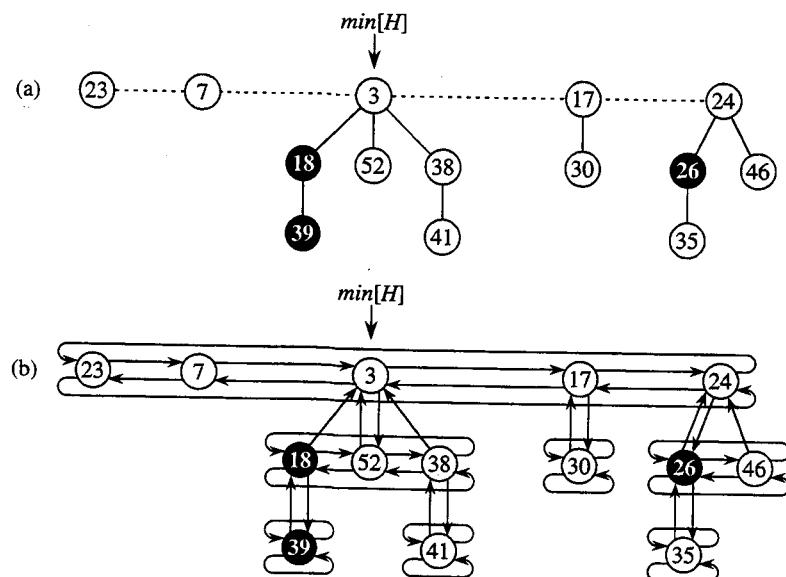


FIGURA 20.1 (a) Um heap de Fibonacci que consiste em cinco árvores ordenadas como heaps mínimos e 14 nós. A linha tracejada indica a lista de raízes. O nó mínimo do heap é o nó que contém a chave 3. Os três nós marcados estão escurecidos. O potencial desse heap de Fibonacci em particular é  $5 + 2 \cdot 3 = 11$ . (b) Uma representação mais completa, mostrando ponteiros  $p$  (setas para cima),  $\text{filho}$  (setas para baixo),  $\text{esquerdo}$  e  $\text{direito}$  (setas laterais). Esses detalhes são omitidos nas figuras restantes deste capítulo, pois todas as informações mostradas aqui podem ser determinadas a partir do que aparece na parte (a)

As listas circulares, duplamente ligadas (ver Seção 10.2), tem suas vantagens para uso em heaps de Fibonacci. Primeiro, podemos remover um nó de uma lista circular, duplamente ligada, no tempo  $O(1)$ . Em segundo lugar, dadas duas dessas listas, podemos concatená-las (ou “entre-laçá-las”) em uma lista circular, duplamente ligada, no tempo  $O(1)$ . Nas descrições de operações de heaps de Fibonacci, faremos referência a essas operações informalmente, deixando o leitor preencher os detalhes de suas implementações.

Dois outros campos em cada nó serão úteis. O número de filhos na lista de filhos do nó  $x$  é armazenado em  $grau[x]$ . O campo de valor booleano  $marca[x]$  indica se o nó  $x$  perdeu um filho desde a última vez que  $x$  se tornou filho de outro nó. Os nós recentemente criados estão desmarcados, e um nó  $x$  se torna desmarcado sempre que passa a ser o filho de outro nó. Até examinarmos a operação DECREASE-KEY na Seção 20.3, simplesmente definiremos todos os campos  $marca$  como FALSE.

É possível obter acesso a um dado heap de Fibonacci  $H$  por um ponteiro  $min[H]$  para a raiz da árvore que contém uma chave mínima; esse nó é chamado **nó mínimo** do heap de Fibonacci. Se um heap de Fibonacci  $H$  está vazio, então  $min[H] = NIL$ .

As raízes de todas as árvores em um heap de Fibonacci são ligadas usando-se seus ponteiros *esquerdo* e *direito* em uma lista circular duplamente ligada chamada **lista de raízes** do heap de Fibonacci. Desse modo, o ponteiro  $min[H]$  aponta para o nó na lista de raízes cuja chave é mínima. A ordem das árvores dentro de uma lista de raízes é arbitrária.

Contamos com um outro atributo para um heap de Fibonacci  $H$ : o número de nós atualmente em  $H$  é mantido em  $n[H]$ .

## Função potencial

Como mencionamos, usaremos o método potencial da Seção 17.3 para analisar o desempenho de operações de heap de Fibonacci. Para um dado heap de Fibonacci  $H$ , indicamos por  $t(H)$  o número de árvores na lista de raízes de  $H$ , e por  $m(H)$  o número de nós marcados em  $H$ . O potencial do heap de Fibonacci  $H$  é então definido por

$$\Phi(H) = t(H) = 2m(H). \quad (20.1)$$

(Ganharemos alguma intuição sobre essa função potencial na Seção 20.3.) Por exemplo, o potencial do heap de Fibonacci mostrado na Figura 20.1 é  $5 + 2 \cdot 3 = 11$ . O potencial de um conjunto de heaps de Fibonacci é a soma dos potenciais de seus heaps de Fibonacci constituintes. Vamos supor que uma unidade de potencial possa compensar uma quantidade constante de trabalho, onde a constante é suficientemente grande para cobrir o custo de qualquer das peças de trabalho específicas de tempo constante que poderíamos encontrar.

Supomos que uma aplicação de um heap de Fibonacci começa sem heaps. Então, o potencial inicial é 0 e, pela equação (20.1), o potencial é não negativo em todos os momentos subsequentes. Da equação (17.3), um limite superior sobre o custo total amortizado é portanto um limite superior sobre o custo real total para a seqüência de operações.

## Grau máximo

As análises amortizadas que realizaremos nas seções restantes deste capítulo pressupõem que existe um limite superior conhecido  $D(n)$  sobre o grau máximo de qualquer nó em um heap de Fibonacci de  $n$  nós. O Exercício 20.2-3 mostra que, quando apenas as operações de heaps intercaláveis são admitidas,  $D(n) \leq \lfloor \lg n \rfloor$ . Na Seção 20.3, mostraremos que, quando admitimos também DECREASE-KEY e DELETE,  $D(n) = O(\lg n)$ .

## 20.2 Operações de heaps intercaláveis

Nesta seção, descreveremos e analisaremos as operações de heaps intercaláveis implementadas para heaps de Fibonacci. Se apenas essas operações – MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN e UNION – tiverem de ser admitidas, cada heap de Fibonacci será simplesmente uma coleção de árvores binomiais “não ordenadas”. Uma *árvore binomial não ordenada* é semelhante a uma árvore binomial e, além disso, também é definida recursivamente. A árvore binomial não ordenada  $U_0$  consiste em um único nó, e uma árvore binomial não ordenada  $U_k$  consiste em duas árvores binomiais não ordenadas  $U_{k-1}$  para as quais a raiz de uma é transformada em *qualquer* filho da raiz da outra. O Lema 19.1, que fornece propriedades de árvores binomiais, é válido também para árvores binomiais não ordenadas, mas tem a seguinte variação na propriedade 4 (ver Exercício 20.2-2):

- 4'. Para a árvore binomial não ordenada  $U_k$ , a raiz tem grau  $k$ , maior que o de qualquer outro nó.  
Os filhos da raiz são raízes de subárvores  $U_0, U_1, \dots, U_{k-1}$  em alguma ordem.

Desse modo, se um heap de Fibonacci de  $n$  nós é uma coleção de árvores binomiais não ordenadas, então  $D(n) = \lg n$ .

A idéia chave nas operações de heaps intercaláveis sobre heaps de Fibonacci é retardar o trabalho o máximo possível. Existe um compromisso de desempenho entre implementações das várias operações. Se o número de árvores em um heap de Fibonacci é pequeno, então podemos determinar com rapidez durante uma operação EXTRACT-MIN qual dos nós restantes se torna o nó mínimo. Porém, como vimos com heaps binomiais no Exercício 19.2-10, pagamos um preço por assegurar que o número de árvores é pequeno: pode demorar até o tempo  $\Omega(\lg n)$  para inserir um nó em um heap binomial, ou para unir dois heaps binomiais. Como veremos, não tentamos consolidar árvores em um heap de Fibonacci quando inserirmos um novo nó ou unirmos dois heaps. Guardamos a consolidação para a operação EXTRACT-MIN, que é quando realmente precisamos encontrar o novo nó mínimo.

### Como criar um novo heap de Fibonacci

Para criar um heap de Fibonacci vazio, o procedimento MAKE-FIB-HEAP aloca e retorna o objeto heap de Fibonacci  $H$ , onde  $n[H] = 0$  e  $\min[H] = \text{NIL}$ ; não existe nenhuma árvore em  $H$ . Tendo em vista que  $t(H) = 0$  e  $m(H) = 0$ , o potencial do heap de Fibonacci vazio é  $\Phi(H) = 0$ . Desse modo, o custo amortizado de MAKE-FIB-HEAP é igual a seu custo real  $O(1)$ .

### Como inserir um nó

O procedimento a seguir insere o nó  $x$  no heap de Fibonacci  $H$ , supondo-se que o nó  $x$  já tenha sido alocado e que  $\text{chave}[x]$  já tenha sido preenchida.

```
FIB-HEAP-INSERT( $H, x$ )
1  $\text{grau}[x] \leftarrow 0$ 
2  $p[x] \leftarrow \text{NIL}$ 
3  $\text{filho}[x] \leftarrow \text{NIL}$ 
4  $\text{esquerdo}[x] \leftarrow x$ 
5  $\text{direito}[x] \leftarrow x$ 
6  $\text{marca}[x] \leftarrow \text{FALSE}$ 
7 concatenar a lista de raízes contendo  $x$  com a lista de raízes  $H$ 
8 if  $\min[H] = \text{NIL}$  or  $\text{chave}[x] < \text{chave}[\min[H]]$ 
9   then  $\min[H] \leftarrow x$ 
10   $n[H] \leftarrow n[H] + 1$ 
```

Depois das linhas 1 a 6 inicializarem os campos estruturais do nó  $x$ , tornando-o sua própria lista circular duplamente ligada, a linha 7 adiciona  $x$  à lista de raízes de  $H$  no tempo real  $O(1)$ . Desse modo, o nó  $x$  se torna uma árvore ordenada como heap mínimo de um único nó, e portanto uma árvore binomial não ordenada, no heap de Fibonacci. Ela não tem nenhum filho e está desmarcada. Então, as linhas 8 e 9 atualizam o ponteiro para o nó mínimo do heap de Fibonacci  $H$ , se necessário. Finalmente, a linha 10 incrementa  $n[H]$  para refletir a adição do novo nó. A Figura 20.2 mostra um nó com chave 21 inserido no heap de Fibonacci da Figura 20.1.

Diferente do procedimento BINOMIAL-HEAP-INSERT, FIB-HEAP-INSERT não faz nenhuma tentativa de consolidar as árvores dentro do heap de Fibonacci. Se ocorrem  $k$  operações FIB-HEAP-INSERT consecutivas, então  $k$  árvores de um único nó são adicionadas à lista de raízes.

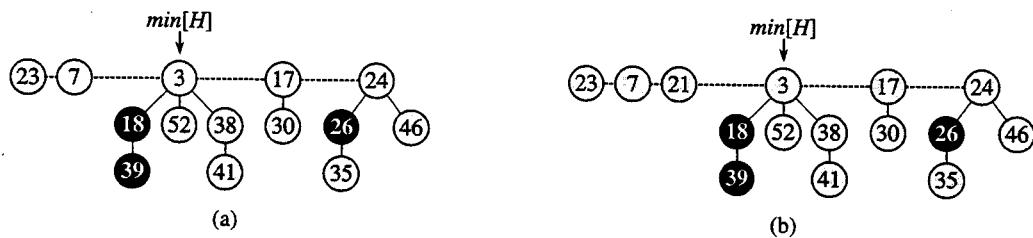


FIGURA 20.2 Inserindo um nó em um heap de Fibonacci. (a) Um heap de Fibonacci  $H$ . (b) O heap de Fibonacci  $H$  depois que o nó com chave 21 é inserido. O nó se torna sua própria árvore ordenada como heap mínimo e é então adicionado à lista de raízes, tornando-se o irmão esquerdo da raiz

Para determinar o custo amortizado de FIB-HEAP-INSERT, seja  $H$  o heap de Fibonacci de entrada e  $H'$  o heap de Fibonacci resultante. Então,  $t(H') = t(H) + 1$  e  $m(H') = m(H)$ , e o aumento em potencial é

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1.$$

Como o custo real é  $O(1)$ , o custo amortizado é  $O(1) + 1 = O(1)$ .

### Como encontrar o nó mínimo

O nó mínimo de um heap de Fibonacci  $H$  é dado pelo ponteiro  $\text{min}[H]$ ; assim, podemos encontrar o nó mínimo no tempo real  $O(1)$ . Tendo em vista que o potencial de  $H$  não muda, o custo amortizado dessa operação é igual a seu custo real  $O(1)$ .

### Como unir dois heaps de Fibonacci

O procedimento a seguir une os heaps de Fibonacci  $H_1$  e  $H_2$ , destruindo  $H_1$  e  $H_2$  no processo. Ele simplesmente concatena as listas de raízes de  $H_1$  e  $H_2$ , e depois determina o novo nó mínimo.

```

FIB-HEAP-UNION( $H_1, H_2$ )
1  $H \leftarrow \text{MAKE-FIB-HEAP}()$ 
2  $\text{min}[H] \leftarrow \text{min}[H_1]$ 
3 concatenar a lista de raízes de  $H_2$  com a lista de raízes de  $H$ 
4 if ( $\text{min}[H_1] = \text{NIL}$ ) or ( $\text{min}[H_2] \neq \text{NIL}$  e  $\text{min}[H_2] < \text{min}[H_1]$ )
5   then  $\text{min}[H] \leftarrow \text{min}[H_2]$ 
6  $n[H] \leftarrow n[H_1] + n[H_2]$ 
7 liberar os objetos  $H_1$  e  $H_2$ 
8 return  $H$ 
```

As linhas 1 a 3 concatenam as listas de raízes de  $H_1$  e  $H_2$  em uma nova lista de raízes  $H$ . As linhas 2, 4 e 5 definem o nó mínimo de  $H$ , e a linha 6 define  $n[H]$  como o número total de nós. Os objetos heap de Fibonacci  $H_1$  e  $H_2$  são liberados na linha 7, e a linha 8 retorna o heap de Fibonacci resultante  $H$ . Como no procedimento FIB-HEAP-INSERT, não ocorre nenhuma consolidação de árvores.

A mudança em potencial é

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) \\ = (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\ = 0,\end{aligned}$$

porque  $t(H) = t(H_1) + t(H_2)$  e  $m(H_1) + m(H_2)$ . Desse modo, o custo amortizado de FIB-HEAP-UNION é igual a seu custo real  $O(1)$ .

## Como extraír o nó mínimo

O processo de extraír o nó mínimo é a mais complicada das operações apresentadas nesta seção. Ele também é o local onde o trabalho adiado de consolidar árvores na lista de raízes finalmente ocorre. O pseudocódigo a seguir extraí o nó mínimo. O código pressupõe por conveniência que, quando um nó é removido de uma lista ligada, os ponteiros restantes na lista são atualizadas, mas os ponteiros no nó extraído se mantêm inalterados. Ele também usa o procedimento auxiliar CONSOLIDATE, que será apresentado em breve.

### FIB-HEAP-EXTRACT-MIN( $H$ )

```

1  $z \leftarrow min[H]$ 
2 if  $z \neq NIL$ 
3   then for cada filho  $x$  de  $z$ 
4     do adicionar  $x$  à lista de raízes de  $H$ 
5      $p[x] \leftarrow NIL$ 
6   remover  $z$  da lista de raízes de  $H$ 
7   if  $z = direito[z]$ 
8     then  $min[H] \leftarrow NIL$ 
9     else  $min[H] \leftarrow direito[z]$ 
10    CONSOLIDATE( $H$ )
11     $n[H] \leftarrow n[H] - 1$ 
12 return  $z$ 
```

Como mostra a Figura 20.3, FIB-HEAP-EXTRACT-MIN funciona primeiro criando uma raiz a partir de cada um dos filhos do nó mínimo e removendo o nó mínimo da lista de raízes. Em seguida, ele consolida a lista de raízes, ligando raízes de mesmo grau até restar no máximo uma raiz de cada grau.

Começamos na linha 1 gravando um ponteiro  $z$  para o nó mínimo; esse ponteiro será retornado no final. Se  $z = NIL$ , então o heap de Fibonacci  $H$  já está vazio, e terminamos. Caso contrário, como no procedimento BINOMIAL-HEAP-EXTRACT-MIN, eliminamos o nó  $z$  de  $H$ , tornando todos os filhos de  $z$  raízes de  $H$  nas linhas 3 a 5 (inserindo-os na lista de raízes) e removendo  $z$  da lista de raízes na linha 6. Se  $z = direito[z]$  depois da linha 6, então  $z$  era o único nó na lista de raízes e não tinha nenhum filho; assim, só falta tornar o heap de Fibonacci vazio na linha 8 antes de retornar  $z$ . Caso contrário, definimos o ponteiro  $min[H]$  na lista de raízes de modo a apontar para um outro nó diferente de  $z$  (nesse caso,  $direito[z]$ ), que não será necessariamente o novo nó mínimo quando FIB-HEAP-EXTRACT-MIN terminar. A Figura 20.3(b) mostra o heap de Fibonacci da Figura 20.3(a) depois da linha 9 ter sido executada.

O próximo passo, no qual reduzimos o número de árvores no heap de Fibonacci, é **consolidar** a lista de raízes de  $H$ ; isso é realizado pela chamada a  $\text{CONSOLIDATE}(H)$ . A consolidação da lista de raízes consiste em executar repetidamente os passos a seguir até que toda raiz na lista de raízes tenha um valor de *grau* distinto.

1. Encontre duas raízes  $x$  e  $y$  na lista de raízes com o mesmo grau, onde  $\text{chave}[x] = \text{chave}[y]$ .
2. **Ligue**  $y$  a  $x$ : Remova  $y$  da lista de raízes e torne  $y$  um filho de  $x$ . Essa operação é executada pelo procedimento **FIB-HEAP-LINK**. O campo  $\text{grau}[x]$  é incrementado, e a marca em  $y$ , se houver, é retirada.

O procedimento **CONSOLIDATE** usa um arranjo auxiliar  $A[0 .. D(n[H])]$ ; se  $A[i] = y$ , então  $y$  é atualmente uma raiz com  $\text{grau}[y] = i$ .

#### **CONSOLIDATE( $H$ )**

```

1 for  $i \leftarrow 0$  to  $D(n[H])$ 
2   do  $A[i] \leftarrow \text{NIL}$ 
3 for cada nó  $w$  na lista de raízes de  $H$ 
4   do  $x \leftarrow w$ 
5      $d \leftarrow \text{grau}[x]$ 
6     while  $A[d] \neq \text{NIL}$ 
7       do  $y \leftarrow A[d]$  > Outro nó com o mesmo grau de  $x$ .
8         if  $\text{chave}[x] > \text{chave}[y]$ 
9           then trocar  $x \leftrightarrow y$ 
10          FIB-HEAP-LINK( $H, y, x$ )
11           $A[d] \leftarrow \text{NIL}$ 
12           $d \leftarrow d + 1$ 
13         $A[d] \leftarrow x$ 
14 min[ $H$ ]  $\leftarrow \text{NIL}$ 
15 for  $i \leftarrow 0$  to  $D(n[H])$ 
16   do if  $A[i] \neq \text{NIL}$ 
17     then adicionar  $A[i]$  à lista de raízes de  $H$ 
18     if  $\text{min}[H] = \text{NIL}$  or  $\text{chave}[A[i]] < \text{chave}[\text{min}[H]]$ 
19       then  $\text{min}[H] \leftarrow A[i]$ 

```

#### **FIB-HEAP-LINK( $H, y, x$ )**

```

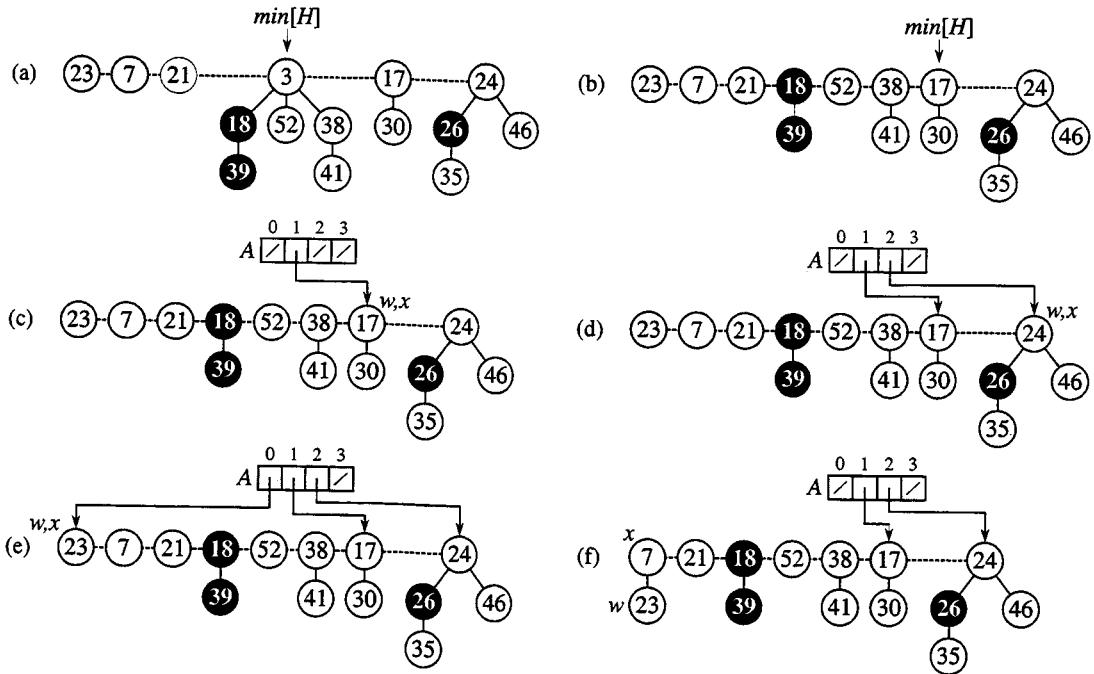
1 remover  $y$  da lista de raízes de  $H$ 
2 tornar  $y$  um filho de  $x$ , incrementando  $\text{grau}[x]$ 
3  $\text{marca}[y] \leftarrow \text{FALSE}$ 

```

Em detalhes, o procedimento **CONSOLIDATE** funciona da maneira descrita a seguir. As linhas 1 e 2 inicializam  $A$  tornando cada entrada **NIL**. O loop **for** das linhas 3 a 13 processa cada raiz  $w$  na lista de raízes. Após o processamento de cada raiz  $w$ , ela termina em uma árvore com raiz em algum nó  $x$ , que pode ou não ser idêntico a  $w$ . Das raízes processadas, nenhuma outra terá o mesmo grau que  $x$ , e então definiremos a entrada do arranjo  $A[\text{grau}[x]]$  de modo a apontar para  $x$ . Quando esse loop **for** terminar, no máximo uma raiz de cada grau permanecerá, e o arranjo  $A$  apontará para cada raiz restante.

O loop **while** das linhas 6 a 12 liga repetidamente a raiz  $x$  da árvore que contém o nó  $w$  a outra árvore cuja raiz tem o mesmo grau que  $x$ , até que nenhuma outra raiz tenha o mesmo grau. Esse loop **while** mantém o seguinte invariante:

No início de cada iteração do loop **while**,  $d = \text{grau}[x]$ .



**FIGURA 20.3** A ação de FIB-HEAP-EXTRACT-MIN. (a) Um heap de Fibonacci  $H$ . (b) A situação depois que o nó mínimo  $z$  é removido da lista de raízes e seus filhos são adicionados à lista de raízes. (c)–(e) O arranjo  $A$  e as árvores depois de cada uma das três primeiras iterações do loop for das linhas 3 a 13 do procedimento CONSOLIDATE. A lista de raízes é processada, começando-se no nó apontado por  $\text{min}[H]$  e seguindo-se os ponteiros *direito*. Cada parte mostra os valores de  $w$  e  $x$  no fim de uma iteração. (f)–(h) A próxima iteração do loop for, com os valores de  $w$  e  $x$  mostrados no fim de cada iteração do loop while das linhas 6 a 12. A parte (f) mostra a situação após a primeira passagem pelo loop while. O nó com chave 23 foi ligado ao nó com chave 7, que agora é apontado por  $x$ . Na parte (g), o nó com chave 17 foi ligado ao nó com chave 7, o qual ainda é apontado por  $x$ . Na parte (h), o nó com chave 24 foi ligado ao nó com chave 7. Como nenhum nó foi apontado anteriormente por  $A[3]$ , no fim da iteração do loop for,  $A[3]$  é definido de modo a apontar para a raiz da árvore resultante. (i)–(l) A situação depois de cada uma das quatro iterações seguintes do loop for. (m) O heap de Fibonacci  $H$  depois da reconstrução da lista de raízes a partir do arranjo  $A$  e da determinação do novo ponteiro  $\text{min}[H]$

Usamos esse loop invariante assim:

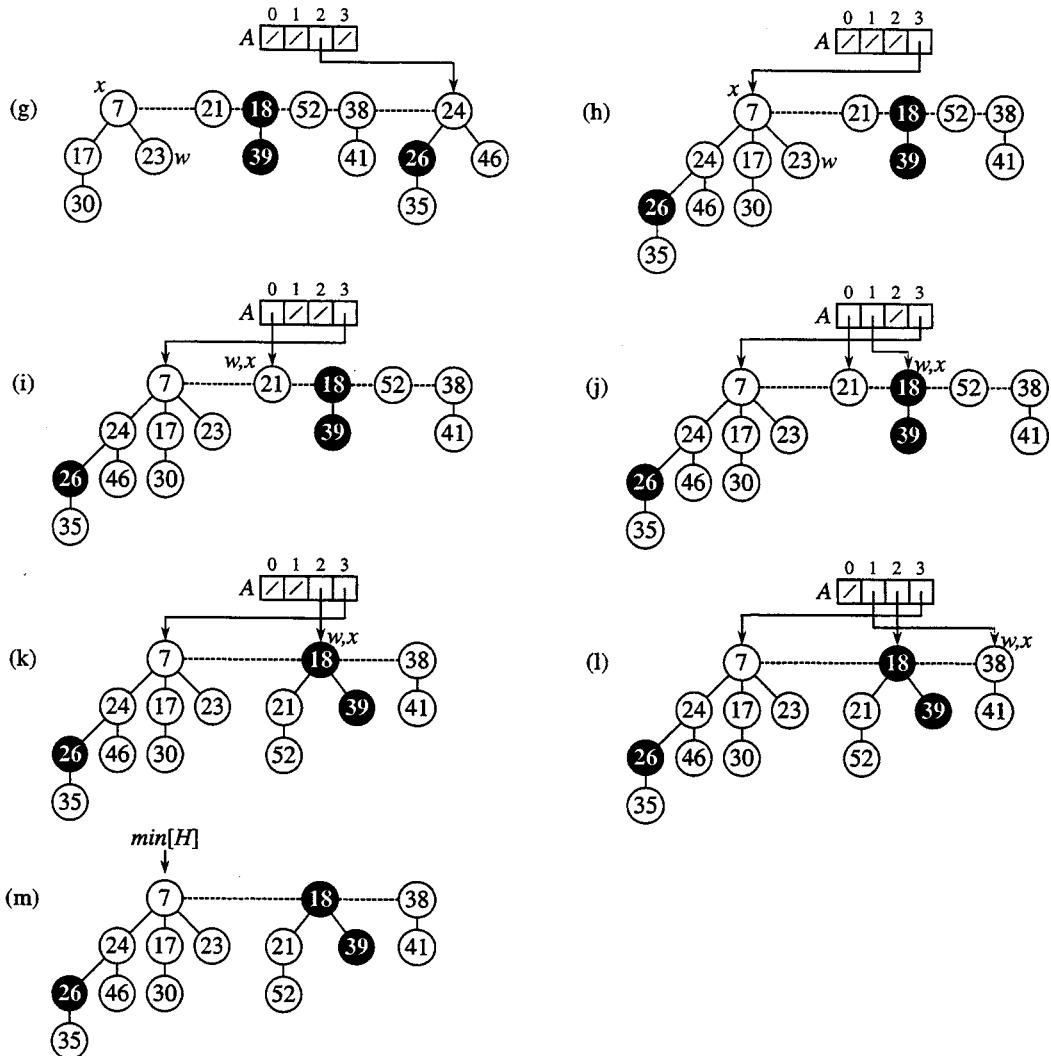
**Inicialização:** A linha 5 assegura que o loop invariante é válido na primeira vez em que entramos no loop.

**Manutenção:** Em cada iteração do loop while,  $A[d]$  aponta para alguma raiz  $y$ . Como  $d = \text{grau}[x] = \text{grau}[y]$ , queremos ligar  $x$  e  $y$ . Entre  $x$  e  $y$ , a que tiver a menor chave se tornará o pai da outra, como resultado da operação de ligação, e assim as linhas 8 e 9 trocam os ponteiros para  $x$  e  $y$ , se necessário. Em seguida, ligamos  $x$  a  $y$  pela chamada FIB-HEAP-LINK( $H, y, x$ ) na linha 10. Essa chamada incrementa  $\text{grau}[x]$ , mas deixa  $\text{grau}[y]$  como  $d$ . Como o nó  $y$  não é mais uma raiz, o ponteiro para ele no arranjo  $A$  é removido na linha 11. Como a chamada de FIB-HEAP-LINK incrementa o valor de  $\text{grau}[x]$ , a linha 12 restaura o invariante de que  $d = \text{grau}[x]$ .

**Término:** Repetimos o loop while até  $A[d] = \text{NIL}$ , e nesse caso não existe nenhuma outra raiz com o mesmo grau que  $x$ .

Depois que o loop while termina, definimos  $A[d]$  como  $x$  na linha 13 e executamos a próxima iteração do loop for.

As Figuras 20.3(c)–(e) mostram o arranjo  $A$  e as árvores resultantes após as três primeiras iterações do loop for das linhas 3 a 13. Na próxima iteração do loop for, ocorrem três ligações; seus



resultados são mostrados nas Figuras 20.3(f)-(h). As Figuras 20.3(i)-(l) mostram o resultado das quatro iterações seguintes do loop **for**.

Agora, só falta a limpeza. Depois que o loop **for** das linhas 3 a 13 se completa, a linha 14 esvazia a lista de raízes, e as linhas 15 a 19 recriam a lista a partir do arranjo  $A$ . O heap de Fibonacci resultante é mostrado na Figura 20.3(m). Depois de consolidar a lista de raízes, FIB-HEAP-EXTRACT-MIN termina decrementando  $n[H]$  na linha 11 e retornando um ponteiro para o nó eliminado  $z$  na linha 12.

Observe que, se todas as árvores no heap de Fibonacci são árvores binomiais não ordenadas antes de FIB-HEAP-EXTRACT-MIN ser executado, então todas elas são árvores binomiais não ordenadas depois disso. Existem dois modos pelos quais as árvores são alteradas. Primeiro, nas linhas 3 a 5 de FIB-HEAP-EXTRACT-MIN, cada filho  $x$  da raiz  $z$  se torna uma raiz. Pelo Exercício 20.2-2, cada nova árvore é ela própria uma árvore binomial não ordenada. Em segundo lugar, as árvores são ligadas por FIB-HEAP-LINK apenas se elas têm o mesmo grau. Tendo em vista que todas as árvores são árvores binomiais não ordenadas antes de ocorrer a ligação, duas árvores cujas raízes tenham cada uma  $k$  filhos devem ter a estrutura  $U_k$ . A árvore resultante tem então a estrutura  $U_{k+1}$ .

Agora, estamos prontos para mostrar que o custo amortizado de extrair o nó mínimo de um heap de Fibonacci de  $n$  nós é  $O(D(n))$ . Seja  $H$  a representação do heap de Fibonacci imediatamente antes da operação FIB-HEAP-EXTRACT-MIN.

O custo real de extrair o nó mínimo pode ser considerado da maneira a seguir. Uma contribuição  $O(D(n))$  vem da existência de no máximo  $D(n)$  filhos do nó mínimo que são processados em FIB-HEAP-EXTRACT-MIN e do trabalho nas linhas 1 e 2 e 14 a 19 de CONSOLIDATE. Resta

analisar a contribuição do loop **for** das linhas 3 a 13. O tamanho da lista de raízes na chamada a CONSOLIDATE é no máximo  $D(n) + t(H) - 1$ , pois ele consiste nos  $t(H)$  nós da lista de raízes original, menos o nó de raiz extraído, mais os filhos do nó extraído, o que soma no máximo  $D(n)$ . Toda vez que se passa pelo loop **while** das linhas 6 a 12, uma das raízes é ligada a outra, e assim o valor total do trabalho executado no loop **for** é no máximo proporcional a  $D(n) + t(H)$ . Portanto, o trabalho total real na extração do nó mínimo é  $O(D(n) + t(H))$ .

O potencial antes da extração do nó mínimo é  $t(H) + 2m(H)$ , e o potencial depois disso é no máximo  $(D(n) + 1) + 2m(H)$ , pois restam no máximo  $D(n) + 1$  raízes, e nenhum nó foi marcado durante a operação. Desse modo, o custo amortizado é no máximo

$$\begin{aligned} O(H)(n) + t(H) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ = O(D(n)) + O(t(H)) - t(H) \\ = O(D(n)), \end{aligned}$$

pois podemos ajustar a escala das unidades de potencial para dominar a constante oculta em  $O(t(H))$ . Intuitivamente, o custo da execução de cada ligação é compensado pela redução em potencial devido ao fato de que o vínculo reduz o número de raízes em uma unidade. Veremos na Seção 20.4 que  $D(n) = O(\lg n)$ , de forma que o custo amortizado de extrair o nó mínimo é  $O(\lg n)$ .

## Exercícios

### 20.2-1

Mostre o heap de Fibonacci resultante da chamada a FIB-HEAP-EXTRACT-MIN sobre o heap de Fibonacci mostrado na Figura 20.3(m).

### 20.2-2

Prove que o Lema 19.1 se mantém válido para árvores binomiais não ordenadas, mas com a propriedade 4' em lugar da propriedade 4.

### 20.2-3

Mostre que, se apenas as operações de heaps intercaláveis forem admitidas, o grau máximo  $D(n)$  em um heap de Fibonacci de  $n$  nós será no máximo  $\lfloor \lg n \rfloor$ .

### 20.2-4

O professor McGee criou uma nova estrutura de dados baseada em heaps de Fibonacci. Um heap de McGee tem a mesma estrutura de um heap de Fibonacci e admite as operações de heaps intercaláveis. As implementações das operações são idênticas às de heaps de Fibonacci, exceto pelo fato de que a inserção e a união executam a consolidação como seu último passo. Qual é o tempo de execução no pior caso das operações sobre heaps de McGee? O quanto é nova a estrutura de dados do professor?

### 20.2-5

Demonstre que, quando as únicas operações sobre chaves são as de comparar duas chaves (como ocorre para todas as implementações deste capítulo), nem todas as operações de heaps intercaláveis podem ser executadas no tempo amortizado  $O(1)$ .

## 20.3 Como decrementar uma chave e eliminar um nó

Nesta seção, mostraremos como decrementar a chave de um nó em um heap de Fibonacci no tempo amortizado  $O(1)$  e como eliminar qualquer nó de um heap de Fibonacci de  $n$  nós no tempo amortizado  $O(D(n))$ . Essas operações não preservam a propriedade de que todas as árvores no heap de Fibonacci são árvores binomiais não ordenadas. Porém, elas são suficientemente

próximas para podermos limitar o grau máximo  $D(n)$  por  $O(\lg n)$ . A prova desse limite, que faremos na Seção 20.4, implicará que FIB-HEAP-EXTRACT-MIN e FIB-HEAP-DELETE serão executados no tempo amortizado  $O(\lg n)$ .

## Como decrementar uma chave

No pseudocódigo a seguir para a operação FIB-HEAP-DECREASE-KEY, supomos como antes que a remoção de um nó de uma lista ligada não muda quaisquer dos campos estruturais no nó removido.

**FIB-HEAP-DECREASE-KEY( $H, x, k$ )**

```

1 if  $k > chave[x]$ 
2 then error "nova chave é maior que chave atual"
3  $chave[x] \leftarrow k$ 
4  $y \leftarrow p[x]$ 
5 if  $y \neq \text{NIL}$  e  $chave[x] < chave[y]$ 
6   then CUT( $H, x, y$ )
7     CASCADING-CUT( $H, y$ )
8 if  $chave[x] < chave[min[H]]$ 
9 then  $min[H] \leftarrow x$ 
```

**CUT( $H, x, y$ )**

```

1 remover  $x$  da lista de filhos de  $y$ , decrementando  $grau[y]$ 
2 adicionar  $x$  à lista de raízes de  $H$ 
3  $p[x] \leftarrow \text{NIL}$ 
4  $marca[x] \leftarrow \text{FALSE}$ 
```

**CASCADING-CUT( $H, y$ )**

```

1  $z \leftarrow p[y]$ 
2 if  $z \neq \text{NIL}$ 
3   then if  $marca[y] = \text{FALSE}$ 
4     then  $marca[y] \leftarrow \text{TRUE}$ 
5     else CUT( $H, y, z$ )
6       CASCADING-CUT( $H, z$ )
```

O procedimento FIB-HEAP-DECREASE-KEY funciona da maneira descrita a seguir. As linhas 1 a 3 asseguram que a nova chave não é maior que a chave atual de  $x$ , e então atribuem a nova chave a  $x$ . Se  $x$  é uma raiz ou se  $chave[x] \geq chave[y]$ , onde  $y$  é pai de  $x$ , então não precisa ocorrer nenhuma mudança estrutural, pois a ordem do heap não foi violada. As linhas 4 e 5 testam essa condição.

Se a ordem de heap mínimo foi violada, muitas mudanças podem ocorrer. Começamos cortando  $x$  na linha 6. O procedimento CUT “corta” o vínculo entre  $x$  e seu pai  $y$ , fazendo de  $x$  uma raiz.

Usamos os campos *marca* para obter os limites de tempo desejados. Eles registram uma pequena fração da história de cada nó. Suponha que os eventos a seguir ocorreram com o nó  $x$ :

1. Em algum momento,  $x$  era uma raiz,
2. depois  $x$  foi ligado a outro nó,
3. então dois filhos de  $x$  foram removidos por cortes.

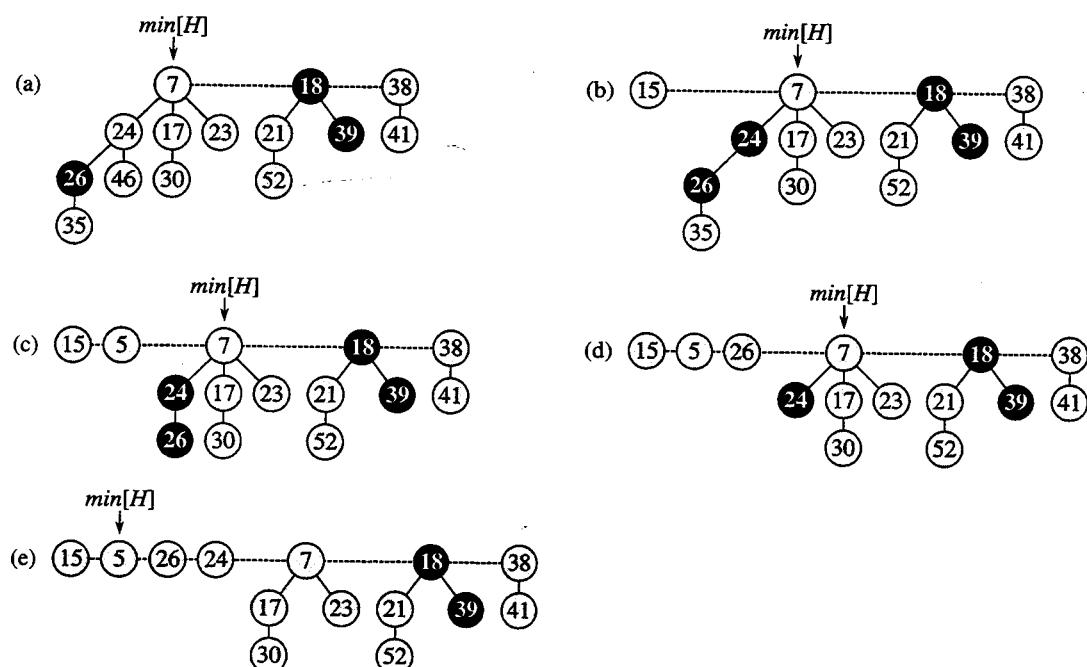
Assim que o segundo filho é perdido,  $x$  é cortado de seu pai, o que o torna uma nova raiz. O campo *marca*[ $x$ ] é TRUE se os Passos 1 e 2 ocorreram e um filho de  $x$  foi cortado. Então, o proce-

dimento CUT retira  $marca[x]$  na linha 4, pois ele executa o Passo 1. (Agora, podemos ver por que a linha 3 de FIB-HEAP-LINK retira  $marca[y]$ : o nó  $y$  está sendo ligado a outro nó, e assim o Passo 2 está sendo executado. Da próxima vez que um filho de  $y$  for cortado,  $marca[y]$  será definido como TRUE.)

Ainda não terminamos, porque  $x$  poderia ser o segundo filho cortado de seu pai  $y$ , desde o tempo em que  $y$  esteve ligado a outro nó. Então, a linha 7 de FIB-HEAP-DECREASE-KEY executa uma operação de **corte em cascata** sobre  $y$ . Se  $y$  é uma raiz, então o teste na linha 2 de CASCADING-CUT faz o procedimento simplesmente retornar. Se  $y$  está desmarcado, o procedimento o marca na linha 4, pois seu primeiro filho acabou de ser cortado, e retorna. Contudo, se  $y$  está marcado, ele perdeu apenas seu segundo filho;  $y$  é cortado na linha 5, e CASCADING-CUT chama a si mesmo recursivamente na linha 6 sobre o pai  $z$  de  $y$ . O procedimento CASCADING-CUT efetua recursões dessa maneira, subindo a árvore até encontrar uma raiz ou um nó não marcado.

Uma vez que tenham ocorrido todos os cortes em cascata, as linhas 8 e 9 de FIB-HEAP-DECREASE-KEY concluem atualizando  $\min[H]$  se necessário. O único nó cuja chave mudou era o nó  $x$  que teve sua chave decrementada. Desse modo, o novo nó mínimo é o nó original ou o nó  $x$ .

A Figura 20.4 mostra a execução de duas chamadas a FIB-HEAP-DECREASE-KEY, começando com o heap de Fibonacci mostrado na Figura 20.4(a). A primeira chamada, mostrada na Figura 20.4(b), não envolve nenhum corte em cascata. A segunda chamada, mostrada nas Figuras 20.4(c)-(e), invoca dois cortes em cascata.



**FIGURA 20.4** Duas chamadas de FIB-HEAP-DECREASE-KEY. (a) O heap de Fibonacci inicial. (b) O nó com chave 46 tem sua chave decrementada para 15. O nó se torna uma raiz, e seu pai (com chave 24), que anteriormente foi desmarcado, se torna marcado. (c)–(e) O nó com chave 35 tem sua chave decrementada para 5. Na parte (c), o nó, agora com chave 5, se torna uma raiz. Seu pai, com chave 26, é marcado, e assim ocorre um corte em cascata. O nó com chave 26 é cortado de seu pai e transformado em uma raiz não marcada em (d). Ocorre outro corte em cascata, pois o nó com chave 24 também é marcado. Esse nó é cortado de seu pai e transformado em uma raiz não marcada na parte (e). Os cortes em cascata param nesse ponto, pois o nó com chave 7 é uma raiz. (Ainda que esse nó não fosse uma raiz, os cortes em cascata parariam, pois ele é não marcado.) O resultado da operação FIB-HEAP-DECREASE-KEY é mostrado na parte (e), com  $\min[H]$  apontando para o novo nó mínimo.

Agora, mostraremos que o custo amortizado de FIB-HEAP-DECREASE-KEY é apenas  $O(1)$ . Começamos determinando seu custo real. O procedimento FIB-HEAP-DECREASE-KEY demora o tempo  $O(1)$ , mais o tempo para executar os cortes em cascata. Suponha que CASCADING-CUT seja chamado recursivamente  $c$  vezes a partir de uma dada invocação de FIB-HEAP-DECREASE-KEY. Cada chamada de CASCADING-CUT demora o tempo  $O(1)$  sem incluir as chamadas recursivas. Desse modo, o custo real de FIB-HEAP-DECREASE-KEY, incluindo todas as chamadas recursivas, é  $O(c)$ .

Em seguida, calculamos a mudança no potencial. Seja  $H$  um valor que denota o heap de Fibonacci imediatamente antes da operação de FIB-HEAP-DECREASE-KEY. Cada chamada recursiva de CASCADING-CUT, com exceção da última, corta um nó marcado e retira o bit de marcação. Posteriormente, existem  $t(H) + c$  árvores (as  $t(H)$  árvores originais,  $c - 1$  árvores produzidas por cortes em cascata e a árvore com raiz em  $x$ ) e no máximo  $m(H) - c + 2$  nós marcados ( $c - 1$  foram desmarcados por cortes em cascata, e a última chamada de CASCADING-CUT pode ter marcado um nó). Então, a mudança em potencial é no máximo

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c.$$

Desse modo, o custo amortizado de FIB-HEAP-DECREASE-KEY é no máximo

$$O(c) + 4 - c = O(1),$$

pois podemos ajustar a escala das unidades de potencial para dominar a constante oculta em  $O(c)$ .

Agora, você pode ver por que a função potencial foi definida para incluir um termo que é duas vezes o número de nós marcados. Quando um nó marcado  $y$  é cortado por uma operação de corte em cascata, seu bit de marcação é limpo, e assim o potencial é reduzido em 2 unidades. Uma unidade potencial compensa o corte e a retirada do bit de marcação, e a outra unidade compensa o aumento de uma unidade no potencial devido à transformação do nó  $y$  em uma raiz.

## Como eliminar um nó

É fácil eliminar um nó de um heap de Fibonacci de  $n$  nós no tempo amortizado  $O(D(n))$ , como é feito pelo pseudocódigo a seguir. Supomos que não existe atualmente nenhum valor de chave  $-\infty$  no heap de Fibonacci.

```
FIB-HEAP-DELETE( $H, x$ )
1 FIB-HEAP-DECREASE-KEY( $H, x, -\infty$ )
2 FIB-HEAP-EXTRACT-MIN( $H$ )
```

FIB-HEAP-DELETE é análogo a BINOMIAL-HEAP-DELETE. Ele faz  $x$  se tornar o nó mínimo no heap de Fibonacci, dando a ele uma chave pequena exclusivamente igual a  $-\infty$ . O nó  $x$  é então removido do heap de Fibonacci pelo procedimento FIB-HEAP-EXTRACT-MIN. O tempo amortizado de FIB-HEAP-DELETE é a soma do tempo amortizado  $O(1)$  de FIB-HEAP-DECREASE-KEY com o tempo amortizado  $O(D(n))$  de FIB-HEAP-EXTRACT-MIN. Tendo em vista que veremos na Seção 20.4 que  $D(n) = O(\lg n)$ , o tempo amortizado de FIB-HEAP-DELETE é  $O(\lg n)$ .

## Exercícios

### 20.3-1

Suponha que uma raiz  $x$  em um heap de Fibonacci esteja marcada. Explique como  $x$  se tornou uma raiz marcada. Demonstre que não importa para a análise que o nó  $x$  esteja marcado, ainda que ele não seja uma raiz que primeiro foi ligada a outro nó e depois perdeu um filho.

### 20.3-2

Justifique o tempo amortizado  $O(1)$  de FIB-HEAP-DECREASE-KEY como um custo médio por operação, usando análise agregada.

## 20.4 Como limitar o grau máximo

Para provar que o tempo amortizado de FIB-HEAP-EXTRACT-MIN e FIB-HEAP-DELETE é  $O(\lg n)$ , devemos mostrar que o limite superior  $D(n)$  sobre o grau de qualquer nó de um heap de Fibonacci de  $n$  nós é  $O(\lg n)$ . Pelo Exercício 20.2-3, quando todas as árvores no heap de Fibonacci são árvores binomiais não ordenadas,  $D(n) = \lfloor \lg n \rfloor$ . Porém, os cortes que ocorrem em FIB-HEAP-DECREASE-KEY podem fazer as árvores dentro do heap de Fibonacci violarem as propriedades de árvores binomiais não ordenadas. Nesta seção, mostraremos que, como cortamos um nó de seu pai logo que ele perde dois filhos,  $D(n)$  é  $O(\lg n)$ . Em particular, mostraremos que  $D(n) \leq \lfloor \log_2 n \rfloor$ , onde  $\theta = (1 + \sqrt{5})/2$ .

A chave para a análise é dada a seguir. Para cada nó  $x$  dentro de um heap de Fibonacci, defina  $\text{tamanho}(x)$  como o número de nós, inclusive o próprio  $x$ , na subárvore com raiz em  $x$ . (Observe que  $x$  não precisa estar na lista de raízes – ele pode ser absolutamente qualquer nó.) Mostraremos que  $\text{tamanho}(x)$  é exponencial em  $\text{grau}[x]$ . Lembre-se de que  $\text{grau}[x]$  é sempre mantido como uma medida precisa do grau de  $x$ .

### Lema 20.1

Seja  $x$  qualquer nó em um heap de Fibonacci, e suponha que  $\text{grau}[x] = k$ . Seja  $y_1, y_2, \dots, y_k$  a série de filhos de  $x$  na ordem em que eles estavam ligados a  $x$ , desde o mais antigo até o mais recente. Então,  $\text{grau}[y_1] \geq 0$  e  $\text{grau}[y_i] \geq i - 2$  para  $i = 2, 3, \dots, k$ .

**Prova** Obviamente,  $\text{grau}[y_1] \geq 0$ .

Para  $i \geq 2$ , observamos que, quando  $y_i$  foi ligado a  $x$ , todos os itens  $y_1, y_2, \dots, y_{k-1}$  eram filhos de  $x$ , e assim também devemos ter tido  $\text{grau}[x] \geq i - 1$ . O nó  $y_i$  é ligado a  $x$  somente se  $\text{grau}[x] = \text{grau}[y_i]$ ; portanto, também devemos ter tido  $\text{grau}[y_i] \geq i - 1$  naquele momento. Desde então, o nó  $y_i$  perdeu no máximo um filho, pois ele teria sido cortado de  $x$  se tivesse perdido dois filhos. Concluímos que  $\text{grau}[y_i] \geq i - 2$ . ■

Por fim, chegamos à parte da análise que explica o nome “heaps de Fibonacci”. Vimos na Seção 3.2 que, para  $k = 0, 1, 2, \dots$ , o  $k$ -ésimo número de Fibonacci é definido pela recorrência

$$F_k = \begin{cases} 0 & \text{se } k = 0 , \\ 1 & \text{se } k = 1 , \\ F_{k-1} + F_{k-2} & \text{se } k \geq 2 . \end{cases}$$

O lema a seguir fornece outro caminho para expressar  $F_k$ .

### Lema 20.2

Para todos os inteiros  $k \geq 0$ ,

$$F_{k+2} = 1 + \sum_{i=0}^k F_i$$

**Prova** A prova é por indução sobre  $k$ . Quando  $k = 0$ ,

$$\begin{aligned} 1 + \sum_{i=0}^k F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= 1 \\ &= F_2 . \end{aligned}$$

Agora, supomos a hipótese indutiva de que  $F_k + 1 = 1 + \sum_{i=0}^{k-1} F_i$ , e temos

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left( 1 + \sum_{i=0}^{k-1} F_i \right) \\ &= 1 + \sum_{i=0}^k F_i . \end{aligned}$$

■

O lema a seguir e seu corolário completam a análise. Eles utilizam a desigualdade (provada no Exercício 3.2-7)

$$F_{k+2} \geq \phi^k ,$$

onde  $\phi$  é a razão áurea definida na equação (3.22) como  $\phi = (1 + \sqrt{5})/2 = 1.61803\dots$

### Lema 20.3

Seja  $x$  qualquer nó em um heap de Fibonacci, e seja  $k = \text{grau}[x]$ . Então,  $\text{tamanho}(x) \geq F_{k+2} \geq \phi^k$ , onde  $\phi = (1 + \sqrt{5})/2$ .

**Prova** Seja  $s_k$  o valor mínimo possível de  $\text{tamanho}(z)$  sobre todos os nós  $z$  tais que  $\text{grau}[z] = k$ . De modo trivial,  $s_0 = 1$ ,  $s_1 = 2$  e  $s_2 = 3$ . O número  $s_k$  é no máximo  $\text{tamanho}(x)$  e, é claro, o valor de  $s_k$  aumenta monotonicamente com  $k$ . Como no Lema 20.1, seja  $y_1, y_2, \dots, y_k$  a série de filhos de  $x$  na ordem em que eles foram ligados a  $x$ . Para calcular um limite inferior sobre  $\text{tamanho}(x)$ , contamos um para o próprio  $x$  e um para o primeiro filho  $y_1$  (para o qual  $\text{tamanho}(y_1) \geq 1$ ), o que nos dá

$$\text{tamanho}(x) \geq s_k$$

$$\begin{aligned} &= 2 + \sum_{i=2}^k S_{\text{grau}[y_i]} \\ &\geq 2 + \sum_{i=2}^k s_{i-2} , \end{aligned}$$

onde a última linha decorre do Lema 20.1 (de forma que  $\text{grau}[y_i] \geq i - 2$ ) e da monotonicidade de  $s_k$  (de forma que  $s_{\text{grau}[y_i]} \geq s_{i-2}$ ).

Agora, mostramos por indução sobre  $k$  que  $s_k \geq F_{k+2}$  para todo inteiro não negativo  $k$ . As bases, para  $k = 0$  e  $k = 1$  são triviais. Para a etapa indutiva, supomos que  $k \geq 2$  e que  $s_i \geq F_{i+2}$  para  $i = 0, 1, \dots, k-1$ . Temos

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^k s_{i-2} \\ &\geq 2 + \sum_{i=2}^k F_i \\ &= 1 + \sum_{i=0}^k F_i \\ &= F_{k+2} \quad (\text{pelo Lema 20.2}) . \end{aligned}$$

Desse modo, mostramos que  $\text{tamanho}(x) \geq s_k \geq F_{k+2} \geq \phi^k$ .

■

### **Corolário 20.4**

O grau máximo  $D(n)$  de qualquer nó em um heap de Fibonacci de  $n$  nós é  $O(\lg n)$ .

**Prova** Seja  $x$  qualquer nó em um heap de Fibonacci de  $n$  nós, e seja  $k = \text{grau}[x]$ . Pelo Lema 20.3, temos  $n \geq \text{tamanho}(x) \geq \phi^k$ . O uso de logaritmos de base  $\phi$  produz  $k \leq \log_\phi n$ . (De fato, como  $k$  é um inteiro,  $k \leq \lfloor \log_\phi n \rfloor$ .) O grau máximo  $D(n)$  de qualquer nó é portanto  $O(\lg n)$ .

## **Exercícios**

### **20.4-1**

O professor Pinocchio afirma que a altura de um heap de Fibonacci de  $n$  nós é  $O(\lg n)$ . Mostre que o professor está equivocado ao demonstrar que, para qualquer inteiro positivo  $n$ , uma seqüência de operações de heaps de Fibonacci cria um heap de Fibonacci consistindo apenas em uma árvore, sendo essa árvore uma cadeia linear de  $n$  nós.

### **20.4-2**

Suponha que a regra do corte em cascata seja generalizada para cortar um nó  $x$  de seu pai logo que ele perde seu  $k$ -ésimo filho, para alguma constante inteira  $k$ . (A regra da Seção 20.3 usa  $k = 2$ .) Para que valores de  $k$  temos  $D(n) = O(\lg n)$ ?

## **Problemas**

### **20-1 Implementação alternativa da eliminação**

O professor Pisano propôs a seguinte variação do procedimento FIB-HEAP-DELETE, afirmando que ele é executado com maior rapidez quando o nó que está sendo eliminado não é o nó apontado por  $\text{min}[H]$ .

PISANO-DELETE( $H, x$ )

```

1 if  $x = \text{min}[H]$ 
2   then FIB-HEAP-EXTRACT-MIN( $H$ )
3 else  $y \leftarrow p[x]$ 
4   if  $y \neq \text{NIL}$ 
5     then CUT( $H, x, y$ )
6       CASCADING-CUT( $H, y$ )
7   adicionar a lista de filhos de  $x$  à lista de raízes de  $H$ 
8   remover  $x$  da lista de raízes de  $H$ 

```

- A afirmação do professor de que esse procedimento é executado com maior rapidez se baseia em parte na hipótese de que a linha 7 pode ser executada no tempo real  $O(1)$ . O que está errado com essa hipótese?
- Forneça um bom limite superior sobre o tempo real de PISANO-DELETE quando  $x$  não é  $\text{min}[H]$ . Seu limite deve ser exposto em termos de  $\text{grau}[x]$  e do número  $c$  de chamadas ao procedimento CASCADING-CUT.
- Suponha que chamamos PISANO-DELETE( $H, x$ ), e seja  $H'$  o heap de Fibonacci resultante da chamada. Considerando que o nó  $x$  não é uma raiz, limite o potencial de  $H'$  em termos de  $\text{grau}[x]$ ,  $c$ ,  $t(H)$  e  $m(H)$ .
- Conclua que o tempo amortizado de PISANO-DELETE não é assintoticamente melhor que o de FIB-HEAP-DELETE, mesmo quando  $x \neq \text{min}[H]$ .

### **20-2 Outras operações de heaps de Fibonacci**

Desejamos ampliar um heap de Fibonacci  $H$  para dar suporte a duas novas operações, sem mudar o tempo de execução amortizado de quaisquer outras operações de heaps de Fibonacci.

- a. A operação  $\text{FIB-HEAP-CHANGE-KEY}(H, x, k)$  troca a chave do nó  $x$  pelo valor  $k$ . Forneça uma implementação eficiente de  $\text{FIB-HEAP-CHANGE-KEY}$  e analise o tempo de execução amortizado de sua implementação para os casos nos quais  $k$  é maior que, menor que ou igual a  $\text{chave}[x]$ .
- b. Forneça uma implementação eficiente de  $\text{FIB-HEAP-PRUNE}(H, r)$ , que elimine  $\min(r, n[H])$  nós de  $H$ . A definição dos nós que serão eliminados deve ser arbitrária. Analise o tempo de execução amortizado de sua implementação. (*Sugestão:* Talvez você precise modificar a estrutura de dados e a função potencial.)

## Notas do capítulo

Os heaps de Fibonacci foram introduzidos por Fredman e Tarjan [98]. Seu artigo também descreve a aplicação de heaps de Fibonacci aos problemas de caminhos mais curtos de origem única, caminhos mais curtos de todos os pares, emparelhamento ponderado bipartido e o problema da árvore espalhada mínima.

Subseqüentemente, Driscoll, Gabow, Shrairman e Tarjan [81] desenvolveram “heaps relaxados” como uma alternativa aos heaps de Fibonacci. Existem duas variedades de heaps relaxados. Uma delas fornece os mesmos limites de tempo amortizado que os heaps de Fibonacci. A outra permite a execução de  $\text{DECREASE-KEY}$  no tempo do pior caso (não amortizado)  $O(1)$  e a execução de  $\text{EXTRACT-MIN}$  e  $\text{DELETE}$  no tempo do pior caso  $O(\lg n)$ . Os heaps relaxados também apresentam algumas vantagens em relação aos heaps de Fibonacci em algoritmos paralelos.

Veja também as notas do capítulo referentes ao Capítulo 6 para outras estruturas de dados que fornecem suporte para operações  $\text{DECREASE-KEY}$  rápidas quando a seqüência de valores retornados por chamadas de  $\text{EXTRACT-MIN}$  são monotonicamente crescentes com o tempo, e os dados são inteiros contidos em um intervalo específico.

---

# *Capítulo 21*

## *Estruturas de dados para conjuntos disjuntos*

Algumas aplicações envolvem o agrupamento de  $n$  elementos distintos em uma coleção de conjuntos disjuntos. Duas operações importantes são então encontrar o conjunto a que pertence um dado elemento e unir dois conjuntos. Este capítulo explora métodos para manter uma estrutura de dados que admite essas operações.

A Seção 21.1 descreve as operações aceitas por uma estrutura de dados de conjuntos disjuntos e apresenta uma aplicação simples. Na Seção 21.2, examinaremos uma implementação de lista ligada simples para conjuntos disjuntos. Uma representação mais eficiente com o uso de árvores enraizadas é dada na Seção 21.3. O tempo de execução com o uso da representação de árvore é linear para todas as finalidades práticas, mas é teoricamente superlinear. A Seção 21.4 define e discute uma função de crescimento muito rápido e sua inversa que cresce muito lentamente, o que se evidencia no tempo de execução de operações sobre a implementação baseada em árvore, e depois utiliza a análise amortizada para provar um limite superior apenas ligeiramente superlinear sobre o tempo de execução.

### **21.1 Operações de conjuntos disjuntos**

Uma *estrutura de dados de conjuntos disjuntos* mantém uma coleção  $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$  de conjuntos dinâmicos disjuntos. Cada conjunto é identificado por um *representante*, que é algum elemento do conjunto. Em algumas aplicações, não importa qual elemento é usado como representante; só nos importamos com o fato de que, se solicitarmos o representante de um conjunto dinâmico duas vezes sem modificar o conjunto entre as solicitações, devemos obter a mesma resposta ambas as vezes. Em outras aplicações, pode haver uma regra previamente especificada para escolher o representante, como a escolha do menor elemento no conjunto (supondo-se, é claro, que os elementos possam ser ordenados).

Como nas outras implementações de conjuntos dinâmicos que estudamos, cada elemento de um conjunto é representado por um objeto. Sendo  $x$  a representação de um objeto, desejamos dar suporte às operações relacionadas a seguir.

**MAKE-SET( $x$ )** cria um novo conjunto cujo único elemento (e portanto o representante) é apontado por  $x$ . Como os conjuntos são disjuntos, exigimos que  $x$  ainda não esteja em algum outro conjunto.

$\text{UNION}(x, y)$  une os conjuntos dinâmicos que contêm  $x$  e  $y$ , digamos  $S_x$  e  $S_y$ , em um novo conjunto que é a união desses dois conjuntos. Os dois conjuntos são admitidos como disjuntos antes da operação. O representante do conjunto resultante é algum elemento de  $S_x \cup S_y$ , embora muitas implementações de UNION escolham especificamente o representante de  $S_x$  ou  $S_y$  como o novo representante. Tendo em vista que exigimos que os conjuntos na coleção sejam disjuntos, “destruímos” os conjuntos  $S_x$  e  $S_y$ , removendo-os da coleção  $\mathcal{S}$ .

$\text{FIND-SET}(x)$  retorna um ponteiro para o representante do (único) conjunto que contém  $x$ .

Ao longo deste capítulo, analisaremos os tempos de execução de estruturas de dados de conjuntos disjuntos em termos de dois parâmetros:  $n$ , o número de operações MAKE-SET e  $m$ , o número total de operações MAKE-SET, UNION e FIND-SET. Tendo em vista que os conjuntos são disjuntos, cada operação UNION reduz o número de conjuntos em uma unidade. Então, depois de  $n - 1$  operações UNION, resta apenas um conjunto. Portanto, o número de operações UNION é no máximo  $n - 1$ . Observe também que, como as operações MAKE-SET estão incluídas no número total de operações  $m$ , temos  $m \geq n$ . Supomos que as  $n$  operações MAKE-SET são as primeiras  $n$  operações executadas.

## Uma aplicação de estruturas de dados de conjuntos disjuntos

Uma das muitas aplicações de estruturas de dados de conjuntos disjuntos surge na determinação dos componentes conexos de um grafo não orientado (ver Seção B.4). Por exemplo, a Figura 21.1(a) mostra um grafo com quatro componentes conexos.

O procedimento CONNECTED-COMPONENTS que se segue utiliza as operações de conjuntos disjuntos para calcular os componentes conexos de um grafo. Uma vez que CONNECTED-COMPONENTS tenha sido executado como uma etapa de pré-processamento, o procedimento SAME-COMPONENT responde a consultas que procuram saber se dois vértices estão no mesmo componente conectado.<sup>1</sup> (O conjunto de vértices de um grafo  $G$  é denotado por  $V[G]$  e o conjunto de arestas é denotado por  $E[G]$ .)

CONNECTED-COMPONENTS( $G$ )

- 1 **for** cada vértice  $v \in V[G]$
- 2   **do** MAKE-SET( $v$ )
- 3 **for** cada aresta  $(u, v) \in E[G]$
- 4   **do if** FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
- 5     **then** UNION( $u, v$ )

SAME-COMPONENT( $u, v$ )

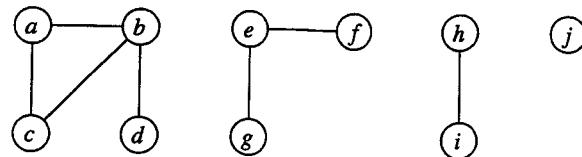
- 1 **if** FIND-SET( $u$ )  $=$  FIND-SET( $v$ )
- 2   **then return** TRUE
- 3   **else return** FALSE

O procedimento CONNECTED-COMPONENTS insere inicialmente cada vértice  $v$  em seu próprio conjunto. Em seguida, para cada aresta  $(u, v)$ , ele une os conjuntos que contêm  $u$  e  $v$ . Pelo Exercício 21.1-2, após todas as arestas serem processadas, dois vértices estão no mesmo componente conectado se e somente se os objetos correspondentes estão no mesmo conjunto. Desse modo, CONNECTED-COMPONENTS calcula conjuntos de tal modo que o procedimento

<sup>1</sup>Quando as arestas do grafo são “estáticas” – não se alteram com o tempo –, os componentes conectados podem ser calculados com maior rapidez pelo uso da pesquisa primeiro na profundidade (Exercício 22.3-11). Porém, às vezes as arestas são adicionadas “dinamicamente” e precisamos manter os componentes conectados à medida que cada aresta é adicionada. Nesse caso, a implementação dada aqui pode ser mais eficiente que a execução de uma nova pesquisa primeiro na profundidade para cada nova aresta.

SAME-COMPONENT pode determinar se dois vértices estão no mesmo componente conectado. A Figura 21.1(b) ilustra como os conjuntos disjuntos são calculados por CONNECTED-COMPONENTS.

Em uma implementação real deste algoritmo de componentes conexos, as representações do grafo e da estrutura de dados de conjuntos disjuntos precisariam fazer referência uma à outra. Isto é, um objeto representando um vértice conteria um ponteiro para o objeto de conjuntos disjuntos correspondente e *vice-versa*. Esses detalhes de programação dependem da linguagem de implementação, e não os examinaremos mais aqui.



(a)

| Aresta processada  | Coleção de conjuntos disjuntos |       |     |     |         |     |     |       |     |     |
|--------------------|--------------------------------|-------|-----|-----|---------|-----|-----|-------|-----|-----|
| conjuntos iniciais | {a}                            | {b}   | {c} | {d} | {e}     | {f} | {g} | {h}   | {i} | {j} |
| (b,d)              | {a}                            | {b,d} | {c} |     | {e}     | {f} | {g} | {h}   | {i} | {j} |
| (e,g)              | {a}                            | {b,d} | {c} |     | {e,g}   | {f} |     | {h}   | {i} | {j} |
| (a,c)              | {a,c}                          | {b,d} |     |     | {e,g}   | {f} |     | {h}   | {i} | {j} |
| (h,i)              | {a,c}                          | {b,d} |     |     | {e,g}   | {f} |     | {h,i} |     | {j} |
| (a,b)              | {a,b,c,d}                      |       |     |     | {e,g}   | {f} |     | {h,i} |     | {j} |
| (e,f)              | {a,b,c,d}                      |       |     |     | {e,f,g} |     |     | {h,i} |     | {j} |
| (b,c)              | {a,b,c,d}                      |       |     |     | {e,f,g} |     |     | {h,i} |     | {j} |

(b)

FIGURA 21.1 (a) Um grafo com quatro componentes conexos: {a, b, c, d}, {e, f, g}, {h, i} e {j}. (b) A coleção de conjuntos disjuntos após o processamento de cada aresta

## Exercícios

### 21.1-1

Suponha que CONNECTED-COMPONENTS seja executado sobre o grafo não orientado  $G = (V, E)$ , onde  $V = \{a, b, c, d, e, f, g, h, i, j, k\}$  e as arestas de  $E$  são processadas na seguinte ordem:  $(d, i)$ ,  $(f, k)$ ,  $(g, i)$ ,  $(b, g)$ ,  $(a, b)$ ,  $(i, j)$ ,  $(d, k)$ ,  $(b, j)$ ,  $(d, f)$ ,  $(g, j)$ ,  $(a, e)$ ,  $(i, d)$ . Liste os vértices em cada componente conectado, depois de cada iteração das linhas 3 a 5.

### 21.1-2

Mostre que, depois de todas as arestas serem processadas por CONNECTED-COMPONENTS, dois vértices estão no mesmo componente conectado se e somente se eles estão no mesmo conjunto.

### 21.1-3

Durante a execução de CONNECTED-COMPONENTS sobre um grafo não orientado  $G = (V, E)$  com  $k$  componentes conexos, quantas vezes FIND-SET é chamado? Quantas vezes UNION é chamado? Expressse suas respostas em termos de  $|V|$ ,  $|E|$  e  $k$ .

## 21.2 Representação de conjuntos disjuntos por listas ligadas

Um modo simples de implementar uma estrutura de dados de conjuntos disjuntos é representar cada conjunto por uma lista ligada. O primeiro objeto em cada lista ligada serve como repre-

sentante do seu conjunto. Cada objeto na lista ligada contém um elemento de conjunto, um ponteiro para o objeto contendo o próximo elemento de conjunto e um ponteiro de volta para o representante. A Figura 21.2(a) mostra dois conjuntos. Dentro de cada lista ligada, os objetos podem aparecer em qualquer ordem (desde que seja obedecida nossa hipótese de que o primeiro objeto em cada lista é o representante).

Com essa representação de lista ligada, tanto MAKE-SET quanto FIND-SET são fáceis, exigindo o tempo  $O(1)$ . Para executar  $\text{MAKE-SET}(x)$ , criamos uma nova lista ligada cujo único objeto é  $x$ . Para  $\text{FIND-SET}(x)$ , simplesmente retornamos o ponteiro de volta para o representante.

### Uma implementação simples de união

A implementação mais simples da operação UNION usando a representação de conjunto de lista ligada toma significativamente mais tempo que MAKE-SET ou FIND-SET. Como mostra a Figura 21.2(b), executamos UNION( $x, y$ ) anexando a lista de  $x$  ao final da lista de  $y$ . Usamos o ponteiro *final* para a lista de  $y$ , a fim de encontrar rapidamente onde acrescentar a lista de  $x$ . O representante do novo conjunto é o elemento que era originalmente o representante do conjunto contendo  $y$ . Infelizmente, devemos atualizar o ponteiro para o representante que corresponde a cada objeto presente na lista original de  $x$ , o que torna o tempo linear no comprimento da lista de  $x$ .

De fato, não é difícil obter uma seqüência de  $n$  operações que exija o tempo  $\Theta(n^2)$ . Suponha que temos objetos  $x_1, x_2, \dots, x_n$ . Executamos a seqüência de  $n$  operações MAKE-SET seguidas por  $n - 1$  operações UNION mostrada na Figura 21.3, de forma que  $m = 2n - 1$ . Gastamos o tempo  $\Theta(n)$  executando as  $n$  operações MAKE-SET. Pelo fato da  $i$ -ésima operação UNION atualizar  $i$  objetos, o número total de objetos atualizados por todas as  $n - 1$  operações UNION é

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

O número total de operações é  $2n - 1$  e, portanto, cada operação exige em média o tempo  $\Theta(n)$ . Isto é, o tempo amortizado de uma operação é  $\Theta(n)$ .

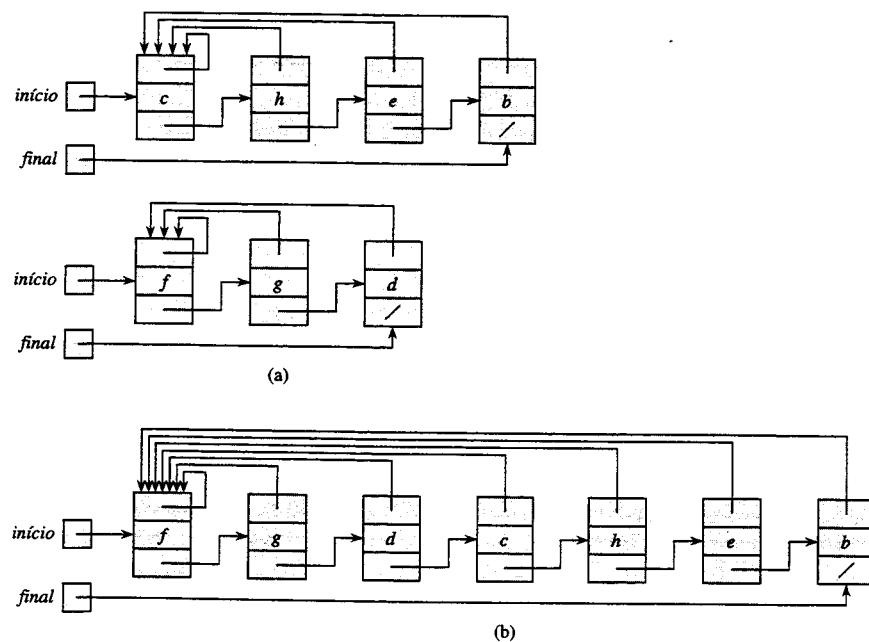


FIGURA 21.2 (a) Representações de dois conjuntos por listas ligadas. Um deles contém os objetos *b*, *c*, *e* e *b*, com *c* como representante, e o outro contém os objetos *d*, *f* e *g*, sendo *f* o representante. Cada objeto na lista contém um elemento de conjunto, um ponteiro para o próximo objeto na lista e um ponteiro de volta ao primeiro objeto na lista, o qual é o representante. (b) O resultado de UNION(*e, g*). O representante do conjunto resultante é *f*

| Operação                | Número de objetos atualizados |
|-------------------------|-------------------------------|
| MAKE-SET( $x_1$ )       | 1                             |
| MAKE-SET( $x_2$ )       | 1                             |
| .                       | .                             |
| MAKE-SET( $x_n$ )       | 1                             |
| UNION( $x_1, x_2$ )     | 1                             |
| UNION( $x_2, x_3$ )     | 2                             |
| UNION( $x_3, x_4$ )     | 3                             |
| .                       | .                             |
| UNION( $x_{n-1}, x_n$ ) | $n - 1$                       |

FIGURA 21.3 Uma seqüência de  $2n - 1$  operações sobre  $n$  objetos que demora o tempo  $\Theta(n^2)$ , ou o tempo  $\Theta(n)$  por operação em média, usando a representação de conjuntos por listas ligadas e a implementação simples de UNION

### Uma heurística de união ponderada

No pior caso, a implementação anterior do procedimento UNION exige um tempo médio  $\Theta(n)$  por chamada devido à possibilidade de estarmos anexando uma lista mais longa a uma lista mais curta; devemos atualizar o ponteiro para o representante em cada elemento da lista mais longa. Em vez disso, suponha que cada representante inclua também o comprimento da lista (o qual é mantido com facilidade), e que sempre seja feita a anexação da lista menor à mais longa, com os vínculos rompidos arbitrariamente. Com essa **heurística de união ponderada** simples, uma única operação UNION ainda pode demorar o tempo  $\Omega(n)$ , se ambos os conjuntos têm  $\Omega(n)$  elementos. Porém, como mostra o teorema a seguir, uma seqüência de  $m$  operações MAKE-SET, UNION e FIND-SET,  $n$  das quais são operações MAKE-SET, demora o tempo  $O(m + n \lg n)$ .

#### Teorema 21.1

Usando a representação de lista ligada de conjuntos disjuntos e a heurística de união ponderada, uma seqüência de  $m$  operações MAKE-SET, UNION e FIND-SET,  $n$  das quais são operações MAKE-SET, demora o tempo  $O(m + n \lg n)$ .

**Prova** Começamos calculando, para cada objeto em um conjunto de tamanho  $n$ , um limite superior sobre o número de vezes que o ponteiro do objeto de volta para o representante é atualizado. Considere um objeto fixo  $x$ . Sabemos que cada vez que o ponteiro representante de  $x$  foi atualizado,  $x$  deve ter começado no conjunto menor. Então, na primeira vez em que o ponteiro do representante de  $x$  foi atualizado, no conjunto resultante deviam existir pelo menos 2 elementos. De modo semelhante, na vez seguinte em que o ponteiro representante de  $x$  foi atualizado, o conjunto resultante devia ter no mínimo 4 elementos. Continuando assim, observamos que para qualquer  $k \# n$ , depois que o ponteiro do representante de  $x$  foi atualizado  $\Omega(n)$  vezes, o conjunto resultante deve ter pelo menos  $k$  elementos. Tendo em vista que o maior conjunto tem no máximo  $n$  elementos, o ponteiro representante de cada objeto foi atualizado no máximo  $\Omega(n)$  vezes sobre todas as operações UNION. Também devemos levar em conta a atualização dos ponteiros *início* e *final* e os comprimentos de listas, que demoram apenas o tempo  $\Theta(1)$  por operação UNION. O tempo total usado na atualização dos  $n$  objetos é portanto  $O(n \lg n)$ .

O tempo para a seqüência inteira de  $m$  operações segue-se com facilidade. Cada operação MAKE-SET e FIND-SET demora o tempo  $O(1)$ , e existem  $O(m)$  delas. O tempo total para a seqüência inteira é portanto  $O(m + n \lg n)$ . ■

## Exercícios

### 21.2-1

Escreva pseudocódigo para MAKE-SET, FIND-SET e UNION usando a representação de lista ligada e a heurística de união ponderada. Suponha que cada objeto  $x$  tenha um atributo  $rep[x]$  apontando para o representante do conjunto que contém  $x$ , e que cada conjunto  $S$  tem atributos  $íncio[S]$ ,  $final[S]$  e  $tamanho[S]$  (que é igual ao comprimento da lista).

### 21.2-2

Mostre a estrutura de dados resultante e as respostas retornadas pelas operações FIND-SET no programa a seguir. Use a representação de lista ligada com a heurística de união ponderada.

```
1 for i ← 1 to 16
2   do MAKE-SET( $x_i$ )
3 for i ← 1 to 15 by 2
4   do UNION( $x_i, x_{i+1}$ )
5 for i ← 1 to 13 by 4
6   do UNION( $x_i, x_{i+2}$ )
7 UNION( $x_1, x_5$ )
8 UNION( $x_{11}, x_{13}$ )
9 UNION( $x_1, x_{10}$ )
10 FIND-SET( $x_2$ )
11 FIND-SET( $x_9$ )
```

Suponha que, se os conjuntos que contêm  $x_i$  e  $x_j$  apresentam o mesmo tamanho, então a operação UNION( $x_i, x_j$ ) acrescenta a lista de  $x_j$  à lista de  $x_i$ .

### 21.2-3

Adapte a prova agregada do Teorema 21.1 com a finalidade de obter limites de tempo amortizados  $O(1)$  para MAKE-SET e FIND-SET, e  $O(\lg n)$  para UNION, utilizando a representação de lista ligada e a heurística de união ponderada.

### 21.2-4

Forneça um limite assintótico restrito sobre o tempo de execução da seqüência de operações da Figura 21.3, supondo a representação de lista ligada e a heurística de união ponderada.

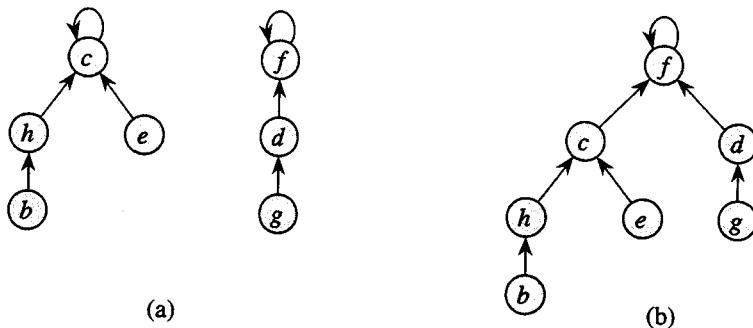
### 21.2-5

Sugira uma mudança simples para o procedimento UNION para a representação de lista ligada que elimine a necessidade de se manter o ponteiro *final* para o último objeto em cada lista. Independente de ser utilizada ou não a heurística de união ponderada, a alteração não deve mudar o tempo de execução assintótico do procedimento UNION. (Sugestão: Em lugar de acrescentar uma lista a outra, reúna as duas listas.)

## 21.3 Florestas de conjuntos disjuntos

Em uma implementação mais rápida de conjuntos disjuntos, representamos conjuntos por árvores enraizadas, com cada nó contendo um elemento e cada árvore representando um conjunto. Em uma **floresta de conjuntos disjuntos**, ilustrada na Figura 21.4(a), cada elemento aponta apenas para seu pai. A raiz de cada árvore contém o representante e é seu próprio pai. Como veremos, embora os algoritmos diretos que utilizam essa representação não sejam mais rápidos | 403

que aqueles que usam a representação de lista ligada, introduzindo duas heurísticas – “união por ordenação” e “compressão de caminho” – podemos obter a estrutura de dados de conjuntos disjuntos assintoticamente mais rápida conhecida.



**FIGURA 21.4** Uma floresta de conjuntos disjuntos. (a) Duas árvores representando os dois conjuntos da Figura 21.2. A árvore da esquerda representa o conjunto  $\{b, c, e, b\}$ , com  $c$  como representante, e a árvore da direita representa o conjunto  $\{d, f, g\}$ , com  $f$  como representante. (b) O resultado de  $\text{UNION}(e, g)$

Executamos as três operações de conjuntos disjuntos como a seguir. Uma operação **MAKE-SET** simplesmente cria uma árvore com apenas um nó. Executamos uma operação **FIND-SET** seguindo ponteiros de pais até encontrarmos a raiz da árvore. Os nós visitados nesse caminho em direção à raiz constituem o **caminho de localização**. Uma operação **UNION**, mostrada na Figura 21.4(b), faz a raiz de uma árvore apontar para a raiz da outra.

### Heurísticas para melhorar o tempo de execução

Até agora, não melhoramos a implementação de listas ligadas. Uma sequência de  $n - 1$  operações **UNION** pode criar uma árvore que é apenas uma cadeia linear de  $n$  nós. Contudo, usando duas heurísticas, podemos conseguir um tempo de execução quase linear no número total de operações  $m$ .

A primeira heurística, **união por ordenação**, é semelhante à heurística de união ponderada que usamos com a representação de listas ligadas. A idéia é fazer a raiz da árvore com menor número de nós apontar para a raiz da árvore com mais nós. Em vez de controlar de modo explícito o tamanho da subárvore com raiz em cada nó, usaremos uma abordagem que facilita a análise. Para cada nó, mantemos uma **ordem** que é um limite superior sobre a altura do nó. Na união por ordenação, a raiz com menor ordem é levada a apontar para a raiz com maior ordem durante uma operação **UNION**.

A segunda heurística, **compressão de caminho**, também é bastante simples e muito eficiente. Como mostra a Figura 21.5, nós a usaremos durante operações **FIND-SET** para fazer cada nó no caminho de localização apontar diretamente para a raiz. A compressão de caminho não altera quaisquer ordens.

### Pseudocódigo para florestas de conjuntos disjuntos

Para implementar uma floresta de conjuntos disjuntos com a heurística de união por ordenação, devemos controlar as ordens. Com cada nó  $x$ , mantemos o valor inteiro  $\text{ordem}[x]$ , que é um limite superior sobre a altura de  $x$  (o número de arestas no caminho mais longo entre  $x$  e uma folha descendente). Quando um conjunto unitário é criado por **MAKE-SET**, a ordem inicial do único nó na árvore correspondente é 0. Cada operação **FIND-SET** deixa todas as ordens inalteradas. Quando aplicamos **UNION** a duas árvores, há dois casos, dependendo do fato das raízes terem ordem igual ou não. Se as raízes têm ordem desigual, fazemos da raiz de ordem mais alta o pai da raiz de ordem mais baixa, mas as próprias ordens permanecem inalteradas. Se, em vez disso, as raízes têm ordens iguais, escolhemos arbitrariamente uma das raízes como o pai e incrementamos sua ordem.

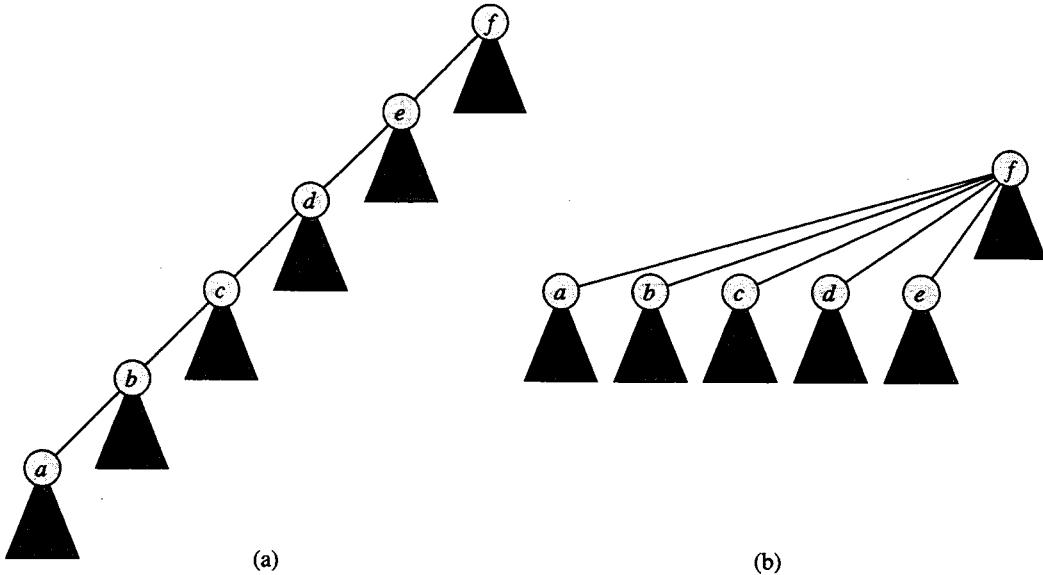


FIGURA 21.5 Compressão de caminho durante a operação FIND-SET. As setas e os autoloops nas raízes foram omitidos. (a) Uma árvore representando um conjunto antes da execução de FIND-SET( $a$ ). Os triângulos representam subárvores cujas raízes são os nós mostrados. Cada nó tem um ponteiro para seu pai. (b) O mesmo conjunto após a execução de FIND-SET( $a$ ). Agora, cada nó no caminho de localização aponta diretamente para a raiz

Vamos colocar esse método em pseudocódigo. Designamos o pai do nó  $x$  por  $p[x]$ . O procedimento LINK, uma sub-rotina chamada por UNION, usa ponteiros para duas raízes como entradas.

**MAKE-SET( $x$ )**

- 1  $p[x] \leftarrow x$
- 2  $ordem[x] \leftarrow 0$

**UNION( $x, y$ )**

- 1 **LINK(FIND-SET( $x$ ), FIND-SET( $y$ ))**

**LINK( $x, y$ )**

- 1 **if**  $ordem[x] > ordem[y]$
- 2 **then**  $p[y] \leftarrow x$
- 3 **else**  $p[x] \leftarrow y$
- 4 **if**  $ordem[x] = ordem[y]$
- 5 **then**  $ordem[y] \leftarrow ordem[y] + 1$

O procedimento FIND-SET com compressão de caminho é bastante simples.

**FIND-SET( $x$ )**

- 1 **if**  $x \neq p[x]$
- 2 **then**  $p[x] \leftarrow \text{FIND-SET}(p[x])$
- 3 **return**  $p[x]$

O procedimento FIND-SET é um *método de duas passagens*: ele efetua uma passagem para cima no caminho de localização, a fim de encontrar a raiz, e depois faz uma segunda passagem de volta para baixo no caminho de localização com o objetivo de atualizar cada nó, de modo que ele aponte diretamente para a raiz. Cada chamada de FIND-SET( $x$ ) retorna  $p[x]$  na linha 3. Se  $x$  é a raiz, então a linha 2 não é executada e  $p[x] = x$  é retornado. Esse é o caso em que a recur-

são chega à parte inferior. Caso contrário, a linha 2 é executada, e a chamada recursiva com o parâmetro  $p[x]$  retorna um ponteiro para a raiz. A linha 2 atualiza o nó  $x$  de modo a apontar diretamente para a raiz, e esse ponteiro é retornado na linha 3.

### Efeito das heurísticas sobre o tempo de execução

Separadamente, a união por ordenação ou a compressão de caminho melhora o tempo de execução das operações sobre florestas de conjuntos disjuntos, e a melhora é ainda maior quando as duas heurísticas são usadas em conjunto. Sozinha, a união por ordenação produz o mesmo tempo de execução de  $O(m \lg n)$  (veja o Exercício 21.4-4), e esse limite é restrito (veja o Exercício 21.30-3). Embora não o demonstremos aqui, se existem  $n$  operações MAKE-SET (e, consequentemente, no máximo  $n - 1$  operações UNION) e  $f$  operações FIND-SET, a heurística de compressão de caminho fornece sozinha um tempo de execução do pior caso igual a  $\Theta(n + f \cdot (1 + \log_2 + f/n n))$ .

Quando usamos a união por ordenação e a compressão de caminho, o tempo de execução do pior caso é  $O(m \alpha(n))$ , onde  $\alpha(n)$  é uma função *muito* lentamente crescente que definimos na Seção 21.4. Em qualquer aplicação concebível de uma estrutura de dados de conjuntos disjuntos,  $\alpha(n) \leq 4$ ; desse modo, podemos considerar o tempo de execução linear em  $m$  para todas as situações práticas. Na Seção 21.4, provaremos esse limite superior.

## Exercícios

### 21.3-1

Faça o Exercício 21.2-2 usando uma floresta de conjuntos disjuntos com união por ordenação e compressão de caminho.

### 21.3-2

Escreva uma versão não recursiva de FIND-SET com compressão de caminho.

### 21.3-3

Forneça uma seqüência de  $m$  operações MAKE-SET, UNION e FIND-SET,  $n$  das quais são operações MAKE-SET, que demore o tempo  $\Omega(m \lg n)$  quando usarmos somente a união por ordenação.

### 21.3-4 \*

Mostre que qualquer seqüência de  $m$  operações MAKE-SET, FIND-SET e LINK, onde todas as operações LINK aparecem antes de qualquer das operações FIND-SET, demora apenas o tempo  $O(m)$  se são usadas a compressão de caminho e a união por ordenação. O que acontecerá na mesma situação se for usada apenas a heurística de compressão de caminho?

## ★ 21.4 Análise da união por ordenação com compressão de caminho

Como observamos na Seção 21.3, o tempo de execução da heurística combinada de união por ordenação e compressão de caminho é  $O(m \alpha(n))$  para  $m$  operações de conjuntos disjuntos sobre  $n$  elementos. Nesta seção, examinaremos a função  $\alpha$  para ver apenas com que lentidão ela cresce. Em seguida, provaremos esse tempo de execução empregando o método potencial de análise amortizada.

### Uma função que cresce muito rapidamente e sua inversa que cresce muito lentamente

Para inteiros  $k \geq 0$  e  $j \geq 1$ , definimos a função  $A_k(j)$  como

$$A_k(j) = \begin{cases} j+1 & \text{se } k=0, \\ A_{k-1}^{(j+1)} & \text{se } k \geq 1, \end{cases}$$

onde a expressão  $A_{k-1}^{(j+1)}(j)$  utiliza a notação de iteração funcional dada na Seção 3.2. Especificamente,  $A_{k-1}^{(0)}(j) = j$  e  $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$  para  $i \geq 1$ . Faremos referência ao parâmetro  $k$  como o **nível** da função  $A$ .

A função  $A_k(j)$  aumenta estritamente com  $j$  e  $k$ . Para ver exatamente com que rapidez essa função cresce, primeiro obtemos expressões de forma fechada para  $A_1(j)$  e  $A_2(j)$ .

### Lema 21.2

Para qualquer inteiro  $j \geq 1$ , temos  $A_1(j) = 2j + 1$ .

**Prova** Primeiro usamos a indução sobre  $i$  para mostrar que  $A_0^{(i)}(j) = j + i$ . Para o caso básico, temos  $A_0^{(0)}(j) = j = j + 0$ . Para a etapa indutiva, suponha que  $A_0^{(i-1)}(j) = j + (i - 1)$ . Então,  $A_0^{(i)}(j) = A_0(A_0^{(i-1)}(j)) = (j + (i - 1)) + 1 = j + i$ . Finalmente, notamos que  $A_1(j) = A_0^{(j+1)}(j) = j + (j + 1) = 2j + 1$ . ■

### Lema 21.3

Para qualquer inteiro  $j \geq 1$ , temos  $A_2(j) = 2^{j+1}(j + 1) - 1$ .

**Prova** Primeiro usamos a indução sobre  $i$  para mostrar que  $A_1^{(i)}(j) = 2^i(j + 1) - 1$ . Para o caso básico, temos  $A_1^{(0)}(j) = j = 2^0(j + 1) - 1$ . Para a etapa indutiva, suponha que  $A_1^{(i-1)}(j) = 2^{i-1}(j + 1) - 1$ . Então,  $A_1^{(i)}(j) = A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j + 1) - 1) = 2 \cdot (2^{i-1}(j + 1) - 1) + 1 = 2^i(j + 1) - 2 + 1 = 2^i(j + 1) - 1$ . Finalmente, observamos que  $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j + 1) - 1$ .

Agora podemos ver com que rapidez  $A_k(j)$  cresce, simplesmente examinando  $A_k(1)$  para os níveis  $k = 0, 1, 2, 3, 4$ . Da definição de  $A_0(k)$  e dos lemas anteriores, temos  $A_0(1) = 1 + 1 = 2$ ,  $A_1(1) = 2 \cdot 1 + 1 = 3$  e  $A_2(1) = 2^{1+1} \cdot (1 + 1) - 1 = 7$ . Também temos

$$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A^2(A^2(1)) \\ &= A^2(7) \\ &= 2^8 \cdot 8 - 1 \\ &= 2^{11} - 1 \\ &= 2047 \end{aligned}$$

e

$$\begin{aligned} A_4(1) &= A_3^{(2)}(1) \\ &= A_3(A_3(1)) \\ &= A_3(2047) \\ &= A_2^{(2048)}(2047) \end{aligned}$$

$$\begin{aligned} &\gg A_2(2047) \\ &= 2^{2048} \cdot 2048 - 1 \\ &= 2^{2048} \\ &= (2^4)^{512} \\ &= 16^{512} \\ &\gg 10^{80}, \end{aligned}$$

que é o número estimado de átomos no universo observável.

Definimos a inversa da função  $A_k(n)$ , para inteiros  $n \geq 0$ , por

$$\alpha(n) = \min \{k : A_k(1) \geq n\}.$$

Textualmente,  $\alpha(n)$  é o mais baixo nível  $k$  para o qual  $A_k(1) \geq n$ . Pelos valores anteriores de  $A_k(1)$ , vemos que

$$\alpha(n) = \begin{cases} 0 & \text{para } 0 \leq n \leq 2, \\ 1 & \text{para } n = 3, \\ 2 & \text{para } 4 \leq n \leq 7, \\ 3 & \text{para } 8 \leq n \leq 2047, \\ 4 & \text{para } 2048 \leq n \leq A_4(1). \end{cases}$$

Apenas para valores impraticavelmente grandes de  $n$  (maiores que  $A_4(1)$ , um número enorme) temos  $\alpha(n) > 4$  e, desse modo,  $\alpha(n) \leq 4$  para todas as finalidades práticas.

## Propriedades de ordenações

No restante desta seção, provaremos um limite  $O(m\alpha(n))$  sobre o tempo de execução das operações de conjuntos disjuntos com união por ordenação e compressão de caminho. Para provar esse limite, primeiro demonstraremos algumas propriedades simples de ordenações.

### Lema 21.4

Para todos os nós  $x$ , temos  $ordem[x] \leq ordem[p[x]]$ , com desigualdade estrita se  $x \neq p[x]$ . O valor de  $ordem[x]$  é inicialmente 0 e aumenta com o tempo até  $x = p[x]$ ; daí em diante,  $ordem[x]$  não muda. O valor de  $ordem[p[x]]$  é uma função monotonicamente crescente do tempo.

**Prova** A prova é uma indução direta sobre o número de operações, usando-se as implementações de MAKE-SET, UNION e FIND-SET que aparecem na Seção 21.3. Vamos deixá-la para o Exercício 21.4-1. ■

### Corolário 21.5

À medida que seguimos o caminho de qualquer nó em direção a uma raiz, as ordens de nós aumentam estritamente. ■

### Lema 21.6

Todo nó tem ordem no máximo igual a  $n - 1$ .

**Prova** A ordem de cada nó começa em 0, e aumenta apenas em operações LINK. Pelo fato de existirem no máximo  $n - 1$  operações UNION, também há no máximo  $n - 1$  operações LINK. Como cada operação LINK deixa todas as ordens como estão ou aumenta a ordem de algum nó em 1, todas as ordens são no máximo  $n - 1$ . ■

O Lema 21.6 fornece um limite fraco sobre ordens. De fato, todo nó tem ordem no máximo  $\lfloor \lg n \rfloor$  (veja o Exercício 21.4-2). Porém, o limite menos restrito do Lema 21.6 bastará para nossos propósitos.

## Como provar o limite de tempo

Usaremos o método de potencial da análise amortizada (consulte a Seção 17.3) para provar o limite de tempo  $O(m\alpha(n))$ . Na realização da análise amortizada, é conveniente supor que invocamos a operação LINK em lugar da operação UNION. Isto é, como os parâmetros do procedimento LINK são ponteiros para duas raízes, supomos que as operações FIND-SET apropriadas são executadas separadamente. O lema a seguir mostra que, mesmo se contarmos as operações FIND-SET extras induzidas por chamadas a UNION, o tempo de execução assintótico permanecerá inalterado.

### Lema 21.7

Vamos supor que convertemos uma seqüência  $S'$  de  $m'$  operações MAKE-SET, UNION e FIND-SET

408 | em uma seqüência  $S$  de  $m$  operações MAKE-SET, LINK e FIND-SET, transformando cada UNION

em duas operações FIND-SET seguidas por uma operação LINK. Então, se a seqüência  $S$  for executada no tempo  $O(m' \alpha(n))$ , a seqüência  $S'$  será executada no tempo  $O(m' \alpha(n))$ .

**Prova** Tendo em vista que cada operação UNION na seqüência  $S'$  é convertida em três operações em  $S$ , temos  $m' \leq m \leq 3m'$ . Como  $m = O(m')$ , um limite de tempo  $O(m \alpha(n))$  para a seqüência convertida  $S$  implica um limite de tempo  $O(m' \alpha(n))$  para a seqüência original  $S'$ . ■

No restante desta seção, vamos supor que a seqüência inicial de  $m'$  operações MAKE-SET, UNION e FIND-SET tenha sido convertida em uma seqüência de  $m$  operações MAKE-SET, LINK e FIND-SET. Agora, provaremos um limite de tempo  $O(m \alpha(n))$  para a seqüência convertida e apelaremos para o Lema 21.6 com o objetivo de provar o tempo de execução  $O(m' \alpha(n))$  da seqüência original de  $m'$  operações.

## Função potencial

A função potencial que usamos atribui um potencial  $\phi_q(x)$  a cada nó  $x$  na floresta de conjuntos disjuntos após  $q$  operações. Somamos os potenciais de nós para formar o potencial da floresta inteira:  $\Phi_q = \sum_x \phi_q(x)$ , onde  $\Phi_q$  denota o potencial da floresta após  $q$  operações. A floresta está vazia antes da primeira operação, e definimos arbitrariamente  $\Phi_0 = 0$ . Nenhum potencial  $\Phi_q$  poderá ser negativo.

O valor de  $\phi_q(x)$  depende do fato de  $x$  ser ou não uma raiz da árvore após a  $q$ -ésima operação. Se ela for, ou se  $ordem[x] = 0$ , então  $\Phi_q(x) = \alpha(n) \cdot ordem[x]$ .

Agora suponha que, após a  $q$ -ésima operação,  $x$  não é uma raiz e  $ordem[x] \geq 1$ . Precisamos definir duas funções auxiliares em  $x$  antes de podermos definir  $\Phi_q(x)$ . Primeiro, definimos

$$\text{nível}(x) = \max \{k : ordem[p[x]] \geq A_k(ordem[x])\}.$$

Isto é,  $\text{nível}(x)$  é o maior nível  $k$  para o qual  $A_k$ , aplicada à ordem  $x$ , não é maior que a ordem do pai de  $x$ .

Afirmamos que

$$0 \leq \text{nível}(x) < \alpha(n), \quad (21.1)$$

o que veremos a seguir. Temos

$$\begin{aligned} ordem[p[x]] &\geq ordem[x] + 1 && (\text{pelo Lema 21.4}) \\ &= A_0(ordem[x]) && (\text{pela definição de } A_0(j)), \end{aligned}$$

o que implica que  $\text{nível}(x) \geq 0$ , e temos

$$\begin{aligned} A_{\alpha(n)}(ordem[x]) &\geq A_{\alpha(n)}(1) && (\text{porque } A_k(j) \text{ é estritamente crescente}) \\ &\geq n && (\text{pela definição de } \alpha(n)) \\ &> ordem[p[x]] && (\text{pelo Lema 21.6}), \end{aligned}$$

o que implica que  $\text{nível}(x) < \alpha(n)$ . Observe que, pelo fato de  $ordem[p[x]]$  aumentar monotonicamente com o tempo, o mesmo ocorre com  $\text{nível}(x)$ .

A segunda função auxiliar é

$$\text{iter}(x) = \max \{i : ordem[p[x]] \geq A_{\text{nível}(x)}^{(i)}(ordem[x])\}$$

Isto é,  $\text{iter}(x)$  é o maior número de vezes que podemos aplicar a  $A_{\text{nível}(x)}$ , iterativamente, aplicada inicialmente à ordem de  $x$ , antes de obtermos um valor maior que a ordem do pai de  $x$ . Afirmamos que

$$1 \leq \text{iter}(x) \leq \text{ordem}[x], \quad (21.2)$$

o que veremos a seguir. Temos

$$\begin{aligned} \text{ordem}[p[x]] &\geq A_{\text{nível}(x)}(\text{ordem}[x]) && (\text{pela definição de nível}(x)) \\ &= A_{\text{nível}(x)}^{(i)}(\text{ordem}[x]) && (\text{pela definição de iteração funcional}), \end{aligned}$$

o que implica que  $\text{iter}(x) \geq 1$ , e temos

$$\begin{aligned} A_{\text{nível}(x)}^{(\text{ordem}[x]+1)}(\text{ordem}[x]) &= A_{\text{nível}(x)+1}(\text{ordem}[x]) && (\text{pela definição de } A_k(j)) \\ &> (\text{ordem}[p[x]]) && (\text{pela definição de nível}(x)), \end{aligned}$$

o que implica que  $\text{iter}(x) \leq \text{ordem}[x]$ . Observe que, como  $\text{ordem}[p[x]]$  cresce monotonicamente com o tempo, para  $\text{iter}(x)$  diminuir, nível( $x$ ) tem de aumentar. Desde que nível( $x$ ) permaneça inalterado,  $\text{iter}(x)$  deve crescer ou permanecer inalterado.

Com essas funções auxiliares estabelecidas, estamos prontos para definir o potencial do nó  $x$  após  $q$  operações:

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot \text{ordem}[x] & \text{se } x \text{ é uma raiz ou } \text{ordem}[x] = 0, \\ (\alpha(n) - \text{nível}(x)) \cdot \text{ordem}[x] - \text{iter}(x) & \text{se } x \text{ não é uma raiz e } \text{ordem}[x] \geq 1. \end{cases}$$

Os dois lemas a seguir fornecem propriedades úteis de potenciais de nós.

### Lema 21.8

Para todo nó  $x$ , e para todos os números de operações  $q$ , temos

$$0 \leq \phi_q(x) \leq \alpha(n) \cdot \text{ordem}[x].$$

**Prova** Se  $x$  é uma raiz ou  $\text{ordem}[x] = 0$ , então  $\phi_q(x) = \alpha(n) \cdot \text{ordem}[x]$  por definição. Agora, suponha que  $x$  não seja uma raiz e que  $\text{ordem}[x] \geq 1$ . Obtemos um limite inferior sobre  $\phi_q(x)$  maximizando nível( $x$ ) e iter( $x$ ). Pelo limite (21.1), nível( $x$ )  $\leq \alpha(n) - 1$  e, pelo limite (21.2) iter( $x$ )  $\leq \text{ordem}[x]$ . Desse modo,

$$\begin{aligned} \phi_q(x) &\geq (\alpha(n) - (\alpha(n) - 1)) \cdot \text{ordem}[x] - \text{ordem}[x] \\ &= \text{ordem}[x] - \text{ordem}[x] \\ &= 0. \end{aligned}$$

De modo semelhante, obtemos um limite superior sobre  $\phi_q(x)$  minimizando nível( $x$ ) e iter( $x$ ). Pelo limite (21.1), nível( $x$ )  $\geq 0$  e, pelo limite (21.2), iter( $x$ )  $\geq 1$ . Portanto,

$$\begin{aligned} \phi_q(x) &\leq (\alpha(n) - 0) \cdot \text{ordem}[x] - 1 \\ &= \alpha(n) \cdot \text{ordem}[x] - 1 \\ &< \alpha(n) \cdot \text{ordem}[x]. \end{aligned}$$

■

## Mudanças potenciais e custos amortizados de operações

Agora, estamos prontos para examinar como as operações de conjuntos disjuntos afetam os potenciais de nós. Com uma compreensão da mudança no potencial devido a cada operação, podemos determinar o custo amortizado de cada operação.

### Lema 21.9

Seja  $x$  um nó que não é uma raiz, e suponha que a  $q$ -ésima operação seja LINK ou FIND-SET. Então, após a  $q$ -ésima operação,  $\phi_q(x) \leq \phi_{q-1}(x)$ . Além disso, se  $ordem[x] \geq 1$  e  $nível(x)$  ou  $iter(x)$  mudar devido à  $q$ -ésima operação, então  $\phi_q(x) \leq \phi_{q-1}(x) - 1$ . Isto é, o potencial de  $x$  não pode aumentar e, se ele tem ordem positiva e  $nível(x)$  ou  $iter(x)$  mudar, o potencial de  $x$  cairá pelo menos uma unidade.

**Prova** Como  $x$  não é uma raiz, a  $q$ -ésima operação não muda  $ordem[x]$  e, como  $n$  não muda após as  $n$  operações iniciais MAKE-SET,  $\alpha(n)$  também permanece inalterada. Conseqüentemente, esses componentes da fórmula para o potencial de  $x$  permanecem os mesmos após a  $q$ -ésima operação. Se  $ordem[x] = 0$ , então  $\phi_q(x) = \phi_{q-1}(x) = 0$ . Agora, suponha que  $ordem[x] \geq 1$ .

Lembre-se de que  $nível(x)$  aumenta monotonicamente com o tempo. Se a  $q$ -ésima operação deixar  $nível(x)$  inalterado, então  $iter(x)$  aumentará ou permanecerá inalterado. Se tanto  $nível(x)$  quanto  $iter(x)$  são inalterados, então  $\phi_q(x) = \phi_{q-1}(x)$ . Se  $nível(x)$  é inalterado e  $iter(x)$  aumenta, então ele aumenta em pelo menos 1, e assim  $\phi_q(x) \leq \phi_{q-1}(x) - 1$ .

Finalmente, se a  $q$ -ésima operação aumentar  $nível(x)$ , ele aumentará em pelo menos 1, de forma que o valor da expressão  $(\alpha(n) - nível(x)) \cdot ordem[x]$  cairá pelo menos  $ordem[x]$ . Como  $nível(x)$  aumentou, o valor de  $iter(x)$  pode cair, mas, de acordo com o limite (21.2), a queda será no máximo  $ordem[x] - 1$ . Desse modo, o aumento em potencial devido à mudança em  $iter(x)$  é menor que a queda em potencial devido à mudança em  $nível(x)$ , e concluímos que  $\phi_q(x) \leq \phi_{q-1}(x) - 1$ . ■

Nossos três lemas finais mostram que o custo amortizado de cada operação MAKE-SET, LINK e FIND-SET é  $O(\alpha(n))$ . Vimos na equação (17.2) que o custo amortizado de cada operação é seu custo real mais o aumento em potencial devido à operação.

### Lema 21.10

O custo amortizado de cada operação MAKE-SET é  $O(1)$ .

**Prova** Suponha que a  $q$ -ésima operação seja MAKE-SET( $x$ ). Essa operação cria o nó  $x$  com ordem 0, de forma que  $\phi_q(x) = 0$ . Nenhuma outra ordem ou potencial se altera, e então  $\Phi_q = \Phi_{q-1}$ . A observação de que o custo real da operação MAKE-SET é  $O(1)$  completa a prova. ■

### Lema 21.11

O custo amortizado de cada operação LINK é  $O(\alpha(n))$ .

**Prova** Suponha que a  $q$ -ésima operação seja LINK( $x, y$ ). O custo real da operação LINK é  $O(1)$ . Sem perda de generalidade, suponha que a operação LINK torne  $y$  o pai de  $x$ .

Para determinar a mudança em potencial devido à operação LINK, notamos que os únicos nós cujos potenciais podem mudar são  $x, y$  e os filhos de  $y$  imediatamente antes da operação. Mostraremos que o único nó cujo potencial pode aumentar devido a LINK é  $y$ , e que seu aumento é no máximo  $\alpha(n)$ :

- Pelo Lema 21.9, qualquer nó que seja filho de  $y$  imediatamente antes de LINK não pode ter seu aumento de potencial devido a LINK.
- Da definição de  $\phi_q(x)$  vemos que, como  $x$  era uma raiz pouco antes da  $q$ -ésima operação,  $\phi_{q-1}(x) = \alpha(n) \cdot ordem[x]$ . Se  $ordem[x] = 0$ , então  $\phi_q(x) = \phi_{q-1}(x)$ . Caso contrário,

$$\begin{aligned}\phi_q(x) &= (\alpha(n) - nível(x)) \cdot ordem[x] - iter(x) \\ &< (\alpha(n) \cdot ordem[x]) \quad (\text{pelas desigualdades (21.1) e (21.2)}).\end{aligned}$$

Como essa última quantidade é  $\phi_{q-1}(x)$ , vemos que o potencial de  $x$  diminui.

- Como  $y$  é uma raiz antes de LINK,  $\phi_{q-1}(y) = \alpha(n) \cdot \text{ordem}[y]$ . A operação LINK deixa  $y$  como raiz e deixa a ordem de  $y$  como está ou aumenta a ordem de  $y$  em 1. Assim,  $\phi_q(y) = \phi_{q-1}(y)$  ou  $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$ .

O aumento de potencial devido à operação LINK é portanto no máximo  $\alpha(n)$ . O custo amortizado da operação LINK é  $O(1) + \alpha(n) = O(\alpha(n))$ . ■

### Lema 21.12

O custo amortizado de cada operação FIND-SET é  $O(\alpha(n))$ .

**Prova** Suponha que a  $q$ -ésima operação seja FIND-SET e que o caminho de localização conte-nha  $s$  nós. O custo real da operação FIND-SET é  $O(s)$ . Mostraremos que nenhum potencial de nó aumenta devido a FIND-SET e que pelo menos  $\max(0, s - (\alpha(n) + 2))$  nós no caminho de localização têm seu potencial diminuído em pelo menos 1.

Para ver que nenhum potencial de nó aumenta, primeiro apelamos para o Lema 21.9 para todos os nós que não é a raiz. Se  $x$  é a raiz, então seu potencial é  $\alpha(n) \cdot \text{ordem}[x]$ , que não muda.

Agora mostramos que pelo menos  $\max(0, s - (\alpha(n) + 2))$  nós têm seu potencial diminuído por pelo menos 1. Seja  $x$  um nó no caminho de localização, tal que  $\text{ordem}[x] > 0$  e  $x$  é seguido em algum lugar no caminho de localização por outro nó  $y$  que não é uma raiz, onde  $\text{nível}(y) = \text{nível}(x)$  imediatamente antes da operação FIND-SET. (O nó  $y$  não precisa seguir  $x$  imediatamente no caminho de localização.) Todos os nós exceto no máximo  $\alpha(n) + 2$  no caminho de localização satisfazem a essas restrições sobre  $x$ . Aqueles que não as satisfazem são o primeiro nó no caminho de localização (se ele tiver ordem 0), o último nó no caminho (isto é, a raiz) e o último nó  $w$  no caminho para o qual  $\text{nível}(w) = k$ , para cada  $k = 0, 1, 2, \dots, \alpha(n) + 1$ .

Vamos fixar tal nó  $x$ , e mostraremos que o potencial de  $x$  diminui por pelo menos 1. Seja  $k = \text{nível}(x) = \text{nível}(y)$ . Imediatamente antes da compressão de caminho causada por FIND-SET, temos

$$\begin{aligned} \text{ordem}[p[x]] &\geq A_k^{(\text{iter}(x))}(\text{ordem}[x]) && \text{(pela definição de iter}(x)\text{)} , \\ \text{ordem}[p[y]] &\geq A_k(\text{ordem}[y]) && \text{(pela definição de nível}(y)\text{)} , \\ \text{ordem}[y] &\geq \text{ordem}[p[x]] && \text{(pelo Corolário 21.5 e porque } y \text{ segue } x \text{ no} \\ &&& \text{caminho de localização)} . \end{aligned}$$

Juntando essas desigualdades e sendo  $i$  o valor de  $\text{iter}(x)$  antes da compressão de caminho, temos

$$\begin{aligned} \text{ordem}[p[y]] &\geq A_k(\text{ordem}[y]) \\ &\geq A_k(\text{ordem}[p[x]]) && \text{(porque } A_k(j) \text{ é estritamente crescente)} \\ &\geq A_k(A_k^{(\text{iter}(x))}(\text{ordem}[x])) \\ &= A_k^{(i+1)}(\text{ordem}[x])) . \end{aligned}$$

Como a compressão de caminho fará  $x$  e  $y$  terem o mesmo pai, sabemos que, após a compressão de caminho,  $\text{ordem}[p[x]] = \text{ordem}[p[y]]$  e que a compressão de caminho não diminui  $\text{ordem}[p[y]]$ . Tendo em vista que  $\text{ordem}[x]$  não muda, após a compressão de caminho temos que  $\text{ordem}[p[x]] \geq A_k^{(i+1)}(\text{ordem}[x])$ . Desse modo, a compressão de caminho fará  $\text{iter}(x)$  aumentar (até pelo menos  $i + 1$ ) ou  $\text{nível}(x)$  aumentar (o que ocorre se  $\text{iter}(x)$  aumentar até pelo menos  $\text{ordem}[x] + 1$ ). Em qualquer caso, pelo Lema 21.9, temos  $\phi_q(x) \leq \phi_{q-1}(x) - 1$ . Consequentemente, o potencial de  $x$  diminui em pelo menos 1.

O custo amortizado da operação FIND-SET é o custo real mais a mudança em potencial. O custo real é  $O(s)$ , e mostramos que o potencial total diminui por pelo menos  $\max(0, s - (\alpha(n) + 2))$ . O custo amortizado então é no máximo  $(O(s) - (s - (\alpha(n) + 2))) = O(s) - s + O(\alpha(n))$ , pois podemos aumentar a escala das unidades de potencial para dominar a constante oculta em  $O(s)$ . ■

Juntando os lemas precedentes, temos o teorema a seguir.

### **Teorema 21.13**

Uma sequência de  $m$  operações MAKE-SET, UNION e FIND-SET, das quais  $n$  são operações MAKE-SET, pode ser executada sobre uma floresta de conjuntos disjuntos com união por ordenação e compressão de caminho no tempo do pior caso  $O(m \alpha(n))$ .

**Prova** Imediata, a partir dos Lemas 21.7, 21.10, 21.11 e 21.12. ■

## **Exercícios**

### **21.4-1**

Prove o Lema 21.4.

### **21.4-2**

Prove que todo nó tem ordem no máximo igual a  $\lfloor \lg n \rfloor$ .

### **21.4-3**

De acordo com o Exercício 21.4-2, quantos bits são necessários para armazenar  $ordem[x]$  para cada nó  $x$ ?

### **21.4-4**

Usando o Exercício 21.4-2 forneça uma prova simples de que as operações sobre uma floresta de conjuntos disjuntos com união por ordenação, mas sem compressão de caminho, são executadas no tempo  $O(m \lg n)$ .

### **21.4-5**

O professor Dante argumenta que, como as ordens de nós aumentam estritamente ao longo de um caminho até a raiz, os níveis de nós devem aumentar monotonicamente ao longo do caminho. Em outras palavras, se  $ordem(x) > 0$  e  $p[x]$  não é uma raiz, então  $nível(x) \leq nível(p[x])$ . O professor está certo?

### **21.4-6 \***

Considere a função  $\alpha'(n) = \min\{k : A_k(1) \geq \lg(n + 1)\}$ . Mostre que  $\alpha'(n) \leq 3$  para todos os valores práticos de  $n$  e, usando o Exercício 21.4-2, mostre como modificar o argumento da função potencial para provar que uma sequência de  $m$  operações MAKE-SET, UNION e FIND-SET, das quais  $n$  são operações MAKE-SET, pode ser executada sobre uma floresta de conjuntos disjuntos com união por ordenação e compressão de caminho no tempo do pior caso  $O(m \alpha'(n))$ .

## **Problemas**

### **21-1 Mínimo off-line**

O problema do **mínimo off-line** nos pede para manter um conjunto dinâmico  $T$  de elementos do domínio  $\{1, 2, \dots, n\}$  sob as operações INSERT e EXTRACT-MIN. Recebemos uma sequência  $S$  de  $n$  chamadas INSERT e  $m$  chamadas EXTRACT-MIN, onde cada chave em  $\{1, 2, \dots, n\}$  é inserida exatamente uma vez. Desejamos determinar qual chave é retornada por cada chamada de EXTRACT-MIN. Especificamente, desejamos preenchermos um arranjo  $extraido[1..m]$ , onde para  $i = 1, 2, \dots, m$ ,  $extraido[i]$  é a chave retornada pela  $i$ -ésima chamada de EXTRACT-MIN. O problema é “off-line” no sentido de que temos a possibilidade de processar a sequência  $S$  inteira antes de determinar quaisquer das chaves retornadas.

- a. Na instância a seguir do problema de mínimo off-line, cada INSERT é representada por um número, e cada EXTRACT-MIN é representada pela letra E:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.

Preencha os valores corretos no arranjo  $extraido$ .

Para desenvolver um algoritmo correspondente a este problema, dividimos a seqüência  $S$  em subseqüências homogêneas. Isto é, representamos  $S$  por

$$I_1, E, I_2, E, I_3, \dots, I_m, E, I_{m+1},$$

onde cada  $E$  representa uma única chamada de EXTRACT-MIN, e cada  $I_j$  representa uma seqüência (possivelmente vazia) de chamadas a INSERT. Para cada subseqüência  $I_j$ , colocamos inicialmente as chaves inseridas por essas operações em um conjunto  $K_j$ , que é vazio se  $I_j$  é vazio. Em seguida, fazemos:

**OFF-LINE-MINIMUM( $m, n$ )**

```

1 for  $i \leftarrow 1$  to  $n$ 
2   do determinar  $j$  tal que  $i \leftarrow K_j$ 
3     if  $j \neq n + 1$ 
4       then  $\text{extraido}[j] \leftarrow i$ 
5         seja  $l$  o menor valor maior que  $j$ 
           para o qual o conjunto  $K_l$  existe
6          $K_l \leftarrow K_j \cup K_l$ , destruindo  $K_j$ 
7   return  $\text{extraido}$ 
```

- b.** Demonstre que o arranjo extraído retornado por OFF-LINE-MINIMUM é correto.
- c.** Descreva como usar uma estrutura de dados de conjuntos disjuntos para implementar OFF-LINE-MINIMUM de modo eficiente. Forneça um limite restrito sobre o tempo de execução do pior caso de sua implementação.

## 21-2 Determinação da profundidade

No problema de determinação da profundidade, mantemos uma floresta  $\mathcal{F} = \{T_i\}$  de árvores enraizadas sob três operações:

**MAKE-TREE( $v$ )** cria uma árvore cujo único nó é  $v$ .

**FIND-DEPTH( $v$ )** retorna a profundidade do nó  $v$  dentro de sua árvore.

**GRAFT( $r, v$ )** faz o nó  $r$ , que se supõe ser a raiz de uma árvore, se tornar o filho do nó  $v$ , o qual se supõe estar em uma árvore diferente de  $r$ , mas que pode ou não ser ele próprio uma raiz.

- a.** Suponha que seja utilizada uma representação de árvore semelhante a uma floresta de conjuntos disjuntos:  $p[v]$  é o pai do nó  $v$ , exceto pelo fato de que  $p[v] = v$  se  $v$  é uma raiz. Se implementarmos GRAFT( $r, v$ ), definindo  $p[r] \leftarrow v$  e FIND-DEPTH( $v$ ), seguindo o caminho de localização até a raiz, retornando uma contagem de todos os nós diferentes de  $v$  encontrados, mostre que o tempo de execução no pior caso de uma seqüência de  $m$  operações MAKE-TREE, FIND-DEPTH e GRAFT é  $\Theta(m^2)$ .

Usando a heurística de união por ordenação e de compressão de caminho, podemos diminuir o tempo de execução no pior caso. Empregamos a floresta de conjuntos disjuntos  $\mathcal{J} = \{S_i\}$ , onde cada conjunto  $S_i$  (que é ele próprio uma árvore) corresponde a uma árvore  $T_i$  na floresta  $\mathcal{F}$ . Porém, a estrutura de árvore dentro de um conjunto  $S_i$  não corresponde necessariamente à de  $T_i$ . De fato, a implementação de  $S_i$  não registra os relacionamentos exatos entre pai e filho mas, apesar disso, nos permite determinar a profundidade de qualquer nó em  $T_i$ .

A idéia-chave é manter em cada nó  $v$  uma “pseudodistância”  $d[v]$ , definida de tal forma que a soma das pseudodistâncias ao longo do caminho desde  $v$  até a raiz de seu conjunto  $S_i$  seja igual à profundidade de  $v$  em  $T_i$ . Isto é, se o caminho de  $v$  até sua raiz em  $S_i$  é  $v_0, v_1, \dots, v_k$ , onde  $v_0 = v$  e  $v_k$  é a raiz de  $S_i$ , então a profundidade de  $v$  em  $T_i$  é  $\sum_{j=0}^k d[v_j]$ .

- b.** Forneça uma implementação de MAKE-TREE.
- c.** Mostre como modificar FIND-SET para implementar FIND-DEPTH. Sua implementação deve executar a compressão de caminho e seu tempo de execução deve ser linear no comprimento da árvore.

mento do caminho de localização. Certifique-se de que sua implementação atualiza corretamente as pseudodistâncias.

- d. Mostre como modificar os procedimentos UNION e LINK para implementar GRAFT( $r, v$ ), que combina os conjuntos contendo  $r$  e  $v$ . Certifique-se de que sua implementação atualiza pseudodistâncias corretamente. Observe que a raiz de um conjunto  $S$ , não é necessariamente a raiz da árvore correspondente  $T_i$ .
- e. Forneça um limite restrito sobre o tempo de execução no pior caso de uma seqüência de  $m$  operações MAKE-TREE, FIND-DEPTH e GRAFT, das quais  $n$  são operações MAKE-TREE.

### 21-3 Algoritmo de ancestrais menos comuns off-line de Tarjan

O **ancestral menos comum** de dois nós  $u$  e  $v$  em uma árvore enraizada  $T$  é o nó  $w$  que é um ancestral tanto de  $u$  quanto de  $v$  e que tem a maior profundidade em  $T$ . No **problema de ancestrais menos comuns off-line**, recebemos uma árvore enraizada  $T$  e um conjunto arbitrário  $P = \{\{u, v\}\}$  de pares não ordenados de nós em  $T$ , e desejamos determinar o ancestral menos comum de cada par em  $P$ .

Para resolver o problema de ancestrais menos comuns off-line, o procedimento a seguir executa o percurso de árvore  $T$  com a chamada inicial LCA( $raiz[T]$ ). Supõe-se que cada nó tem a cor WHITE antes do percurso.

LCA( $u$ )

- 1 MAKE-SET( $u$ )
- 2  $ancestral[FIND-SET(u)] \leftarrow u$
- 3 **for** cada filho  $v$  de  $u$  em  $T$
- 4   **do** LCA( $v$ )
- 5     UNION( $u, v$ )
- 6      $ancestral[FIND-SET(u)] \leftarrow u$
- 7  $cor[u] \leftarrow BLACK$
- 8 **for** cada nó  $v$  tal que  $\{u, v\} \in P$
- 9   **do if**  $cor[v] = BLACK$
- 10     **then** imprimir "O ancestral menos comum de"  
      " " $u$ " "e" " $v$ " " $\leftarrow$ "  $ancestral[FIND-SET(v)]$

- a. Demonstre que a linha 10 é executada exatamente uma vez para cada par  $\{u, v\} \in P$ .
- b. Demonstre que, no momento da chamada LCA( $u$ ), o número de conjuntos na estrutura de dados de conjuntos disjuntos é igual à profundidade de  $u$  em  $T$ .
- c. Prove que LCA imprime corretamente o ancestral menos comum de  $u$  e  $v$  para cada par  $\{u, v\} \in P$ .
- d. Analise o tempo de execução de LCA, supondo que usamos a implementação da estrutura de dados de conjuntos disjuntos da Seção 21.3.

## Notas do capítulo

Muitos resultados importantes para estruturas de dados de conjuntos disjuntos se devem pelo menos em parte a R. E. Tarjan. O limite superior em termos a inversa que cresce muito lentamente é  $\hat{\alpha}(m, n)$  da função de Ackermann foi dado primeiro por Tarjan [290, 292] com o uso de análise agregada. (A função  $A_k(j)$  dada na Seção 21.4 é semelhante à função de Ackermann, e a função  $\alpha(n)$  é semelhante à inversa. Tanto  $\alpha(n)$  quanto  $\hat{\alpha}(m, n)$  são no máximo 4 para todos os valores concebíveis de  $m$  e  $n$ .) Um limite superior  $O(m \lg^* n)$  foi provado antes por Hopcroft e Ullman [5, 155]. O tratamento da Seção 21.4 foi adaptado de uma análise posterior de Tarjan [294] que, por sua vez, é baseada em uma análise de Kozen [193]. Harfst e Reingold [139] fornecem uma versão baseada em potencial do limite anterior de Tarjan.

Tarjan e van Leeuwen [295] descrevem variantes sobre a heurística de compressão de caminho, inclusive “métodos de uma passagem” que, às vezes, oferecem melhores fatores constantes em seu desempenho que os métodos de duas passagens. Como ocorre com as análises anteriores de Tarjan da heurística básica de compressão de caminho, as análises de Tarjan e van Leeuwen são agregadas. Harfst e Reingold [139] mostraram mais tarde como fazer uma pequena mudança na função potencial para adaptar sua análise de compressão de caminho a essas variantes de uma passagem. Gabow e Tarjan [103] mostram que, em certas aplicações, as operações de conjuntos disjuntos podem ser levadas à execução no tempo  $O(m)$ .

Tarjan [291] mostrou que um limite inferior de tempo  $\Omega(m \hat{\alpha}(m, n))$  é exigido para operações sobre qualquer estrutura de dados de conjuntos disjuntos que atenda a certas condições técnicas. Mais tarde, esse limite inferior foi generalizado por Fredman e Saks [97] que mostraram que, no pior caso, deve haver o acesso a  $\Omega(m \hat{\alpha}(m, n))$  palavras de memória de  $(\lg n)$  bits.

---

## *Parte VI*

# *Algoritmos de grafos*

### **Introdução**

Os grafos são estruturas de dados sempre presentes em ciência da computação, e os algoritmos para trabalhar com eles são fundamentais na área. Existem centenas de problemas computacionais interessantes definidos em termos de grafos. Nesta parte, examinaremos alguns dos mais significativos.

O Capítulo 22 mostra como podemos representar um grafo em um computador e depois discute os algoritmos com base na pesquisa de um grafo, utilizando a busca em largura ou a busca em profundidade. Duas aplicações da busca em profundidade são apresentadas: a ordenação topológica de um grafo acíclico orientado e a decomposição de um grafo orientado em seus componentes fortemente conectados.

O Capítulo 23 descreve como calcular uma árvore espalhada de peso mínimo de um grafo. Tal árvore é definida como o caminho de menor peso capaz de conectar todos os vértices entre si, quando cada aresta tem um peso associado. Os algoritmos para calcular árvores espalhadas mínimas são bons exemplos de algoritmos gulosos (consulte o Capítulo 16).

Os Capítulos 24 e 25 consideram o problema de calcular caminhos mais curtos entre vértices quando cada aresta tem um comprimento ou “peso” associado. O Capítulo 24 considera o cálculo de caminhos mais curtos a partir de um determinado vértice de origem até todos os outros vértices, e o Capítulo 25 considera o cálculo de caminhos mais curtos entre cada par de vértices.

Finalmente, o Capítulo 26 mostra como calcular um fluxo máximo de material em uma rede (um grafo orientado) que tem uma origem de material especificada, um sorvedor especificado e capacidades especificadas para a quantidade de material que pode percorrer cada aresta orientada. Esse problema geral surge sob muitas formas, e um bom algoritmo para calcular fluxos máximos pode ser usado para resolver de modo eficiente uma grande variedade de problemas relacionados entre si.

Na descrição do tempo de execução de um algoritmo de grafo sobre um determinado grafo  $G = (V, E)$ , normalmente medimos o tamanho da entrada em termos do número de vértices  $|V|$  e do número de arestas  $|E|$  do grafo. Ou seja, existem dois parâmetros relevantes que descrevem o tamanho da entrada, não apenas um. Adotamos uma convenção de notação comum para esses parâmetros. Dentro da notação assintótica (como a notação de  $O$  ou a notação de  $\Theta$ ), e *somente* dentro de tal notação, o símbolo  $V$  denota  $|V|$  e o símbolo  $E$  denota  $|E|$ . Por exemplo, po-

deríamos dizer “o algoritmo é executado em tempo  $O(VE)$ ”, significando que o algoritmo é executado no tempo  $O(|V| |E|)$ . Essa convenção torna as fórmulas de tempo de execução mais fáceis de ler, sem risco de ambigüidade.

Outra convenção que adotamos aparece no pseudocódigo. Denotamos o conjunto de vértices de um grafo  $G$  por  $V[G]$  e seu conjunto de arestas por  $E[G]$ . Isto é, o pseudocódigo vê os conjuntos de vértices e arestas como atributos de um grafo.

---

## *Capítulo 22*

# *Algoritmos elementares de grafos*

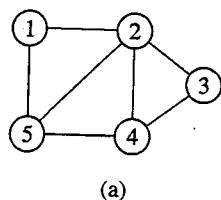
Este capítulo apresenta métodos para representar um grafo e para pesquisar um grafo. Pesquisar um grafo significa acompanhar sistematicamente as arestas do grafo de modo a alcançar os vértices do grafo. Um algoritmo de pesquisa de grafo pode descobrir muito sobre a estrutura de um grafo. Muitos algoritmos começam pesquisando seu grafo de entrada para obter essas informações estruturais. Outros algoritmos de grafos são organizados como elaborações simples de algoritmos básicos de pesquisa de grafos. As técnicas para pesquisar um grafo estão no núcleo do campo de algoritmos de grafos.

A Seção 22.1 discute as duas representações computacionais mais comuns de grafos: como listas de adjacências e como matrizes de adjacências. A Seção 22.2 apresenta um algoritmo simples de pesquisa de grafos, denominado busca em largura, e mostra como criar uma árvore primeiro na extensão. A Seção 22.3 apresenta a busca em profundidade e demonstra alguns resultados padrão sobre a ordem em que a busca em profundidade alcança os vértices. A Seção 22.4 fornece nossa primeira aplicação real de busca em profundidade: a ordenação de forma topológica de um grafo acíclico orientado. Uma segunda aplicação da busca em profundidade, consistindo em localizar os componentes fortemente conectados de um grafo orientado, é dada na Seção 22.5.

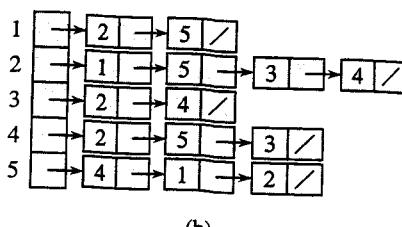
### **22.1 Representações de grafos**

Existem duas maneiras padrão para representar um grafo  $G = (V, E)$ : como uma coleção de listas de adjacências ou como uma matriz de adjacências. A representação de lista de adjacências em geral é preferida, porque ela fornece um modo compacto de representar grafos *esparsos* – aqueles para os quais  $|E|$  é muito menor que  $|V|^2$ . A maioria dos algoritmos de grafos apresentados neste livro pressupõe que um grafo de entrada é representado sob a forma de lista de adjacências. Contudo, uma representação de matriz de adjacências pode ser preferível, quando o grafo é *denso* –  $|E|$  está próximo de  $|V|^2$  – ou quando precisamos ter a possibilidade de saber com rapidez se existe uma aresta conectando dois vértices dados. Por exemplo, dois dos algoritmos de caminhos mais curtos de todos os pares apresentados no Capítulo 25 pressupõem que seus grafos de entrada são representados por matrizes de adjacências.

A *representação de lista de adjacências* de um grafo  $G = (V, E)$  consiste em um arranjo  $\text{Adj}$  de  $|V|$  listas, uma para cada vértice em  $V$ . Para cada  $u \in V$ , a lista de adjacências  $\text{Adj}[u]$  contém (ponteiros para) todos os vértices  $v$  tais que existe uma aresta  $(u, v) \in E$ . Isto é,  $\text{Adj}[u]$  consiste em todos os vértices adjacentes a  $u$  em  $G$ . (Como outra alternativa, ela pode conter ponteiros para esses vértices.) Em geral, os vértices em cada lista de adjacências estão armazenados em uma ordem arbitrária. A Figura 22.1(b) é uma representação de lista de adjacências do grafo não orientado da Figura 22.1(a). De modo semelhante, a Figura 22.2(b) é uma representação de lista de adjacências do grafo orientado da Figura 22.2(a).



(a)

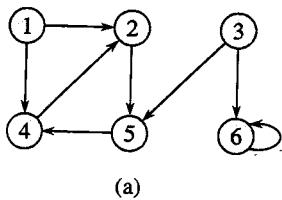


(b)

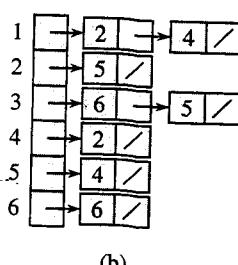
|   | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

(c)

FIGURA 22.1 Duas representações de um grafo não orientado. (a) Um grafo não orientado  $G$  que tem cinco vértices e sete arestas. (b) Uma representação de  $G$  como uma lista de adjacências. (c) A representação de  $G$  como uma matriz de adjacências



(a)



(b)

|   | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

(c)

FIGURA 22.2 Duas representações de um grafo orientado. (a) Um grafo orientado  $G$  que tem seis vértices e oito arestas. (b) Uma representação de  $G$  como uma lista de adjacências. (c) A representação de  $G$  como uma matriz de adjacências

Se  $G$  é um grafo orientado, a soma dos comprimentos de todas as listas de adjacências é  $|E|$ , pois uma aresta da forma  $(u, v)$  é representada fazendo-se  $v$  aparecer em  $\text{Adj}[u]$ . Se  $G$  é um grafo não orientado, a soma dos comprimentos de todas as listas de adjacências é  $2|E|$  pois, se  $(u, v)$  é uma aresta não orientada, então  $u$  aparece na lista de adjacências de  $v$  e *vice-versa*. Quer um grafo seja orientado ou não, a representação de lista de adjacências tem a interessante propriedade de que a quantidade de memória que ela exige é  $\Theta(V + E)$ .

As listas de adjacências podem ser prontamente adaptadas para representar *grafos ponderados*, isto é, grafos nos quais cada aresta tem um *peso* associado a ela, normalmente dado por uma função peso  $w : E \rightarrow \mathbb{R}$ . Por exemplo, seja  $G = (V, E)$  um grafo ponderado com função peso  $w$ . O peso  $w(u, v)$  da aresta  $(u, v) \in E$  está simplesmente armazenado com o vértice  $v$  na lista de adjacências de  $u$ . A representação de lista de adjacências é bastante robusta nesse aspecto, podendo ser modificada para admitir muitas outras variantes de grafos.

Uma desvantagem potencial da representação de lista de adjacências é que não existe nenhum modo mais rápido para determinar se uma dada aresta  $(u, v)$  está presente no grafo do que procurar por  $v$  na lista de adjacências  $\text{Adj}[u]$ . Essa desvantagem pode ser contornada por uma representação de matriz de adjacências do grafo, ao custo de utilizar assintoticamente mais memória. (Veja no Exercício 22.1-8 sugestões de variações sobre listas de adjacências que permitem a pesquisa de arestas mais rápida.)

No caso da **representação de matriz de adjacências** de um grafo  $G = (V, E)$ , supomos que os vértices são numerados  $1, 2, \dots, |V|$  de alguma maneira arbitrária. Então, a representação de matriz de adjacências de um grafo  $G$  consiste em uma matriz  $|V| \times |V| A = (a_{ij})$  tal que

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E, \\ 0 & \text{em caso contrário.} \end{cases}$$

As Figuras 22.1(c) e 22.2(c) são as matrizes de adjacências do grafo não orientado e do grafo orientado das Figuras 22.1(a) e 22.2(a), respectivamente. A matriz de adjacências de um grafo exige a memória  $\Theta(V^2)$ , independentemente do número de arestas no grafo.

Observe a simetria ao longo da diagonal principal da matriz de adjacências na Figura 22.1(c). Definimos a **transposta** de uma matriz  $A = (a_{ij})$  como a matriz  $A^T = (a_{ij}^T)$  dada por  $a_{ij}^T = a_{ji}$ . Tendo em vista que, em um grafo não orientado,  $(u, v)$  e  $(v, u)$  representam a mesma aresta, a matriz de adjacências  $A$  de um grafo não orientado é sua própria transposta:  $A = A^T$ . Em algumas aplicações, compensa armazenar apenas as entradas contidas na diagonal e acima da diagonal da matriz de adjacências, reduzindo assim quase pela metade a memória necessária para armazenar o grafo.

Da mesma forma que a representação de lista de adjacências de um grafo, a representação de matriz de adjacências pode ser usada no caso de grafos ponderados. Por exemplo, se  $G = (V, E)$  é um grafo ponderado com função peso de aresta  $w$ , o peso  $w(u, v)$  da aresta  $(u, v) \in E$  é simplesmente armazenado como a entrada na fila  $u$  e na coluna  $v$  da matriz de adjacências. Se uma aresta não existe, pode ser armazenado um valor NIL como sua entrada de matriz correspondente, embora em muitos problemas seja conveniente usar um valor como 0 ou  $\infty$ .

Embora a representação de lista de adjacências seja assintoticamente pelo menos tão eficiente quanto a representação de matriz de adjacências, a simplicidade de uma matriz de adjacências pode torná-la preferível quando os grafos são razoavelmente pequenos. Além disso, se o grafo é não ponderado, existe uma vantagem adicional em termos de espaço de armazenamento, favorável à representação de matriz de adjacências. Em lugar de usar uma palavra de memória de computador para cada entrada da matriz, a matriz de adjacências usa somente um bit por entrada.

## Exercícios

### 22.1-1

Dada uma representação de lista de adjacências de um grafo orientado, qual o tempo necessário para calcular o grau de saída de todo vértice? Qual o tempo necessário para calcular os graus de entrada?

### 22.1-2

Forneça uma representação de lista de adjacências para uma árvore binária completa sobre 7 vértices. Forneça uma representação de matriz de adjacências equivalente. Suponha que os vértices estejam numerados de 1 até 7 como em um heap binário.

### 22.1-3

A **transposta** de um grafo orientado  $G = (V, E)$  é o grafo  $G^T = (V, E^T)$ , onde  $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$ . Desse modo,  $G^T$  é  $G$  com todas as suas arestas invertidas. Descreva algoritmos eficientes para calcular  $G^T$  a partir de  $G$ , para a representação de lista de adjacências e também para a representação de matriz de adjacências de  $G$ . Analise os tempos de execução de seus algoritmos.

### 22.1-4

Dada uma representação de lista de adjacências de um multigrafo  $G = (V, E)$ , descreva um algoritmo de tempo  $O(V + E)$  para calcular a representação de lista de adjacências do grafo não orientado “equivalente”  $G' = (V, E')$ , onde  $E'$  consiste nas arestas em  $E$  com todas as arestas múltiplas entre dois vértices substituídas por uma aresta única e com todos os autoloops removidos.

### 22.1-5

O **quadrado** de um grafo orientado  $G = (V, E)$  é o grafo  $G^2 = (V, E^2)$  tal que  $(u, w) \in E^2$  se e somente se, para algum  $v \in V$ , tem-se  $(u, v) \in E$  e também  $(v, w) \in E$ . Ou seja,  $G^2$  contém uma aresta entre  $u$  e  $w$  sempre que  $G$  contém um caminho com exatamente duas arestas entre  $u$  e  $w$ . Descreva algoritmos eficientes para calcular  $G^2$  a partir de  $G$  para uma representação de lista de adjacências e para uma representação de matriz de adjacências de  $G$ . Analise os tempos de execução de seus algoritmos.

### 22.1-6

Quando uma representação de matriz de adjacências é usada, a maioria dos algoritmos de grafos exige o tempo  $\Omega(V^2)$ , mas existem algumas exceções. Mostre que detectar se um grafo orientado  $G$  contém um **sorvedor universal** – um vértice com grau de entrada  $|V| - 1$  e grau de saída 0 – é uma operação que pode ser realizada no tempo  $O(V)$ , dada uma matriz de adjacências para  $G$ .

### 22.1-7

A **matriz de incidência** de um grafo orientado  $G = (V, E)$  é uma matriz  $|V| \times |E| B = (b_{ij})$  tal que

$$b_{ij} = \begin{cases} -1 & \text{se a aresta } j \text{ sai do vértice } i, \\ 1 & \text{se a aresta } j \text{ entra no vértice } i, \\ 0 & \text{em caso contrário.} \end{cases}$$

Descreva o que representam as entradas do produto de matrizes  $BB^T$ , onde  $B^T$  é a transposta de  $B$ .

### 22.1-8

Suponha que, em vez de uma lista ligada, cada entrada de arranjo  $Adj[u]$  é uma tabela hash contendo os vértices  $v$  para os quais  $(u, v) \in E$ . Se todas as pesquisas de arestas forem igualmente prováveis, qual será o tempo esperado para determinar se uma aresta está no grafo? Que desvantagens esse esquema apresenta? Sugira uma estrutura de dados alternativa para cada lista de arestas que resolva esses problemas. Sua alternativa tem desvantagens em comparação com a tabela hash?

## 22.2 Busca em largura

A **busca em largura** é um dos algoritmos mais simples para se pesquisar um grafo e é o arquétipo de muitos algoritmos de grafos importantes. O algoritmo de caminhos mais curtos de origem única de Dijkstra (Seção 24.3) e o algoritmo de árvore espalhada mínima de Prim (Seção 23.2) utilizam idéias semelhantes às que aparecem na busca em largura.

Dado um grafo  $G = (V, E)$  e um vértice de **origem** distinta  $s$ , a busca em largura explora sistematicamente as arestas de  $G$  até “descobrir” cada vértice acessível a partir de  $s$ . O algoritmo calcula a distância (menor número de arestas) desde  $s$  até todos os vértices acessíveis desse tipo. Ele também produz uma “árvore primeiro na extensão” com raiz  $s$  que contém todos os vértices acessíveis. Para qualquer vértice  $v$  acessível a partir de  $s$ , o caminho na árvore primeiro na extensão de  $s$  até  $v$  corresponde a um “caminho mais curto” de  $s$  até  $v$  em  $G$ , ou seja, um caminho que contém o número mínimo de arestas. O algoritmo funciona sobre grafos orientados e também não orientados.

A busca em largura recebe esse nome porque expande a fronteira entre vértices descobertos e não descobertos uniformemente ao longo da extensão da fronteira. Isto é, o algoritmo desobre todos os vértices à distância  $k$  a partir de  $s$ , antes de descobrir quaisquer vértices à distância  $k + 1$ .

Para controlar o andamento, a busca em largura pinta cada vértice de branco, cinza ou preto. No início, todos os vértices são brancos, e mais tarde eles podem se tornar acinzentadas e depois pretos. Um vértice é **descoberto** na primeira vez em que é encontrado durante a pesquisa, e nesse momento ele se torna não branco. Portanto, vértices em cor cinza e preta foram descober-

tos, mas a busca em largura faz distinção entre eles para assegurar que a pesquisa continuará de maneira a seguir primeiro na extensão. Se  $(u, v) \in E$  e o vértice  $u$  é preto, então o vértice  $v$  é cinza ou preto; isto é, todos os vértices adjacentes a vértices pretos foram descobertos. Vértices de cor cinza podem ter alguns vértices adjacentes brancos; eles representam a fronteira entre vértices descobertos e não descobertos.

A busca em largura constrói uma árvore primeiro na extensão, contendo inicialmente apenas sua raiz, a qual é o vértice de origem  $s$ . Sempre que um vértice branco  $v$  é descoberto no curso da varredura da lista de adjacências de um vértice  $u$  já descoberto, o vértice  $v$  e a aresta  $(u, v)$  são adicionados à árvore. Dizemos que  $u$  é o *predecessor* ou *pai* de  $v$  na árvore primeiro na extensão. Tendo em vista que um vértice é descoberto no máximo uma vez, ele tem no máximo um pai. Relacionamentos de ancestral e descendente na árvore primeiro na extensão são definidos em relação à raiz  $s$  da maneira usual: se  $u$  está em um caminho na árvore a partir da raiz  $s$  até o vértice  $v$ , então  $u$  é um ancestral de  $v$ , e  $v$  é um descendente de  $u$ .

O procedimento de busca em largura BFS mostrado a seguir pressupõe que o grafo de entrada  $G = (V, E)$  é representado com o uso de listas de adjacências. Ele mantém várias estruturas de dados adicionais com cada vértice no grafo. A cor de cada vértice  $u \in V$  é armazenada na variável  $cor[u]$ , e o predecessor de  $u$  é armazenado na variável  $\pi[u]$ . Se  $u$  não tem nenhum predecessor (por exemplo, se  $u = s$  ou se  $u$  não foi descoberto), então  $\pi[u] = NIL$ . A distância desde a origem  $s$  até o vértice  $u$  calculada pelo algoritmo fica armazenada em  $d[u]$ . O algoritmo também utiliza uma fila  $Q$  de primeiro a entrar, primeiro a sair (consulte a Seção 10.1) para gerenciar o conjunto de vértices de cor cinza.

$BFS(G, s)$

```

1  for cada vértice  $u \in V[G] - \{s\}$ 
2    do  $cor[u] \leftarrow$  BRANCO
3     $d[u] \leftarrow \infty$ 
4     $\alpha[u] \leftarrow NIL$ 
5   $cor[s] \leftarrow$  CINZA
6   $d[s] \leftarrow 0$ 
7   $\alpha[s] \leftarrow NIL$ 
8   $Q \leftarrow 0$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq 0$ 
11   do  $u \leftarrow$  DEQUEUE( $Q$ )
12     for cada  $v \leftarrow Adj[u]$ 
13       do if  $cor[v] =$  BRANCO
14         then  $cor[v] \leftarrow$  CINZA
15          $d[v] \leftarrow d[u] + 1$ 
16          $\pi[v] \leftarrow u$ 
17         ENQUEUE( $Q, v$ )
18    $cor[u] \leftarrow$  PRETO

```

A Figura 22.3 ilustra o andamento de BFS sobre uma amostra de grafo.

O procedimento BFS funciona da maneira descrita a seguir. As linhas 1 a 4 pintam todos os vértices de branco, definem  $d[u]$  como infinito para todo vértice  $u$ , e definem o pai de todo vértice como NIL. A linha 5 pinta o vértice de origem  $s$  de cinza, pois ele é considerado descoberto quando o procedimento começa. A linha 6 inicializa  $d[s]$  como 0, e a linha 7 define o predecessor da origem como NIL. As linhas 8 e 9 inicializam  $Q$  como a fila contendo apenas o vértice  $s$ .

O loop **while** das linhas 10 a 18 itera enquanto continuam a existir vértices cinza, os quais são vértices descobertos que ainda não tiveram sua lista de adjacências completamente examinada. Esse loop **while** mantém o seguinte invariante:

No teste da linha 10, a fila  $Q$  consiste no conjunto de vértices de cor cinza.

Embora não usemos esse loop invariante para provar a correção, é fácil ver que ele é válido antes da primeira iteração e que cada iteração do loop mantém o invariante. Antes da primeira iteração, o único vértice cinza, e o único vértice em  $Q$ , é o vértice de origem  $s$ . A linha 11 detecta o vértice cinza  $u$  no início da fila  $Q$  e o remove de  $Q$ . O loop **for** das linhas 12 a 17 considera cada vértice  $v$  na lista de adjacências de  $u$ . Se  $v$  é branco, então ele ainda não foi descoberto, e o algoritmo o descobre executando as linhas 14 a 17. Primeiro ele é colorido de cinza, e sua distância  $d[v]$  é definida como  $d[u] + 1$ . Em seguida,  $u$  é registrado como seu pai. Finalmente, ele é colocado no fim da fila  $Q$ . Quando todos os vértices na lista de adjacências de  $u$  são examinados,  $u$  é pintado de preto nas linhas 11 e 18. O loop invariante é mantido porque sempre que um vértice é pintado de cinza (na linha 14), ele também é enfileirado (na linha 17) e, sempre que um vértice é desenfileirado (na linha 11), também é pintado de preto (na linha 18).

Os resultados da busca em largura podem depender da ordem na qual os vizinhos de um determinado vértice são visitados na linha 12; a árvore primeiro na extensão pode variar, mas as distâncias  $d$  calculadas pelo algoritmo não irão variar. (Veja o Exercício 22.2-4.)

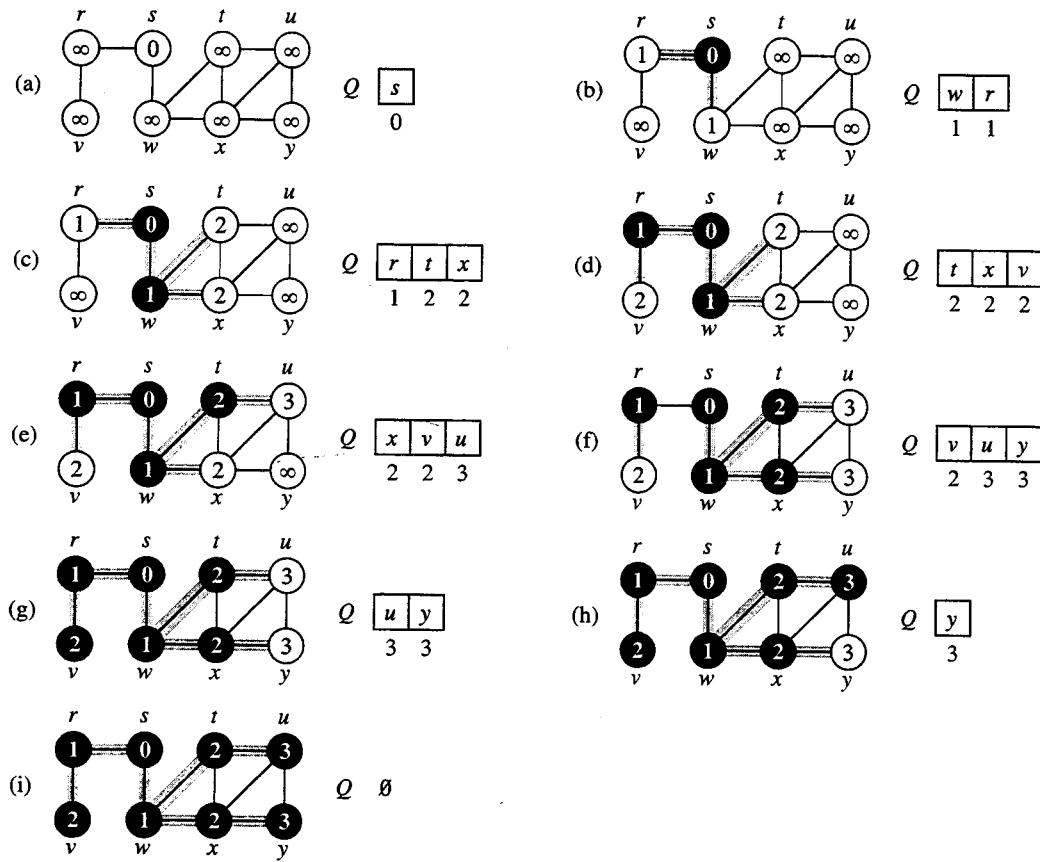


FIGURA 22.3 A operação de BFS sobre um grafo não orientado. As arestas da árvore são mostradas sombreadas à medida que são produzidas por BFS. Dentro de cada vértice  $u$  está representada  $d[u]$ . A fila  $Q$  é mostrada no princípio de cada iteração do loop **while** das linhas 10 a 18. As distâncias de vértices são mostradas ao lado de cada vértice na fila.

## Análise

Antes de provar todas as diversas propriedades da busca em largura, empreenderemos o trabalho um pouco mais fácil que consiste em analisar seu tempo de execução sobre um grafo de entrada  $G = (V, E)$ . Utilizamos a análise agregada, que vimos na Seção 17.1. Depois da inicialização, nenhum vértice estará mais embranquecido e, desse modo, o teste na linha 13 assegura que cada vértice é colocado na fila no máximo uma vez, e portanto é retirado da fila no máximo uma vez. As operações de enfileirar e desenfileirar demoram o tempo  $O(1)$ , e assim o tempo total dedicado a operações de filas é  $O(V)$ . Pelo fato de a lista de adjacências de cada vértice ser examina-

da somente quando o vértice é desenfileirado, a lista de adjacências de cada vértice é examinada no máximo uma vez. Tendo em vista que a soma dos comprimentos de todas as listas de adjacências é  $\Theta(E)$ , é gasto no máximo o tempo  $O(E)$  na varredura total das listas de adjacências. A sobre-carga correspondente à inicialização é  $O(V)$  e, desse modo, o tempo de execução total de BFS é  $O(V + E)$ . Assim, a pesquisa primeiro na extensão é executada em tempo linear no tamanho da representação de lista de adjacências de  $G$ .

## Caminhos mais curtos

No início desta seção, afirmamos que a busca em largura encontra a distância até cada vértice acessível em um grafo  $G = (V, E)$  a partir de um dado vértice de origem  $s \in V$ . Defina a **distância do caminho mais curto**  $\delta(s, v)$  desde  $s$  até  $v$ ; se não há nenhum caminho de  $s$  até  $v$ , então  $\delta(s, v) = \infty$ . Um caminho de comprimento  $\delta(s, v)$  desde  $s$  até  $v$  é dito um **caminho mais curto**<sup>1</sup> de  $s$  até  $v$ . Antes de mostrar que a busca em largura calcula realmente distâncias de caminhos mais curtos, vamos investigar uma propriedade importante das distâncias de caminhos mais curtos.

### Lema 22.1

Seja  $G = (V, E)$  um grafo orientado ou não orientado, e seja  $s \in V$  um vértice arbitrário. Então, para qualquer aresta  $(u, v) \in E$ ,

$$\delta(s, v) \leq \delta(s, u) + 1.$$

**Prova** Se  $u$  é acessível a partir de  $s$ , então o mesmo ocorre com  $v$ . Nesse caso, o caminho mais curto desde  $s$  até  $v$  não pode ser mais longo que o caminho mais curto de  $s$  até  $u$  seguido pela aresta  $(u, v)$ , e desse modo a desigualdade se mantém válida. Se  $u$  não é acessível a partir de  $s$ , então  $\delta(s, u) = \infty$ , e a desigualdade se mantém. ■

Queremos mostrar que BFS calcula de maneira apropriada  $d[v] = \delta(s, v)$  para cada vértice  $v \in V$ . Primeiro, vamos mostrar que  $d[v]$  limita  $\delta(s, v)$ , a partir do que foi dito anteriormente.

### Lema 22.2

Seja  $G = (V, E)$  um grafo orientado ou não orientado, e suponha que BFS seja executado sobre  $G$  a partir de um dado vértice de origem  $s \in V$ . Então, no final, para cada vértice  $v \in V$ , o valor  $d[v]$  calculado por BFS satisfaz a  $d[v] \geq \delta(s, v)$ .

**Prova** Utilizamos a indução sobre o número de operações ENQUEUE. Nossa hipótese indutiva é que  $d[v] \geq \delta(s, v)$  para todo  $v \in V$ .

A base da indução é a situação imediatamente após  $s$  ser inserido em  $Q$  na linha 9 de BFS. A hipótese indutiva se mantém válida nesse caso, porque  $d[s] = 0 = \delta(s, v)$  e  $d[v] = \infty \geq \delta(s, v)$  para todo  $v \in V - \{s\}$ .

Para a etapa indutiva, considere um vértice branco  $v$  que é descoberto durante a pesquisa a partir de um vértice  $u$ . A hipótese indutiva implica que  $d[u] \geq \delta(s, u)$ . Da atribuição executada pela linha 15 e do Lema 22.1, obtemos

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v). \end{aligned}$$

<sup>1</sup> Nos Capítulos 24 e 25, iremos generalizar nosso estudo de caminhos mais curtos para grafos ponderados, em que cada aresta tem um valor de peso real e o peso de um caminho é a soma dos pesos de suas arestas constituintes. Os grafos considerados neste capítulo são grafos não-ponderados ou, de modo equivalente, todas as arestas têm peso unitário.

O vértice  $v$  é então inserido na fila  $Q$ , e nunca será inserido novamente porque ele também é acinzentado, e a cláusula **then** das linhas 14 a 17 só é executada para vértices brancos. Desse modo, o valor de  $d[v]$  nunca mudará, e a hipótese induutiva será mantida. ■

Para provar que  $d[v] = \delta(s, v)$ , primeiro devemos mostrar com maior precisão como a fila  $Q$  opera durante o curso de BFS. O próximo lema mostra que, em todas as vezes, existem no máximo dois valores  $d$  distintos na fila.

### Lema 22.3

Suponha que, durante a execução de BFS sobre um grafo  $G = (V, E)$ , a fila  $Q$  contenha os vértices  $\langle v_1, v_2, \dots, v_r \rangle$ , onde  $v_1$  é o início de  $Q$  e  $v_r$  é o final. Então,  $d[v_r] \leq d[v_1] + 1$  e  $d[v_i] \leq d[v_{i+1}]$  para  $i = 1, 2, \dots, r - 1$ .

**Prova** A prova é por indução sobre o número de operações de fila. Inicialmente, quando a fila contém apenas  $s$ , o lema certamente é válido.

Para a etapa induutiva, devemos provar que o lema se mantém válido depois do desenfileiramento e do enfileiramento de um vértice. Se o início  $v_1$  da fila é desenfileirado, o novo início é  $v_2$ . (Se a fila se torna vazia, então o lema se mantém vagamente válido.) Pela hipótese induutiva,  $d[v_1] \leq d[v_2]$ . Entretanto, temos  $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$ , e as desigualdades restantes não são afetadas. Desse modo, o lema segue com  $v_2$  como início.

A ação de enfileirar um vértice exige o exame mais atento do código. Quando enfileiramos um vértice  $v$  na linha 17 de BFS, ele se torna  $v_{r+1}$ . Nesse momento, já removemos o vértice  $u$  cuja lista de adjacências está sendo atualmente examinada, a partir da fila  $Q$  e da hipótese induutiva, o novo início  $v_1$  tem  $d[v_1] \geq d[u]$ . Desse modo,  $d[v_{i+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$ . Pela hipótese induutiva, também temos  $d[v_r] \leq d[u] + 1$ , e assim  $d[v_r] \leq d[u] + 1 = d[v] = d[v_{i+1}]$ , e as desigualdades restantes não são afetadas. Portanto, o lema segue quando  $v$  é enfileirado. ■

O corolário a seguir mostra que os valores  $d$  no momento em que os vértices são enfileirados são monotonicamente crescentes com o decorrer do tempo.

### Corolário 22.4

Suponha que os vértices  $v_i$  e  $v_j$  sejam enfileirados durante a execução de BFS, e que  $v_i$  seja enfileirado antes de  $v_j$ . Então  $d[v_i] \leq d[v_j]$  no momento em que  $v_j$  é enfileirado.

**Prova** Imediata a partir do Lema 22.3 e da propriedade de que cada vértice recebe um valor  $d$  finito no máximo uma vez durante a execução de BFS.

Agora, podemos provar que a busca em largura encontra corretamente distâncias de caminhos mais curtos.

### Teorema 22.5 (Correção da busca em largura)

Seja  $G = (V, E)$  um grafo orientado ou não orientado, e suponha que BFS seja executado sobre  $G$  a partir de um dado vértice de origem  $s \in V$ . Então, durante sua execução, BFS descobre todo vértice  $v \in V$  que é acessível a partir da origem  $s$  e, no final,  $d[v] = \delta(s, v)$  para todo  $v \in V$ . Além disso, para qualquer vértice  $v \neq s$  que seja acessível a partir de  $s$ , um dos caminhos mais curtos de  $s$  até  $v$  é um caminho mais curto de  $s$  para  $\pi[v]$  seguido pela aresta  $(\pi[v], v)$ .

**Prova** Suponha, para fins de contradição, que algum vértice receba um valor  $d$  não igual à distância de seu caminho mais curto. Seja  $v$  o vértice com  $\delta(s, v)$  mínimo que recebe tal valor  $d$  incorreto; é claro que  $v \neq s$ . Pelo Lema 22.2,  $d[v] \geq \delta(s, v)$ , e portanto temos  $d[v] > \delta(s, v)$ . O vértice  $v$  deve ser acessível a partir de  $s$  porque, se não for, então  $\delta(s, v) = \infty \geq d[v]$ . Seja  $u$  o vértice imediatamente anterior a  $v$  em um caminho mais curto de  $s$  para  $v$ , de forma que  $\delta(s, v) = \delta(s, u) + 1$ . Como  $\delta(s, u) < \delta(s, v)$ , e devido à forma como escolhemos  $v$ , temos  $d[u] = \delta(s, u)$ . Juntando essas propriedades, temos

$$426 \quad | \quad d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 . \quad (22.1)$$

Agora considere o momento em que BFS opta por desenfileirar o vértice  $u$  de  $Q$  na linha 11. Nesse momento, o vértice  $v$  é branco, cinza ou preto. Mostraremos que, em cada um desses casos, derivamos uma contradição para a desigualdade (22.1). Se  $v$  é branco, então a linha 15 define  $d[v] = d[u] + 1$ , contradizendo a desigualdade (22.1). Se  $v$  é preto, então ele já foi removido da fila  $e$ , pelo Corolário 22.4, temos  $d[v] \leq d[u]$ , novamente contradizendo a desigualdade (22.1). Se  $v$  é cinza, então ele foi pintado de cinza ao ser desenfileirado algum vértice  $w$ , que foi removido de  $Q$  antes de  $u$  e para o qual  $d[v] = d[w] + 1$ . Porém, pelo Corolário 22.4,  $d[w] \leq d[u]$ , e então temos  $d[v] \leq d[u] + 1$ , uma vez mais contradizendo a desigualdade (22.1).

Desse modo, concluímos que  $d[v] = \delta(s, v)$  para todo  $v \in V$ . Todos os vértices acessíveis a partir de  $s$  devem ser descobertos porque, se não fossem, eles teriam valores  $d$  infinitos. Para concluir a prova do teorema, observe que, se  $\pi[v] = u$ , então  $d[v] = d[u] + 1$ . Desse modo, podemos obter um caminho mais curto desde  $s$  até  $v$  tomando um caminho mais curto de  $s$  até  $\pi[v]$ , e depois percorrendo a aresta  $(\pi[v], v)$ . ■

## Árvores primeiro na extensão

O procedimento BFS constrói uma árvore primeiro na extensão à medida que pesquisa o grafo, como ilustra a Figura 22.3. A árvore é representada pelo campo  $\pi$  em cada vértice. Mais formalmente, para um grafo  $G = (V, E)$  com origem  $s$ , definimos o **subgrafo predecessor** de  $G$  como  $G_\pi = (V_\pi, E_\pi)$ , onde

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\} .$$

e

$$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\} .$$

O subgrafo predecessor  $G_\pi$  é uma **árvore primeiro na extensão** se  $V_\pi$  consiste nos vértices acessíveis a partir de  $s$  e, para todo  $v \in V_\pi$ , existe um caminho único simples desde  $s$  até  $v$  em  $G_\pi$  que também é um caminho mais curto de  $s$  até  $v$  em  $G$ . Uma árvore primeiro na extensão é de fato uma árvore, pois está conectada e  $|E_\pi| = |V_\pi| - 1$  (ver Teorema B.2). As arestas em  $E_\pi$  são chamadas **arestas da árvore**.

Depois que BFS é executado a partir de uma origem  $s$  sobre um grafo  $G$ , o lema a seguir mostra que o subgrafo predecessor é uma árvore primeiro na extensão.

### Lema 22.6

Quando aplicado a um grafo orientado ou não orientado  $G = (V, E)$ , o procedimento BFS constrói  $\pi$  de tal forma que o subgrafo predecessor  $G_\pi = (V_\pi, E_\pi)$  seja uma árvore primeiro na extensão.

**Prova** A linha 16 de BFS define  $\pi[v] = u$  se e somente se  $(u, v) \in E$  e  $\delta(s, v) < \infty$  – isto é, se  $v$  é acessível a partir de  $s$  – e assim  $V_\pi$  consiste nos vértices em  $V$  acessíveis a partir de  $s$ . Tendo em vista que  $G_\pi$  forma uma árvore, pelo Teorema B.2, ele contém um caminho único de  $s$  até cada vértice em  $V_\pi$ . Aplicando o Teorema 22.5 de forma indutiva, concluímos que todo caminho desse tipo é um caminho mais curto. ■

O procedimento a seguir imprime os vértices em um caminho mais curto desde  $s$  até  $v$ , supondo-se que BFS já tinha sido executado para calcular a árvore do caminho mais curto.

**PRINT-PATH( $G, s, v$ )**

```

1 if  $v = s$ 
2 then imprimir  $s$ 
3 else if  $\pi[v] = \text{NIL}$ 
4     then imprimir “nenhum caminho de”  $s$  “para”  $v$  “existente”
5     else PRINT-PATH( $G, s, \pi[v]$ )
6     imprimir  $v$ 

```

Esse procedimento é executado em tempo linear no número de vértices no caminho impresso, pois cada chamada recursiva é para um caminho um vértice mais curto.

## Exercícios

### 22.2-1

Mostre os valores de  $d$  e  $\pi$  que resultam da execução da busca em largura sobre o grafo orientado da Figura 22.2(a), usando o vértice 3 como origem.

### 22.2-2

Mostre os valores de  $d$  e  $\pi$  que resultam da execução da busca em largura sobre o grafo não orientado da Figura 22.3, usando o vértice  $u$  como origem.

### 22.2-3

Qual é o tempo de execução de BFS se o seu grafo de entrada é representado por uma matriz de adjacências e o algoritmo é modificado para manipular essa forma de entrada?

### 22.2-4

Mostre que, em uma busca em largura, o valor  $d[u]$  atribuído a um vértice  $u$  é independente da ordem na qual são dados os vértices em cada lista de adjacências. Usando a Figura 22.3 como um exemplo, mostre que a árvore primeiro na extensão calculada por BFS pode depender da ordenação dentro de listas de adjacências.

### 22.2-5

Forneça um exemplo de um grafo orientado  $G = (V, E)$ , um vértice de origem  $s \in V$  e um conjunto de arestas de árvore  $E_\pi \subseteq E$  tal que, para cada vértice  $v \in V$ , o caminho único em  $(V, E_\pi)$  de  $s$  até  $v$  é um caminho mais curto em  $G$ , ainda que o conjunto de arestas  $E_\pi$  não possa ser produzido pela execução de BFS sobre  $G$ , não importando o modo como os vértices estão ordenados em cada lista de adjacências.

### 22.2-6

Existem dois tipos de lutadores profissionais: “bons sujeitos” e “maus sujeitos”. Entre qualquer par de lutadores profissionais pode ou não haver uma rivalidade. Suponha que temos  $n$  lutadores profissionais e temos uma lista de  $r$  pares de lutadores para os quais existem rivalidades. Dê um algoritmo de tempo  $O(n + r)$  que determine se é possível designar alguns dos lutadores como bons sujeitos e os restantes como maus sujeitos, de tal forma que a rivalidade ocorra em cada caso entre um bom sujeito e um mau sujeito. Se for possível realizar tal designação, seu algoritmo devia produzi-la.

### 22.2-7 \*

O **diâmetro** de uma árvore  $T = (V, E)$  é dado por

$$\max_{u, v \in V} \delta(u, v) ;$$

isto é, o diâmetro é a maior de todas as distâncias de caminhos mais curtos na árvore. Forneça um algoritmo eficiente para calcular o diâmetro de uma árvore e analise o tempo de execução de seu algoritmo.

### 22.2-8

Seja  $G = (V, E)$  um grafo conectado não orientado. Forneça um algoritmo de tempo  $O(V + E)$  para calcular um caminho em  $G$  que percorra cada aresta de  $E$  exatamente uma vez em cada sentido. Descreva como você pode encontrar a saída de um labirinto se receber uma grande provisão de moedas de centavos.

## 22.3 Busca em profundidade

A estratégia seguida pela busca em profundidade é, como seu nome implica, procurar “mais fundo” no grafo sempre que possível. Na busca em profundidade, as arestas são exploradas a partir do vértice  $v$  mais recentemente descoberto que ainda tem arestas inexploradas saindo dele. Quando todas as arestas de  $v$  são exploradas, a busca “regressa” para explorar as arestas que deixam o vértice a partir do qual  $v$  foi descoberto. Esse processo continua até descobrirmos todos os vértices acessíveis a partir do vértice de origem inicial. Se restarem quaisquer vértices não descobertos, então um deles será selecionado como uma nova origem, e a busca se repetirá a partir daquela origem. Esse processo inteiro será repetido até que todos os vértices sejam descobertos.

Como ocorre no caso da busca em largura, sempre que um vértice  $v$  é descoberto durante uma varredura da lista de adjacências de um vértice já descoberto  $u$ , a busca em profundidade registra esse evento definindo o campo predecessor de  $v$ , o campo  $\pi[v]$ , como  $u$ . Diferente da busca em largura, cujo subgrafo predecessor forma uma árvore, o subgrafo predecessor produzido por uma busca em profundidade pode ser composto por várias árvores, porque a busca pode ser repetida a partir de várias origens.<sup>2</sup> O **subgrafo predecessor** de uma busca em profundidade é então definido de forma ligeiramente diferente daquela de uma busca em largura: fazemos  $G_\pi = (V, E_\pi)$ , onde

$$E_\pi = \{(\pi[v], v) : v \in V \text{ e } \pi[v] \neq \text{NIL}\}.$$

O subgrafo predecessor de uma busca em profundidade forma uma **floresta primeiro na profundidade** composta por várias **árvores primeiro na profundidade**. As arestas em  $E_\pi$  são chamadas **arestas de árvore**.

Como ocorre no caso da busca em largura, os vértices são coloridos durante a busca, a fim de indicar seu estado. Cada vértice é inicialmente branco, é acinzentado ao ser **descoberto** na busca, e depois é enegrecido quando é **terminado**, isto é, quando sua lista de adjacências é completamente examinada. Essa técnica garante que cada vértice acaba em exatamente uma árvore primeiro na profundidade, de forma que essas árvores sejam disjuntas.

Além de criar uma floresta primeiro na profundidade, a busca em profundidade também identifica cada vértice com um **carimbo de tempo**. Cada vértice  $v$  tem dois carimbos de tempo: o primeiro carimbo de tempo  $d[v]$  registra quando  $v$  é descoberto pela primeira vez (e acinzentado), e o segundo carimbo de tempo  $f[v]$  registra quando a busca termina de examinar a lista de adjacências de  $v$  (e pinta  $v$  de preto). Esses carimbos de tempo são usados em muitos algoritmos de grafos e em geral são úteis no raciocínio sobre o comportamento da busca em profundidade.

O procedimento DFS a seguir registra na variável  $d[u]$  o momento em que descobre o vértice  $u$ , e registra na variável  $f[u]$  o momento em que termina o vértice  $u$ . Esses carimbos de tempo são inteiros entre 1 e 2  $|V|$ , pois existe um evento de descoberta e um evento de término para cada um dos  $|V|$  vértices. Para todo vértice  $u$ ,

$$d[u] < f[u]. \quad (22.2)$$

O vértice  $u$  é BRANCO antes do tempo  $d[u]$ , CINZA entre o tempo  $d[u]$  e o tempo  $f[u]$  e PRETO depois disso.

---

<sup>2</sup> Pode parecer arbitrário que a busca em largura se limite apenas a uma origem, enquanto a busca em profundidade pode pesquisar a partir de várias origens. Embora em termos conceituais a busca em largura possa se dar a partir de várias origens e a busca em profundidade possa se limitar a uma origem, nossa abordagem reflete a forma como os resultados dessas buscas são normalmente usados. Em geral, a busca em largura é empregada para encontrar distâncias de caminhos mais curtos (e o subgrafo predecessor associado) a partir de uma origem dada. Com freqüência, a busca em profundidade é uma sub-rotina em outro algoritmo, como veremos mais adiante neste capítulo.

O pseudocódigo a seguir é o algoritmo básico de pesquisa primeiro na profundidade. O grafo de entrada  $G$  pode ser orientado ou não orientado. A variável  $\text{tempo}$  é uma variável global que utilizamos para definir carimbos de tempo.

$\text{DFS}(G)$

```

1 for cada vértice  $u \leftarrow V[G]$ 
2   do  $\text{cor}[u] \leftarrow \text{BRANCO}$ 
3    $\pi[u] \leftarrow \text{NIL}$ 
4    $\text{tempo} \leftarrow 0$ 
5 for cada vértice  $u \in V[G]$ 
6   do if  $\text{cor}[u] = \text{BRANCO}$ 
7     then  $\text{DFS-VISIT}(u)$ 
```

$\text{DFS-VISIT}(u)$

```

1  $\text{cor}[u] \leftarrow \text{CINZA}$        $\triangleright$  Branco, o vértice  $u$  acabou de ser descoberto.
2  $\text{tempo} \leftarrow \text{tempo} + 1$ 
3  $d[u] \leftarrow \text{tempo}$ 
4 for cada  $v \in \text{Adj}[u]$      $\triangleright$  Explora a aresta  $(u, v)$ .
5   do if  $\text{cor}[v] = \text{BRANCO}$ 
6     then  $\pi[v] \leftarrow u$ 
7        $\text{DFS-VISIT}(v)$ 
8  $\text{cor}[u] \leftarrow \text{PRETO}$        $\triangleright$  Enegrece  $u$ ; terminado.
9  $f[u] \leftarrow \text{tempo} \leftarrow \text{tempo} + 1$ 
```

A Figura 22.4 ilustra o andamento de DFS sobre o grafo mostrado na Figura 22.2.

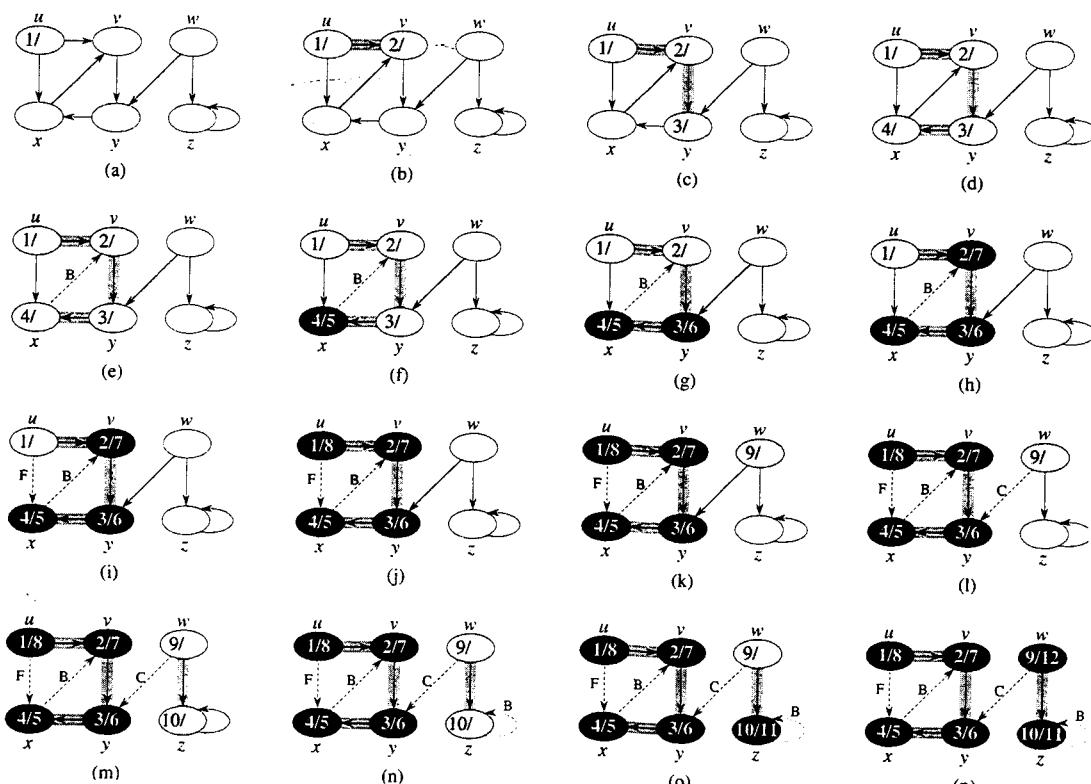


FIGURA 22.4 O andamento do algoritmo DFS de busca em profundidade sobre um grafo orientado. À medida que as arestas são exploradas pelo algoritmo, elas são mostradas como sombreadas (se são arestas de árvores) ou tracejadas (em caso contrário). Areostas que não são de árvores são identificadas como B, C ou F, de acordo com o fato de serem arestas de retorno, cruzadas ou diretas. Os vértices recebem um carimbo de tempo para identificação pelo tempo de descoberta/tempo de término

O procedimento DFS funciona da maneira descrita a seguir. As linhas 1 a 3 pintam todos os vértices de branco e inicializam seus campos  $\pi$  como NIL. A linha 4 reajusta o contador de tempo global. As linhas 5 a 7 verificam cada vértice de  $V$  por sua vez e, quando um vértice branco é encontrado, alcançam-no usando DFS-VISIT. Toda vez que  $\text{DFS-VISIT}(u)$  é chamado na linha 7, o vértice  $u$  se torna a raiz de uma nova árvore na floresta primeiro na profundidade. Quando DFS retorna, todo vértice  $u$  recebe a atribuição de um tempo de descoberta  $d[u]$  e um tempo de término  $f[u]$ .

Em cada chamada a  $\text{DFS-VISIT}(u)$ , o vértice  $u$  é inicialmente branco. A linha 1 pinta  $u$  de cinza, a linha 2 incrementa a variável global  $\text{tempo}$  e a linha 3 registra o novo valor de  $\text{tempo}$  como o tempo de descoberta  $d[u]$ . As linhas 4 a 7 examinam cada vértice  $v$  adjacente a  $u$  e alcançam recursivamente  $v$  se ele é branco. À medida que cada vértice  $v \in \text{Adj}[u]$  é considerado na linha 4, dizemos que a aresta  $(u, v)$  é **explorada** pela busca em profundidade. Finalmente, depois que toda aresta que deixa  $u$  é explorada, as linhas 8 e 9 pintam  $u$  de preto e registram o tempo de término em  $f[u]$ .

Observe que os resultados de busca em profundidade podem depender da ordem em que os vértices são examinados na linha 5 de DFS e da ordem em que os vizinhos de um vértice são alcançados na linha 4 de DFS-VISIT. Essas ordens de visitação diferentes tendem a não causar problemas na prática, pois o resultado de *qualquer* busca em profundidade pode em geral ser usado de forma eficiente, com resultados essencialmente equivalentes.

Qual é o tempo de execução de DFS? Os loops nas linhas 1 a 3 e nas linhas 5 a 7 de DFS demoram o tempo  $\Theta(V)$ , fora o tempo para executar as chamadas a DFS-VISIT. Como fizemos no caso da busca em largura, usamos análise agregada. O procedimento DFS-VISIT é chamado exatamente uma vez para cada vértice  $v \in V$ , pois DFS-VISIT é invocado somente sobre vértices brancos e a primeira ação é pintar o vértice de cinza. Durante uma execução de  $\text{DFS-VISIT}(v)$ , o loop nas linhas 4 a 7 é executado  $|\text{Adj}[v]|$  vezes. Tendo em vista que

$$\sum_{v \in V} |\text{Adj}[v]| = \Theta(E),$$

o custo total da execução das linhas 4 a 7 de DFS-VISIT é  $\Theta(E)$ . Portanto, o tempo de execução de DFS é  $\Theta(V + E)$ .

## Propriedades da busca em profundidade

A busca em profundidade produz muitas informações sobre a estrutura de um grafo. Talvez a propriedade mais básica da busca em profundidade seja o fato de que o subgrafo predecessor  $G_\pi$  de fato forma uma floresta de árvores, pois a estrutura das árvores primeiro na profundidade reflete exatamente a estrutura de chamadas recursivas de DFS-VISIT. Isto é,  $u = \pi[v]$  se e somente se  $\text{DFS-VISIT}(v)$  foi chamado durante uma pesquisa da lista de adjacências de  $u$ . Além disso, o vértice  $v$  é um descendente do vértice  $u$  na floresta primeiro na profundidade se e somente se  $v$  é descoberto durante o tempo em que  $u$  é cinza.

Outra propriedade importante da busca em profundidade é que os tempos de descoberta e término têm **estrutura de parênteses**. Se representarmos a descoberta do vértice  $u$  com um parêntese esquerdo “( $u$ ” e representarmos seu término por um parêntese direito “ $u$ ”), então a história de descobertas e términos irá gerar uma expressão bem formada, no sentido de que os parênteses serão aninhados de modo apropriado. Por exemplo, a busca em profundidade da Figura 22.5(a) corresponde à colocação entre parênteses mostrada na Figura 22.5(b). Outro modo de enunciar a condição de estrutura de parênteses é dada no teorema a seguir.

### Teorema 22.6 (Teorema dos parênteses)

Em qualquer busca em profundidade de um grafo (orientado ou não orientado)  $G = (V, E)$ , para dois vértices quaisquer  $u$  e  $v$ , exatamente uma das três condições a seguir é válida:

- Os intervalos  $[d[u], f[u]]$  e  $[d[v], f[v]]$  são completamente disjuntos, e nem  $u$  nem  $v$  é um descendente do outro na floresta primeiro na profundidade.
- O intervalo  $[d[u], f[u]]$  está contido inteiramente dentro do intervalo  $[d[v], f[v]]$ , e  $u$  é um descendente de  $v$  na árvore primeiro na profundidade.
- O intervalo  $[d[v], f[v]]$  está contido inteiramente dentro do intervalo  $[d[u], f[u]]$ , e  $v$  é um descendente de  $u$  em uma árvore primeiro na profundidade.

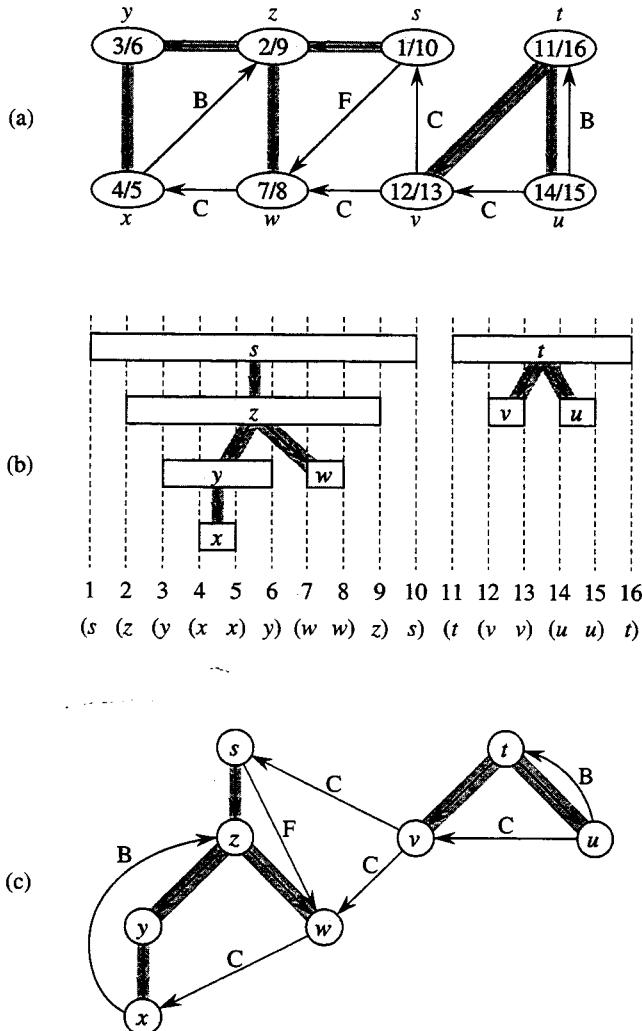


FIGURA 22.5 Propriedades da busca em profundidade. (a) O resultado de uma busca em profundidade em um grafo orientado. Os vértices são identificados por carimbos de tempo e os tipos de arestas são indicados como na Figura 22.4. (b) Intervalos correspondentes ao tempo de descoberta e ao tempo de término de cada vértice correspondem à colocação dos parênteses mostrada. Cada retângulo comprehende o intervalo dado pelos tempos de descoberta e término do vértice correspondente. As arestas de árvore são mostradas. Se dois intervalos se superpõem, então um deles é aninhado no outro, e o vértice correspondente ao menor intervalo é um descendente do vértice que corresponde ao maior. (c) O grafo da parte (a) redesenhado, com todas as arestas de árvore e diretas descendo no interior de uma árvore primeiro na profundidade e todas as arestas de retorno subindo de um descendente para um ancestral

**Prova** Começamos com o caso no qual  $d[u] < d[v]$ . Existem dois subcasos a considerar, de acordo com o fato de  $d[v] < f[u]$  ou não. No primeiro subcaso,  $d[v] < f[u]$ , e assim  $v$  foi descoberto enquanto  $u$  ainda estava cinza. Isso implica que  $v$  é um descendente de  $u$ . Além disso, como  $v$  foi descoberto mais recentemente que  $u$ , todas as suas arestas de saída são exploradas, e

$v$  é terminado, antes da pesquisa retornar a  $u$  e terminá-lo. Portanto, nesse caso, o intervalo  $[d[v], f[v]]$  está completamente contido dentro do intervalo  $[d[u], f[u]]$ . No outro subcaso,  $f[u] < d[v]$ , e a desigualdade (22.2) implica que os intervalos  $[d[u], f[u]]$  e  $[d[v], f[v]]$  são disjuntos.

O caso em que  $d[v] < d[u]$  é semelhante, com os papéis de  $u$  e  $v$  invertidos no argumento anterior. ■

#### **Corolário 22.8 (Aninhamento de intervalos dos descendentes)**

O vértice  $v$  é um descendente próprio do vértice  $u$  na floresta primeiro na profundidade correspondente a um grafo (orientado ou não orientado)  $G$  se e somente se  $d[u] < d[v] < f[v] < f[u]$ .

**Prova** Imediata, a partir do Teorema 22.7. ■

O próximo teorema fornece outra caracterização importante do momento em que um vértice é um descendente de outro na floresta primeiro na profundidade.

#### **Teorema 22.9 (Teorema do caminho branco)**

Em uma floresta primeiro na profundidade de um grafo (orientado ou não orientado)  $G = (V, E)$ , o vértice  $v$  é um descendente do vértice  $u$  se e somente se no momento  $d[u]$  em que a procura descobre  $u$ , o vértice  $v$  pode ser alcançado a partir de  $u$  ao longo de um caminho que consiste inteiramente em vértices brancos.

**Prova**  $\Rightarrow$ : Suponha que  $v$  seja um descendente de  $u$ . Seja  $w$  qualquer vértice sobre o caminho entre  $u$  e  $v$  na árvore primeiro na profundidade, de forma que  $w$  seja um descendente de  $u$ . Pelo Corolário 22.8,  $d[u] < d[w]$ , e assim  $w$  é branco no momento  $d[u]$ .

$\Leftarrow$ : Suponha que o vértice  $v$  seja acessível a partir de  $u$  ao longo de um caminho de vértices brancos no momento  $d[u]$ , mas  $v$  não se torne um descendente de  $u$  na árvore primeiro na profundidade. Sem perda de generalidade, suponha que cada outro vértice ao longo do caminho se torne um descendente de  $u$ . (Caso contrário, seja  $v$  o vértice mais próximo de  $u$  ao longo do caminho que não se torna um descendente de  $u$ .) Seja  $w$  o predecessor de  $v$  no caminho, de modo que  $w$  seja um descendente de  $u$  ( $w$  e  $u$  podem de fato ser o mesmo vértice) e, pelo Corolário 22.8,  $f[w] \leq f[u]$ . Observe que  $v$  tem de ser descoberto depois de  $u$  ser descoberto, mas antes de  $w$  ser terminado. Então,  $d[u] < d[v] < f[w] \leq f[u]$ . O Teorema 22.7 implica então que o intervalo  $[d[v], f[v]]$  está completamente contido dentro do intervalo  $[d[u], f[u]]$ . Pelo Corolário 22.8,  $v$  deve afinal ser um descendente de  $u$ . ■

### **Classificação de arestas**

Outra propriedade interessante da busca em profundidade é que a pesquisa pode ser usada para classificar as arestas do grafo de entrada  $G = (V, E)$ . Essa classificação de arestas pode ser empregada para reunir informações importantes sobre um grafo. Por exemplo, na próxima seção, veremos que um grafo orientado é acíclico se e somente se uma busca em profundidade não produz nenhuma aresta “de retorno” (Lema 22.11).

Podemos definir quatro tipos de arestas em termos da floresta primeiro na profundidade  $G_\pi$  produzida por uma busca em profundidade sobre  $G$ .

1. **Arestas de árvore** são arestas na floresta primeiro na profundidade  $G_\pi$ . A aresta  $(u, v)$  é uma aresta de árvore se  $v$  foi descoberto primeiro pela exploração da aresta  $(u, v)$ .
2. **Arestas de retorno** são as arestas  $(u, v)$  que conectam um vértice  $u$  a um ancestral  $v$  em uma árvore primeiro na profundidade. Autoloops são considerados arestas de retorno.
3. **Arestas diretas** são as arestas  $(u, v)$  não de árvore que conectam um vértice  $u$  a um descendente  $v$  em uma árvore primeiro na profundidade.
4. **Arestas cruzadas** são todas as outras arestas. Elas podem estar entre vértices na mesma árvore primeiro na profundidade, desde que um vértice não seja um ancestral do outro, ou podem estar entre vértices em diferentes árvores primeiro na profundidade.

Nas Figuras 22.4 e 22.5, as arestas são identificadas para indicar seu tipo. A Figura 22.5(c) também mostra de que maneira o grafo da Figura 22.5(a) pode ser redesenhado de tal modo que todas as arestas de árvores e diretas desçam em uma árvore primeiro na profundidade, e que todas as arestas de retorno subam nessa árvore. Qualquer grafo pode ser redesenhado dessa maneira.

O algoritmo DFS pode ser modificado para classificar arestas à medida que as encontra. A idéia chave é que cada aresta  $(u, v)$  pode ser classificada pela cor do vértice  $v$  que é alcançado quando a aresta é explorada primeiro (a não ser pelo fato de que as arestas diretas e cruzadas não se distinguem uma da outra):

1. BRANCO indica uma aresta de árvore.
2. CINZA indica uma aresta de retorno.
3. PRETO indica uma aresta direta ou cruzada.

O primeiro caso é imediato a partir da especificação do algoritmo. Para o segundo caso, observe que os vértices cinza sempre formam uma cadeia linear de descendentes que corresponde à pilha de invocações ativas de DFS-VISIT; o número de vértices cinza é uma unidade maior que a profundidade na floresta primeiro na profundidade do vértice mais recentemente descoberto. A exploração sempre prossegue a partir do vértice cinza mais profundo; assim, uma aresta que alcança outro vértice cinza alcança um ancestral. O terceiro caso trata da possibilidade restante; pode-se mostrar que tal aresta  $(u, v)$  é uma aresta direta se  $d[u] < d[v]$  e uma aresta cruzada se  $d[u] > d[v]$ . (Consulte o Exercício 22.3-4.)

Em um grafo não orientado, pode haver alguma ambigüidade na ordenação de tipos, pois  $(u, v)$  e  $(v, u)$  são na realidade a mesma aresta. Em tal caso, a aresta é classificada como do *primeiro* tipo na lista de classificação que se aplica. De forma equivalente (consulte o Exercício 22.3-5), a aresta é classificada de acordo com o par dentre  $(u, v)$  e  $(v, u)$  que seja encontrado primeiro durante a execução do algoritmo.

Agora, mostramos que as arestas diretas e cruzadas nunca ocorrem em uma busca em profundidade de um grafo não orientado.

### **Teorema 22.10**

Em uma busca em profundidade de um grafo não orientado  $G$ , toda aresta de  $G$  é uma aresta de árvore ou uma aresta de retorno.

**Prova** Seja  $(u, v)$  uma aresta arbitrária de  $G$ , e suponha sem perda de generalidade que  $d[u] < d[v]$ . Então,  $v$  deve ser descoberto e terminado antes de terminarmos  $u$  (enquanto  $u$  é cinza), pois  $v$  está na lista de adjacências de  $u$ . Se a aresta  $(u, v)$  é explorada primeiro no sentido de  $u$  para  $v$ , então  $v$  é não descoberto (branco) até esse momento porque, do contrário, já teríamos explorado essa aresta no sentido de  $v$  para  $u$ . Portanto,  $(u, v)$  se torna uma aresta de árvore. Se  $(u, v)$  é explorada primeiro no sentido de  $v$  para  $u$ , então  $(u, v)$  é uma aresta de retorno, pois  $u$  ainda é cinza no momento em que a aresta é explorada pela primeira vez. ■

Veremos várias aplicações desses teoremas nas seções seguintes.

## **Exercícios**

### **22.3-1**

Faça um diagrama de 3 por 3 com identificações de linhas e colunas BRANCO, CINZA e PRETO. Em cada célula  $(i, j)$ , indique se, em qualquer instante durante uma busca em profundidade de um grafo orientado, pode existir uma aresta de um vértice de cor  $i$  até um vértice de cor  $j$ . Para cada aresta possível, indique de quais tipos de arestas ela pode ser. Crie um segundo diagrama como esse para a busca em profundidade de um grafo não orientado.

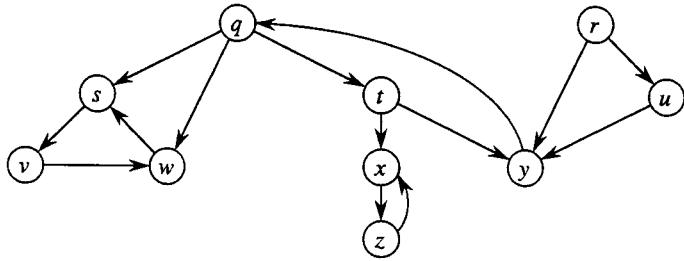


FIGURA 22.6 Um grafo orientado para uso nos Exercícios 22.3-2 e 22.5-2

### 22.3-2

Mostre como a busca em profundidade funciona sobre o grafo da Figura 22.6. Suponha que o loop for das linhas 5 a 7 do procedimento DFS considere os vértices em ordem alfabética, e suponha que cada lista de adjacências esteja em ordem alfabética. Mostre os tempos de descoberta e término para cada vértice, e mostre também a classificação de cada aresta.

### 22.3-3

Mostre a estrutura de parênteses da busca em profundidade apresentada na Figura 22.4.

### 22.3-4

Mostre que a aresta  $(u, v)$  é

- Uma aresta de árvore ou aresta direta se e somente se  $d[u] < d[v] < f[v] < f[u]$ .
- Uma aresta de retorno se e somente se  $d[v] < d[u] < f[u] < f[v]$ .
- Uma aresta cruzada se e somente se  $d[v] < f[v] < d[u] < f[u]$ .

### 22.3-5

Mostre que em um grafo não orientado, a classificação de uma aresta  $(u, v)$  como uma aresta de árvore ou uma aresta de retorno de acordo com o fato de  $(u, v)$  ou  $(v, u)$  ser encontrado primeiro durante a busca em profundidade é equivalente a classificá-la de acordo com a prioridade de tipos no esquema de classificação.

### 22.3-6

Reescreva o procedimento DFS, utilizando uma pilha para eliminar a recursão.

### 22.3-7

Forneça um contra-exemplo para a hipótese de que, se existe um caminho de  $u$  para  $v$  em um grafo orientado  $G$ , e se  $d[i] < d[v]$  em uma busca em profundidade de  $G$ , então  $v$  é um descendente de  $u$  na floresta primeiro na profundidade produzida.

### 22.3-8

Forneça um contra-exemplo para a hipótese de que, se existe um caminho de  $u$  para  $v$  em um grafo orientado  $G$ , então qualquer busca em profundidade deve resultar em  $d[v] \leq f[u]$ .

### 22.3-9

Modifique o pseudocódigo para a busca em profundidade, de tal modo que ele imprima toda aresta no grafo orientado  $G$ , juntamente com seu tipo. Mostre quais modificações, se for o caso, devem ser feitas se  $G$  for não orientado.

### 22.3-10

Explique como um vértice  $u$  de um grafo orientado pode acabar em uma árvore primeiro na profundidade contendo apenas  $u$ , embora  $u$  tenha tanto arestas de entrada quanto de saída em  $G$ .

### 22.3-11

Mostre que uma busca em profundidade de um grafo não orientado  $G$  pode ser usada para identificar os componentes conexos de  $G$ , e que a floresta primeiro na profundidade contém tantas árvores quantos componentes conexos existem em  $G$ . Mais precisamente, mostre como modificar a busca em profundidade de modo que cada vértice  $v$  receba a atribuição de uma etiqueta inteira  $cc[v]$  entre 1 e  $k$ , onde  $k$  é o número de componentes conexos de  $G$ , de tal forma que  $cc[u] = cc[v]$  se e somente se  $u$  e  $v$  estiverem no mesmo componente conectado.

### 22.3-12 \*

Um grafo orientado  $G = (V, E)$  é **isoladamente conectado** se  $u \rightarrow v$  implica que existe no máximo um caminho simples de  $u$  até  $v$  para todos os vértices  $u, v \in V$ . Forneça um algoritmo eficiente para detectar se um grafo orientado é ou não isoladamente conectado.

## 22.4 Ordenação topológica

Esta seção mostra como a busca em profundidade pode ser usada para executar ordenações topológicas de grafos acíclicos orientados, ou “gaos”, como eles são chamados às vezes. Uma **ordenação topológica** de um gao  $G = (V, E)$  é uma ordenação linear de todos os seus vértices, tal que se  $G$  contém uma aresta  $(u, v)$ , então  $u$  aparece antes de  $v$  na ordenação. (Se o grafo não é acíclico, então não é possível nenhuma ordenação linear.) Uma ordenação topológica de um grafo pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal, de tal forma que todas as arestas orientadas sigam da esquerda para a direita. Desse modo, a ordenação topológica é diferente do tipo habitual de “ordenação” estudado na Parte II.

Grafos acíclicos orientados são usados em muitas aplicações para indicar precedências entre eventos. A Figura 22.7 mostra um exemplo que surge quando o professor Bumstead se veste pela manhã. O professor deve vestir certas peças de roupa antes de outras (por exemplo, meias antes de sapatos). Outros itens podem ser colocados em qualquer ordem (por exemplo, meias e calças). Uma aresta orientada  $(u, v)$  no gao da Figura 22.7(a) indica que a peça de roupa  $u$  deve ser vestida antes da peça  $v$ . Uma ordenação topológica desse gao fornece portanto uma ordem para o processo de se vestir. A Figura 22.7(b) mostra o gao topologicamente ordenado como uma ordenação de vértices ao longo de uma linha horizontal, tal que todas as arestas orientadas sigam da esquerda para a direita.

O algoritmo simples a seguir ordena topologicamente um gao.

**TOPOLOGICAL-SORT( $G$ )**

- 1 chamar  $\text{DFS}(G)$  para calcular o tempo de término  $f[v]$  para cada vértice  $v$
- 2 à medida que cada vértice é terminado, inserir o vértice à frente de uma lista ligada
- 3 **return** a lista ligada de vértices

A Figura 22.7(b) mostra como os vértices topologicamente ordenados aparecem na ordem inversa de seus tempos de término.

Podemos executar uma ordenação topológica no tempo  $\Theta(V + E)$ , pois a busca em profundidade demora o tempo  $\Theta(V + E)$  e leva o tempo  $O(1)$  para inserir cada um dos  $|V|$  vértices à frente da lista ligada.

Demonstramos a correção desse algoritmo utilizando o seguinte lema chave que caracteriza grafos acíclicos orientados.

### Lema 22.11

Um grafo orientado  $G$  é acíclico se e somente se uma busca em profundidade de  $G$  não produz nenhuma aresta de retorno.

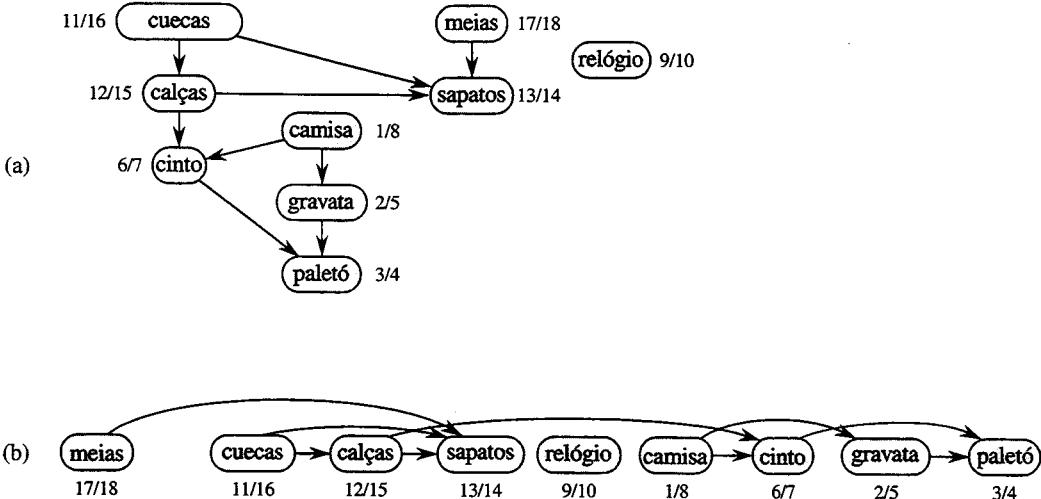


FIGURA 22.7 (a) O professor Bumstead ordena topologicamente sua roupa ao se vestir. Cada aresta orientada  $(u, v)$  significa que a peça de roupa  $u$  deve ser vestida antes da peça  $v$ . Os tempos de descoberta e término de uma busca em profundidade são mostrados ao lado de cada vértice. (b) O mesmo grafo mostrado com uma ordenação topológica. Seus vértices estão organizados da esquerda para a direita, em ordem de tempo de término decrescente. Observe que todas as arestas orientadas seguem da esquerda para a direita.

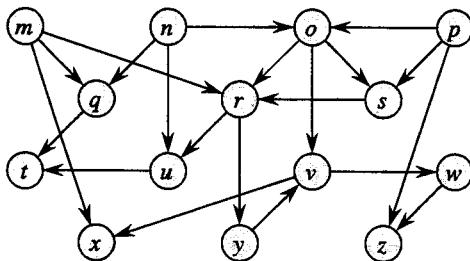


FIGURA 22.8 Um gao para ordenação topológica

**Prova**  $\Rightarrow$ : Suponha que exista uma aresta de retorno  $(u, v)$ . Então, o vértice  $v$  é um ancestral do vértice  $u$  na floresta primeiro na profundidade. Desse modo, existe um caminho de  $v$  até  $u$  em  $G$ , e a aresta de retorno  $(u, v)$  completa um ciclo.

$\Leftarrow$ : Suponha que  $G$  contém um ciclo  $c$ . Mostramos que uma busca em profundidade de  $G$  produz uma aresta de retorno. Seja  $v$  o primeiro vértice a ser descoberto em  $c$ , e seja  $(u, v)$  a aresta precedente em  $c$ . No momento  $d[v]$ , existe um caminho de vértices brancos desde  $v$  até  $u$ . Pelo teorema do caminho branco, o vértice  $u$  se torna um descendente de  $v$  na floresta primeiro na profundidade. Então,  $(u, v)$  é uma aresta de retorno. ■

### Teorema 22.12

TOPOLOGICAL-SORT( $G$ ) produz uma ordenação topológica de um grafo acíclico orientado  $G$ .

**Prova** Suponha que DFS seja executado em um dado gao  $G = (V, E)$  para determinar tempos de término para seus vértices. É suficiente mostrar que, para qualquer par de vértices distintos  $u, v \in V$ , se existe uma aresta em  $G$  de  $u$  até  $v$ , então  $f[v] < f[u]$ . Considere qualquer aresta  $(u, v)$  explorada por  $\text{DFS}(G)$ . Quando essa aresta é explorada,  $v$  não pode ser cinza, pois nesse caso  $v$  seria um ancestral de  $u$  e  $(u, v)$  seria uma aresta de retorno, contradizendo o Lema 22.11. Por essa razão,  $v$  deve ser branco ou preto. Se  $v$  é branco, ele se torna um descendente de  $u$ , e assim  $f[v] < f[u]$ . Se  $v$  é preto, ele já terminou, de forma que  $f[v]$  já foi definido. Como ainda estamos explorando  $u$ , ainda temos de atribuir um carimbo de tempo a  $f[u]$  e então, uma vez que o fizermos, também teremos  $f[v] < f[u]$ . Desse modo, para qualquer aresta  $(u, v)$  no gao, temos  $f[v] < f[u]$ , provando o teorema. ■

## Exercícios

### 22.4-1

Mostre a ordenação de vértices produzida por TOPOLOGICAL-SORT quando ele é executado sobre o grafo na Figura 22.8, sob a hipótese do Exercício 22.3-2.

### 22.4-2

Forneça um algoritmo de tempo linear que tome como entrada um grafo acíclico orientado  $G = (V, E)$  e dois vértices  $s$  e  $t$ , e retorne o número de caminhos de  $s$  para  $t$  em  $G$ . Por exemplo, no grafo acíclico orientado da Figura 22.8, existem exatamente quatro caminhos do vértice  $p$  para o vértice  $v$ :  $pov$ ,  $poryv$ ,  $posryv$  e  $psryv$ . (Seu algoritmo só precisa contar os caminhos, não listá-los.)

### 22.4-3

Forneça um algoritmo que determine se um dado grafo não orientado  $G = (V, E)$  contém um ciclo. Seu algoritmo deve ser executado no tempo  $O(V)$ , independente de  $|E|$ .

### 22.4-4

Prove ou conteste: se um grafo orientado  $G$  contém ciclos, então TOPOLOGICAL-SORT( $G$ ) produz uma ordenação de vértices que minimiza o número de arestas “ruins” que são incompatíveis com a ordenação produzida.

### 22.4-5

Outro modo de executar a ordenação topológica sobre um grafo acíclico orientado  $G = (V, E)$  é encontrar repetidamente um vértice de grau de entrada 0, dar saída a ele e removê-lo do grafo, bem como todas as suas arestas de saída. Explique como implementar essa idéia, de tal forma que ela seja executada no tempo  $O(V + E)$ . O que acontecerá a esse algoritmo se  $G$  tiver ciclos?

## 22.5 Componentes fortemente conectados

Agora, vamos considerar uma aplicação clássica de busca em profundidade: a decomposição de um grafo orientado em seus componentes fortemente conectados. Esta seção mostra como fazer essa decomposição usando duas pesquisas primeiro na profundidade. Muitos algoritmos que funcionam com grafos orientados começam por uma decomposição desse tipo. Após a decomposição, o algoritmo é executado separadamente em cada componente fortemente conectado. As soluções são então combinadas de acordo com a estrutura de conexões entre componentes.

Vimos no Apêndice B que um componente fortemente conectado de um grafo orientado  $G = (V, E)$  é um conjunto máximo de vértices  $C \subseteq V$  tal que, para todo par de vértices  $u$  e  $v$  em  $C$ , temos ao mesmo tempo  $u \sim v$  e  $v \sim u$ ; isto é, os vértices  $u$  e  $v$  são acessíveis um a partir do outro. A Figura 22.9 mostra um exemplo.

Nosso algoritmo para localização de componentes fortemente conectados de um grafo  $G = (V, E)$  usa a transposta de  $G$ , que é definida no Exercício 22.1-3 como o grafo  $G^T = (V, E^T)$ , onde  $E^T = \{(u, v) : (v, u) \in E\}$ . Ou seja,  $E^T$  consiste nas arestas de  $G$  com seus sentidos invertidos. Dada uma representação de lista de adjacências de  $G$ , o tempo de criar  $G^T$  é  $O(V + E)$ . É interessante observar que  $G$  e  $G^T$  têm exatamente os mesmos componentes fortemente conectados:  $u$  e  $v$  são acessíveis um a partir do outro em  $G$  se e somente se eles são acessíveis um a partir do outro em  $G^T$ . A Figura 22.9(b) mostra a transposta do grafo na Figura 22.9(a), com os componentes fortemente conectados sombreados.

O algoritmo de tempo linear (isto é, de tempo  $\Theta(V + E)$ ) apresentado a seguir calcula os componentes fortemente conectados de um grafo orientado  $G = (V, E)$  usando duas pesquisas primeiro na profundidade, uma sobre  $G$  e uma sobre  $G^T$ .

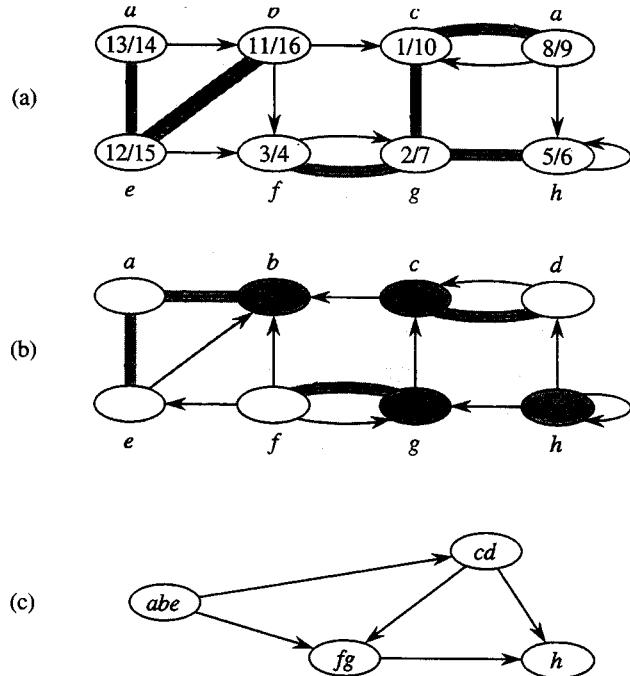


FIGURA 22.9 (a) Um grafo orientado  $G$ . Os componentes fortemente conectados de  $G$  são mostrados como regiões sombreadas. Cada vértice é identificado com seu tempo de descoberta e de término. As arestas de árvore são sombreadas. (b) O grafo  $G^T$ , a transposta de  $G$ . A árvore primeiro na profundidade calculada na linha 3 de STRONGLY-CONNECTED-COMPONENTS é mostrada, com arestas de árvore sombreadas. Cada componente fortemente conectado corresponde a uma árvore primeiro na profundidade. Os vértices  $b, c, g$  e  $h$ , que são fortemente sombreados, são as raízes das árvores primeiro na profundidade produzidas pela busca em profundidade de  $G^T$ . (c) O grafo de componentes acíclicos  $G^{SCC}$  obtido pela condensação de cada componente fortemente conectado de  $G$ , de modo que apenas um único vértice permaneça em cada componente

#### STRONGLY-CONNECTED-COMPONENTS( $G$ )

- 1 chamar  $\text{DFS}(G)$  para calcular o tempo de término  $f[u]$  para cada vértice  $u$
- 2 calcular  $G^T$
- 3 chamar  $\text{DFS}(G^T)$  mas, no loop principal de DFS, considerar os vértices em ordem decrescente de  $f[u]$  (calculada na linha 1)
- 4 dar saída aos vértices de cada árvore na floresta primeiro na profundidade formada na linha 3 como um componente fortemente conectado separado

A idéia por trás desse algoritmo vem de uma propriedade fundamental do **grafo de componentes**  $G^{SCC} = (V^{SCC}, E^{SCC})$ , que definimos como a seguir. Suponha que  $G$  tenha componentes fortemente conectados  $C_1, C_2, \dots, C_k$ . O conjunto de vértices  $V^{SCC}$  é  $\{v_1, v_2, \dots, v_k\}$  e contém um vértice  $v_i$  para cada componente fortemente conectado  $C_i$  de  $G$ . Existe uma aresta  $(v_i, v_j) \in E^{SCC}$  se  $G$  contém uma aresta orientada  $(x, y)$  para algum  $x \in C_i$  e algum  $y \in C_j$ . Visto de outro modo, pela contração de todas as arestas cujos vértices incidentes estão dentro do mesmo componente fortemente conectado de  $G$ , o grafo resultante é  $G^{SCC}$ . A Figura 22.9(c) mostra o grafo de componentes do grafo da Figura 22.9(a).

A propriedade fundamental é que o grafo de componentes é um gao, o que implica o lema a seguir.

#### Lema 22.13

Sejam  $C$  e  $C'$  componentes fortemente conectados distintos em um grafo orientado  $G = (V, E)$ , seja  $u, v \in C$ , seja  $u', v' \in C'$  e suponha que exista um caminho  $u \rightsquigarrow u'$  em  $G$ . Então, não pode haver também um caminho  $v' \rightsquigarrow v$  em  $G$ .

**Prova** Se existe um caminho  $v' \rightsquigarrow v$  em  $G$ , então existem caminhos  $u \rightsquigarrow u' \rightsquigarrow v'$  e  $v' \rightsquigarrow v \rightsquigarrow u$  em  $G$ . Desse modo,  $u$  e  $v'$  são acessíveis a partir um do outro, contradizendo assim a hipótese de que  $C$  e  $C'$  são componentes fortemente conectados distintos. ■

Veremos que, considerando vértices na segunda busca em profundidade em ordem decrescente dos tempos de término que foram calculados na primeira busca em profundidade, estamos, em essência, alcançando os vértices do grafo de componentes (cada um dos quais corresponde a um componente fortemente conectado de  $G$ ) em seqüência ordenada topologicamente.

Pelo fato de STRONGLY-CONNECTED-COMPONENTS executar duas pesquisas primeiro na profundidade, existe o potencial para ambigüidade quando discutimos  $d[u]$  ou  $f[u]$ . Nesta seção, esses valores sempre se referem aos tempos de descoberta e término calculados pela primeira chamada de DFS, na linha 1.

Estendemos a notação de tempos de descoberta e término aos conjuntos de vértices. Se  $U \subseteq V$ , então definimos  $d(U) = \min_{u \in U} d(u)$  e  $f(U) = \max_{u \in U} \{f(u)\}$ . Isto é,  $d(U)$  e  $f(U)$  são o tempo de descoberta mais antigo e o tempo de término mais recente, respectivamente, de qualquer vértice em  $U$ .

O lema a seguir e seu corolário fornecem uma propriedade fundamental relacionada a componentes fortemente conectados e tempos de término na busca em profundidade.

#### Lema 22.14

Sejam  $C$  e  $C'$  componentes distintos fortemente conectados em grafo orientado  $G = (V, E)$ . Suponha que exista uma aresta  $(u, v) \in E$ , onde  $u \in C$  e  $v \in C'$ . Então,  $f(C) > f(C')$ .

**Prova** Há dois casos, dependendo de qual componente fortemente conectado,  $C$  ou  $C'$ , teve o primeiro vértice descoberto durante a busca em profundidade.

Se  $d(C) < d(C')$ , seja  $x$  o primeiro vértice descoberto em  $C$ . No tempo  $d[x]$ , todos os vértices em  $C$  e  $C'$  são brancos. Existe um caminho em  $G$  de  $x$  até cada vértice em  $C$  consistindo apenas em vértices brancos. Como  $(u, v) \in E$ , para qualquer vértice  $w \in C'$ , também existe um caminho no tempo  $d[x]$  de  $x$  até  $w$  em  $G$  que consiste apenas em vértices brancos:  $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$ . Pelo teorema do caminho branco, todos os vértices em  $C$  e  $C'$  se tornam descendentes de  $x$  na árvore primeiro na profundidade. Pelo Corolário 22.8,  $f[x] = f(C) > f(C')$ .

Se, em vez disso, tivermos  $d(C) > d(C')$ , seja  $y$  o primeiro vértice descoberto em  $C'$ . No tempo  $d[y]$ , todos os vértices em  $C'$  são brancos e existe um caminho em  $G$  de  $y$  para até cada vértice em  $C$  consistindo apenas em vértices brancos. Pelo teorema do caminho branco, todos os vértices em  $C$  se tornam descendentes de  $y$  na árvore primeiro na profundidade e, pelo Corolário 22.8,  $f[y] = f(C')$ . No tempo  $d[y]$ , todos os vértices em  $C$  são brancos. Como existe uma aresta  $(u, v)$  de  $C$  até  $C'$ , o Lema 22.13 implica que não pode existir um caminho de  $C'$  até  $C$ . Conseqüentemente, nenhum vértice em  $C$  é acessível a partir de  $y$ . Portanto, no tempo  $f[y]$ , todos os vértices em  $C$  ainda são brancos. Desse modo, para qualquer vértice  $w \in C$ , temos  $f[w] > f[y]$ , o que implica que  $f(C) > f(C')$ . ■

O corolário a seguir nos diz que cada aresta em  $G^T$  entre diferentes componentes fortemente conectados vai de um componente com um tempo de término anterior (na primeira busca em profundidade) até um componente com um tempo de término posterior.

#### Corolário 22.15

Sejam  $C$  e  $C'$  componentes fortemente conectados distintos no grafo orientado  $G = (V, E)$ . Suponha que exista uma aresta  $(u, v) \in E^T$ , onde  $u \in C$  e  $v \in C'$ . Então,  $f(C) < f(C')$ .

**Prova** Como  $(u, v) \in E^T$ , temos  $(v, u) \in E$ . Tendo em vista que os componentes fortemente conectados de  $G$  e  $G^T$  são os mesmos, o Lema 22.14 implica que  $f(C) < f(C')$ . ■

O Corolário 22.15 fornece a chave para se compreender por que o procedimento STRONGLY-CONNECTED-COMPONENTS funciona. Vamos examinar o que acontece quando executamos a segunda busca em profundidade, que está em  $G^T$ . Começamos com o componente fortemente conectado  $C$  cujo tempo de término  $f(C)$  é máximo. A pesquisa começa em algum vértice  $x \in C$  e alcança todos os vértices em  $C$ . Pelo Corolário 22.15, não há nenhuma aresta em  $G^T$  de  $C$  para qualquer outro componente fortemente conectado, e assim a pesquisa de  $x$  não alcançará vértices em qualquer outro componente. Desse modo, a árvore com raiz em  $x$  contém exatamente os vértices de  $C$ . Tendo completado a visitação a todos os vértices em  $C$ , a pesquisa na linha 3 seleciona como raiz um vértice de algum outro componente fortemente conectado  $C'$  cujo tempo de término  $f(C')$  é máximo sobre todos os outros componentes além de  $C$ . Mais uma vez, a pesquisa alcançará todos os vértices em  $C'$  mas, pelo Corolário 22.15, as únicas arestas em  $G^T$  de  $C'$  para qualquer outro componente deve ser para  $C$ , que já alcançamos. Em geral, quando a busca em profundidade de  $G^T$  na linha 3 alcança qualquer componente fortemente conectado, quaisquer arestas de saída desse componente devem ser para componentes que já foram alcançados. Então, cada árvore primeiro na profundidade será exatamente um componente fortemente conectado. O teorema a seguir formaliza esse argumento.

### **Teorema 22.16**

STRONGLY-CONNECTED-COMPONENTS( $G$ ) calcula corretamente os componentes fortemente conectados de um grafo orientado  $G$ .

**Prova** Mostramos por indução sobre o número de árvores primeiro na profundidade encontradas na busca em profundidade de  $G^T$  da linha 3 que os vértices de cada árvore formam um componente fortemente conectado. A hipótese indutiva é que as primeiras  $k$  árvores produzidas na linha 3 são componentes fortemente conectados. A base para a indução, quando  $k = 0$ , é trivial.

Na etapa indutiva, supomos que cada uma das  $k$  primeiras árvores primeiro na profundidade produzidas na linha 3 é um componente fortemente conectado, e consideramos a  $(k + 1)$ -ésima árvore produzida. Seja a raiz dessa árvore o vértice  $u$  do componente fortemente conectado  $C$ . Devido ao modo como escolhemos raízes na busca em profundidade na linha 3,  $f[u] = f(C) > f(C')$  para qualquer componente fortemente conectado  $C'$  diferente de  $C$  que ainda tenha de ser alcançado. Pela hipótese indutiva, no momento em que a pesquisa alcança  $u$ , todos os outros vértices de  $C$  são brancos. Então, pelo teorema do caminho branco, todos os outros vértices de  $C$  são descendentes de  $u$  nessa árvore primeiro na profundidade. Além disso, pela hipótese indutiva e pelo Corolário 22.15, todas as arestas de  $G^T$  que saem de  $C$  devem ser para componentes fortemente conectados que já foram alcançados. Desse modo, nenhum vértice em um componente fortemente conectado diferente de  $C$  será um descendente de  $u$  durante a busca em profundidade de  $G^T$ . Portanto, os vértices da árvore primeiro na profundidade em  $G^T$  que têm raiz em  $u$  formam exatamente um componente fortemente conectado, o que completa a etapa indutiva e a prova. ■

Aqui está outro modo de ver como opera a segunda busca em profundidade. Considere o grafo de componentes  $(G^T)^{SCC}$  de  $G^T$ . Se mapearmos cada componente fortemente conectado alcançado na segunda busca em profundidade para um vértice de  $(G^T)^{SCC}$ , os vértices de  $(G^T)^{SCC}$  são alcançados na ordem inversa de uma sequência ordenada topologicamente. Se invertermos as arestas de  $(G^T)^{SCC}$ , obteremos o grafo  $((G^T)^{SCC})^T$ . Como  $((G^T)^{SCC})^T = (G^T)^{SCC}$  (veja o Exercício 22.5-4), a segunda busca em profundidade alcança os vértices de  $G^{SCC}$  em sequência ordenada topologicamente.

## **Exercícios**

### **22.5-1**

De que maneira o número de componentes fortemente conectados de um grafo se altera se uma nova aresta é adicionada?

### 22.5-2

Mostre como o procedimento STRONGLY-CONNECTED-COMPONENTS funciona sobre o grafo da Figura 22.6. Especificamente, mostre os tempos de término calculados na linha 1 e a floresta produzida na linha 3. Suponha que o loop das linhas 5 a 7 de DFS considere os vértices em ordem alfabética e que as listas de adjacências estejam em ordem alfabética.

### 22.5-3

O professor Deaver afirma que o algoritmo para componentes fortemente conectados pode ser simplificado pelo uso do grafo original (em lugar da transposta) na segunda pesquisa primeiro na profundidade e pela varredura dos vértices na ordem de tempos de término *crescentes*. O professor está correto?

### 22.5-4

Prove que, para qualquer grafo orientado  $G$ , temos  $((G^T)^{SCC})^T = (G^T)^{SCC}$ . Isto é, a transposta do grafo de componentes de  $G^T$  é igual ao grafo de componentes de  $G$ .

### 22.5-5

Forneça um algoritmo de tempo  $O(V + E)$  para calcular o grafo de componentes de um grafo orientado  $G = (V, E)$ . Certifique-se de que existe no máximo uma aresta entre dois vértices no grafo de componentes que o seu algoritmo produz.

### 22.5-6

Dado um grafo orientado  $G = (V, E)$ , explique como criar outro grafo  $G' = (V, E')$  tal que (a)  $G'$  tenha os mesmos componentes fortemente conectados que  $G$ , (b)  $G'$  tenha o mesmo grafo de componentes que  $G$ , e (c)  $E'$  seja tão pequeno quanto possível. Descreva um algoritmo rápido para calcular  $G'$ .

### 22.5-7

Um grafo orientado  $G = (V, E)$  é dito **semiconectado** se, para todos os pares de vértices  $u, v \in V$ , temos  $u \sim v$  ou  $v \sim u$ . Forneça um algoritmo eficiente para determinar se  $G$  é ou não semiconectado. Prove que o algoritmo é correto e analise seu tempo de execução.

## Problemas

### 22-1 Classificação de arestas por busca em largura

Uma floresta primeiro na profundidade classifica as arestas de um grafo em arestas de árvore, de retorno, diretas e cruzadas. Uma árvore primeiro na extensão também pode ser usada para classificar as arestas acessíveis a partir da origem da pesquisa nas mesmas quatro categorias.

- a. Prove que, em uma busca em largura de um grafo não orientado, são válidas as propriedades a seguir:
  1. Não existem arestas de retorno nem arestas diretas.
  2. Para cada aresta de árvore  $(u, v)$ , temos  $d[v] = d[u] + 1$ .
  3. Para cada aresta cruzada  $(u, v)$ , temos  $d[v] = d[u]$  ou  $d[v] = d[u] + 1$ .
- b. Prove que em uma busca em largura de um grafo orientado, são válidas as propriedades a seguir:
  1. Não existem arestas diretas.
  2. Para cada aresta de árvore  $(u, v)$ , temos  $d[v] = d[u] + 1$ .
  3. Para cada aresta cruzada  $(u, v)$ , temos  $d[v] \leq d[u] + 1$ .
  4. Para cada aresta de retorno  $(u, v)$ , temos  $0 \leq d[v] < d[u]$ .

## 22-2 Pontos de articulação, pontes e componentes biconectados

Seja  $G = (V, E)$  um grafo conectado não orientado. Um **ponto de articulação** de  $G$  é um vértice cuja remoção desconecta  $G$ . Uma **ponte** de  $G$  é uma aresta cuja remoção desconecta  $G$ . Um **componente biconectado** de  $G$  é um conjunto máximo de arestas tal que duas arestas quaisquer no conjunto estão dispostas em um ciclo simples comum. A Figura 22.10 ilustra essas definições. Podemos determinar pontos de articulação, pontes e componentes biconectados empregando a busca em profundidade. Seja  $G_\pi = (V, E_\pi)$  uma árvore primeiro na profundidade de  $G$ .

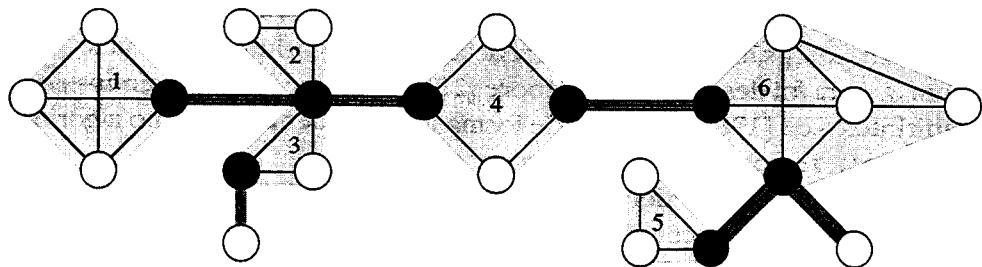


FIGURA 22.10 Os pontos de articulação, as pontes e os componentes biconectados de um grafo conectado não orientado para uso no Problema 22-2. Os pontos de articulação são os vértices fortemente sombreados, as pontes são as arestas fortemente sombreadas e os componentes biconectados são as arestas nas regiões sombreadas, com uma numeração  $bcc$  mostrada

- a. Prove que a raiz de  $G_\pi$  é um ponto de articulação de  $G$  se e somente se ele tem pelo menos dois filhos em  $G_\pi$ .
- b. Seja  $v$  um vértice não de raiz em  $G_\pi$ . Prove que  $v$  é um ponto de articulação de  $G$  se e somente se  $v$  tem um filho  $s$  tal que não existe nenhuma aresta de retorno de  $s$  ou de qualquer descendente de  $s$  para um ancestral próprio de  $v$ .
- c. Seja

$$\text{inferior}[v] = \begin{cases} d[v], \\ d[w] : (u, w) \text{ é uma aresta de retorno para algum descendente } u \text{ de } v. \end{cases}$$

Mostre como calcular  $\text{inferior}[v]$  para todos os vértices  $v \in V$  no tempo  $O(E)$ .

- d. Mostre como calcular todos os pontos de articulação no tempo  $O(E)$ .
- e. Prove que uma aresta de  $G$  é uma ponte se e somente se ela não reside em qualquer ciclo simples de  $G$ .
- f. Mostre como calcular todas as pontes de  $G$  no tempo  $O(E)$ .
- g. Prove que os componentes biconectados de  $G$  particionam as arestas não pontes de  $G$ .
- h. Forneça um algoritmo de tempo  $O(E)$  para identificar cada aresta  $e$  de  $G$  com um inteiro positivo  $bcc[e]$  tal que  $bcc[e] = bcc[e']$  se e somente se  $e$  e  $e'$  estão no mesmo componente biconectado.

## 22-3 Percurso de Euler

Um **percurso de Euler** de um grafo conectado orientado  $G = (V, E)$  é um ciclo que percorre cada aresta de  $G$  exatamente uma vez, embora possa alcançar um vértice mais de uma vez.

- a. Mostre que  $G$  tem um percurso de Euler se e somente se  $\text{grau de entrada}(v) = \text{grau de saída}(v)$  para cada vértice  $v \in V$ .
- b. Descreva um algoritmo de tempo  $O(E)$  para encontrar um percurso de Euler de  $G$  se existir um. (Sugestão: Intercale ciclos disjuntos de arestas.)

## 22.4 Acessibilidade

Seja  $G = (V, E)$  um grafo orientado no qual cada vértice  $u \in V$  é identificado com um inteiro exclusivo  $L(u)$  do conjunto  $\{1, 2, \dots, |V|\}$ . Para cada vértice  $u \in V$ , seja  $R(u) = \{v \in V : u \rightsquigarrow v\}$  o conjunto de vértices acessíveis a partir de  $u$ . Defina  $\min(u)$  como o vértice em  $R(u)$  cuja etiqueta é mínima, isto é,  $\min(u)$  é o vértice  $v$  tal que  $L(v) = \min\{L(w) : w \in R(u)\}$ . Forneça um algoritmo de tempo  $O(V + E)$  que calcule  $\min(u)$  para todos os vértices  $u \in V$ .

## Notas do capítulo

Even [87] e Tarjan [292] são excelentes referências sobre algoritmos de grafos.

A busca em largura foi descoberta por Moore [226] no contexto de caminhos de localização através de labirintos. Lee [198] descobriu de forma independente o mesmo algoritmo no contexto de roteamento de fios em placas de circuitos.

Hopcroft e Tarjan [154] defenderam o uso da representação de listas de adjacências sobre a representação de matriz de adjacências no caso de grafos esparsos e foram os primeiros a reconhecer a importância algorítmica da busca em profundidade. A busca em profundidade teve ampla utilização desde o final da década de 1950, especialmente em programas de inteligência artificial.

Tarjan [289] forneceu um algoritmo de tempo linear para encontrar componentes fortemente conectados. O algoritmo para componentes fortemente conectados da Seção 22.5 foi adaptado de Aho, Hopcroft e Ullman [6], que o creditam a S. R. Kosaraju (não publicado) e M. Sharir [276]. Gabow [101] também desenvolveu um algoritmo para componentes fortemente conectados baseado na contração de ciclos e utiliza duas pilhas para executá-lo em tempo linear. Knuth [182] foi o primeiro a fornecer um algoritmo de tempo linear para ordenação topológica.

---

## *Capítulo 23*

# *Árvores espalhadas mínimas*

No projeto de circuitos eletrônicos, freqüentemente é necessário tornar os pinos de vários componentes eletricamente equivalentes, juntando a fiação de todos eles. Para interconectar um conjunto de  $n$  pinos, podemos usar um arranjo de  $n - 1$  fios, cada qual conectando dois pinos. De todos os arranjos possíveis, aquele que utiliza a quantidade mínima de fio é normalmente o mais desejável.

Podemos modelar esse problema de fiação com um grafo conectado não orientado  $G = (V, E)$ , onde  $V$  é o conjunto de pinos,  $E$  é o conjunto de interconexões possíveis entre pares de pinos e, para cada aresta  $(u, v) \in E$ , temos um peso  $w(u, v)$  especificando o custo (a quantidade de fio necessária) para conectar  $u$  e  $v$ . Então, desejamos encontrar um subconjunto acíclico  $T \subseteq E$  que conecte todos os vértices e cujo peso total

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

seja minimizado. Tendo em vista que  $T$  é acíclico e conecta todos os vértices, ele deve formar uma árvore, que chamaremos **árvore espalhada**, pois “se estende” pela amplitude do grafo  $G$ . Chamamos o problema de determinar a árvore  $T$  de **problema da árvore espalhada mínima**.<sup>1</sup> A Figura 23.1 mostra um exemplo de grafo conectado e sua árvore espalhada mínima.

Neste capítulo, examinaremos dois algoritmos para resolver o problema da árvore espalhada mínima: o algoritmo de Kruskal e o algoritmo de Prim. Cada um deles pode ser criado com facilidade para ser executado no tempo  $O(E \lg V)$  com o uso de heaps binários ordinários. Usando heaps de Fibonacci, o algoritmo de Prim pode ser acelerado até ser executado no tempo  $O(E + V \lg V)$ , que representa uma melhoria se  $|V|$  é muito menor que  $|E|$ .

---

<sup>1</sup>A expressão “árvore espalhada mínima” é uma forma abreviada da expressão “árvore espalhada de peso mínimo”. Por exemplo, não estamos minimizando o número de arestas em  $T$ , pois todas as árvores espalhadas têm exatamente  $|V| - 1$  arestas, de acordo com o Teorema B.2.

Os dois algoritmos são algoritmos gulosos, como descreve o Capítulo 16. Em cada passo de um algoritmo, uma entre diversas opções possíveis deve ser escolhida. Os defensores da estratégia gulosa fazem a escolha que é a melhor no momento. Tal estratégia geralmente não oferece a garantia de encontrar soluções globalmente ótimas para problemas. Porém, no caso do problema da árvore espalhada mínima, podemos provar que certas estratégias gulosas produzem de fato uma árvore espalhada com peso mínimo. Embora este capítulo possa ser lido de forma independente do Capítulo 16, os métodos gulosos apresentados aqui são uma aplicação clássica das noções teóricas introduzidas naquele capítulo.

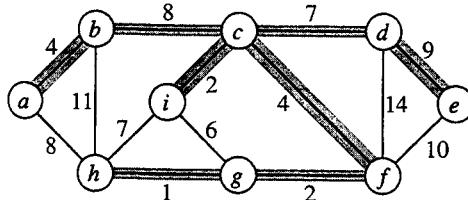


FIGURA 23.1 Uma árvore de amplitude mínima para um grafo conectado. Os pesos nas arestas são mostrados, e as arestas em uma árvore de amplitude mínima estão sombreadas. O peso total da árvore mostrada é 37. Essa árvore de amplitude mínima não é única: a remoção da aresta  $(b, c)$  e a sua substituição pela aresta  $(a, b)$  produz outra árvore de amplitude com peso 37

A Seção 23.1 introduz um algoritmo “genérico” de árvore espalhada mínima que faz uma árvore espalhada aumentar adicionando uma aresta de cada vez. A Seção 23.2 fornece duas maneiras de se implementar o algoritmo genérico. O primeiro algoritmo, devido a Kruskal, é semelhante ao algoritmo de componentes conexos da Seção 21.1. O segundo, devido a Prim, é semelhante ao algoritmo de caminhos mais curtos de Dijkstra (Seção 24.3).

## 23.1 Como aumentar uma árvore espalhada mínima

Vamos supor que temos um grafo conectado não orientado  $G = (V, E)$  com uma função peso  $w : E \rightarrow \mathbb{R}$  e desejamos encontrar uma árvore espalhada mínima correspondente a  $G$ . Os dois algoritmos que consideramos neste capítulo utilizam uma abordagem gulosa para o problema, embora sejam diferentes no modo como aplicam essa abordagem.

Essa estratégia gulosa é captada pelo algoritmo “genérico” a seguir, que aumenta a árvore espalhada mínima uma aresta de cada vez. O algoritmo administra um conjunto de arestas  $A$ , mantendo o seguinte loop invariante:

Antes de cada iteração,  $A$  é um subconjunto de alguma árvore espalhada mínima.

Em cada etapa, determinamos uma aresta  $(u, v)$  que pode ser adicionada a  $A$  sem violar esse invariante, no sentido de que  $A \cup \{(u, v)\}$  também é um subconjunto de uma árvore espalhada mínima. Chamamos tal aresta de **aresta segura** para  $A$ , pois ela pode ser adicionada com segurança a  $A$ , ao mesmo tempo que mantém o invariante.

**GENERIC-MST( $G, w$ )**

- 1  $A \leftarrow \emptyset$
- 2 **while**  $A$  não formar uma árvore espalhada
- 3   **do** encontrar uma aresta  $(u, v)$  que seja segura para  $A$
- 4      $A \leftarrow A \cup \{(u, v)\}$
- 5 **return**  $A$

**Inicialização:** Depois da linha 1, o conjunto  $A$  satisfaz de forma trivial ao loop invariante.

**Manutenção:** O loop nas linhas 2 a 4 mantém o invariante, adicionando apenas arestas seguras.

**Término:** Todas as arestas adicionadas a  $A$  estão em uma árvore espalhada mínima, e então o conjunto  $A$  retornado na linha 5 deve ser uma árvore espalhada mínima.

É claro que a parte complicada é encontrar uma aresta segura na linha 3. Deve existir uma, pois quando a linha 3 é executada, o invariante estabelece que existe uma árvore espalhada  $T$  tal que  $A \subseteq T$ . Dentro do corpo do loop **while**,  $A$  deve ser um subconjunto próprio de  $T$ , e então deve haver uma aresta  $(u, v) \in T$  tal que  $(u, v) \notin A$  e  $(u, v)$  é segura para  $A$ .

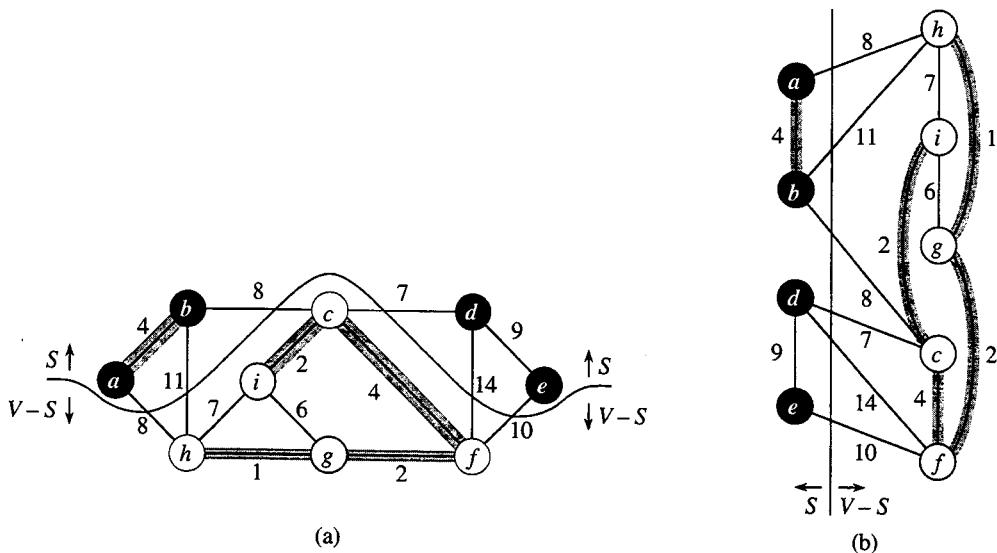


FIGURA 23.2 Duas maneiras de visualizar um corte  $(S, V - S)$  do grafo da Figura 23.1. (a) Os vértices no conjunto  $S$  são mostrados em preto e aqueles em  $V - S$  são mostrados em branco. As arestas que cruzam o corte são as que conectam vértices brancos com vértices pretos. A aresta  $(d, c)$  é a única aresta leve que cruza o corte. Um subconjunto  $A$  das arestas está sombreado; observe que o corte  $(S, V - S)$  respeita  $A$ , pois nenhuma aresta de  $A$  cruza o corte. (b) O mesmo grafo com os vértices no conjunto  $S$  à esquerda e os vértices no conjunto  $V - S$  à direita. Uma aresta cruza o corte se ela conecta o vértice à esquerda com um vértice à direita.

No restante desta seção, forneceremos uma regra (Teorema 23.1) para reconhecer arestas seguras. A próxima seção descreve dois algoritmos que usam essa regra para encontrar arestas seguras de modo eficiente.

Primeiro, precisamos de algumas definições. Um **corte**  $(S, V - S)$  de um grafo não orientado  $G = (V, E)$  é uma partição de  $V$ . A Figura 23.2 ilustra essa noção. Dizemos que uma aresta  $(u, v) \in E$  **cruza** o corte  $(S, V - S)$  se um de seus pontos extremos está em  $S$  e o outro está em  $V - S$ . Dizemos que um corte **respeita** o conjunto  $A$  de arestas se nenhuma aresta em  $A$  cruza o corte. Uma aresta é uma **aresta leve** cruzando um corte se seu peso é o mínimo de qualquer aresta que cruza o corte. Observe que pode existir mais de uma aresta leve cruzando um corte no caso de laços. De modo mais geral, dizemos que uma aresta é uma **aresta leve que satisfaz** a uma dada propriedade se seu peso é o mínimo de qualquer aresta que satisfaz à propriedade.

Nossa regra para reconhecer arestas seguras é dada pelo teorema a seguir.

### Teorema 23.1

Seja  $G = (V, E)$  um grafo conectado não orientado com uma função peso de valor real  $w$  definido em  $E$ . Seja  $A$  um subconjunto de  $E$  que está incluído em alguma árvore espalhada mínima correspondente a  $G$ , seja  $(S, V - S)$  qualquer corte de  $G$  que respeita  $A$  e seja  $(u, v)$  uma aresta leve cruzando  $(S, V - S)$ . Então, a aresta  $(u, v)$  é segura para  $A$ .

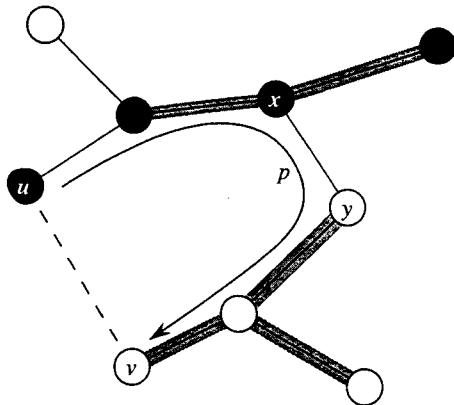


FIGURA 23.3 A prova do Teorema 23.1. Os vértices em  $S$  são pretos, e os vértices em  $V - S$  são brancos. As arestas na árvore espalhada mínima  $T$  são mostradas, mas as arestas no grafo não são. As arestas em  $A$  são sombreadas, e  $(u, v)$  é uma aresta leve que cruza o corte  $(S, V - S)$ . A aresta  $(x, y)$  é uma aresta no caminho único  $p$  de  $u$  até  $v$  em  $T$ . Uma árvore espalhada mínima  $T'$  que contém  $(u, v)$  é formada pela remoção da aresta  $(x, y)$  de  $T$  e pela adição da aresta  $(u, v)$

**Prova** Seja  $T$  uma árvore espalhada mínima que inclui  $A$ , e suponha que  $T$  não contém a aresta leve  $(u, v)$  pois, se ela contiver a aresta, terminamos. Vamos construir outra árvore espalhada mínima  $T'$  que inclui  $A \cup \{(u, v)\}$  usando uma técnica de recortar e colar, mostrando assim que  $(u, v)$  é uma aresta segura para  $A$ .

A aresta  $(u, v)$  forma um ciclo com as arestas no caminho  $p$  de  $u$  até  $v$  em  $T$ , como ilustra a Figura 23.3. Tendo em vista que  $u$  e  $v$  estão em lados opostos do corte  $(S, V - S)$ , existe no mínimo uma aresta em  $T$  no caminho  $p$  que também cruza o corte. Seja  $(x, y)$  qualquer dessas arestas. A aresta  $(x, y)$  não está em  $A$ , porque o corte respeita  $A$ . Como  $(x, y)$  está no caminho único de  $u$  até  $v$  em  $T$ , a remoção de  $(x, y)$  divide  $T$  em dois componentes. A adição de  $(u, v)$  os reconecta para formar uma nova árvore espalhada  $T' = T - (x, y) \cup \{(u, v)\}$ .

Em seguida, mostramos que  $T'$  é uma árvore espalhada mínima. Tendo em vista que  $(u, v)$  é uma aresta leve que cruza  $(S, V - S)$  e  $(x, y)$  também cruza esse corte,  $w(u, v) \leq w(x, y)$ . Então,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T). \end{aligned}$$

Porém,  $T$  é uma árvore espalhada mínima, de modo que  $w(T) \leq w(T')$ ; portanto,  $T'$  também deve ser uma árvore espalhada mínima.

Resta mostrar que  $(u, v)$  é realmente uma aresta segura para  $A$ . Temos  $A \subseteq T'$ , pois  $A \subseteq T$  e  $(x, y) \notin A$ ; assim,  $A \cup \{(u, v)\} \subseteq T'$ . Conseqüentemente, como  $T'$  é uma árvore de amplitude mínima,  $(u, v)$  é segura para  $A$ . ■

O Teorema 23.1 nos dá uma compreensão melhor do funcionamento do algoritmo GENERIC-MST sobre um grafo conectado  $G = (V, E)$ . À medida que o algoritmo progride, o conjunto  $A$  é sempre acíclico; caso contrário, uma árvore espalhada mínima incluindo  $A$  conteria um ciclo, o que é uma contradição. Em qualquer ponto na execução do algoritmo, o grafo  $G_A = (V, A)$  é uma floresta, e cada um dos componentes conexos de  $G_A$  é uma árvore. (Algumas das árvores podem conter apenas um vértice, como ocorre, por exemplo, quando o algoritmo começa:  $A$  está vazio e a floresta contém  $|V|$  árvores, uma para cada vértice.) Além disso, qualquer aresta segura  $(u, v)$  correspondente a  $A$  conecta componentes distintos de  $G_A$ , pois  $A \cup \{(u, v)\}$  deve ser acíclico.

O loop das linhas 2 a 4 de GENERIC-MST é executado  $|V| - 1$  vezes à medida que cada uma das  $|V| - 1$  arestas de uma árvore espalhada mínima é sucessivamente determinada. No início,

quando  $A = 0$ , existem  $|V|$  árvores em  $G_A$ , e cada iteração reduz esse número em uma unidade. Quando a floresta contém apenas uma única árvore, o algoritmo termina.

Os dois algoritmos da Seção 23.2 utilizam o corolário do Teorema 23.1 apresentado a seguir.

### **Corolário 23.2**

Seja  $G = (V, E)$  um grafo conectado não orientado com uma função de peso de valor real  $w$  definida em  $E$ . Seja  $A$  um subconjunto de  $E$  que está incluído em alguma árvore espalhada mínima para  $G$ , e seja  $C = (V_C, E_C)$  um componente conectado (árvore) na floresta  $G_A = (V, A)$ . Se  $(u, v)$  é uma aresta leve conectando  $C$  a algum outro componente em  $G_A$ , então  $(u, v)$  é segura para  $A$ .

**Prova** O corte  $(V_C, V - V_C)$  respeita  $A$ , e  $(u, v)$  é então uma aresta leve para esse corte. Então,  $(u, v)$  é segura para  $A$ . ■

## **Exercícios**

### **23.1-1**

Seja  $(u, v)$  uma aresta de peso mínimo em um grafo  $G$ . Mostre que  $(u, v)$  pertence a alguma árvore espalhada mínima de  $G$ .

### **23.1-2**

O professor Sabatier supõe a recíproca do Teorema 23.1 dada a seguir. Seja  $G = (V, E)$  um grafo conectado não orientado com uma função peso de valor real  $w$  definida em  $E$ . Seja  $A$  um subconjunto de  $E$  que está incluído em alguma árvore espalhada mínima para  $G$ , seja  $(S, V - S)$  qualquer corte de  $G$  que respeita  $A$ , e seja  $(u, v)$  uma aresta segura para  $A$  que cruza  $(S, V - S)$ . Então,  $(u, v)$  é uma aresta leve para o corte. Mostre que a hipótese do professor é incorreta, fornecendo um contra-exemplo.

### **23.1-3**

Mostre que, se uma aresta  $(u, v)$  está contida em alguma árvore espalhada mínima, então ela é uma aresta leve que cruza algum corte do grafo.

### **23.1-4**

Forneça um exemplo simples de um grafo conectado tal que o conjunto de arestas  $\{(u, v) : \text{existe um corte } (S, V - S) \text{ tal que } (u, v) \text{ é uma aresta leve cruzando } (S, V - S)\}$  não forma uma árvore espalhada mínima.

### **23.1-5**

Seja  $e$  uma aresta de peso máximo em algum ciclo de  $G = (V, E)$ . Prove que existe uma árvore espalhada mínima de  $G' = (V, E - \{e\})$  que também é uma árvore espalhada mínima de  $G$ .

### **23.1-6**

Mostre que um grafo tem uma árvore espalhada mínima única se, para todo corte do grafo, existe uma aresta leve única cruzando o corte. Mostre que a recíproca não é verdadeira, fornecendo um contra-exemplo.

### **23.1-7**

Mostre que, se todos os pesos de arestas de um grafo são positivos, então qualquer subconjunto de arestas que conecte todos os vértices e tenha peso total mínimo deve ser uma árvore. Forneça um exemplo para mostrar que a mesma conclusão não decorre se permitirmos que alguns pesos sejam não positivos.

### **23.1-8**

Seja  $T$  uma árvore espalhada mínima de um grafo  $G$ , e seja  $L$  a lista ordenada dos pesos das arestas de  $T$ . Mostre que, para qualquer outra árvore espalhada mínima  $T'$  de  $G$ , a lista  $L$  também é a lista ordenada de pesos de arestas de  $T'$ .

### 23.1-9

Seja  $T$  uma árvore espalhada mínima de um grafo  $G = (V, E)$ , e seja  $V'$  um subconjunto de  $V$ . Seja  $T'$  o subgrafo de  $T$  induzido por  $V'$ , e seja  $G'$  o subgrafo de  $G$  induzido por  $V'$ . Mostre que, se  $T'$  é conectado, então  $T'$  é uma árvore de amplitude mínima de  $G'$ .

### 23.1-10

Dado um grafo  $G$  e uma árvore espalhada mínima  $T$ , suponha que diminuíssemos o peso de uma das arestas em  $T$ . Mostre que  $T$  ainda é uma árvore de amplitude mínima para  $G$ . Mais formalmente, seja  $T$  uma árvore espalhada mínima para  $G$  com pesos de arestas dados pela função peso  $w$ . Escolha uma aresta  $(x, y) \in T$  e um número positivo  $k$ , e defina a função peso  $w'$  por

$$w'(u, v) = \begin{cases} w(u, v) & \text{se } (u, v) \neq (x, y), \\ w(x, y) + k & \text{se } (u, v) = (x, y). \end{cases}$$

Mostre que  $T$  é uma árvore espalhada mínima para  $G$  com pesos de arestas dados por  $w'$ .

### 23.1-11 \*

Dado um grafo  $G$  e uma árvore espalhada mínima  $T$ , suponha que diminuímos o peso de uma das arestas não presentes em  $T$ . Forneça um algoritmo para encontrar a árvore espalhada mínima no grafo modificado.

## 23.2 Os algoritmos de Kruskal e Prim

Os dois algoritmos de árvore espalhada mínima descritos nesta seção são elaborações do algoritmo genérico. Cada um deles utiliza uma regra específica para determinar uma aresta segura na linha 3 de GENERIC-MST. No algoritmo de Kruskal, o conjunto  $A$  é uma floresta. A aresta segura adicionada a  $A$  é sempre uma aresta de peso mínimo no grafo que conecta dois componentes distintos. No algoritmo de Prim, o conjunto  $A$  forma uma árvore única. A aresta segura adicionada a  $A$  é sempre uma aresta de peso mínimo que conecta a árvore a um vértice não presente na árvore.

### Algoritmo de Kruskal

O algoritmo de Kruskal se baseia diretamente no algoritmo genérico de árvore espalhada mínima dado na Seção 23.1. Ele encontra uma aresta segura para adicionar à floresta crescente encontrando, de todas as arestas que conectam duas árvores quaisquer na floresta, uma aresta  $(u, v)$  de peso mínimo. Sejam  $C_1$  e  $C_2$  as duas árvores conectadas por  $(u, v)$ . Tendo em vista que  $(u, v)$  deve ser uma aresta leve conectando  $C_1$  a alguma outra árvore, o Corolário 23.2 implica que  $(u, v)$  é uma aresta segura para  $C_1$ . O algoritmo de Kruskal é um algoritmo guloso, porque em cada etapa ele adiciona à floresta uma árvore de peso mínimo possível.

Nossa implementação de algoritmo de Kruskal é semelhante ao algoritmo para calcular componentes conexos da Seção 21.1. Ele utiliza uma estrutura de dados de conjuntos disjuntos para manter vários conjuntos disjuntos de elementos. Cada conjunto contém os vértices em uma árvore da floresta atual. A operação FIND-SET( $u$ ) retorna um elemento representativo do conjunto que contém  $u$ . Portanto, podemos determinar se dois vértices  $u$  e  $v$  pertencem à mesma árvore testando se FIND-SET( $u$ ) é igual a FIND-SET( $v$ ). A combinação de árvores é realizada pelo procedimento UNION.

MST-KRUSKAL( $G, w$ )

- 1  $A \leftarrow \emptyset$
- 2 **for** cada vértice  $v \in V[G]$
- 3     **do** MAKE-SET( $v$ )

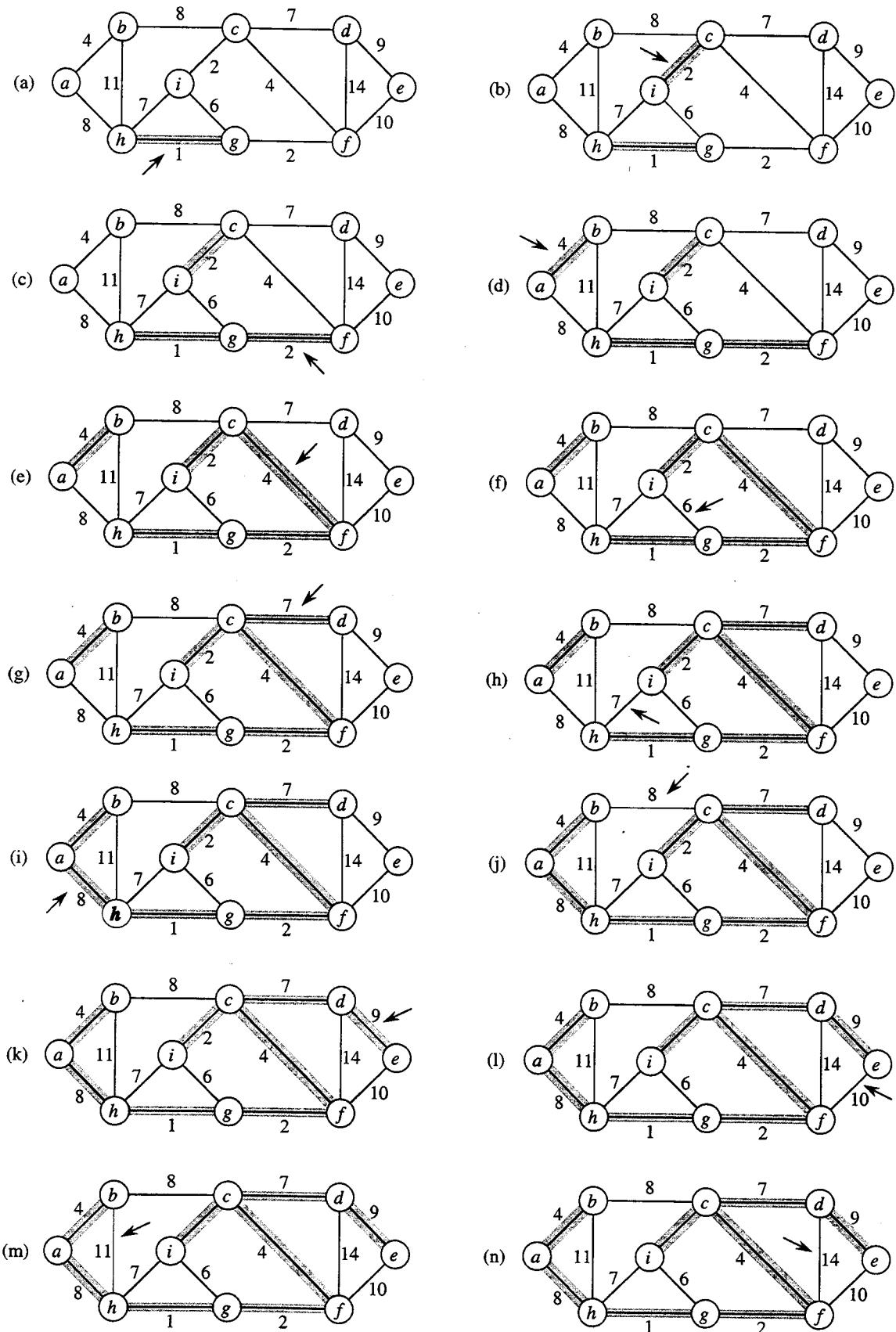


FIGURA 23.4 A execução do algoritmo de Kruskal sobre o grafo da Figura 23.1. As arestas sombreadas pertencem à floresta que está sendo aumentada. As arestas são consideradas pelo algoritmo em seqüência ordenada por peso. Uma seta aponta para a aresta que está sendo considerada em cada etapa do algoritmo. Se a aresta une duas árvores distintas na floresta, ela é adicionada à floresta, intercalando assim as duas árvores

```

4 ordenar as arestas de  $E$  por peso  $w$  não decrescente
5 for cada aresta  $(u, v) \in E$ , em ordem de peso não decrescente
6   do if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7     then  $A \leftarrow A \cup \{(u, v)\}$ 
8       UNION( $u, v$ )
9 return  $A$ 

```

O algoritmo de Kruskal funciona como mostra a Figura 23.4. As linhas 1 a 3 inicializam o conjunto  $A$  como o conjunto vazio e criam  $|V|$  árvores, cada uma contendo um vértice. As arestas em  $E$  são ordenadas em seqüência por peso não decrescente na linha 4. O loop **for** das linhas 5 a 8 verifica, para cada aresta  $(u, v)$ , se os pontos extremos  $u$  e  $v$  pertencem à mesma árvore. Se pertencerem, então a aresta  $(u, v)$  não poderá ser adicionada à floresta sem criar um ciclo, e a aresta será descartada. Caso contrário, os dois vértices pertencem a árvores diferentes. Nesse caso, a aresta  $(u, v)$  é adicionada a  $A$  na linha 7 e os vértices nas duas árvores são intercalados na linha 8.

O tempo de execução do algoritmo de Kruskal para um grafo  $G = (V, E)$  depende da implementação da estrutura de dados de conjuntos disjuntos. Vamos supor a implementação da floresta de conjuntos disjuntos da Seção 21.3 com as heurísticas de união por ordenação e compressão de caminho, pois ela é a implementação assintoticamente mais rápida conhecida. A inicialização do conjunto  $A$  na linha 1 demora o tempo  $O(1)$ , e o tempo para ordenar as arestas na linha 4 é  $O(E \lg E)$ . (Consideraremos em breve o custo das  $|V|$  operações MAKE-SET no loop **for** das linhas 2 a 3.) O loop **for** das linhas 5 a 8 executa  $O(E)$  operações FIND-SET e UNION sobre a floresta de conjuntos disjuntos. Juntamente com as  $|V|$  operações MAKE-SET, essas operações demoram ao todo o tempo  $O((V + E) \alpha(V))$ , onde  $\alpha$  é a função de crescimento muito lento definida na Seção 21.4. Pelo fato de  $G$  ser supostamente conectada, temos  $|E| \geq |V| - 1$ , e assim as operações de conjuntos disjuntos demoram o tempo  $O(E \alpha(V))$ . Além disso, tendo em vista que  $\alpha(|V|) = O(\lg V) = O(\lg E)$ , o tempo de execução total do algoritmo de Kruskal é  $O(E \lg E)$ . Observando que  $|E| < |V|^2$ , temos  $\lg |E| = O(\lg V)$ , e portanto podemos redefinir o tempo de execução do algoritmo de Kruskal como  $O(E \lg V)$ .

## Algoritmo de Prim

Como o algoritmo de Kruskal, o algoritmo de Prim é um caso especial do algoritmo genérico de árvore espalhada mínima da Seção 23.1. O algoritmo de Prim opera de modo muito semelhante ao algoritmo de Dijkstra para localizar caminhos mais curtos em um grafo, que veremos na Seção 24.3. O algoritmo de Prim tem a propriedade de que as arestas no conjunto  $A$  sempre formam uma árvore única. Conforme ilustramos na Figura 23.5, a árvore começa a partir de um vértice de raiz arbitrária  $r$  e aumenta até a árvore alcançar todos os vértices em  $V$ . Em cada etapa, é adicionada à árvore  $A$  uma aresta leve que conecta  $A$  a um vértice isolado de  $G_A = (V, A)$ . Pelo Corolário 23.2, essa regra adiciona apenas arestas que são seguras para  $A$ ; desse modo, quando o algoritmo termina, as arestas em  $A$  formam uma árvore espalhada mínima. Essa estratégia é gula, porque a árvore é aumentada em cada etapa com uma aresta que contribui com a quantidade mínima possível para o peso da árvore.

A chave para implementar o algoritmo de Prim de forma eficiente é tornar fácil a seleção de uma nova aresta a ser adicionada à árvore formada pelas arestas em  $A$ . No pseudocódigo a seguir, o grafo conectado  $G$  e a raiz  $r$  da árvore espalhada mínima a ser aumentada são entradas para o algoritmo. Durante a execução do algoritmo, todos os vértices que *não* estão na árvore residem em uma fila de prioridade mínima  $Q$  baseada em um campo *chave*. Para cada vértice  $v$ ,  $chave[v]$  é o peso mínimo de qualquer aresta que conecta  $v$  a um vértice na árvore; por convenção,  $chave[v] = \infty$  se não existe nenhuma aresta desse tipo. O campo  $\pi[v]$  denomina o “pai” de  $v$  na árvore. Durante o algoritmo, o conjunto  $A$  de GENERIC-MST é mantido implicitamente como

452  $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$ .

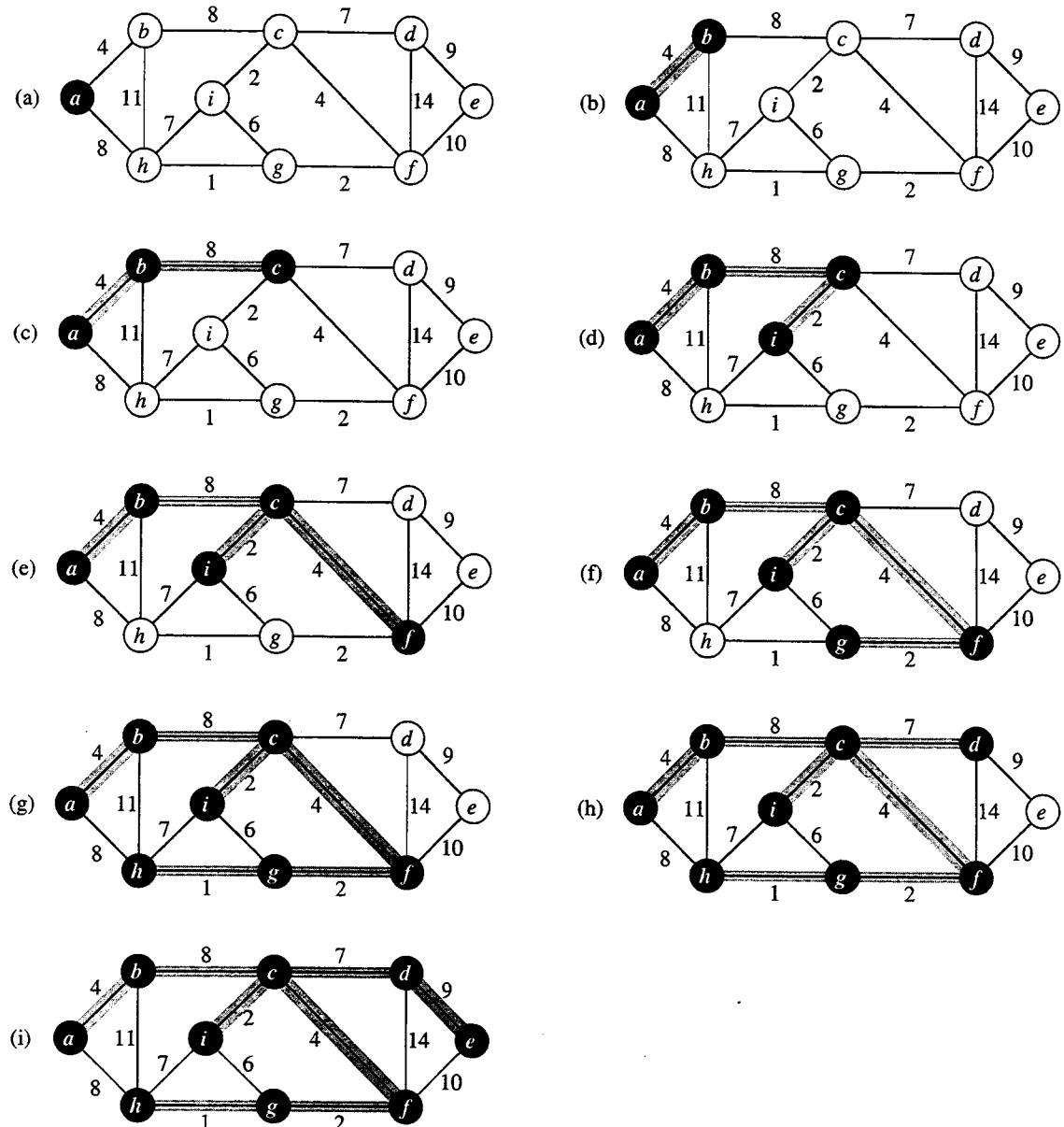


FIGURA 23.5 A execução do algoritmo de Prim sobre o grafo da Figura 23.1. O vértice de raiz é  $a$ . Areias sombreadas estão na árvore que está sendo aumentada, e os vértices na árvore são mostrados em preto. Em cada etapa do algoritmo, os vértices na árvore determinam um corte do grafo, e uma aresta leve cruzando o corte é acrescentada à árvore. Na segunda etapa, por exemplo, o algoritmo tem a opção de adicionar a aresta  $(b, c)$  ou a aresta  $(a, b)$  à árvore, desde que ambas sejam arestas leves cruzando o corte

Quando o algoritmo termina, a fila de prioridades  $Q$  está vazia; a árvore espalhada mínima  $A$  para  $G$  é portanto

$$A = \{(v, \pi[v]) : v \in V - \{r\}\}.$$

MST-PRIM( $G, w, r$ )

- 1 **for** cada  $u \in V[G]$
- 2     **do**  $chave[u] \leftarrow \infty$
- 3          $\pi[u] \leftarrow \text{NIL}$
- 4      $chave[r] \leftarrow 0$
- 5      $Q \leftarrow V[G]$

```

6 while  $Q \neq 0$ 
7   do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
8     for cada  $v \in \text{Adj}[u]$ 
9       do if  $v \in Q$  e  $w(u, v) < \text{chave}[v]$ 
10      then  $\pi[v] \leftarrow u$ 
11       $\text{chave}[v] \leftarrow w(u, v)$ 

```

O algoritmo de Prim funciona conforme mostra a Figura 23.5. As linhas 1 a 5 definem a chave de cada vértice como  $\infty$  (com exceção da raiz  $r$ , cuja chave é definida como 0, de forma que será o primeiro vértice processado), inicializa o pai de cada vértice como NIL e inicializa a fila de prioridade mínima  $Q$  para conter todos os vértices. O algoritmo mantém o seguinte loop invariante de três partes:

Antes de cada iteração do loop **while** das linhas 6 a 11,

1.  $A = (v, \pi[v]) : v \in V - \{r\} - Q\}$ .
2. Os vértices já colocados na árvore espalhada mínima são aqueles em  $V - Q$ .
3. Para todos os vértices  $v \in Q$ , se  $\pi[v] \neq \text{NIL}$ , então  $\text{chave}[v] < \infty$  e  $\text{chave}[v]$  é o peso de uma aresta leve  $(v, \pi[v])$  que conecta  $v$  a algum vértice já inserido na árvore espalhada mínima.

A linha 7 identifica um vértice  $u \in Q$  incidente em uma aresta leve que cruza o corte  $(V - Q, Q)$  (com exceção da primeira iteração, na qual  $u = r$  devido à linha 4). A remoção de  $u$  do conjunto  $Q$  o acrescenta ao conjunto  $V - Q$  de vértices na árvore, adicionando assim  $(u, \pi[u])$  a  $A$ . O loop **for** das linhas 8 a 11 atualiza os campos *chave* e  $\pi$  de cada vértice  $v$  adjacente a  $u$ , mas não na árvore. A atualização mantém a terceira parte do loop invariante.

O desempenho do algoritmo de Prim depende de como implementamos a fila de prioridades  $Q$ . Se  $Q$  é implementada como um heap mínimo binário (ver Capítulo 6), podemos usar o procedimento BUILD-MIN-HEAP para executar a inicialização nas linhas 1 a 5 no tempo  $O(V)$ . O corpo do loop **while** é executado  $|V|$  vezes, e como cada operação de EXTRACT-MIN demora o tempo  $O(\lg V)$ , o tempo total para todas as chamadas a EXTRACT-MIN é  $O(V \lg V)$ . O loop **for** nas linhas 8 a 11 é executado completamente  $O(E)$  vezes, pois a soma dos comprimentos de todas as listas de adjacências é  $2 |E|$ . Dentro do loop **for**, o teste de pertinência a  $Q$  na linha 9 pode ser implementado em tempo constante, mantendo-se um bit para cada vértice que informa se ele está ou não em  $Q$ , e atualizando-se o bit quando o vértice é removido de  $Q$ . A atribuição na linha 11 envolve uma operação implícita de DECREASE-KEY sobre o heap mínimo, a qual pode ser implementada em um heap mínimo binário no tempo  $O(\lg V)$ . Desse modo, o tempo total para o algoritmo de Prim é  $O(V \lg V + E \lg V)$   $O(E \lg V)$ , que é assintoticamente igual ao da nossa implementação do algoritmo de Kruskal.

Contudo, o tempo de execução assintótico do algoritmo de Prim pode ser melhorado, usando-se heaps de Fibonacci. O Capítulo 20 mostra que, se  $|V|$  elementos estão organizados em um heap de Fibonacci, podemos executar uma operação EXTRACT-MIN em tempo amortizado  $O(\lg V)$  e uma operação DECREASE-KEY (para implementar a linha 11) no tempo amortizado  $O(1)$ . Então, se usarmos um heap de Fibonacci para implementar a fila de prioridade mínima  $Q$ , o tempo de execução do algoritmo de Prim melhora até  $O(E + V \lg V)$ .

## Exercícios

### 23.2-1

O algoritmo de Kruskal pode retornar diferentes árvores espalhadas para o mesmo grafo de entrada  $G$ , dependendo de como os laços são rompidos quando as arestas são ordenadas. Mostre que, para cada árvore espalhada mínima  $T$  de  $G$ , existe um modo de ordenar as arestas de  $G$  no algoritmo de Kruskal, de tal forma que o algoritmo retorne  $T$ .

### 23.2-2

Suponha que o grafo  $G = (V, E)$  seja representado como uma matriz de adjacências. Forneça uma implementação simples do algoritmo de Prim para esse caso que seja executada no tempo  $O(V^2)$ .

### 23.2-3

A implementação de heap de Fibonacci do algoritmo de Prim é assintoticamente mais rápida que a implementação de heap binário para um grafo esparsa  $G = (V, E)$ , onde  $|E| = \Theta(V)$ ? E no caso de um grafo denso, onde  $|E| = \Theta(V^2)$ ? De que modo  $|E|$  e  $|V|$  devem estar relacionados para que a implementação de heap de Fibonacci seja assintoticamente mais rápida que a implementação de heap binário?

### 23.2-4

Suponha que todos os pesos de arestas em um grafo sejam inteiros no intervalo de 1 a  $|V|$ . Com que rapidez é possível executar o algoritmo de Kruskal? E se os pesos de arestas forem inteiros no intervalo de 1 a  $W$  para alguma constante  $W$ ?

### 23.2-5

Suponha que todos os pesos de arestas em um grafo sejam inteiros no intervalo de 1 a  $|V|$ . Com que rapidez é possível executar o algoritmo de Prim? E se os pesos de arestas forem inteiros no intervalo de 1 a  $W$  para alguma constante  $W$ ?

### 23.2-6 \*

Suponha que os pesos de arestas em um grafo estejam uniformemente distribuídos sobre o intervalo fechado  $[0, 1]$ . Qual algoritmo, o de Kruskal ou o de Prim, pode tornar a execução mais rápida? Descreva um algoritmo eficiente que, dado um grafo não orientado  $G$ , determine uma árvore espalhada de  $G$  cujo maior peso de aresta seja mínimo sobre todas as árvores espalhadas de  $G$ .

### 23.2-7 \*

Suponha que um grafo  $G$  tenha uma árvore espalhada mínima já calculada. Com que rapidez a árvore espalhada mínima pode ser atualizada se um novo vértice e arestas incidentes são adicionados a  $G$ ?

### 23.2-8

O professor Toole propõe um novo algoritmo de dividir e conquistar para calcular árvores espalhadas mínimas, que apresentamos a seguir. Dado um grafo  $G = (V, E)$ , particione o conjunto  $V$  de vértices em dois conjuntos  $V_1$  e  $V_2$ , tais que  $|V_1|$  e  $|V_2|$  sejam diferentes por no máximo 1. Seja  $E_1$  o conjunto de arestas incidentes apenas em vértices de  $V_1$ , e seja  $E_2$  o conjunto de arestas incidentes em vértices de  $V_2$ . Resolva recursivamente um problema de árvore espalhada mínima sobre cada um dos dois subgrafos  $G_1 = (V_1, E_1)$  e  $G_2 = (V_2, E_2)$ . Finalmente, selecione a aresta de peso mínimo em  $E$  que cruze o corte  $(V_1, V_2)$  e use essa aresta para unir as duas árvores espalhadas mínimas resultantes em uma única árvore espalhada.

Demonstre que o algoritmo calcula corretamente uma árvore espalhada mínima de  $G$ , ou forneça um exemplo para o qual o algoritmo não funciona.

## Problemas

### 23.1 Segunda melhor árvore espalhada mínima

Seja  $G = (V, E)$  um grafo conectado não orientado com função peso  $w : E \rightarrow \mathbb{R}$ , e suponha que  $|E| \geq |V|$  e que todos os pesos de arestas são distintos.

Uma segunda melhor árvore espalhada mínima é definida como a seguir. Seja  $\mathcal{T}$  o conjunto de todas as árvores de  $G$ , e seja  $T'$  uma árvore espalhada mínima de  $G$ . Então, uma **segunda melhor árvore espalhada mínima** é uma árvore espalhada  $T$  tal que  $w(T) = \min_{T' \in \mathcal{T} - \{T'\}} \{w(T')\}$ .

- Mostre que a árvore espalhada mínima é única, mas que a segunda melhor árvore espalhada mínima não precisa ser única.

- b.** Seja  $T$  uma árvore espalhada mínima de  $G$ . Prove que existem arestas  $(u, v) \in T$  e  $(x, y) \notin T$  tais que  $T - \{(u, v)\} \cup \{(x, y)\}$  é uma segunda melhor árvore espalhada mínima de  $G$ .
- c.** Seja  $T$  uma árvore espalhada de  $G$  e, para dois vértices quaisquer  $u, v \in V$ , seja  $\max[u, v]$  uma aresta de peso máximo no caminho único entre  $u$  e  $v$  em  $T$ . Descreva um algoritmo de tempo  $O(V^2)$  que, dado  $T$ , calcule  $\max[u, v]$  para todo  $u, v \in V$ .
- d.** Forneça um algoritmo eficiente para calcular a segunda melhor árvore espalhada mínima de  $G$ .

### 23-2 Árvore espalhada mínima em grafos esparsos

Para um grafo conectado muito esparso  $G = (V, E)$ , podemos melhorar ainda mais o tempo de execução  $O(E + V \lg V)$  do algoritmo de Prim com heaps de Fibonacci, fazendo o “pré-processamento” de  $G$  para diminuir o número de vértices antes da execução do algoritmo de Prim. Em particular, escolhemos para cada vértice  $u$  a aresta de peso mínimo  $(u, v)$  incidente em  $u$ , e inserimos  $(u, v)$  na árvore espalhada mínima em construção. Depois, condensamos todas as arestas escolhidas (veja a Seção B.4). Em lugar de condensar essas arestas uma de cada vez, primeiro identificamos conjuntos de vértices que estão unidos no mesmo novo vértice. Então, criamos o grafo que teria resultado da condensação dessas arestas uma de cada vez, mas fazemos isso “renomeando” arestas de acordo com os conjuntos em que suas extremidades foram colocadas. Várias arestas do grafo original podem ser renomeadas da mesma forma que outras. Em tal caso, só resulta uma aresta, e seu peso é o mínimo entre os pesos das arestas originais correspondentes.

Inicialmente, definimos a árvore espalhada mínima  $T$  que está sendo construída como vazia e, para cada aresta  $(u, v) \in E$ , definimos  $\text{orig}[u, v] = (u, v)$  e  $c[u, v] = w(u, v)$ . Usamos o atributo  $\text{orig}$  para fazer referência à aresta do grafo inicial que está associada a uma aresta no grafo condensado. O atributo  $c$  contém o peso de uma aresta e, à medida que as arestas são condensadas, ele é atualizado de acordo com o esquema anterior para escolha de pesos de arestas. O procedimento MST-REDUCE recebe as entradas  $G$ ,  $\text{orig}$ ,  $c$  e  $T$ , e retorna um grafo condensado  $G'$  e atributos  $\text{orig}'$  e  $c'$  para o grafo  $G'$ . O procedimento também acumula arestas de  $G$  na árvore espalhada mínima  $T$ .

```

MST-REDUCE( $\text{orig}, G, c, T$ )
1  for cada  $v \in V[G]$ 
2    do  $\text{marca}[v] \leftarrow \text{FALSE}$ 
3    MAKE-SET( $v$ )
4  for cada  $u \in V[G]$ 
5    do if  $\text{marca}[u] = \text{FALSE}$ 
6      then escolher  $v \in \text{Adj}[u]$  tal que  $c[u, v]$  seja minimizado
7        UNION( $u, v$ )
8         $T \leftarrow T \cup \{\text{orig}[u, v]\}$ 
9         $\text{marca}[u] \leftarrow \text{marca}[v] \leftarrow \text{TRUE}$ 
10  $V[G'] \leftarrow \{\text{FIND-SET}(v) : v \in V[G]\}$ 
11  $E[G'] \leftarrow 0$ 
12 for cada  $(x, y) \in E[G]$ 
13   do  $u \leftarrow \text{FIND-SET}(x)$ 
14      $v \leftarrow \text{FIND-SET}(y)$ 
15     if  $(u, v) \notin E[G']$ 
16       then  $E[G'] \leftarrow E[G'] \cup \{(u, v)\}$ 
17          $\text{orig}'[u, v] \leftarrow \text{orig}[x, y]$ 
18          $c'[u, v] \leftarrow c[x, y]$ 
19       else if  $c[x, y] < c'[u, v]$ 
20         then  $\text{orig}'[u, v] \leftarrow \text{orig}[x, y]$ 
21            $c'[u, v] \leftarrow c[x, y]$ 
22 construir listas de adjacências  $\text{Adj}$  para  $G'$ 
23 return  $G', \text{orig}', c'$  e  $T$ 

```

- a.** Seja  $T$  o conjunto de arestas retornadas por MST-REDUCE, e seja  $A$  uma árvore espalhada mínima do grafo  $G'$  formada pela chamada MST-PRIM( $G', c', r$ ), onde  $r$  é qualquer vértice em  $V[G']$ . Prove que  $T \cup \{orig'[x, y] : (x, y) \in A\}$  é uma árvore espalhada mínima de  $G$ .
- b.** Demonstre que  $|V[G']| \leq |V|/2$ .
- c.** Mostre como implementar MST-REDUCE de forma que ele seja executado no tempo  $O(E)$ . (*Sugestão:* Utilize estruturas de dados simples.)
- d.** Suponha que executamos  $k$  fases de MST-REDUCE, usando as saídas  $G'$ ,  $orig'$  e  $c'$  produzidas por uma fase como as entradas  $G$ ,  $orig$  e  $c$  para a fase seguinte e acumulando arestas em  $T$ . Demonstre que o tempo de execução global das  $k$  fases é  $O(kE)$ .
- e.** Suponha que, depois de executar  $k$  fases de MST-REDUCE, como na parte (d), executamos o algoritmo de Prim chamando MST-PRIM( $G', c', r$ ), onde  $G'$  e  $c'$  são retornados pela última fase e  $r$  é qualquer vértice em  $V[G']$ . Mostre como escolher  $k$  de tal forma que o tempo de execução global seja  $O(E \lg \lg V)$ . Demonstre que sua escolha de  $k$  minimiza o tempo de execução assintótico global.
- f.** Para quais valores de  $|E|$  (em termos de  $|V|$ ) o algoritmo de Prim com pré-processamento é superior assintoticamente ao algoritmo de Prim sem pré-processamento?

### 23-3 Árvore espalhada de gargalo

Uma **árvore espalhada de gargalo**  $T$  de um grafo não orientado  $G$  é uma árvore espalhada de  $G$  cujo maior peso de aresta é mínimo sobre todas as árvores espalhadas de  $G$ . Dizemos que o valor da árvore espalhada de gargalo é o peso da aresta de peso máximo em  $T$ .

- a.** Demonstre que uma árvore espalhada mínima é uma árvore espalhada de gargalo.

A parte (a) mostra que encontrar uma árvore espalhada de gargalo não é mais difícil que encontrar uma árvore espalhada mínima. Nas partes restantes, mostraremos que é possível encontrá-la em tempo linear.

- b.** Forneça um algoritmo de tempo linear que, dado um grafo  $G$  e um inteiro  $b$ , determine se o valor da árvore espalhada de gargalo é no máximo  $b$ .
- c.** Use seu algoritmo para a parte (b) como uma sub-rotina em um algoritmo de tempo linear para o problema da árvore espalhada de gargalo. (*Sugestão:* Talvez você queira usar uma sub-rotina que condense conjuntos de arestas, como no procedimento MST-REDUCE descrito no Problema 23-2.)

### 23-4 Algoritmos alternativos de árvores de amplitude mínima

Neste problema, apresentamos o pseudocódigo para três algoritmos diferentes. Cada um toma um grafo como entrada e retorna um conjunto de arestas  $T$ . Para cada algoritmo, você deve provar que  $T$  é uma árvore espalhada mínima ou provar que  $T$  não é uma árvore espalhada mínima. Descreva também a implementação mais eficiente de cada algoritmo, quer ele calcule ou não uma árvore espalhada mínima.

- a.** MAYBE-MST-A( $G, w$ )
  - 1 ordenar as arestas em ordem não crescente de pesos de arestas  $w$
  - 2  $T \leftarrow E$
  - 3 **for** cada aresta  $e$ , tomada em ordem não crescente de peso
  - 4     **do if**  $T - \{e\}$  é um grafo conectado
  - 5         **then**  $T \leftarrow T - e$
  - 6 **return**  $T$
- b.** MAYBE-MST-B( $G, w$ )
  - 1  $T \leftarrow 0$
  - 2 **for** cada aresta  $e$ , tomada em ordem arbitrária

```

3   do if  $T \cup \{e\}$  não tem nenhum ciclo
4       then  $T \leftarrow T \cup e$ 
5   return  $T$ 

c. MAYBE-MST-C( $G, w$ )
1  $T \leftarrow \emptyset$ 
2 for cada aresta  $e$ , tomada em ordem arbitrária
3   do  $T \leftarrow T \cup \{e\}$ 
4       if  $T$  tem um ciclo  $c$ 
5           then seja  $e'$  a aresta de peso máximo em  $c$ 
6            $T \leftarrow T - \{e'\}$ 
7 return  $T$ 

```

## Notas do capítulo

Tarjan [292] examina o problema da árvore espalhada mínima e fornece excelente material avançado. Um histórico do problema da árvore espalhada mínima foi escrito por Graham e Hell [131].

Tarjan atribui o primeiro algoritmo de árvore espalhada mínima a um documento de 1926 produzido por O. Boruvka. O algoritmo de Boruvka consiste na execução de  $O(\lg V)$  iterações do procedimento MST-REDUCE descrito no Problema 23-2. O algoritmo de Kruskal foi apresentado por Kruskal [195] em 1956. O algoritmo comumente conhecido como algoritmo de Prim foi de fato desenvolvido por Prim [250], mas também foi criado antes por V. Jarník, em 1930.

A razão pela qual os algoritmos gulosos são eficientes na localização de árvores espalhadas mínima é que o conjunto de florestas de um grafo forma um matróide gráfico. (Consulte a Seção 16.4.)

Quando  $|E| = \Omega(V \lg V)$ , o algoritmo de Prim implementado com heaps de Fibonacci é executado no tempo  $O(E)$ . No caso de grafos mais esparsos, usando uma combinação das idéias de algoritmo do Prim, do algoritmo do Kruskal e do algoritmo de Boruvka, juntamente com estruturas de dados avançadas, Fredman e Tarjan [98] fornecem um algoritmo que é executado no tempo  $O(E \lg^* V)$ . Gabow, Galil, Spencer e Tarjan [102] melhoraram esse algoritmo para ser executado no tempo  $O(E \lg \lg^* V)$ . Chazelle [53] fornece um algoritmo que é executado no tempo  $O(E \hat{\alpha}(E, V))$ , onde  $\hat{\alpha}(E, V)$  é a função inversa da função de Ackermann. (Veja nas notas do capítulo correspondentes ao Capítulo 21 uma breve discussão da função de Ackermann e de sua inversa.) Diferente dos algoritmos de árvore espalhada mínima anteriores, o algoritmo de Chazelle não segue o método guloso.

Um problema relacionado é a *verificação de árvores espalhadas, na qual temos um grafo  $G = (V, E)$  e uma árvore  $T \subseteq E$  e desejamos determinar se  $T$  é uma árvore espalhada mínima de  $G$* . King [177] fornece um algoritmo de tempo linear para verificação de árvores espalhadas, fundamentado no trabalho anterior de Komlós [188] e de Dixon, Rauch e Tarjan [77].

Os algoritmos anteriores são todos determinísticos e se enquadram no modelo baseado em comparação descrito no Capítulo 8. Karger, Klein e Tarjan [169] fornecem um algoritmo de árvore espalhada mínima aleatório que é executado no tempo esperado  $O(V + E)$ . Esse algoritmo emprega recursão de maneira semelhante ao algoritmo de seleção de tempo linear da Seção 9.3: uma chamada recursiva sobre um problema auxiliar identifica um subconjunto das arestas  $E'$  que não podem estar em qualquer árvore espalhada mínima. Outra chamada recursiva em  $E - E'$  encontra então a árvore espalhada mínima. O algoritmo também usa idéias do algoritmo de Boruvka e do algoritmo de King para verificação de árvores espalhadas.

Fredman e Willard [100] mostraram como encontrar uma árvore espalhada mínima no tempo  $O(V + E)$  usando um algoritmo determinístico que não é baseado em comparação. Seu algoritmo supõe que os dados são inteiros de  $b$  bits e que a memória do computador consiste em palavras endereçáveis de  $b$  bits.

---

## Capítulo 24

# Caminhos mais curtos de única origem

Um motorista deseja encontrar a rota mais curta possível do Rio de Janeiro a São Paulo. Dado um mapa rodoviário do Brasil no qual a distância entre cada par de interseções adjacentes esteja marcada, como podemos determinar essa rota mais curta?

Um modo possível é enumerar todas as rotas do Rio de Janeiro a São Paulo, somar as distâncias em cada rota e selecionar a mais curta. Porém, é fácil ver que até mesmo se deixarmos de lado as rotas que contêm ciclos, haverá milhões de possibilidades, a maioria das quais simplesmente não valerá a pena considerar. Por exemplo, uma rota do Rio de Janeiro a Brasília e daí a São Paulo é sem dúvida uma escolha ruim, porque Brasília está alguns milhares de quilômetros fora do caminho.

Neste capítulo e no Capítulo 25, mostraremos como resolver de forma eficiente tais problemas. Em um **problema de caminhos mais curtos**, temos um grafo orientado ponderado  $G = (V, E)$ , com função peso  $w : E \rightarrow \mathbb{R}$  mapeando arestas para pesos de valores reais. O **peso** do caminho  $p = \langle v_0, v_1, \dots, v_k \rangle$  é o somatório dos pesos de suas arestas constituintes:

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i).$$

Definimos o **peso do caminho mais curto** desde  $u$  até  $v$  por

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow[p]{} v\} & \text{se existe um caminho de } u \text{ até } v, \\ \infty & \text{em caso contrário.} \end{cases}$$

Um **caminho mais curto** desde o vértice  $u$  até o vértice  $v$  é então definido como qualquer caminho  $p$  com peso  $w(p) = \delta(u, v)$ .

No exemplo da rota desde o Rio de Janeiro até São Paulo, podemos modelar o mapa rodoviário como um grafo: os vértices representam interseções, as arestas representam segmentos de estradas entre interseções e pesos de arestas representam distâncias rodoviárias. Nossa meta é encontrar um caminho mais curto desde uma dada interseção no Rio de Janeiro (digamos, a esquina da rua México com a rua Santa Luzia) até uma dada interseção em São Paulo (digamos, avenida Paulista com rua Augusta).

Os pesos de arestas podem ser interpretados como medidas, em vez de distâncias. Eles são usados com freqüência para representar tempo, custo, penalidades, perdas ou qualquer outra quantidade que se acumule linearmente ao longo de um caminho e que alguém deseje minimizar.

O algoritmo de busca em largura examinado na Seção 22.2 é um algoritmo de caminhos mais curtos que funciona sobre grafos não ponderados, isto é, grafos nos quais cada aresta pode ser considerada como tendo um peso unitário. Pelo fato de muitos conceitos da busca em largura surgirem no estudo de caminhos mais curtos em grafos ponderados, aconselhamos o leitor a revisar a Seção 22.2 antes de continuar.

## Variantes

Neste capítulo, focalizaremos o **problema de caminhos mais curtos de única origem**: dado um grafo  $G = (V, E)$ , queremos encontrar um caminho mais curto desde um determinado vértice de **origem**  $s \in V$  até todo vértice  $v \in V$ . Muitos outros problemas podem ser resolvidos pelo algoritmo correspondente ao problema de única origem, inclusive as variantes a seguir.

**Problema de caminhos mais curtos de destino único:** Encontrar um caminho mais curto até um determinado vértice de **destino**  $t$  a partir de cada vértice  $v$ . Invertendo o sentido de cada aresta no grafo, podemos reduzir esse problema a um problema de única origem.

**Problema do caminho mais curto de par único:** Encontrar um caminho mais curto desde  $u$  até  $v$  para determinados vértices  $u$  e  $v$ . Se resolvermos o problema de única origem com o vértice de origem  $u$ , também resolvemos esse problema. Além disso, não é conhecido nenhum algoritmo para esse problema que seja executado assintoticamente mais rápido que os melhores algoritmos de única origem no pior caso.

**Problema de caminhos mais curtos de todos os pares:** Encontrar um caminho mais curto desde  $u$  até  $v$  para todo par de vértices  $u$  e  $v$ . Embora esse problema possa ser resolvido executando-se um algoritmo de única origem uma vez a partir de cada vértice, em geral ele pode ser resolvido com maior rapidez. Além disso, sua estrutura tem interesse por si só. O Capítulo 25 estuda em detalhes o problema de todos os pares.

## Subestrutura ótima de um caminho mais curto

Em geral, os algoritmos de caminhos mais curtos se baseiam na propriedade de que um caminho mais curto entre dois vértices contém outros caminhos mais curtos em seu interior. (O algoritmo de fluxo máximo de Edmonds-Karp no Capítulo 26 também se baseia nessa propriedade.) Essa propriedade da subestrutura ótima é uma característica da aplicabilidade da programação dinâmica (Capítulo 15) e do método guloso (Capítulo 16). O algoritmo de Dijkstra, que veremos na Seção 24.3, é um algoritmo guloso, e o algoritmo de Floyd-Warshall, que encontra caminhos mais curtos entre todos os pares de vértices (consulte o Capítulo 25), é um algoritmo de programação dinâmica. O lema a seguir enuncia com maior exatidão a propriedade de subestrutura ótima de caminhos mais curtos.

### Lema 24.1 (Subcaminhos de caminhos mais curtos são caminhos mais curtos)

Dado um grafo orientado ponderado  $G = (V, E)$  com função peso  $w : E \rightarrow \mathbb{R}$ , seja  $p = \langle v_1, v_2, \dots, v_k \rangle$  um caminho mais curto do vértice  $v_1$  até o vértice  $v_k$  e, para quaisquer  $i$  e  $j$  tais que  $1 \leq i \leq j \leq k$ , seja  $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$  o subcaminho  $p$  desde o vértice  $v_i$  até o vértice  $v_j$ . Então,  $p_{ij}$  é um caminho mais curto de  $v_i$  até  $v_j$ .

**Prova** Se fizermos a decomposição do caminho  $p$  em  $v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$ , então teremos  $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$ . Agora, suponha que existisse um caminho  $p'_{ij}$  de  $v_i$  até  $v_j$  com peso  $w(p'_{ij}) < w(p_{ij})$ . Então,  $v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$  é um caminho de  $v_1$  até  $v_k$  cujo peso  $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$  é menor que  $w(p)$ , o que contradiz a hipótese de que  $p$  é um caminho mais curto de  $v_1$  até  $v_k$ . ■

## Arestas de peso negativo

Em algumas instâncias do problema de caminhos mais curtos de única origem, pode haver arestas cujos pesos são negativos. Se o grafo  $G = (V, E)$  não contém nenhum ciclo de peso negativo acessível a partir da origem  $s$ , então para todo  $v \in V$ , o peso do caminho mais curto  $\delta(s, v)$  permanece bem definido, mesmo tendo um valor negativo. Contudo, se existe um ciclo de peso negativo acessível a partir de  $s$ , os pesos de caminhos mais curtos não são bem definidos. Nenhum caminho desde  $s$  até um vértice no ciclo pode ser um caminho mais curto – sempre é possível encontrar um caminho de peso menor que segue o caminho “mais curto” proposto e depois atravessa o ciclo de peso negativo. Se existe um ciclo de peso negativo em algum caminho desde  $s$  até  $v$ , definimos  $\delta(s, v) = -\infty$ .

A Figura 24.1 ilustra o efeito de pesos negativos sobre pesos de caminhos mais curtos. Pelo fato de existir apenas um caminho de  $s$  até  $a$  (o caminho  $\langle s, a \rangle$ ),  $\delta(s, a) = w(s, a) = 3$ . De modo semelhante, existe apenas um caminho de  $s$  até  $b$ , e assim  $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$ . Existem infinitamente muitos caminhos desde  $s$  até  $c$ :  $\langle s, c \rangle, \langle s, c, d, c \rangle, \langle s, c, d, c, d, c \rangle$ , e assim por diante. Pelo fato do ciclo  $\langle c, d, c \rangle$ , ter peso  $6 + (-3) = 3 > 0$ , o caminho mais curto desde  $s$  até  $c$  é  $\langle s, c \rangle$ , com peso  $\delta(s, c) = 5$ . De modo semelhante, o caminho mais curto desde  $s$  até  $d$  é  $\langle s, c, d \rangle$ , com peso  $\delta(s, d) = w(s, c) + w(c, d) = 11$ . De forma análoga, existem infinitamente muitos caminhos desde  $s$  até  $e$ :  $\langle s, e \rangle, \langle s, e, f, e \rangle, \langle s, e, f, e, f, e \rangle$ , e assim por diante. Porém, tendo em vista que o ciclo  $\langle e, f, e \rangle$ , tem peso  $3 + (-6) = -3 < 0$ , não existe nenhum caminho mais curto desde  $s$  até  $e$ . Percorrendo o ciclo de peso negativo  $\langle e, f, e \rangle$  arbitrariamente muitas vezes, podemos encontrar caminhos desde  $s$  até  $e$  com pesos negativos arbitrariamente grandes, e assim  $\delta(s, e) = -\infty$ . De modo semelhante,  $\delta(s, f) = -\infty$ . Como  $g$  é acessível a partir de  $f$ , também podemos encontrar caminhos com pesos negativos arbitrariamente grandes desde  $s$  até  $g$  e  $\delta(s, g) = -\infty$ . Os vértices  $b, i$  e  $j$  também formam um ciclo de peso negativo. Contudo, eles não são acessíveis a partir de  $s$ , e assim  $\delta(s, b) = \delta(s, i) = \delta(s, j) = \infty$ .

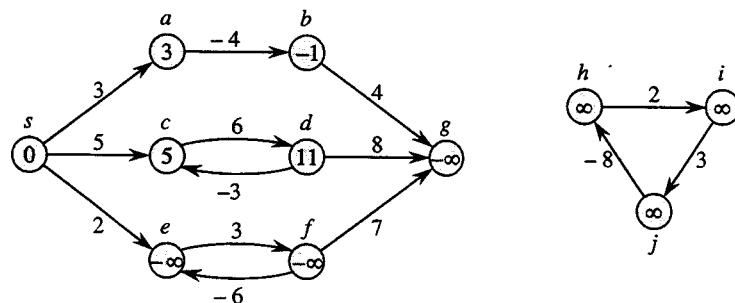


FIGURA 24.1 Pesos de arestas negativos em um grafo orientado. Mostramos dentro de cada vértice o peso de seu caminho mais curto a partir da origem  $s$ . Como os vértices  $e$  e  $f$  formam um ciclo de peso negativo acessível a partir de  $s$ , eles têm pesos de caminhos mais curtos iguais a  $-\infty$ . Como o vértice  $g$  é acessível a partir de um vértice cujo peso de caminho mais curto é  $-\infty$ , ele também tem um peso de caminho mais curto igual a  $-\infty$ . Vértices como  $b, i$  e  $j$  não são acessíveis a partir de  $s$ , e assim os pesos de seus caminhos mais curtos são iguais a  $\infty$ , embora eles residam em um ciclo de peso negativo.

Alguns algoritmos de caminhos mais curtos, como o algoritmo de Dijkstra, supõem que todos os pesos de arestas no grafo de entrada são não negativos, como no exemplo do mapa rodoviário. Outros, como o algoritmo de Bellman-Ford, permitem arestas de peso negativo no grafo de entrada e produzem uma resposta correta enquanto nenhum ciclo de peso negativo é acessível a partir da origem. Em geral, se houver um ciclo de peso negativo desse tipo, o algoritmo poderá detectar e relatar sua existência.

## Ciclos

Um caminho mais curto pode conter um ciclo? Como acabamos de ver, ele não pode conter um ciclo de peso negativo. Nem pode conter um ciclo de peso positivo, pois a remoção do ciclo do caminho produz um caminho com os mesmos vértices de origem e destino e um peso de caminho mais baixo. Isto é, se  $p = \langle v_0, v_1, \dots, v_k \rangle$  é um caminho e  $c = \langle v_i, v_{i+1}, \dots, v_j \rangle$  é um ciclo de peso positivo nesse caminho (de forma que  $v_i = v_j$  e  $w(c) > 0$ ), então o caminho  $p' = \langle v_0, v_1, \dots, v_i, v_{j+1}, v_{j+2}, \dots, v_k \rangle$  tem peso  $w(p') = w(p) - w(c) < w(p)$ , e então  $p$  não pode ser um caminho mais curto de  $v_0$  até  $v_k$ .

Isso deixa apenas ciclos de peso 0. Podemos remover um ciclo de peso 0 de qualquer caminho para produzir outro caminho cujo peso é o mesmo. Desse modo, se existe um caminho mais curto de um vértice de origem  $s$  até um vértice de destino  $v$  que contém um ciclo de peso 0, então existe outro caminho mais curto de  $s$  para  $v$  sem esse ciclo. Tendo em vista que um caminho mais curto tem ciclos de peso 0, podemos remover repetidamente esses ciclos do caminho até que tenhamos um caminho mais curto livre de ciclos. Então, sem perda de generalidade, podemos supor que, quando estamos encontrando caminhos mais curtos, eles não têm nenhum ciclo. Considerando que qualquer caminho acíclico em um grafo  $G = (V, E)$  contém no máximo  $|V|$  vértices distintos, ele também contém no máximo  $|V| - 1$  arestas. Desse modo, podemos restringir nossa atenção a caminhos mais curtos de no máximo  $|V| - 1$  arestas.

## Representação de caminhos mais curtos

Com freqüência, desejamos calcular não apenas pesos de caminhos mais curtos, mas também os vértices nos caminhos mais curtos. A representação que usamos para caminhos mais curtos é semelhante à que utilizamos para árvores primeiro na extensão na Seção 22.2. Dado um grafo  $G = (V, E)$ , mantemos para cada vértice  $v \in V$  um **predecessor**  $\pi[v]$  que é outro vértice ou NIL. Os algoritmos de caminhos mais curtos deste capítulo definem os atributos  $\pi$  de tal forma que a cadeia de predecessores com origem em um vértice  $v$  seja percorrida ao contrário ao longo de um caminho mais curto desde  $s$  até  $v$ . Desse modo, dado um vértice  $v$  para o qual  $\pi[v] \neq \text{NIL}$  o procedimento PRINT-PATH( $G, s, v$ ) da Seção 22.2 pode ser usado para imprimir um caminho mais curto desde  $s$  até  $v$ .

Entretanto, durante a execução de um algoritmo de caminhos mais curtos, os  $\pi$  valores não precisam indicar caminhos mais curtos. Como na busca em largura, estaremos interessados no **subgrafo predecessor**  $G_\pi = (V_\pi, E_\pi)$  induzido pelos valores de  $\pi$ . Novamente, definimos o conjunto de vértices  $V_\pi$  como o conjunto de vértices de  $G$  com predecessores não NIL, mais a origem  $s$ :

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\} .$$

O conjunto de arestas orientadas  $E_\pi$  é o conjunto de arestas induzidas pelos  $\pi$  valores correspondentes aos vértices em  $V_\pi$ :

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\} .$$

Provaremos que os  $\pi$  valores produzidos pelos algoritmos neste capítulo têm a propriedade de que, no final,  $G_\pi$  é uma “árvore de caminhos mais curtos” – informalmente, uma árvore enraizada que contém um caminho mais curto desde a origem  $s$  até todo vértice acessível a partir de  $s$ . Uma árvore de caminhos mais curtos é semelhante à árvore primeiro na extensão da Seção 22.2, mas contém caminhos mais curtos desde a origem definidos em termos de pesos de arestas, em vez de números de arestas. Para sermos precisos, seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbb{R}$ , e suponha que  $G$  não contém nenhum ciclo de peso negativo acessível a partir do vértice de origem  $s \in V$ , e assim esses caminhos mais curtos são bem definidos.

Uma **árvore de caminhos mais curtos** com raiz em  $s$  é um subgrafo orientado  $G' = (V', E')$ , onde  $V' \subseteq V$  e  $E' \subseteq E$ , tal que

1.  $V'$  é o conjunto de vértices acessíveis a partir de  $s$  em  $G$ ,
2.  $G'$  forma uma árvore enraizada com raiz  $s$ , e
3. para todo  $v \in V'$ , o único caminho simples desde  $s$  até  $v$  em  $G'$  é um caminho mais curto desde  $s$  até  $v$  em  $G$ .

Caminhos mais curtos não são necessariamente únicos, e nem árvores de caminhos mais curtos. Por exemplo, a Figura 24.2 mostra um grafo orientado ponderado e duas árvores de caminhos mais curtos com a mesma raiz.

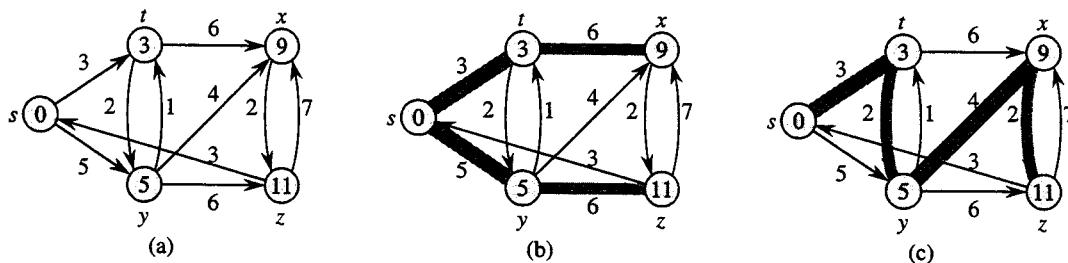


FIGURA 24.2 (a) Um grafo orientado ponderado com pesos de caminhos mais curtos desde a origem  $s$ . (b) As arestas sombreadas formam uma árvore de caminhos mais curtos com raiz na origem  $s$ . (c) Outra árvore de caminhos mais curtos com a mesma raiz

## Relaxamento

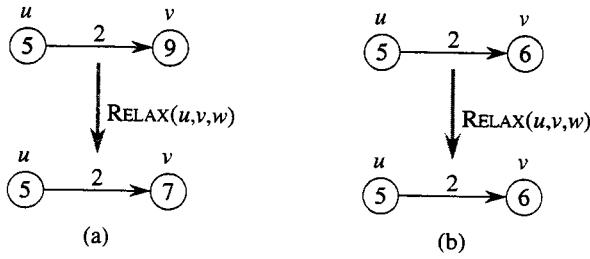
Os algoritmos neste capítulo usam a técnica de **relaxamento**. Para cada vértice  $v \in V$ , mantemos um atributo  $d[v]$ , que é um limite superior sobre o peso de um caminho mais curto desde a origem  $s$  até  $v$ . Chamamos  $d[v]$  uma **estimativa de caminho mais curto**. Inicializamos as estimativas de caminhos mais curtos e os predecessores por meio do procedimento de tempo  $\Theta(V)$  apresentado a seguir.

```
INITIALIZE-SINGLE-SOURCE( $G, s$ )
1 for cada vértice  $v \in V[G]$ 
2   do  $d[v] \leftarrow \infty$ 
3    $\pi[v] \leftarrow \text{NIL}$ 
4  $d[s] \leftarrow 0$ 
```

Após a inicialização,  $\pi[v]$  para todo  $v \in V$ ,  $d[s] = 0$  e  $d[v] = \infty$  para  $v \in V - \{s\}$ .

O processo de **relaxar**<sup>1</sup> uma aresta  $(u, v)$  consiste em testar se podemos melhorar o caminho mais curto para  $v$  encontrado até agora pela passagem através de  $u$  e, nesse caso, atualizar  $d[v]$  e  $\pi[v]$ . Uma etapa de relaxamento pode diminuir o valor da estimativa de caminho mais curto  $d[v]$  e atualizar o campo predecessor de  $v$ ,  $\pi[v]$ . O código a seguir executa uma etapa de relaxamento sobre a aresta  $(u, v)$ .

<sup>1</sup>Pode parecer estranho que o termo “relaxamento” seja usado para uma operação que restringe um limite superior. O uso do termo é histórico. A saída de uma etapa de relaxamento pode ser vista como um relaxamento da restrição  $d[v] \leq d[u] + w(u, v)$  que, pela desigualdade de triângulos (Lema 24.10), deve ser satisfeita se  $d[u] = \delta(s, u)$  e  $d[v] = \delta(s, v)$ . Ou seja, se  $d[v] \leq d[u] + w(u, v)$ , não existe nenhuma “pressão” para satisfazer essa restrição, e assim a restrição é “relaxada”.



**FIGURA 24.3** Relaxamento de uma aresta  $(u, v)$  com peso  $w(u, v) = 2$ . A estimativa de caminhos mais curtos de cada vértice é mostrada dentro do vértice. (a) Como  $d[v] > d[u] + w(u, v)$  antes do relaxamento, o valor de  $d[v]$  diminui. (b) Aqui,  $d[v] \leq d[u] + w(u, v)$  antes da etapa de relaxamento, e assim  $d[v]$  não é alterada pelo relaxamento

**RELAX( $u, v, w$ )**

- 1 **if**  $d[v] > d[u] + w(u, v)$
- 2   **then**  $d[v] \leftarrow d[u] + w(u, v)$
- 3     $\pi[v] \leftarrow u$

A Figura 24.3 mostra dois exemplos de relaxamento de uma aresta, um no qual a estimativa de caminhos mais curtos diminui, e um exemplo no qual nenhuma estimativa se altera.

Cada algoritmo neste capítulo chama INITIALIZE-SINGLE-SOURCE, e depois relaxa repetidamente as arestas. Além disso, o relaxamento é o único meio pelo qual se alteram estimativas de caminhos mais curtos e predecessores. Os algoritmos deste capítulo diferem na quantidade de vezes que eles relaxam cada aresta e na ordem em que as arestas são relaxadas. No algoritmo de Dijkstra e no algoritmo de caminhos mais curtos para grafos acíclicos orientados, cada aresta é relaxada exatamente uma vez. No algoritmo de Bellman-Ford, cada aresta é relaxada muitas vezes.

## Propriedades de caminhos mais curtos e relaxamento

Para demonstrar a correção dos algoritmos deste capítulo, recorreremos a várias propriedades de caminhos mais curtos e relaxamento. Enunciamos essas propriedades aqui, e a Seção 24.5 apresentará sua prova formal. Para sua referência, cada propriedade declarada aqui inclui o número do lema ou corolário apropriado da Seção 24.5. As cinco últimas dessas propriedades, que se referem a estimativas de caminhos mais curtos ou ao subgrafo predecessor, supõem implicitamente que o grafo é inicializado com uma chamada a INITIALIZE-SINGLE-SOURCE( $G, s$ ) e que o único modo de modificar as estimativas de caminhos mais curtos e o subgrafo predecessor é empregar alguma seqüência de etapas de relaxamento.

### Desigualdade de triângulos (Lema 24.10)

Para qualquer aresta  $(u, v) \in E$ , temos  $\delta(s, v) \leq \delta(s, u) + w(u, v)$ .

### Propriedade do limite superior (Lema 24.11)

Sempre temos  $d[v] \geq \delta(s, v)$  para todos os vértices  $v \in V$  e, uma vez que  $d[v]$  alcança o valor  $\delta(s, v)$ , ele nunca muda.

### Propriedade de nenhum caminho (Corolário 24.12)

Se não existe nenhum caminho de  $s$  para  $v$ , então sempre temos  $d[v] = \delta(s, v) = \infty$ .

### Propriedade de convergência (Lema 24.14)

Se  $s \sim u \rightarrow v$  é um caminho mais curto em  $G$  para algum  $u, v \in V$ , e se  $d[u] = \delta(s, u)$  em qualquer instante antes de se relaxar a aresta  $(u, v)$ , então  $d[v] = \delta(s, v)$  em todos os momentos posteriores.

### Propriedade de relaxamento de caminho (Lema 24.15)

Se  $p = \langle v_0, v_1, \dots, v_k \rangle$  é um caminho mais curto de  $s = v_0$  a  $v_k$ , e as arestas de  $p$  são relaxadas na ordem  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ , então  $d[v_k] = \delta(s, v_k)$ . Essa propriedade é válida, não importando quaisquer outras etapas de relaxamento que ocorram, ainda que elas estejam entremeadas com relaxamentos das arestas de  $p$ .

### Propriedade de subgrafo predecessor (Lema 24.17)

Uma vez que  $d[v] = \delta(s, v)$  para todo  $v \in V$ , o subgrafo predecessor é uma árvore de caminhos mais curtos com raiz em  $s$ .

## Esboço do capítulo

A Seção 24.1 apresenta o algoritmo de Bellman-Ford, que resolve o problema de caminhos mais curtos de única origem no caso geral em que arestas podem ter peso negativo. O algoritmo de Bellman-Ford é notável em sua simplicidade, e ele tem a vantagem adicional de detectar se um ciclo de peso negativo é acessível a partir da origem. A Seção 24.2 fornece um algoritmo de tempo linear para calcular caminhos mais curtos a partir de uma única origem em um grafo acíclico orientado. A Seção 24.3 cobre o algoritmo de Dijkstra, que tem um tempo de execução menor que o do algoritmo de Bellman-Ford, mas exige que os pesos de arestas sejam não negativos. A Seção 24.4 mostra como o algoritmo de Bellman-Ford pode ser usado para resolver um caso especial de “programação linear”. Finalmente, a Seção 24.5 prova as propriedades de caminhos mais curtos e relaxamento que enunciámos na seção anterior.

Precisamos de algumas convenções para efetuar cálculos aritméticos com valores infinitos. Consideraremos que, para qualquer número real  $a \neq -\infty$ , temos  $a + \infty = \infty + a = \infty$ . Além disso, para tornar nossas demonstrações (ou provas) válidas na presença de ciclos de peso negativo, partiremos da premissa de que, para qualquer número real  $a \neq \infty$ , temos  $a + (-\infty) = (-\infty) + a = -\infty$ .

Todos os algoritmos neste capítulo supõem que o grafo orientado  $G$  está armazenado na representação de lista de adjacências. Além disso, está armazenado com cada aresta seu peso, de forma que, à medida que percorremos cada lista de adjacências, podemos determinar os pesos de arestas no tempo  $O(1)$  por aresta.

## 24.1 O algoritmo de Bellman-Ford

O **algoritmo de Bellman-Ford** resolve o problema de caminhos mais curtos de única origem no caso mais geral, no qual os pesos das arestas podem ser negativos. Dado um grafo orientado ponderado  $G = (V, E)$  com origem  $s$  e função peso  $w : E \rightarrow \mathbb{R}$ , o algoritmo de Bellman-Ford retorna um valor booleano indicando se existe ou não um ciclo de peso negativo acessível a partir da origem. Se existe tal ciclo, o algoritmo indica que não existe nenhuma solução. Se não existe tal ciclo, o algoritmo produz os caminhos mais curtos e seus pesos.

O algoritmo usa o relaxamento, diminuindo progressivamente uma estimativa  $d[v]$  no peso de um caminho mais curto da origem  $s$  até cada vértice  $v \in V$ , até alcançar o peso real de caminho mais curto  $\delta(s, v)$ . O algoritmo retorna TRUE se e somente se o grafo não contém nenhum ciclo de peso negativo que seja acessível a partir da origem.

```
BELLMAN-FORD( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2 for  $i \leftarrow 1$  to  $|V[G]| - 1$ 
3   do for cada aresta  $(u, v) \in E[G]$ 
4     do RELAX( $u, v, w$ )
5 for cada aresta  $(u, v) \in E[G]$ 
6   do if  $d[v] > d[u] + w(u, v)$ 
7     then return FALSE
8 return TRUE
```

A Figura 24.4 mostra a execução do algoritmo de Bellman-Ford sobre um grafo com 5 vértices. Depois de inicializar os valores de  $d$  e  $\pi$  de todos os vértices na linha 1, o algoritmo faz  $|V| - 1$  passagens sobre as arestas do grafo. Cada passagem é uma iteração do loop `for` das linhas 2 a 4 e consiste em relaxar cada aresta do grafo uma vez. A Figura 24.4(b)-(e) mostra o estado do algoritmo após cada uma das quatro passagens sobre as arestas. Depois de fazer  $|V| - 1$  passagens, as linhas 5 a 8 procuram por um ciclo de peso negativo e retornam o valor booleano apropriado. (Veremos um pouco mais adiante por que essa verificação funciona.)

O algoritmo de Bellman-Ford é executado no tempo  $O(VE)$ , pois a inicialização na linha 1 demora o tempo  $\Theta(V)$ , cada uma das  $|V| - 1$  passagens sobre as arestas nas linhas 2 a 4 demora o tempo  $\Theta(E)$ , e o loop `for` das linhas 5 a 7 demora o tempo  $O(E)$ .

Para provar a correção do algoritmo de Bellman-Ford, começamos mostrando que, se não existir nenhum ciclo de peso negativo, o algoritmo calculará pesos de caminhos mais curtos corretos para todos os vértices acessíveis a partir da origem.

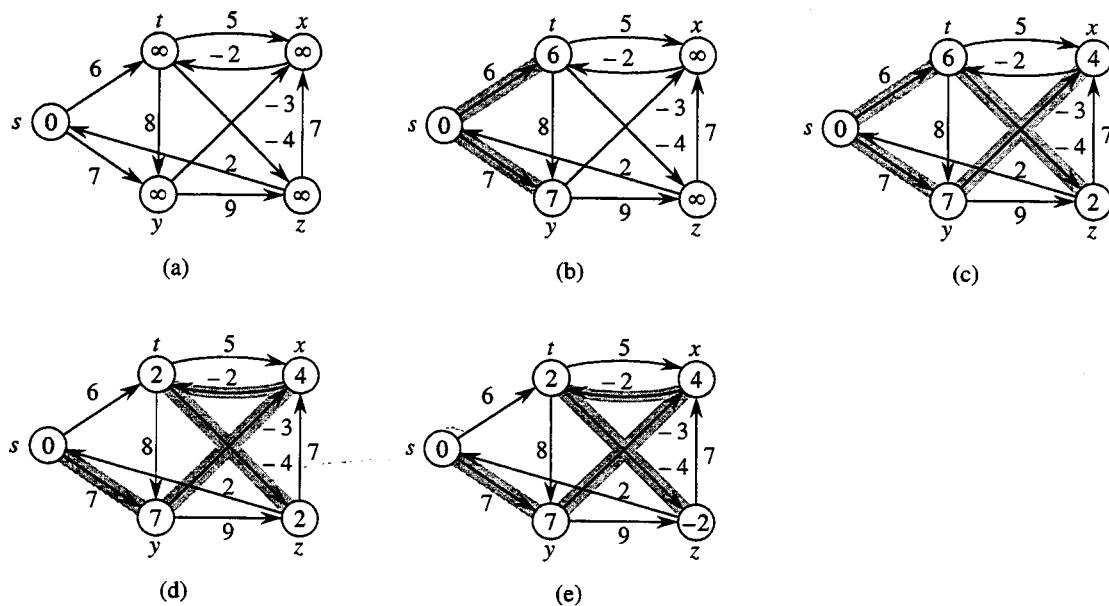


FIGURA 24.4 A execução do algoritmo de Bellman-Ford. A origem é o vértice  $s$ . Os valores de  $d$  são mostrados dentro dos vértices, e as arestas sombreadas indicam os valores de predecessores; se a aresta  $(u, v)$  estiver sombreada, então  $\pi[v] = u$ . Nesse exemplo específico, cada passagem relaxa as arestas na ordem  $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$ . (a) A situação imediatamente antes da primeira passagem sobre as arestas. (b)-(e) A situação após cada passagem sucessiva sobre as arestas. Os valores de  $d$  e  $\pi$  na parte (e) são os valores finais. O algoritmo de Bellman-Ford retorna TRUE nesse exemplo

### Lema 24.2

Seja  $G = (V, E)$  um grafo orientado ponderado com origem  $s$  e função peso  $w : E \rightarrow \mathbf{R}$ , e suponha que  $G$  não contenha nenhum ciclo de peso negativo que seja acessível a partir de  $s$ . Então, após as  $|V| - 1$  iterações do loop `for` das linhas 2 a 4 de BELLMAN-FORD, temos  $d[v] = \delta(s, v)$  para todos os vértices  $v$  acessíveis a partir de  $s$ .

**Prova** Provamos o lema apelando para a propriedade de relaxamento de caminho. Considere qualquer vértice  $v$  que seja acessível a partir de  $s$ , e seja  $p = \langle v_0, v_1, \dots, v_k \rangle$ , onde  $v_0 = s$  e  $v_k = v$ , qualquer acíclico caminho mais curto de  $s$  para  $v$ . O caminho  $p$  tem no máximo  $|V| - 1$  arestas, e assim  $k \leq |V| - 1$ . Cada uma das  $|V| - 1$  iterações do loop `for` das linhas 2 a 4 relaxa todas as  $|E|$  arestas. Entre as arestas relaxadas na  $i$ -ésima iteração, para  $i = 1, 2, \dots, k$ , encontra-se  $(v_{i-1}, v_i)$ . Então, pela propriedade de relaxamento de caminho,  $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$ . ■

### Corolário 24.3

Seja  $G = (V, E)$  um grafo orientado ponderado com vértice de origem  $s$  e função peso  $w : E \rightarrow \mathbb{R}$ . Então, para cada vértice  $v \in V$ , existe um caminho de  $s$  para  $v$  se e somente se BELLMAN-FORD termina com  $d[v] < \infty$  quando é executado sobre  $G$ .

**Prova** A prova fica para o Exercício 24.1-2. ■

### Teorema 24.4 (Correção do algoritmo de Bellman-Ford)

Seja o algoritmo de BELLMAN-FORD executado sobre um grafo orientado ponderado  $G = (V, E)$  com origem  $s$  e função peso  $w : E \rightarrow \mathbb{R}$ . Se  $G$  não contém nenhum ciclo de peso negativo que seja acessível a partir de  $s$ , então o algoritmo retorna TRUE, temos  $d[v] = \delta(s, v)$  para todos os vértices  $v \in V$ , e o subgrafo predecessor  $G_\pi$  é uma árvore de caminhos mais curtos com raiz em  $s$ . Se  $G$  contém um ciclo de peso negativo acessível a partir de  $s$ , então o algoritmo retorna FALSE.

**Prova** Suponha que o grafo  $G$  não contenha nenhum ciclo de peso negativo que seja acessível a partir da origem  $s$ . Primeiro, provamos a afirmação de que, no término,  $d[v] = \delta(s, v)$  para todos os vértices  $v \in V$ . Se o vértice  $v$  é acessível a partir de  $s$ , então o Lema 24.2 prova essa afirmação. Se  $v$  não é acessível a partir de  $s$ , então a afirmação decorre da propriedade de nenhum caminho. Portanto, a afirmação está provada. A propriedade de subgrafo predecessor, juntamente com a afirmação, implica que  $G_\pi$  é uma árvore de caminhos mais curtos. Agora, usamos a afirmação para mostrar que BELLMAN-FORD retorna TRUE. No término, temos para todas as arestas  $(u, v) \in E$ ,

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \quad (\text{pela desigualdade de triângulos}) \\ &= d[u] + w(u, v), \end{aligned}$$

e assim nenhum dos testes na linha 6 faz BELLMAN-FORD retornar FALSE. Então, ele retorna TRUE.

Reciprocamente, suponha que o grafo  $G$  contenha um ciclo de peso negativo acessível a partir da origem  $s$ ; seja esse ciclo  $c = \langle v_0, v_1, \dots, v_k \rangle$ , onde  $v_0 = v_k$ . Então,

$$\sum_{i=1}^k w(v_{i-1}, v_i) < 0. \tag{24.1}$$

Suponha, para fins de contradição, que o algoritmo de Bellman-Ford retorne TRUE. Desse modo,  $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$  para  $i = 1, 2, \dots, k$ . A soma das desigualdades em torno do ciclo  $c$  nos dá

$$\begin{aligned} \sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i). \end{aligned}$$

Como  $v_0 = v_k$ , cada vértice em  $c$  aparece exatamente uma vez em cada um dos somatórios  $\sum_{i=1}^k d[v_i]$  e  $\sum_{i=1}^k d[v_{i-1}]$ , e assim

$$\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}].$$

Além disso, pelo Corolário 24.3,  $d[v_i]$  é finito para  $i = 1, 2, \dots, k$ . Portanto,

$$0 \leq \sum_{i=1}^k w(v_{i-1}, v_i),$$

o que contradiz a desigualdade (24.1). Concluímos que o algoritmo de Bellman-Ford retorna TRUE se o grafo  $G$  não contém nenhum ciclo de peso negativo acessível a partir da origem, e FALSE em caso contrário. ■

## Exercícios

### 24.1-1

Execute o algoritmo de Bellman-Ford no grafo orientado da Figura 24.4, usando o vértice  $z$  como origem. Em cada passagem, relaxe as arestas na mesma ordem da figura e mostre os valores de  $d$  e  $\pi$  após cada passagem. Agora, mude o peso da aresta  $(z, x)$  para 4 e execute o algoritmo novamente, usando  $s$  como origem.

### 24.1-2

Prove o Corolário 24.3.

### 24.1-3

Dado um grafo orientado ponderado  $G = (V, E)$  sem ciclos de peso negativo, seja  $m$  o máximo sobre todos os pares de vértices  $u, v \in V$  do número mínimo de arestas em um caminho mais curto de  $u$  até  $v$ . (Aqui, o caminho mais curto é por peso, não pelo número de arestas.) Sugira uma mudança simples no algoritmo de Bellman-Ford que permita encerrá-lo em  $m + 1$  passagens.

### 24.1-4

Modifique o algoritmo de Bellman-Ford de modo que ele defina  $d[v]$  como  $-\infty$  para todos os vértices  $v$  para os quais existe um ciclo de peso negativo em algum caminho a partir da origem até  $v$ .

### 24.1-5 \*

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbb{R}$ . Forneça um algoritmo de tempo  $O(VE)$  a fim de encontrar, para cada vértice  $v \in V$ , o valor  $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$ .

### 24.1-6 \*

Suponha que um grafo orientado ponderado  $G = (V, E)$  tenha um ciclo de peso negativo. Forneça um algoritmo eficiente para listar os vértices de um tal ciclo. Prove que seu algoritmo é correto.

## 24.2 Caminhos mais curtos de única origem em grafos acíclicos orientados

Relaxando as arestas de um gao (grafo acíclico orientado) ponderado  $G = (V, E)$  de acordo com uma ordenação topológica de seus vértices, podemos calcular caminhos mais curtos a partir de uma única origem no tempo  $\Theta(V + E)$ . Caminhos mais curtos são sempre bem definidos em um gao, pois, mesmo se existissem arestas de peso negativo, não poderia existir nenhum ciclo de peso negativo.

O algoritmo começa ordenando topologicamente o gao (ver Seção 22.4) para impor uma ordenação linear sobre os vértices. Se existe um caminho do vértice  $u$  até o vértice  $v$ , então  $u$  precede  $v$  na ordem topológica. Fazemos apenas uma passagem sobre os vértices na seqüência topologicamente ordenada. À medida que cada vértice é processado, todas as arestas que deixam o vértice são relaxadas.

DAG-SHORTEST-PATHS( $G, w, s$ )

- 1 ordenar topologicamente os vértices de  $G$
- 2 INITIALIZE-SINGLE-SOURCE( $G, s$ )
- 3 **for** cada vértice  $u$  tomado em seqüência topologicamente ordenada
- 4     **do for** cada vértice  $v \in \text{Adj}[u]$
- 5         **do** RELAX( $u, v, w$ )

Um exemplo da execução desse algoritmo é mostrado na Figura 24.5.

É fácil analisar o tempo de execução desse algoritmo. Como mostra a Seção 22.4, a ordem topológica da linha 1 pode ser executada no tempo  $\Theta(V + E)$ . A chamada de INITIALIZE-SINGLE-SOURCE na linha 2 demora o tempo  $\Theta(V)$ . Existe uma iteração por vértice no loop for das linhas 3 a 5. Para cada vértice, cada uma das arestas que deixam o vértice é examinada exatamente uma vez. Desse modo, existe um total de  $|E|$  iterações do loop for mais interno das linhas 4 e 5. (Usamos aqui uma análise agregada.) Como cada iteração do loop for mais interno demora o tempo de  $\Theta(1)$ , o tempo de execução total é  $\Theta(V + E)$ , que é linear no tamanho de uma representação de lista de adjacências do grafo.

O teorema a seguir mostra que o procedimento DAG-SHORTEST-PATHS calcula corretamente os caminhos mais curtos.

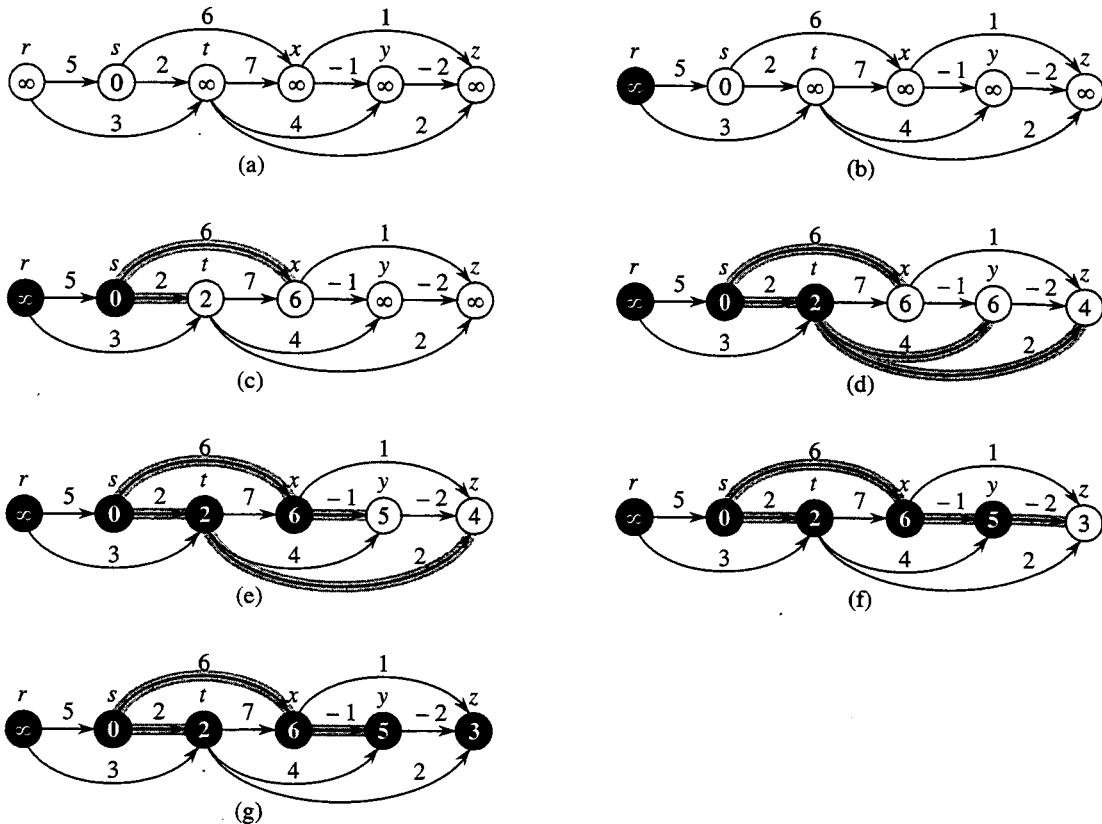


FIGURA 24.5 A execução do algoritmo para caminhos mais curtos em um grafo acíclico orientado. Os vértices são ordenados topologicamente da esquerda para a direita. O vértice de origem é  $s$ . Os valores de  $d$  são mostrados dentro dos vértices, e arestas sombreadas indicam os valores de  $\pi$ . (a) A situação antes da primeira iteração do loop for das linhas 3 a 5. (b)–(g) A situação após cada iteração do loop for das linhas 3 a 5. O vértice enegrecido recentemente em cada iteração foi usado como  $u$  nessa iteração. Os valores mostrados na parte (g) são os valores finais

### Teorema 24.5

Se um grafo orientado ponderado  $G = (V, E)$  tem vértice de origem  $s$  e nenhum ciclo, então no término do procedimento DAG-SHORTEST-PATHS,  $d[v] = \delta(s, v)$  para todos os vértices  $v \in V$ , e o subgrafo predecessor  $G_\pi$  é uma árvore de caminhos mais curtos.

**Prova** Primeiro, mostramos que  $d[v] = \delta(s, v)$  para todos os vértices  $v \in V$  no término. Se  $v$  não é acessível a partir de  $s$ , então  $d[v] = \delta(s, v) = \infty$  pela propriedade de nenhum caminho. Agora, suponha que  $v$  seja acessível a partir de  $s$ , de modo que exista um caminho mais curto  $p = \langle v_0, v_1, \dots, v_k \rangle$ , onde  $v_0 = s$  e  $v_k = v$ . Como processamos os vértices em seqüência ordenada topo-

logicamente, as arestas em  $p$  são relaxadas na ordem  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ . A propriedade de relaxamento de caminho implica que  $d[v_i] = \delta(s, v_i)$  no término para  $i = 0, 1, \dots, k$ . Finalmente, pela propriedade de subgrafo predecessor,  $G_\pi$  é uma árvore de caminhos mais curtos. ■

Uma aplicação interessante desse algoritmo surge na determinação de caminhos críticos na análise *diagramas PERT*<sup>2</sup>. As arestas representam serviços a serem executados, e os pesos de arestas representam os tempos necessários para execução de determinados serviços. Se a aresta  $(u, v)$  entra no vértice  $v$  e a aresta  $(v, x)$  sai de  $v$ , então o serviço  $(u, v)$  deve ser executado antes do serviço  $(v, x)$ . Um caminho através desse grafo representa uma seqüência de serviços que devem ser executados em uma determinada ordem. Um *caminho crítico* é um caminho *mais longo* pelo grafo, correspondendo ao tempo mais longo para execução de uma seqüência ordenada de serviços. O peso de um caminho crítico é um limite inferior sobre o tempo total para execução de todos os serviços. Podemos encontrar um caminho crítico de duas maneiras:

- tornando negativos os pesos de arestas e executando DAG-SHORTEST-PATHS, ou
- executando DAG-SHORTEST-PATHS, substituindo “ $\infty$ ” por “ $-\infty$ ” na linha 2 de INITIALIZE-SINGLE-SOURCE e “ $>$ ” por “ $<$ ” no procedimento RELAX.

## Exercícios

### 24.2-1

Execute DAG-SHORTEST-PATHS sobre o grafo orientado da Figura 24.5, usando o vértice  $r$  como origem.

### 24.2-2

Suponha que mudamos a linha 3 de DAG-PATHS para

3 **for** os primeiros  $|V| - 1$  vértices, tomados em seqüência ordenada topologicamente

Mostre que o procedimento continuaria sendo correto.

### 24.2-3

A formulação de diagramas PERT dada anteriormente é um tanto antinatural. Seria mais natural que os vértices representassem serviços e arestas, restrições de seqüenciamento; isto é, a aresta  $(u, v)$  indicaria que o serviço  $u$  deve ser executado antes do serviço  $v$ . Então, seriam atribuídos pesos a vértices, e não a arestas. Modifique o procedimento DAG-SHORTEST-PATHS de forma que ele encontre um caminho mais longo em um grafo acíclico orientado com vértices ponderados em tempo linear.

### 24.2-4

Forneça um algoritmo eficiente para contar o número total de caminhos em um grafo acíclico orientado. Analise seu algoritmo.

## 24.3 Algoritmo de Dijkstra

O algoritmo de Dijkstra resolve o problema de caminhos mais curtos de única origem em um grafo orientado ponderado  $G = (V, E)$  para o caso no qual todos os pesos de arestas são não negativos. Então, nesta seção, iremos supor que  $w(u, v) \geq 0$  para cada aresta  $(u, v) \in E$ . Como veremos, com uma boa implementação, o tempo de execução do algoritmo de Dijkstra é inferior ao do algoritmo de Bellman-Ford.

O algoritmo de Dijkstra mantém um conjunto  $S$  de vértices cujos pesos finais de caminhos mais curtos desde a origem  $s$  já foram determinados. O algoritmo seleciona repetidamente o vértice  $u \in V - S$  com a estimativa mínima de caminhos mais curtos, adiciona  $u$  a  $S$  e relaxa todas as arestas que saem de  $u$ . Na implementação a seguir, manteremos uma fila de prioridade mínima  $Q$  de vértices, tendo como chaves seus valores de  $d$ .

DIJKSTRA( $G, w, s$ )

```

1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$ 
4 while  $Q \neq \emptyset$ 
5   do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6      $S \leftarrow S \cup \{u\}$ 
7     for cada vértice  $v \in \text{Adj}[u]$ 
8       do RELAX( $u, v, w$ )

```

O algoritmo de Dijkstra relaxa arestas como mostra a Figura 24.6. A linha 1 executa a inicialização habitual dos valores de  $d$  e  $\pi$ , e a linha 2 inicializa o conjunto  $S$  como o conjunto vazio. O algoritmo mantém o invariante de que  $Q = V - S$  no início de cada iteração do loop **while** das linhas 4 a 8. A linha 3 inicializa a fila de prioridade mínima  $Q$  para conter todos os vértices em  $V$ ; tendo em vista que  $S = \emptyset$  nesse momento, o invariante é verdadeiro após a linha 3. Em cada passagem pelo loop **while** das linhas 4 a 8, um vértice  $u$  é extraído de  $Q = V - S$  e inserido no conjunto  $S$ , mantendo assim o invariante. (Na primeira passagem por esse loop,  $u = s$ .) Então, o vértice  $u$  tem a menor estimativa de caminhos mais curtos em comparação com qualquer vértice em  $V - S$ . Em seguida, as linhas 7 e 8 relaxam cada aresta  $(u, v)$  que saem de  $u$ , atualizando assim a estimativa  $d[v]$  e o predecessor  $\pi[v]$  se o caminho mais curto até  $v$  pode ser melhorado mediante a passagem por  $u$ . Observe que os vértices nunca são inseridos em  $Q$  após a linha 3, e que cada vértice é extraído de  $Q$  e inserido em  $S$  exatamente uma vez, de modo que o loop **while** das linhas 4 a 8 itera exatamente  $|V|$  vezes.

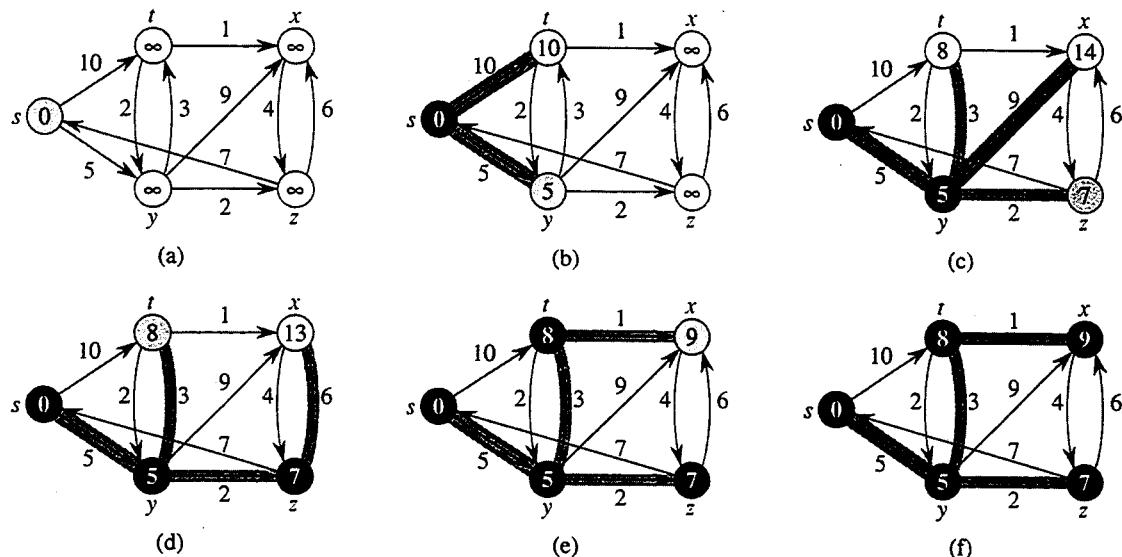


FIGURA 24.6 A execução do algoritmo de Dijkstra. A origem  $s$  é o vértice mais à esquerda. As estimativas de caminhos mais curtos são mostradas dentro dos vértices, e as arestas sombreadas indicam valores de predecessores. Vértices pretos estão no conjunto  $S$ , e vértices brancos estão na fila de prioridade mínima  $Q = V - S$ . (a) A situação imediatamente antes da primeira iteração do loop **while** das linhas 4 a 8. O vértice sombreado tem o valor de  $d$  mínimo e é escolhido como vértice  $u$  na linha 5. (b)-(f) A situação após cada iteração sucessiva do loop **while**. O vértice sombreado em cada parte é escolhido como vértice  $u$  na linha 5 da próxima iteração. Os valores de  $d$  e  $\pi$  mostrados na parte (f) são os valores finais

Pelo fato do algoritmo de Dijkstra sempre escolher o vértice “mais leve” ou “mais próximo” em  $V - S$  para adicionar ao conjunto  $S$ , dizemos que ele utiliza uma estratégia gulosa. As estratégias gulosas são apresentadas em detalhes no Capítulo 16, mas você não precisa ler aquele capítulo para entender o algoritmo de Dijkstra. As estratégias gulosas nem sempre produzem resultados ótimos em geral, mas, como mostram o teorema a seguir e seu corolário, o algoritmo de Dijkstra realmente calcula caminhos mais curtos. A chave é mostrar que cada vez que um vértice  $u$  é inserido no conjunto  $S$ , temos  $d[u] = \delta(s, u)$ .

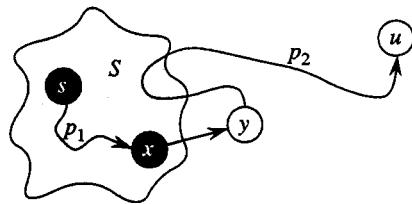


FIGURA 24.7 A prova do Teorema 24.6. O conjunto  $S$  é não vazio imediatamente antes do vértice  $u$  ser inserido nele. Um caminho mais curto  $p$  desde a origem  $s$  até o vértice  $u$  pode ser decomposto em  $s \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$ , onde  $y$  é o primeiro vértice sobre o caminho que não está em  $S$  e  $x \in S$  precede imediatamente  $y$ . Os vértices  $x$  e  $y$  são distintos, mas podemos ter  $s = x$  ou  $y = u$ . O caminho  $p_2$  pode reentrar ou não no conjunto  $S$

### **Teorema 24.6 (Correção do algoritmo de Dijkstra)**

Se executarmos o algoritmo de Dijkstra sobre um grafo orientado ponderado  $G = (V, E)$  com função peso não negativa  $w$  e origem  $s$ , ele termina com  $d[u] = \delta(s, u)$  para todos os vértices  $u \in V$ .

**Prova** Usamos o loop invariante a seguir:

No início de cada iteração do loop **while** das linhas 4 a 8,  $d[v] = \delta(s, v)$  para cada vértice  $v \in S$ .

Basta mostrar que, para cada vértice  $u \in V$ , temos  $d[u] = \delta(s, u)$  no momento em que  $u$  é adicionado ao conjunto  $S$ . Uma vez que mostramos que  $d[u] = \delta(s, u)$ , recorremos à propriedade do limite superior para mostrar que a igualdade é válida em todos os momentos daí em diante.

**Inicialização:** Inicialmente,  $S = \emptyset$ , e assim o invariante é verdadeiro de forma trivial.

**Manutenção:** Desejamos mostrar que, em cada iteração,  $d[u] = \delta(s, u)$  para o vértice adicionado ao conjunto  $S$ . Para fins de contradição, seja  $u$  o primeiro vértice para o qual  $d[u] \neq \delta(s, u)$  quando ele é adicionado ao conjunto  $S$ . Concentraremos nossa atenção na situação existente no início da iteração do loop **while** em que  $u$  é adicionado a  $S$  e derivamos a contradição de que  $d[u] = \delta(s, u)$  nesse momento, examinando um caminho mais curto de  $s$  até  $u$ . Deveremos ter  $u \neq s$  porque  $s$  é o primeiro vértice adicionado ao conjunto  $S$  e  $d[s] = \delta(s, s) = 0$  nesse momento. Pelo fato de  $u \neq s$ , também temos  $S \neq \emptyset$  logo antes de  $u$  ser adicionado a  $S$ . Deve haver algum caminho de  $s$  até  $u$  pois, do contrário,  $d[u] = \delta(s, u) = \infty$  pela propriedade de nenhum caminho, o que violaria nossa hipótese de que  $d[u] \neq \delta(s, u)$ . Como há pelo menos um caminho, existe um caminho mais curto  $p$  de  $s$  até  $u$ . Antes de se adicionar  $u$  a  $S$ , o caminho  $p$  conecta um vértice em  $S$ , isto é  $s$ , a um vértice em  $V - S$ , ou seja,  $u$ . Vamos considerar o primeiro vértice  $y$  ao longo de  $p$  tal que  $y \in V - S$ , e seja  $x \in S$  o predecessor de  $y$ . Portanto, como mostra a Figura 24.7, o caminho  $p$  pode ser decomposto como  $s \xrightarrow{p_1} x \rightarrow y \xrightarrow{p_2} u$ . (Um dos caminhos  $p_1$  ou  $p_2$  pode não ter nenhuma aresta.)

Afirmamos que  $d[y] = \delta(s, y)$  quando  $u$  é adicionado a  $S$ . Para provar essa afirmação, observe que  $x \in S$ . Assim, como  $u$  foi escolhido como o primeiro vértice para o qual  $d[u] \neq \delta(s, u)$  quando foi adicionado a  $S$ , tínhamos  $d[x] = \delta(s, x)$  quando  $x$  foi adicionado a  $S$ . A aresta  $(x, y)$  foi relaxada nesse momento, e assim a afirmação decorre da propriedade de convergência.

Podemos agora obter uma contradição para provar que  $d[u] = \delta(s, u)$ . Como  $y$  ocorre antes de  $u$  em um caminho mais curto de  $s$  para  $u$  e todos os pesos de arestas são não negativos (especialmente das arestas do caminho  $p_2$ ), temos  $\delta(s, y) \leq \delta(s, u)$  e, desse modo,

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq d[u] \quad (\text{pela propriedade do limite superior}) . \end{aligned} \tag{24.2}$$

Porém, como ambos os vértices  $u$  e  $y$  estavam em  $V - S$  quando  $u$  foi escolhido na linha 5, temos  $d[u] \leq d[y]$ . Desse modo, as duas desigualdades em (24.2) são de fato igualdades, dando

$$d[y] = \delta(s, y) = \delta(s, u) = d[u] .$$

Conseqüentemente,  $d[u] = \delta(s, u)$ , o que contradiz nossa escolha de  $u$ . Concluímos que  $d[u] = \delta(s, u)$  quando  $u$  é adicionado a  $S$ , e que essa igualdade é mantida em todos os momentos daí em diante.

**Término:** No término,  $Q = \emptyset$  e, juntamente com nosso invariante anterior de que  $Q = V - S$ , isso implica que  $S = V$ . Desse modo,  $d[u] = \delta(s, u)$  para todos os vértices  $u \in V$ . ■

### Corolário 24.7

Se executarmos o algoritmo de Dijkstra sobre um grafo orientado ponderado  $G = (V, E)$  com função peso não negativa  $w$  e origem  $s$ , então, no término, o subgrafo predecessor  $G_\pi$  será uma árvore de caminhos mais curtos com raiz em  $s$ .

**Prova** Imediata a partir do Teorema 24.6 e da propriedade de subgrafo predecessor. ■

## Análise

Qual é a rapidez do algoritmo de Dijkstra? Ele mantém a fila de prioridade mínima  $Q$  chamando três operações de filas de prioridades: INSERT (implícita na linha 3), EXTRACT-MIN (linha 5) e DECREASE-KEY (implícita em RELAX, que é chamado na linha 8). INSERT é invocado uma vez por vértice, como é EXTRACT-MIN. Pelo fato de cada vértice  $v \in V$  ser adicionado ao conjunto  $S$  exatamente uma vez, cada aresta na lista de adjacências  $Adj[v]$  é examinada no loop for das linhas 7 e 8 exatamente uma vez durante o curso do algoritmo. Tendo em vista que o número total de arestas em todas as listas de adjacências é  $|E|$ , existe um total de  $|E|$  iterações desse loop for, e há portanto um total de no máximo  $|E|$  operações DECREASE-KEY. (Observe uma vez mais que estamos usando análise agregada.)

O tempo de execução do algoritmo de Dijkstra depende de como a fila de prioridade mínima é implementada. Considere primeiro o caso no qual mantemos a fila de prioridade mínima, tirando proveito do fato de que os vértices são numerados de 1 a  $|V|$ . Simplesmente armazenamos  $d[v]$  na  $v$ -ésima entrada de um arranjo. Cada operação INSERT e DECREASE-KEY demora o tempo  $O(1)$ , e cada operação EXTRACT-MIN demora o tempo  $O(V)$  (pois temos de pesquisar pelo arranjo inteiro), dando um tempo total  $O(V^2 + E) = O(V^2)$ .

Se o grafo é suficientemente esparsão – em particular,  $E = o(V^2/\lg V)$  – é prático implementar a fila de prioridade mínima  $Q$  com um heap mínimo binário. (Conforme discutimos na Seção 6.5, um detalhe de implementação importante é que os vértices e os elementos do heap correspondentes devem manter descritores um para o outro.) Cada operação EXTRACT-MIN demora então o tempo  $O(\lg V)$ . Como antes, existem  $|V|$  dessas operações. O tempo para construir o heap mínimo binário é  $O(V)$ . Cada operação DECREASE-KEY demora o tempo  $O(\lg V)$ , e ainda há no máximo  $|E|$  de tais operações. Então, o tempo de execução total é  $O((V + E) \lg V)$ , que é  $O(E \lg V)$  se todos os vértices são acessíveis a partir da origem. Esse tempo de execução é uma melhoria sobre o tempo  $O(V^2)$  de implementação direta se  $E = o(V^2/\lg V)$ .

Podemos de fato alcançar um tempo de execução igual a  $O(V \lg V + E)$  implementando a fila de prioridade mínima  $Q$  com um heap de Fibonacci (ver Capítulo 20). O custo amortizado de cada uma das  $|V|$  operações EXTRACT-MIN é  $O(\lg V)$ , e cada chamadas de DECREASE-KEY, das quais existe no máximo  $|E|$ , demora apenas o tempo amortizado  $O(1)$ . Historicamente, o desenvolvimento de heaps de Fibonacci foi motivado pela observação de que, no algoritmo de Dijkstra, existem tipicamente muito mais chamadas DECREASE-KEY que chamadas EXTRACT-MIN; assim, qualquer método de redução do tempo amortizado de cada operação DECREASE-KEY para  $O(\lg V)$  sem aumentar o tempo amortizado de EXTRACT-MIN produziria uma implementação assintoticamente mais rápida do que aquela que utiliza heaps binários.

O algoritmo de Dijkstra exibe alguma semelhança, tanto em relação à busca em largura (ver Seção 22.2) quanto em relação ao algoritmo de Prim para calcular árvores espalhadas mínimas (ver Seção 23.2). Ele é semelhante à busca em largura no fato de que o conjunto  $S$  corresponde ao conjunto de vértices pretos em uma busca em largura; exatamente como os vértices em  $S$  têm seus pesos finais de caminhos mais curtos, os vértices pretos em uma busca em largura têm suas distâncias corretas primeiro na extensão. O algoritmo de Dijkstra é semelhante ao algoritmo de Prim no fato de que ambos os algoritmos usam uma fila de prioridade mínima para encontrar o vértice “mais leve” fora de um conjunto dado (o conjunto  $S$  no algoritmo de Dijkstra, e a árvore que está sendo aumentada no algoritmo de Prim), inserem esse vértice no conjunto e ajustam os pesos dos vértices restantes fora do conjunto de acordo com ele.

## Exercícios

### 24.3-1

Execute o algoritmo de Dijkstra sobre o grafo orientado da Figura 24.2, primeiro usando o vértice  $s$  como origem, e depois usando o vértice  $y$  como origem. No estilo da Figura 24.6, mostre os valores de  $d$  e  $\pi$  e os vértices no conjunto  $S$  após cada iteração do loop **while**.

### 24.3-2

Forneça um exemplo simples de um grafo orientado com arestas de peso negativo para o qual o algoritmo de Dijkstra produza respostas incorretas. Por que a prova do Teorema 24.6 não é válida quando são permitidas arestas de peso negativo?

### 24.3-3

Suponha que mudamos a linha 4 do algoritmo de Dijkstra para o seguinte:

4 **while**  $|Q| > 1$

Essa mudança faz o loop **while** ser executado  $|V| - 1$  vezes em lugar de  $|V|$  vezes. Esse algoritmo proposto é correto?

### 24.3-4

Temos um grafo orientado  $G = (V, E)$  no qual cada aresta  $(u, v) \in E$  tem um valor associado  $r(u, v)$ , o qual é um número real no intervalo  $0 \leq r(u, v) \leq 1$  que representa a confiabilidade de um canal de comunicação do vértice  $u$  até o vértice  $v$ . Interpretamos  $r(u, v)$  como a probabilidade de que o canal de  $u$  até  $v$  não venha a falhar, e supomos que essas probabilidades são independentes. Forneça um algoritmo eficiente para encontrar o caminho mais confiável entre dois vértices dados.

### 24.3-5

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \{1, 2, \dots, W\}$  para algum inteiro positivo  $W$ , e suponha que não existam dois vértices com os mesmos pesos de caminhos mais curtos a partir do vértice de origem  $s$ . Agora, suponha que definimos um grafo orientado não ponderado  $G' = (V \cup V', E')$  substituindo cada aresta  $(u, v) \in E$  por  $w(u, v)$  arestas de peso unitário em série. Quantos vértices  $G'$  tem? Suponha agora que executamos uma busca em

largura sobre  $G'$ . Mostre que a ordem em que os vértices em  $V$  são coloridos de preto na busca em largura de  $G'$  é igual à ordem em que os vértices de  $V$  são extraídos da fila de prioridades na linha 5 de DIJKSTRA quando executado sobre  $G$ .

#### 24.3-6

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \{0, 1, \dots, W\}$  para algum inteiro não negativo  $W$ . Modifique o algoritmo de Dijkstra para calcular os caminhos mais curtos a partir de um vértice de origem  $s$  dado no tempo  $O(WV + E)$ .

#### 24.3-7

Modifique seu algoritmo do Exercício 24.3-6 para ser executado no tempo  $O((V + E) \lg W)$ . (Sugestão: Quantos valores de caminhos mais curtos distintos podem existir em  $V - S$  em qualquer instante?)

#### 24.3-8

Suponha que temos um grafo orientado ponderado  $G = (V, E)$  em que as arestas que saem do vértice de origem  $s$  pode ter pesos negativos, todos os outros pesos de arestas são não negativos e não existe nenhum ciclo de peso negativo. Demonstre que o algoritmo de Dijkstra encontra corretamente caminhos mais curtos a partir de  $s$  nesse grafo.

## 24.4 Restrições de diferenças e caminhos mais curtos

O Capítulo 29 estuda o problema geral de programação linear, no qual desejamos otimizar uma função linear de acordo com um conjunto de desigualdades lineares. Nesta seção, investigaremos um caso especial de programação linear que pode ser reduzido a encontrar caminhos mais curtos a partir de uma única origem. O problema de caminhos mais curtos de única origem resultante pode então ser resolvido com o uso do algoritmo de Bellman-Ford, resolvendo assim também o problema de programação linear.

### Programação linear

No **problema de programação linear** geral, temos uma matriz  $A$   $m \times n$ , um vetor de  $m$  elementos  $b$ , e um vetor de  $n$  elementos  $c$ . Desejamos encontrar um vetor  $x$  de  $n$  elementos que maximize a **função objetivo**  $\sum_{i=1}^n c_i x_i$  de acordo com as  $m$  restrições dadas por  $Ax \leq b$ .

Embora o algoritmo simplex focalizado no Capítulo 29 nem sempre funcione em tempo polinomial no tamanho de sua entrada, há outros algoritmos de programação linear que são executados em tempo polinomial. Existem várias razões pelas quais é importante compreender a configuração de problemas de programação linear. Primeiro, saber que um dado problema pode ser moldado como um problema de programação linear de tamanho polinomial significa imediatamente que existe um algoritmo de tempo polinomial para o problema. Em segundo lugar, há muitos casos especiais de programação linear para os quais existem algoritmos mais rápidos. Por exemplo, como mostramos nesta seção, o problema dos caminhos mais curtos de única origem é um caso especial de programação linear. Outros problemas que podem ser moldados como programação linear incluem o problema de caminhos mais curtos de um único par (Exercício 24.4-4) e o problema de fluxo máximo (Exercício 26.1-8).

Algumas vezes, realmente não nos importamos com a função objetivo; apenas desejamos encontrar qualquer **solução viável**, ou seja, qualquer vetor  $x$  que satisfaça a  $Ax \leq b$ , ou então determinar que não existe nenhuma solução viável. Concentraremos nossa atenção em um tal **problema de viabilidade**.

### Sistemas de restrições de diferenças

Em um **sistema de restrições de diferenças**, cada linha da matriz de programação linear  $A$  contém um valor 1 e um valor  $-1$ , e todas as outras entradas de  $A$  são iguais a 0. Desse modo, as

restrições dadas por  $Ax \leq b$  são um conjunto de  $m$  **restrições de diferenças** envolvendo  $n$  incógnitas, no qual cada restrição é uma desigualdade linear simples da forma

$$x_j = x_i \leq b_k,$$

onde  $1 \leq i, j \leq n$  e  $1 \leq k \leq m$ .

Por exemplo, considere o problema de encontrar o vetor de 5 elementos  $x = (x_i)$  que satisfaz a

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}.$$

Esse problema é equivalente a encontrar as incógnitas  $x_i$ , para  $i = 1, 2, \dots, 5$ , tais que as 8 restrições de diferenças a seguir sejam satisfeitas:

$$x_1 - x_2 \leq 0, \quad (24.3)$$

$$x_1 - x_5 \leq -1, \quad (24.4)$$

$$x_2 - x_5 \leq 1, \quad (24.5)$$

$$x_3 - x_1 \leq 5, \quad (24.6)$$

$$x_4 - x_1 \leq 4, \quad (24.7)$$

$$x_4 - x_3 \leq -1, \quad (24.8)$$

$$x_5 - x_3 \leq -3, \quad (24.9)$$

$$x_5 - x_4 \leq -3. \quad (24.10)$$

Uma solução para esse problema é  $x = (-5, -3, 0, -1, -4)$ , como podemos verificar diretamente examinando cada desigualdade. De fato, existe mais de uma solução para esse problema. Outra é  $x' = (0, 2, 5, 4, 1)$ . Essas duas soluções estão relacionadas: cada componente de  $x'$  é 5 unidades maior que o componente correspondente de  $x$ . Esse fato não é mera coincidência.

### Lema 24.8

Seja  $x = (x_1, x_2, \dots, x_n)$  uma solução para um sistema  $Ax \leq b$  de restrições de diferenças, e seja  $d$  qualquer constante. Então,  $x + d = (x_1 + d, x_2 + d, \dots, x_n + d)$  também é uma solução para  $Ax \leq b$ .

**Prova** Para cada  $x_i$  e  $x_j$ , temos  $(x_j + d) - (x_i + d) = x_j - x_i$ . Desse modo, se  $x$  satisfaz a  $Ax \leq b$ ,  $x + d$  também satisfaz a essa desigualdade. ■

Os sistemas de restrições de diferenças ocorrem em muitas aplicações diferentes. Por exemplo, as incógnitas  $x_i$  podem ser horários nos quais eventos devem ocorrer. Cada restrição pode ser vista como a declaração de que deve haver pelo menos um certo espaço de tempo, ou no má-

ximo um certo período entre dois eventos. Talvez os eventos sejam serviços a serem executados durante a montagem de um produto. Se aplicarmos no momento  $x_1$  um adesivo que demora duas horas para se fixar e temos de esperar até ele se fixar para instalar uma peça no momento  $x_2$ , então temos a restrição de que  $x_2 \geq x_1 + 2$  ou, de modo equivalente, que  $x_1 - x_2 \leq -2$ . Como outra alternativa, poderíamos exigir que a peça fosse instalada depois da aplicação do adesivo, mas não depois do momento em que o adesivo é fixado até a metade. Nesse caso, obtemos o par de restrições  $x_2 \geq x_1$  e  $x_2 \leq x_1 + 1$  ou, de modo equivalente,  $x_1 - x_2 \leq 0$  e  $x_2 - x_1 \leq 1$ .

## Grafos de restrições

É vantajoso interpretar sistemas de restrições de diferenças a partir de um ponto de vista de teoria de grafos. A idéia é que, em um sistema  $Ax \leq b$  de restrições de diferenças, a matriz de programação linear  $A$   $n \times m$  pode ser vista como uma matriz de incidência (ver Exercício 22.1-7) para um grafo com  $n$  vértices e  $m$  arestas. Cada vértice  $v_i$  no grafo, para  $i = 1, 2, \dots, n$ , corresponde a uma das  $n$  incógnitas  $x_i$ . Cada aresta orientada no grafo corresponde a uma das  $m$  desigualdades que envolvem duas incógnitas.

De modo mais formal, dado um sistema  $Ax \leq b$  de restrições de diferenças, o *grafo de restrição* correspondente é um grafo orientado ponderado  $G = (V, E)$ , onde

$$V = \{v_0, v_1, \dots, v_n\}$$

e

$$\begin{aligned} E = & \{(v_i, v_j) : x_j - x_i \leq b_k \text{ é uma restrição}\} \\ & \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \dots, (v_0, v_n)\}. \end{aligned}$$

O vértice adicional  $v_0$  é incorporado, como veremos em breve, para garantir que todo outro vértice será acessível a partir dele. Desse modo, o conjunto de vértices  $V$  consiste em um vértice  $v_i$  para cada incógnita  $x_i$ , mais um vértice adicional  $v_0$ . O conjunto de arestas  $E$  contém uma aresta para cada restrição de diferença, mais uma aresta  $(v_0, v_i)$  para cada incógnita  $x_i$ . Se  $x_j - x_i \leq b_k$  é uma restrição de diferença, então o peso da aresta  $(v_i, v_j)$  é  $w(v_i, v_j) = b_k$ . O peso de cada aresta que sai de  $v_0$  é 0. A Figura 24.8 mostra o grafo de restrição para o sistema (24.3)-(24.10) de restrições de diferenças.

O teorema a seguir mostra que uma solução para um sistema de restrições de diferenças pode ser obtida encontrando-se os pesos de caminhos mais curtos no grafo de restrição correspondente.

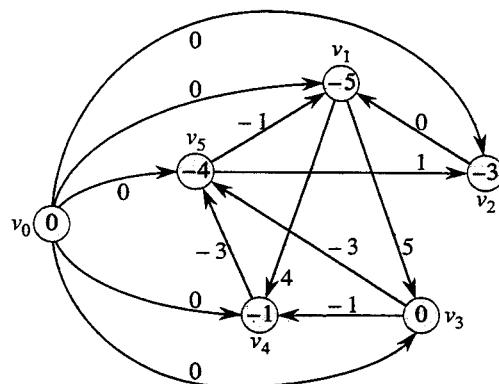


FIGURA 24.8 O grafo de restrição correspondente ao sistema (24.3)-(24.10) de restrições de diferenças. O valor de  $\delta(v_0, v_1)$  é mostrado em cada vértice  $v_i$ . Uma solução viável para o sistema é  $\mathbf{x} = (-5, -3, 0, -1, -4)$

### **Teorema 24.9**

Dado um sistema  $Ax \leq b$  de restrições de diferenças, seja  $G = (V, E)$  o grafo de restrição correspondente. Se  $G$  não contém nenhum ciclo de peso negativo, então

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \dots, \delta(v_0, v_n)) \quad (24.11)$$

é uma solução viável para o sistema. Se  $G$  contém um ciclo de peso negativo, então não existe nenhuma solução viável para o sistema.

**Prova** Primeiro mostramos que, se o grafo de restrição não contém nenhum ciclo de peso negativo, então a equação (24.11) fornece uma solução viável. Considere qualquer aresta  $(v_i, v_j) \in E$ . Pela desigualdade de triângulos,  $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$  ou, de modo equivalente,  $\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$ . Assim, fazer  $x_i = \delta(v_0, v_i)$  e  $x_j = \delta(v_0, v_j)$  satisfaz à restrição de diferença  $x_j - x_i \leq w(v_i, v_j)$  que corresponde à aresta  $(v_i, v_j)$ .

Agora mostramos que, se o grafo de restrição contém um ciclo de peso negativo, então o sistema de restrições de diferenças não tem nenhuma solução viável. Sem perda de generalidade, seja o ciclo de peso negativo  $c = \langle v_1, v_2, \dots, v_k \rangle$ , onde  $v_1 = v_k$ . (O vértice  $v_0$  não pode estar no ciclo  $c$ , porque ele não tem nenhuma aresta de entrada.) O ciclo  $c$  corresponde às seguintes restrições de diferenças:

$$\begin{aligned} x_2 - x_1 &\leq w(v_1, v_2), \\ x_3 - x_2 &\leq w(v_2, v_3), \\ &\vdots \\ x_k - x_{k-1} &\leq w(v_{k-1}, v_k), \\ x_1 - x_k &\leq w(v_k, v_1). \end{aligned}$$

Suponha que exista uma solução para  $x$  que satisfaça a cada uma dessas  $k$  desigualdades. Essa solução também deve satisfazer à desigualdade que resulta quando somarmos as  $k$  desigualdades. Se somarmos os lados esquerdos, cada incógnita  $x_i$  será adicionada uma vez e subtraída uma vez, de forma que o lado esquerdo da soma seja 0. A soma dos valores do lado direito é  $w(c)$ , e assim obtemos  $0 \leq w(c)$ . Porém, tendo em vista que  $c$  é um ciclo de peso negativo,  $w(c) < 0$ , e obtemos a contradição de que  $0 \leq w(c) < 0$ . ■

### **Resolução de sistemas de restrições de diferenças**

O Teorema 24.9 nos informa que podemos usar o algoritmo de Bellman-Ford para resolver um sistema de restrições de diferenças. Pelo fato de existirem arestas do vértice de origem  $v_0$  até todos os outros vértices no grafo de restrição, qualquer ciclo de peso negativo no grafo de restrição é acessível a partir de  $v_0$ . Se o algoritmo de Bellman-Ford retorna TRUE, então os pesos de caminhos mais curtos oferecem uma solução viável para o sistema. Por exemplo, na Figura 24.8, os pesos de caminhos mais curtos fornecem a solução viável  $x = (-5, -3, 0, -1, -4)$  e, pelo Lema 24.8,  $x = (d-5, d-3, d, d-1, d-4)$  também é uma solução viável para qualquer constante  $d$ . Se o algoritmo de Bellman-Ford retorna FALSE, não existe nenhuma solução viável para o sistema de restrições de diferenças.

Um sistema de restrições de diferenças com  $m$  restrições sobre  $n$  incógnitas produz um grafo com  $n+1$  vértices e  $n+m$  arestas. Desse modo, usando o algoritmo de Bellman-Ford, podemos resolver o sistema no tempo  $O((n+1)(n+m)) = O(n^2 + nm)$ . O Exercício 24.4-5 lhe pede para modificar o algoritmo de forma a executá-lo no tempo  $O(nm)$ , ainda que  $m$  seja muito menor que  $n$ .

## Exercícios

### 24.4-1

Encontre uma solução viável ou descubra que não existe nenhuma solução viável para o sistema de restrições de diferenças a seguir:

$$\begin{aligned}x_1 - x_2 &\leq 1, \\x_1 - x_4 &\leq -4, \\x_2 - x_3 &\leq 2, \\x_2 - x_5 &\leq 7, \\x_2 - x_6 &\leq 5, \\x_3 - x_6 &\leq 10, \\x_4 - x_2 &\leq 2, \\x_5 - x_1 &\leq -1, \\x_5 - x_4 &\leq 3, \\x_6 - x_3 &\leq -8.\end{aligned}$$

### 24.4-2

Encontre uma solução viável ou descubra que não existe nenhuma solução viável para o sistema de restrições de diferenças a seguir:

$$\begin{aligned}x_1 - x_2 &\leq 4, \\x_1 - x_5 &\leq 5, \\x_2 - x_4 &\leq -6, \\x_3 - x_2 &\leq 1, \\x_4 - x_1 &\leq 3, \\x_4 - x_3 &\leq 5, \\x_4 - x_5 &\leq 10, \\x_5 - x_3 &\leq -4, \\x_5 - x_4 &\leq -8.\end{aligned}$$

### 24.4-3

Algum peso de caminho mais curto a partir do novo vértice  $v_0$  em um grafo de restrição pode ser positivo? Explique.

### 24.4-4

Expresse o problema de caminhos mais curtos de um único par como um programa linear.

### 24.4-5

Mostre como modificar ligeiramente o algoritmo de Bellman-Ford de tal forma que, quando ele for utilizado para resolver um sistema de restrições de diferenças com  $m$  desigualdades sobre  $n$  incógnitas, o tempo de execução seja  $O(nm)$ .

### 24.4-6

Suponha que, além de um sistema de restrições de diferenças, queremos manipular **restrições de igualdade** da forma  $x_i = x_j + b_k$ . Mostre como o algoritmo de Bellman-Ford pode ser adaptado para resolver essa variedade de sistema de restrições.

### 24.4-7

Mostre como um sistema de restrições de diferenças pode ser resolvido por um algoritmo semelhante ao de Bellman-Ford que seja executado sobre um grafo de restrição sem o vértice extra  $v_0$ .

#### 24.4-8 \*

Seja  $Ax \leq b$  um sistema de  $m$  restrições de diferenças em  $n$  incógnitas. Mostre que o algoritmo de Bellman-Ford, quando é executado sobre o grafo de restrição correspondente, maximiza  $\sum_{i=1}^n x_i$ , de acordo com  $Ax \leq b$  e  $x_i \leq 0$  para todo  $x_i$ .

#### 24.4-9 \*

Mostre que o algoritmo de Bellman-Ford, quando executado sobre o grafo de restrição para um sistema  $Ax \leq b$  de restrições de diferenças, minimiza a quantidade  $(\max\{x_i\} - \min\{x_i\})$  de acordo com  $Ax \leq b$ . Explique como esse fato poderia ser útil se o algoritmo fosse usado para programar serviços de construção.

#### 24.4-10

Suponha que toda linha na matriz  $A$  de um programa linear  $Ax \leq b$  corresponda a uma restrição de diferença, uma restrição de variável única da forma  $x_i \leq b_k$ , ou a uma restrição de variável única da forma  $-x_i \leq b_k$ . Mostre como o algoritmo de Bellman-Ford pode ser adaptado para resolver essa variedade de sistema de restrições.

#### 24.4-11

Forneça um algoritmo eficiente para resolver um sistema  $Ax \leq b$  de restrições de diferenças quando todos os elementos de  $b$  são valores reais e todas as incógnitas  $x_i$  devem ser inteiros.

#### 24.4-12 \*

Forneça um algoritmo eficiente para resolver um sistema  $Ax \leq b$  de restrições de diferenças quando todos os elementos de  $b$  são valores reais e um subconjunto especificado de algumas, mas não necessariamente todas, as incógnitas  $x_i$  devem ser inteiros.

## 24.5 Provas de propriedades de caminhos mais curtos

Ao longo deste capítulo, nossos argumentos de correção se basearam na desigualdade de triângulos, na propriedade do limite superior, na propriedade de nenhum caminho, na propriedade de convergência, na propriedade de relaxamento de caminhos e na propriedade de subgrafo predecessor. Enunciámos essas propriedades sem prová-las no início deste capítulo. Nesta seção, vamos demonstrar cada uma delas.

### A desigualdade de triângulos

No estudo da busca em largura (Seção 22.2), provamos no Lema 22.1 uma propriedade simples das distâncias mais curtas em grafos não-ponderados. A desigualdade de triângulos a seguir generaliza a propriedade para grafos ponderados.

#### Lema 24.10 (Desigualdade de triângulos)

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbf{R}$  e vértice de origem  $s$ . Então, para todas as arestas  $(u, v) \in E$ , temos

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

**Prova** Suponha que houvesse um caminho mais curto  $p$  desde a origem  $s$  até o vértice  $v$ . Então,  $p$  não tem peso maior que qualquer outro caminho desde  $s$  até  $v$ . Especificamente, o caminho  $p$  não tem peso maior que o caminho específico que toma um caminho mais curto da origem  $s$  até o vértice  $u$ , e depois toma a aresta  $(u, v)$ .

O Exercício 24.5-3 lhe pede para tratar o caso em que não existe nenhum caminho mais cur-

## Efeitos do relaxamento sobre estimativas de caminhos mais curtos

O próximo grupo de lemas descreve como as estimativas de caminhos mais curtos são afetadas quando executamos uma seqüência de etapas de relaxamento nas arestas de um grafo orientado ponderado que foi inicializado por INITIALIZE-SINGLE-SOURCE.

### Lema 24.11 (Propriedade de limite superior)

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbb{R}$ . Seja  $s \in V$  o vértice de origem, e seja o grafo inicializado por INITIALIZE-SINGLE-SOURCE( $G, s$ ). Então,  $d[v] \geq \delta(s, v)$  para todo  $v \in V$ , e esse invariante é mantido sobre qualquer seqüência de etapas de relaxamento nas arestas de  $G$ . Além disso, uma vez que  $d[v]$  alcança seu limite inferior  $\delta(s, v)$ , ela nunca se altera.

**Prova** Provamos o invariante  $d[v] \geq \delta(s, v)$  para todos os vértices  $v \in V$  por indução sobre o número de etapas de relaxamento.

No caso básico,  $d[v] \geq \delta(s, v)$  certamente é verdadeiro após a inicialização, pois  $d[s] = 0 \geq \delta(s, s)$  (observe que  $\delta(s, s)$  é  $-\infty$  se  $s$  está em um ciclo de peso negativo e 0 em caso contrário) e  $d[v] = \infty$  implica  $d[v] \geq \delta(s, v)$  para todo  $v \in V - \{s\}$ .

Para a etapa indutiva, considere o relaxamento de uma aresta  $(u, v)$ . Pela hipótese indutiva,  $d[x] \geq \delta(s, x)$  para todo  $x \in V$  antes do relaxamento. O único valor  $d$  que pode mudar é  $d[v]$ . Se ele mudar, temos

$$\begin{aligned} d[v] &= d[u] + w(u, v) \\ &\geq \delta(s, u) + w(u, v) \quad (\text{por hipótese indutiva}) \\ &\geq \delta(s, v) \quad (\text{pela desigualdade de triângulos}), \end{aligned}$$

e assim o invariante é mantido.

Para ver que o valor de  $d[v]$  nunca muda depois que  $d[v] = \delta(s, v)$ , observe que, tendo alcançado seu limite inferior,  $d[v]$  não pode diminuir porque acabamos de mostrar que  $d[v] \geq \delta(s, v)$ , e ele não pode aumentar porque as etapas de relaxamento não aumentam valores de  $d$ . ■

### Corolário 24.12 (Propriedade de nenhum caminho)

Vamos supor que, em um grafo orientado ponderado  $G = (V, E)$  com função peso  $w : E \rightarrow \mathbb{R}$ , nenhum caminho conecte um vértice de origem  $s \in V$  a um dado vértice  $v \in V$ . Então, depois que o grafo é inicializado por INITIALIZE-SINGLE-SOURCE( $G, s$ ), temos  $d[v] = \delta(s, v) = \infty$ , e essa igualdade é mantida como um invariante sobre qualquer seqüência de etapas de relaxamento nas arestas de  $G$ .

**Prova** Pela propriedade de limite superior, sempre temos  $\infty = \delta(s, v) \leq d[v]$  e, portanto,  $d[v] = \infty = \delta(s, v)$ . ■

### Lema 24.13

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbb{R}$ , e seja  $(u, v) \in E$ . Então, imediatamente depois de relaxar a aresta  $(u, v)$  pela execução de RELAX( $u, v, w$ ), temos  $d[v] \leq d[u] + w(u, v)$ .

**Prova** Se, imediatamente antes de relaxar a aresta  $(u, v)$ , temos  $d[v] > d[u] + w(u, v)$ , então  $d[v] = d[u] + w(u, v)$  daí em diante. Se, em vez disso,  $d[v] \leq d[u] + w(u, v)$  imediatamente antes do relaxamento, então nem  $d[u]$  nem  $d[v]$  se altera, e assim  $d[v] \leq d[u] + w(u, v)$  daí em diante. ■

### Lema 24.14 (Propriedade de convergência)

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbb{R}$ , seja  $s \in V$  um vértice de origem, e seja  $s \rightsquigarrow u \rightarrow v$  um caminho mais curto em  $G$  para alguns vértices  $u, v \in V$ . Suponha que  $G$  seja inicializado por INITIALIZE-SINGLE-SOURCE( $G, s$ ) e que depois seja executada uma

seqüência de etapas de relaxamento que inclua a chamada  $\text{RELAX}(u, v, w)$  sobre as arestas de  $G$ . Se  $d[u] = \delta(s, u)$  em qualquer instante anterior à chamada, então  $d[u] = \delta(s, v)$  em todos os momentos após a chamada.

**Prova** Pela propriedade do limite superior, se  $d[u] = \delta(s, u)$  em um certo ponto antes de se relaxar a aresta  $(u, v)$ , então essa igualdade se mantém válida daí em diante. Em particular, após o relaxamento da aresta  $(u, v)$ , temos ■

$$\begin{aligned} d[v] &\geq d[u] + w(u, v) \\ &= \delta(s, u) + w(u, v) \quad (\text{pelo Lema 24.13}) \\ &= \delta(s, v) \quad (\text{pelo Lema 24.1}) . \end{aligned}$$

Pela propriedade do limite superior,  $d[v] \geq \delta(s, v)$ ; disso concluímos que  $d[v] = \delta(s, v)$  e essa igualdade é mantida daí em diante. ■

### Lema 24.15 (Propriedade de relaxamento de caminho)

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbf{R}$ , e seja  $s \in V$  um vértice de origem. Considere qualquer caminho mais curto  $p = \langle v_0, v_1, \dots, v_k \rangle$  de  $s = v_0$  a  $v_k$ . Se  $G$  é inicializado por  $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$  e depois ocorre uma seqüência de etapas de relaxamento que inclui, em ordem, os relaxamentos de arestas  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ , então  $d[v_k] = \delta(s, v_k)$  depois desses relaxamentos e em todo momento daí em diante.

Essa propriedade se mantém válida, não importando que outros relaxamentos de arestas ocorram, inclusive relaxamentos que sejam entremeados com relaxamentos das arestas de  $p$ .

**Prova** Mostramos por indução que, depois que a  $i$ -ésima aresta do caminho  $p$  é relaxada, temos  $d[v_i] = \delta(s, v_i)$ . Para a base,  $i = 0$ , e antes de quaisquer arestas de  $p$  terem sido relaxadas, temos a partir da inicialização que  $d[v_0] = d[s] = 0 = \delta(s, s)$ . Pela propriedade do limite superior, o valor de  $d[s]$  nunca se altera depois da inicialização.

Para a etapa indutiva, supomos que  $d[v_{i-1}] = \delta(s, v_{i-1})$ , e examinamos o relaxamento de aresta  $(v_{i-1}, v_i)$ . Pela propriedade de convergência, depois desse relaxamento, temos  $d[v_i] = \delta(s, v_i)$ , e essa igualdade é mantida por todo o tempo depois disso. ■

## Relaxamento e árvores de caminhos mais curtos

Mostramos agora que, uma vez que uma seqüência de relaxamentos calcula os pesos reais de caminhos mais curtos, o subgrafo predecessor  $G_\pi$  induzido pelos valores de  $\pi$  resultantes é uma árvore de caminhos mais curtos para  $G$ . Começamos com o lema a seguir, que mostra que o subgrafo predecessor sempre forma uma árvore enraizada cuja raiz é a origem.

### Lema 24.16

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbf{R}$  e seja o vértice de origem  $s \in V$ , e suponha que  $G$  não contenha nenhum ciclo de peso negativo que seja acessível a partir de  $s$ . Então, depois que o grafo é inicializado por  $\text{INITIALIZE-SINGLE-SOURCE}(G, s)$ , o subgrafo predecessor  $G_\pi$  forma uma árvore enraizada com raiz  $s$ , e qualquer seqüência de etapas de relaxamento sobre arestas de  $G$  mantém essa propriedade como um invariante.

**Prova** Inicialmente, o único vértice em  $G_\pi$  é o vértice de origem, e o lema é trivialmente verdadeiro. Considere um subgrafo predecessor  $G_\pi$  que surge após uma seqüência de etapas de relaxamento. Primeiro, provaremos que  $G_\pi$  é acíclico. Suponha por contradição que alguma etapa de relaxamento crie um ciclo no grafo  $G_\pi$ . Seja o ciclo  $c = \langle v_0, v_1, \dots, v_k \rangle$ , onde  $v_k = v_0$ . Então,  $\pi[v_i] = v_{i-1}$  para  $i = 1, 2, \dots, k$  e, sem perda de generalidade, podemos supor que foi o relaxamento da aresta  $(v_{k-1}, v_k)$  que criou o ciclo em  $G_\pi$ .

Afirmamos que todos os vértices no ciclo  $c$  são acessíveis a partir da origem  $s$ . Por quê? Cada vértice em  $c$  tem um predecessor não NIL, e assim cada vértice em  $c$  recebe a atribuição de uma estimativa finita de caminhos mais curtos quando recebe a atribuição de seu valor  $\pi$  não NIL. Pela propriedade do limite superior, cada vértice no ciclo  $c$  tem um peso de caminho mais curto finito, o que implica que ele é acessível a partir de  $s$ .

Examinaremos as estimativas de caminhos mais curtos em  $c$  imediatamente antes da chamada a  $\text{RELAX}(v_{k-1}, v_k, w)$  e mostraremos que  $c$  é um ciclo de peso negativo, contradizendo assim a hipótese de que  $G$  não contém nenhum ciclo de peso negativo que seja acessível a partir da origem. Imediatamente antes da chamada, temos  $\pi[v_i] = v_{i-1}$  para  $i = 1, 2, \dots, k-1$ . Desse modo, para  $i = 1, 2, \dots, k-1$ , a última atualização para  $d[v_i]$  foi feita pela atribuição  $d[v_i] \leftarrow d[v_{i-1}] + w(v_{i-1}, v_i)$ . Se  $d[v_{i-1}]$  mudou desde então, ela diminuiu. Por essa razão, imediatamente antes da chamada a  $\text{RELAX}(v_{k-1}, v_k, w)$ , temos

$$d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i) \quad \text{para todo } i = 1, 2, \dots, k-1. \quad (24.12)$$

Como  $\pi[v_k]$  é alterado pela chamada, imediatamente antes também temos a desigualdade estrita

$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k).$$

Somando essa desigualdade estrita com as  $k-1$  desigualdades (24.12), obtemos o somatório dos valores de caminhos mais curtos em torno do ciclo  $c$ :

$$\begin{aligned} \sum_{i=1}^k d[v_i] &> \sum_{i=1}^k (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i). \end{aligned}$$

Porém,

$$\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}],$$

pois cada vértice no ciclo  $c$  aparece exatamente uma vez em cada somatório. Essa igualdade implica

$$0 > \sum_{i=1}^k w(v_{i-1}, v_i).$$

Desse modo, a soma dos pesos em torno do ciclo  $c$  é negativa, fornecendo assim a contradição desejada.

Agora, provamos que  $G_\pi$  é um grafo acíclico orientado. Para mostrar que ele forma uma árvore enraizada com raiz  $s$ , basta (ver Exercício B.5-2) provar que, para cada vértice  $v \in V_\pi$ , existe um único caminho desde  $s$  até  $v$  em  $G_\pi$ .

Primeiro, devemos mostrar que existe um caminho de  $s$  até cada vértice em  $V_\pi$ . Os vértices em  $V_\pi$  são aqueles com valores  $\pi$  não NIL, mais  $s$ . No caso, a idéia é provar por indução que existe um caminho de  $s$  até todos os vértices em  $V_\pi$ . Os detalhes são deixados para o Exercício 24.5-6.

Para completar a prova do lema, devemos mostrar agora que para qualquer vértice  $v \in V_\pi$ , existe no máximo um caminho de  $s$  até  $v$  no grafo  $G_\pi$ . Vamos supor o caso contrário. Isto é, suponha que existam dois caminhos simples de  $s$  até algum vértice  $v$ :  $p_1$ , que pode ser decomposto em  $s \sim u \sim x \sim z \sim v$ , e  $p_2$ , que pode ser decomposto em  $s \sim u \sim y \sim z \sim v$ , onde  $x \neq y$ . (Ver Figura 24.9.) Porém, então  $\pi[z] = x$  e  $\pi[z] = y$ , o que implica a contradição de que  $x = y$ . Concluímos que existe um caminho simples único em  $G_\pi$  desde  $s$  até  $v$ , e portanto  $G_\pi$  forma uma árvore enraizada com raiz  $s$ .

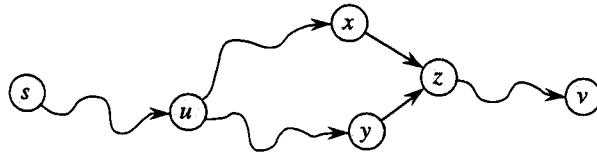


FIGURA 24.9 Mostrando que um caminho em  $G_\pi$  desde a origem  $s$  até o vértice  $v$  é único. Se existissem dois caminhos  $p_1$  ( $s \rightarrow u \rightarrow x \rightarrow z \rightarrow v$ ) e  $p_2$  ( $s \rightarrow u \rightarrow y \rightarrow z \rightarrow v$ ), onde  $x \neq y$ , então  $\pi[z] = x$  e  $\pi[z] = y$ , uma contradição

Agora, podemos mostrar que, se depois de executarmos uma seqüência de etapas de relaxamento, todos os vértices tiverem recebido a atribuição de seus pesos de caminhos mais curtos verdadeiros, então o subgrafo predecessor  $G_\pi$  será uma árvore de caminhos mais curtos.

#### **Lema 24.17 (Propriedade de subgrafo predecessor)**

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbb{R}$  e o vértice de origem  $s \in V$ , e suponha que  $G$  não contenha nenhum ciclo de peso negativo que seja acessível a partir de  $s$ . Vamos chamar INITIALIZE-SINGLE-SOURCE( $G, s$ ), e depois executar qualquer seqüência de etapas de relaxamento sobre arestas de  $G$  que produza  $d[v] = \delta(s, v)$  para todo  $v \in V$ . Então, o subgrafo predecessor  $G_\pi$  é uma árvore de caminhos mais curtos com raiz em  $s$ .

**Prova** Devemos provar que as três propriedades de árvores de caminhos mais curtos se mantêm válidas para  $G_\pi$ . Para ilustrar a primeira propriedade, devemos mostrar que  $V_\pi$  é o conjunto de vértices acessíveis a partir de  $s$ . Por definição, um peso de caminho mais curto  $\delta(s, v)$  é finito se e somente se  $v$  é acessível a partir de  $s$ , e portanto os vértices que são acessíveis a partir de  $s$  são exatamente aqueles que têm valores de  $d$  finitos. Porém, um vértice  $v \in V - \{s\}$  recebe a atribuição de um valor finito para  $d[v]$  se e somente se  $\pi[v] \neq \text{NIL}$ . Desse modo, os vértices em  $V_\pi$  são exatamente aqueles acessíveis a partir de  $s$ .

A segunda propriedade decorre diretamente do Lema 24.16.

Então, resta provar a última propriedade de árvores de caminhos mais curtos: para cada vértice  $v \in V_\pi$ , o único caminho simples  $\xrightarrow{p}$  em  $G_\pi$  é um caminho mais curto desde  $s$  até  $v$  em  $G$ . Seja  $p = \langle v_0, v_1, \dots, v_k \rangle$ , onde  $v_0 = s$  e  $v_k = v$ . Para  $i = 1, 2, \dots, k$ , temos  $d[v_i] = \delta(s, v_i)$  e  $d[v_i] = \geq d[v_{i-1}] + w(v_{i-1}, v_i)$ , de onde concluímos  $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$ . O somatório dos pesos ao longo do caminho  $p$  produz

$$\begin{aligned} w(p) &= \sum_{i=1}^k w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^k (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) \quad (\text{porque o somatório se encaixa}) \\ &= \delta(s, v_k) \quad (\text{porque } \delta(s, v_0) = \delta(s, s) = 0). \end{aligned}$$

Desse modo,  $w(p) \leq \delta(s, v_k)$ . Tendo em vista que  $\delta(s, v_k)$  é um limite inferior sobre o peso de qualquer caminho de  $s$  até  $v_k$ , concluímos que  $w(p) \leq \delta(s, v_k)$ , e portanto  $p$  é um caminho mais curto desde  $s$  até  $v = v_k$ . ■

## Exercícios

### 24.5-1

Forneça duas árvores de caminhos mais curtos para o grafo orientado da Figura 24.2, além das duas mostradas.

### 24.5-2

Forneça um exemplo de um grafo orientado ponderado  $G = (V, E)$  com função peso  $w : E \rightarrow \mathbf{R}$  e origem  $s$  tal que  $G$  satisfaça à seguinte propriedade: para toda aresta  $(u, v) \in E$ , existe uma árvore de caminhos mais curtos com raiz em  $s$  que contém  $(u, v)$  e outra árvore de caminhos mais curtos com raiz em  $s$  que não contém  $(u, v)$ .

### 24.5-3

Faça um aperfeiçoamento na prova do Lema 24.10 para manipular casos nos quais os pesos de caminhos mais curtos são  $\infty$  ou  $-\infty$ .

### 24.5-4

Seja  $G = (V, E)$  um grafo orientado ponderado com vértice de origem  $s$ , e seja  $G$  inicializado por INITIALIZE-SINGLE-SOURCE( $G, s$ ). Prove que, se uma seqüência de etapas de relaxamento definir  $\pi[s]$  com um valor não NIL, então  $G$  conterá um ciclo de peso negativo.

### 24.5-5

Seja  $G = (V, E)$  um grafo orientado ponderado sem arestas de peso negativo. Seja  $s \in V$  o vértice de origem, e seja a definição de  $\pi[v]$  da maneira habitual:  $\pi[v]$  é o predecessor de  $v$  em qualquer caminho mais curto até  $v$  desde a origem  $s$ , se  $v \in V - \{s\}$  é acessível a partir de  $s$ , e NIL em caso contrário. Forneça um exemplo de tal grafo  $G$  e de uma atribuição de valores de  $\pi$  que produza um ciclo em  $G_\pi$ . (Pelo Lema 24.16, tal atribuição não pode ser produzida por uma seqüência de etapas de relaxamento.)

### 24.5-6

Seja  $G = (V, E)$  um grafo orientado ponderado com função peso  $w : E \rightarrow \mathbf{R}$  e sem ciclos de peso negativo. Seja  $s \in V$  o vértice de origem, e seja  $G$  inicializado por INITIALIZE-SINGLE-SOURCE( $G, s$ ). Prove que, para todo vértice  $v \in V_\pi$ , existe um caminho desde  $s$  até  $v$  em  $G_\pi$ , e que essa propriedade é mantida como um invariante sobre qualquer seqüência de relaxamentos.

### 24.5-7

Seja  $G = (V, E)$  um grafo orientado ponderado que não contém nenhum ciclo de peso negativo. Seja  $s \in V$  o vértice de origem, e seja  $G$  inicializado por INITIALIZE-SINGLE-SOURCE( $G, s$ ). Prove que existe uma seqüência de  $|V| - 1$  etapas de relaxamento que produz  $d[v] = \delta(s, v)$  para todo  $v \in V$ .

### 24.5-8

Seja  $G$  um grafo orientado ponderado arbitrário com um ciclo de peso negativo acessível a partir do vértice de origem  $s$ . Mostre que sempre pode ser construída uma seqüência infinita de relaxamentos das arestas de  $G$  tal que todo relaxamento provoque a mudança em uma estimativa de caminho mais curto.

## Problemas

### 24.1 Aperfeiçoamento de Yen para Bellman-Ford

Suponha que façamos a ordenação dos relaxamentos de arestas em cada passagem do algoritmo de Bellman-Ford como a seguir. Antes da primeira passagem, atribuímos uma ordem linear arbitrária  $v_1, v_2, \dots, v_{|V|}$  aos vértices do grafo de entrada  $G = (V, E)$ . Em seguida, particionamos o conjunto de arestas  $E$  em  $E_f \cup E_b$ , onde  $E_f = \{(v_i, v_j) \in E : i < j\}$  e  $E_b = \{(v_i, v_j) \in E : i > j\}$ . (Suponha que  $G$  não contém nenhum autoloop, de forma que toda aresta está em  $E_f$  ou  $E_b$ .) Defina  $G_f = (V, E_f)$  e  $G_b = (V, E_b)$ .

- Prove que  $G_f$  é acíclico com ordenação topológica  $\langle v_1, v_2, \dots, v_{|V|} \rangle$  e que  $G_b$  é acíclico com ordenação topológica  $\langle v_{|V|}, v_{|V|-1}, \dots, v_1 \rangle$ .

Suponha que implementamos cada passagem do algoritmo de Bellman-Ford da maneira descrita a seguir. Visitamos cada vértice na ordem  $v_1, v_2, \dots, v_{|V|}$ , relaxando as arestas de  $E_f$  que deixam o vértice. Depois, visitamos cada vértice na ordem  $v_{|V|}, v_{|V|-1}, \dots, v_1$ , relaxando as arestas de  $E_b$  que deixam o vértice.

- b.** Prove que, com esse esquema, se  $G$  não contém nenhum ciclo de peso negativo que seja acessível a partir do vértice de origem  $s$ , então após apenas  $\lceil |V|/2 \rceil$  passagens sobre as arestas,  $d[v] = \delta(s, v)$  para todos os vértices  $v \in V$ .
- c.** Esse esquema melhora o tempo de execução assintótico do algoritmo de Bellman-Ford?

#### 24-2 Aninhamento de caixas

Uma caixa  $d$  dimensional com dimensões  $(x_1, x_2, \dots, x_d)$  se **aninha** dentro de outra caixa com dimensões  $(y_1, y_2, \dots, y_d)$  se existe uma permutação  $\pi$  sobre  $\{1, 2, \dots, d\}$  tal que  $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \dots, x_{\pi(d)} < y_d$ .

- a.** Demonstre que a relação de aninhamento é transitiva.
- b.** Descreva um método eficiente para determinar se uma caixa  $d$  dimensional se aninha ou não dentro de outra.
- c.** Suponha que você tenha recebido um conjunto de  $n$  caixas  $d$  dimensionais  $\{B_1, B_2, \dots, B_n\}$ . Descreva um algoritmo eficiente para determinar a mais longa seqüência  $\langle B_{i_1}, B_{i_2}, \dots, B_{i_k} \rangle$  de caixas tal que  $B_{i_j}$  fique aninhada dentro de  $B_{i_{j+1}}$  para  $j = 1, 2, \dots, k-1$ . Expresse o tempo de execução do seu algoritmo em termos de  $n$  e  $d$ .

#### 24-3 Arbitragem

A **arbitragem** é o uso de discrepâncias em taxas de câmbio para transformar uma unidade de uma moeda em mais de uma unidade da mesma moeda. Por exemplo, suponha que 1 dólar dos EUA compre 46,4 rupias indianas. Uma rupia Indiana compra 2,5 ienes japoneses, e um iene japonês compra 0,0091 dólares americanos. Então, pela conversão de moedas, um comerciante pode começar com 1 dólar americano e comprar  $46,4 \times 2,5 \times 0,0091 = 1,0556$  dólares americanos, obtendo assim um lucro de 5,56%.

Suponha que tenhamos recebido  $n$  moedas correntes  $c_1, c_2, \dots, c_n$  e uma tabela  $n \times n R$  de taxas de câmbio, tal que uma unidade da moeda  $c_i$  compre  $R[i, j]$  unidades da moeda  $c_j$ .

- a.** Forneça um algoritmo eficiente para determinar se existe ou não uma seqüência de moedas  $\langle c_{i_1}, c_{i_2}, \dots, c_{i_k} \rangle$  tal que

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Analise o tempo de execução do seu algoritmo.

- b.** Forneça um algoritmo eficiente para imprimir tal seqüência, se ela existir. Analise o tempo de execução do seu algoritmo.

#### 24-4 Algoritmo de escalonamento de Gabow para caminhos mais curtos de única origem

O algoritmo de **escalonamento** resolve um problema considerando inicialmente apenas o bit de mais alta ordem de cada valor de entrada relevante (como um peso de aresta). Em seguida, ele refina a solução inicial, observando os dois bits de mais alta ordem. Progressivamente, ele examina bits de ordem cada vez mais alta, refinando a solução a cada vez, até todos os bits terem sido considerados e a solução correta ter sido calculada.

Neste problema, examinaremos um algoritmo para calcular os caminhos mais curtos desde uma única origem, pelo escalonamento de pesos de arestas. Temos um grafo orientado  $G = (V, E)$  com pesos de arestas inteiros não negativos  $w$ . Seja  $W = \max_{(u, v) \in E} \{w(u, v)\}$ . Nossa meta é desenvolver um algoritmo que seja executado no tempo  $O(E \lg W)$ . Supomos que todos os vértices são acessíveis a partir da origem.

O algoritmo descobre os bits na representação binária dos pesos de arestas um de cada vez, desde o bit mais significativo até o bit menos significativo. Especificamente, seja  $k = \lceil \lg(W + 1) \rceil$  o número de bits na representação binária de  $W$  e, para  $i = 1, 2, \dots, k$ , seja  $w_i(u, v) = \lfloor w(u, v)/2^{k-i} \rfloor$ . Isto é,  $w_i(u, v)$  é a versão “em escala reduzida” de  $w(u, v)$  dada pelos  $i$  bits mais significativos de  $w(u, v)$ . (Desse modo,  $w_k(u, v) = w(u, v)$  para todo  $(u, v) \in E$ .) Por exemplo, se  $k = 5$  e  $w(u, v) = 25$ , que tem a representação binária  $\langle 11001 \rangle$ , então  $w_3(u, v) = \langle 110 \rangle = 6$ . Como outro exemplo com  $k = 5$ , se  $w(u, v) = \langle 00100 \rangle = 4$ , então  $w_3(u, v) = \langle 001 \rangle = 1$ . Vamos definir  $\delta(u, v)$  como o peso do caminho mais curto desde o vértice  $u$  até o vértice  $v$ , utilizando a função peso  $w_i$ . Desse modo,  $\delta_k(u, v) = \delta(u, v)$  para todo  $u, v \in V$ . Para um dado vértice de origem  $s$ , o algoritmo de escalonamento calcula primeiro os pesos de caminhos mais curtos  $\delta_1(s, v)$  para todo  $v \in V$ , depois calcula  $\delta_2(s, v)$  para todo  $v \in V$  e assim por diante, até calcular  $\delta_k(s, v)$  para todo  $v \in V$ . Supomos em todo esse processo que  $|E| \geq |V| - 1$ , e veremos que o cálculo de  $\delta_i$  a partir de  $\delta_{i-1}$  demora o tempo  $O(E)$ , de modo que o algoritmo inteiro demora o tempo  $O(kE) = O(E \lg W)$ .

- a. Suponha que para todos os vértices  $v \in V$ , temos  $\delta(s, v) \leq |E|$ . Mostre que podemos calcular  $\delta(s, v)$  para todo  $v \in V$  no tempo  $O(E)$ .
- b. Mostre que podemos calcular  $\delta_1(s, v)$  para todo  $v \in V$  no tempo  $O(E)$ .

Agora, vamos nos concentrar no cálculo de  $\delta_i$  a partir de  $\delta_{i-1}$ .

- c. Prove que, para  $i = 2, 3, \dots, k$ , temos também  $w_i(u, v) = 2w_{i-1}(u, v)$  ou  $w_i(u, v) = 2w_{i-1}(u, v) + 1$ . Em seguida, prove que

$$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$$

para todo  $v \in V$ .

- d. Defina, para  $i = 2, 3, \dots, k$  e para todo  $(u, v) \in E$ ,

$$\hat{w}_i(u, v) = w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v).$$

Prove que, para  $i = 2, 3, \dots, k$  e todo  $u, v \in V$ , o valor “reponderado”  $\hat{w}_i(u, v)$  da aresta  $(u, v)$  é um inteiro não negativo.

- e. Agora, defina  $\hat{\delta}(s, v)$  como o peso do caminho mais curto desde  $s$  até  $v$ , usando a função peso  $\hat{w}_i$ . Prove que, para  $i = 2, 3, \dots, k$  e todo  $v \in V$ ,

$$\delta_i(s, v) = \hat{\delta}(s, v) + 2\delta_{i-1}(s, v)$$

e que  $\hat{\delta}(s, v) \leq |E|$ .

- f. Mostre como calcular  $\delta(s, v)$  a partir de  $\delta_{i-1}(s, v)$  para todo  $v \in V$  no tempo  $O(E)$ , e conclua que  $\delta(s, v)$  pode ser calculado para todo  $v \in V$  no tempo  $O(E \lg W)$ .

#### 24-5 Algoritmo do ciclo de peso médio mínimo de Karp

Seja  $G = (V, E)$  um grafo orientado com função peso  $w : E \rightarrow \mathbb{R}$ , e seja  $n = |V|$ . Definimos o **peso médio** de um ciclo  $c = \langle e_1, e_2, \dots, e_k \rangle$  de arestas em  $E$  como

$$\mu(c) = \frac{1}{k} \sum_{i=1}^k w(e_i).$$

Seja  $\mu^* = \min_c \mu(c)$ , onde  $c$  varia sobre todos os ciclos orientados em  $G$ . Um ciclo  $c$  para o qual  $\mu(c) = \mu^*$  é chamado **ciclo de peso médio mínimo**. Este problema investiga um algoritmo eficiente para cálculo de  $\mu^*$ .

Suponha sem perda de generalidade que todo vértice  $v \in V$  seja acessível a partir de um vértice de origem  $s \in V$ . Seja  $\delta(s, v)$  o peso de um caminho mais curto desde  $s$  até  $v$ , e seja  $\delta_k(s, v)$  o peso de um caminho mais curto desde  $s$  até  $v$  consistindo em exatamente  $k$  arestas. Se não existe nenhum caminho de  $s$  até  $v$  com exatamente  $k$  arestas, então  $\delta_k(s, v) = \infty$ .

- Mostre que, se  $\mu^* = 0$ , então  $G$  não contém nenhum ciclo de peso negativo e  $\delta(s, v) = \min_{0 \leq k \leq n-1} \delta_k(s, v)$  para todos os vértices  $v \in V$ .
- Mostre que, se  $\mu^* = 0$ , então

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0$$

para todos os vértices  $v \in V$ . (Sugestão: Use ambas as propriedades da parte (a).)

- Seja  $c$  um ciclo de peso 0, e sejam  $u$  e  $v$  dois vértices quaisquer em  $c$ . Suponha que o peso do caminho desde  $u$  até  $v$  ao longo do ciclo seja  $x$ . Prove que  $\delta(s, v) = \delta(s, u) + x$ . (Sugestão: O peso do caminho desde  $v$  até  $u$  ao longo do ciclo é  $-x$ .)
- Mostre que, se  $\mu^* = 0$ , então existe um vértice  $v$  no ciclo de peso médio mínimo tal que

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

(Sugestão: Mostre que um caminho mais curto até qualquer vértice no ciclo de peso médio mínimo pode ser estendido ao longo do ciclo para formar um caminho mais curto até o próximo vértice no ciclo.)

- Mostre que, se  $\mu^* = 0$ , então

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

- Mostre que, se adicionarmos uma constante  $t$  ao peso de cada aresta de  $G$ , então  $\mu^*$  será aumentado em  $t$ . Use isso para mostrar que

$$\mu^* = \min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

- Forneça um algoritmo de tempo  $O(VE)$  para calcular  $\mu^*$ .

#### 24-6 Caminhos mais curtos bitônicos

Uma seqüência é **bitônica** se aumenta monotonicamente e depois diminui monotonicamente, ou se ela pode ser deslocada de forma circular para aumentar monotonicamente e depois diminuir monotonicamente. Por exemplo, as seqüências  $\langle 1, 4, 6, 8, 3, -2 \rangle$ ,  $\langle 92, -4, -10, -5 \rangle$  e  $\langle 1, 2, 3, 4 \rangle$  são bitônicas, mas  $\langle 1, 3, 12, 4, 2, 10 \rangle$  não é bitônica. (Veja no Capítulo 27 uma discussão sobre ordenadores bitônicos, e veja no Problema 15-1 o problema do caixeiro-viajante euclidiano bitônico.)

Suponha que temos um grafo orientado  $G = (V, E)$  com função peso  $w : E \rightarrow \mathbf{R}$  e desejamos encontrar caminhos mais curtos de única origem a partir de um vértice de origem  $s$ . Temos um item adicional de informação: para cada vértice  $v \in V$ , os pesos das arestas ao longo de qualquer caminho mais curto de  $s$  até  $v$  formam uma seqüência bitônica.

Forneça o algoritmo mais eficiente que puder para resolver esse problema, e analise seu tempo de execução.

## Notas do capítulo

O algoritmo de Dijkstra [75] surgiu em 1959, mas não continha nenhuma menção a uma fila de prioridades. O algoritmo de Bellman-Ford se baseia em algoritmos separados criados por Bellman [35] e Ford [93]. Bellman descreve a relação entre caminhos mais curtos e restrições de diferenças. Lawler [196] descreve o algoritmo de tempo linear para caminhos mais curtos em um gao, que ele considera parte do folclore.

Quando os pesos de arestas são inteiros relativamente pequenos, podem ser usados algoritmos mais eficientes para resolver o problema de caminhos mais curtos de única origem. A seqüência de valores retornados pelas chamadas de EXTRACT-MIN no algoritmo de Dijkstra é monotonicamente crescente no decorrer do tempo. Como vemos nas notas do capítulo relativas ao Capítulo 6, nesse caso existem várias estruturas de dados que podem implementar diversas operações de filas de prioridades de modo mais eficiente que um heap binário ou um heap de Fibonacci. Ahuja, Mehlhorn, Orlin e Tarjan [8] fornecem um algoritmo que é executado no tempo  $O(E + V\sqrt{\lg W})$  sobre grafos com pesos de arestas não negativos, onde  $W$  é o maior peso de qualquer aresta no grafo. Os melhores limites são dados por Thorup [299], que fornece um algoritmo que funciona no tempo  $O(E \lg \lg V)$ , e por Raman, que oferece um algoritmo que funciona no tempo  $O(E + V \min\{(\lg V)^{1/3+\varepsilon}, (\lg W)^{1/4+\varepsilon}\})$ . Esses dois algoritmos usam uma quantidade de espaço que depende do tamanho da palavra da máquina subjacente. Embora a quantidade de espaço utilizada possa ser ilimitada no tamanho da entrada, ela pode ser reduzida para ser linear no tamanho da entrada, empregando-se o hash aleatório.

Para grafos não orientados com pesos inteiros, Thorup [298] apresenta um algoritmo de tempo  $O(V + E)$  para caminhos mais curtos de única origem. Em contraste com os algoritmos mencionados no parágrafo anterior, esse algoritmo não é uma implementação do algoritmo de Dijkstra, pois a seqüência de valores retornados por chamadas de EXTRACT-MIN não é monotonicamente crescente com o tempo.

No caso de grafos com pesos de arestas negativos, um algoritmo devido a Gabow e Tarjan [104] é executado no tempo  $O(\sqrt{V}E \lg(VW))$ , e um algoritmo de Goldberg [118] funciona no tempo  $O(\sqrt{V}E \lg W)$ , onde  $W = \max_{(u, v) \in E} \{|w(u, v)|\}$ .

Cherkassky, Goldberg e Radzik [57] conduziram experiências extensas comparando vários algoritmos de caminhos mais curtos.

---

## *Capítulo 25*

### *Caminhos mais curtos de todos os pares*

Neste capítulo, consideraremos o problema de encontrar caminhos mais curtos entre todos os pares de vértices em um grafo. Esse problema poderia surgir, por exemplo, na elaboração de uma tabela de distâncias entre todos os pares de cidades para um atlas rodoviário. Como no Capítulo 24, temos um grafo orientado ponderado  $G = (V, E)$  com uma função peso  $w : E \rightarrow \mathbb{R}$  que mapeia arestas como pesos de valores reais. Desejamos encontrar, para todo par de vértices  $u, v \in V$ , um caminho mais curto (de peso mínimo) desde  $u$  até  $v$ , onde o peso de um caminho é a soma dos pesos de suas arestas constituintes. Em geral, queremos que a saída esteja em forma tabular: a entrada na linha de  $u$  e na coluna de  $v$  deve ser o peso de um caminho mais curto desde  $u$  até  $v$ .

Podemos resolver um problema de caminhos mais curtos de todos os pares executando um algoritmo de caminhos mais curtos de única origem  $|V|$  vezes, uma para cada vértice usado como a origem. Se todos os pesos de arestas são não negativos, podemos usar o algoritmo de Dijkstra. Se usarmos a implementação de arranjo linear da fila de prioridades, o tempo de execução será  $O(V^3 + VE) = O(V^3)$ . A implementação de heap binário da fila de prioridade mínima resulta em um tempo de execução  $O(VE \lg V)$ , que é uma melhoria se o grafo é esparsa. Como alternativa, podemos implementar a fila de prioridade mínima com um heap de Fibonacci, resultando em um tempo de execução  $O(V^2 \lg V + VE)$ .

Se forem permitidas arestas de peso negativo, o algoritmo de Dijkstra não poderá mais ser usado. Em vez disso, devemos executar o algoritmo de Bellman-Ford, mais lento, uma vez para cada vértice. O tempo de execução resultante é  $O(V^2E)$  que, em um grafo denso, é  $O(V^4)$ . Neste capítulo, veremos como fazer melhor. Também investigaremos a relação entre o problema de caminhos mais curtos de todos os pares e a multiplicação de matrizes, e estudaremos sua estrutura algébrica.

Diferente dos algoritmos de única origem, que pressupõem uma representação do grafo como uma lista de adjacências, a maioria dos algoritmos neste capítulo utiliza uma representação de matriz de adjacências. (O algoritmo de Johnson para grafos esparsos usa listas de adjacências.) Por conveniência, supomos que os vértices estão numerados como  $1, 2, \dots, |V|$ , de modo que a entrada é uma matriz  $W n \times n$  que representa os pesos de arestas de um grafo orientado de  $n$  vértices  $G = (V, E)$ . Isto é,  $W = (w_{ij})$ , onde

$$w_{ij} = \begin{cases} 0 & \text{se } i = j, \\ \text{o peso da aresta orientada } (i, j) & \text{se } i \neq j \text{ e } (i, j) \in E, \\ \infty & \text{se } i \neq j \text{ e } (i, j) \notin E. \end{cases} \quad (25.1)$$

Arestas de peso negativo são permitidas, mas supomos por enquanto que o grafo de entrada não contém nenhum ciclo de peso negativo.

A saída tabular dos algoritmos de caminhos mais curtos de todos os pares apresentados neste capítulo é uma matriz  $n \times n D = (d_{ij})$ , onde a entrada  $d_{ij}$  contém o peso de um caminho mais curto desde o vértice  $i$  até o vértice  $j$ . Ou seja, se fazemos  $\delta(i, j)$  denotar o peso do caminho mais curto desde o vértice  $i$  até o vértice  $j$  (como no Capítulo 24), então  $d_{ij} = \delta(i, j)$  no término.

Para resolver o problema de caminhos mais curtos de todos os pares em uma matriz de adjacências de entrada, precisamos calcular não apenas os pesos de caminhos mais curtos, mas também uma **matriz predecessora**  $\Pi = (\pi_{ij})$ , onde  $\pi_{ij}$  é NIL se  $i = j$  ou se não existe nenhum caminho desde  $i$  até  $j$  e, caso contrário,  $\pi_{ij}$  é o predecessor de  $j$  em um caminho mais curto a partir de  $i$ . Da mesma maneira que o subgrafo predecessor  $G_\pi$  do Capítulo 24 é uma árvore de caminhos mais curtos para um dado vértice de origem, o subgrafo induzido pela  $i$ -ésima linha da matriz  $\Pi$  deve ser uma árvore de caminhos mais curtos com raiz  $i$ . Para cada vértice  $i \in V$ , definimos o **subgrafo predecessor** de  $G$  para  $i$  como  $G_{\pi, i} = (V_{\pi, i}, E_{\pi, i})$ , onde

$$V_{\pi, i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

e

$$E_{\pi, i} = \{(\pi_{ij}, j) : j \in V_{\pi, i} - \{i\}\}.$$

Se  $G_{\pi, i}$  é uma árvore de caminhos mais curtos, então o procedimento a seguir, que é uma versão modificada do procedimento PRINT-PATH do Capítulo 22, imprime um caminho mais curto desde o vértice  $i$  até o vértice  $j$ .

```
PRINT-ALL-PAIRS-SHORTEST-PATH ( $\Pi, i, j$ )
1 if  $i = j$ 
2   then print  $i$ 
3 else if  $\pi_{ij} = \text{NIL}$ 
4   then print "nenhum caminho de"  $i$  "para"  $j$  "existente"
5 else PRINT-ALL-PAIRS-SHORTEST-PATH( $\Pi, i, \pi_{ij}$ )
6   print  $j$ 
```

Para realçar os recursos essenciais dos algoritmos de todos os pares deste capítulo, não abordaremos a criação e as propriedades de matrizes predecessoras tão extensivamente quanto lidamos com subgrafos predecessores no Capítulo 24. Os fundamentos serão focalizados por alguns dos exercícios.

## Esboço do capítulo

A Seção 25.1 apresenta um algoritmo de programação dinâmica baseado em multiplicação de matrizes para resolver o problema de caminhos mais curtos de todos os pares. Usando a técnica de “elevação ao quadrado repetida”, esse algoritmo pode ser elaborado para execução no tempo  $\Theta(V^3 \lg V)$ . Outro algoritmo de programação dinâmica, o algoritmo de Floyd-Warshall, é dado na Seção 25.2. O algoritmo de Floyd-Warshall é executado no tempo  $\Theta(V^3)$ . A Seção 25.2 também aborda o problema de encontrar o fechamento transitivo de um grafo orientado, o qual está

relacionado com o problema de caminhos mais curtos de todos os pares. Finalmente, a Seção 25.3 apresenta o algoritmo de Johnson. Diferente dos outros algoritmos neste capítulo, o algoritmo de Johnson utiliza a representação de um grafo por lista de adjacências. Ele resolve o problema de caminhos mais curtos de todos os pares no tempo  $O(V^2 \lg V + VE)$ , o que o torna um bom algoritmo para grafos grandes e esparsos.

Antes de prosseguirmos, precisamos estabelecer algumas convenções para representações de matrizes de adjacências. Primeiro, iremos supor em geral que o grafo de entrada  $G = (V, E)$  tem  $n$  vértices, de modo que  $n = |V|$ . Em segundo lugar, usaremos a convenção de denotar matrizes por letras maiúsculas, como  $W, L$  ou  $D$ , e seus elementos individuais por letras minúsculas com subscritos, como  $w_{ij}, l_{ij}$  ou  $d_{ij}$ . Algumas matrizes terão sobrescritos entre parênteses, como em  $L^{(m)} = (l_{ij}^{(m)})$  ou  $D^{(m)} = (d_{ij}^{(m)})$ , a fim de indicar repetições. Por fim, para uma dada matriz  $A$   $n \times n$ , faremos a suposição de que o valor de  $n$  está armazenado no atributo *linhas[A]*.

## 25.1 Caminhos mais curtos e multiplicação de matrizes

Esta seção apresenta um algoritmo de programação dinâmica para o problema de caminhos mais curtos de todos os pares sobre um grafo orientado  $G = (V, E)$ . Cada loop principal do programa dinâmico invocará uma operação que é muito semelhante à multiplicação de matrizes, de modo que o algoritmo se parecerá com uma multiplicação de matrizes repetida. Começaremos desenvolvendo um algoritmo de tempo  $\Theta(V^4)$  para o problema de caminhos mais curtos de todos os pares, e depois melhoraremos seu tempo de execução até  $\Theta(V^3 \lg V)$ .

Antes de continuar, vamos recapitular rapidamente os passos dados no Capítulo 15 para desenvolver um algoritmo de programação dinâmica.

1. Caracterizar a estrutura de uma solução ótima.
2. Definir recursivamente o valor de uma solução ótima.
3. Calcular o valor de uma solução ótima de baixo para cima.

(O quarto passo, a construção de uma solução ótima a partir de informações calculadas, será examinado nos exercícios.)

### A estrutura de um caminho mais curto

Começaremos caracterizando a estrutura de uma solução ótima. Para o problema de caminhos mais curtos de todos os pares sobre um grafo  $G = (V, E)$ , provamos (Lema 24.1) que todos os subcaminhos de um caminho mais curto são caminhos mais curtos. Suponha que o grafo seja representado por uma matriz de adjacências  $W = (w_{ij})$ . Considere um caminho mais curto  $p$  desde o vértice  $i$  até o vértice  $j$ , e suponha que  $p$  contenha no máximo  $m$  arestas. Supondo-se que não exista nenhum ciclo de peso negativo,  $m$  será finito. Se  $i = j$ , então  $p$  tem peso 0 e nenhuma aresta. Se os vértices  $i$  e  $j$  são distintos, então decomponemos o caminho  $p$  em  $i \xrightarrow{k} k \rightarrow j$ , onde o caminho  $p'$  agora contém no máximo  $m - 1$  arestas. Pelo Lema 24.1,  $p'$  é um caminho mais curto desde  $i$  até  $k$ ; desse modo,  $\delta(i, j) = \delta(i, k) + w_{kj}$ .

### Uma solução recursiva para o problema de caminhos mais curtos de todos os pares

Agora, seja  $l_{ij}^{(m)}$  o peso mínimo de qualquer caminho desde o vértice  $i$  até o vértice  $j$  que contém no máximo  $m$  arestas. Quando  $m = 0$ , existe um caminho mais curto desde  $i$  até  $j$  sem arestas se e somente se  $i = j$ . Portanto,

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{se } i = j, \\ \infty & \text{se } i \neq j, \end{cases}$$

Para  $m \geq 1$ , calculamos  $l_{ij}^{(m)}$  como o mínimo de  $l_{ij}^{(m-1)}$  (o peso do caminho mais curto desde  $i$  até  $j$ , que consiste no máximo em  $m - 1$  arestas) e o peso mínimo de qualquer caminho desde  $i$  até  $j$ , que consiste no máximo em  $m$  arestas, obtido pelo exame de todos os possíveis predecesores  $k$  de  $j$ . Desse modo, definimos recursivamente

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{l_{ij}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1 \leq k \leq n}\left\{l_{ij}^{(m-1)} + w_{kj}\right\}. \end{aligned} \quad (25.2)$$

A última igualdade decorre de  $w_{jj} = 0$  para todo  $j$ .

Quais são os pesos reais de caminhos mais curtos  $\delta(i, j)$ ? Se o grafo não contém nenhum ciclo de peso negativo, então para todo par de vértices  $i$  e  $j$  para o qual  $\delta(i, j) < \infty$ , existe um caminho mais curto de  $i$  até  $j$  que é simples e, portanto, contém no máximo  $n - 1$  arestas. Um caminho desde o vértice  $i$  até o vértice  $j$  com mais de  $n - 1$  arestas não pode ter menos peso que um caminho mais curto desde  $i$  até  $j$ . Os pesos reais de caminhos mais curtos são então dados por

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \dots. \quad (25.3)$$

### Como calcular os pesos de caminhos mais curtos de baixo para cima

Tomando como nossa entrada a matriz  $W = (w_{ij})$ , calculamos agora uma série de matrizes  $L^{(1)}, L^{(2)}, \dots, L^{(n-1)}$ , onde, para  $m = 1, 2, \dots, n - 1$ , temos  $L^{(m)}(l_{ij}^{(m)})$ . A matriz final  $L^{(n-1)}$  contém os pesos reais de caminhos mais curtos. Observe que, considerando-se  $l_{ij}^{(1)} = w_{ij}$  para todos os vértices  $i, j \in V$ , temos  $L^{(1)} = W$ .

O núcleo do algoritmo é o procedimento a seguir que, dadas as matrizes  $L^{(m-1)}$  e  $W$ , retorna a matriz  $L^{(m)}$ . Isto é, ele estende os caminhos mais curtos calculados até agora por mais uma aresta.

```
EXTEND-SHORTEST-PATHS( $L, W$ )
1  $n \leftarrow \text{linhas}[L]$ 
2 seja  $L' = (l'_{ij})$  uma matriz  $n \times n$ 
3 for  $i \leftarrow 1$  to  $n$ 
4   do for  $j \leftarrow 1$  to  $n$ 
5     do  $l'_{ij} \leftarrow \infty$ 
6     for  $k \leftarrow 1$  to  $n$ 
7       do  $l'_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w_{kj})$ 
8 return  $L'$ 
```

O procedimento calcula uma matriz  $L' = (l'_{ij})$ , que ele retorna no final. Ele faz isso calculando a equação (25.2) para todo  $i$  e  $j$ , utilizando  $L$  para  $L^{(m-1)}$  e  $L'$  para  $L^{(m)}$ . (Ele é escrito sem os subscritos para tornar suas matrizes de entrada e saída independentes de  $m$ .) Seu tempo de execução é  $\Theta(n^3)$ , devido aos três loops **for** aninhados.

Podemos agora ver a relação com a multiplicação de matrizes. Suponha que desejamos calcular o produto de matrizes  $C = A \cdot B$  de duas matrizes  $n \times n$   $A$  e  $B$ . Então, para  $i, j = 1, 2, \dots, n$ , calculamos

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}. \quad (25.4)$$

Observe que, se fizermos as substituições

$l^{(m-1)} \rightarrow a$ ,  
 $w \rightarrow b$ ,  
 $l^{(m)} \rightarrow c$ ,  
 $\min \rightarrow +$ ,  
 $+ \rightarrow \cdot$

Na equação (25.2), obteremos a equação (25.4). Desse modo, se fizermos essas mudanças em EXTEND-SHORTEST-PATHS e também substituirmos  $\infty$  (a identidade correspondente a  $\min$ ) por 0 (a identidade correspondente a  $+$ ), obteremos o procedimento direto de tempo  $\Theta(n^3)$  para multiplicação de matrizes:

**MATRIX-MULTIPLY( $A, B$ )**

- 1  $n \leftarrow \text{linhas}[A]$
- 2 seja  $C$  uma matriz  $n \times n$
- 3 **for**  $i \leftarrow 1$  **to**  $n$
- 4     **do for**  $j \leftarrow 1$  **to**  $n$
- 5         **do**  $c_{ij} \leftarrow 0$
- 6         **for**  $k \leftarrow 1$  **to**  $n$
- 7             **do**  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
- 8 **return**  $C$

Retornando ao problema de caminhos mais curtos de todos os pares, calculamos os pesos de caminhos mais curtos estendendo os caminhos mais curtos aresta por aresta. Fazendo  $A \cdot B$  denota o “produto” de matrizes retornado por EXTEND-SHORTEST-PATHS( $A, B$ ), calculamos a seqüência de  $n - 1$  matrizes

$$\begin{aligned}
 L^{(1)} &= L^{(0)} \cdot W = W, \\
 L^{(2)} &= L^{(1)} \cdot W = W^2, \\
 L^{(3)} &= L^{(2)} \cdot W = W^3, \\
 &\vdots \\
 L^{(n-1)} &= L^{(n-2)} \cdot W = W^{n-1}.
 \end{aligned}$$

Como demonstramos antes, a matriz  $L^{(n-1)} = W^{n-1}$  contém os pesos de caminhos mais curtos. O procedimento a seguir calcula essa seqüência no tempo  $\Theta(n^4)$ .

**SLOW-ALL-PAIRS-SHORTEST-PATHS( $W$ )**

- 1  $n \leftarrow \text{linhas}[W]$
- 2  $L^{(1)} \leftarrow W$
- 3 **for**  $m \leftarrow 2$  **to**  $n - 1$
- 4     **do**  $L^{(m)} \leftarrow \text{EXTEND-SHORTEST-PATHS}(L^{(m-1)}, W)$
- 5 **return**  $L^{(n-1)}$

A Figura 25.1 mostra um grafo e as matrizes  $L^{(m)}$  calculadas pelo procedimento SLOW-ALL-PAIRS-SHORTEST-PATHS.

### Como melhorar o tempo de execução

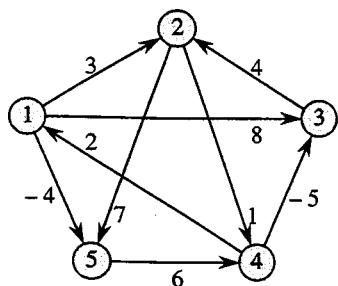
Entretanto, nosso objetivo não é calcular *todas* as matrizes  $L^{(m)}$ : só estamos interessados na matriz  $L^{(n-1)}$ . Lembre-se de que, na ausência de ciclos de peso negativo, a equação (25.3) implica  $L^{(m)} = L^{(n-1)}$  para todos os inteiros  $m \geq n - 1$ . Da mesma forma que a multiplicação de matrizes

tradicional é associativa, também é associativa a multiplicação de matrizes definida pelo procedimento EXTEND-SHORTEST-PATHS (veja o Exercício 25.1-4). Então, podemos calcular  $L^{(n-1)}$  apenas com  $\lceil \lg(n-1) \rceil$  produtos de matrizes, calculando a seqüência

$$\begin{aligned} L^{(1)} &= \\ L^{(2)} &= W^2 = W \cdot W, \\ L^{(4)} &= W^4 = W^3 \cdot W^2 \\ L^{(8)} &= W^8 = W^4 \cdot W^4, \\ &\vdots \\ L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil-1}} \cdot W^{2^{\lceil \lg(n-1) \rceil-1}} \end{aligned}$$

Tendo em vista que  $2^{\lceil \lg(n-1) \rceil} \geq n-1$ , o produto final  $L^{(2^{\lceil \lg(n-1) \rceil})}$  é igual a  $L^{(n-1)}$ .

O procedimento a seguir calcula a seqüência de matrizes anterior, empregando essa técnica de *elevação ao quadrado repetida*.



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

FIGURA 25.1 Um grafo orientado e a seqüência de matrizes  $L^{(m)}$  calculada por SLOW-ALL-PAIRS-SHORTEST-PATHS. O leitor pode verificar que  $L^{(5)} = L^{(4)} \cdot W$  é igual a  $L^{(4)}$  e, desse modo,  $L^{(m)} = L^{(4)}$  para todo  $m \geq 4$

#### FASTER-ALL-PAIRS-SHORTEST-PATHS( $W$ )

- 1  $n \leftarrow \text{linhas}[W]$
- 2  $L^{(1)} \leftarrow W$
- 3  $m \leftarrow 1$
- 4 **while**  $m < n-1$
- 5   **do**  $L^{(2m)} \leftarrow \text{EXTEND-SHORTEST-PATHS}(L^{(m)}, L^{(m)})$
- 6     $m \leftarrow 2m$
- 7 **return**  $L^{(m)}$

Em cada iteração do loop **while** das linhas 4 a 6, calculamos  $L^{(2m)} = (L^{(2m)})^2$ , começando com  $m = 1$ . No final de cada iteração, duplicamos o valor de  $m$ . A iteração final calcula  $L^{(n-1)}$  calculando na realidade  $L^{(2m)}$  para algum  $n-1 \leq 2m < 2n-2$ . Pela equação (25.3),  $L^{(2m)} = L^{(n-1)}$ . Na próxima vez em que o teste da linha 4 é executado,  $m$  é duplicado, e assim agora  $m \geq n-1$ , o teste falha e o procedimento retorna a última matriz que calculou.

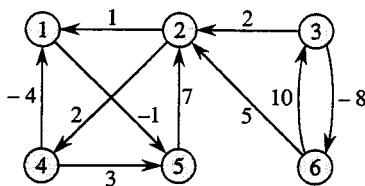


FIGURA 25.2 Um grafo orientado ponderado para uso nos Exercícios 25.1-1, 25.2-1 e 25.3-1

O tempo de execução de FASTER-ALL-PAIRS-SHORTEST-PATHS é  $\Theta(n^3 \lg n)$ , pois cada um dos  $\lceil \lg(n-1) \rceil$  produtos de matrizes demora o tempo  $\Theta(n^3)$ . Observe que o código é compacto, não contendo nenhuma estrutura de dados elaborada e que, portanto, a constante oculta na notação de  $\Theta$  é pequena.

## Exercícios

### 25.1-1

Execute SLOW-ALL-PAIRS-SHORTEST-PATHS sobre o grafo orientado ponderado da Figura 25.2, mostrando as matrizes que resultam de cada iteração do loop. Depois, faça o mesmo para FASTER-ALL-PAIRS-SHORTEST-PATHS.

### 25.1-2

Por que exigimos que  $w_{ii} = 0$  para todo  $1 \leq i \leq n$ ?

### 25.1-3

A que corresponde em multiplicação de matrizes comum a matriz

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

usada nos algoritmos de caminhos mais curtos?

### 25.1-4

Mostre que a multiplicação de matrizes definida por EXTEND-SHORTEST-PATHS é associativa.

### 25.1-5

Mostre como expressar o problema de caminhos mais curtos de única origem como um produto de matrizes e um vetor. Descreva como a avaliação desse produto corresponde a um algoritmo como o de Bellman-Ford (consulte a Seção 24.1).

### 25.1-6

Suponha que também desejamos calcular os vértices em caminhos mais curtos nos algoritmos desta seção. Mostre como calcular a matriz predecessora  $\Pi$  a partir da matriz  $L$  completada de pesos de caminhos mais curtos no tempo  $O(n^3)$ .

### 25.1-7

Os vértices em caminhos mais curtos também podem ser calculados ao mesmo tempo que os pesos de caminhos mais curtos. Vamos definir  $\pi_{ij}^{(m)}$  como o predecessor do vértice  $j$  sobre qualquer caminho de peso mínimo desde  $i$  até  $j$  que contém no máximo  $m$  arestas. Modifique EXTEND-SHORTEST-PATHS e SLOW-ALL-PAIRS-SHORTEST-PATHS para calcular as matrizes  $\Pi^{(1)}, \Pi^{(2)}, \dots, \Pi^{(n-1)}$  à medida que as matrizes  $L^{(1)}, L^{(2)}, \dots, L^{(n-1)}$  são calculadas.

### 25.1-8

O procedimento FASTER-ALL-PAIRS-SHORTEST-PATHS, como foi escrito, nos obriga a armazenar  $\lceil \lg(n-1) \rceil$  matrizes, cada uma com  $n^2$  elementos, exigindo um espaço total igual a  $\Theta(n^2 \lg n)$ . Modifique o procedimento para exigir apenas o espaço  $\Theta(n^2)$ , usando somente duas matrizes  $n \times n$ .

### 25.1-9

Modifique FASTER-ALL-PAIRS-SHORTEST-PATHS de modo que ele possa detectar a presença de um ciclo de peso negativo.

### 25.1-10

Forneça um algoritmo eficiente para encontrar o comprimento (número de arestas) de um ciclo de peso negativo de comprimento mínimo em um grafo.

## 25.2 O algoritmo de Floyd-Warshall

Nesta seção, usaremos uma formulação de programação dinâmica diferente para resolver o problema de caminhos mais curtos de todos os pares em um grafo orientado  $G = (V, E)$ . O algoritmo resultante, conhecido como **algoritmo de Floyd-Warshall**, é executado no tempo  $\Theta(V^3)$ . Como antes, arestas de peso negativo podem estar presentes, mas iremos supor que não existe nenhum ciclo de peso negativo. Como na Seção 25.1, seguiremos o processo de programação dinâmica para desenvolver o algoritmo. Depois de estudar o algoritmo resultante, apresentaremos um método semelhante para encontrar o fecho transitivo de um grafo orientado.

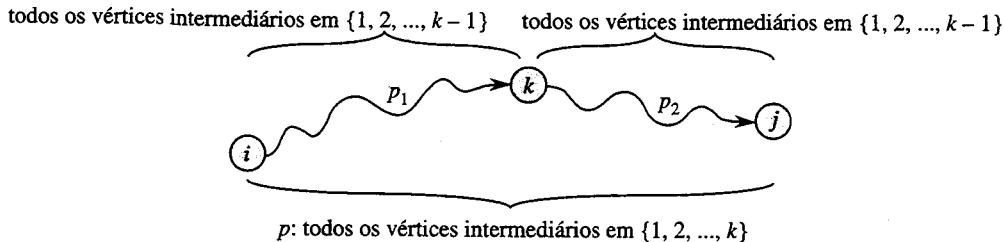
### A estrutura de um caminho mais curto

No algoritmo de Floyd-Warshall, usamos uma caracterização da estrutura de um caminho mais curto diferente daquela que usamos nos algoritmos de todos os pares baseados na multiplicação de matrizes. O algoritmo considera os vértices “intermediários” de um caminho mais curto, onde um vértice **intermediário** de um caminho simples  $p = \langle v_1, v_2, \dots, v_l \rangle$  é qualquer vértice de  $p$  diferente de  $v_1$  ou  $v_l$ , isto é, qualquer vértice no conjunto  $\{v_2, v_3, \dots, v_{l-1}\}$ .

O algoritmo de Floyd-Warshall se baseia na observação a seguir. Sejam  $V = \{1, 2, \dots, n\}$  os vértices de  $G$ , e considere um subconjunto  $\{1, 2, \dots, k\}$  de vértices para algum  $k$ . Para qualquer par de vértices  $i, j \in V$ , considere todos os caminhos desde  $i$  até  $j$  cujos vértices intermediários são todos traçados a partir de  $\{1, 2, \dots, k\}$ , e seja  $p$  um caminho de peso mínimo dentre eles. (O caminho  $p$  é simples.) O algoritmo de Floyd-Warshall explora um relacionamento entre o caminho  $p$  e caminhos mais curtos desde  $i$  até  $j$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k-1\}$ . O relacionamento depende do fato de  $k$  ser ou não um vértice intermediário do caminho  $p$ .

- Se  $k$  não é um vértice intermediário do caminho  $p$ , então todos os vértices intermediários do caminho  $p$  estão no conjunto  $\{1, 2, \dots, k-1\}$ . Desse modo, um caminho mais curto desde o vértice  $i$  até o vértice  $j$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k-1\}$  também é um caminho mais curto desde  $i$  até  $j$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k\}$ .

- Se  $k$  é um vértice intermediário do caminho  $p$ , então desmembramos  $p$  em  $i \xrightarrow{p_1} k \xrightarrow{p_2} j$ , como mostra a Figura 25.3. Pelo Lema 24.1,  $p_1$  é um caminho mais curto desde  $i$  até  $k$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k\}$ . Como o vértice  $k$  não é um vértice intermediário do caminho  $p_1$ , vemos que  $p_1$  é um caminho mais curto desde  $i$  até  $k$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k-1\}$ . De modo semelhante,  $p_2$  é um caminho mais curto desde o vértice  $k$  até o vértice  $j$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k-1\}$ .



**FIGURA 25.3** O caminho  $p$  é um caminho mais curto desde o vértice  $i$  até o vértice  $j$  e  $k$  é o vértice intermediário de  $p$  com numeração mais alta. O caminho  $p_1$ , a porção de caminho  $p$  desde o vértice  $i$  até o vértice  $k$ , tem todos os vértices intermediários no conjunto  $\{1, 2, \dots, k-1\}$ . O mesmo vale para o caminho  $p_2$  desde o vértice  $k$  até o vértice  $j$ .

**Uma solução recursiva para o problema de caminhos mais curtos de todos os pares**

Com base nas observações anteriores, definimos uma formulação recursiva de valores de caminhos mais curtos diferente daquela que definimos na Seção 25.1. Seja  $d_{ij}^{(k)}$  o peso de um caminho mais curto desde o vértice  $i$  até o vértice  $j$  para o qual todos os vértices intermediários estão no conjunto  $\{1, 2, \dots, k\}$ . Quando  $k = 0$ , um caminho desde o vértice  $i$  até o vértice  $j$  sem vértices intermediários com numeração mais alta que 0 não tem absolutamente nenhum vértice intermediário. Tal caminho tem no máximo uma aresta, e então  $d_{ij}^{(0)} = w_{ij}$ . Uma definição recursiva que segue a discussão anterior é dada por

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{se } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{ki}^{(k-1)}) & \text{se } k \geq 1 \end{cases} \quad (25.5)$$

Considerando-se que, para qualquer caminho, todos os vértices intermediários estão no conjunto  $\{1, 2, \dots, n\}$ , a matriz  $D^{(n)} = (d_{ij}^{(n)})$  fornece a resposta final:  $d_{ij}^{(n)} = \delta(i, j)$  para todo  $i, j \in V$ .

**Como calcular os pesos de caminhos mais curtos de baixo para cima**

Com base na recorrência (25.5), o procedimento de baixo para cima dado a seguir pode ser usado para calcular os valores  $d_{ij}^{(k)}$  em ordem de valores crescentes de  $k$ . Sua entrada é uma matriz  $W n \times n$  definida como na equação (25.1). O procedimento retorna a matriz  $D^{(n)}$  de pesos de caminhos mais curtos.

## FLOYD-WARSHALL(W)

1  $n \leftarrow \text{linbas}[W]$

2  $D(0) \leftarrow W$

3 for  $k \leftarrow 1$  to  $n$

4     do for  $i \leftarrow 1$  to  $n$

do for  $i \leftarrow 1$  to  $n$

$$d \leftarrow d^{(k)} \leftarrow \min(d^{(k-1)}, d^{(k-1)} + d^{(k-1)})$$

408 | 7 **return**  $D^{(n)}$

$$\begin{array}{l}
D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix} \\
\\
D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}
\end{array}$$

FIGURA 25.4 A seqüência de matrizes  $D^{(k)}$  e  $\Pi^{(k)}$  calculada pelo algoritmo de Floyd-Warshall para o grafo da Figura 25.1

A Figura 25.4 mostra as matrizes  $D^{(k)}$  calculadas pelo algoritmo de Floyd-Warshall para o grafo da Figura 25.1.

O tempo de execução do algoritmo de Floyd-Warshall é determinado pelos loops **for** triplicamente aninhados das linhas 3 a 6. Cada execução da linha 6 demora o tempo  $O(1)$ . Portanto, o algoritmo é executado no tempo  $\Theta(n^3)$ . Como no algoritmo final da Seção 25.1, o código é compacto, sem estruturas de dados elaboradas, e assim a constante oculta na notação de  $\Theta$  é pequena. Desse modo, o algoritmo de Floyd-Warshall é bastante prático até mesmo para grafos de entrada de dimensões moderadas.

## Como construir um caminho mais curto

Existe uma variedade de métodos diferentes para construir caminhos mais curtos no algoritmo de Floyd-Warshall. Um deles é calcular a matriz  $D$  de pesos de caminhos mais curtos, e depois construir a matriz predecessora  $\Pi$  a partir da matriz  $D$ . Esse método pode ser implementado para execução no tempo  $O(n^3)$  (Exercício 25.1-6). Dada a matriz predecessora  $\Pi$ , o procedimento PRINT-ALL-PAIRS-SHORTEST-PATH pode ser usado para imprimir os vértices em um dado caminho mais curto.

Podemos calcular a matriz predecessora  $\Pi$  “on-line”, exatamente como o algoritmo de Floyd-Warshall calcula as matrizes  $D^{(k)}$ . Especificamente, calculamos uma seqüência de matrizes  $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$ , onde  $\Pi = \Pi^{(n)}$ , e  $\pi_{ij}^{(k)}$  é definida como o predecessor do vértice  $j$  sobre um caminho mais curto desde o vértice  $i$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k\}$ .

Podemos dar uma formulação recursiva de  $\pi_{ij}^{(k)}$ . Quando  $k = 0$ , um caminho mais curto desde  $i$  até  $j$  não tem absolutamente nenhum vértice intermediário. Desse modo,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ ou } w_{ij} = \infty, \\ i & \text{se } i \neq j \text{ e } w_{ij} < \infty. \end{cases} \quad (25.6)$$

Para  $k \geq 1$ , se tomarmos o caminho  $i \rightsquigarrow k \rightsquigarrow j$ , onde  $k \neq j$ , então o predecessor de  $j$  que escolheremos será igual ao predecessor de  $j$  que escolhemos sobre um caminho mais curto desde  $k$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k-1\}$ . Caso contrário, escolheremos o mesmo predecessor de  $j$  que escolhemos sobre um caminho mais curto desde  $i$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k-1\}$ . Formalmente, para  $k \geq 1$ ,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \quad (25.7)$$

Deixamos a incorporação dos  $\Pi^{(k)}$  cálculos de matrizes ao procedimento de FLOYD-WARSHALL como o Exercício 25.2-3. A Figura 25.4 mostra a seqüência de  $\Pi^{(k)}$  matrizes que o algoritmo resultante calcula para o grafo da Figura 25.1. O exercício também lhe pede para realizar a tarefa mais difícil de provar que o subgrafo predecessor  $G_{\pi, i}$  é uma árvore de caminhos mais curtos com raiz  $i$ . Outra maneira de reconstruir caminhos mais curtos é dada como o Exercício 25.2-7.

## Fecho transitivo de um grafo orientado

Dado um grafo orientado  $G = (V, E)$  com o conjunto de vértices  $V = \{1, 2, \dots, n\}$ , desejamos descobrir se existe um caminho em  $G$  desde  $i$  até  $j$  para todos os pares de vértices  $i, j \in V$ . O **fecho transitivo** de  $G$  é definido como o grafo  $G^* = (V, E^*)$ , onde

$$E^* = \{(i, j) : \text{existe um caminho desde o vértice } i \text{ até o vértice } j \text{ em } G\}.$$

Um modo de calcular o fecho transitivo de um grafo no tempo  $\Theta(n^3)$  é atribuir o peso 1 a cada aresta de  $A$  e executar o algoritmo de Floyd-Warshall. Se existe um caminho desde o vértice  $i$  até o vértice  $j$ , obtemos  $d_{ij} < n$ . Caso contrário, obtemos  $d_{ij} = \infty$ .

Há outro modo semelhante de calcular o fecho transitivo de  $G$  no tempo  $\Theta(n^3)$  que pode poupar tempo e espaço na prática. Esse método envolve a substituição das operações aritméticas min e + no algoritmo de Floyd-Warshall pelas operações lógicas  $\vee$  (OU lógico) e  $\wedge$  (E lógico). Para  $i, j, k = 1, 2, \dots, n$ , definimos  $t_{ij}^{(k)}$  como 1 se existe um caminho no grafo  $G$  desde o vértice  $i$  até o vértice  $j$  com todos os vértices intermediários no conjunto  $\{1, 2, \dots, k\}$  e 0 em caso contrário. Construímos o fecho transitivo  $G^* = (V, E^*)$  inserindo a aresta  $(i, j)$  em  $E^*$  se e somente se  $t_{ij}^{(k)} = 1$ . Uma definição recursiva de  $t_{ij}^{(k)}$ , análoga à recorrência (25.5), é

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i, j) \notin E, \\ i & \text{se } i = j \text{ ou } (i, j) \in E, \end{cases}$$

e para  $k \geq 1$ ,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}). \quad (25.8)$$

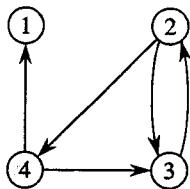
Como no algoritmo de Floyd-Warshall, calculamos as matrizes  $T^{(k)} = (t_{ij}^{(k)})$  em ordem de  $k$  crescente.

**TRANSITIVE-CLOSURE( $G$ )**

```

1  $n \leftarrow |V[G]|$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   do for  $j \leftarrow 1$  to  $n$ 
4     do if  $i = j$  or  $(i, j) \in E[G]$ 
5       then  $t_{ij}^{(0)} \leftarrow 1$ 
6       else  $t_{ij}^{(0)} \leftarrow 0$ 
7 for  $k \leftarrow 1$  to  $n$ 
8   do for  $i \leftarrow 1$  to  $n$ 
9     do for  $j \leftarrow 1$  to  $n$ 
10    do  $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ 
11 return  $T^{(n)}$ 
```

A Figura 25.5 mostra as matrizes  $T^{(k)}$  calculadas pelo procedimento TRANSITIVE-CLOSURE em um grafo de amostra. Como ocorre no algoritmo de Floyd-Warshall, o tempo de execução do procedimento TRANSITIVE-CLOSURE é  $\Theta(n^3)$ . Entretanto, em alguns computadores, operações lógicas sobre valores de um único bit são executadas mais rapidamente que operações aritméticas sobre palavras de dados inteiras. Além disso, como o algoritmo direto de fecho transitivo usa somente valores booleanos em lugar de valores inteiros, seu requisito de espaço é menor que o do algoritmo de Floyd-Warshall por um fator correspondente ao tamanho de armazenamento de uma palavra de computador.



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

FIGURA 25.5 Um grafo orientado e as matrizes  $T^{(k)}$  calculadas pelo algoritmo de fecho transitivo

## Exercícios

### 25.2-1

Execute o algoritmo de Floyd-Warshall sobre o grafo orientado ponderado da Figura 25.2. Mostre a matriz  $D^{(k)}$  que resulta de cada iteração do loop externo.

### 25.2-2

Mostre como calcular o fecho transitivo empregando a técnica da Seção 25.1.

### 25.2-3

Modifique o procedimento FLOYD-WARSHALL para incluir o cálculo das matrizes  $\Pi^{(k)}$  de acordo com as equações (25.6) e (25.7). Prove rigorosamente que, para todo  $i \in V$ , o subgrafo predecessor  $G_{\pi_i}$  é uma árvore de caminhos mais curtos com raiz  $i$ . (Sugestão: Para mostrar que  $G_{\pi_i}$  é acíclico, primeiro mostre que  $\pi_{ij}^{(k)} = l$  implica  $d_{ij}^{(k)} \geq d_{il}^{(k)} + w_{lj}$ , de acordo com a definição de  $\pi_{ij}^{(k)}$ . Depois, adapte a prova do Lema 24.16.)

### 25.2-4

Como foi apresentado, o algoritmo de Floyd-Warshall exige o espaço  $\Theta(n^3)$ , pois calculamos  $d_{ij}^{(k)}$  para  $i, j, k = 1, 2, \dots, n$ . Mostre que o procedimento a seguir, que simplesmente retira todos os sobrescritos, é correto e, desse modo, apenas o espaço  $\Theta(n^2)$  é necessário.

#### FLOYD-WARSHALL'( $W$ )

```
1  $n \leftarrow \text{linhas}[W]$ 
2  $D \leftarrow W$ 
3 for  $k \leftarrow 1$  to  $n$ 
4   do for  $i \leftarrow 1$  to  $n$ 
5     do for  $j \leftarrow 1$  to  $n$ 
6        $d_{ij} \leftarrow \min(d_{ij}, d_{ik} + d_{kj})$ 
7 return  $D$ 
```

### 25.2-5

Suponha que modificamos a maneira como a igualdade é tratada na equação (25.7):

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Essa definição alternativa da matriz predecessora  $\Pi$  é correta?

### 25.2-6

De que modo a saída do algoritmo de Floyd-Warshall pode ser usada para detectar a presença de um ciclo de peso negativo?

### 25.2-7

Outro modo de reconstruir caminhos mais curtos no algoritmo de Floyd-Warshall utiliza valores  $\phi_{ij}^{(k)}$  para  $i, j, k = 1, 2, \dots, n$ , onde  $\phi_{ij}^{(k)}$  é o vértice intermediário de numeração mais alta de um caminho mais curto desde  $i$  até  $j$  no qual todos os vértices intermediários estão no conjunto  $\{1, 2, \dots, k\}$ . Apresente uma formulação recursiva para  $\phi_{ij}^{(k)}$ , modifique o procedimento FLOYD-WARSHALL para calcular os valores de  $\phi_{ij}^{(k)}$  e reescreva o procedimento PRINT-ALL-PAIRS-SHORTEST-PATH para usar a matriz  $\Phi = (\phi_{ij}^{(k)})$  como uma entrada. De que maneira a matriz  $\Phi$  é semelhante à tabela  $s$  no problema de multiplicação de cadeias de matrizes da Seção 15.2?

### 25.2-8

Forneça um algoritmo de tempo  $O(VE)$  para calcular o fecho transitivo de um grafo orientado  $G$ .

### 25.2.9

Suponha que o fecho transitivo de um grafo acíclico orientado possa ser calculado no tempo  $f(|V|, |E|)$ , onde  $f$  é uma função monotonicamente crescente de  $|V|$  e  $|E|$ . Mostre que o tempo para calcular o fecho transitivo  $G^* = (V, E^*)$  de um grafo orientado geral  $G = (V, E)$  é  $f(|V|, |E|) + O(V + E^*)$ .

## 25.3 Algoritmo de Johnson para grafos esparsos

O algoritmo de Johnson encontra caminhos mais curtos entre todos os pares no tempo  $O(V^2 \lg V + VE)$ . Para grafos esparsos, ele é assintoticamente melhor que a elevação ao quadrado repetida de matrizes ou o algoritmo de Floyd-Warshall. O algoritmo retorna uma matriz de pesos de caminhos mais curtos para todos os pares de vértices ou informa que o grafo de entrada contém um ciclo de peso negativo. O algoritmo de Johnson usa como sub-rotinas o algoritmo de Dijkstra e o algoritmo de Bellman-Ford, que são descritos no Capítulo 24.

O algoritmo de Johnson emprega a técnica de *reponderar*, que funciona da maneira descrita a seguir. Se todos os pesos de arestas  $w$  em um grafo  $G = (V, E)$  são não negativos, podemos encontrar caminhos mais curtos entre todos os pares de vértices executando o algoritmo de Dijkstra uma vez a partir de cada vértice; com a fila de prioridade mínima do heap de Fibonacci, o tempo de execução desse algoritmo de todos os pares é  $O(V^2 \lg V + VE)$ . Se  $G$  tem arestas de peso negativo, mas nenhum ciclo de peso negativo, simplesmente calculamos um novo conjunto de pesos de arestas não negativos que nos permita utilizar o mesmo método. O novo conjunto de pesos de arestas  $\hat{w}$  deve satisfazer a duas propriedades importantes.

1. Para todos os pares de vértices  $u, v \in V$ , um caminho  $p$  é um caminho mais curto de  $u$  até  $v$  usando a função peso  $w$  se e somente se  $p$  também é um caminho mais curto desde  $u$  até  $v$  usando a função peso  $\hat{w}$ .
2. Para todas as arestas  $(u, v)$ , o novo peso  $\hat{w}(u, v)$  é não negativo.

Como veremos em breve, o pré-processamento de  $G$  para determinar a nova função peso  $\hat{w}$  pode ser executado no tempo  $O(VE)$ .

### Preservando caminhos mais curtos por reponderação

Como mostra o lema a seguir, é fácil apresentar uma reponderação das arestas que satisfaça à primeira propriedade descrita anteriormente. Utilizamos  $\delta$  para denotar pesos de caminhos mais curtos derivados da função peso  $w$  e  $\hat{\delta}$  para denotar pesos de caminhos mais curtos derivados da função peso  $\hat{w}$ .

#### *Lema 25.1 (A reponderação não altera caminhos mais curtos)*

Dado um grafo orientado ponderado  $G = (V, E)$  com a função peso  $w : E \rightarrow \mathbb{R}$ , seja  $b : V \rightarrow \mathbb{R}$  qualquer função que mapeia vértices para números reais. Para cada aresta  $(u, v) \in E$ , definimos

$$\hat{w}(u, v) = w(u, v) + b(u) - b(v) . \quad (25.9)$$

Seja  $p = \langle v_0, v_1, \dots, v_k \rangle$  qualquer caminho desde o vértice  $v_0$  até o vértice  $v_k$ . Então,  $p$  é um caminho mais curto de  $v_0$  até  $v_k$  com função peso  $w$  se e somente se ele é um caminho mais curto com função peso  $\hat{w}$ . Ou seja,  $w(p) = \delta(v_0, v_k)$  se e somente se  $\hat{w}(p) = \hat{\delta}(v_0, v_k)$ . Além disso,  $G$  tem um ciclo de peso negativo usando a função peso  $w$  se e somente se  $G$  tem um ciclo de peso negativo usando a função peso  $\hat{w}$ .

**Prova** Começamos mostrando que

$$\hat{w}(p) = w(p) = b(v_0) - b(v_k) . \quad (25.10)$$

Temos

$$\begin{aligned}
 \hat{w}(p) &= \sum_{i=1}^k \hat{w}(v_{i-1}, v_i) \\
 &= \sum_{i=1}^k (w(v_{i-1}, v_i) + b(v_{i-1}) - b(v_i)) \\
 &= \sum_{i=1}^k w(v_{i-1}, v_i) + b(v_0) - b(v_k) \quad (\text{porque a soma se encaixa}) \\
 &= w(p) + b(v_0) - b(v_k).
 \end{aligned}$$

Então, qualquer caminho  $p$  de  $v_0$  até  $v_k$  tem  $\hat{w}(p) = w(p) + b(v_0) - b(v_k)$ . Se um caminho de  $v_0$  até  $v_k$  é mais curto que outro usando função peso  $w$ , então ele também é mais curto usando  $\hat{w}$ . Desse modo,  $w(p) = \delta(v_0, v_k)$  se e somente se  $\hat{w}(p) = \hat{\delta}(v_0, v_k)$ .

Finalmente, mostramos que  $G$  tem um ciclo de peso negativo usando a função peso  $w$  se e somente se  $G$  tem um ciclo de peso negativo usando a função peso  $\hat{w}$ . Considere qualquer ciclo  $c = \langle v_0, v_1, \dots, v_k \rangle$  onde  $v_0 = v_k$ . Pela equação (25.10),

$$\begin{aligned}
 \hat{w}(c) &= w(c) + b(v_0) - b(v_k) \\
 &= w(c),
 \end{aligned}$$

e, desse modo,  $c$  tem peso negativo usando  $w$  se e somente se ele tem peso negativo usando  $\hat{w}$ . ■

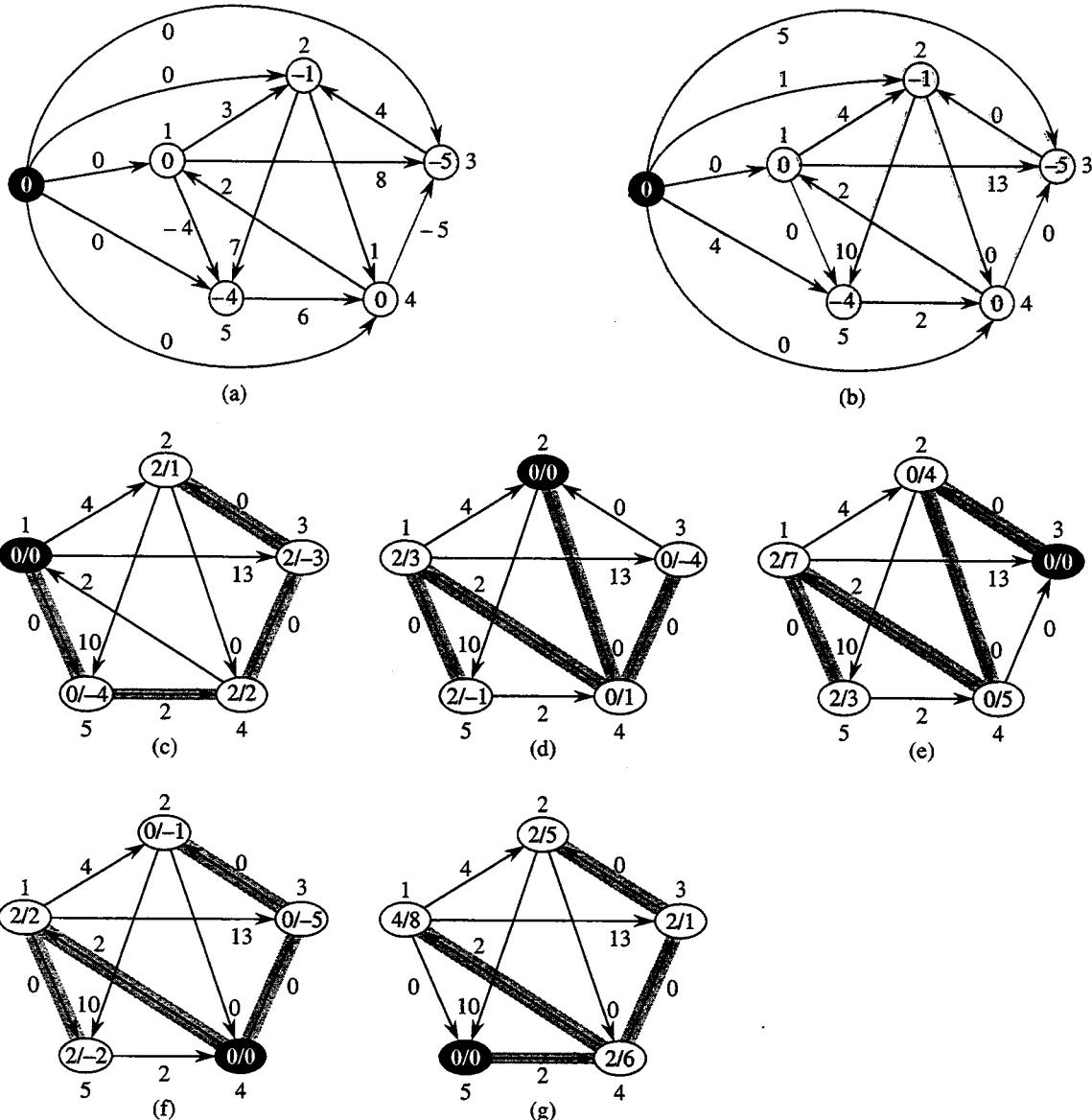
## Produção de pesos não negativos pela técnica de reponderar

Nossa próxima meta é assegurar que a segunda propriedade é válida: queremos que  $\hat{w}(u, v)$  seja não negativo para todas as arestas  $(u, v) \in E$ . Dado um grafo orientado ponderado  $G = (V, E)$  com função peso  $w : E \rightarrow \mathbb{R}$ , criamos um novo grafo  $G' = (V', E')$ , onde  $V' = V \cup \{s\}$  para algum novo vértice  $s \notin V$  e  $E' = E \cup \{s, v : v \in V\}$ . Estendemos a função peso  $w$  de tal modo que  $w(s, v) = 0$  para todo  $v \in V$ . Observe que, pelo fato de  $s$  não ter nenhuma aresta que entre nele, nenhum caminho mais curto em  $G'$ , além daqueles com origem  $s$ , contém  $s$ . Além disso,  $G'$  não tem nenhum ciclo de peso negativo se e somente se  $G$  não tem nenhum ciclo de peso negativo. A Figura 25.6(a) mostra o grafo  $G'$  correspondente ao grafo  $G$  da Figura 25.1.

Agora, suponha que  $G$  e  $G'$  não tenham ciclos de pesos negativos. Vamos definir  $b(v) = \delta(s, v)$  para todo  $v \in V$ . Pela desigualdade de triângulos (Lema 24.10), temos  $b(v) \leq b(u) + w(u, v)$  para todas as arestas  $(u, v) \in E'$ . Portanto, se definirmos os novos pesos  $\hat{w}$  de acordo com a equação (25.9), teremos  $\hat{w}(u, v) = w(u, v) + b(u) - b(v) \geq 0$ , e a segunda propriedade será satisfeita. A Figura 25.6(b) mostra o grafo  $G'$  da Figura 25.6(a) com arestas reponderadas.

## Como calcular caminhos mais curtos de todos os pares

O algoritmo de Johnson para calcular caminhos mais curtos de todos os pares emprega o algoritmo de Bellman-Ford (Seção 24.1) e o algoritmo de Dijkstra (Seção 24.3) como sub-rotinas. Ele supõe que as arestas estão armazenadas em listas de adjacências. O algoritmo retorna a matriz  $|V| \times |V|$  habitual  $D = d_{ij}$ , onde  $d_{ij} = \delta(i, j)$ , ou informa que o grafo de entrada contém um ciclo de peso negativo. Como é típico no caso de um algoritmo de caminhos curtos de todos os pares, supomos que os vértices são numerados de 1 até  $|V|$ .



**FIGURA 25.6** O algoritmo de caminhos mais curtos de todos os pares de Johnson executado sobre o grafo da Figura 25.1. (a) O grafo  $G'$  com a função peso original  $w$ . O novo vértice  $s$  é preto. Dentro de cada vértice  $v$  está  $b(v) = \delta(s, v)$ . (b) Cada aresta  $(u, v)$  é reponderada com a função peso  $\hat{w}(u, v) = w(u, v) + b(v)$ . (c)–(g) O resultado da execução do algoritmo de Dijkstra em cada vértice de  $G$  usando a função peso  $\hat{w}$ . Em cada parte, o vértice de origem  $u$  é preto, e as arestas sombreadas estão na árvore de caminhos mais curtos calculada pelo algoritmo. Dentro de cada vértice  $v$  estão os valores  $\hat{\delta}(u, v)$  e  $\delta(u, v)$ , separados por uma barra. O valor  $d_{uv} = \delta(u, v)$  é igual a  $\hat{\delta}(u, v) + b(v) - b(u)$

### JOHNSON( $G$ )

- 1 calcular  $G'$ , onde  $V[G'] = V[G] \cup \{s\}$ ,  
 $E[G'] = E[G] \cup \{(s, v) : v \in V[G]\}$  e  
 $w(s, v) = 0$  para todo  $v \in V[G]$
- 2 if BELLMAN-FORD( $G', w, s$ ) = FALSE  
 3 then imprimir “o grafo de entrada contém um ciclo de peso negativo”  
 4 else for cada vértice  $v \in V[G']$   
 5     do definir  $b(v)$  como o valor de  $\delta(s, v)$   
       calculado pelo algoritmo de Bellman-Ford  
 6     for cada aresta  $(u, v) \in E[G']$   
 7         do  $\hat{w}(u, v) \leftarrow w(u, v) + b(u) - b(v)$

```

8   for cada vértice  $u \in V[G]$ 
9     do executar DIJKSTRA( $G, \hat{w}, u$ ) para calcular  $\hat{\delta}(u, v)$  para todo  $v \in V[G]$ 
10    for cada vértice  $v \in V[G]$ 
11      do  $d_{uv} \leftarrow \hat{\delta}(u, v) + b(v) - b(u)$ 
12  return  $D$ 

```

Esse código simplesmente executa as ações que especificamos anteriormente. A linha 1 produz  $G'$ . A linha 2 executa o algoritmo de Bellman-Ford sobre  $G'$  com função peso  $w$  e vértice de origem  $s$ . Se  $G'$ , e consequentemente  $G$ , contém um ciclo de peso negativo, a linha 3 relata o problema. As linhas 4 a 11 supõem que  $G'$  não contém nenhum ciclo de peso negativo. As linhas 4 e 5 definem  $b(v)$  como o peso do caminho mais curto  $\delta(s, v)$  calculado pelo algoritmo de Bellman-Ford para todo  $v \in V'$ . As linhas 6 e 7 calculam os novos pesos  $\hat{w}$ . Para cada par de vértices  $u, v \in V$ , o loop **for** das linhas 8 a 11 calcula o peso do caminho mais curto  $\hat{\delta}(u, v)$ , chamando o algoritmo de Dijkstra uma vez para cada vértice em  $V$ . A linha 11 armazena na entrada de matriz  $d_{uv}$  o peso correto do caminho mais curto  $\delta(u, v)$ , calculado com o uso da equação (25.10). Finalmente, a linha 12 retorna a matriz  $D$  concluída. A Figura 25.6 mostra a execução do algoritmo de Johnson.

Se a fila de prioridade mínima no algoritmo de Dijkstra for implementada por um heap de Fibonacci, o tempo de execução do algoritmo de Johnson será  $O(V^2 \lg V + VE)$ . A implementação mais simples de heap mínimo binário resulta em um tempo de execução igual a  $O(VE \lg V)$ , que ainda é assintoticamente mais rápido que o algoritmo de Floyd-Warshall, se o grafo é esparsa.

## Exercícios

### 25.3-1

Use o algoritmo de Johnson para encontrar os caminhos mais curtos entre todos os pares de vértices no grafo da Figura 25.2. Mostre os valores de  $b$  e  $\hat{w}$  calculados pelo algoritmo.

### 25.3-2

Qual é o propósito de se adicionar o novo vértice  $s$  a  $V$ , produzindo  $V'$ ?

### 25.3-3

Suponha que  $w(u, v) \geq 0$  para todas as arestas  $(u, v) \in E$ . Qual é o relacionamento entre as funções peso  $w$  e  $\hat{w}$ ?

### 25.3-4

O professor Greenstreet afirma que existe um modo mais simples de responder arestas que o método usado no algoritmo de Johnson. Fazendo-se  $w^* = \min_{(u, v) \in E} \{w(u, v)\}$ , basta definir  $\hat{w}(u, v) = w(u, v) - w^*$  para todas as arestas  $(u, v) \in E$ . O que está errado no método de repondendo do professor?

### 25.3-5

Suponha que executamos o algoritmo de Johnson em um grafo orientado  $G$  com função peso  $w$ . Mostre que, se  $G$  contém um ciclo  $c$  de pesos 0, então  $\hat{w}(u, v) = 0$  para toda aresta  $(u, v)$  em  $c$ .

### 25.3-6

O professor Michener afirma que não há necessidade de criar um novo vértice de origem na linha 1 de JOHNSON. Ele afirma que, em vez disso, podemos simplesmente usar  $G' = G$  e fazer  $s$  ser qualquer vértice em  $V[G]$ . Forneça um exemplo de um grafo orientado ponderado  $G$  para o qual a incorporação da idéia do professor em JOHNSON provoca respostas incorretas. Em seguida, mostre que, se  $G$  é fortemente conectado (todo vértice é acessível a partir de todos os outros vértices), os resultados retornados por JOHNSON com a modificação do professor são corretos.

## Problemas

### 25-1 Fecho transitivo de um grafo dinâmico

Suponha que desejamos manter o fecho transitivo de um grafo orientado  $G = (V, E)$  à medida que inserimos arestas em  $E$ . Isto é, após cada aresta ter sido inserida, queremos atualizar o fecho transitivo das arestas inseridas até o momento. Suponha que o grafo  $G$  não tenha nenhuma aresta inicialmente, e que o fecho transitivo deva ser representado como uma matriz booleana.

- a. Mostre como o fecho transitivo  $G^* = (V, E^*)$  de um grafo  $G = (V, E)$  pode ser atualizado no tempo  $O(V^2)$  quando uma nova aresta é adicionada a  $G$ .
- b. Forneça um exemplo de um grafo  $G$  e uma aresta  $e$  tal que seja necessário o tempo  $\Omega(V^2)$  para atualizar o fecho transitivo após a inserção de  $e$  em  $G$ .
- c. Descreva um algoritmo eficiente para atualizar o fecho transitivo à medida que são inseridas arestas no grafo. Para qualquer seqüência de  $n$  inserções, seu algoritmo deve ser executado no tempo total  $\sum_{i=1}^n t_i = O(V^3)$ , onde  $t_i$  é o tempo para atualizar o fecho transitivo quando a  $i$ -ésima aresta é inserida. Prove que seu algoritmo alcança esse limite de tempo.

### 25-2 Caminhos mais curtos em grafos $\varepsilon$ -densos

Um grafo  $G = (V, E)$  é  $\varepsilon$ -denso se  $|E| = \Theta(V^{1+\varepsilon})$  para alguma constante  $\varepsilon$  no intervalo  $0 < \varepsilon \leq 1$ . Utilizando heaps mínimos  $d$ -ários (ver Problema 6-2) em algoritmos de caminhos mais curtos sobre grafos  $\varepsilon$ -densos, podemos alcançar os tempos de execução de algoritmos baseados em heaps de Fibonacci sem utilizar uma estrutura de dados tão complicada.

- a. Quais são os tempos de execução assintóticos para INSERT, EXTRACT-MIN e DECREASE-KEY, como uma função de  $d$  e o número  $n$  de elementos em um heap  $d$ -ário? Quais são esses tempos de execução se escolhemos  $d = \Theta(n^\alpha)$  para alguma constante  $0 < \alpha \leq 1$ ? Compare esses tempos de execução com os custos amortizados dessas operações para um heap de Fibonacci.
- b. Mostre como calcular caminhos mais curtos a partir de uma única origem sobre um grafo orientado  $\varepsilon$ -denso  $G = (V, E)$  sem arestas de peso negativo no tempo  $O(E)$ . (Sugestão: Escolha  $d$  como uma função de  $\varepsilon$ .)
- c. Mostre como resolver o problema de caminhos mais curtos de todos os pares sobre um grafo orientado  $\varepsilon$ -denso  $G = (V, E)$  sem arestas de peso negativo no tempo  $O(VE)$ .
- d. Mostre como resolver o problema de caminhos mais curtos de todos os pares no tempo  $O(VE)$  sobre um grafo orientado  $\varepsilon$ -denso  $G = (V, E)$  que pode ter arestas de peso negativo, mas não tem nenhum ciclo de peso negativo.

## Notas do capítulo

Lawler [196] tem uma boa descrição do problema de caminhos mais curtos de todos os pares, embora não analise soluções para grafos esparsos. Ele atribui o algoritmo de multiplicação de matrizes ao folclore. O algoritmo de Floyd-Warshall se deve a Floyd [89], que se baseou para criá-lo em um teorema de Warshall [308] que descreve como calcular o fecho transitivo de matrizes booleanas. O algoritmo de Johnson foi obtido a partir de [168].

Vários pesquisadores apresentaram algoritmos melhorados para calcular caminhos mais curtos via multiplicação de matrizes. Fredman [95] mostra que o problema de caminhos mais curtos de todos os pares pode ser resolvido com o uso de  $O(V^{5/2})$  comparações entre somas de pesos de arestas e obtém um algoritmo que funciona no tempo  $O(V^3 (\lg \lg V / \lg V)^{1/3})$ , ligeiramente melhor que o tempo de execução do algoritmo de Floyd-Warshall. Outra linha de pesquisa demonstra que os algoritmos para multiplicação rápida de matrizes (veja as notas do capítulo refe-

rentes ao Capítulo 28) podem ser aplicados ao problema de caminhos mais curtos de todos os pares. Seja  $O(n^\omega)$  o tempo de execução do algoritmo mais rápido para multiplicar matrizes  $n \times n$ ; atualmente,  $\omega < 2,376$  [70]. Galil e Margalit [105, 106] e Seidel [270] criaram algoritmos que resolvem o problema de caminhos mais curtos de todos os pares em grafos não orientados e não-ponderados no tempo  $(V^\omega p(V))$ , onde  $p(n)$  denota uma função específica com limite polilogarítmico em  $n$ . Em grafos densos, esses algoritmos são mais rápidos que o tempo  $O(VE)$  necessário para executar  $|V|$  pesquisas primeiro na extensão. Vários pesquisadores estenderam esses resultados para fornecer algoritmos que resolvem o problema de caminhos mais curtos de todos os pares em grafos não orientados nos quais os pesos de arestas são inteiros no intervalo  $\{1, 2, \dots, W\}$ . O algoritmo assintoticamente mais rápido entre eles, criado por Shoshan e Zwick [278], é executado no tempo  $O(WV^\omega p(VW))$ .

Karger, Koller e Phillips [170] e independentemente McGeoch [215] forneceram um limite de tempo que depende de  $E^*$ , o conjunto de arestas em  $E$  que participam de algum caminho mais curto. Dado um grafo com pesos de arestas não negativos, seus algoritmos são executados no tempo  $O(VE^* + V^2 \lg V)$  e são aperfeiçoamentos em relação aos tempos de execução  $|V|$  do algoritmo de Dijkstra quando  $|E^*| = o(E)$ .

Aho, Hopcroft e Ullman [5] definiram uma estrutura algébrica conhecida como um “semi-anel fechado”, que serve como uma estrutura geral para resolver problemas de caminhos em grafos orientados. Tanto o algoritmo de Floyd-Warshall quanto o algoritmo de fecho transitivo da Seção 25.2 são instanciações de um algoritmo de todos os pares baseado em semi-anéis fechados. Maggs e Plotkin [208] mostrou como encontrar árvores espalhadas mínimas usando um semi-anel fechado.

---

## *Capítulo 26*

### *Fluxo máximo*

Da mesma maneira que podemos elaborar um modelo de um mapa rodoviário como um grafo orientado com a finalidade de encontrar o menor caminho de um ponto até outro, também podemos interpretar um grafo orientado como um “fluxo em rede” e usá-lo para responder a perguntas sobre fluxos de materiais. Imagine um material percorrendo um sistema desde uma origem, onde o material é produzido, até um sorvedor, onde ele é consumido. A origem produz o material em alguma taxa fixa, e o sorvedor consome o material na mesma taxa. O “fluxo” do material em qualquer ponto no sistema é intuitivamente a taxa na qual o material se move. O fluxo em redes pode ser usado para modelar líquidos fluindo por tubos, peças por linhas de montagem, corrente por redes elétricas, informações por redes de comunicação e assim por diante.

Cada aresta orientada em um fluxo em rede pode ser imaginada como um canal para o material. Cada canal tem uma capacidade estabelecida, dada como uma taxa máxima na qual o material pode fluir pelo canal, como 200 galões de líquido por hora através de um tubo ou 20 ampères de corrente elétrica por um fio. Vértices são junções de canais e, além da origem e do sorvedor, o material flui pelos vértices, sem acumulação. Em outras palavras, a taxa na qual o material entra no vértice deve ser igual à taxa em que ele deixa o vértice. Chamamos essa propriedade “conservação do fluxo” e ela é igual à lei de corrente de Kirchhoff quando o material é a corrente elétrica.

No problema de fluxo máximo, desejamos calcular a maior taxa na qual o material pode ser enviado desde a origem até o sorvedor sem violar quaisquer restrições de capacidade. Esse é um dos problemas mais simples relacionados a fluxo em redes e, como veremos neste capítulo, ele pode ser resolvido por algoritmos eficientes. Além disso, as técnicas básicas usadas em algoritmos de fluxo máximo podem ser adaptadas para resolver outros problemas de fluxo em rede.

Este capítulo apresenta dois métodos gerais para resolver o problema do fluxo máximo. A Seção 26.1 formaliza as noções de fluxo em redes e fluxos, definindo formalmente o problema de fluxo máximo. A Seção 26.2 descreve o método clássico de Ford e Fulkerson para encontrar fluxos máximos. Uma aplicação desse método, consistindo em encontrar um emparelhamento máximo em um grafo bipartido não orientado, é dada na Seção 26.3. A Seção 26.4 apresenta o método push-relabel, que serve de base para muitos dos algoritmos mais rápidos para problemas de fluxo em rede. A Seção 26.5 abrange o algoritmo “relabel-to-front”, uma implementação particular do método push-relabel que é executado no tempo  $O(V^3)$ . Embora esse algoritmo não seja o algoritmo mais rápido conhecido, ele ilustra algumas das técnicas usadas nos algoritmos assintoticamente mais rápidos e tem uma eficiência razoável na prática.

## 26.1 Fluxo em redes

Nesta seção, daremos uma definição da teoria de grafos para o fluxo em redes, discutiremos suas propriedades e definiremos com exatidão o problema de fluxo máximo. Introduziremos também algumas regras úteis de notação.

### Fluxo em redes e fluxos

Um **fluxo em rede**  $G = (V, E)$  é um grafo orientado em que cada aresta  $(u, v) \in E$  tem uma **capacidade** não negativa  $c(u, v) \geq 0$ . Se  $(u, v) \notin E$ , supomos que  $c(u, v) = 0$ . Distinguimos dois vértices em um fluxo em rede: uma **origem**  $s$  e um sorvedor  $t$ . Por conveniência, supomos que cada vértice reside em algum caminho desde a origem até o sorvedor. Isto é, para todo vértice  $v \in V$ , existe um caminho  $s \sim v \sim t$ . Então, o grafo é conectado, e  $|E| \geq |V| - 1$ . A Figura 26.1 mostra um exemplo de um fluxo em rede.

Agora, estamos prontos para definir fluxos de modo mais formal. Seja  $G = (V, E)$  um fluxo em rede com uma função de capacidade  $c$ . Seja  $s$  a origem da rede, e seja  $t$  o sorvedor. Um **fluxo** em  $G$  é uma função de valor real  $f: V \times V \rightarrow \mathbb{R}$  que satisfaz às três propriedades seguintes:

**Restrição de capacidade:** Para todo  $u, v \in V$ , exigimos  $f(u, v) \leq c(u, v)$ .

**Anti-simetria oblíqua:** Para todo  $u, v \in V$ , exigimos  $f(u, v) = -f(v, u)$ .

**Conservação de fluxo:** Para todo  $u \in V - \{s, t\}$ , exigimos

$$\sum_{v \in V} f(u, v) = 0 .$$

A quantidade  $f(u, v)$ , que pode ser positiva, 0 ou negativa, é chamada **fluxo** do vértice  $u$  até o vértice  $v$ . O **valor** de um fluxo  $f$  é definido como

$$|f| = \sum_{v \in V} f(s, v) , \quad (26.1)$$

ou seja, o fluxo total que sai da origem. (Aqui, a notação  $|\cdot|$  denota valor de fluxo e não valor absoluto ou cardinalidade.) No **problema de fluxo máximo**, temos um fluxo em rede  $G$  com origem  $s$  e sorvedor  $t$ , e desejamos encontrar um fluxo de valor máximo.

Antes de ver um exemplo de um problema de fluxo em rede, vamos explorar brevemente as três propriedades de fluxo. A restrição de capacidade simplesmente afirma que o fluxo de um vértice até outro não deve exceder a capacidade dada. A anti-simetria é uma conveniência de notação que afirma que o fluxo de um vértice  $u$  até um vértice  $v$  é o valor negativo do fluxo no sentido inverso. A propriedade de conservação de fluxo afirma que o fluxo total para fora de um vértice que não seja a origem ou o sorvedor é 0. Por anti-simetria, podemos reescrever a propriedade de conservação de fluxo como

$$\sum_{u \in V} f(u, v) = 0$$

para todo  $v \in V - \{s, t\}$ . Isto é, o fluxo total em um vértice é 0.

Quando nem  $(u, v)$  nem  $(v, u)$  está em  $E$ , então não pode haver nenhum fluxo entre  $u$  e  $v$ , e  $f(u, v) = f(v, u) = 0$ . (O Exercício 26.1-1 lhe pede para provar formalmente essa propriedade.)

Nossa última observação a respeito das propriedades de fluxo lida com fluxos que são positivos. O **fluxo total positivo** que entra em um vértice  $v$  é definido por

$$\sum_{\substack{u \in V \\ f(u, v) > 0}} f(u, v) . \quad (26.2)$$

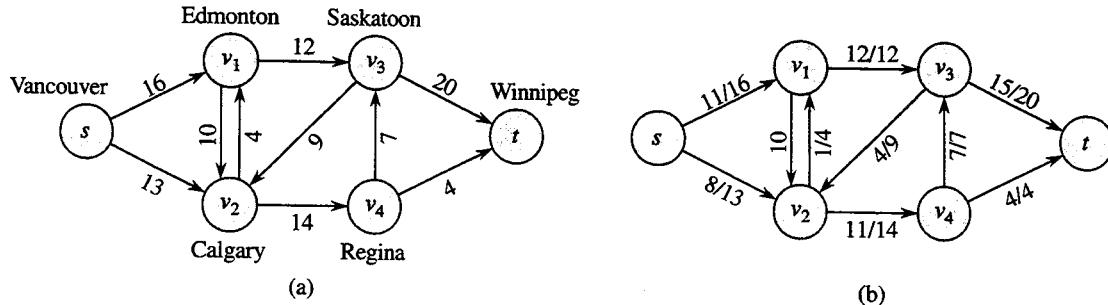


FIGURA 26.1 (a) Um fluxo em rede  $G = (V, E)$  para o problema do transporte da Lucky Puck Company. A fábrica de Vancouver é a origem  $s$ , e o armazém de Winnipeg é o sorvedor  $t$ . Os discos para hóquei são transportados por cidades intermediárias, mas apenas  $c(u, v)$  caixotes por dia podem ir da cidade  $u$  para a cidade  $v$ . Cada aresta é identificada com sua capacidade. (b) Um fluxo  $f$  em  $G$  com valor  $|f| = 19$ . São mostrados apenas fluxos positivos. Se  $f(u, v) > 0$ , a aresta  $(u, v)$  é identificada por  $f(u, v)/c(u, v)$ . (A notação de barra é usada apenas para separar o fluxo e a capacidade; ela não indica a operação de divisão.) Se  $f(u, v) \leq 0$ , a aresta  $(u, v)$  é identificada apenas por sua capacidade

O fluxo total positivo que sai de um vértice é definido de forma simétrica. Definimos o **fluxo total líquido** em um vértice como o fluxo total positivo que sai de um vértice menos o fluxo total positivo que entra em um vértice. Uma interpretação da propriedade de conservação de fluxo é que o fluxo total positivo que entra em um outro vértice que não a origem ou o sorvedor deve ser igual ao fluxo total positivo que sai desse vértice. Essa propriedade, de que o fluxo total líquido em um vértice deve ser igual 0, é referenciada com freqüência de modo informal como “o fluxo que entra é igual ao fluxo que sai”.

### Um exemplo de fluxo

Um fluxo em rede pode modelar o problema de transporte mostrado na Figura 26.1(a). A Lucky Puck Company tem uma fábrica (origem  $s$ ) em Vancouver que produz discos para hóquei e tem um armazém (sorvedor  $t$ ) em Winnipeg que os mantém em estoque. A Lucky Puck aluga espaço em caminhões de outra firma para transportar os discos da fábrica até o armazém. Pelo fato de os caminhões viajarem por rotas especificadas (arestas) entre cidades (vértices) e terem uma capacidade limitada, a Lucky Puck pode transportar no máximo  $c(u, v)$  caixotes por dia entre cada par de cidades  $u$  e  $v$  na Figura 26.1(a). A Lucky Puck não tem nenhum controle sobre essas rotas e capacidades, e assim não pode alterar o fluxo em rede mostrado na Figura 26.1(a). Sua meta é determinar o maior número  $p$  de caixotes por dia que podem ser transportados, e depois produzir essa quantidade, pois não há nenhum sentido em produzir mais discos do que é possível transportar para o armazém.

A Lucky Puck não está preocupada com o tempo que leva um determinado disco para ir da fábrica ao armazém; a empresa se preocupa apenas em fazer  $p$  caixotes por dia saírem da fábrica e  $p$  caixotes por dia chegarem ao armazém.

À primeira vista, parece apropriado modelar o “fluxo” de remessas com um fluxo nessa rede, porque o número de caixotes transportados por dia de uma cidade para outra está sujeito a uma restrição de capacidade. Além disso, a conservação de fluxo deve ser obedecida de modo que, em um estado constante, a taxa na qual os discos entram em uma cidade intermediária tem de ser igual à taxa em que eles saem da cidade. Caso contrário, os caixotes se acumulariam em cidades intermediárias.

Porém, há uma sutil diferença entre remessas e fluxos. A Lucky Puck pode transportar discos de Edmonton para Calgary, e também pode transportar discos de Calgary para Edmonton. Suponha que 8 caixotes sejam transportados por dia de Edmonton ( $v_1$  na Figura 26.1) para Calgary ( $v_2$ ) e 3 caixotes por dia sejam transportados de Calgary para Edmonton. Pode parecer natural representar essas remessas diretamente por fluxos, mas isso não é possível. A restrição de antisimetria exige que  $f(v_1, v_2) = -f(v_2, v_1)$ , mas é claro que esse não é o caso, se considerarmos  $f(v_1, v_2) = 8$  e  $f(v_2, v_1) = 3$ .

A Lucky Puck pode perceber que é inútil transportar 8 caixotes por dia de Edmonton para Calgary e 3 caixotes de Calgary para Edmonton, quando seria possível obter o mesmo efeito líquido transportando 5 caixotes de Edmonton para Calgary e 0 caixote de Calgary para Edmonton (e, presumivelmente, usar menos recursos no processo). Representamos esse último cenário com um fluxo: temos  $f(v_1, v_2) = 5$  e  $f(v_2, v_1) = -5$ . Na realidade, 3 dos 8 caixotes por dia de  $v_1$  para  $v_2$  são **cancelados** por 3 caixotes por dia de  $v_2$  para  $v_1$ .

Em geral, o cancelamento nos permite representar as remessas entre duas cidades por um fluxo que é positivo ao longo de no máximo uma das duas arestas entre os vértices correspondentes. Ou seja, qualquer situação na qual os discos são transportados em ambos os sentidos entre duas cidades pode ser transformada com o uso de cancelamento em uma situação equivalente, na qual os discos são transportados somente em um sentido: o sentido de fluxo positivo.

Dado um fluxo  $f$  que surgiu de, digamos, remessas físicas, não podemos reconstruir as remessas exatas. Se soubermos que  $f(u, v) = 5$ , esse fluxo pode ocorrer porque 5 unidades foram transportadas de  $u$  para  $v$ , ou porque 8 unidades foram transportadas de  $u$  para  $v$  e 3 unidades foram transportadas de  $v$  para  $u$ . Em geral, não nos preocuparemos com a forma como as remessas físicas reais estão configuradas; para qualquer par de vértices, só nos preocuparemos com a quantidade líquida que trafega entre eles. Se nos preocuparmos com as remessas subjacentes, então devemos usar um modelo diferente, um modelo que retenha informações sobre remessas em ambos os sentidos.

O cancelamento surgirá implicitamente nos algoritmos deste capítulo. Suponha que a aresta  $(u, v)$  tenha um valor de fluxo igual a  $f(u, v)$ . No curso de um algoritmo, podemos aumentar o fluxo na aresta  $(v, u)$  por alguma quantidade  $d$ . Matematicamente, essa operação deve diminuir  $f(u, v)$  por  $d$  e, conceitualmente, podemos pensar nessas  $d$  unidades como o cancelamento de  $d$  unidades de fluxo que já estão na aresta  $(u, v)$ .

## Redes com várias origens e vários sorvedores

Um problema de fluxo máximo pode ter várias origens e vários sorvedores, em vez de apenas uma unidade de cada. Por exemplo, a Lucky Puck Company poderia na realidade ter um conjunto de  $m$  fábricas  $\{s_1, s_2, \dots, s_m\}$  e um conjunto de  $n$  armazéns  $\{t_1, t_2, \dots, t_n\}$ , como mostra a Figura 26.2(a). Felizmente, esse problema não é mais difícil que o fluxo máximo comum.

Podemos reduzir o problema de determinar um fluxo máximo em uma rede com várias origens e vários sorvedores a um problema de fluxo máximo comum. A Figura 26.2(b) mostra como a rede de (a) pode ser convertida em um fluxo em rede comum com apenas uma única origem e um único sorvedor. Adicionamos uma **superorigem**  $s$  e acrescentamos a ela uma aresta orientada  $(s, s_i)$  com capacidade  $c(s, s_i) = \infty$  para cada  $i = 1, 2, \dots, m$ . Também criamos um novo **supersorvedor**  $t$  e acrescentamos a ele uma aresta orientada  $(t_i, t)$  com capacidade  $c(t_i, t) = \infty$  para cada  $i = 1, 2, \dots, n$ . Intuitivamente, qualquer fluxo na rede em (a) corresponde a um fluxo na rede em (b), e *vice-versa*. A origem única  $s$  simplesmente fornece tanto fluxo quanto desejado para as várias origens  $s_i$ , e o único sorvedor  $t$  consome igualmente tanto fluxo quanto necessário para os vários sorvedores  $t_i$ . O Exercício 26.1-3 lhe pede para provar formalmente que os dois problemas são equivalentes.

## Como utilizar fluxos

Vamos lidar com diversas funções (como  $f$ ) que utilizam como argumentos dois vértices em um fluxo em rede. Neste capítulo, utilizaremos uma **notação de somatório implícito** na qual qualquer argumento, ou ambos, pode ser um **conjunto** de vértices, com a interpretação de que o valor denotado é a soma de todos os modos possíveis de substituir os argumentos por seus membros. Por exemplo, se  $X$  e  $Y$  são conjuntos de vértices, então

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y).$$

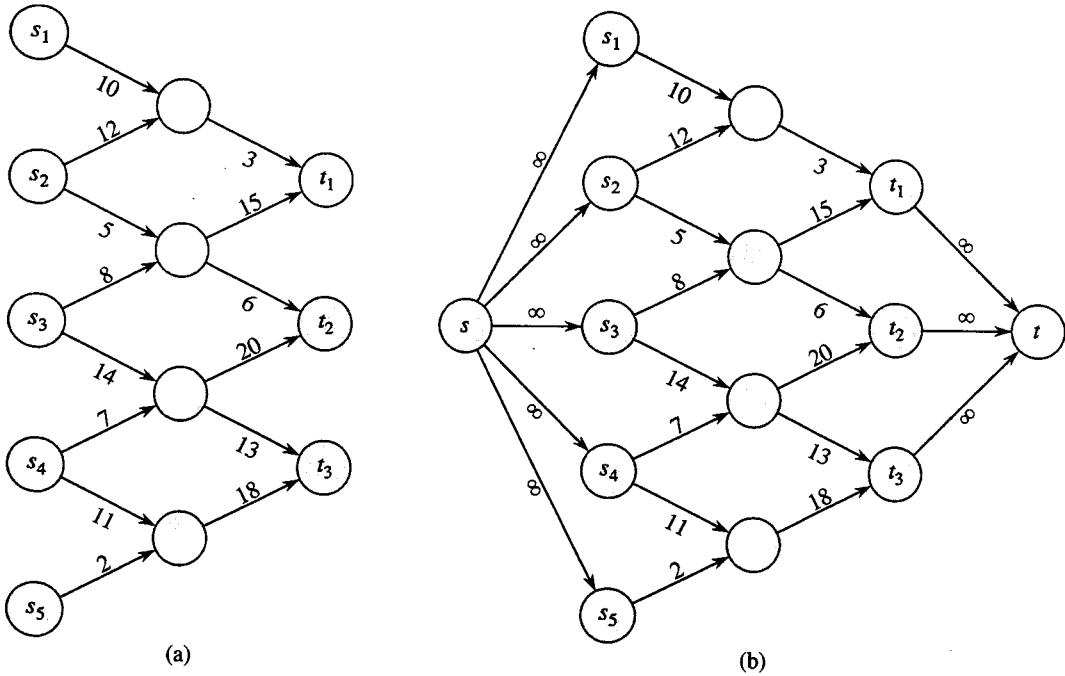


FIGURA 26.2 Convertendo um problema de fluxo máximo de várias origens e vários sorvedores em um problema com uma única origem e um único sorvedor. (a) Um fluxo em rede com cinco origens  $S = \{s_1, s_2, s_3, s_4, s_5\}$  e três sorvedores  $T = \{t_1, t_2, t_3\}$  (b) Um fluxo em rede equivalente de origem única e sorvedor único. Adicionamos uma superorigem  $s$  e uma aresta com capacidade infinita desde  $s$  até cada uma das várias origens. Adicionamos também um superdepósito  $t$  e uma aresta com capacidade infinita de cada um dos vários depósitos até  $t$

Portanto, a restrição de conservação do fluxo pode ser expressa como a condição de que  $f(u, V) = 0$  para todo  $u \in V - \{s, t\}$ . Além disso, por conveniência, em geral omitiremos as chaves de conjuntos quando elas tiverem de ser usadas de modo diferente na notação de somatório implícita. Por exemplo, na equação  $f(s, V - s) = f(s, V)$ , a expressão  $V - s$  significa o conjunto  $V - \{s\}$ .

A notação de conjuntos implícitos freqüentemente simplifica equações envolvendo fluxos. O lema a seguir, cuja prova fica para o Exercício 26.1-4, capta várias das identidades de ocorrência mais comum que envolvem fluxos e a notação de conjuntos implícitos.

### Lema 26.1

Seja  $G = (V, E)$  um fluxo em rede, e seja  $f$  um fluxo em  $G$ . Então, são válidas as igualdades a seguir:

1. Para todo  $X \subseteq V$ , temos  $f(X, X) = 0$ .
2. Para todo  $X, Y \subseteq V$ , temos  $f(X, Y) = -f(Y, X)$ .
3. Para todo  $X, Y, Z \subseteq V$  com  $X \cap Y = 0$ , temos as somas  $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$  e  $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$ . ■

Como um exemplo de emprego da notação de somatório implícito, podemos provar que o valor de um fluxo é o fluxo total no sorvedor; ou seja,

$$|f| = f(V, t) . \quad (26.3)$$

Intuitivamente, esperamos que essa propriedade seja válida. Por conservação do fluxo, todos os vértices que não a origem e o sorvedor têm quantidade igual de fluxo total positivo entrando e saindo. Por definição, a origem tem um fluxo total líquido maior que zero; isto é, sai da origem um fluxo positivo maior que o fluxo que entra na origem. Simetricamente, o sorvedor é o único vértice que pode ter um fluxo total líquido menor que 0; ou seja, entra no sorvedor um fluxo positivo maior que o fluxo que sai dele. Nossa prova formal é a seguinte:

$$\begin{aligned}
|f| &= f(s, V) && \text{(por definição)} \\
&= f(V, V) - f(V - s, V) && \text{(pelo Lema 26.1, parte (3))} \\
&= -f(V - s, V) && \text{(pelo Lema 26.1, parte (1))} \\
&= f(V, V - s) && \text{(pelo Lema 26.1, parte (2))} \\
&= f(V, t) + f(V, V - s - t) && \text{(pelo Lema 26.1, parte (3))} \\
&= f(V, t) && \text{(por conservação de fluxo)} .
\end{aligned}$$

Mais adiante neste capítulo, iremos generalizar esse resultado (Lema 26.5).

## Exercícios

### 26.1-1

Usando a definição de um fluxo prove que, se  $(u, v) \notin E$  e  $(v, u) \notin E$ , então  $f(u, v) = f(v, u) = 0$ .

### 26.1-2

Prove que, para qualquer vértice  $v$  diferente da origem ou do sorvedor, o fluxo total positivo que entra em  $v$  deve ser igual ao fluxo total positivo que sai de  $v$ .

### 26.1-3

Estenda as propriedades de fluxo e definições para o problema de várias origens e vários sorvedores. Mostre que qualquer fluxo de um fluxo em rede de várias origens e vários sorvedores corresponde a um fluxo de valor idêntico na rede de origem única e sorvedor único obtido pela adição de uma superorigem e um supersorvedor, e vice-versa.

### 26.1-4

Prove o Lema 26.1.

### 26.1-5

Para o fluxo em rede  $G = (V, E)$  e o fluxo  $f$  mostrados na Figura 26.1(b), encontre um par de subconjuntos  $X, Y \subseteq V$  para o qual  $f(X, Y) = -f(V - X, Y)$ . Em seguida, encontre um par de subconjuntos  $X, Y \subseteq V$  para o qual  $f(X, Y) \neq -f(V - X, Y)$ .

### 26.1-6

Dado um fluxo em rede  $G = (V, E)$ , sejam  $f_1$  e  $f_2$  funções de  $V \times V$  para  $\mathbf{R}$ . A **soma de fluxo**  $f_1 + f_2$  é a função de  $V \times V$  para  $\mathbf{R}$  definida por

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \quad (26.4)$$

para todo  $u, v \in V$ . Se  $f_1$  e  $f_2$  são fluxos em  $G$ , quais das três propriedades de fluxo devem ser satisfeitas pela soma de fluxo  $f_1$  e  $f_2$ , e quais ela poderia violar?

### 26.1-7

Seja  $f$  um fluxo em uma rede, e seja  $\alpha$  um número real. O **produto de fluxo escalar**  $\alpha$  é uma função de  $V \times V$  para  $\mathbf{R}$  definida por

$$(\alpha f)(u, v) = \alpha \cdot f(u, v)$$

Prove que os fluxos em uma rede formam um **conjunto convexo**. Isto é, mostre que, se  $f_1$  e  $f_2$  são fluxos, então  $\alpha f_1 + (1 - \alpha) f_2$  também é um fluxo para todo  $\alpha$  no intervalo  $0 \leq \alpha \leq 1$ .

### 26.1-8

Enuncie o problema de fluxo máximo como um problema de programação linear.

### 26.1-9

O professor Ademar tem dois filhos que, infelizmente, não gostam nem um pouco um do outro. O problema é tão grave que eles não só se recusam a caminhar até a escola juntos, mas de fato cada um se recusa a passar por qualquer quadra que o outro filho tenha percorrido no mesmo dia. Os filhos não têm nenhum problema com o fato de seus percursos se cruzarem em uma esquina. Felizmente, tanto a casa do professor quanto a escola estão em esquinas, mas além disso ele não está certo de que vai ser possível enviar ambos os filhos à mesma escola. O professor tem um mapa de sua cidade. Mostre como formular o problema de determinar se os dois filhos do professor podem freqüentar a mesma escola como um problema de fluxo máximo.

## 26.2 O método de Ford-Fulkerson

Esta seção apresenta o método de Ford-Fulkerson para resolver o problema de fluxo máximo. Ele é chamado um “método” em vez de um “algoritmo” porque engloba diversas implementações com diferentes tempos de execução. O método de Ford-Fulkerson depende de três idéias importantes que transcendem o método e são relevantes para muitos algoritmos de fluxo e problemas: redes residuais, caminhos em ampliação e cortes. Essas idéias são essenciais para o importante teorema de fluxo máximo e corte mínimo (Teorema 26.7), que caracteriza o valor de um fluxo máximo em termos de cortes do fluxo em rede. Encerramos esta seção apresentando uma implementação específica do método de Ford-Fulkerson e analisando seu tempo de execução.

O método de Ford-Fulkerson é iterativo. Começamos com  $f(u, v) = 0$  para todo  $u, v \in V$ , dando um fluxo inicial de valor 0. A cada iteração, aumentamos o valor do fluxo, encontrando um “caminho aumentante”, que podemos imaginar simplesmente como um caminho desde a origem  $s$  até o sorvedor  $t$  ao longo do qual podemos empurrar mais fluxo, e depois aumentar o fluxo ao longo desse caminho. Repetimos esse processo até não ser possível encontrar nenhum caminho aumentante. O teorema de fluxo máximo e corte mínimo mostrará que, no final, esse processo produz um fluxo máximo.

FORD-FULKERSON-METHOD( $G, s, t$ )

- 1 inicializar fluxo  $f$  como 0
- 2 **while** existir um caminho aumentante  $p$
- 3     **do** ampliar fluxo  $f$  ao longo de  $p$
- 4 **return**  $f$

## Redes residuais

Intuitivamente, dados um fluxo em rede e um fluxo, a rede residual consiste em arestas que podem admitir mais fluxo. De modo mais formal, suponha que temos um fluxo em rede  $G = (V, E)$  com origem  $s$  e sorvedor  $t$ . Seja  $f$  um fluxo em  $G$ , e considere um par de vértices  $u, v \in V$ . A quantidade de fluxo *adicional* que podemos empurrar desde  $u$  até  $v$  antes de exceder a capacidade  $c(u, v)$  é a **capacidade residual** de  $(u, v)$ , dada por

$$c_f(u, v) = c(u, v) - f(u, v). \quad (26.5)$$

Por exemplo, se  $c(u, v) = 16$  e  $f(u, v) = 11$ , podemos aumentar  $f(u, v)$  em  $c_f(u, v) = 5$  unidades antes de excedermos a restrição de capacidade sobre a aresta  $(u, v)$ . Quando o fluxo  $f(u, v)$  é negativo, a capacidade residual  $c_f(u, v)$  é maior que a capacidade  $c(u, v)$ . Por exemplo, se  $c(u, v) = 16$  e  $f(u, v) = -4$ , a capacidade residual  $c_f(u, v)$  é 20. Podemos interpretar essa situação da maneira descrita a seguir. Existe um fluxo de 4 unidades de  $v$  para  $u$ , que podemos cancelar empurrando um fluxo de 4 unidades de  $u$  para  $v$ . Em seguida, podemos empurrar outras 16 unidades de  $u$  para  $v$  antes de violar a restrição de capacidade sobre a aresta  $(u, v)$ . Desse modo, empurra-

mos 20 unidades de fluxo adicionais, começando com um fluxo  $f(u, v) = -4$ , antes de alcançar a restrição de capacidade.

Dados um fluxo em rede  $G = (V, E)$  e um fluxo  $f$ , a **rede residual** de  $G$  induzida por  $f$  é  $G_f = (V, E_f)$ , onde

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}.$$

Isto é, como mencionamos antes, cada aresta da rede residual, ou **aresta residual**, pode admitir um fluxo maior que 0. A Figura 26.3(a) repete o fluxo em rede  $G$  e o fluxo  $f$  da Figura 26.1(b), e a Figura 26.3(b) mostra a rede residual correspondente  $G_f$ .

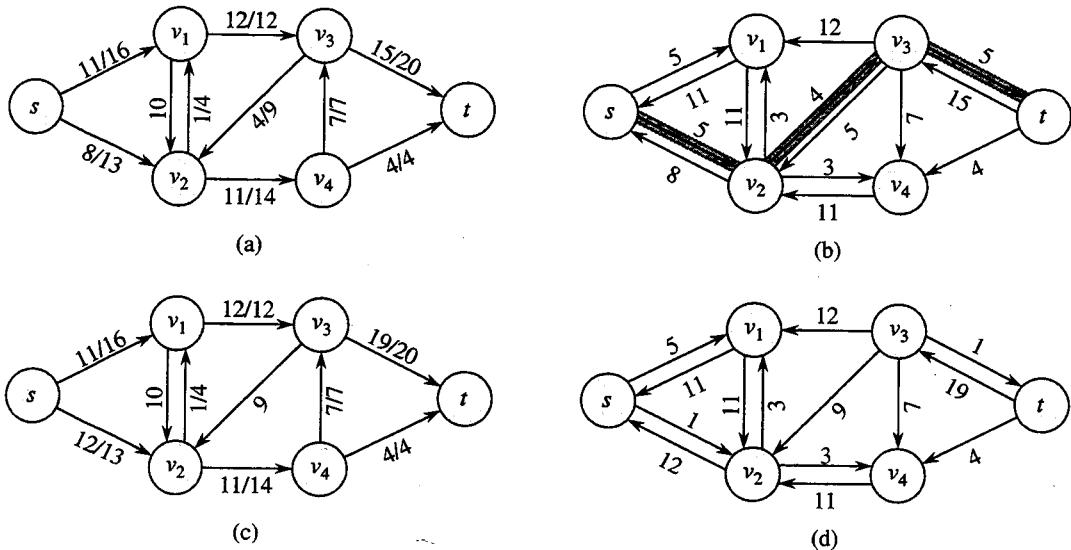


FIGURA 26.3 (a) O fluxo em rede  $G$  e o fluxo  $f$  da Figura 26.1(b). (b) A rede residual  $G_f$ , com o caminho aumentante  $p$  sombreado; sua capacidade residual é  $c_f(p) = c_f(v_2, v_3) = 4$ . (c) O fluxo em  $G$  que resulta da ampliação ao longo do caminho  $p$  por sua capacidade residual 4. (d) A rede residual induzida pelo fluxo em (c)

As arestas em  $E_f$  são arestas em  $E$  ou suas inversas. Se  $f(u, v) < c(u, v)$  para uma aresta  $(u, v) \in E$ , então  $c_f(u, v) = c(u, v) - f(u, v) > 0$  e  $(u, v) \in E_f$ . Se  $f(u, v) > 0$  para uma aresta  $(u, v) \in E$ , então  $f(v, u) < 0$ . Nesse caso,  $c_f(v, u) = c(v, u) - f(v, u) > 0$ , e então  $(v, u) \in E_f$ . Se nem  $(u, v)$  nem  $(v, u)$  aparecer na rede original, então  $c(u, v) = c(v, u) = 0$ ,  $f(u, v) = f(v, u) = 0$  (pelo Exercício 26.1-1) e  $c_f(u, v) = c_f(v, u) = 0$ . Concluímos que uma aresta  $(u, v)$  só pode aparecer em uma rede residual se pelo menos uma aresta entre  $(u, v)$  e  $(v, u)$  aparecer na rede original e, desse modo,

$$|E_f| \leq 2 |E|.$$

Observe que a rede residual  $G_f$  é ela própria um fluxo em rede com capacidades dadas por  $c_f$ . O lema a seguir mostra como um fluxo em uma rede residual se relaciona com um fluxo presente no fluxo em rede original.

### Lema 26.2

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$  e sorvedor  $t$ , e seja  $f$  um fluxo em  $G$ . Seja  $G_f$  a rede residual de  $G$  induzida por  $f$ , e seja  $f'$  um fluxo em  $G_f$ . Então, a soma de fluxo  $f + f'$  definida pela equação (26.4) é um fluxo em  $G$  com valor  $|f| + |f'| = |f| + |f'|$ .

**Prova** Devemos verificar se a anti-simetria, as restrições de capacidade e a conservação de fluxo são obedecidas. No caso da anti-simetria, observe que para todo  $u, v \in V$ , temos

$$\begin{aligned}
(f + f')(u, v) &= f(u, v) + f'(u, v) \\
&= -f(u, v) - f'(u, v) \\
&= -(f(u, v) + f'(u, v)) \\
&= -(f + f')(u, v).
\end{aligned}$$

No caso das restrições de capacidade, observe que  $f'(u, v) \leq c_f(u, v)$  para todo  $u, v \in V$ . Pela equação (26.5), temos então,

$$\begin{aligned}
(f + f')(u, v) &= f(u, v) + f'(u, v) \\
&\leq f(u, v) + c(c(u, v) - f'(u, v)) \\
&= c(u, v).
\end{aligned}$$

No caso da conservação de fluxo, observe que para todo  $u \in V - \{s, t\}$ , temos

$$\begin{aligned}
\sum_{v \in V} (f + f')(u, v) &= \sum_{v \in V} (f(u, v) + f'(u, v)) \\
&= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) \\
&= 0 + 0 \\
&= 0.
\end{aligned}$$

Finalmente, temos

$$\begin{aligned}
|f + f'| &= \sum_{v \in V} (f + f')(s, v) \\
&= \sum_{v \in V} (f(s, v) + f'(s, v)) \\
&= \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) \\
&= |f| + |f'|.
\end{aligned}$$

## Caminhos aumentantes

Dados um fluxo em rede  $G = (V, E)$  e um fluxo  $f$ , um **caminho aumentante**  $p$  é um caminho simples desde  $s$  até  $t$  na rede residual  $G_f$ . Pela definição de rede residual, cada aresta  $(u, v)$  em um caminho aumentante admite algum fluxo positivo adicional de  $u$  até  $v$  sem violar a restrição de capacidade sobre a aresta.

O caminho sombreado na Figura 26.3(b) é um caminho aumentante. Tratando a rede residual  $G_f$  na figura como um fluxo em rede, podemos transportar até 4 unidades de fluxo adicional por cada aresta desse caminho, sem violar uma restrição de capacidade, pois a menor capacidade residual nesse caminho é  $c_f(v_2, v_3) = 4$ . Chamamos a quantidade máxima pela qual podemos aumentar o fluxo em cada aresta de um caminho aumentante  $p$  de **capacidade residual** de  $p$ , dada por

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ está em } p\}.$$

O lema a seguir, cuja prova deixamos para o Exercício 26.2-7, torna mais preciso o argumento anterior.

### Lema 26.3

Seja  $G = (V, E)$  um fluxo em rede, seja  $f$  um fluxo em  $G$  e seja  $p$  um caminho aumentante em  $G_f$ . Defina uma função  $f_p : V \times V \rightarrow \mathbb{R}$  por

$$f_p(u, v) = \begin{cases} c_f & \text{se } (u, v) \text{ está em } p, \\ -c_f(p) & \text{se } (v, u) \text{ está em } p, \\ 0 & \text{em caso contrário.} \end{cases} \quad (26.6)$$

Então,  $f_p$  é um fluxo em  $G_f$  com valor  $|f_p| = c_f(p) > 0$ . ■

O corolário a seguir mostra que, se adicionarmos  $f_p$  a  $f$ , obteremos outro fluxo em  $G$  cujo valor está mais próximo do máximo. A Figura 26.3(c) mostra o resultado da adição de  $f_p$  na Figura 26.3(b) a  $f$  da Figura 26.3(a).

### Corolário 26.4

Seja  $G = (V, E)$  um fluxo em rede, seja  $f$  um fluxo em  $G$  e seja  $p$  um caminho aumentante em  $G_f$ . Seja  $f_p$  definido como na equação (26.6). Defina uma função  $f_p : V \times V \rightarrow \mathbb{R}$  por  $f' = f + f_p$ . Então,  $f'$  é um fluxo em  $G$  com valor  $|f'| = |f| + |f_p| > |f|$ .

**Prova** Imediata, a partir dos Lemas 26.2 e 26.3. ■

## Cortes de fluxo em redes

O método de Ford-Fulkerson aumenta repetidamente o fluxo ao longo de caminhos aumentantes até ser encontrado um fluxo máximo. O teorema de fluxo máximo e corte mínimo, que demonstraremos em breve, nos informa que um fluxo é máximo se e somente se sua rede residual não contém nenhum caminho aumentante. Entretanto, para provar esse teorema, devemos primeiro explorar a noção de um corte de um fluxo em rede.

Um **corte**  $(S, T)$  de um fluxo em rede  $G = (V, E)$  é uma partição de  $V$  em  $S$  e  $T = V - S$  tal que  $s \in S$  e  $t \in T$ . (Essa definição é semelhante à definição de “corte” que usamos para árvores espalhadas mínimas no Capítulo 23, exceto pelo fato de que aqui estamos cortando um grafo orientado, em vez de um grafo não orientado, e insistimos que  $s \in S$  e  $t \in T$ .) Se  $f$  é um fluxo, então o **fluxo líquido** pelo corte  $(S, T)$  é definido como  $f(S, T)$ . A **capacidade** do corte  $(S, T)$  é  $c(S, T)$ . Um **corte mínimo** de uma rede é um corte cuja capacidade é mínima dentre todos os cortes da rede.

A Figura 26.4 mostra o corte  $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$  no fluxo em rede da Figura 26.1(b). O fluxo líquido por esse corte é

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) &= 12 + (-4) + 11 \\ &= 19, \end{aligned}$$

e sua capacidade é

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26. \end{aligned}$$

Observe que o fluxo líquido por um corte pode incluir fluxos negativos entre vértices, mas que a capacidade de um corte é inteiramente composta de valores não negativos. Em outras palavras, o fluxo líquido por um corte  $(S, T)$  consiste em fluxos positivos em ambas os sentidos; o fluxo positivo de  $S$  para  $T$  é adicionado, enquanto o fluxo positivo de  $T$  para  $S$  é subtraído. Por outro lado, a capacidade de um corte  $(S, T)$  é calculada somente a partir de arestas que vão de  $S$  para  $T$ . Arestras que vão de  $T$  para  $S$  não são incluídas no cálculo de  $c(S, T)$ .

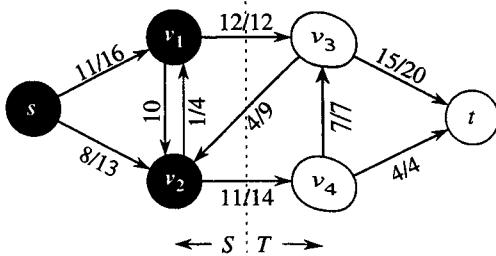


FIGURA 26.4 Um corte  $(S, T)$  no fluxo em rede da Figura 26.1(b), onde  $S = \{s, v_1, v_2\}$  e  $T = \{v_3, v_4, t\}$ . Os vértices em  $S$  são pretos, e os vértices em  $T$  são brancos. O fluxo líquido por  $(S, T)$  é  $f(S, T) = 19$ , e a capacidade é  $c(S, T) = 26$

O lema a seguir mostra que o fluxo líquido por qualquer corte é o mesmo, e que ele é igual ao valor do fluxo.

### Lema 26.5

Seja  $f$  um fluxo de um fluxo em rede  $G$  com origem  $s$  e sorvedor  $t$ , e seja  $(S, T)$  um corte de  $G$ . Então, o fluxo líquido por  $(S, T)$  é  $f(S, T) = |f|$ .

**Prova** Observando que  $f(S - s, V) = 0$  por conservação de fluxo, temos

$$\begin{aligned}
 f(S, T) &= f(S, V) - f(S, S) && \text{(pelo Lema 26.1, parte (3))} \\
 &= f(S, V) && \text{(pelo Lema 26.1, parte (1))} \\
 &= f(s, V) + f(S - s, V) && \text{(pelo Lema 26.1, parte (3))} \\
 &= f(s, V) && \text{(tendo em vista que } f(S - s, V) = 0) \\
 &= |f|. && \blacksquare
 \end{aligned}$$

Um corolário imediato para o Lema 26.5 é o resultado que fornecemos antes – a equação (26.3) – de que o valor de um fluxo é o fluxo total para o interior do sorvedor.

Outro corolário para o Lema 26.5 mostra como as capacidades de corte podem ser usadas para limitar o valor de um fluxo.

### Corolário 26.6

O valor de qualquer fluxo  $f$  em um fluxo em rede  $G$  é limitado a partir do anterior pela capacidade de qualquer corte de  $G$ .

**Prova** Seja  $(S, T)$  qualquer corte de  $G$  e seja  $f$  qualquer fluxo. Pelo Lema 26.5 e pelas restrições de capacidade,

$$|f| = f(S, T)$$

$$\begin{aligned}
 &= \sum_{u \in S} \sum_{v \in T} f(u, v) \\
 &\leq \sum_{u \in S} \sum_{v \in T} c(u, v) \\
 &= c(S, T)
 \end{aligned}$$

Uma consequência imediata do Corolário 26.6 é que o fluxo máximo em uma rede é limitado na parte superior pela capacidade de um corte mínimo da rede. O importante teorema de fluxo máximo e corte mínimo que enunciamos e provamos agora, informa que o valor de um fluxo de máximo é de fato igual à capacidade de um corte mínimo.

### **Teorema 26.7 (Teorema de fluxo máximo e corte mínimo)**

Se  $f$  é um fluxo em um fluxo em rede  $G = (V, E)$  com origem  $s$  e sorvedor  $t$ , então as condições a seguir são equivalentes:

1.  $f$  é um fluxo máximo em  $G$ .
2. A rede residual  $G_f$  não contém nenhum caminho aumentante.
3.  $|f| = c(S, T)$  para algum corte  $(S, T)$  de  $G$ .

**Prova** (1)  $\Rightarrow$  (2): Suponha para fins de contradição que  $f$  seja um fluxo máximo em  $G$ , mas que  $G_f$  tenha um caminho aumentante  $p$ . Então, pelo Corolário 26.4, a soma de fluxo  $f + f_p$ , onde  $f_p$  é dado pela equação (26.6), é um fluxo em  $G$  com valor estritamente maior que  $|f|$ , contradizendo a hipótese de que  $f$  é um fluxo máximo.

(2)  $\Rightarrow$  (3): Suponha que  $G_f$  não tenha nenhum caminho aumentante, ou seja, que  $G_f$  não contenha nenhum caminho desde  $s$  até  $t$ . Definimos

$$S = \{v \in V : \text{existe um caminho desde } s \text{ até } v \text{ em } G_f\}$$

e  $T = V - S$ . A partição  $(S, T)$  é um corte: temos  $s \in S$  trivialmente e  $t \notin S$ , porque não existe nenhum caminho desde  $s$  até  $t$  em  $G_f$ . Para cada par de vértices  $u$  e  $v$  tal que  $u \in S$  e  $v \in T$ , temos  $f(u, v) = c(u, v)$  pois, caso contrário,  $(u, v) \in E_f$ , e  $v$  estaria no conjunto  $S$ . Pelo Lema 26.5, portanto,  $|f| = f(S, T) = c(S, T)$ .

(3)  $\Rightarrow$  (1): Pelo Corolário 26.6,  $|f| \leq c(S, T)$  para todos os cortes  $(S, T)$ . Assim, a condição  $f = c(S, T)$  implica que  $f$  é um fluxo máximo. ■

## **O algoritmo básico de Ford-Fulkerson**

Em cada iteração do método de Ford-Fulkerson, encontramos *algum* caminho aumentante  $p$  e aumentamos o fluxo  $f$  em cada aresta de  $p$  pela capacidade residual  $c_f(p)$ . A implementação do método a seguir calcula o fluxo máximo em um grafo  $G = (V, E)$  atualizando o fluxo  $f[u, v]$  entre cada par  $u, v$  de vértices que estão conectados por uma aresta.<sup>1</sup> Se  $u$  e  $v$  não estão conectados por uma aresta em um ou outro sentido, supomos implicitamente que  $f[u, v] = 0$ . As capacidades  $c(u, v)$  são consideradas dadas juntamente com o grafo, e  $c(u, v) = 0$  se  $(u, v) \notin E$ . A capacidade residual  $c_f(u, v)$  é calculada de acordo com a fórmula (26.5). A expressão  $c_f(p)$  no código é na realidade apenas uma variável temporária que armazena a capacidade residual do caminho  $p$ .

**FORD-FULKERSON( $G, s, t$ )**

- 1 **for** cada aresta  $(u, v) \leftarrow E[G]$
- 2   **do**  $f[u, v] \leftarrow 0$
- 3     $f[v, u] \leftarrow 0$
- 4 **while** existir um caminho  $p$  de  $s$  até  $t$  na rede residual  $G_f$
- 5   **do**  $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \text{ está em } p\}$
- 6    **for** cada aresta  $(u, v)$  em  $p$
- 7     **do**  $f[u, v] \leftarrow f[u, v] + c_f(p)$
- 8       $f[v, u] \leftarrow -f[u, v]$

<sup>1</sup> Usamos colchetes quando tratamos um identificador – por exemplo,  $f$  – como um campo mutável, e usamos parênteses quando o tratamos como uma função.

O algoritmo FORD-FULKERSON é simplesmente uma expansão do pseudocódigo FORD-FULKERSON-METHOD dado antes. A Figura 26.6 mostra o resultado de cada iteração em um exemplo de execução. As linhas 1 a 3 inicializam o fluxo  $f$  como 0. O loop **while** das linhas 4 a 8 encontra repetidamente um caminho aumentante  $p$  em  $G_f$  e amplia o fluxo  $f$  ao longo de  $p$  pela capacidade residual  $c_f(p)$ . Quando não existe nenhum caminho aumentante, o fluxo  $f$  é um fluxo máximo.

## Análise de Ford-Fulkerson

O tempo de execução de FORD-FULKERSON depende de como o caminho aumentante  $p$  na linha 4 é determinado. Se ele for mal escolhido, o algoritmo poderá nem mesmo terminar: o valor do fluxo aumentará com ampliações sucessivas, mas não precisará sequer convergir para o valor de fluxo máximo.<sup>2</sup> Porém, se o caminho aumentante for escolhido pelo uso de uma busca em largura (que vimos na Seção 22.2), o algoritmo será executado em tempo polinomial. Contudo, antes de provar esse resultado, obtemos um limite simples para o caso no qual o caminho aumentante é escolhido arbitrariamente e todas as capacidades são valores inteiros.

Mais freqüentemente na prática, o problema de fluxo máximo surge com capacidades inteiros. Se as capacidades são números racionais, uma transformação de escala apropriada pode ser usada para torná-las todas inteiras. Sob essa hipótese, uma implementação direta de FORD-FULKERSON é executada no tempo  $O(E |f^*|)$ , onde  $f^*$  é o fluxo máximo encontrado pelo algoritmo. A análise é dada a seguir. As linhas 1 a 3 demoram o tempo  $\Theta(E)$ . O loop **while** das linhas 4 a 8 é executado no máximo  $|f^*|$  vezes, pois o valor de fluxo aumenta em pelo menos uma unidade em cada iteração.

O trabalho feito dentro do loop **while** pode se tornar eficiente se administrarmos de forma eficaz a estrutura de dados usada para implementar a rede  $G = (V, E)$ . Vamos supor que seja mantida uma estrutura de dados correspondente a um grafo orientado  $G' = (V, E')$ , onde  $E' = \{(u, v) : (u, v) \in E \text{ ou } (v, u) \in E\}$ . As arestas na rede  $G$  também são arestas em  $G'$ , e portanto é uma questão simples manter capacidades e fluxos nessa estrutura de dados. Dado um fluxo  $f$  em  $G$ , as arestas na rede residual  $G_f$  consistem em todas as arestas  $(u, v)$  de  $G'$  tais que  $c(u, v) - f[u, v] \neq 0$ . O tempo para encontrar um caminho em uma rede residual é portanto  $O(V + E') = O(E)$  se usamos a busca em profundidade ou a busca em largura. Cada iteração do loop **while** demora portanto o tempo  $O(E)$ , tornando o tempo de execução total de FORD-FULKERSON igual a  $O(E |f^*|)$ .

Quando as capacidades são inteiras e o valor de fluxo ótimo  $|f^*|$  é pequeno, o tempo de execução do algoritmo de Ford-Fulkerson é bom. A Figura 26.6(a) mostra um exemplo do que pode ocorrer em um fluxo em rede simples para o qual  $|f^*|$  é grande. Um fluxo máximo nessa rede tem valor 2.000.000: 1.000.000 unidades de fluxo atravessam o caminho  $s \rightarrow u \rightarrow t$ , outras 1.000.000 unidades atravessam o caminho  $s \rightarrow v \rightarrow t$ . Se o primeiro caminho aumentante encontrado por FORD-FULKERSON é  $s \rightarrow u \rightarrow v \rightarrow t$ , mostrado na Figura 26.6(a), o fluxo tem valor 1 após a primeira iteração. A rede residual resultante é mostrada na Figura 26.6(b). Se a segunda iteração encontra o caminho aumentante  $s \rightarrow v \rightarrow u \rightarrow t$ , como mostra a Figura 26.6(b), o fluxo tem então o valor 2. A Figura 26.6(c) mostra a rede residual resultante. Podemos continuar, escolhendo o caminho aumentante  $s \rightarrow u \rightarrow v \rightarrow t$  nas iterações de numeração ímpar, e o caminho aumentante  $s \rightarrow v \rightarrow u \rightarrow t$  nas iterações de numeração par. Executariammos ao todo 2.000.000 ampliações, aumentando o valor do fluxo em apenas uma unidade em cada ampliação.

---

<sup>2</sup> O método de Ford-Fulkerson poderia deixar de terminar apenas se as capacidades de arestas fossem números irracionais. Contudo, na prática, números irracionais não podem ser armazenados em computadores de precisão finita.

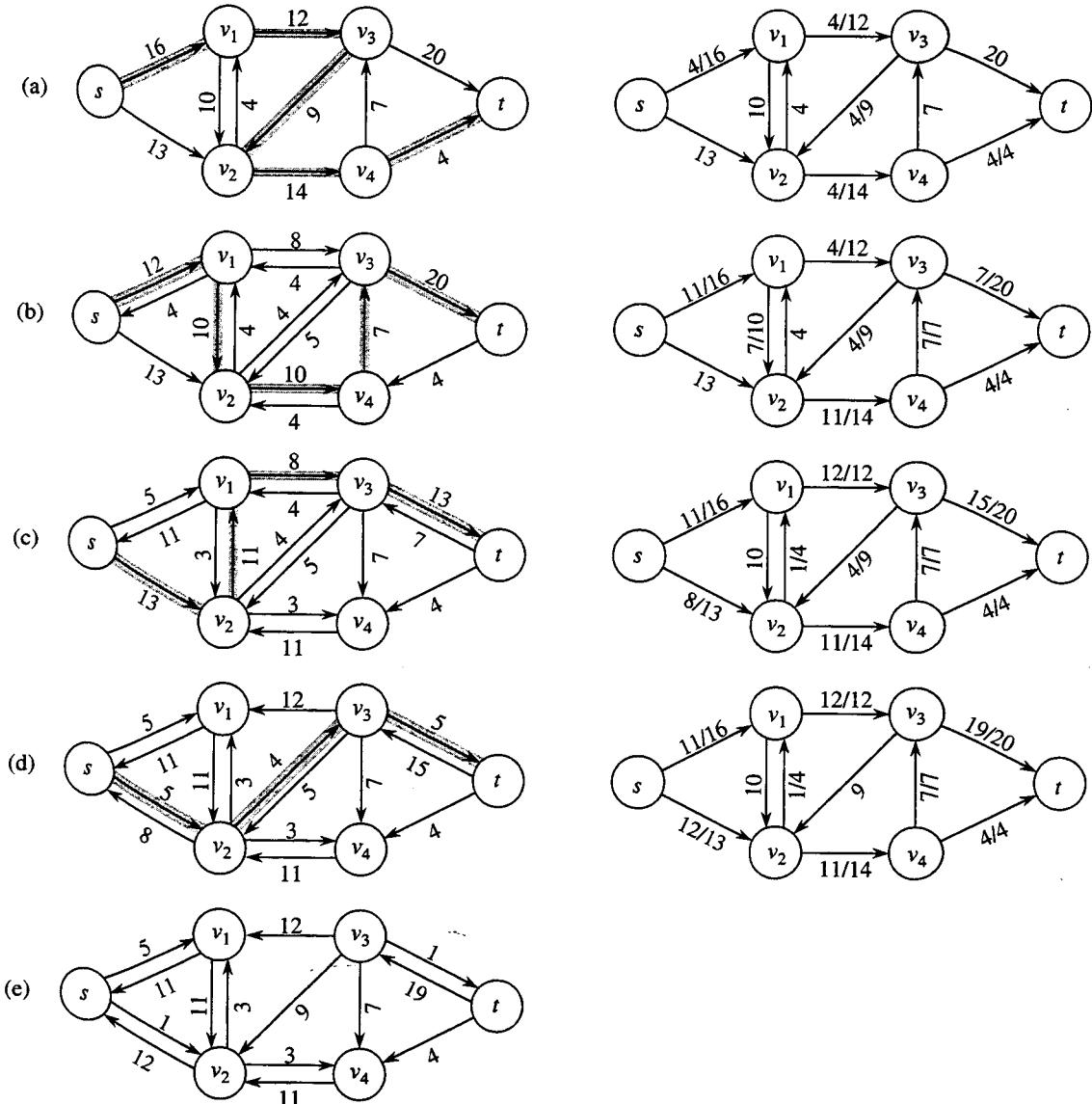


FIGURA 26.5 A execução do algoritmo básico de Ford-Fulkerson. (a)–(d) Iterações sucessivas do loop **while**. O lado esquerdo de cada parte mostra a rede residual  $G_f$  da linha 4 com um caminho aumentante sombreado  $p$ . O lado direito de cada parte mostra o novo fluxo  $f$  que resulta da adição de  $f_p$  a  $f$ . A rede residual em (a) é a rede de entrada  $G$ . (e) A rede residual no último teste do loop **while**. Ela não tem nenhum caminho aumentante, e o fluxo  $f$  mostrado em (d) é então um fluxo máximo

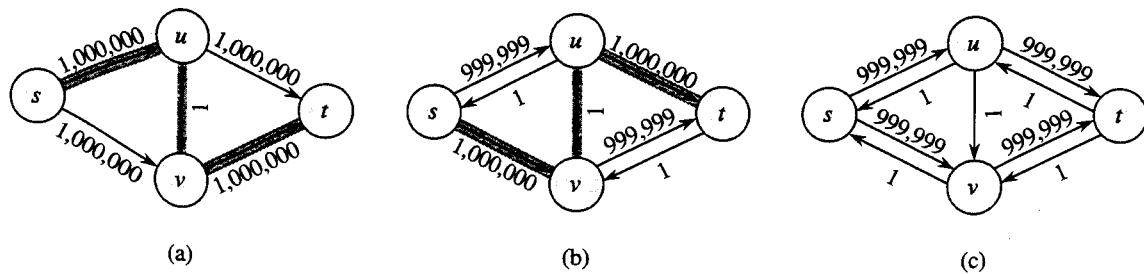


FIGURA 26.6 (a) Um fluxo em rede para o qual FORD-FULKERSON pode demorar o tempo  $\Theta(E |f^*|)$ , onde  $f^*$  é um fluxo máximo, mostrado aqui com  $|f^*| = 2.000.000$ . Um caminho aumentante com capacidade residual 1 é mostrado. (b) A rede residual resultante. Outro caminho aumentante com capacidade residual 1 é mostrado. (c) A rede residual resultante

## O algoritmo de Edmonds-Karp

O limite em FORD-FULKERSON pode ser melhorado se implementarmos o cálculo do caminho aumentante  $p$  na linha 4 com uma busca em largura, ou seja, se o caminho aumentante for um caminho *mais curto* desde  $s$  até  $t$  na rede residual, onde cada aresta tem distância (peso) unitária. Chamamos o método de Ford-Fulkerson assim implementado de **algoritmo de Edmonds-Karp**. Agora, vamos provar que o algoritmo de Edmonds-Karp é executado no tempo  $O(VE^2)$ .

A análise depende das distâncias até os vértices na rede residual  $G_f$ . O lema a seguir utiliza a notação  $\delta_f(u, v)$  para a distância do caminho mais curto desde  $u$  até  $v$  em  $G_f$ , onde cada aresta tem distância unitária.

### Lema 26.8

Se o algoritmo de Edmonds-Karp for executado em um fluxo em rede  $G = (V, E)$  com origem  $s$  e sorvedor  $t$ , então para todos os vértices  $v \in V - \{s, t\}$ , a distância do caminho mais curto  $v \in V - \{s, t\}$  na rede residual  $G_f$  aumentará monotonicamente com cada ampliação de fluxo.

**Prova** Suponha para fins de contradição que, para algum vértice  $v \in V - \{s, t\}$ , existe uma ampliação de fluxo que faz a distância do caminho mais curto de  $s$  até  $v$  diminuir, e então derivaremos uma contradição. Seja  $f$  o fluxo imediatamente antes da primeira ampliação que diminui alguma distância de caminho mais curto, e seja  $f'$  o fluxo imediatamente após. Seja  $v$  o vértice com  $\delta_f(s, v)$  mínimo, cuja distância foi diminuída pela ampliação, de forma que  $\delta_{f'}(s, v) < \delta_f(s, v)$ . Seja  $p = s \rightarrow u \rightarrow v$  um caminho mais curto desde  $s$  até  $v$  em  $G_f$ , de modo que  $(u, v) \in E_f$  e

$$\delta_{f'}(s, u) = \delta_f(s, v) - 1. \quad (26.7)$$

Devido ao modo como escolhemos  $v$ , sabemos que a etiqueta de distância do vértice  $u$  não diminuiu, isto é,

$$\delta_{f'}(s, u) \geq \delta_f(s, uv). \quad (26.8)$$

Afirmamos que  $(u, v) \notin E_f$ . Por quê? Se tivéssemos  $(u, v) \in E_f$ , então também teríamos

$$\begin{aligned} \delta_{f'}(s, v) &\leq \delta_{f'}(s, u) + 1 && (\text{pelo Lema 24.10, a desigualdade de triângulos}) \\ &\leq \delta_f(s, u) + 1 && (\text{pela desigualdade (26.8)}) \\ &= \delta_f(s, v) && (\text{pela equação (26.7)}), \end{aligned}$$

o que contradiz nossa hipótese de que  $\delta_{f'}(s, v) < \delta_f(s, v)$ .

Como podemos ter  $(u, v) \notin E_f$  e  $(u, v) \in E_{f'}$ ? A ampliação deve ter aumentado o fluxo de  $v$  para  $u$ . O algoritmo de Edmonds-Karp sempre aumenta o fluxo ao longo de caminhos mais curtos e, portanto, o caminho mais curto de  $s$  até  $u$  em  $G_f$  tem  $(v, u)$  como sua última aresta. Assim,

$$\begin{aligned} \delta_{f'}(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 && (\text{pela desigualdade (26.8)}) \\ &= \delta_{f'}(s, u) - 2 && (\text{pela equação (26.7)}), \end{aligned}$$

o que contradiz nossa hipótese de que  $\delta_{f'}(s, v) < \delta_f(s, v)$ . Concluímos que nossa hipótese de que tal vértice  $v$  existe é incorreta. ■

O próximo teorema limita o número de iterações do algoritmo de Edmonds-Karp.

### **Teorema 26.9**

Se o algoritmo de Edmonds-Karp é executado em um fluxo em rede  $G = (V, E)$  com origem  $s$  e sorvedor  $t$ , então o número total de ampliações de fluxo executadas pelo algoritmo é no máximo  $O(VE)$ .

**Prova** Dizemos que uma aresta  $(u, v)$  em uma rede residual  $G_f$  é **crítica** em um caminho aumentante  $p$  se a capacidade residual de  $p$  é a capacidade residual de  $(u, v)$ , isto é, se  $c_f(p) = c_f(u, v)$ . Depois de termos ampliado o fluxo ao longo de um caminho aumentante, qualquer aresta crítica no caminho desaparece da rede residual. Além disso, pelo menos uma aresta em qualquer caminho aumentante deve ser crítica. Mostraremos que cada uma das  $|E|$  arestas pode se tornar crítica no máximo  $|V|/2 - 1$  vezes.

Sejam  $u$  e  $v$  vértices em  $V$  que estão conectados por uma aresta em  $E$ . Tendo em vista que os caminhos em ampliação são caminhos mais curtos, quando  $(u, v)$  é crítica pela primeira vez, temos

$$\delta_f(s, v) = \delta_f(s, v) + 1.$$

Uma vez que o fluxo é ampliado, a aresta  $(u, v)$  desaparece da rede residual. Ela não poderá reaparecer mais tarde em outro caminho aumentante até o fluxo desde  $u$  até  $v$  ter diminuído, e isso só acontecerá se  $(v, u)$  aparecer em um caminho aumentante. Se  $f'$  for o fluxo em  $G$  quando esse evento ocorrer, então teremos

$$\delta_{f'}(s, v) = \delta_{f'}(s, v) + 1.$$

Tendo em vista que  $\delta_f(s, v) \leq \delta_{f'}(s, v)$  pelo Lema 26.8, temos

$$\begin{aligned} \delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\geq \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2. \end{aligned}$$

Conseqüentemente, a partir do momento em que  $(u, v)$  se torna crítica até o momento em que ela se torna crítica outra vez, a distância de  $u$  a partir da origem aumenta em pelo menos 2. A distância de  $u$  desde a origem é inicialmente pelo menos 0. Os vértices intermediários em um caminho mais curto de  $s$  até  $u$  não podem conter  $s$ ,  $u$  ou  $t$  (tendo em vista que  $(u, v)$  no caminho crítico implica que  $u \neq t$ ). Então, até  $u$  se tornar inacessível a partir da origem, se isso ocorrer, sua distância será no máximo  $|V| - 2$ . Portanto,  $(u, v)$  pode se tornar crítica no máximo  $(|V| - 2)/2 = |V|/2 - 1$  vezes. Tendo em vista que existem  $O(|E|)$  pares de vértices que podem ter entre eles uma aresta em um grafo residual, o número total de arestas críticas durante toda a execução do algoritmo de Edmonds-Karp será  $O(VE)$ . Cada caminho aumentante terá pelo menos uma aresta crítica, e consequentemente o teorema será válido. ■

Tendo em vista que cada iteração de FORD-FULKERSON pode ser implementada no tempo  $O(|E|)$  quando o caminho aumentante é encontrado por busca em largura, o tempo de execução total do algoritmo de Edmonds-Karp é  $O(VE^2)$ . Veremos que os algoritmos de push-relabel podem produzir limites ainda melhores. O algoritmo da Seção 26.4 fornece um método para alcançar um tempo de execução  $O(V^2E)$ , que forma a base para o algoritmo de tempo  $O(V^3)$  da Seção 26.5.

## Exercícios

### 26.2-1

Na Figura 26.1(b), qual é o fluxo pelo corte ( $\{s, v_2, v_4\}$ ,  $\{v_1, v_3, t\}$ )? Qual é a capacidade desse corte?

### 26.2-2

Mostre a execução do algoritmo de Edmonds-Karp sobre o fluxo em rede da Figura 26.1(a).

### 26.2-3

No exemplo da Figura 26.6, qual é o corte mínimo correspondente ao fluxo máximo mostrado? Dos caminhos em ampliação que aparecem no exemplo, quais são os dois que cancelam o fluxo?

### 26.2-4

Prove que, para qualquer par de vértices  $u$  e  $v$  e quaisquer funções de capacidade e de fluxo  $c$  e  $f$ , temos  $c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u)$ .

### 26.2-5

Lembre-se de que a construção na Seção 26.1 que converte um fluxo em rede de várias origens e vários sorvedores em uma rede de origem única e sorvedor único adiciona arestas com capacidade infinita. Prove que qualquer fluxo na rede resultante tem um valor finito se as arestas da rede original de várias origens e vários sorvedores têm capacidade finita.

### 26.2-6

Suponha que cada origem  $s_i$  em um problema de várias origens e vários sorvedores produza exatamente  $p_i$  unidades de fluxo, de modo que  $f(s_i, V) = p_i$ . Suponha também que cada sorvedor  $t_j$  consome exatamente  $q_j$  unidades, de forma que  $f(V, t_j) = q_j$ , onde  $\sum_i p_i = \sum_j q_j$ . Mostre como converter o problema de encontrar um fluxo  $f$  que obedeça a essas restrições adicionais no problema de encontrar um fluxo máximo em um fluxo em rede de origem única e sorvedor único.

### 26.2-7

Prove o Lema 26.3.

### 26.2-8

Mostre que um fluxo máximo em uma rede  $G = (V, E)$  sempre pode ser encontrado por uma sequência de no máximo  $|E|$  caminhos em ampliação. (Sugestão: Determine os caminhos depois de encontrar o fluxo máximo.)

### 26.2-9

A **conectividade de aresta** de um grafo não orientado é o número mínimo  $k$  de arestas que devem ser removidas para desconectar o grafo. Por exemplo, a conectividade de aresta de uma árvore é 1, e a conectividade de aresta de uma cadeia cílica de vértices é 2. Mostre como a conectividade de aresta de um grafo não orientado  $G = (V, E)$  pode ser determinada pela execução de um algoritmo de fluxo máximo sobre no máximo  $|V|$  fluxos em rede, cada um com  $O(V)$  vértices e  $O(E)$  arestas.

### 26.2-10

Suponha que um fluxo em rede  $G = (V, E)$  tem arestas simétricas, isto é,  $(u, v) \in E$  se e somente se  $(v, u) \in E$ . Mostre que o algoritmo de Edmonds-Karp termina depois de no máximo  $|V||E|/4$  iterações. (Sugestão: Para qualquer aresta  $(u, v)$ , considere a maneira como  $\delta(s, u)$  e  $\delta(v, t)$  mudam entre os momentos nos quais  $(u, v)$  é crítica.)

## 26.3 Emparelhamento bipartido máximo

Alguns problemas combinatórios podem ser modelados com facilidade como problemas de fluxo máximo. O problema de fluxo máximo de várias origens e vários sorvedores da Seção 26.1

nos deu um exemplo. Existem outros problemas combinatórios que, vistos superficialmente, têm pouca relação com fluxo em redes, mas que podem de fato ser reduzidos a um problema de fluxo máximo. Esta seção apresenta um desses problemas: encontrar um emparelhamento máximo em um grafo bipartido (ver Seção B.4). Para resolver esse problema, tiraremos proveito de uma propriedade de integralidade proporcionada pelo método de Ford-Fulkerson. Também veremos que o método de Ford-Fulkerson pode ser levado a resolver o problema de emparelhamento bipartido máximo sobre um grafo  $G = (V, E)$  no tempo  $O(VE)$ .

### O problema do emparelhamento bipartido máximo

Dado um grafo não orientado  $G = (V, E)$ , um **emparelhamento** é um subconjunto de arestas  $M \subseteq E$  tais que, para todos os vértices  $v \in V$ , no máximo uma aresta de  $M$  é incidente sobre  $v$ . Dizemos que um vértice  $v \in V$  é **correspondido** pelo emparelhamento  $M$  se alguma aresta em  $M$  é incidente sobre  $v$ ; caso contrário,  $v$  é **não correspondido**. Um **emparelhamento máximo** é um emparelhamento de cardinalidade máxima, ou seja, um emparelhamento  $M$  tal que, para qualquer emparelhamento  $M'$ , temos  $|M| \geq |M'|$ . Nesta seção, vamos restringir nossa atenção à localização de emparelhamentos máximos em grafos bipartidos. Supomos que o conjunto de vértices pode ser particionado em  $V = L \cup R$ , onde  $L$  e  $R$  são disjuntos e todas as arestas em  $E$  passam entre  $L$  e  $R$ . Vamos supor ainda que todo vértice em  $V$  tem pelo menos uma aresta incidente. A Figura 26.7 ilustra a noção de um emparelhamento.

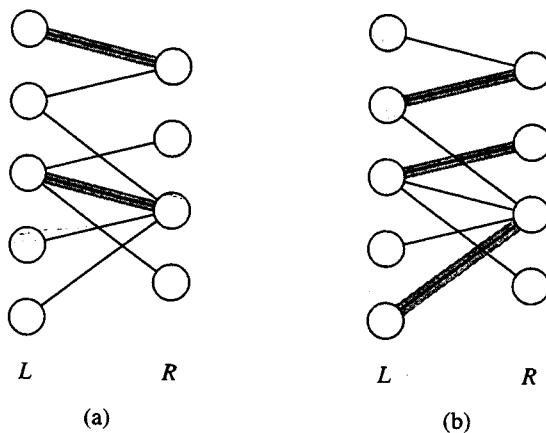


FIGURA 26.7 Um grafo bipartido  $G = (V, E)$  com partição de vértices  $V = L \cup R$ . (a) Um emparelhamento com cardinalidade 2. (b) Um emparelhamento máximo com cardinalidade 3

O problema de encontrar um emparelhamento máximo em um grafo bipartido tem muitas aplicações práticas. Como um exemplo, poderíamos considerar o emparelhamento de um conjunto  $L$  de máquinas com um conjunto  $R$  de tarefas a serem executadas simultaneamente. Consideramos a presença da aresta  $(u, v)$  em  $E$  com o significado de que uma determinada máquina  $u \in L$  é capaz de executar uma dada tarefa  $v \in R$ . Um emparelhamento máximo fornece trabalho para tantas máquinas quanto possível.

### Como encontrar um emparelhamento bipartido máximo

Podemos usar o método de Ford-Fulkerson para encontrar um emparelhamento máximo em um grafo bipartido não orientado  $G = (V, E)$  em tempo polinomial em  $|V|$  e  $|E|$ . O artifício é construir um fluxo em rede na qual os fluxos equivalem a emparelhamentos, como mostra a Figura 26.8. Definimos o **fluxo em rede correspondente**  $G' = (V', E')$  para o grafo bipartido  $G$  como a seguir. Sejam a origem  $s$  e o sorvedor  $t$  novos vértices não pertencentes a  $V$ , e seja  $V' = V \cup \{s, t\}$ . Se a partição de vértices de  $G$  é  $V = L \cup R$ , as arestas orientadas de  $G'$  são as arestas de  $E$ , orientadas de  $L$  para  $R$ , junto com  $|V|$  novas arestas:

$$\begin{aligned}
E' = & \{(s, u) : u \in L \\
& \cup \{(u, v) : u \in L, v \in R, e (u, v) \in E\} \\
& \cup \{(v, t) : v \in R\} .
\end{aligned}$$

Para completar a construção, atribuímos capacidade unitária a cada aresta em  $E'$ . Tendo em vista que cada vértice em  $V$  tem pelo menos uma aresta incidente,  $|E| \geq |V|/2$ . Desse modo,  $|E| \leq |E'| = |E| + |V| \leq 3|E|$ , e então  $|E'| = \Theta(|E|)$ .

O lema a seguir mostra que um emparelhamento em  $G$  equivale diretamente a um fluxo no fluxo em rede  $G'$  correspondente a  $G$ . Dizemos que um fluxo  $f$  em um fluxo em rede  $G = (V, E)$  é de **valor inteiro** se  $f(u, v)$  é um inteiro para todo  $(u, v) \in V \times V$ .

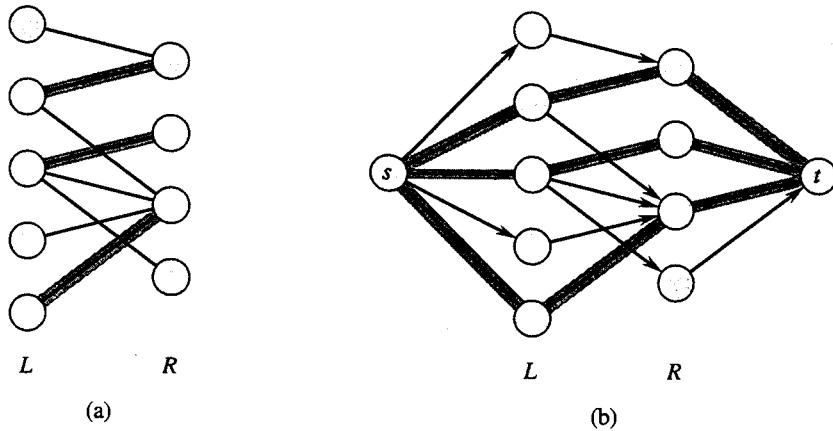


FIGURA 26.8 O fluxo em rede correspondente a um grafo bipartido. (a) O grafo bipartido  $G = (V, E)$  com partição de vértices  $V = L \cup R$  da Figura 26.7. Um emparelhamento máximo é mostrado por arestas sombreadas. (b) O fluxo em rede correspondente  $G'$  com um fluxo máximo mostrado. Cada aresta tem capacidade unitária. Arestas sombreadas têm um fluxo igual a 1, e todas as outras arestas não conduzem nenhum fluxo. As arestas sombreadas de  $L$  até  $R$  equivalem às arestas de um emparelhamento máximo do grafo bipartido

### Lema 26.10

Seja  $G = (V, E)$  um grafo bipartido com partição de vértices  $V = L \cup R$ , e seja  $G' = (V', E')$  seu fluxo em rede correspondente. Se  $M$  é um emparelhamento em  $G$ , então existe um fluxo de valor inteiro  $f$  em  $G'$  com valor  $|f| = |M|$ . De modo recíproco, se  $f$  é um fluxo de valor inteiro em  $G'$ , então existe um emparelhamento  $M$  em  $G$  com cardinalidade  $|M| = |f|$ .

**Prova** Primeiro mostramos que um emparelhamento  $M$  em  $G$  equivale a um fluxo  $f$  de valor inteiro em  $G'$ . Defina  $f$  como a seguir. Se  $(u, v) \in M$ , então  $f(s, u) = f(u, v) = f(v, t) = 1$  e  $f(u, s) = f(v, u) = f(t, v) = -1$ . Para todas as outras arestas  $f(u, v) = 0$ , definimos  $f(u, v) = 0$ . É simples verificar que  $f$  satisfaz à anti-simetria, as restrições de capacidade e a conservação de fluxo.

Intuitivamente, cada aresta  $(u, v) \in M$  corresponde a 1 unidade de fluxo em  $G'$  que percorre o caminho  $s \rightarrow u \rightarrow t$ . Além disso, os caminhos induzidos por arestas em  $M$  são de vértices disjuntos, exceto para  $s$  e  $t$ . O fluxo líquido pelo corte  $(L \cup \{s\}, R \cup \{t\})$  é igual a  $|M|$ ; portanto, pelo Lema 26.5, o valor do fluxo é  $|f| = |M|$ .

Para provar a recíproca, seja  $f$  um fluxo de valor inteiro em  $G'$  e seja

$$M = \{(u, v) : u \in L, v \in R, e f(u, v) > 0\} .$$

Cada vértice  $u \in L$  tem apenas uma aresta de entrada, ou seja  $(s, u)$ , e sua capacidade é 1. Desse modo, cada  $u \in L$  tem no máximo uma unidade de fluxo positivo entrando nele e, se entra

uma unidade de fluxo positivo, pela conservação de fluxo, uma unidade de fluxo positivo deve sair. Além disso, tendo em vista que  $f$  tem valor inteiro, para cada  $u \in L$ , a única unidade de fluxo pode entrar em no máximo uma aresta e pode sair no máximo de uma aresta. Desse modo, uma unidade de fluxo positivo entra em  $u$  se e somente se existe exatamente um vértice  $v \in R$  tal que  $f(u, v) = 1$ , e no máximo uma aresta que sai de cada  $u \in L$  transporta fluxo positivo. Um argumento simétrico pode ser criado para cada  $v \in R$ . O conjunto  $M$  é então um emparelhamento.

Para verificar que  $|M| = |f|$ , observe que, para todo vértice correspondente  $u \in L$ , temos  $f(s, u) = 1$ , e para toda aresta  $(u, v) \in E - M$ , temos  $f(u, v) = 0$ . Conseqüentemente,

$$\begin{aligned} |M| &= f(L, R) \\ &= f(L, V') - f(L, L) - f(L, s) - f(L, t) \quad (\text{pelo Lema 26.1}) . \end{aligned}$$

Podemos simplificar consideravelmente a expressão anterior. A conservação de fluxo implica que  $f(L, V') = 0$ ; o Lema 26.1 implica que  $f(L, L) = 0$ ; a anti-simetria implica que  $-f(L, s) = f(s, L)$ ; e, como não existe nenhuma aresta de  $L$  para  $t$ , temos  $f(L, t) = 0$ . Desse modo,

$$\begin{aligned} |M| &= f(L, s) \\ &= f(V', s) \quad (\text{pois todas as arestas que saem de } s \text{ vão para } L) \\ &= |f| \quad (\text{pela definição de } |f|) . \end{aligned}$$

Com base no Lema 26.10, gostaríamos de concluir que um emparelhamento máximo em um grafo bipartido  $G$  equivale a um fluxo máximo em seu fluxo em rede correspondente  $G'$ , e podemos então calcular uma comparação máxima em  $G$  executando um algoritmo de fluxo máximo em  $G'$ . O único senão nesse raciocínio é que o algoritmo de fluxo máximo poderia retornar um fluxo em  $G'$  para o qual algum  $f(u, v)$  não é um inteiro, ainda que o valor de fluxo  $|f|$  tenha de ser um inteiro. O teorema a seguir mostra que, se usarmos o método de Ford-Fulkerson, essa dificuldade não pode surgir.

### **Teorema 26.11 (Teorema de integralidade)**

Se a função de capacidade  $c$  utiliza apenas valores inteiros, então o fluxo máximo  $f$  produzido pelo método Ford-Fulkerson apresenta a propriedade de que  $|f|$  tem valor inteiro. Além disso, para todos os vértices  $u$  e  $v$ , o valor de  $f(u, v)$  é um inteiro.

**Prova** A prova é por indução sobre o número de iterações. Vamos deixá-la para o Exercício 26.3-2.

Agora, podemos provar o corolário a seguir para o Lema 26.10.

### **Corolário 26.12**

A cardinalidade de um emparelhamento máximo em um grafo bipartido  $G$  é igual ao valor de um fluxo máximo  $f$  em seu fluxo em rede correspondente  $G'$ .

**Prova** Usamos a nomenclatura do Lema 26.10. Suponha que  $M$  seja um emparelhamento máximo em  $G$  e que o fluxo correspondente  $f$  em  $G'$  não seja máximo. Então existe um fluxo máximo  $f'$  em  $G'$  tal que  $|f'| > |f|$ . Tendo em vista que as capacidades em  $G'$  são valores inteiros, pelo Teorema 26.1, podemos supor que  $f'$  tem valor inteiro. Desse modo,  $f'$  equivale a um emparelhamento  $M'$  em  $G$  com cardinalidade  $|M'| = |f'| > |f| = |M|$ , contradizendo nossa hipótese de que  $M$  é um emparelhamento máximo. De modo semelhante, podemos mostrar que, se  $f$  for um fluxo máximo em  $G'$ , seu emparelhamento equivalente será um emparelhamento máximo em  $G$ . ■

Portanto, dado um grafo bipartido não orientado  $G$ , podemos encontrar um emparelhamento máximo criando o fluxo em rede  $G'$ , executando o método de Ford-Fulkerson e obtendo direta-

mente um emparelhamento máximo  $M$  a emparelhamento partir do fluxo máximo de valor inteiro  $f$  encontrado. Tendo em vista que qualquer emparelhamento em um grafo bipartido tem cardinalidade no máximo  $\min(L, R) = O(V)$ , o valor do fluxo máximo em  $G'$  é  $O(V)$ . Podemos então encontrar um emparelhamento máximo em um grafo bipartido no tempo  $O(VE') = O(VE)$ , pois  $|E'| = \Theta(E)$ .

## Exercícios

### 26.3-1

Execute o algoritmo de Ford-Fulkerson sobre o fluxo em rede da Figura 26.8(b) e mostre a rede residual após cada ampliação de fluxo. Numere os vértices em  $L$  de cima para baixo, desde 1 até 5, e em  $R$  de cima para baixo desde 6 até 9. Para cada iteração, escolha o caminho aumentante que seja lexicograficamente menor.

### 26.3-2

Prove o Teorema 26.11.

### 26.3-3

Seja  $G = (V, E)$  um grafo bipartido com partição de vértices  $V = L \cup R$ , e seja  $G'$  seu fluxo em rede correspondente. Forneça um bom limite superior sobre o comprimento de qualquer caminho aumentante encontrado em  $G'$  durante a execução de FORD-FULKERSON.

### 26.3-4 \*

Um **emparelhamento perfeito** é um emparelhamento na qual todo vértice corresponde. Seja  $G = (V, E)$  um grafo bipartido não orientado com partição de vértices  $V = L \cup R$ , onde  $|L| = |R|$ . Para qualquer  $X \subseteq V$ , defina a **vizinhança** de  $X$  como

$$N(X) = \{y \in V : (x, y) \in E \text{ para algum } x \in X\},$$

isto é, o conjunto de vértices adjacentes a algum membro de  $X$ . Prove o **teorema de Hall**: existe um emparelhamento perfeito em  $G$  se e somente se  $|A| \leq |N(A)|$  para todo subconjunto  $A \subseteq L$ .

### 26.3-5 \*

Dizemos que um grafo bipartido  $G = (V, E)$ , onde  $V = L \cup R$  é  **$d$ -regular** se todo vértice  $v \in V$  tem grau exatamente igual a  $d$ . Todo grafo bipartido  $d$ -regular tem  $|L| = |R|$ . Prove que todo grafo bipartido  $d$ -regular tem um emparelhamento de cardinalidade  $|L|$ , demonstrando que um corte mínimo do fluxo em rede correspondente tem capacidade  $|L|$ .

## ★ 26.4 Algoritmos de push-relabel

Nesta seção, apresentamos a abordagem “push-relabel” para calcular fluxos máximos. Muitos dos algoritmos de fluxo máximo assintoticamente mais rápidos conhecidos até hoje são algoritmos push-relabel, e as mais rápidas implementações reais de algoritmos de fluxo máximo se baseiam no método de push-relabel. Outros problemas de fluxo, como o problema do fluxo de custo mínimo, podem ser resolvidos de forma eficiente por métodos de push-relabel. Esta seção introduz o algoritmo de fluxo máximo “genérico” de Goldberg, o qual tem uma implementação simples que é executada no tempo  $O(V^2E)$ , melhorando assim o limite de  $O(VE^2)$  do algoritmo de Edmonds-Karp. A Seção 26.5 aprimora o algoritmo genérico para obter outro algoritmo de push-relabel que é executado no tempo  $O(V^3)$ .

Os algoritmos de push-relabel funcionam de uma maneira mais localizada que o método de Ford-Fulkerson. Em lugar de examinar toda a rede residual até encontrar um caminho aumentante, os algoritmos de push-relabel funcionam sobre um vértice de cada vez, examinando apenas os vizinhos do vértice na rede residual. Além disso, diferente do método de Ford-Fulkerson,

os algoritmos de push-relabel não mantêm a propriedade de conservação de fluxo ao longo de toda a sua execução. Entretanto, eles mantêm um **pré-fluxo**, que é uma função  $f: V \times V \rightarrow \mathbb{R}$  que satisfaz à simetria  $f(V, u) \geq 0$  para todos os vértices  $u \in V - \{s\}$ . Chamamos essa quantidade **fluxo em excesso** para  $u$ , dado por

$$e(u) = f(V, u). \quad (26.9)$$

Dizemos que um vértice  $u \in V - \{s, t\}$  está **transbordando** se  $e(u) > 0$ .

Iniciaremos esta seção descrevendo a intuição por trás do método de push-relabel. Em seguida, investigaremos as duas operações empregadas pelo método: “empurrar” um pré-fluxo e “suspenso” um vértice. Finalmente, apresentaremos um algoritmo genérico de push-relabel e analisaremos sua correção e seu tempo de execução.

## Intuição

A intuição por trás do método push-relabel provavelmente é mais bem compreendida em termos de fluxos de fluidos: consideramos um fluxo em rede  $G = (V, E)$  um sistema de tubos interconectados de capacidades dadas. Aplicando essa analogia ao método de Ford-Fulkerson, podemos dizer que cada caminho aumentante na rede ocasiona uma série adicional de fluido, sem pontos de desvios, fluindo desde a origem até o sorvedor. O método de Ford-Fulkerson adiciona iterativamente mais séries de fluxo até não ser mais possível adicioná-los.

O algoritmo genérico de push-relabel tem uma intuição bastante diferente. Como antes, arestas orientadas correspondem a tubos. Vértices, que são junções de tubos, têm duas propriedades interessantes. Primeiro, para acomodar o fluxo em excesso, cada vértice tem um tubo de saída de fluxo que conduz a um reservatório arbitrariamente grande e que pode acumular fluido. Em segundo lugar, cada vértice, seu reservatório e todas as suas conexões de tubos estão em uma plataforma cuja altura aumenta à medida que o algoritmo progride.

As alturas dos vértices determinam de que modo o fluxo é empurrado: só empurramos o fluxo em declive, isto é, de um vértice mais alto para um vértice mais baixo. O fluxo de um vértice mais baixo para um vértice mais alto pode ser positivo, mas as operações que empurram o fluxo só o empurram em declive. A altura da origem é fixada em  $|V|$ , e a altura do sorvedor é fixada em 0. As alturas de todos os outros vértices começam em 0 e aumentam com o tempo. Primeiro, o algoritmo envia o máximo fluxo possível em declive a partir da origem em direção ao sorvedor. A quantidade enviada é exatamente o suficiente para preencher completamente cada tubo de saída a partir da origem; ou seja, ele envia a capacidade do corte  $(s, V - s)$ . Quando o fluxo entra primeiro em um vértice intermediário, ele se junta ao conteúdo do reservatório do vértice. De lá, ele é mais tarde empurrado em declive.

Eventualmente, pode acontecer de os únicos tubos que saem de um vértice  $u$  e ainda não estão saturados com fluxo se conectem a vértices que se encontram no mesmo nível que  $u$  ou que estejam acima de  $u$ . Nesse caso, para livrar um vértice transbordante  $u$  de seu excesso de fluxo, devemos aumentar sua altura – uma operação chamada “elevar” o vértice  $u$ . A altura desse vértice é aumentada até uma unidade a mais que a altura do mais baixo de seus vizinhos para o qual ele tenha um tubo não-saturado. Então, depois que um vértice é elevado, existe pelo menos um tubo de saída pelo qual é possível empurrar mais fluxo.

Em um determinado momento, todo o fluxo que tem a possibilidade de percorrer o caminho até o sorvedor já chegou lá. Não é possível chegar mais nenhum fluxo, porque os tubos obedecem a restrições de capacidade; a quantidade de fluxo que passa por qualquer corte ainda é limitada pela capacidade do corte. Para tornar o pré-fluxo um fluxo “válido”, o algoritmo envia então o excesso coletado nos reservatórios de vértices transbordantes de volta à origem, continuando a elevar vértices até uma altura maior que a altura  $|V|$  fixada para a origem. Como vemos, uma vez que todos os reservatórios tenham sido esvaziados, o pré-fluxo não é apenas um fluxo “válido”; ele também é um fluxo máximo.

## As operações básicas

Da discussão precedente, vemos que existem duas operações básicas executadas por um algoritmo de push-relabel: empurrar o fluxo em excesso de um vértice para um de seus vizinhos e elevar um vértice. A aplicabilidade dessas operações depende das alturas dos vértices, que agora definimos com exatidão.

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$  e sorvedor  $t$ , e seja  $f$  um pré-fluxo em  $G$ . Uma função  $b : V \rightarrow \mathbb{N}$  é uma **função de altura**<sup>3</sup> se  $b(s) = |V|$ ,  $b(t) = 0$  e

$$b(u) \leq b(v) + 1$$

para toda aresta residual  $(u, v) \in E_f$ . Obtemos imediatamente o lema a seguir.

### Lema 26.13

Seja  $G = (V, E)$  um fluxo em rede, seja  $f$  um pré-fluxo em  $G$ , e seja  $b$  uma função de altura em  $V$ . Para dois vértices quaisquer  $u, v \in V$ , se  $b(u) > b(v) + 1$ , então  $(u, v)$  não é uma aresta no grafo residual. ■

### A operação de empurrar

A operação básica  $\text{PUSH}(u, v)$  pode ser aplicada se  $u$  é um vértice transbordante,  $c_f(u, v) > 0$  e  $b(u) = b(v) + 1$ . O pseudocódigo a seguir atualiza o pré-fluxo  $f$  em uma rede implícita  $G = (V, E)$ . Ele pressupõe que as capacidades residuais também podem ser calculadas em tempo constante dados  $c$  e  $f$ . O fluxo em excesso armazenado em um vértice  $u$  é mantido como o atributo  $e[u]$ , e a altura de  $u$  é mantida como o atributo  $b[u]$ . A expressão  $d_f(u, v)$  é uma variável temporária que armazena a quantidade de fluxo que pode ser empurrada desde  $u$  até  $v$ .

#### PUSH( $u, v$ )

- 1 ▷ **Aplica-se quando:**  $u$  está transbordando,  $c_f(u, v) > 0$  e  $b[u] = b[v] + 1$ .
- 2 ▷ **Ação:** Empurrar  $d_f(u, v) = \min(e[u], c_f(u, v))$  unidades de fluxo desde  $u$  até  $v$ .
- 3  $d_f(u, v) \leftarrow \min(e[u], c_f(u, v))$
- 4  $f[u, v] \leftarrow f[u, v] + d_f(u, v)$
- 5  $f[v, u] \leftarrow -f[u, v]$
- 6  $e[u] \leftarrow e[u] - d_f(u, v)$
- 7  $e[v] \leftarrow e[v] + d_f(u, v)$

O código de PUSH opera da maneira mostrada a seguir. Supõe-se que o vértice  $u$  tenha um excesso positivo  $e[u]$ , e a capacidade residual de  $(u, v)$  seja positiva. Desse modo, podemos aumentar o fluxo de  $u$  até  $v$  por  $d_f(u, v) = \min(e[u], c_f(u, v))$  sem fazer  $e[u]$  se tornar negativo ou permitir que a capacidade  $c(u, v)$  seja excedida. O valor  $d_f(u, v)$  é calculado na linha 3, e atualizamos  $f$  nas linhas 4 e 5, e  $e$  nas linhas 6 e 7. Assim, se  $f$  é um pré-fluxo antes de PUSH ser chamado, ele continua a ser um pré-fluxo depois disso.

Observe que nada no código de PUSH depende das alturas de  $u$  e  $v$ , ainda que tenhamos proibido que ele seja invocado, a menos que  $b[u] = b[v] + 1$ . Portanto, o excesso de fluxo só é empurrado em declive por um diferencial de altura igual a 1. Pelo Lema 26.13, não existe nenhuma aresta residual entre dois vértices cujas alturas diferem por mais de 1, e assim não existe nenhum ganho em se permitir que o fluxo seja empurrado em declive por um diferencial de altura maior que 1.

<sup>3</sup> Na literatura, uma função de altura é chamada normalmente uma “função de distância”, e a altura de um vértice é chamada “etiqueta de distância”. Usamos o termo “altura” porque ele é mais sugestivo da intuição por trás do algoritmo. Conservamos o uso do termo “elevar” para fazer referência à operação que aumenta a altura de um vértice. A altura de um vértice está relacionada à sua distância desde o sorvedor  $t$ , como seria encontrado em uma busca em largura da transposta  $G^T$ .

Chamamos a operação  $\text{PUSH}(u, v)$  um **empurrão** de  $u$  até  $v$ . Se uma operação de empurrão se aplicar a alguma aresta  $(u, v)$  que deixa um vértice  $u$ , também diremos que a operação de empurrão se aplica a  $u$ . Ela é um **empurrão saturante** se a aresta  $(u, v)$  se torna **saturada** ( $c_f(u, v) = 0$  posteriormente); caso contrário, ela é um **empurrão não saturante**. Se uma aresta é saturada, ela não aparece na rede residual. Um lema simples caracteriza um resultado de um empurrão não saturante.

#### Lema 26.14

Depois de um empurrão não saturante de  $u$  para  $v$ , o vértice  $u$  não é mais transbordante.

**Prova** Tendo em vista que o empurrão foi não saturante, a quantidade de fluxo  $d_f(u, v)$  realmente empurrada tem de ser igual a  $e[u]$  antes do empurrão. Como  $e[u]$  é reduzido por essa quantidade, ele se torna 0 depois do empurrão.

#### A operação de elevar

A operação básica  $\text{RELABEL}(u)$  se aplica se  $u$  é transbordante e se  $b[u] \leq b[v]$  para todas as arestas  $(u, v) \in E_f$ . Em outras palavras, podemos elevar um vértice transbordante  $u$  se, para todo vértice  $v$  para o qual existe capacidade residual de  $u$  até  $v$ , o fluxo não puder ser empurrado de  $u$  para  $v$ , porque  $v$  não está abaixo de  $u$ . (Lembre-se de que, por definição, nem a origem  $s$  nem o sorvedor  $t$  podem ser transbordantes, e assim nem  $s$  nem  $t$  podem ser elevados.)

#### $\text{RELABEL}(u)$

- 1 ▷ **Aplica-se quando:**  $u$  está transbordando e, para todo  $v \in V$  tal que  $(u, v) \in E_f$ , temos  $b[u] \leq b[v]$ .
- 2 ▷ **Ação:** Aumentar a altura de  $u$ .
- 3  $b[u] \leftarrow 1 + \min\{b[v] : (u, v) \in E_f\}$

Quando chamamos a operação  $\text{RELABEL}(u)$ , dizemos que o vértice  $u$  é **elevado**. É importante observar que, quando  $u$  é elevado,  $E_f$  deve conter pelo menos uma aresta que deixe  $u$ , de modo que a minimização no código seja sobre um conjunto não vazio. Essa propriedade decorre da hipótese de que  $u$  está transbordando. Tendo em vista que  $e[u] > 0$ , temos  $e[u] = f[V, u] > 0$ , e consequentemente deve existir pelo menos um vértice  $v$  tal que  $f[v, u] > 0$ . Entretanto,

$$\begin{aligned} c_f(u, v) &= c(u, v) - f[u, v] \\ &= c(u, v) - f[u, v] \\ &> 0, \end{aligned}$$

o que implica que  $(u, v) \in E_f$ . Portanto, a operação  $\text{RELABEL}(u)$  fornece a  $u$  a maior altura permitida pelas restrições sobre funções de altura.

#### O algoritmo genérico

O algoritmo genérico de push-relabel utiliza a sub-rotina a seguir para criar um pré-fluxo inicial no fluxo em rede.

#### INITIALIZE-PREFLOW( $G, s$ )

- 1 **for** cada vértice  $u \in V[G]$
- 2   **do**  $b[u] \leftarrow 0$
- 3     $e[u] \leftarrow 0$
- 4 **for** cada aresta  $(u, v) \in E[G]$
- 5   **do**  $f[u, v] \leftarrow 0$
- 6     $f[v, u] \leftarrow 0$

```

7  $b[s] \leftarrow |V[G]|$ 
8 for cada vértice  $u \in Adj[s]$ 
9   do  $f[s, u] \leftarrow c(s, u)$ 
10   $f[u, s] \leftarrow -c(s, u)$ 
11   $e[u] \leftarrow c(s, u)$ 
12   $e[s] \leftarrow e[s] - c(s, u)$ 

```

INITIALIZE-PREFLOW cria um pré-fluxo inicial  $f$  definido por

$$f[u, v] = \begin{cases} c(u, v) & \text{se } u = s, \\ -c(v, u) & \text{se } v = s, \\ 0 & \text{caso contrário.} \end{cases} \quad (26.10)$$

Isto é, cada aresta que deixa a origem  $s$  é preenchida até a capacidade total, e todas as outras arestas não levam nenhum fluxo. Para cada vértice  $v$  adjacente à origem, temos inicialmente  $e[v] = c(s, v)$ , e  $e[s]$  é inicializado como o negativo da soma dessas capacidades. O algoritmo genérico também começa com uma função de altura inicial  $b$ , dada por

$$b[u] = \begin{cases} |V| & \text{se } u = s, \\ 0 & \text{caso contrário.} \end{cases}$$

Essa é uma função de altura porque as únicas arestas  $(u, v)$  para as quais  $b[u] > b[v] + 1$  são aquelas para as quais  $u = s$ , e essas arestas são saturadas, o que significa que elas não estão na rede residual.

A inicialização, seguida por uma seqüência de operações de empurrão e elevação, executadas sem qualquer ordem particular, produz o algoritmo GENERIC-PUSH-RELABEL:

GENERIC-PUSH-RELABEL( $G$ )

- 1 INITIALIZE-PREFLOW( $G, s$ )
- 2 **while** existir uma operação de empurrão ou elevação aplicável
- 3 **do** selecionar uma operação de empurrão ou elevação aplicável e executá-la

O lema a seguir nos informa que, desde que exista um vértice transbordante, pelo menos uma das duas operações básicas se aplica.

**Lema 26.15 (Um vértice transbordante pode ser empurrado ou elevado)**

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$  e sorvedor  $t$ , seja  $f$  um pré-fluxo e seja  $b$  qualquer função de altura para  $f$ . Se  $u$  é qualquer vértice transbordante, então uma operação de empurrão ou elevação se aplica a ele.

**Prova** Para qualquer aresta residual  $(u, v)$ , temos  $b(u) \leq b(v) + 1$ , porque  $b$  é uma função de altura. Se uma operação de empurrão não se aplica a  $u$ , então, para todas as arestas residuais  $(u, v)$ , devemos ter  $b(u) < b(v) + 1$ , o que implica  $b(u) \leq b(v)$ . Desse modo, uma operação de elevação pode ser aplicada a  $u$ . ■

### Correção do método push-relabel

Para mostrar que o algoritmo genérico de push-relabel resolve o problema de fluxo máximo, devemos primeiro provar que, se ele terminar, o pré-fluxo  $f$  será um fluxo máximo. Devemos mais tarde provar que ele termina. Começamos com algumas observações sobre a função de altura  $b$ .

**Lema 26.16 (Alturas de vértices nunca diminuem)**

Durante a execução do algoritmo GENERIC-PUSH-RELABEL sobre um fluxo em rede  $G = (V, E)$ , | 533

para cada vértice  $u \in V$ , a altura  $b[u]$  nunca diminui. Além disso, sempre que uma operação de elevação é aplicada a um vértice  $u$ , sua altura  $b[u]$  aumenta por pelo menos 1.

**Prova** Pelo fato de que as alturas de vértices só mudam durante operações de elevação, basta provar a segunda declaração do lema. Se o vértice  $u$  está prestes a ser elevado, então, para todos os vértices  $v$  tais que  $(u, v) \in E_f$ , temos  $b[u] \leq b[v]$ . Portanto,  $b[u] < 1 + \min\{b[v] : (u, v) \in E_f\}$  e assim a operação deve aumentar  $b[u]$ . ■

### Lema 26.17

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$  e sorvedor  $t$ . Durante a execução de GENERIC-PUSH-RELABEL sobre  $G$ , o atributo  $b$  é mantido como uma função de altura.

**Prova** A prova é por indução sobre o número de operações básicas executadas. Inicialmente,  $b$  é uma função de altura, como já observamos.

Afirmamos que, se  $b$  é uma função de altura, então uma operação RELABEL( $u$ ) faz de  $b$  uma função de altura. Se examinarmos uma aresta residual  $(u, v) \in E_f$  que sai de  $u$ , então a operação RELABEL( $u$ ) assegura que  $b[u] \leq b[v] + 1$  depois disso. Agora, considere uma aresta residual  $(w, u)$  que entra em  $u$ . Pelo Lema 26.16,  $b[w] \leq b[u] + 1$  antes da operação RELABEL( $u$ ) implicar  $b[w] < b[u] + 1$  depois disso. Desse modo, a operação RELABEL( $u$ ) faz de  $b$  uma função de altura.

Agora, considere uma operação PUSH( $u, v$ ). Essa operação pode adicionar a aresta  $(v, u)$  a  $E_f$  e pode remover  $(u, v)$  de  $E_f$ . No primeiro caso, temos  $b[v] = b[u] - 1 < b[u] + 1$ , e assim  $b$  continua a ser uma função de altura. No último caso, a remoção de  $(u, v)$  da rede residual remove a restrição correspondente, e  $b$  mais uma vez continua a ser uma função de altura. ■

O lema a seguir fornece uma importante propriedade de funções de altura.

### Lema 26.18

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$  e sorvedor  $t$ , seja  $f$  um pré-fluxo em  $G$  e seja  $b$  uma função de altura em  $V$ . Então, não existe nenhum caminho desde a origem  $s$  até o depósito  $t$  na rede residual  $G_f$ .

**Prova** Suponha para fins de contradição que exista um caminho  $p = \langle v_0, v_1, \dots, v_k \rangle$  desde  $s$  até  $t$  em  $G_f$ , onde  $v_0 = s$  e  $v_k = t$ . Sem perda de generalidade,  $p$  é um caminho simples, e assim  $k < |V|$ . Para  $i = 0, 1, \dots, k-1$ , a aresta  $(v_i, v_{i+1})$ . Como  $b$  é uma função de altura,  $b(v_i) \leq b(v_{i+1}) + 1$  para  $i = 0, 1, \dots, k-1$ . A combinação dessas desigualdades sobre o caminho  $p$  produz  $b(s) \leq b(t)$ . Porém, como  $b(t) = 0$ , temos  $b(v_i) \leq k < |V|$ , o que contradiz o requisito de que  $b(s) = |V|$  em uma função de altura.

Agora, estamos prontos para mostrar que, se o algoritmo genérico de push-relabel terminar, o pré-fluxo que ele calcula será um fluxo máximo. ■

### Teorema 26.19 (Correção do algoritmo genérico de push-relabel)

Se o algoritmo GENERIC-PUSH-RELABEL terminar quando for executado sobre um fluxo em rede  $G = (V, E)$  com origem  $s$  e sorvedor  $t$ , então o pré-fluxo  $f$  que ele calcula será um fluxo máximo para  $G$ .

**Prova** Usamos o loop invariante a seguir:

Cada vez que o teste do loop while da linha 2 em GENERIC-PUSH-RELABEL é executado,  $f$  é um pré-fluxo.

**Inicialização:** INITIALIZE-PREFLOW faz de  $f$  um pré-fluxo.

**Manutenção:** As únicas operações dentro do loop while das linhas 2 e 3 são empurrar e elevar.

As operações de elevar afetam apenas os atributos de altura e não os valores de fluxo; consequentemente, elas não afetam o fato de  $f$  ser ou não um pré-fluxo. Como demonstramos na página (531), se  $f$  é um pré-fluxo antes de uma operação de empurrão, ele continua a ser um pré-fluxo depois disso.

**Término:** No término, cada vértice em  $V - \{s, t\}$  deve ter um excesso 0 porque, pelos Lemas 26.15 e 26.17, e pelo invariante de que  $f$  é sempre um pré-fluxo, não há nenhum vértice transbordante. Então,  $f$  é um fluxo. Como  $b$  é uma função de altura, o Lema 26.18 nos diz que não existe nenhum caminho de  $s$  até  $t$  na rede residual  $G_f$ . Então, pelo teorema de fluxo máximo e corte mínimo (Teorema 26.7),  $f$  é um fluxo de máximo. ■

## Análise do método de push-relabel

Para mostrar que o algoritmo genérico de push-relabel de fato termina, limitaremos o número de operações que ele executa. Cada um dos três tipos de operações – elevações, empurros saturantes e empurros não saturantes – é limitado separadamente. Com o conhecimento desses limites, é um problema simples construir um algoritmo que seja executado no tempo  $O(V^2E)$ . Contudo, antes de iniciarmos a análise, provaremos um importante lema.

### Lema 26.20

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$  e sorvedor  $t$ , e seja  $f$  um pré-fluxo em  $G$ . Então, para qualquer vértice transbordante  $u$ , existe um caminho simples desde  $u$  até  $s$  na rede residual  $G_f$ .

**Prova** Para um vértice transbordante  $u$ , seja  $U = \{v : \text{existe um caminho simples desde } u \text{ até } v \text{ em } G_f\}$ , e suponha para fins de contradição que  $s \notin U$ . Seja  $\bar{U} = V - U$ . Afirmamos para cada par de vértices  $v \in U$  e  $w \in \bar{U}$  que  $f(w, v) \leq 0$ . Por quê? Se  $f(w, v) > 0$ , então  $f(v, w) < 0$ , o que implica que  $c_f(v, w) = c(v, w) - f(v, w) > 0$ . Conseqüentemente, existe uma aresta  $(v, w) \in E_f$ , e assim um caminho simples da forma  $u \sim v \rightarrow w$  em  $G_f$ , contradizendo nossa escolha de  $w$ .

Desse modo, devemos ter  $f(\bar{U}, U) \leq 0$ , pois todo termo nesse somatório implícito é não positivo, e então

$$\begin{aligned} e(U) &= f(V, U) && (\text{pela equação (26.9)}) \\ &= f(\bar{U}, U) + f(U, U) && (\text{pelo Lema 26.1, parte (3)}) \\ &= f(\bar{U}, U) && (\text{pelo Lema 26.1, parte (1)}) . \\ &\leq 0 . \end{aligned}$$

Os excessos são não negativos para todos os vértices em  $V - \{s\}$ ; como temos pressuposto que  $U \subseteq V - \{s\}$ , devemos então ter  $e(v) = 0$  para todos os vértices  $v \in U$ . Em particular,  $e(u) = 0$ , o que contradiz a hipótese de que  $u$  é de transbordamento. ■

O próximo lema limita as alturas dos vértices, e seu corolário limita o número de operações de elevação que são executadas ao todo.

### Lema 26.21

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$  e sorvedor  $t$ . Em qualquer instante durante a execução de GENERIC-PUSH-RELABEL sobre  $G$ , temos  $b[u] \leq 2|V| - 1$  para todos os vértices  $u \in V$ .

**Prova** As alturas da origem  $s$  e do sorvedor  $t$  nunca mudam, porque esses vértices são por definição não transbordantes. Desse modo, sempre temos  $b[s] = |V|$  e  $b[t] = 0$ , e ambos não são maiores que  $2|V| - 1$ .

Agora considere qualquer vértice  $u \in V - \{s, t\}$ . Inicialmente,  $b[u] = 0 \leq 2|V| - 1$ . Mostraremos que, após cada operação de elevação, ainda temos  $b[u] \leq 2|V| - 1$ . Quando  $u$  é elevado, ele está transbordando e o Lema 26.20 nos informa que existe um caminho simples  $p$  desde  $u$  até  $s$  em  $G_f$ . Seja  $p = \langle v_0, v_1, \dots, v_k \rangle$ , onde  $v_0 = u$ ,  $v_k = s$  e  $k \leq |V| - 1$ , porque  $p$  é simples. Para  $i = 0, 1, \dots, k - 1$ , temos  $(v_i, v_{i+1}) \in E_f$  e, conseqüentemente, pelo Lema 26.17,  $b[v_i] \leq b[v_{i+1}] + 1$ . A expansão dessas desigualdades sobre o caminho  $p$  resulta em  $b[u] = b[v_0] \leq b[v_k] + k \leq b[s] + (|V| - 1) = 2|V| - 1$ . ■

**Corolário 26.22 (Limite sobre operações de elevação)**

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$  e sorvedor  $t$ . Então, durante a execução de GENERIC-PUSH-RELABEL sobre  $G$ , o número de operações de elevação é no máximo  $2|V| - 1$  por vértice  $e$  e no máximo  $(2|V| - 1)(|V| - 2) < 2|V|^2$  de modo geral.

**Prova** Somente os  $|V| - 2$  vértices em  $V - \{s, t\}$  podem ser elevados. Seja  $u \in V - \{s, t\}$ . A operação RELABEL( $u$ ) aumenta  $b[u]$ . O valor de  $b[u]$  é inicialmente 0 e, pelo Lema 26.21, cresce até no máximo  $2|V| - 1$ . Assim, cada vértice  $u \in V - \{s, t\}$  é elevado no máximo  $2|V| - 1$  vezes, e o número total de operações de elevação executadas é no máximo  $(2|V| - 1)(|V| - 2) < 2|V|^2$ . ■

O Lema 26.21 também nos ajuda a limitar o número de empurrões saturantes.

**Lema 26.23 (Limite sobre empurrões saturantes)**

Durante a execução de GENERIC-PUSH-RELABEL em qualquer rede de fluxo  $G = (V, E)$ , o número de empurrões saturantes é menor que  $2|V||E|$ .

**Prova** Para qualquer par de vértices  $u, v \in V$ , considere os empurrões saturantes desde  $u$  até  $v$  e desde  $v$  até  $u$  juntos, chamando-os de empurrões saturantes entre  $u$  e  $v$ . Se existem quaisquer desses empurrões, pelo menos um de  $(u, v)$  e  $(v, u)$  é na realidade uma aresta em  $E$ . Agora, suponha que tenha ocorrido um empurrão saturante desde  $u$  até  $v$ . Nesse momento,  $b[v] = b[u] - 1$ . Para ocorrer mais tarde outro empurrão de  $u$  até  $v$ , o algoritmo deve primeiro empurrar o fluxo desde  $v$  até  $u$ , o que não pode acontecer até  $b[v] = b[u] + 1$ . Tendo em vista que  $b[u]$  nunca diminui, para  $b[v] = b[u] + 1$ , o valor de  $b[v]$  deve aumentar pelo menos 2 unidades. Da mesma forma,  $b[u]$  deve aumentar por pelo menos 2 entre empurrões saturantes de  $v$  até  $u$ . As alturas começam em 0 e, pelo Lema 26.21, nunca excedem  $2|V| - 1$ , o que implica que o número de vezes que qualquer vértice pode ter sua altura aumentada por 2 é menor que  $|V|$ . Tendo em vista que pelo menos um dentre  $b[u]$  e  $b[v]$  deve aumentar por 2 entre dois empurrões saturantes quaisquer entre  $u$  e  $v$ , existem menos de  $2|V|$  empurrões saturantes entre  $u$  e  $v$ . A multiplicação pelo número de arestas fornece um limite de menos de  $2|V||E|$  sobre o número total de empurrões saturantes. ■

O lema a seguir limita o número de empurrões não saturantes no algoritmo genérico de push-relabel.

**Lema 26.24 (Limite sobre empurrões não saturantes)**

Durante a execução de GENERIC-PUSH-RELABEL em qualquer rede de fluxo  $G = (V, E)$ , o número de empurrões não saturantes é menor que  $4|V|^2(|V| + |E|)$ .

**Prova** Defina uma função potencial  $\Phi = \sum_{v:e(v)>0} b[v]$ . Inicialmente,  $\Phi = 0$ , e o valor de  $\Phi$  pode mudar após cada elevação, empurrão saturante e empurrão não saturante. Limitaremos a quantidade com que os empurrões saturantes e as elevações podem contribuir para o aumento de  $\Phi$ . Então, mostraremos que cada empurrão não saturante deve diminuir  $\Phi$  por pelo menos 1, e usaremos esses limites para derivar um limite superior sobre o número de empurrões não saturantes.

Vamos examinar os dois modos pelos quais  $\Phi$  pode aumentar. Primeiro, a elevação de um vértice  $u$  aumenta  $\Phi$  por menos de  $2|V|$ , pois o conjunto sobre o qual a soma é tomada é o mesmo, e a elevação não pode aumentar a altura de  $u$  por mais que sua altura máxima possível que, pelo Lema 26.21, é no máximo  $2|V| - 1$ . Em segundo lugar, um empurrão saturante de um vértice  $u$  até um vértice  $v$  aumenta  $\Phi$  por menos que  $2|V|$ , pois nenhuma altura muda e só o vértice  $v$ , cuja altura é no máximo  $2|V| - 1$ , talvez possa se tornar transbordante.

Agora, mostramos que um empurrão não saturante de  $u$  para  $v$  diminui  $\Phi$  por pelo menos 1. Por quê? Antes do empurrão não saturante,  $u$  era transbordante, e  $v$  pode ou não ter sido transbordante. Pelo Lema 26.14,  $u$  não é mais transbordante após o empurrão. Além disso,  $v$  deve ser transbordante depois do empurrão, a menos que ele seja a origem. Então, a função potencial  $\Phi$

diminuiu por exatamente  $b[u]$  e aumentou por 0 ou  $b[v]$ . Tendo em vista que  $b[u] - b[v] = 1$ , o efeito líquido é que a função potencial diminuiu por pelo menos 1.

Desse modo, durante o curso do algoritmo, a quantidade total de aumento em  $\Phi$  se deve a elevações e empurrões saturados, e é limitada pelo Corolário 26.22 e pelo Lema 26.23 a ser no máximo  $(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V| + |E|)$ . Tendo em vista que  $\Phi$ , a quantidade total de diminuição, e portanto o número total de empurrões não saturantes, é menor que  $4|V|^2(|V| + |E|)$ . ■

Tendo limitado o número de elevações, empurrões saturantes e empurrões não saturantes, temos definido o cenário para a análise seguinte do procedimento GENERIC-PUSH-RELABEL e, consequentemente, de qualquer algoritmo baseado no método de push-relabel.

### **Teorema 26.25**

Durante a execução de GENERIC-PUSH-RELABEL sobre qualquer fluxo em rede  $G = (V, E)$ , o número de operações básicas é  $O(V^2E)$ .

**Prova** Imediata a partir do Corolário 26.22 e dos Lemas 26.23 e 26.24. ■

Desse modo, o algoritmo termina depois de  $O(V^2E)$  operações. Tudo que resta é fornecer um método eficiente para implementar cada operação e para escolher uma operação apropriada a executar.

### **Corolário 26.26**

Existe uma implementação do algoritmo genérico de push-relabel que é executada no tempo  $O(V^2E)$  sobre qualquer fluxo em rede  $G = (V, E)$ .

**Prova** O Exercício 26.4-1 lhe pede para mostrar como implementar o algoritmo genérico com uma sobrecarga  $O(V)$  por operação de elevação e  $O(1)$  por empurrão. Ele também lhe pede para projetar uma estrutura de dados que lhe permite escolher uma operação aplicável no tempo  $O(1)$ . E assim decorre o corolário.

## **Exercícios**

### **26.4-1**

Mostre como implementar o algoritmo genérico de push-relabel usando o tempo  $O(V)$  por operação de elevação, o tempo  $O(1)$  por empurrão e o tempo  $O(1)$  para selecionar uma operação aplicável, resultando o tempo total  $O(V^2E)$ .

### **26.4-2**

Prove que o algoritmo genérico de push-relabel despende ao todo apenas o tempo  $O(VE)$  na execução de todas as  $O(V^2)$  operações de elevação.

### **26.4-3**

Suponha que um fluxo máximo tenha sido encontrado em um fluxo em rede  $G = (V, E)$  usando-se um algoritmo de push-relabel. Forneça um algoritmo rápido para encontrar um corte mínimo em  $G$ .

### **26.4-4**

Forneça um algoritmo de push-relabel eficiente para encontrar um emparelhamento máximo em um grafo bipartido. Analise seu algoritmo.

### **26.4-5**

Suponha que todas as capacidades de arestas em um fluxo em rede  $G = (V, E)$  estejam no conjunto  $\{1, 2, \dots, k\}$ . Analise o tempo de execução do algoritmo genérico de push-relabel em termos de  $|V|$ ,  $|E|$  e  $k$ . (Sugestão: Quantas vezes cada aresta pode admitir um empurrão não saturante antes de se tornar saturada?)

#### 26.4-6

Mostre que a linha 7 de INITIALIZE-PREFLOW pode ser alterada para

$$7 \quad b[s] \leftarrow |V[G]| - 2$$

sem afetar a correção ou o desempenho assintótico do algoritmo genérico de push-relabel.

#### 26.4-7

Seja  $\delta_f(u, v)$  a distância (número de arestas) desde  $u$  até  $v$  na rede residual  $G_f$ . Mostre que GENERIC-PUSH-RELABEL mantém as propriedades de que  $b[u] < |V|$  implica  $b[u] \leq \delta_f(u, t)$  e de que  $b[u] \geq |V|$  implica  $b[u] - |V| \leq \delta_f(u, s)$ .

#### 26.4-8 \*

Como no exercício anterior, seja  $\delta_f(u, v)$  a distância desde  $u$  até  $v$  na rede residual  $G_f$ . Mostre como o algoritmo genérico de push-relabel pode ser modificado para manter a propriedade de que  $b[u] < |V|$  implica  $b[u] = \delta_f(u, t)$ , e de que  $b[u] \geq |V|$  implica  $b[u] - |V| = \delta_f(u, s)$ . O tempo total que sua implementação dedicará à manutenção dessa propriedade deve ser  $O(VE)$ .

#### 26.4-9

Mostre que o número de empurrões não saturantes executados por GENERIC-PUSH-RELABEL sobre um fluxo em rede  $G = (V, E)$  é no máximo  $4|V|^2|E|$  para  $|V| \geq 4$

## ★ 26.5 O algoritmo de relabel-to-front

O método push-relabel nos permite aplicar as operações básicas em absolutamente qualquer ordem. Porém, escolhendo a ordem com cuidado e administrando com eficiência a estrutura de dados da rede, podemos resolver o problema de fluxo máximo com maior rapidez que o limite  $O(V^2E)$  dado pelo Corolário 26.26. Examinaremos agora o algoritmo de relabel-to-front, um algoritmo de push-relabel cujo tempo de execução é  $O(V^3)$ , o qual é assintoticamente pelo menos tão bom quanto  $O(V^2E)$ , e melhor no caso de redes densas.

O algoritmo de relabel-to-front mantém uma lista dos vértices na rede. Começando na frente, o algoritmo varre a lista, selecionando repetidamente um vértice de transbordamento  $u$  e depois “descarregando-o”, ou seja, executando operações de empurrar e elevar até  $u$  não ter mais um excesso positivo. Sempre que um vértice é elevado, ele é deslocado para a frente da lista (daí o nome “relabel-to-front”) e o algoritmo começa uma nova varredura.

A correção e a análise do algoritmo de relabel-to-front dependem da noção de arestas “admissíveis”: aquelas arestas na rede residual pelas quais o fluxo pode ser empurrado. Depois de provar algumas propriedades sobre a rede de arestas admissíveis, investigaremos a operação de descarga e depois apresentaremos e analisaremos o algoritmo relabel-to-front propriamente dito.

### Arestas e redes admissíveis

Se  $G = (V, E)$  é um fluxo em rede com origem  $s$  e sorvedor  $t$ ,  $f$  é um pré-fluxo em  $G$ , e  $b$  é uma função de altura, então dizemos que  $(u, v)$  é uma **aresta admissível** se  $c_f(u, v) > 0$  e  $b(u) = b(v) + 1$ . Caso contrário,  $(u, v)$  é **inadmissível**. A **rede admissível** é  $G_{f,b} = (V, E_{f,b})$ , onde  $E_{f,b}$  é o conjunto de arestas admissíveis.

A rede admissível consiste naquelas arestas pelas quais o fluxo pode ser empurrado. O lema a seguir mostra que essa rede é um grafo acíclico orientado (gao).

#### Lema 26.27 (A rede admissível é acíclica)

Se  $G = (V, E)$  é um fluxo em rede,  $f$  é um pré-fluxo em  $G$ , e  $b$  é uma função de altura em  $G$ , então a rede admissível  $G_{f,b} = (V, E_{f,b})$  é acíclica.

**Prova** A prova é por contradição. Suponha que  $G_{f,b}$  contenha um ciclo  $p = \langle v_0, v_1, \dots, v_k \rangle$ , onde  $v_0 = v_k$  e  $k > 0$ . Tendo em vista que cada aresta em  $p$  é admissível, temos  $b(v_{i-1}) = b(v_i) + 1$  para  $i = 1, 2, \dots, k$ . O somatório em torno do ciclo fornece

$$\begin{aligned} \sum_{i=1}^k b(v_i - 1) &= \sum_{i=1}^k (b(v_i) + 1) \\ &= \sum_{i=1}^k b(v_i) + k. \end{aligned}$$

Pelo fato de cada vértice no ciclo  $p$  aparecer uma vez em cada um dos somatórios, derivamos a contradição de que  $0 = k$ .

Os dois lemas seguintes mostram como as operações de empurrar e elevar mudam a rede admissível.

### Lema 26.28

Seja  $G = (V, E)$  um fluxo em rede, seja  $f$  um pré-fluxo em  $G$ , e seja  $b$  uma função de altura. Se um vértice  $u$  é de transbordamento e  $(u, v)$  é uma aresta admissível, então  $\text{PUSH}(u, v)$  se aplica. A operação não cria quaisquer novas arestas admissíveis, mas pode fazer  $(u, v)$  se tornar inadmissível.

**Prova** Pela definição de uma aresta admissível, o fluxo pode ser empurrado desde  $u$  até  $v$ . Tendo em vista que  $u$  é de transbordamento, a operação  $\text{PUSH}(u, v)$  se aplica. A única aresta residual nova que pode ser criada empurrando-se o fluxo de  $u$  até  $v$  é a aresta  $(v, u)$ . Como  $b(v) = b(u) - 1$ , a aresta  $(v, u)$  não pode se tornar admissível. Se a operação é um empurrão saturante, então  $c_f(u, v) = 0$  daí em diante e  $(u, v)$  se torna inadmissível. ■

### Lema 26.29

Seja  $G = (V, E)$  um fluxo em rede, seja  $f$  um pré-fluxo em  $G$  e suponha que o atributo  $b$  seja uma função de altura. Se um vértice  $u$  é de transbordamento e não existe nenhuma aresta admissível saindo de  $u$ , então  $\text{RELABEL}(u)$  se aplica. Após a operação de elevação, existe pelo menos uma aresta admissível saindo de  $u$ , mas não existe nenhuma aresta admissível entrando em  $u$ .

**Prova** Se  $u$  é de transbordamento então, pelo Lema 26.15, uma operação de empurrão ou de elevação se aplica a ele. Se não existe nenhuma aresta admissível saindo de  $u$ , nenhum fluxo pode ser empurrado a partir de  $u$  e  $\text{RELABEL}(u)$  se aplica. Depois da operação de elevação,  $b[u] = 1 + \min\{b[v] : (u, v) \in E_f\}$ . Desse modo, se  $v$  é um vértice que realiza o mínimo nesse conjunto, a aresta  $(u, v)$  se torna admissível. Conseqüentemente, após a elevação, existe pelo menos uma aresta admissível saindo de  $u$ .

Para mostrar que nenhuma aresta admissível entra em  $u$  depois de uma operação de elevação, suponha que exista um vértice  $v$  tal que  $(v, u)$  seja admissível. Então,  $b[v] = b[u] + 1$  após a elevação, e assim  $b[v] > b[u] + 1$  imediatamente antes da elevação. Porém, pelo Lema 26.13, não existe nenhuma aresta residual entre os vértices cujas alturas diferem por mais de 1. Além disso, a elevação de um vértice não muda a rede residual. Portanto,  $(v, u)$  não está na rede residual e, conseqüentemente, não pode estar na rede admissível. ■

## Listas de vizinhos

As arestas no algoritmo de relabel-to-front estão organizadas em “listas de vizinhos”. Dado um fluxo em rede  $G = (V, E)$ , a **lista de vizinhos**  $N[u]$  para um vértice  $u \in V$  é uma lista unicamente ligada dos vizinhos de  $u$  em  $G$ . Desse modo, o vértice  $v$  aparece na lista  $N[u]$  se  $(u, v) \in E$  ou  $(v, u) \in E$ . A lista de vizinhos  $N[u]$  contém exatamente os vértices  $v$  para os quais pode existir uma aresta residual  $(u, v)$ . O primeiro vértice em  $N[u]$  é indicado por  $\text{início}[N[u]]$ . O vértice que segue  $v$  em uma lista de vizinhos é indicado por  $\text{próximo-vizinho}[v]$ ; esse ponteiro é NIL se  $v$  é o último vértice na lista de vizinhos.

O algoritmo de relabel-to-front circula por cada lista de vizinhos em uma ordem arbitrária que é fixa por toda a execução do algoritmo. Para cada vértice  $u$ , o campo  $atual[u]$  aponta para o vértice que está sendo considerado atualmente em  $N[u]$ . No início  $atual[u]$  é definido como  $início[N[u]]$ .

## Como descarregar um vértice de transbordamento

Um vértice de transbordamento  $u$  é *descarregado* empurrando-se todo o seu excesso por arestas admissíveis para vértices vizinhos, elevando-se  $u$  conforme necessário para fazer as arestas que saem de  $u$  se tornarem admissíveis. O pseudocódigo é dado a seguir.

```
DISCHARGE( $u$ )
1 while  $e[u] > 0$ 
2   do  $v \leftarrow atual[u]$ 
3     if  $v = NIL$ 
4       then RELABEL( $u$ )
5          $atual[u] \leftarrow início[N[u]]$ 
6     elseif  $c_f(u, v) > 0$  e  $b[u] = b[v] + 1$ 
7       then PUSH( $u, v$ )
8     else  $atual[u] \leftarrow próximo-vizinho[v]$ 
```

A Figura 26.9 percorre passo a passo diversas iterações do loop **while** das linhas 1 a 8, o qual é executado enquanto o vértice  $u$  tem excesso positivo. Cada iteração executa exatamente uma entre três ações, dependendo do vértice atual  $v$  na lista de vizinhos  $N[u]$ .

1. Se  $v$  é NIL, então chegamos ao fim de  $N[u]$ . A linha 4 eleva o vértice  $u$ , e então a linha 5 redefine o vizinho atual de  $u$  como o primeiro em  $N[u]$ . (O Lema 26.30 a seguir declara que a operação de elevação se aplica nessa situação.)
2. Se  $v$  é não NIL e  $(u, v)$  é uma aresta admissível (o que é determinado pelo teste na linha 6), então a linha 7 empurra algum (ou possivelmente todo) excesso de  $u$  para o vértice  $v$ .
3. Se  $v$  é não NIL mas  $(u, v)$  é inadmissível, então a linha 8 *avança*  $atual[u]$  mais uma posição na lista de vizinhos  $N[u]$ .

Observe que, se DISCHARGE é chamado em um vértice transbordante  $u$ , então a última ação executada por DISCHARGE deve ser um empurrão a partir de  $u$ . Por quê? O procedimento só termina quando  $e[u]$  se torna zero, e nem a operação de elevação nem o avanço do ponteiro  $atual[u]$  afeta o valor de  $e[u]$ .

Devemos estar certos de que, quando PUSH ou RELABEL é chamado por DISCHARGE, a operação se aplica. O próximo lema prova esse fato.

### Lema 26.30

Se DISCHARGE chama PUSH( $u, v$ ) na linha 7, então uma operação de empurrão se aplica a  $(u, v)$ . Se DISCHARGE chama RELABEL( $u$ ) na linha 4, então uma operação de elevação se aplica a  $u$ .

**Prova** Os testes das linhas 1 e 6 asseguram que uma operação de empurrão só ocorrerá se a operação se aplicar, o que prova a primeira declaração do lema.

Para provar a segunda declaração, de acordo com o teste na linha 1 e com o Lema 26.29, precisamos apenas mostrar que todas as arestas que saem de  $u$  são inadmissíveis. Observe que, à medida que DISCHARGE( $u$ ) é chamado repetidamente, o ponteiro  $atual[u]$  se desloca para baixo na lista  $N[u]$ . Cada “passagem” começa no início de  $N[u]$  e termina com  $atual[u] = NIL$ , e nesse ponto  $u$  é elevado e uma nova passagem começa. Para o ponteiro  $atual[u]$  avançar além de um vértice  $v \in N[u]$  durante uma passagem, a aresta  $(u, v)$  deve ser julgada inadmissível pelo teste da linha 6. Desse modo, quando a passagem se completa, toda aresta que sai de  $u$  é consi-

derada inadmissível em algum momento durante a passagem. A observação fundamental é que, no final da passagem, toda aresta que sai de  $u$  ainda é inadmissível. Por quê? Pelo Lema 26.28, empurros não podem criar quaisquer arestas admissíveis, deixando apenas uma sair de  $u$ . Desse modo, qualquer aresta admissível deve ser criada por uma operação de elevação. Contudo, o vértice  $u$  não é elevado durante a passagem e, pelo Lema 26.29, qualquer outro vértice  $v$  que é elevado durante a passagem não tem nenhuma aresta de entrada admissível. Portanto, no fim da passagem, todas as arestas que saem de  $u$  permanecem inadmissíveis, e o lema é provado. ■

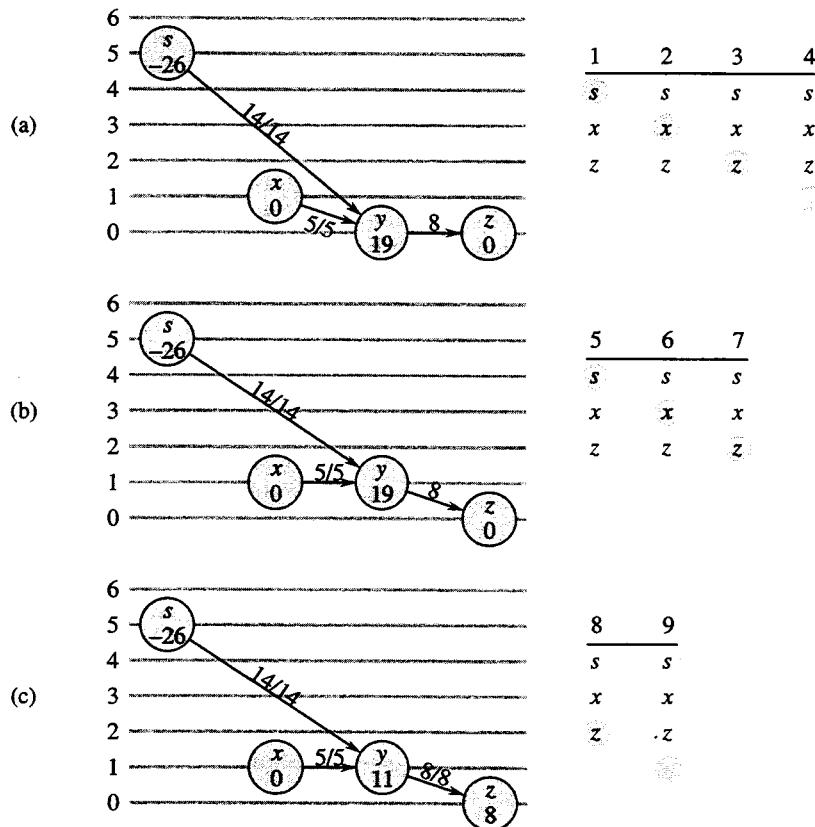


FIGURA 26.9 Descarregando um vértice  $y$ . São necessárias 15 iterações do loop **while** de DISCHARGE para empurrar todo o fluxo em excesso a partir do vértice  $y$ . Somente os vizinhos de  $y$  e as arestas que entram ou partem de  $y$  são mostrados. Em cada parte, o número dentro de cada vértice é seu excesso no início da primeira iteração apresentada na parte, e cada vértice é mostrado com sua altura ao longo da parte. À direita é mostrada a lista de vizinhos  $N[y]$  no início de cada iteração, com o número da iteração na parte superior. O vizinho sombreado é  $atual[y]$ . (a) Inicialmente, existem 19 unidades de excesso para empurrar a partir de  $y$ , e  $atual[y] = s$ . As iterações 1, 2 e 3 simplesmente avançam  $atual[y]$ , pois não existe nenhuma aresta admissível saindo de  $y$ . Na iteração 4,  $atual[y] = \text{NIL}$  (mostrado pelo sombreamento abaixo da lista de vizinhos), e assim  $y$  é elevado e  $atual[y]$  é redefinido como o início da lista de vizinhos. (b) Após a elevação, o vértice  $y$  tem altura 1. Nas iterações 5 e 6, descobrimos que as arestas  $(y, s)$  e  $(y, x)$  são inadmissíveis, mas 8 unidades de fluxo em excesso são empurradas desde  $y$  até  $z$  na iteração 7. Em virtude do empurro,  $atual[y]$  não é avançado nessa iteração. (c) Tendo em vista que o empurro na iteração 7 saturou a aresta  $(y, z)$ , descobrimos que ela é inadmissível na iteração 8. Na iteração 9,  $atual[y] = \text{NIL}$ , e assim o vértice  $y$  é novamente elevado, e  $atual[y]$  é redefinido. (d) Na iteração 10,  $(y, s)$  é inadmissível, mas 5 unidades de fluxo em excesso são empurradas de  $y$  para  $x$  na iteração 11. (e) Como  $atual[y]$  não foi avançado na iteração 11, a iteração 12 descobre que  $(y, x)$  é inadmissível. A iteração 13 descobre que  $(y, z)$  é inadmissível, e a iteração 14 eleva o vértice  $y$  e redefine  $atual[y]$ . (f) A iteração 15 empurra 6 unidades de fluxo em excesso desde  $y$  até  $s$ . (g) O vértice  $y$  não tem agora nenhum fluxo em excesso, e DISCHARGE termina. Neste exemplo, DISCHARGE começa e também termina com o ponteiro atual no início da lista de vizinhos mas, em geral, essa necessidade não existe

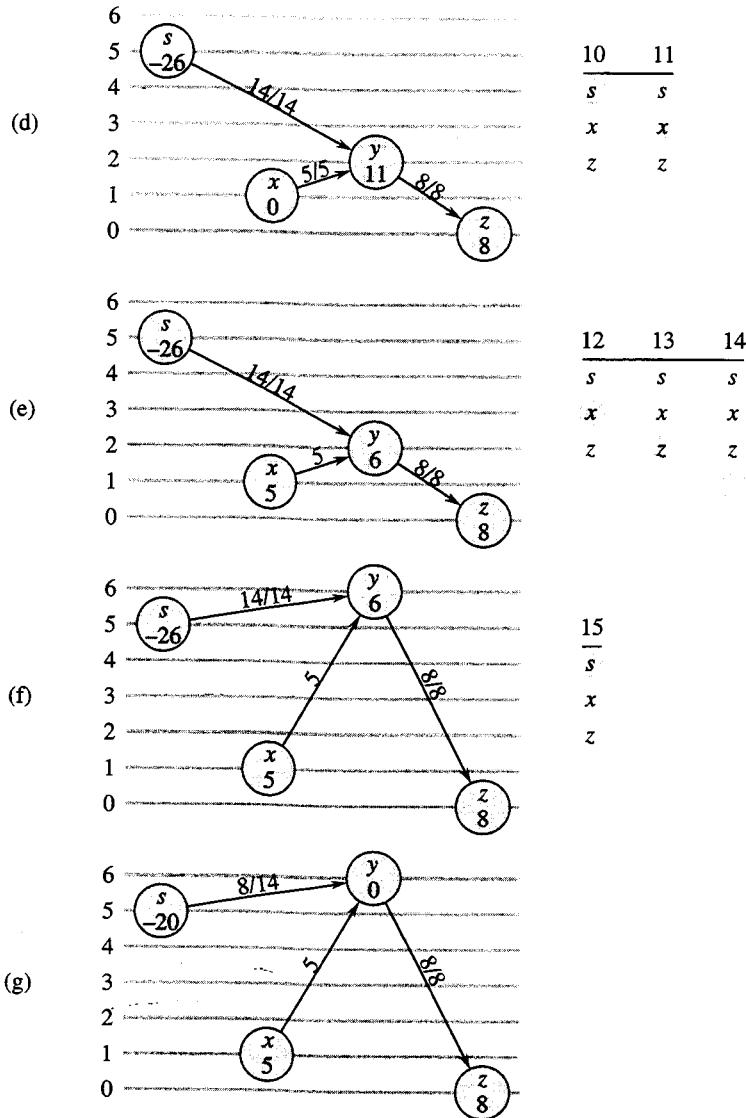


FIGURA 26.9 Continuação

### O algoritmo de relabel-to-front

No algoritmo de relabel-to-front, mantemos uma lista ligada  $L$  que consiste em todos os vértices em  $V - \{s, t\}$ . Uma propriedade fundamental é que os vértices em  $L$  são ordenados topologicamente de acordo com a rede admissível, como veremos no invariante loop a seguir. (Vimos no Lema 26.27 que a rede admissível é um gao.)

O pseudocódigo para o algoritmo de relabel-to-front supõe que as listas de vizinhos  $N[u]$  já foram criadas para cada vértice  $u$ . Ele também pressupõe que  $\text{próximo}[u]$  aponta para o vértice que segue  $u$  na lista  $L$  e que, como de hábito,  $\text{próximo}[u] = \text{NIL}$  se  $u$  é o último vértice na lista.

**RELABEL-TO-FRONT( $G, s, t$ )**

- 1 **INITIALIZE-PREFLOW( $G, s$ )**
- 2  $L \leftarrow V[G] - \{s, t\}$ , em qualquer ordem
- 3 **for** cada vértice  $u \in V[G] - \{s, t\}$
- 4   **do**  $\text{atual}[u] \leftarrow \text{início}[N[u]]$
- 5    $u \leftarrow \text{início}[L]$
- 6 **while**  $u \neq \text{NIL}$
- 7   **do**  $\text{altura-antiga} \leftarrow b[u]$
- 8    **DISCHARGE( $u$ )**

```

9   if  $b[u] > altura\text{-antiga}$ 
10  then mover  $u$  para a frente da lista  $L$ 
11   $u \leftarrow próximo[u]$ 

```

O algoritmo de relabel-to-front funciona da maneira descrita a seguir. A linha 1 inicializa o pré-fluxo e as alturas para os mesmos valores que no algoritmo genérico de push-relabel. A linha 2 inicializa a lista  $L$  para conter todos os vértices potencialmente de transbordamento, em qualquer ordem. As linhas 3 e 4 inicializam o ponteiro *atual* de cada vértice  $u$  como o primeiro vértice na lista de vizinhos de  $u$ .

Como mostra a Figura 26.10, o loop **while** das linhas 6 a 11 percorre a lista  $L$ , descarregando os vértices. A linha 5 faz com que ele comece no primeiro vértice da lista. A cada passagem pelo loop, um vértice  $u$  é descarregado na linha 8. Se  $u$  foi elevado pelo procedimento DISCHARGE, a linha 10 o desloca para a frente da lista  $L$ . Essa determinação é feita gravando-se a altura de  $u$  na variável *altura-antiga* antes da operação de descarga (linha 7) e comparando-se essa altura gravada com a altura de  $u$  daí em diante (linha 9). A linha 11 faz a próxima iteração do loop **while** usar o vértice que segue  $u$  na lista  $L$ . Se  $u$  foi movido para a frente da lista, o vértice usado na próxima iteração é aquele que segue  $u$  em sua nova posição na lista.

Para demonstrar que RELABEL-TO-FRONT calcula um fluxo máximo, mostraremos que ele é uma implementação do algoritmo genérico de push-relabel. Primeiro, observe que ele só executa operações de elevação e empurrão quando elas se aplicam, pois o Lema 26.30 garante que DISCHARGE somente executará essas operações quando elas se aplicarem. Resta mostrar que, quando RELABEL-TO-FRONT termina, nenhuma operação básica se aplica. O restante do argumento de correção se baseia no loop invariante a seguir:

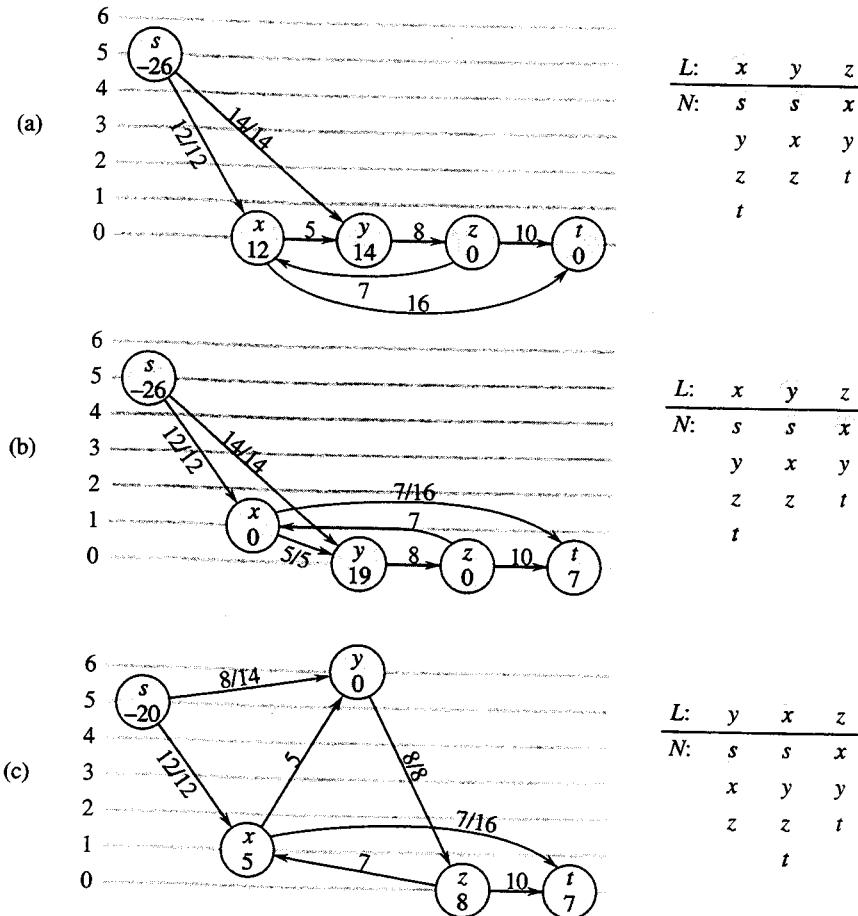
Em cada teste na linha 6 de RELABEL-TO-FRONT, a lista  $L$  é uma ordenação topológica dos vértices na rede admissível  $G_{f,b} = (V, E_{f,b})$ , e nenhum vértice antes de  $u$  na lista tem fluxo em excesso.

**Inicialização:** Imediatamente depois de INITIALIZE-PREFLOW ter sido executado,  $b[s] = |V|$  e  $b[v] = 0$  para todo  $v \in V - \{s\}$ . Tendo em vista que  $|V| \geq 2$  (porque  $V$  contém pelo menos  $s$  e  $t$ ), nenhuma aresta pode ser admissível. Desse modo,  $E_{f,b} = \emptyset$ , e qualquer ordenação de  $V - \{s, t\}$  é uma ordenação topológica de  $G_{f,b}$ .

Tendo em vista que  $u$  é inicialmente o início da lista  $L$ , não existe nenhum vértice antes dele, e assim não há nenhum antes dele com fluxo em excesso.

**Manutenção:** Para ver que a ordenação topológica é mantida por cada iteração do loop **while**, começamos observando que a rede admissível é alterada apenas por operações de empurrão e elevação. Pelo Lema 26.28, as operações de empurrão não fazem as arestas se tornarem admissíveis. Desse modo, as arestas admissíveis podem ser criadas somente por operações de elevação. Contudo, depois de um vértice  $u$  ser elevado, o Lema 26.29 declara que não existe nenhuma aresta admissível entrando em  $u$ , mas pode haver arestas admissíveis saindo de  $u$ . Desse modo, movendo  $u$  para a frente de  $L$ , o algoritmo assegura que quaisquer arestas admissíveis que saem de  $u$  satisfazem à ordenação topológica.

Para ver que nenhum vértice que precede  $u$  em  $L$  tem fluxo em excesso, denotamos o vértice que será  $u$  na próxima iteração por  $u'$ . Os vértices que precederão  $u'$  na próxima iteração incluem o  $u$  atual (devido à linha 11) e nenhum outro vértice (se  $u$  é elevado) ou os mesmos vértices de antes (se  $u$  não é elevado). Tendo em vista que  $u$  é descarregado, ele não tem nenhum fluxo em excesso depois disso. Portanto, se  $u$  for elevado durante a descarga, nenhum vértice que precede  $u'$  terá fluxo em excesso. Se  $u$  não é elevado durante a descarga, nenhum vértice antes dele na lista adquiriu fluxo em excesso durante essa descarga, porque  $L$  permaneceu ordenado logicamente em todos os momentos durante a descarga (conforme destacamos antes, arestas admissíveis são criadas apenas por elevação, e não por empurrão), e assim cada operação de empurrão faz o fluxo em excesso se mover apenas para vértices mais abaixo na lista (ou para  $s$  ou  $t$ ). Novamente, nenhum vértice que precede  $u'$  tem fluxo em excesso.



**FIGURA 26.10** A ação de RELABEL-TO-FRONT. (a) Um fluxo em rede imediatamente antes da primeira iteração do loop **while**. No início, 26 unidades de fluxo deixam a origem  $s$ . No lado direito é mostrada a lista inicial  $L = \langle x, y, z \rangle$ , onde inicialmente  $u = x$ . Sob cada vértice na lista  $L$  encontra-se sua lista de vizinhos, com o vizinho atual sombreado. O vértice  $x$  é descarregado. Ele é elevado até a altura 1, 5 unidades de fluxo em excesso são empurradas até  $y$ , e as 7 unidades restantes em excesso são empurradas para o sorvedor  $t$ . Pelo fato de  $x$  ser elevado, ele se moverá para o início de  $L$ , o que nesse caso não mudará a estrutura de  $L$ . (b) Depois de  $x$ , o próximo vértice em  $L$  a ser descarregado é  $y$ . A Figura 26.9 mostra a ação detalhada de descarregar  $y$  nessa situação. Pelo fato de ser elevado, ele é movido para o início de  $L$ . (c) O vértice  $x$  agora segue  $y$  em  $L$ , e assim ele é novamente descarregado, empurrando todas as 5 unidades de fluxo em excesso para  $t$ . Como o vértice  $x$  não é elevado nessa operação de descarga, ele permanece em seu lugar na lista  $L$ . (d) Tendo em vista que o vértice  $z$  segue o vértice  $x$  em  $L$ , ele é descarregado. O vértice é elevado até a altura 1 e todas as 8 unidades de fluxo em excesso são empurrados para  $t$ . Como  $z$  é elevado, ele é movido para a frente de  $L$ . (e) O vértice  $y$  agora segue o vértice  $z$  em  $L$  e é então descarregado. Porém, pelo fato de  $y$  não ter nenhum excesso, DISCHARGE retorna de imediato, e  $y$  permanece em seu lugar em  $L$ . O vértice  $x$  é então descarregado. Considerando-se que ele também não tem nenhum excesso, DISCHARGE retorna mais uma vez, e  $x$  permanece em seu lugar em  $L$ . RELABEL-TO-FRONT alcança o fim da lista  $L$  e se encerra. Não existe nenhum vértice de transbordamento, e o pré-fluxo é um fluxo máximo.

**Término:** Quando o loop termina,  $u$  está imediatamente além do fim de  $L$ , e então o loop invariante assegura que o excesso de todo vértice é 0. Desse modo, nenhuma operação básica se aplica.

## Análise

Agora, mostraremos que RELABEL-TO-FRONT é executado no tempo  $O(V^3)$  sobre qualquer fluxo em rede  $G = (V, E)$ . Tendo em vista que o algoritmo é uma implementação do algoritmo ge-

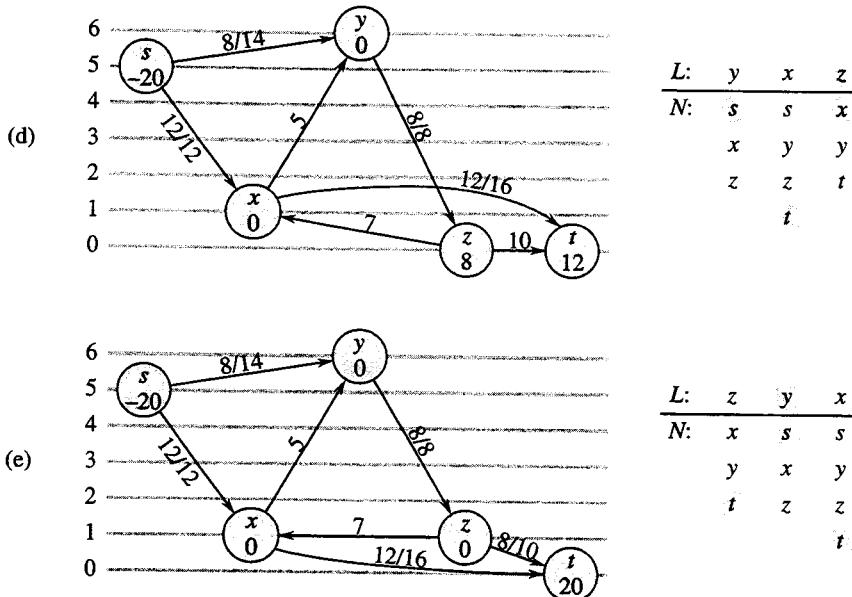


FIGURA 26.10 Continuação

nérico de push-relabel, aproveitaremos o Corolário 26.22, que fornece um limite  $O(V)$  sobre o número de operações de elevação executadas por vértice, e um limite  $O(V^2)$  sobre o número total de elevações globais. Além disso, o Exercício 26.4-2 fornece um limite  $O(VE)$  sobre o tempo total gasto na execução de operações de elevação, e o Lema 26.23 fornece um limite  $O(VE)$  sobre o número total de operações de empurrões saturantes.

### Teorema 26.31

O tempo de execução de RELABEL-TO-FRONT sobre qualquer fluxo em rede  $G = (V, E)$  é  $O(V^3)$ .

**Prova** Vamos admitir como uma “fase” do algoritmo de relabel-to-front o tempo entre duas operações de elevação consecutivas. Existem  $O(V^2)$  fases, pois há  $O(V^2)$  operações de elevação. Cada fase consiste em no máximo  $|V|$  chamadas a DISCHARGE, o que podemos ver a seguir. Se DISCHARGE não executar uma operação de elevação, a próxima chamada a DISCHARGE descerá ainda mais na lista  $L$ , e o comprimento de  $L$  será menor que  $|V|$ . Se DISCHARGE executar uma elevação, a próxima chamada a DISCHARGE pertencerá a uma fase diferente. Tendo em vista que cada fase contém no máximo  $|V|$  chamadas a DISCHARGE, e existem  $O(V^2)$  fases, o número de vezes que DISCHARGE será chamado na linha 8 de RELABEL-TO-FRONT será  $O(V^3)$ . Desse modo, o trabalho total executado pelo loop `while` em RELABEL-TO-FRONT, excluindo-se o trabalho executado dentro de DISCHARGE, será no máximo  $O(V^3)$ .

Devemos agora limitar o trabalho executado dentro de DISCHARGE durante a execução do algoritmo. Cada iteração do loop `while` dentro de DISCHARGE executa uma dentre três ações. Analisaremos a quantidade total de trabalho envolvido na execução de cada uma dessas ações.

Começamos com operações de elevação (linhas 4 e 5). O Exercício 26.4-2 fornece um limite de tempo  $O(VE)$  sobre todas as  $O(V^2)$  elevações executadas.

Agora, suponha que a ação atualize o ponteiro `atual[u]` na linha 8. Essa ação ocorre  $O(\text{grau}(u))$  vezes sempre que o vértice  $u$  é elevado, e  $O(V \cdot \text{grau}(u))$  vezes de modo global para o vértice. Assim, para todos os vértices, a quantidade total de trabalho feito para avançar ponteiros em listas de vizinhos é  $O(VE)$  pelo lema do aperto de mão (Exercício B.4-1).

O terceiro tipo de ação executada por DISCHARGE é uma operação de empurrão (linha 7). Já sabemos que o número total de operações de empurrão saturante é  $O(VE)$ . Observe que, se um empurrão não saturante é executado, DISCHARGE retorna imediatamente, pois o empurrão reduz o excesso para 0. Desse modo, pode existir no máximo um empurrão não saturante por chamada a DISCHARGE. Como observamos, DISCHARGE é chamado  $O(V^3)$  vezes, e portanto o tempo total gasto na execução de empurrões não saturantes é  $O(V^3)$ .

O tempo de execução de RELABEL-TO-FRONT é então  $O(V^3 + VE)$ , que é  $O(V^3)$ . ■

## Exercícios

### 26.5-1

Ilustre a execução de RELABEL-TO-FRONT da maneira descrita na Figura 26.10 para o fluxo em rede da Figura 26.1(a). Suponha que a ordenação inicial de vértices em  $L$  seja  $\langle v_1, v_2, v_3, v_4 \rangle$  e que as listas de vizinhos sejam

$$\begin{aligned}N[v_1] &= \langle s, v_2, v_3 \rangle, \\N[v_2] &= \langle s, v_1, v_3, v_4 \rangle, \\N[v_3] &= \langle v_1, v_2, v_4, t \rangle, \\N[v_4] &= \langle v_2, v_3, t \rangle.\end{aligned}$$

### 26.5-2 \*

Gostaríamos de implementar um algoritmo de push-relabel no qual mantivéssemos uma fila do tipo primeiro a entrar, primeiro a sair de vértices transbordantes. O algoritmo descarrega repetidamente o vértice no início da fila, e quaisquer vértices que não eram transbordantes antes da descarga mas que são transbordantes após a descarga são colocados no final da fila. Depois que o vértice no início da fila é descarregado, ele é removido. Quando a fila está vazia, o algoritmo termina. Mostre que esse algoritmo pode ser implementado para calcular um fluxo máximo no tempo  $O(V^3)$ .

### 26.5-3

Mostre que o algoritmo genérico ainda funciona se RELABEL atualiza  $b[u]$  simplesmente calculando  $b[u] \leftarrow b[u] + 1$ . De que modo essa mudança afetaria a análise de RELABEL-TO-FRONT?

### 26.5-4 \*

Mostre que, se sempre descarregamos um vértice transbordante mais alto, o método de push-relabel pode ser executado no tempo  $O(V^3)$ .

### 26.5-5

Suponha que, em algum ponto na execução de um algoritmo de push-relabel, existe um inteiro  $0 < k \leq |V| - 1$  para o qual nenhum vértice tem  $b[v] = k$ . Mostre que todos os vértices com  $b[v] > k$  estão no lado de origem de um corte mínimo. Se tal  $k$  existir, a **heurística de lacuna** atualiza todo vértice  $v \in V - s$  para o qual  $b[v] > k$  para definir  $b[v] \leftarrow \max(b[v], |V| + 1)$ . Mostre que o atributo resultante  $b$  é uma função de altura. (A heurística de lacuna é crucial para fazer as implementações do método push-relabel funcionarem bem na prática.)

## Problemas

### 26-1 O problema de escape

Uma *grade*  $n \times n$  é um grafo não orientado que consiste em  $n$  linhas e  $n$  colunas de vértices, como mostra a Figura 26.12. Denotamos o vértice na  $i$ -ésima linha e na  $j$ -ésima coluna por  $(i, j)$ . Todos os vértices em uma grade têm exatamente quatro vizinhos, com exceção dos vértices dos limites, que são os pontos  $(i, j)$  para os quais  $i = 1$ ,  $i = n$ ,  $j = 1$  ou  $j = n$ .

Dados  $m \leq n^2$  pontos de partida  $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$  na grade, o **problema de escape** consiste em determinar se existem ou não  $m$  caminhos disjuntos de vértices desde os pontos de partida até quaisquer  $m$  pontos diferentes no limite. Por exemplo, a grade na Figura 26.11(a) tem um escape, mas a grade na Figura 26.11(b) não tem.

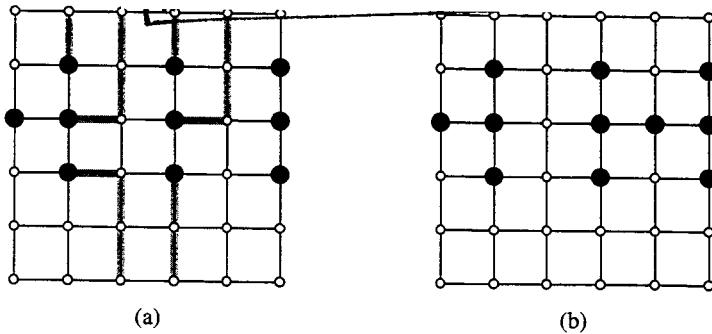


FIGURA 26.11 Grades para o problema de escape. Os pontos de partida são pretos, e os outros vértices da grade são brancos. (a) Uma grade com um escape, mostrado por caminhos sombreados. (b) Uma grade sem escape

- Considere um fluxo em rede no qual os vértices, como também as arestas, tenham capacidades. Isto é, o fluxo total positivo que entra em qualquer vértice dado está sujeito a uma restrição de capacidade. Mostre que a determinação do fluxo máximo em uma rede com capacidades de arestas e vértices pode ser reduzida a um problema de fluxo máximo comum em um fluxo em rede de tamanho comparável.
- Descreva um algoritmo eficiente para resolver o problema de escape e analise seu tempo de execução.

### 26-2 Cobertura de caminho mínima

Uma **cobertura de caminho** de um grafo orientado  $G = (V, E)$  é um conjunto  $P$  de caminhos disjuntos de vértices tais que todo vértice em  $V$  está incluído em exatamente um caminho em  $P$ . Os caminhos podem começar e terminar em qualquer lugar, e podem ser de qualquer comprimento, inclusive 0.

Uma **cobertura de caminho mínima** de  $G$  é uma cobertura de caminho contendo o número mínimo possível de caminhos.

- Forneça um algoritmo eficiente para encontrar uma cobertura de caminho mínima de um grafo acíclico orientado  $G = (V, E)$ . (Sugestão: Supondo que  $V = \{1, 2, \dots, n\}$ , construa o grafo  $G' = (V, E')$ , onde

$$V' = \{x_0, x_1, \dots, x_n\} \cup \{y_0, y_1, \dots, y_n\}, \\ E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_i) : (i, j) \in E\},$$

e execute um algoritmo de fluxo máximo.)

- Seu algoritmo funciona para grafos orientados que contêm ciclos? Explique.

### 26-3 As experiências do ônibus espacial

O professor Spock presta consultoria à NASA, que está planejando uma série de vôos do ônibus espacial e tem de decidir quais experiências comerciais serão executadas e quais instrumentos deverão estar a bordo em cada vôo. Para cada vôo, a NASA considera um conjunto  $E = \{E_1, E_2, \dots, E_m\}$  de experimentos, e o patrocinador comercial do experimento  $E_j$  concordou em pagar à NASA  $p_j$  dólares pelos resultados do experimento. Os experimentos utilizam um conjunto  $I = \{I_1, I_2, \dots, I_n\}$  de instrumentos; cada experiência  $E_j$  exige todos os instrumentos de um subconjunto  $R_j \subseteq I$ . O custo de levar o instrumento  $I_k$  é  $c_k$  dólares. O trabalho do professor é encontrar um algoritmo eficiente para determinar que experimentos executar e que instrumentos levar em um determinado vôo, a fim de maximizar o lucro líquido, que é a renda total das experiências executadas menos o custo total de todos os instrumentos transportados.

Considere a rede  $G$  a seguir. A rede contém um vértice de origem  $s$ , vértices  $I_1, I_2, \dots, I_n$ , vértices  $E_1, E_2, \dots, E_m$  e um vértice de sorvedor  $t$ . Para  $k = 1, 2, \dots, n$ , existe uma aresta  $(s, I_k)$  de capacidade  $c_k$  e, para  $j = 1, 2, \dots, m$ , existe uma aresta  $(E_j, t)$  de capacidade  $p_j$ . Para  $k = 1, 2, \dots, n$  e  $j = 1, 2, \dots, m$ , se  $I_k \in R_j$ , então existe uma aresta  $(I_k, E_j)$  de capacidade infinita.

- Mostre que, se  $E_j \in T$  para um corte de capacidade finita  $(S, T)$  de  $G$ ,  $I_k \in T$  para cada  $I_k \in R_j$ .
- Mostre como determinar o lucro líquido máximo a partir da capacidade do corte mínimo de  $G$  e dos valores de  $p_j$  dados.
- Forneça um algoritmo eficiente para determinar que experimentos executar e que instrumentos levar. Analise o tempo de execução de seu algoritmo em termos de  $m$ ,  $n$  e  $r = \sum_{j=1}^m |R_j|$ .

#### 26-4 Atualização do fluxo máximo

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$ , sorvedor  $t$  e capacidades inteiras. Vamos supor que recebemos um fluxo máximo em  $G$ .

- Suponha que a capacidade de uma aresta única  $(u, v) \in E$  seja aumentada por 1. Forneça um algoritmo de tempo  $O(V + E)$  para atualizar o fluxo máximo.
- Suponha que a capacidade de uma aresta única  $(u, v) \in E$  seja diminuída de 1. Forneça um algoritmo de tempo  $O(V + E)$  para atualizar o fluxo máximo.

#### 26-5 Fluxo máximo por escalonamento

Seja  $G = (V, E)$  um fluxo em rede com origem  $s$ , sorvedor  $t$  e uma capacidade inteira  $c(u, v)$  em cada aresta  $(u, v) \in E$ . Seja  $C = \max_{(u, v) \in E} c(u, v)$ .

- Demonstre que um corte mínimo de  $G$  tem capacidade no máximo igual a  $C|E|$ .
- Para um dado número  $K$ , mostre que é possível encontrar um caminho aumentante de capacidade pelo menos  $K$  no tempo  $O(E)$ , se tal caminho existir.

A modificação de FORD-FULKERSON-METHOD a seguir pode ser usada para calcular um fluxo máximo em  $G$ .

**MAX-FLOW-BY-SCALING( $G, s, t$ )**

```

1  $C \leftarrow \max_{(u, v) \in E} c(u, v)$ 
2 inicializar fluxo  $f$  como 0
3  $K \leftarrow 2^{\lfloor \lg C \rfloor}$ 
4 while  $K \geq 1$ 
5   do while existir um caminho aumentante  $p$  de capacidade pelo menos igual a  $K$ 
6     do ampliar o fluxo  $f$  ao longo de  $p$ 
7      $K \leftarrow K/2$ 
8 return  $f$ 
```

- Demonstre que MAX-FLOW-BY-SCALING retorna um fluxo máximo.
- Mostre que a capacidade de um corte mínimo do grafo residual  $G_f$  é no máximo  $2K|E|$  a cada vez que a linha 4 é executada.
- Demonstre que o loop while interno das linhas 5 e 6 é executado  $O(E)$  vezes para cada valor de  $K$ .
- Conclua que MAX-FLOW-BY-SCALING pode ser implementado para execução no tempo  $O(E^2 \lg C)$ .

#### 26-6 Fluxo máximo com capacidades negativas

Suponha que permitimos a um fluxo em rede ter capacidades de arestas negativas (como também positivas). Em tal rede, um fluxo possível não precisa existir.

- a. Considere uma aresta  $(u, v)$  em um fluxo em rede  $G = (V, E)$  com  $c(u, v) < 0$ . Explique brevemente o que significa tal capacidade negativa em termos do fluxo entre  $u$  e  $v$ .

Seja  $G = (V, E)$  um fluxo em rede com capacidades de arestas negativas, e sejam  $s$  e  $t$  ser a origem e o sorvedor de  $G$ . Construa o fluxo em rede comum  $G' = (V', E')$  com função de capacidade  $c'$ , origem  $s'$  e sorvedor  $t'$ , onde

$$V' = V \cup \{s', t'\}$$

e

$$\begin{aligned} E' = E &\cup \{(u, v) : (v, u) \in E\} \\ &\cup \{(s', v) : v \in V\} \\ &\cup \{(u, t') : u \in V\} \\ &\cup \{(s, t) : (t, s)\}. \end{aligned}$$

Atribuímos capacidades às arestas como a seguir. Para cada aresta  $(u, v) \in E$ , definimos

$$c'(u, v) = c'(v, u) = (c(u, v) + c(v, u))/2$$

Para cada vértice  $u \in V$ , definimos

$$c'(s', u) = \max(0, (c(V, u) - c(u, V))/2$$

e

$$c'(s', t) = \max(0, (c(V, u) - c(u, V))/2$$

Também definimos  $c'(s, t) = c'(t, s) = \infty$ .

- b. Prove que, se existir um fluxo possível em  $G$ , então todas as capacidades em  $G'$  serão não negativas e existirá um fluxo máximo em  $G'$  tal que todas as arestas no sorvedor  $t'$  serão saturadas.
- c. Prove a recíproca da parte (b). Sua prova deve ser construtiva, isto é, dado um fluxo em  $G'$  que sature todas as arestas em  $t'$ , sua prova deve mostrar como obter um fluxo possível em  $G$ .
- d. Descreva um algoritmo que encontre um fluxo máximo possível em  $G$ . Denote por  $MF(|V|, |E|)$  o tempo de execução no pior caso de um algoritmo de fluxo máximo comum em um grafo com  $|V|$  vértices e  $|E|$  arestas. Analise seu algoritmo para calcular o fluxo máximo de um fluxo em rede com capacidades negativas em termos de  $MF$ .

### 26-7 O algoritmo de emparelhamento bipartido de Hopcroft-Karp

Neste problema, descrevemos um algoritmo mais rápido, devido a Hopcroft e Karp, para encontrar um emparelhamento máximo em um grafo bipartido. O algoritmo funciona no tempo  $O(\sqrt{|V|} |E|)$ . Dado um grafo não orientado bipartido  $G = (V, E)$ , onde  $V = L \cup R$  e todas as arestas têm exatamente uma extremidade em  $L$ , seja  $M$  um emparelhamento em  $G$ . Dizemos que um caminho simples  $P$  em  $G$  é um **caminho aumentante** com respeito a  $M$  se ele começa em um vértice não correspondido em  $L$ , termina em um vértice não correspondido em  $R$  e suas arestas pertencem alternadamente a  $M$  e a  $E - M$ . (Essa definição de um caminho aumentante está relacionada a, embora seja diferente de, um caminho aumentante de um fluxo em rede.) Neste problema, tratamos um caminho como uma seqüência de arestas, em vez de tratá-lo como um emparelhamento de vértices. Um caminho aumentante mais curto com respeito a uma correspondência  $M$  é um caminho aumentante com um número mínimo de arestas.

Dados dois conjuntos  $A$  e  $B$ , a **diferença simétrica**  $A \oplus B$  é definida como  $(A - B) \cup (B - A)$ , isto é, os elementos que estão em exatamente um dos dois conjuntos.

- a. Mostre que, se  $M$  é um emparelhamento e  $P$  é um caminho aumentante com relação a  $M$ , então a diferença simétrica  $M \oplus P$  é um emparelhamento e  $|M| \oplus |P| = |M| + 1$ . Mostre que, se  $P_1, P_2, \dots, P_k$  são caminhos em ampliação de vértices disjuntos em relação a  $M$ , então a diferença simétrica  $M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$  é um emparelhamento com cardinalidade  $|M| + k$ .

A estrutura geral de nosso algoritmo é a seguinte:

HOPCROFT-KARP( $G$ )

```

1  $M \leftarrow \emptyset$ 
2 repeat
3   seja  $\mathcal{P} \leftarrow \{P_1, P_2, \dots, P_k\}$  um conjunto máximo de
      caminhos mais curtos em ampliação de vértices disjuntos
      em relação a  $M$ 
4    $M \leftarrow M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ 
5 until  $\mathcal{P} = \emptyset$ 
6 return  $M$ 
```

O restante deste problema lhe pede para analisar o número de iterações no algoritmo (isto é, o número de iterações no loop **repeat**) e descrever uma implementação da linha 3.

- b. Dados dois emparelhamentos  $M$  e  $M^*$  em  $G$ , mostre que todo vértice no grafo  $G' = (V, M \oplus M^*)$  tem grau máximo 2. Conclua que  $G'$  é uma união disjunta de caminhos ou ciclos simples. Demonstre que arestas em cada caminho ou ciclo simples pertencem alternadamente a  $M$  ou  $M^*$ . Prove que, se  $|M| \leq |M^*|$ , então  $M \oplus M^*$  contém pelo menos  $|M^*| - |M|$  caminhos em ampliação de vértices disjuntos em relação a  $M$ .

Seja  $l$  o comprimento de um caminho aumentante mais curto em relação a um emparelhamento  $M$ , e seja  $P_1, P_2, \dots, P_k$  um conjunto máximo de caminhos aumentantes de vértices disjuntos de comprimento  $l$  em relação a  $M$ . Seja  $M' = M \oplus (P_1 \cup \dots \cup P_k)$  e suponha que  $P$  seja um caminho mais curto em ampliação em relação a  $M'$ .

- c. Mostre que, se  $P$  é de vértices disjuntos de  $P_1, P_2, \dots, P_k$ , então  $P$  tem mais de  $l$  arestas.
- d. Agora, suponha que  $P$  não seja de vértices disjuntos de  $P_1, P_2, \dots, P_k$ . Seja  $A$  o conjunto de arestas  $(M \oplus M') \oplus P$ . Mostre que  $A = (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P$  e que  $|A| \geq (k+1)l$ . Conclua que  $P$  tem mais de  $l$  arestas.
- e. Prove que, se um caminho mais curto em ampliação para  $M$  tem comprimento  $l$ , o tamanho do emparelhamento máximo é no máximo  $|M| + |V|/l$ .
- f. Mostre que o número de iterações do loop **repeat** no algoritmo é no máximo  $2\sqrt{V}$ . (Sugestão: O quanto  $M$  pode crescer após a iteração número  $\sqrt{V}$ ?)
- g. Forneça um algoritmo que funcione no tempo  $O(E)$  para encontrar um conjunto máximo de caminhos mais curtos em ampliação de vértices disjuntos  $P_1, P_2, \dots, P_k$  para um dado emparelhamento  $M$ . Conclua que o tempo de execução total de HOPCROFT-KARP é  $O(\sqrt{V}E)$ .

## Notas do capítulo

Ahuja, Magnanti e Orlin [7], Even [87], Lawler [196], Papadimitriou e Steiglitz [237] e ainda Tarjan [292] são boas referências para fluxo em rede e algoritmos relacionados. Goldberg, Tardos e Tarjan [119] também fornecem uma ótima pesquisa de algoritmos para problemas de fluxo em

rede, e Schrijver [267] escreveu uma resenha interessante sobre os desenvolvimentos históricos no campo de fluxos em rede.

O método de Ford-Fulkerson se deve a Ford e a Fulkerson [93], que iniciaram o estudo formal de muitos dos problemas na área de fluxo em rede, inclusive os problemas do fluxo máximo e de emparelhamento bipartido. Muitas implementações iniciais do método de Ford-Fulkerson encontravam caminhos em ampliação usando a busca em largura; Edmonds e Karp [86] e, independentemente, Dinic [76], provaram que essa estratégia produz um algoritmo de tempo polinomial. Uma idéia relacionada, a de se utilizarem “fluxos de bloqueio”, também foi desenvolvida primeiro por Dinic [76]. Karzanov [176] foi o primeiro a desenvolver a idéia de pré-fluxos. O método de push-relabel se deve a Goldberg [117] e a Goldberg e Tarjan [121]. Goldberg e Tarjan apresentaram um algoritmo de tempo  $O(V^3)$  que emprega uma fila para manter o conjunto de vértices transbordantes, como também um algoritmo que usa árvores dinâmicas para alcançar um tempo de execução de  $O(VE \lg (V^2/E + 2))$ . Vários outros pesquisadores desenvolveram algoritmos de push-relabel de fluxo máximo. Ahuja e Orlin [9] e Ahuja, Orlin e Tarjan [10] forneceram algoritmos que usavam escalonamento. Cherian e Maheshwari [55] propuseram empurrar o fluxo a partir do vértice transbordante de altura máxima. Cherian e Hagerup [54] sugeriram permutar aleatoriamente a lista de vizinhos, e vários pesquisadores [14, 178, 241] desenvolveram “desaleatoriedades” inteligentes dessa idéia, levando a uma sequência de algoritmos mais rápidos. O algoritmo de King, Rao e Tarjan [178] é o mais rápido desses algoritmos e funciona no tempo  $O(VE \log_{E/(V \lg V)} V)$ .

O algoritmo assintoticamente mais rápido criado até hoje para o problema de fluxo máximo é devido a Goldberg e Rao [120] e funciona no tempo  $O(\min(V^{2/3}, E^{1/2}) E \lg(V^2/E + 2) \lg C)$ , onde  $C = \max_{(u, v) \in E} c(u, v)$ . Esse algoritmo não emprega o método de push-relabel; em vez disso, é baseado na localização de fluxos de bloqueio. Todos os algoritmos de fluxo máximo anteriores, inclusive os que examinamos neste capítulo, usam alguma noção de distância (os algoritmos de push-relabel utilizam a noção análoga de altura), com o comprimento 1 atribuído implicitamente a cada aresta. Esse novo algoritmo adota uma abordagem diferente e atribui o comprimento 0 a arestas de capacidade alta e o comprimento 1 a arestas de capacidade baixa. Informalmente, em relação a esses comprimentos, caminhos mais curtos da origem até o sorvedor tendem a ter capacidade alta, o que significa que menos iterações precisam ser executadas.

Na prática, os algoritmos de push-relabel dominam atualmente os algoritmos de caminho aumentante ou os algoritmos baseados em programação linear para o problema de fluxo máximo. Um estudo de Cherkassky e Goldberg [56] sublinha a importância de se usarem duas heurísticas quando se implementa um algoritmo de push-relabel. A primeira heurística é para executar periodicamente uma busca em largura do grafo residual, a fim de obter valores de altura mais precisa. A segunda heurística é a heurística de intervalos, descrita no Exercício 26.5-5. Eles concluem que a melhor escolha de variantes de push-relabel é a que opta por descarregar o vértice transbordante com a altura máxima.

O melhor algoritmo criado até hoje para emparelhamento bipartido máximo, descoberto por Hopcroft e Karp [152], é executado no tempo  $O(\sqrt{V}E)$  e é descrito no Problema 26-7. O livro de Lovász e Plummer [207] é uma excelente referência sobre problemas de correspondência.



---

## *Parte VII*

# *Tópicos selecionados*

### **Introdução**

Esta parte contém uma seleção de tópicos de algoritmos que estendem e complementam o material examinado anteriormente neste livro. Alguns capítulos introduzem novos modelos de computação, como circuitos combinacionais ou computadores paralelos. Outros abrangem domínios especializados, tais como a geometria computacional ou a teoria dos números. Os dois últimos capítulos discutem algumas das limitações conhecidas para o projeto de algoritmos eficientes e introduzem técnicas para se lidar com essas limitações.

O Capítulo 27 apresenta um modelo paralelo de computação: as redes de comparação. Em termos gerais, uma rede de comparação é um algoritmo que permite que sejam feitas muitas comparações simultaneamente. Esse capítulo mostra como elaborar uma rede de comparação capaz de ordenar  $n$  números no tempo  $O(\lg^2 n)$ .

O Capítulo 28 estuda algoritmos eficientes para operação sobre matrizes. Depois de examinar algumas propriedades básicas de matrizes, ele explora o algoritmo de Strassen, capaz de multiplicar duas matrizes  $n \times n$  no tempo  $O(n^{2.81})$ . Em seguida, ele apresenta dois métodos gerais – a decomposição LU e a decomposição LUP – para resolver equações lineares por eliminação gaussiana no tempo  $O(n^3)$ . Ele também mostra que a inversão de matrizes e a multiplicação de matrizes podem ser executadas com a mesma rapidez. O capítulo conclui mostrando como uma solução aproximada de mínimos quadrados pode ser calculada quando um conjunto de equações lineares não tem nenhuma solução exata.

O Capítulo 29 estuda a programação linear, na qual desejamos maximizar ou minimizar um objetivo, sendo dados recursos limitados e restrições concorrentes. A programação linear surge em uma variedade de áreas práticas de aplicação. Esse capítulo cobre a formulação e a solução de programas lineares. O método de solução abordado é o algoritmo simplex, o mais antigo algoritmo para programação linear. Em contraste com muitos algoritmos deste livro, o algoritmo simplex não funciona em tempo polinomial no pior caso, mas é bastante eficiente e tem ampla utilização na prática.

O Capítulo 30 estuda operações sobre polinômios e mostra que uma técnica bem conhecida de processamento de sinais – a transformação rápida de Fourier (FFT – Fast Fourier Transform) – pode ser usada para multiplicar dois polinômios de grau  $n$  no tempo  $O(n \lg n)$ . Ele também investiga implementações eficientes da FFT, inclusive um circuito paralelo.

O Capítulo 31 apresenta algoritmos de teoria dos números. Depois de uma revisão da teoria elementar dos números, ele apresenta o algoritmo de Euclides para calcular o máximo divisor comum. Os algoritmos para resolver equações lineares modulares e para elevar um número a uma potência de módulo igual a outro número são apresentados em seguida. Depois, veremos uma aplicação importante dos algoritmos de teoria dos números: o sistema criptográfico de chave pública RSA. Esse sistema de criptografia não apenas pode ser usado para criptografar mensagens de modo que um adversário não possa lê-las, mas também pode ser utilizado para produzir assinaturas digitais. O capítulo apresenta então o teste aleatório de números primos de Miller-Rabin, que pode ser usado para encontrar de forma eficiente grandes números primos – um requisito essencial para o sistema RSA. Por fim, o capítulo focaliza a heurística “rho” de Pollard para fatorar inteiros e discute a situação atual da fatoração de números inteiros.

O Capítulo 32 estuda o problema de encontrar todas as ocorrências de uma cadeia padrão específica em uma cadeia de texto dada, um problema que surge com freqüência em programas de edição de textos. Depois de examinar a abordagem simples, o capítulo apresenta uma abordagem elegante que se deve a Rabin e Karp. Em seguida, após examinar uma solução eficiente baseada em autômatos finitos, o capítulo apresenta o algoritmo de Knuth-Morris-Pratt, que alcança eficiência por meio do pré-processamento elegante do padrão.

A geometria computacional é o assunto do Capítulo 33. Depois de discutir primitivas básicas da geometria computacional, o capítulo mostra como um método de “varredura” pode determinar de modo eficiente se um conjunto de segmentos de linhas contém ou não alguma interseção. Dois algoritmos inteligentes para encontrar a envoltória convexa de um conjunto de pontos – o exame de Graham e a marcha de Jarvis – também ilustram a capacidade dos métodos de varredura. O capítulo termina com um algoritmo eficiente para encontrar o par mais próximo entre um dado conjunto de pontos no plano.

O Capítulo 34 estuda os problemas NP-completos. Muitos problemas computacionais interessantes são NP-completos, mas não se conhece nenhum algoritmo de tempo polinomial para resolver qualquer um deles. Esse capítulo apresenta técnicas para determinar quando um problema é NP-completo. Diversos problemas clássicos se mostraram NP-completos: determinar se um grafo tem um ciclo hamiltoniano, determinar se uma fórmula booleana pode ser satisfeita e determinar se um dado conjunto de números tem um subconjunto que se soma a um determinado valor de destino. O capítulo também prova que o famoso problema do caixeiro-viajante é NP-completo.

O Capítulo 35 mostra como os algoritmos de aproximação podem ser usados de forma eficiente com o objetivo de encontrar soluções aproximadas para problemas NP-completos. No caso de alguns problemas NP-completos, soluções aproximadas que são quase ótimas são bastante fáceis de produzir; porém, para outros problemas, até mesmo os melhores algoritmos de aproximação conhecidos funcionam de forma cada vez mais pobre à medida que o tamanho do problema aumenta progressivamente. Então, existem alguns problemas para os quais se pode investir quantidades crescentes de tempo de computação em troca de soluções aproximadas cada vez melhores. Esse capítulo ilustra tais possibilidades com o problema da cobertura de vértices (nas versões não-ponderada e ponderada), uma versão de otimização da possibilidade de satisfação da 3FNC, o problema do caixeiro-viajante, o problema da cobertura de conjuntos e o problema da soma de subconjuntos.

---

## *Capítulo 27*

# *Redes de ordenação*

Na Parte II, examinamos algoritmos de ordenação para computadores seriais (máquinas de acesso aleatório, ou RAMs) que só permitem a execução de uma única operação de cada vez. Neste capítulo, investigaremos os algoritmos de ordenação baseados em um modelo de computação de rede de comparação na qual muitas operações de comparação podem ser executadas simultaneamente.

As redes de comparação diferem das RAMs em dois aspectos importantes. Primeiro, elas só podem executar comparações. Desse modo, um algoritmo como a ordenação por contagem (ver Seção 8.2) não pode ser implementado em uma rede de comparação. Em segundo lugar, diferente do modelo de RAM, no qual as operações ocorrem de modo serial (consecutivo) – isto é, uma após outra – as operações em uma rede de comparação podem ocorrer ao mesmo tempo, ou “em paralelo”. Como veremos, essa característica permite a construção de redes de comparação que ordenam  $n$  valores em tempo sublinear.

Começaremos na Seção 27.1 definindo redes de comparação e redes de ordenação. Também daremos uma definição natural para o “tempo de execução” de uma rede de comparação em termos da profundidade da rede. A Seção 27.2 prova o “princípio zero um” que facilita bastante a tarefa de analisar a correção de redes de ordenação.

A rede de ordenação eficiente que projetaremos é em essência uma versão paralela do algoritmo de ordenação por intercalação da Seção 2.3.1. Nossa construção terá três etapas. A Seção 27.3 apresenta o projeto de um ordenador “bitônico” que será nosso bloco de construção básico. Modificaremos ligeiramente o ordenador bitônico na Seção 27.4 para produzir uma rede de intercalação que possa intercalar duas seqüências ordenadas em uma única seqüência ordenada. Finalmente, na Seção 27.5, montaremos essas redes de intercalação para formar uma rede de ordenação capaz de ordenar  $n$  valores no tempo  $O(\lg^2 n)$ .

### **27.1 Redes de comparação**

As redes de ordenação são redes de comparação que sempre ordenam suas entradas, e assim faz sentido começar nossa discussão com as redes de comparação e suas características. Uma rede de comparação é formada unicamente de fios e comparadores. Um **comparador**, mostrado na Figura 27.1(a), é um dispositivo com duas entradas,  $x$  e  $y$ , e duas saídas,  $x'$  e  $y'$ , que executa a função a seguir:

$$x' = \min(x, y),$$

$$y' = \max(x, y).$$

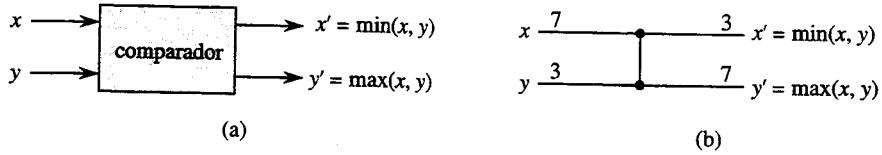


FIGURA 27.1 (a) Um comparador com entradas  $x$  e  $y$  e saídas  $x'$  e  $y'$ . (b) O mesmo comparador, desenhando como uma única linha vertical. As entradas  $x = 7$ ,  $y = 3$  e as saídas  $x' = 3$ ,  $y' = 7$  são mostradas

Pelo fato de a representação pictórica de um comparador na Figura 27.1(a) ser muito volumosa para nossos propósitos, adotaremos a convenção de desenhar comparadores como linhas verticais únicas, como mostra a Figura 27.1(b). As entradas aparecem à esquerda e as saídas à direita, com o menor valor de entrada aparecendo na saída superior e o maior valor de entrada aparecendo na saída inferior. Podemos assim imaginar um comparador como um dispositivo que ordena suas duas entradas.

Partiremos do princípio de que cada comparador opera no tempo  $O(1)$ . Em outras palavras, supomos que o tempo entre o aparecimento dos valores de entrada  $x$  e  $y$  e a produção dos valores de saída  $x'$  e  $y'$  é uma constante.

Um **fio** transmite um valor de um lugar para outro. Os fios podem conectar a saída de um comparador à entrada de outro mas, em caso contrário, eles são fios de entrada de rede ou fios de saída de rede. Ao longo deste capítulo, faremos a suposição de que uma rede de comparação contém  $n$  fios de entrada  $a_1, a_2, \dots, a_n$ , através dos quais os valores a serem ordenados entram na rede, e  $n$  fios de saída  $b_1, b_2, \dots, b_n$ , que produzem os resultados calculados pela rede. Além disso, mencionaremos a **seqüência de entrada**  $\langle a_1, a_2, \dots, a_n \rangle$  e a **seqüência de saída**  $\langle b_1, b_2, \dots, b_n \rangle$ , que se referem aos valores contidos nos fios de entrada e de saída. Ou seja, empregaremos o mesmo nome para um fio e para o valor que ele transporta. Nossa intenção sempre será clara a partir do contexto.

A Figura 27.2 mostra uma **rede de comparação**, que é um conjunto de comparadores interconectados por fios. Desenharemos uma rede de comparação sobre  $n$  entradas como uma coleção de  $n$  linhas horizontais com comparadores estendidos verticalmente. Observe que uma linha *não* representa um único fio, mas sim uma seqüência de fios distintos que conectam diversos comparadores. Por exemplo, a linha superior na Figura 27.2 representa três fios: o fio de entrada  $a_1$ , que se conecta a uma entrada do comparador  $A$ ; um fio que conecta a saída superior do comparador  $A$  a uma entrada do comparador  $C$ ; e um fio de saída  $b_1$ , que vem da saída superior do comparador  $C$ . A entrada de cada comparador é conectada a um fio que é um dos  $n$  fios de entrada  $a_1, a_2, \dots, a_n$  da rede ou está conectado à saída de outro comparador. De modo semelhante, a saída de cada comparador está conectada a um fio que é um dos  $n$  fios  $b_1, b_2, \dots, b_n$  de saída da rede ou está conectado à entrada de outro comparador. O principal requisito para interconectar comparadores é que o grafo de interconexões tem de ser acíclico: se traçarmos um caminho desde a saída de um dado comparador até a entrada de outro e assim sucessivamente, o caminho que traçarmos nunca deverá formar um ciclo de retorno sobre si mesmo e passar duas vezes pelo mesmo comparador. Desse modo, como na Figura 27.2, podemos desenhar uma rede de comparação com entradas de rede no lado esquerdo e saídas de rede no lado direito; os dados se deslocam pela rede da esquerda para a direita.

Cada comparador produz seus valores de saída somente quando ambos os valores de entrada estão disponíveis para ele. Por exemplo, na Figura 27.2(a), suponha que a seqüência  $\langle 9, 5, 2, 6 \rangle$  apareça nos fios de entrada no instante 0. Então, no instante 0, somente os comparadores  $A$  e  $B$  têm todos os seus valores de entrada disponíveis. Supondo que cada comparador exija uma unidade de tempo para calcular seus valores de saída, os comparadores  $A$  e  $B$  produzem suas saídas no instante 1; os valores resultantes são mostrados na Figura 27.2(b). Observe que os comparadores  $A$  e  $B$  produzem seus valores ao mesmo tempo, ou “em paralelo”. Agora, no instante 1, os comparadores  $C$  e  $D$ , mas não  $E$ , têm todos os seus valores de entrada disponíveis. Uma unidade de tempo mais tarde, no instante 2, eles produzem suas saídas, como mostra a Figura

27.2(c). Os comparadores  $C$  e  $D$  também operam em paralelo. A saída superior do comparador  $C$  e a saída inferior do comparador  $D$  se conectam aos fios de saída  $b_1$  e  $b_4$ , respectivamente, da rede de comparação, e esses fios de saída da rede transportam portanto seus valores finais no instante 2. Enquanto isso, no instante 2, o comparador  $E$  tem suas saídas disponíveis, e a Figura 27.2(d) mostra que ele produz seus valores de saída no instante 3. Esses valores são transportados pelos fios de saída de rede  $b_2$  e  $b_3$ , e agora a seqüência de saída  $\langle 2, 5, 6, 9 \rangle$  se completa.

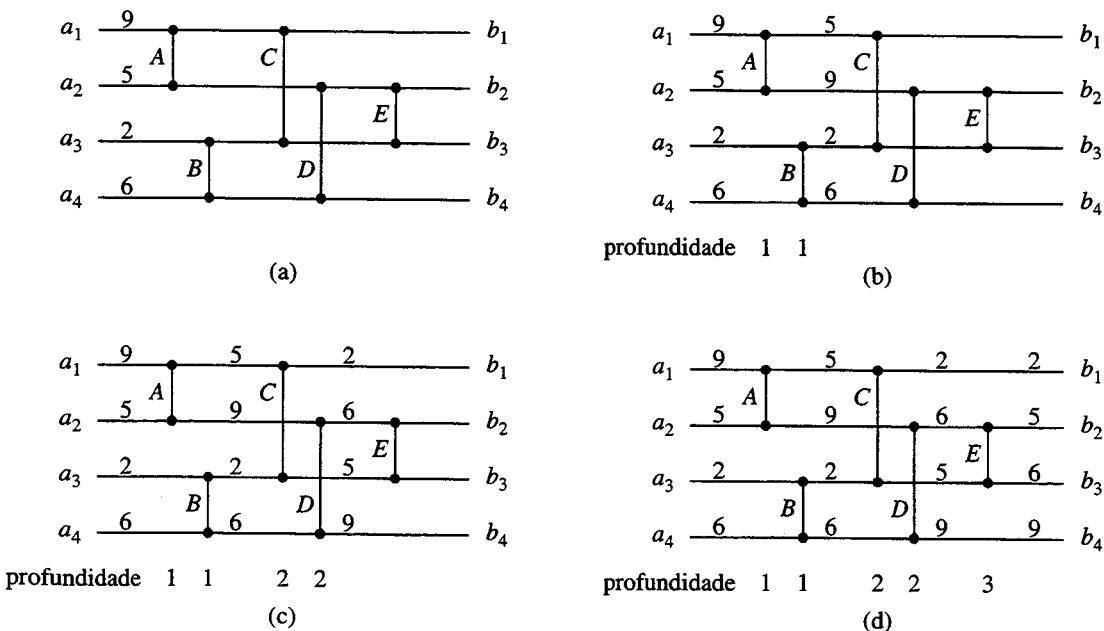
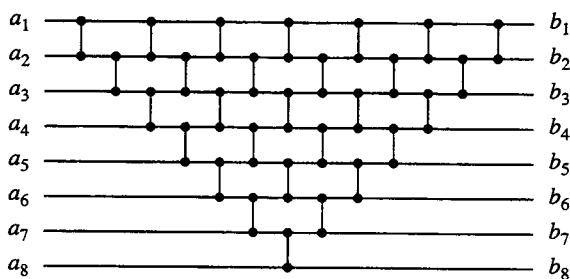


FIGURA 27.2 (a) Uma rede de comparação de 4 entradas e 4 saídas, que é de fato uma rede de ordenação. No instante 0, os valores de entrada mostrados aparecem nos quatro fios de entrada. (b) No instante 1, os valores mostrados aparecem nas saídas dos comparadores  $A$  e  $B$ , que estão na profundidade 1. (c) No instante 2, os valores mostrados aparecem nas saídas dos comparadores  $C$  e  $D$ , na profundidade 2. Agora os fios de saída  $b_1$  e  $b_4$  têm seus valores finais, mas os fios de saída  $b_2$  e  $b_3$  não os têm. (d) No instante 3, os valores mostrados aparecem nas saídas do comparador  $E$ , na profundidade 3. Agora os fios de saída  $b_2$  e  $b_3$  têm seus valores finais

Sob a hipótese de que cada comparador demora um tempo unitário, podemos definir o “tempo de execução” de uma rede de comparação, ou seja, o tempo necessário para que todos os fios de saída recebam seus valores, uma vez que os fios de entrada tenham recebido os seus. Informalmente, esse tempo é o maior número de comparadores pelos quais qualquer elemento de entrada pode passar à medida que viaja de um fio de entrada para um fio de saída. De modo mais formal, definimos a **profundidade** de um fio como a seguir. Um fio de entrada de uma rede de comparação tem profundidade 0. Agora, se um comparador tem dois fios de entrada com profundidades  $d_x$  e  $d_y$ , então seus fios de saída têm profundidade  $\max(d_x, d_y) + 1$ . Pelo fato de não existir nenhum ciclo de comparadores em uma rede de comparação, a profundidade de um fio é bem definida, e definimos a profundidade de um comparador como a profundidade de seus fios de saída. A Figura 27.2 mostra as profundidades de comparadores. A profundidade de uma rede de comparação é a profundidade máxima de um fio de saída ou, de modo equivalente, a profundidade máxima de um comparador. Por exemplo, a rede de comparação da Figura 27.2 tem profundidade 3, porque o comparador  $E$  tem profundidade 3. Se cada comparador demora um tempo unitário para produzir seu valor de saída, e se as entradas de rede aparecem no instante 0, um comparador à profundidade  $d$  produz suas saídas no instante  $d$ ; consequentemente, a profundidade da rede é igual ao tempo necessário para que a rede produza valores em todos os seus fios de saída.

Uma **rede de ordenação** é uma rede de comparação para a qual a seqüência de saída é monotonicamente crescente (isto é,  $b_1 \leq b_2 \leq \dots \leq b_n$ ) para *toda* seqüência de entrada. É claro que nem toda rede de comparação é uma rede de ordenação, mas a rede da Figura 27.2 é. Para ver por que, observe que após o instante 1, o mínimo dos quatro valores de entrada foi produzido pela saída superior do comparador A ou pela saída superior do comparador B. Então, após o instante 2, ele deve estar na saída superior do comparador C. Um argumento simétrico mostra que, após o instante 2, o máximo dos quatro valores de entrada foi produzido pela saída inferior do comparador D. Tudo o que resta para o comparador E é assegurar que os dois valores intermediários ocuparão suas posições de saída corretas, o que acontece no instante 3.

Uma rede de comparação é semelhante a um procedimento pelo fato de especificar o modo como as comparações devem ocorrer, mas é diferente de um procedimento no sentido de que seu **tamanbo** – o número de comparadores que ela contém – depende do número de entradas e saídas. Então, na realidade descreveremos “famílias” de redes de comparação. Por exemplo, o objetivo deste capítulo é desenvolver uma família SORTER de redes de ordenação eficientes. Especificaremos uma determinada rede dentro de uma família pelo nome da família e pelo número de entradas (que é igual ao número de saídas). Por exemplo, a rede de ordenação de  $n$  entradas e  $n$  saídas na família SORTER é denominada SORTER[ $n$ ].



**FIGURA 27.3** Uma rede de ordenação baseada na ordenação por inserção para uso no Exercício 27.1-6

## **Exercícios**

27.1-1

Mostre os valores que aparecem em todos os fios da rede da Figura 27.2 quando ela recebe a sequência de entrada  $\langle 9, 5, 6, 2 \rangle$ .

27.1-2

Seja  $n$  uma potência exata de 2. Mostre como construir uma rede de comparação de  $n$  entradas e  $n$  saídas de profundidade  $\lg n$ , na qual o fio de saída superior sempre transporta o valor de entrada mínimo e o fio de saída inferior sempre transporta o valor de entrada máximo.

27.1-3

É possível tomar uma rede de ordenação e adicionar a ela um comparador, resultando em uma rede de comparação que não é uma rede de ordenação. Mostre como adicionar um comparador à rede da Figura 27.2, de tal modo que a rede resultante não ordene toda permutação de entrada.

27.1-4

Prove que qualquer rede de ordenação sobre  $n$  entradas tem profundidade no mínimo igual a  $\lg n$ .

27.1-5

Prove que o número de comparadores em qualquer rede de ordenação é  $\Omega(n \lg n)$ .

27.1-6

558 | Considere a rede de comparação mostrada na Figura 27.3. Prove que ela é de fato uma rede de ordenação, e descreva como sua estrutura se relaciona à da ordenação por inserção (Seção 2.1).

### 27.1-7

Podemos representar uma rede de comparação de  $n$  entradas com  $c$  comparadores como uma lista de  $c$  pares de inteiros no intervalo de 1 até  $n$ . Se dois pares contêm um inteiro em comum, a ordem dos comparadores correspondente na rede é determinada pela ordem dos pares na lista. Dada essa representação, descreva um algoritmo de tempo (serial)  $O(n + c)$  para determinar a profundidade de uma rede de comparação.

### 27.1-8 \*

Suponha que, além do tipo padrão de comparador, introduzimos um comparador “de cabeça para baixo” que produz sua saída mínima no fio inferior e sua saída máxima no fio superior. Mostre como converter qualquer rede de ordenação que utilize um total de  $c$  comparadores padrão e de cabeça para baixo em uma que utilize  $c$  comparadores padrão. Prove que seu método de conversão é correto.

## 27.2 O princípio zero um

O **princípio zero um** estabelece que, se uma rede de ordenação funciona corretamente quando cada entrada é tirada do conjunto  $\{0, 1\}$ , então ela funciona de modo correto sobre números de entrada arbitrários. (Os números podem ser inteiros, reais ou, em geral, qualquer conjunto de valores obtidos a partir de qualquer conjunto ordenado linearmente.) À medida que construirmos redes de ordenação e outras redes de comparação, o princípio zero um nos permitirá concentrar nossa atenção em sua operação para seqüências de entrada consistindo somente em valores 0 e 1. Depois que tivermos construído uma rede de ordenação e provado que ela pode ordenar seqüências formadas apenas por valores zero e um, apelaremos para o princípio zero um com o objetivo de mostrar que ele ordena de maneira adequada seqüências de valores arbitrários.

A prova do princípio zero um se baseia na noção de função monotonicamente crescente (Seção 3.2).

### Lema 27.1

Se uma rede de comparação transforma a seqüência de entrada  $a = \langle a_1, a_2, \dots, a_n \rangle$  na seqüência de saída  $b = \langle b_1, b_2, \dots, b_n \rangle$ , então, para qualquer função monotonicamente crescente  $f$ , a rede transforma a seqüência de entrada  $f(a) = \langle f(a_1), f(a_2), \dots, f(a_n) \rangle$  na seqüência de saída  $f(b) = \langle f(b_1), f(b_2), \dots, f(b_n) \rangle$ .

**Prova** Primeiro, provaremos a afirmação de que, se  $f$  é uma função monotonicamente crescente, então um único comparador com entradas  $f(x)$  e  $f(y)$  produz saídas  $f(\min(x, y))$  e  $f(\max(x, y))$ . Em seguida, utilizaremos a indução para provar o lema.

Para provar a afirmação, considere um comparador cujos valores de entrada sejam  $x$  e  $y$ . A saída superior do comparador é  $\min(x, y)$  e a saída inferior é  $\max(x, y)$ . Suponha que agora aplicarmos  $f(x)$  e  $f(y)$  às entradas do comparador, como mostra a Figura 27.4. A operação do comparador produz o valor  $\min(f(x), f(y))$  na saída superior e o valor  $\max(f(x), f(y))$  na saída inferior. Como  $f$  é monotonicamente crescente,  $x = y$  implica  $f(x) = f(y)$ . Conseqüentemente, temos as identidades

$$\begin{aligned}\min(f(x), f(y)) &= f(\min(x, y)), \\ \max(f(x), f(y)) &= f(\max(x, y)).\end{aligned}$$

Desse modo, o comparador produz os valores  $f(\min(x, y))$  e  $f(\max(x, y))$  quando  $f(x)$  e  $f(y)$  são suas entradas, o que completa a prova da afirmação.

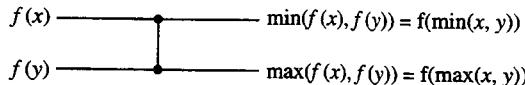


FIGURA 27.4 A operação do comparador na prova do Lema 27.1. A função  $f$  é monotonicamente crescente

Podemos usar a indução sobre a profundidade de cada fio em uma rede de comparação geral para provar um resultado mais forte que o enunciado do lema: se um fio assume o valor  $a_i$ , quando a seqüência de entrada  $\alpha$  é aplicada à rede, então ele assume o valor  $f(a_i)$  quando a seqüência de entrada  $f(\alpha)$  é aplicada. Pelo fato de os fios de saída estarem incluídos nesse enunciado, prová-lo irá provar o lema.

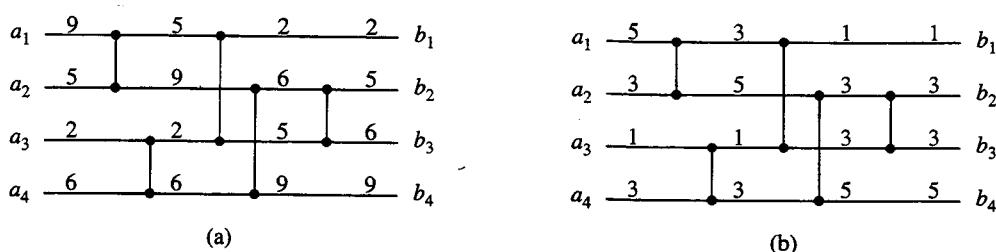
Para a base, considere um fio na profundidade 0, ou seja, um fio de entrada  $a_i$ . O resultado se segue de modo trivial: quando  $f(a)$  é aplicado à rede, o fio de entrada transporta  $f(a_i)$ . Para a etapa induutiva, considere um fio na profundidade  $d$ , onde  $d = 1$ . O fio é a saída de um comparador na profundidade  $d$ , e os fios de entrada para esse comparador estão em uma profundidade estritamente menor que  $d$ . Então, pela hipótese induutiva, se os fios de entrada para o comparador transportarem valores  $a_i$  e  $a_j$  quando a seqüência de entrada  $a$  for aplicada, então eles transportarão  $f(a_i)$  e  $f(a_j)$  quando a seqüência de entrada  $f(a)$  for aplicada. De acordo com nossa afirmação anterior, os fios de saída desse comparador transportarão então  $f(\min(a_i, a_j))$  e  $f(\max(a_i, a_j))$ . Considerando-se que eles transportam  $\min(a_i, a_j)$  e  $\max(a_i, a_j)$  quando a seqüência de entrada é  $a$ , o lema está provado.

Como um exemplo da aplicação do Lema 27.1, a Figura 27.5(b) mostra a rede de ordenação da Figura 27.2 (repetida na Figura 27.5(a)) com a função monotonicamente crescente  $f(x) = \lceil x/2 \rceil$  aplicada às entradas. O valor em todo fio é aplicado ao valor sobre o mesmo fio na Figura 27.2.

Quando uma rede de comparação é uma rede de ordenação, o Lema 27.1 nos permite provar o seguinte resultado notável.

### **Teorema 27.2 (Princípio zero um)**

Se uma rede de comparação com  $n$  entradas ordena todas as  $2^n$  seqüências possíveis de valores 0 e 1 corretamente, então ela ordena corretamente todas as seqüências de números arbitrários.



**FIGURA 27.5** (a) A rede de ordenação da Figura 27.2 com a seqüência de entrada  $\langle 9, 5, 2, 6 \rangle$  (b) A mesma rede de ordenação com a função monotonicamente crescente  $f(x) = f(\lceil x/2 \rceil)$ , aplicada às entradas. Cada fio nessa rede tem o valor de  $f$  aplicado ao valor no fio correspondente em (a)

**Prova** Suponha para fins de contradição que a rede ordene todas as seqüências zero um, mas que exista uma seqüência de números arbitrários que a rede não ordena corretamente. Isto é, existe uma seqüência de entrada  $\langle a_1, a_2, \dots, a_n \rangle$  contendo elementos  $a_i$  e  $a_j$  tais que  $a_i < a_j$ , mas a rede coloca  $a_j$  antes de  $a_i$  na seqüência de saída. Definimos uma função monotonicamente crescente  $f$  como

$$560 \quad | \quad f(x) = \begin{cases} 0 & \text{se } x \leq a_i, \\ 1 & \text{se } x > a_i. \end{cases}$$

Tendo em vista que a rede coloca  $a_j$  antes de  $a_i$  na seqüência de saída quando a entrada é  $\langle a_1, a_2, \dots, a_n \rangle$ , decore do Lema 27.1 que ela coloca  $f(a_j)$  antes  $f(a_i)$  na seqüência de saída quando a entrada é  $\langle f(a_1), f(a_2), \dots, f(a_n) \rangle$ . Porém, como  $f(a_j) = 1$  e  $f(a_i) = 0$ , obtemos a contradição de que a rede deixa de ordenar corretamente a seqüência zero um  $\langle f(a_1), f(a_2), \dots, f(a_n) \rangle$ .

## Exercícios

### 27.2-1

Prove que a aplicação de uma função monotonicamente crescente a uma seqüência ordenada produz uma seqüência ordenada.

### 27.2-2

Prove que uma rede de comparação com  $n$  entradas ordena corretamente a seqüência de entrada  $\langle n, n-1, \dots, 1 \rangle$  se e somente se ela ordena corretamente as  $n-1$  seqüências zero um  $\langle 1, 0, 0, \dots, 0, 0 \rangle, \langle 1, 1, 0, \dots, 0, 0 \rangle, \dots, \langle 1, 1, 1, \dots, 1, 0 \rangle$ .

### 27.2-3

Use o princípio zero um para provar que a rede de comparação mostrada na Figura 27.6 é uma rede de ordenação.

### 27.2-4

Enuncie e prove uma analogia do princípio zero um para um modelo de árvore de decisão. (Sugestão: Certifique-se de tratar a igualdade de modo apropriado.)

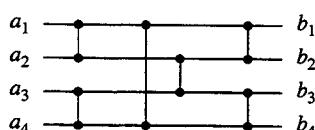


FIGURA 27.6 Uma rede de ordenação para ordenar 4 números

### 27.2-5

Prove que uma rede de ordenação de  $n$  entradas deve conter pelo menos um comparador entre a  $i$ -ésima e a  $(i+1)$ -ésima linhas para todo  $i = 1, 2, \dots, n-1$ .

## 27.3 Uma rede de ordenação bitônica

O primeiro passo em nossa construção de uma rede de ordenação eficiente é construir uma rede de comparação que possa ordenar qualquer **seqüência bitônica**: uma seqüência que aumenta monotonicamente e depois diminui monotonicamente, ou então que pode ser deslocada de forma circular para se tornar monotonicamente crescente e depois monotonicamente decrescente. Por exemplo, as seqüências  $\langle 1, 4, 6, 8, 3, 2 \rangle, \langle 6, 9, 4, 2, 3, 5 \rangle$  e  $\langle 9, 8, 3, 2, 4, 6 \rangle$  são todas bitônicas. Como uma condição limite, dizemos que qualquer seqüência de apenas 1 ou 2 números é bitônica. As seqüências zeros um que são bitônicas têm uma estrutura simples. Elas têm a forma  $0^i 1^j 0^k$  ou a forma  $1^i 0^j 1^k$  para algum  $i, j, k \geq 0$ . Observe que uma seqüência que é monotonicamente crescente ou monotonicamente decrescente também é bitônica.

O ordenador bitônico que construiremos é uma rede de comparação que ordena seqüências bitônicas de valores 0 e 1. O Exercício 27.3-6 lhe pede para mostrar que o ordenador bitônico pode ordenar seqüências bitônicas de números arbitrários.

## O meio-limpador

Um ordenador bitônico é constituído por vários estágios, cada um deles chamado um **meio-limpador**. Cada meio-limpador é uma rede de comparação de profundidade 1 na qual a linha

de entrada  $i$  é comparada com a linha  $i + n/2$  para  $i = 1, 2, \dots, n/2$ . (Supomos que  $n$  é par.) A Figura 27.7 mostra HALF-CLEANER[8], o meio-limpador com 8 entradas e 8 saídas.

Quando uma seqüência bitônica de valores 0 e 1 é aplicada como entrada a um meio-limpador, o meio-limpador produz uma seqüência de saída em que valores menores estão na metade superior, valores maiores estão na metade inferior e ambas as metades são bitônicas. De fato, pelo menos uma das metades é *limpa* – consistindo toda ela em valores 0 ou toda em valores 1 – e é dessa propriedade que deriva o nome “meio-limpador”. (Observe que todas as seqüências limpas são bitônicas.) O próximo lema prova essas propriedades dos meio-limpadores.

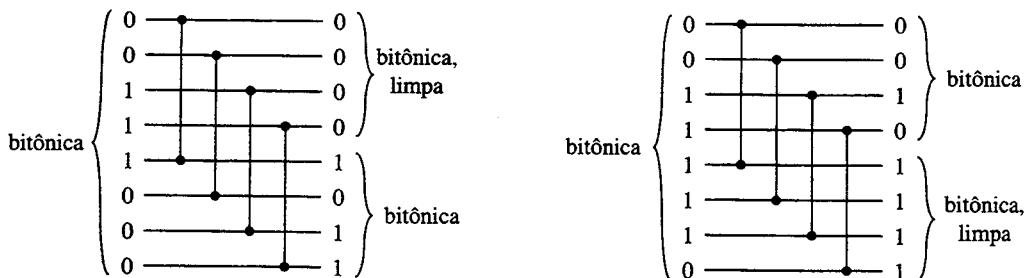


FIGURA 27.7 A rede de comparação HALF-CLEANER[8]. Dois exemplos diferentes de valores de entrada e saída zero um são mostrados. A entrada é considerada bitônica. Um meio-limpador assegura que todo elemento de saída da metade superior é pelo menos tão pequeno quanto todo elemento de saída da metade inferior. Além disso, ambas as metades são bitônicas, e pelo menos uma metade é limpa

### Lema 27.3

Se a entrada de um meio-limpador é uma seqüência bitônica de valores 0 e 1, então a saída satisfaz às seguintes propriedades: tanto a metade superior quanto a metade inferior são bitônicas, todo elemento na metade superior é pelo menos tão pequeno quanto todo elemento na metade inferior, e pelo menos uma metade é limpa.

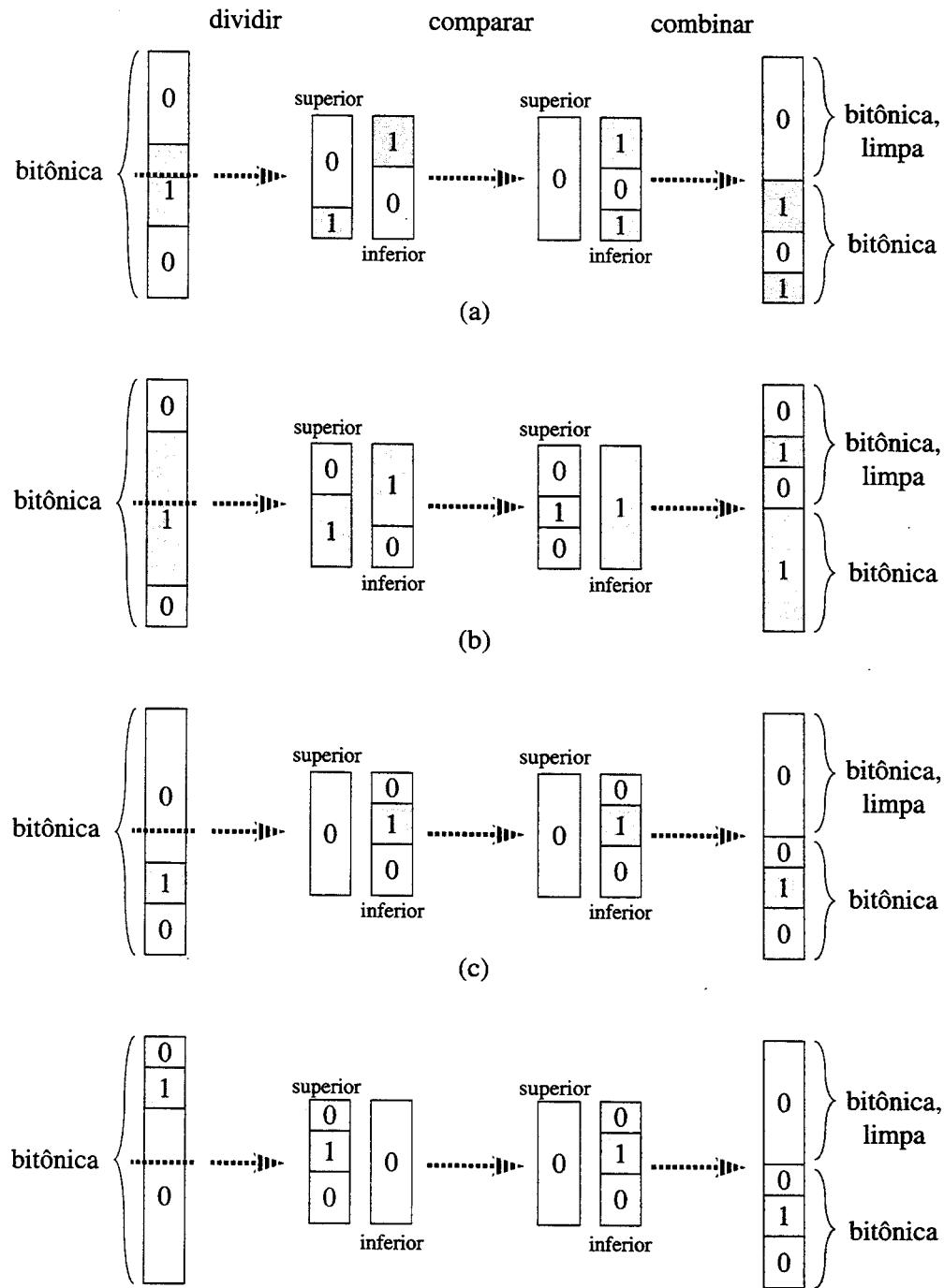
**Prova** A rede de comparação HALF-CLEANER[ $n$ ] compara entradas  $i$  e  $i + n/2$  para  $i = 1, 2, \dots, n/2$ . Sem perda de generalidade, suponha que a entrada tenha a forma 00...011...100...0. (A situação na qual a entrada é da forma 11...100...011...1 é simétrica.) Existem três casos possíveis, dependendo do bloco de valores 0 ou 1 consecutivos no qual recai o ponto médio  $n/2$ , e um desses casos (aquele no qual o ponto médio ocorre no bloco de valores 1) é dividido ainda mais em dois casos. Os quatro casos são mostrados na Figura 27.8. Em cada caso mostrado, o lema é válido. ■

## O ordenador bitônico

Combinando recursivamente meio-limpadores, como mostra a Figura 27.9, podemos construir um **ordenador bitônico**, o qual é uma rede que ordena seqüências bitônicas. O primeiro estágio de BITONIC-SORTER[ $n$ ] consiste em HALF-CLEANER[ $n$ ] que, pelo Lema 27.3, produz duas seqüências bitônicas de metade do tamanho, tais que todo elemento na metade superior é pelo menos tão pequeno quanto todo elemento na metade inferior. Desse modo, podemos completar a ordenação usando duas cópias de BITONIC-SORTER[ $n/2$ ] para ordenar as duas metades recursivamente. Na Figura 27.9(a), a recursão foi mostrada de forma explícita e, na Figura 27.9(b), a recursão foi estendida para mostrar os meio-limpadores progressivamente menores que constituem o restante do ordenador bitônico. A profundidade  $D(n)$  de BITONIC-SORTER[ $n$ ] é dada pela recorrência

$$D(n) = \begin{cases} 0 & \text{se } n = 1, \\ D(n/2) + 1 & \text{se } n = 2^k \text{ e } k \geq 1, \end{cases}$$

cuja solução é  $D(n) = \lg n$ .



**FIGURA 27.8** As comparações possíveis em HALF-CLEANER[n]. A seqüência de entrada é considerada uma seqüência bitônica de valores 0 e 1 e, sem perda de generalidade, supomos que ela tem a forma 00...011...100...0. As subsequências de valores 0 são brancas, e as subsequências de valores 1 são cinza. Podemos imaginar que as  $n$  entradas estão divididas em duas metades tais que, para  $i = 1, 2, \dots, n/2$ , as entradas  $i$  e  $i + n/2$  são comparadas. (a)–(b) Casos em que a divisão ocorre na subsequência intermediária de valores 1. (c)–(d) Casos em que a divisão ocorre em uma subsequência de valores 0. Em todos os casos, cada elemento na metade superior é pelo menos tão pequeno quanto todo elemento na metade inferior, ambas as metades são bitônicas e pelo menos uma metade é limpa

Desse modo, uma seqüência bitônica zero um pode ser ordenada por BITONIC-SORTER, que tem uma profundidade igual a  $\lg n$ . Segue-se pela analogia do princípio zero um dada no Exercício 27.3-6 que qualquer seqüência bitônica de números arbitrários pode ser ordenada por essa rede.

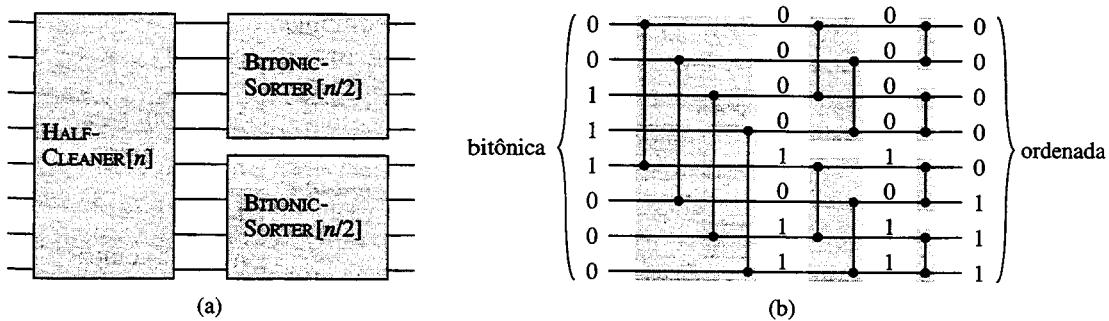


FIGURA 27.9 A rede de comparação BITONIC-SORTER[ $n$ ], mostrada aqui para  $n = 8$ . (a) A construção recursiva: HALF-CLEANER[ $n$ ] seguido por duas cópias de BITONIC-SORTER[ $n/2$ ] que operam em paralelo. (b) A rede depois de se estender a recursão. Cada meio-limpador está sombreado. Exemplos de valores zero um são mostrados nos fios

## Exercícios

### 27.3-1

Quantas seqüências bitônicas zero um de comprimento  $n$  existem?

### 27.3-2

Mostre que BITONIC-SORTER[ $n$ ], onde  $n$  é uma potência exata de 2, contém  $\Theta(n \lg n)$  comparadores.

### 27.3-3

Descreva como um ordenador bitônico de profundidade  $O(\lg n)$  pode ser construído quando o número  $n$  de entradas não é uma potência exata de 2.

### 27.3-4

Se a entrada para um meio-limpador é uma seqüência bitônica de números arbitrários, prove que a saída satisfaz às seguintes propriedades: tanto a metade superior quanto a metade inferior são bitônicas, e todo elemento na metade superior é pelo menos tão pequeno quanto todo elemento na metade inferior.

### 27.3-5

Considere duas seqüências de valores 0 e 1. Prove que, se todo elemento em uma seqüência é pelo menos tão pequeno quanto todo elemento na outra seqüência, então uma das duas seqüências é limpa.

### 27.3-6

Prove a seguinte analogia do princípio zero um para redes de ordenação bitônicas: uma rede de comparação que pode ordenar qualquer seqüência bitônica de valores 0 e 1 pode ordenar qualquer seqüência bitônica de números arbitrários.

## 27.4 Uma rede de intercalação

Nossa rede de ordenação será construída a partir de *redes de intercalação*, que são redes que podem intercalar duas seqüências de entrada ordenadas em uma única seqüência de saída ordenada. Modificamos BITONIC-SORTER[ $n$ ] para criar a rede de intercalação MERGER[ $n$ ]. Como ocorre com o ordenador bitônico, provaremos a correção da rede de intercalação apenas para entradas que são seqüências zero um. O Exercício 27.4-1 lhe pede para mostrar como a prova pode ser estendida a valores de entrada arbitrários.

A rede de intercalação se baseia na intuição a seguir. Dadas duas seqüências ordenadas, se invertermos a ordem da segunda seqüência e depois concatenarmos as duas seqüências, a seqüência resultante será bitônica. Por exemplo, dadas as seqüências zero um ordenadas  $X = 00000111$  e  $Y = 00001111$ , invertemos  $Y$  para obter  $Y^R = 11110000$ . A concatenação de  $X$  e  $Y^R$

produz 000001111110000, que é bitônica. Desse modo, para intercalar as duas seqüências de entrada  $X$  e  $Y$ , basta executar uma ordenação bitônica sobre  $X$  concatenada com  $Y^R$ .

Podemos construir  $\text{MERGER}[n]$  modificando o primeiro meio-limpador de  $\text{BITONIC-SORTER}[n]$ . A chave é executar a inversão da segunda metade das entradas de forma implícita. Dadas duas seqüências ordenadas  $\langle a_1, a_2, \dots, a_{n/2} \rangle$  e  $\langle a_{n/2+1}, a_{n/2+2}, \dots, a_n \rangle$  a serem intercaladas, queremos obter o efeito da ordenação bitônica da seqüência  $\langle a_1, a_2, \dots, a_{n/2}, a_n, a_{n-1}, \dots, a_{n/2+1} \rangle$ . Tendo em vista que o primeiro meio-limpador de  $\text{BITONIC-SORTER}[n]$  compara entradas  $i$  e  $n/2 + i$  para  $i = 1, 2, \dots, n/2$ , fazemos o primeiro estágio da rede de intercalação comparar as entradas  $i$  e  $n - i + 1$ . A Figura 27.10 mostra a correspondência. A única sutileza é que a ordem das saídas da parte inferior do primeiro estágio de  $\text{MERGER}[n]$  estão invertidas em comparação com a ordem das saídas de um meio-limpador comum. Porém, como a inversão de uma seqüência bitônica é bitônica, as saídas superior e inferior do primeiro estágio da rede de intercalação satisfazem às propriedades do Lema 27.3, e assim as partes superior e inferior podem ser ordenadas em paralelo pela ordenação bitônica para produzir a saída ordenada da rede de intercalação.

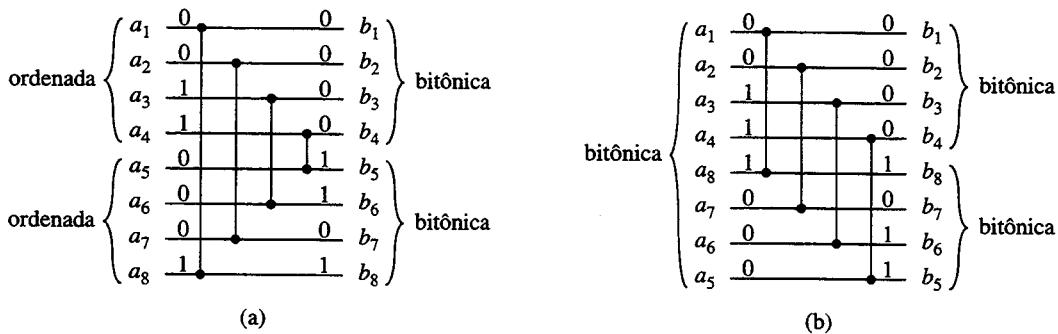


FIGURA 27.10 Comparando o primeiro estágio de  $\text{MERGER}[n]$  com  $\text{HALF-CLEANER}[n]$ , para  $n = 8$ . (a) O primeiro estágio de  $\text{MERGER}[n]$  transforma as duas seqüências de entrada monotônicas  $\langle a_1, a_2, \dots, a_{n/2} \rangle$  e  $\langle a_{n/2+1}, a_{n/2+2}, \dots, a_n \rangle$  em duas seqüências bitônicas  $\langle b_1, b_2, \dots, b_{n/2} \rangle$  e  $\langle b_{n/2+1}, b_{n/2+2}, \dots, b_n \rangle$ . (b) A operação equivalente para  $\text{HALF-CLEANER}[n]$ . A seqüência de entrada bitônica  $\langle a_1, a_2, \dots, a_{n/2-1}, a_n, a_{n-1}, \dots, a_{n/2+2}, a_{n/2+1} \rangle$  é transformada nas duas seqüências bitônicas  $\langle b_1, b_2, \dots, b_{n/2} \rangle$  e  $\langle b_n, b_{n-1}, \dots, b_{n/2+1} \rangle$

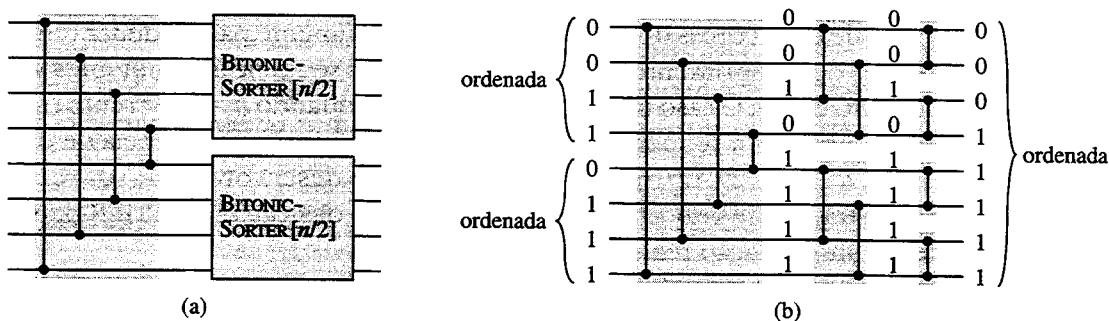


FIGURA 27.11 Uma rede que intercala duas seqüências de entrada ordenadas em uma única seqüência de saída ordenada. A rede  $\text{MERGER}[n]$  pode ser vista como  $\text{BITONIC-SORTER}[n]$  com o primeiro meio-limpador alterado para comparar entradas  $i$  e  $n - i + 1$  para  $i = 1, 2, \dots, n/2$ . No caso,  $n = 8$ . (a) A rede decomposta no primeiro estágio, seguida pelas duas cópias paralelas de  $\text{BITONIC-SORTER}[n/2]$ . (b) A mesma rede com a recursão demonstrada. Exemplos de valores zero um são mostrados nos fios, e os estágios estão sombreados

A rede de intercalação resultante é mostrada na Figura 27.11. Apenas o primeiro estágio de  $\text{MERGER}[n]$  é diferente de  $\text{BITONIC-SORTER}[n]$ . Conseqüentemente, a profundidade de  $\text{MERGER}[n]$  é  $\lg n$ , igual à de  $\text{BITONIC-SORTER}[n]$ .

## Exercícios

### 27.4-1

Prove uma analogia do princípio zero um para redes de intercalação. Especificamente, mostre que uma rede de comparação que pode intercalar duas seqüências monotonicamente crescentes quaisquer de valores 0 e 1 pode intercalar duas seqüências monotonicamente crescentes quaisquer de números arbitrários.

### 27.4-2

Quantas seqüências de entrada zero um diferentes devem ser aplicadas à entrada de uma rede de comparação para se verificar que ela é uma rede de intercalação?

### 27.4-3

Mostre que qualquer rede que pode intercalar um item com  $n - 1$  itens ordenados para produzir uma seqüência ordenada de comprimento  $n$  deve ter profundidade pelo menos igual a  $\lg n$ .

### 27.4-4 \*

Considere uma rede de intercalação com entradas  $a_1, a_2, \dots, a_n$ , sendo  $n$  uma potência exata de 2, na qual as duas seqüências monotônicas a serem intercaladas são  $\langle a_1, a_3, \dots, a_{n-1} \rangle$  e  $\langle a_2, a_4, \dots, a_n \rangle$ . Prove que o número de comparadores nesse tipo de rede de intercalação é  $\Omega(n \lg n)$ . Por que esse é um limite inferior interessante? (Sugestão: Particione os comparadores em três conjuntos.)

### 27.4-5 \*

Prove que qualquer rede de intercalação, independente da ordem das entradas, requer  $\Omega(n \lg n)$  comparadores.

## 27.5 Uma rede de ordenação

Agora temos todas as ferramentas necessárias para construir uma rede capaz de ordenar qualquer seqüência de entrada. A rede de ordenação SORTER[ $n$ ] utiliza a rede de intercalação para implementar uma versão paralela de ordenação por intercalação do tipo visto na Seção 2.3.1. A construção e a operação da rede de ordenação estão ilustradas na Figura 27.12.

A Figura 27.12(a) mostra a construção recursiva de SORTER[ $n$ ]. Os  $n$  elementos de entrada são ordenados pelo uso de duas cópias de SORTER[ $n/2$ ] recursivamente para ordenar (em paralelo) duas subseqüências de comprimento  $n/2$  cada uma. As duas seqüências resultantes são então intercaladas por MERGER[ $n$ ]. O caso limite para a recursão é quando  $n = 1$ , em cujo caso podemos usar um fio para ordenar a seqüência de um elemento, pois uma seqüência de um elemento já está ordenada. A Figura 27.12(b) mostra o resultado do desenvolvimento da recursão, e a Figura 27.12(c) mostra a rede real obtida pela substituição das caixas MERGER na Figura 27.12(b) pelas redes de intercalação reais.

Os dados passam através de  $\lg n$  estágios na rede SORTER[ $n$ ]. Cada uma das entradas individuais para a rede já é uma seqüência ordenada de um elemento. O primeiro estágio SORTER[ $n$ ] consiste em  $n/2$  cópias de MERGER[2] que funcionam em paralelo para intercalar pares de seqüências de um elemento, a fim de produzir seqüências ordenadas de comprimento 2. O segundo estágio consiste em  $n/4$  cópias de MERGER[4] que intercalam pares dessas seqüências ordenadas de 2 elementos para produzir seqüências ordenadas de comprimento 4. Em geral, para  $k = 1, 2, \dots, \lg n$ , o estágio  $k$  consiste em  $n/2^k$  cópias de MERGER[ $2^k$ ] que intercalam pares das seqüências ordenadas de  $2^{k-1}$  elementos para produzir seqüências ordenadas de comprimento  $2^k$ . No estágio final, é produzida uma única seqüência ordenada que consiste em todos os valores de entrada. É possível mostrar por indução que essa rede de ordenação pode ordenar seqüências zero um e, consequentemente, pelo princípio zero um (Teorema 27.2), ela pode ordenar valores arbitrários.

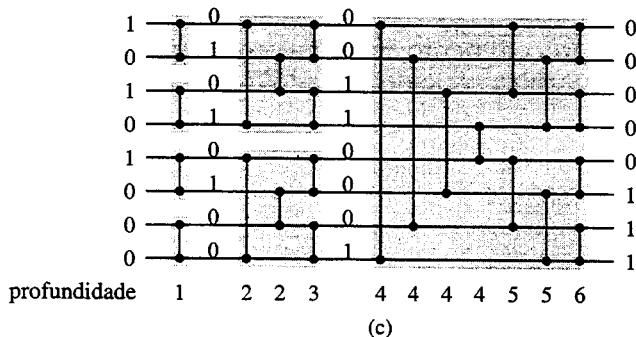
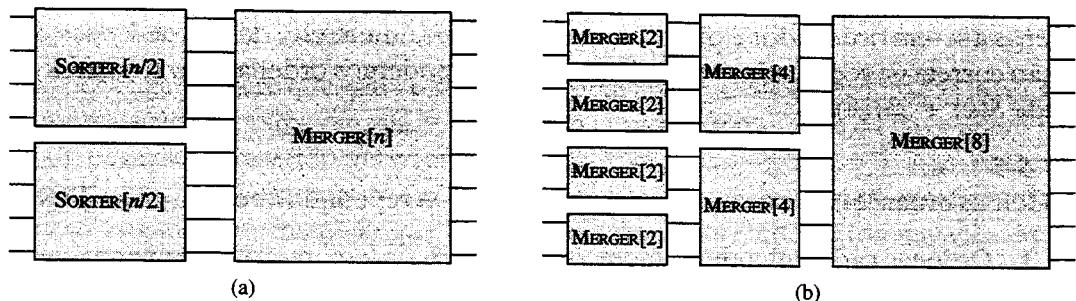


FIGURA 27.12 A rede de ordenação  $\text{SORTER}[n]$  construída pela combinação recursiva de redes de intercalação. (a) A construção recursiva. (b) Desenvolvendo a recursão. (c) Substituindo as caixas MERGER pelas redes de intercalação reais. A profundidade de cada comparador está indicada, e exemplos de valores zero um são mostrados nos fios

Podemos analisar a profundidade da rede de ordenação recursivamente. A profundidade  $D(n)$  de  $\text{SORTER}[n]$  é a profundidade  $D(n/2)$  de  $\text{SORTER}[n/2]$  (existem duas cópias de  $\text{SORTER}[n/2]$ , mas elas operam em paralelo) somada à profundidade  $\lg n$  de  $\text{MERGER}[n]$ . Consequentemente, a profundidade de  $\text{SORTER}[n]$  é dada pela recorrência

$$D(n) = \begin{cases} 0 & \text{se } n = 1 , \\ D(n/2) + \lg n & \text{se } n = 2^k \text{ e } k \geq 1 , \end{cases}$$

cuja solução é  $D(n) = \Theta(\lg^2 n)$ . (Use a versão do método mestre dada no Exercício 4.4-2.) Desse modo, podemos ordenar  $n$  números em paralelo no tempo  $O(\lg^2 n)$ .

## Exercícios

### 27.5-1

Quantos comparadores existem em  $\text{SORTER}[n]$ ?

### 27.5-2

Mostre que a profundidade de  $\text{SORTER}[n]$  é exatamente  $(\lg n)(\lg n + 1)/2$ .

### 27.5-3

Vamos supor que temos  $2n$  elementos  $\langle a_1, a_2, \dots, a_{2n} \rangle$  e desejamos particioná-los nos  $n$  menores e nos  $n$  maiores. Prove que podemos fazer isso em profundidade adicional constante depois de ordenar separadamente  $\langle a_1, a_2, \dots, a_n \rangle$  e  $\langle a_{n+1}, a_{n+2}, \dots, a_{2n} \rangle$ .

### 27.5-4 \*

Seja  $S(k)$  a profundidade de uma rede de ordenação com  $k$  entradas, e seja  $M(k)$  a profundidade de uma rede de intercalação com  $2k$  entradas. Vamos supor que temos uma seqüência de  $n$  nú-

meros a serem ordenados e que sabemos que todo número está dentro de  $k$  posições de sua posição correta na seqüência ordenada. Mostre que podemos ordenar os  $n$  números em profundidade  $S(k) + 2M(k)$ .

### 27.5-5 \*

Podemos ordenar os elementos de uma matriz  $m \times m$  repetindo  $k$  vezes o procedimento a seguir:

1. Ordenar cada linha de numeração ímpar em ordem monotonicamente crescente.
2. Ordenar cada linha de numeração par em ordem monotonicamente decrescente.
3. Ordenar cada coluna em ordem monotonicamente crescente.

Quantas iterações  $k$  são necessárias para esse procedimento efetuar a ordenação, e qual é o padrão da saída ordenada?

## Problemas

### 27-1 Redes de ordenação de transposição

Uma rede de comparação é uma **rede de transposição** se cada comparador conecta linhas adjacentes, como na rede da Figura 27.3.

- a. Mostre que qualquer rede de ordenação de transposição com  $n$  entradas tem  $\Omega(n^2)$  comparadores.
- b. Prove que uma rede de transposição com  $n$  entradas é uma rede de ordenação se e somente se ela ordena a seqüência  $\langle n, n-1, \dots, 1 \rangle$ . (Sugestão: Use um argumento de indução análogo ao da prova do Lema 27.1.)

Uma **rede de ordenação ímpar-par** sobre  $n$  entradas  $\langle a_1, a_2, \dots, a_n \rangle$  tem  $n$  níveis de comparadores conectados no padrão de “tijolos” ilustrado na Figura 27.13. Conforme vemos na figura, para  $i = 1, 2, \dots, n$  e  $d = 1, 2, \dots, n$ , a linha  $i$  é conectada por um comparador de profundidade  $d$  à linha  $j = i + (-1)^{i+d}$  se  $1 \leq j \leq n$ .

- c. Prove que as redes de ordenação ímpar-par realmente fazem a ordenação.

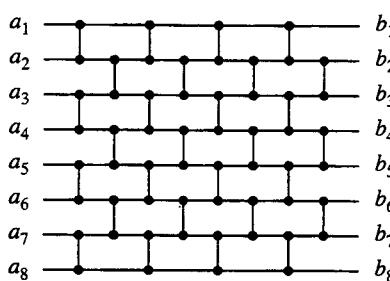


FIGURA 27.13 Uma rede de ordenação ímpar-par sobre 8 entradas

### 27-2 Rede de intercalação ímpar-par de Batcher

Na Seção 27.4, vimos como construir uma rede de intercalação baseada na ordenação bitônica. Neste problema, construiremos uma **rede de intercalação ímpar-par**. Vamos supor que  $n$  é uma potência exata de 2, e desejamos intercalar a seqüência ordenada de elementos nas linhas  $\langle a_1, a_2, \dots, a_n \rangle$  com os elementos das linhas  $\langle a_{n+1}, a_{n+2}, \dots, a_{2n} \rangle$ . Se  $n = 1$ , inserimos um comparador entre as linhas  $a_1$  e  $a_2$ . Caso contrário, construímos recursivamente duas redes de intercalação ímpar-par que operam em paralelo. A primeira intercala a seqüência das linhas  $\langle a_1, a_3, \dots, a_{n-1} \rangle$  com a seqüência das linhas  $\langle a_{n+1}, a_{n+3}, \dots, a_{2n-1} \rangle$  (os elementos ímpares). A segunda intercala  $\langle a_2, a_4, \dots, a_n \rangle$  com  $\langle a_{n+2}, a_{n+4}, \dots, a_{2n} \rangle$  (os elementos pares). Para combinar as duas subsequências ordenadas, inserimos um comparador entre  $a_{2i}$  e  $a_{2i+1}$ , para  $i = 1, 2, \dots, n-1$ .

- Desenhe uma rede de intercalação de  $2n$  entradas para  $n = 4$ .
- O professor Corrigan sugere que, para combinar as duas subsequências ordenadas produzidas pela intercalação recursiva, em vez de inserir um comparador entre  $a_{2i}$  e  $a_{2i+1}$  para  $i = 1, 2, \dots, n-1$ , deveria ser inserido um comparador entre  $a_{2i-1}$  e  $a_{2i}$  para  $i = 1, 2, \dots, n$ . Desenhe tal rede de  $2n$  entradas para  $n = 4$  e apresente um contra-exemplo para mostrar que o professor está enganado ao imaginar que a rede produzida é uma rede de intercalação. Mostre que a rede de intercalação de  $2n$  entradas da parte (a) funciona corretamente em seu exemplo.
- Use o princípio zero um para provar que qualquer rede de intercalação ímpar-par de  $2n$  entradas é de fato uma rede de intercalação.
- Qual é a profundidade de uma rede de intercalação ímpar-par de  $2n$  entradas? Qual é seu tamanho?

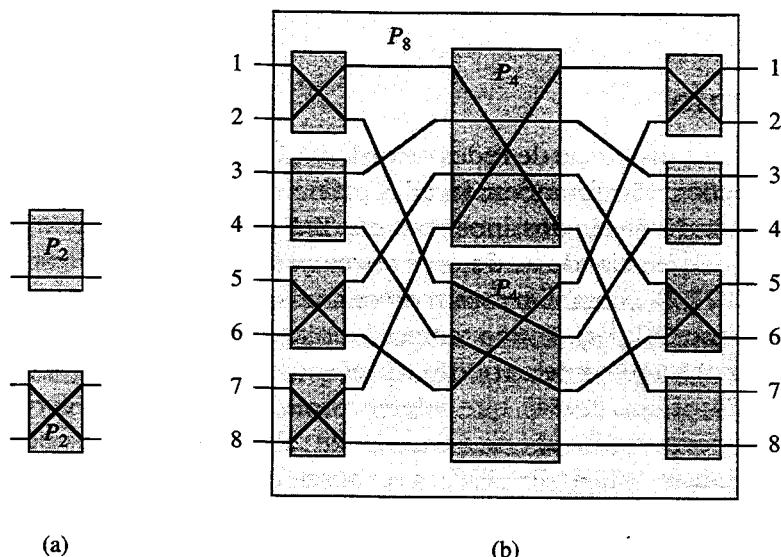


FIGURA 27.14 Redes de permutação. (a) A rede de permutação  $P_2$ , que consiste em uma única chave que pode ser configurada de uma das duas maneiras mostradas. (b) A construção recursiva de  $P_8$  a partir de 8 chaves e duas redes  $P_4$ . As chaves e as redes  $P_4$  estão configuradas para realizar a permutação  $\pi = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$

### 27-3 Redes de permutação

Uma **rede de permutação** sobre  $n$  entradas e  $n$  saídas tem chaves que permitem que ela conecte suas entradas a suas saídas de acordo com qualquer das  $n!$  permutações possíveis. A Figura 27.14(a) mostra a rede de permutação  $P_2$  de duas entradas e duas saídas, que consiste em uma única chave que pode ser configurada para conectar suas entradas diretamente a suas saídas ou para cruzá-las.

- Demonstre que, se substituirmos cada comparador em uma rede de ordenação pela chave da Figura 27.14(a), a rede resultante será uma rede de permutação. Ou seja, para qualquer permutação  $\pi$ , existe um modo de definir as chaves na rede para que a entrada  $i$  seja conectada à saída  $\pi(i)$ .

A Figura 27.14(b) mostra a construção recursiva de uma rede de permutação  $P_8$  de 8 entradas e 8 saídas que utiliza duas cópias de  $P_4$  e 8 chaves. As chaves foram configuradas para realizar a permutação  $\pi = \langle \pi(1), \pi(2), \dots, \pi(8) \rangle = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$ , a qual exige (recursivamente) que a  $P_4$  superior realize  $\langle 4, 2, 3, 1 \rangle$  e a  $P_4$  inferior realize  $\langle 2, 3, 1, 4 \rangle$ .

- b.** Mostre como realizar a permutação  $\langle 5, 3, 4, 6, 1, 8, 2, 7 \rangle$  em  $P_8$ , desenhando as configurações de chaves e as permutações executadas pelas duas redes  $P_4$ .

Seja  $n$  uma potência exata de 2. Defina  $P_n$  recursivamente em termos de duas  $P_{n/2}$ , de uma maneira semelhante à forma como definimos  $P_8$ .

- c.** Descreva um algoritmo de tempo  $O(n)$  (máquina de acesso aleatório comum) que configure as  $n$  chaves conectadas às entradas e saídas de  $P_n$  e que especifique as permutações que devem ser realizadas por cada  $P_{n/2}$  para executar qualquer permutação de  $n$  elementos dados. Prove que seu algoritmo é correto.
- d.** Quais são a profundidade e o tamanho de  $P_n$ ? Quanto tempo ele leva em uma máquina de acesso aleatório comum para calcular todas as configurações de chaves, inclusive aquelas que estão dentro das redes  $P_{n/2}$ ?
- e.** Demonstre que, para  $n > 2$ , qualquer rede de permutação – não apenas  $P_n$  – deve realizar alguma permutação por duas combinações distintas de configurações de chaves.

## Notas do capítulo

Knuth [185] contém uma discussão de redes de ordenação e utiliza diagramas para mostrar seu histórico. Aparentemente, elas foram exploradas primeiro em 1954 por P. N. Armstrong, R. J. Nelson e D. J. O'Connor. No início dos anos sessenta, K. E. Batcher descobriu a primeira rede capaz de intercalar duas seqüências de  $n$  números no tempo  $O(\lg n)$ . Ele utilizou a intercalação ímpar-par (ver Problema 27-2), e também mostrou como essa técnica poderia ser usada para ordenar  $n$  números no tempo  $O(\lg^2 n)$ . Pouco tempo depois, ele descobriu um ordenador bitônico de profundidade  $O(\lg n)$  semelhante ao que foi apresentado na Seção 27.3. Knuth atribui o princípio zero um a W. G. Bouricius (1954), que o demonstrou no contexto de árvores de decisão.

Por muito tempo, permaneceu aberta a questão de saber se existiria ou não uma rede de ordenação com profundidade  $O(\lg n)$ . Em 1983, a resposta foi mostrada como um sim um tanto insatisfatório. A rede de ordenação AKS (assim denominada em homenagem a seus desenvolvedores, Ajtai, Komlós e Szemerédi [11]) pode ordenar  $n$  números em profundidade  $O(\lg n)$  utilizando  $O(n \lg n)$  comparadores. Infelizmente, as constantes ocultas pela notação de  $O$  são muito grandes (muitos milhares), e desse modo ela não pode ser considerada prática.

---

## *Capítulo 28*

# *Operações sobre matrizes*

As operações sobre matrizes estão no núcleo da computação científica. Os algoritmos eficientes para o trabalho com matrizes são portanto de considerável interesse prático. Este capítulo fornece uma breve introdução à teoria de matrizes e às operações de matrizes, enfatizando os problemas de multiplicação de matrizes e resolvendo conjuntos de equações lineares simultâneas.

Após a Seção 28.1 introduzir conceitos e notações básicas de matrizes, a Seção 28.2 apresenta o surpreendente algoritmo de Strassen para multiplicar duas matrizes  $n \times n$  no tempo  $\Theta(n^{\lg 7}) = O(n^{2.81})$ . A Seção 28.3 mostra como resolver um conjunto de equações lineares usando decomposições LUP. Em seguida, a Seção 28.4 explora o estreito relacionamento entre o problema de multiplicar matrizes e o problema de inverter uma matriz. Finalmente, a Seção 28.5 discute a importante classe de matrizes simétricas definidas como positivas e mostra como elas podem ser usadas para encontrar uma solução de mínimos quadrados para um conjunto superdeterminado de equações lineares.

Uma questão importante que surge na prática é a *estabilidade numérica*. Devido à precisão limitada de representações de ponto flutuante em computadores reais, erros de arredondamento em computações numéricas podem se ampliar durante o curso de uma computação, levando a resultados incorretos; tais computações são numericamente instáveis. Embora façamos considerações breves sobre a estabilidade numérica, não a abordaremos neste capítulo. Indicamos ao leitor o excelente livro de Golub e Van Loan [125], que contém uma discussão completa das questões de estabilidade.

### **28.1 Propriedades de matrizes**

Nesta seção, revisaremos alguns conceitos básicos da teoria de matrizes e algumas propriedades fundamentais de matrizes, concentrando-nos naquelas que serão necessárias em seções posteriores.

#### **Matrizes e vetores**

Uma **matriz** é um arranjo retangular de números. Por exemplo,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad (28.1)$$

é uma matriz  $2 \times 3$ .  $A = (a_{ij})$  onde, para  $i = 1, 2$  e  $j = 1, 2, 3$ , o elemento da matriz na linha  $i$  e na coluna  $j$  é  $a_{ij}$ . Usamos letras maiúsculas para denotar matrizes e letras minúsculas subscritas correspondentes para denotar seus elementos. O conjunto de todas as matrizes  $m \times n$  com entradas de valores reais é denotado por  $\mathbb{R}^{m \times n}$ . Em geral, o conjunto de matrizes  $m \times n$  com entradas retiradas de um conjunto  $S$  é denotado por  $S^{m \times n}$ .

A **transposta** de uma matriz  $A$  é a matriz  $A^T$  obtida pela troca das linhas e colunas de  $A$ . Para a matriz  $A$  da equação (28.1),

$$A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Um **vetor** é um arranjo unidimensional de números. Por exemplo,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \quad (28.2)$$

é um vetor de tamanho 3. Usamos letras minúsculas para denotar vetores, e denotamos o  $i$ -ésimo elemento de um vetor  $x$  de tamanho  $n$  por  $x_i$ , para  $i = 1, 2, \dots, n$ . Tomamos a forma padrão de um vetor como um **vetor coluna** equivalente a uma matriz  $n \times 1$ ; o **vetor linha** correspondente é obtido tomando-se a transposta:

$$x^T = (2 \ 3 \ 5).$$

O **vetor unitário**  $e_i$  é o vetor cujo  $i$ -ésimo elemento é 1 e todos os outros elementos são iguais a 0. Em geral, o tamanho de um vetor unitário fica claro a partir do contexto.

Uma **matriz zero** é uma matriz na qual toda entrada é 0. Tal matriz é freqüentemente denotada por 0, pois a ambigüidade entre o número 0 e uma matriz de valores 0 normalmente é resolvida com facilidade a partir do contexto. Se uma matriz de valores 0 deve ser representada, então o tamanho da matriz também precisa ser derivado a partir do contexto.

Matrizes **quadradas**  $n \times n$  surgem com freqüência. Diversos casos especiais de matrizes quadradas têm interesse particular:

1. Uma **matriz diagonal** tem  $a_{ij} = 0$  sempre que  $i \neq j$ . Como todos os elementos fora da diagonal são iguais a zero, a matriz pode ser especificada listando-se os elementos ao longo da diagonal:

$$\text{diag}(a_{11}, a_{22}, \dots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}.$$

2. A **matriz identidade**  $I_n$   $n \times n$  é uma matriz diagonal com valores 1 ao longo da diagonal:

$$I_n = \text{diag}(1, 1, \dots, 1)$$

$$= \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Quando  $I$  aparece sem um subscrito, seu tamanho pode ser derivado a partir do contexto. A  $i$ -ésima coluna de uma matriz identidade é o vetor unitário  $e_i$ .

3. Uma **matriz tridiagonal**  $T$  é aquela para a qual  $t_{ij} = 0$  se  $|i-j| > 1$ . Entradas diferentes de zero só aparecem na diagonal principal, imediatamente acima da diagonal principal ( $t_{i+1,i}$  para  $i = 1, 2, \dots, n-1$ ), ou imediatamente abaixo da diagonal principal ( $t_{i-1,i}$  para  $i = 1, 2, \dots, n-1$ ):

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \cdots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \cdots & 0 & 0 & 0 \\ 0 & t_{32} & t_{34} & t_{11} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \cdots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \cdots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix}.$$

4. Uma **matriz triangular superior**  $U$  é aquela para a qual  $u_{ij} = 0$  se  $i > j$ . Todos os itens abaixo da diagonal são iguais a zero:

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}.$$

Uma matriz triangular superior é **triangular superior unitária** se tem apenas valores 1 ao longo da diagonal.

5. Uma **matriz triangular inferior**  $L$  é aquela para a qual  $l_{ij} = 0$  se  $i < j$ . Todos os itens acima da diagonal são iguais a zero:

$$U = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix}.$$

Uma matriz triangular inferior é **triangular inferior unitária** se tem apenas valores 1 ao longo da diagonal.

6. Uma **matriz permutação**  $P$  tem exatamente um valor 1 em cada linha ou coluna, e valores 0 em todas as outras posições. Um exemplo de matriz permutação é

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Tal matriz é chamada matriz permutação porque a multiplicação de um vetor  $x$  por uma matriz permutação tem o efeito de permutar (rearranjar) os elementos de  $x$ .

7. Uma **matriz simétrica**  $A$  satisfaz à condição  $A = A^T$ . Por exemplo,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

é uma matriz simétrica.

## Operações sobre matrizes

Os elementos de uma matriz ou um vetor são números de um sistema numérico, como os números reais, os números complexos ou módulos inteiros de um primo. O sistema numérico define como somar e multiplicar números. Podemos estender essas definições para englobar a adição e a multiplicação de matrizes.

Definimos a **adição de matrizes** como a seguir. Se  $A = (a_{ij})$  e  $B = (b_{ij})$  são matrizes  $m \times n$ , então sua matriz soma  $C = (c_{ij}) = A + B$  é a matriz  $m \times n$  definida por

$$c_{ij} = a_{ij} + b_{ij}$$

para  $i = 1, 2, \dots, m$  e  $j = 1, 2, \dots, n$ . Isto é, a adição de matrizes é executada no nível de componente. Uma matriz zero é a identidade para a adição de matrizes:

$$\begin{aligned} A + 0 &= A \\ &= 0 + A \end{aligned}$$

Se  $\lambda$  é um número e  $A = (a_{ij})$  é uma matriz, então  $\lambda A$  é o **múltiplo escalar** de  $A$  obtido pela multiplicação de cada um de seus elementos por  $\lambda$ . Como um caso especial, definimos a **negativa** de uma matriz  $A = (a_{ij})$  como  $-1 \cdot A = -A$ , de forma que a  $ij$ -ésima entrada de  $-A$  é  $-a_{ij}$ . Desse modo,

$$\begin{aligned} A + (-A) &= 0 \\ &= (-A) + A \end{aligned}$$

Dada essa definição, podemos definir a **subtração de matrizes** como a adição da negativa de uma matriz:  $A - B = A + (-B)$ .

Definimos a **multiplicação de matrizes** da maneira descrita a seguir. Começamos com duas matrizes  $A$  e  $B$  que são **compatíveis** no sentido de que o número de colunas de  $A$  é igual ao número de linhas de  $B$ . (Em geral, uma expressão contendo um produto de matrizes  $AB$  é sempre suposta como a implicação de que as matrizes  $A$  e  $B$  são compatíveis.) Se  $A = (a_{ij})$  é uma matriz  $m \times n$  e  $B = (b_{jk})$  é uma matriz  $n \times p$ , então seu produto de matrizes  $C = AB$  é a matriz  $m \times p$   $C = (c_{ik})$ , onde

$$c_{ik} = \sum_{j=1}^n a_{ij} b_{jk} \quad (28.3)$$

para  $i = 1, 2, \dots, m$  e  $k = 1, 2, \dots, p$ . O procedimento MATRIX-MULTIPLY da Seção 25.1 implementa a multiplicação de matrizes de maneira direta baseada na equação (28.3), supondo-se que as matrizes sejam quadradas:  $m = n = p$ . Para multiplicar matrizes  $n \times n$ , MATRIX-MULTIPLY executa  $n^3$  multiplicações e  $n^2(n - 1)$  adições, e seu tempo de execução é  $\Theta(n^3)$ .

As matrizes têm muitas (mas não todas) das propriedades algébricas típicas de números. As matrizes identidade são identidades para multiplicação de matrizes:

$$I_m A = A I_n = A$$

para qualquer matriz  $A$   $m \times n$ . A multiplicação por uma matriz zero fornece uma matriz zero:

$$A 0 = 0.$$

A multiplicação de matrizes é associativa:

$$A(BC) = (AB)C \quad (28.4)$$

para matrizes compatíveis  $A$ ,  $B$  e  $C$ . A multiplicação de matrizes se distribui sobre a adição:

$$\begin{aligned} A(B + C) &= AB + AC, \\ (B + C)D &= BD + CD. \end{aligned} \quad (28.5)$$

Para  $n > 1$ , a multiplicação de matrizes  $n \times n$  não é comutativa. Por exemplo, se  $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$  e  $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ , então

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

e

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Os produtos de matrizes por vetores ou de vetores por vetores são definidos como se o vetor fosse a matriz  $n \times 1$  equivalente (ou uma matriz  $1 \times n$ , no caso de um vetor linha). Desse modo, se  $A$  é uma matriz  $m \times n$  e  $x$  é um vetor de tamanho  $n$ , então  $Ax$  é um vetor de tamanho  $m$ . Se  $x$  e  $y$  são vetores de tamanho  $n$ , então

$$x^T y = \sum_{i=1}^n x_i y_i$$

é um número (na realidade, uma matriz  $1 \times 1$ ) chamada **produto interno** de  $x$  e  $y$ . A matriz  $xy^T$  é uma matriz  $Z$   $n \times n$  chamada **produto exterior** de  $x$  e  $y$ , com  $z_{ij} = x_i y_j$ . A **norma (euclidiana)**  $\|x\|$  de um vetor  $x$  de tamanho  $n$  é definida por

$$\begin{aligned} \|x\| &= (x_1^2 + x_2^2 + \dots + x_n^2)^{1/2} \\ &= (x^T x)^{1/2}. \end{aligned}$$

Portanto, a norma de  $x$  é seu comprimento no espaço euclidiano de  $n$  dimensões.

## Inversas, ordenações e determinantes de matrizes

Definimos a **inversa** de uma matriz  $A n \times n$  como a matriz  $n \times n$ , denotada por  $A^{-1}$  (se existir), tal que  $AA^{-1} = I_n = A^{-1}A$ . Por exemplo,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}.$$

Muitas matrizes  $n \times n$  diferentes de zero não têm inversas. Uma matriz sem uma inversa é chamada **não invertível** ou **singular**. Um exemplo de uma matriz singular não nula é

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

Se uma matriz tem uma inversa, ela é chamada **invertível** ou **não singular**. Inversas de matrizes, quando existem, são únicas. (Ver Exercício 28.1-3.) Se  $A$  e  $B$  são matrizes  $n \times n$  não singulares, então

$$(BA)^{-1} = A^{-1}B^{-1}. \quad (28.6)$$

A operação inversa comuta com a operação transposta:

$$(A^{-1})^T = (A^T)^{-1}.$$

Os vetores  $x_1, x_2, \dots, x_n$  são **linearmente dependentes** se existem coeficientes  $c_1, c_2, \dots, c_n$ , nem todos iguais a zero, tais que  $c_1x_1 + c_2x_2 + \dots + c_nx_n = 0$ . Por exemplo, os vetores linhas  $x_1 = (1 \ 2 \ 3)$ ,  $x_2 = (2 \ 6 \ 4)$  e  $x_3 = (4 \ 11 \ 9)$  são linearmente dependentes, pois  $2x_1 + 3x_2 - 2x_3 = 0$ . Se os vetores não são linearmente dependentes, eles são **linearmente independentes**. Por exemplo, as colunas de uma matriz identidade são linearmente independentes.

A **ordem de colunas** de uma matriz  $A m \times n$  não nula é o tamanho do maior conjunto de colunas linearmente independentes de  $A$ . De modo semelhante, a **ordem de linhas** de  $A$  é o tamanho do maior conjunto de linhas linearmente independentes de  $A$ . Uma propriedade fundamental de qualquer matriz  $A$  é que sua ordem de linhas é sempre igual a sua ordem de colunas; assim, podemos simplesmente nos referir à **ordem** de  $A$ . A ordem de uma matriz  $m \times n$  é um inteiro entre 0 e  $\min(m, n)$ , inclusive. (A ordem de uma matriz zero é 0, e a ordem de uma matriz identidade  $n \times n$  é  $n$ .) Uma definição alternativa, embora equivalente e com freqüência mais útil, é que a ordem de uma matriz  $m \times n$  não nula  $A$  é o menor número  $r$  tal que existem matrizes  $B$  e  $C$  de tamanhos respectivos  $m \times r$  e  $r \times n$  tais que

$$A = BC.$$

Uma matriz quadrada  $n \times n$  tem **ordem total** se sua ordem é  $n$ . Uma matriz  $m \times n$  tem **ordem total de colunas** se sua ordem é  $n$ . Uma propriedade fundamental de ordens é dada pelo teorema a seguir.

### Teorema 28.1

Uma matriz quadrada tem ordem total se e somente se ela é não singular. ■

Um **vetor nulo** para uma matriz  $A$  é um vetor diferente de zero  $x$  tal que  $Ax = 0$ . O teorema a seguir, cuja prova fica para o Exercício 28.1-9, e seu corolário relacionam as noções de ordem de colunas e singularidade a vetores nulos.

### **Teorema 28.2**

Uma matriz  $A$  tem ordem total de colunas se e somente se ela não tem um vetor nulo.

### **Corolário 28.3**

Uma matriz quadrada  $A$  é singular se e somente se ela tem um vetor nulo.

O  $ij$ -ésimo **menor** de uma matriz  $A$   $n \times n$ , para  $n > 1$ , é a matriz  $(n - 1) \times (n - 1) A_{[ij]}$  obtida pela eliminação da  $i$ -ésima linha e da  $j$ -ésima coluna de  $A$ . O **determinante** de uma matriz  $A$   $n \times n$  pode ser definido recursivamente em termos de seus menores por

$$\det(A) = \begin{cases} a_{11} & \text{se } n = 1, \\ \sum_{j=1}^n (-1)^{1+j} a_{1j} \det(A_{[1j]}) & \text{se } n > 1. \end{cases} \quad (28.7)$$

O termo  $(-1)^{1+j} \det(A_{[1j]})$  é conhecido como *co-fator* do elemento  $a_{1j}$ .

Os teoremas a seguir, cujas provas são omitidas aqui, expressam propriedades fundamentais do determinante.

### **Teorema 28.4 (Propriedades de determinantes)**

O determinante de uma matriz quadrada  $A$  tem as seguintes propriedades:

- Se qualquer linha ou qualquer coluna de  $A$  é zero, então  $\det(A) = 0$ .
- O determinante de  $A$  é multiplicado por  $\lambda$  se as entradas de qualquer linha única (ou de qualquer coluna única) de  $A$  são todas multiplicadas por  $\lambda$ .
- O determinante de  $A$  não se altera se as entradas em uma linha (respectivamente, coluna) são adicionadas às de outra linha (respectivamente, coluna).
- O determinante de  $A$  é igual ao determinante de  $A^T$ .
- O determinante de  $A$  é multiplicado por  $-1$  se duas linhas quaisquer (respectivamente, colunas) são trocadas.

Além disso, para quaisquer matrizes quadradas  $A$  e  $B$ , temos  $\det(AB) = \det(A)\det(B)$ .

### **Teorema 28.5**

Uma matriz  $A$   $n \times n$  é singular se e somente se  $\det(A) = 0$ .

## **Matrizes definidas como positivas**

As matrizes definidas como positivas desempenham um papel importante em muitas aplicações. Uma matriz  $A$   $n \times n$  é **definida como positiva** se  $x^T Ax > 0$  para todos os vetores de tamanho  $n \times 1 \neq 0$ . Por exemplo, a matriz identidade é definida como positiva pois, para qualquer vetor não nulo  $x = (x_1 \ x_2 \ \dots \ x_n)^T$ ,

$$x^T I_n x = x^T x$$

$$= \sum_{i=1}^n x_i^2$$

$$> 0.$$

Como veremos, matrizes que surgem em aplicações freqüentemente são definidas como positivas devido ao teorema a seguir.

### **Teorema 28.6**

Para qualquer matriz  $A$  com ordenação total de colunas, a matriz  $A^T A$  é definida como positiva.

**Prova** Devemos mostrar que  $x^T(A^T A)x > 0$  para qualquer vetor não nulo  $x$ . Para qualquer vetor  $x$ ,

$$\begin{aligned} x^T(A^T A)x &= (Ax)^T(Ax) \quad (\text{pelo Exercício 28.1-2}) \\ &= \|Ax\|^2. \end{aligned}$$

Observe que  $\|Ax\|^2$  é apenas a soma dos quadrados dos elementos do vetor  $Ax$ . Então,  $\|Ax\|^2 \geq 0$ . Se  $\|Ax\|^2 = 0$ , todo elemento de  $Ax$  é 0, o que significa que  $Ax = 0$ . Tendo em vista que  $A$  tem ordenação total de colunas,  $Ax = 0$  implica  $x = 0$ , pelo Teorema 28.2. Conseqüentemente,  $A^T A$  é definida como positiva. ■

Outras propriedades de matrizes definidas como positivas serão exploradas na Seção 28.5.

## Exercícios

### 28.1-1

Mostre que, se  $A$  e  $B$  são matrizes  $n \times n$  simétricas, então  $A + B$  e  $A - B$  também são simétricas.

### 28.1-2

Prove que  $(AB)^T = B^T A^T$  e que  $A^T A$  é sempre uma matriz simétrica.

### 28.1-3

Prove que matrizes inversas são únicas; isto é, se  $B$  e  $C$  são inversas de  $A$ , então  $B = C$ .

### 28.1-4

Prove que o produto de duas matrizes triangulares inferiores é triangular inferior. Prove que o determinante de uma matriz triangular inferior ou de uma matriz triangular superior é igual ao produto de seus elementos diagonais. Prove que a inversa de uma matriz triangular inferior, se existir, é triangular inferior.

### 28.1-5

Prove que, se  $P$  é uma matriz permutação  $n \times n$  e  $A$  é uma matriz  $n \times n$ , então  $PA$  pode ser obtida a partir de  $A$  pela permutação de suas linhas, e  $AP$  pode ser obtida a partir de  $A$  pela permutação de suas colunas. Prove que o produto de duas matrizes permutação é uma matriz permutação. Prove que, se  $P$  é uma matriz permutação, então  $P$  é invertível, sua inversa é  $P^T$ , e  $P^T$  é uma matriz permutação.

### 28.1-6

Sejam  $A$  e  $B$  matrizes  $n \times n$  tais que  $AB = I$ . Prove que, se  $A'$  é obtida a partir de  $A$  pela adição da linha  $j$  à linha  $i$ , então a inversa  $B'$  de  $A'$  pode ser obtida subtraindo-se a coluna  $i$  da coluna  $j$  de  $B$ .

### 28.1-7

Seja  $A$  uma matriz não singular  $n \times n$  com entradas complexas. Mostre que toda entrada de  $A^{-1}$  é real se e somente se toda entrada de  $A$  é real.

### 28.1-8

Mostre que, se  $A$  é uma matriz não singular simétrica  $n \times n$ , então  $A^{-1}$  é simétrica. Mostre que, se  $B$  é uma matriz arbitrária  $m \times n$ , então a matriz  $m \times m$  dada pelo produto  $BAB^T$  é simétrica.

### 28.1-9

Prove o Teorema 28.2. Ou seja, mostre que uma matriz  $A$  tem ordem total de colunas se e somente se  $Ax = 0$  implica  $x = 0$ . (Sugestão: Expresse a dependência linear de uma coluna sobre as outras como uma equação de vetores de matrizes.)

### 28.1-10

Prove que, para duas matrizes compatíveis quaisquer  $A$  e  $B$ ,

$$\text{ordem}(AB) \leq \min(\text{ordem}(A), \text{ordem}(B)) ,$$

onde a igualdade se mantém se  $A$  ou  $B$  é uma matriz quadrada não singular. (Sugestão: Use a definição alternativa de ordem de uma matriz.)

### 28.1-11

Dados números  $x_0, x_1, \dots, x_{n-1}$ , prove que o determinante da **matriz de Vandermonde**

$$V(x_0, x_1, \dots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

é

$$\det(V(x_0, x_1, \dots, x_{n-1})) = \prod_{0 \leq j < k \leq n-1} (x_k - x_j).$$

(Sugestão: Multiplique a coluna  $i$  por  $-x_0$  e adicione-a à coluna  $i + 1$  para  $i = n-1, n, n-2, \dots, 1$ , e depois use indução.)

## 28.2 Algoritmo de Strassen para multiplicação de matrizes

Esta seção apresenta o importante algoritmo recursivo de Strassen para multiplicar matrizes  $n \times n$  que é executado no tempo  $\Theta(n^{\lg 7}) = O(n^{2.81})$ . Então, para  $n$  suficientemente grande, ele supera o algoritmo simples  $\Theta(n^3)$  de multiplicação de matrizes MATRIX-MULTIPLY da Seção 25.1.

### Uma visão geral do algoritmo

O algoritmo de Strassen pode ser visto como uma aplicação de uma técnica de projeto familiar: dividir e conquistar. Vamos supor que desejamos calcular o produto  $C = AB$ , onde  $A$ ,  $B$  e  $C$  são matrizes  $n \times n$ . Supondo que  $n$  é uma potência exata de 2, dividimos  $A$ ,  $B$  e  $C$  em quatro matrizes  $n/2 \times n/2$  cada uma, reescrevendo a equação  $C = AB$  da seguinte maneira:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix}. \quad (28.8)$$

(O Exercício 28.2-2 lida com a situação em que  $n$  não é uma potência exata de 2.) A equação (28.9) corresponde às quatro equações

$$r = ae + bg, \quad (28.9)$$

$$s = af + bh, \quad (28.10)$$

$$t = ce + dg, \quad (28.11)$$

$$u = cf + db. \quad (28.12)$$

Cada uma dessas quatro equações especifica duas multiplicações de matrizes  $n/2 \times n/2$  e a adição de seus produtos  $n/2 \times n/2$ . Usando essas equações para definir uma estratégia direta de dividir e conquistar, derivamos a recorrência a seguir referente ao tempo  $T(n)$  para multiplicar duas matrizes  $n \times n$ :

$$T(n) = 8T(n/2) + \Theta(n^2). \quad (28.13)$$

Infelizmente, a recorrência (28.13) tem a solução  $T(n) = \Theta(n^3)$ , e portanto esse método não é mais rápido que o comum.

Strassen descobriu uma abordagem recursiva diferente que exige apenas 7 multiplicações recursivas de matrizes  $n/2 \times n/2$  e  $\Theta(n^2)$  adições e subtrações escalares, produzindo a recorrência

$$\begin{aligned} T(n) &= 7T(n/2) + \Theta(n^2) \\ &= \Theta(n^{\lg 7}) \\ &= \Theta(n^{2.81}). \end{aligned} \quad (28.14)$$

O método do Strassen tem quatro passos:

1. Dividir as matrizes de entrada  $A$  e  $B$  em submatrizes  $n/2 \times n/2$ , como na equação (28.8).
2. Usando  $\Theta(n^2)$  adições e subtrações escalares, calcular 14 matrizes  $A_1, A_2, \dots, A_7, B_1, B_2, \dots, B_7$ , cada uma das quais é  $n/2 \times n/2$ .
3. Calcular recursivamente os sete produtos de matrizes  $P_i = A_i B_i$  para  $i = 1, 2, \dots, 7$ .
4. Calcular as submatrizes desejadas  $r, s, t, u$  da matriz resultado  $C$ , somando e/ou subtraindo várias combinações das  $P_i$  matrizes, usando apenas  $\Theta(n^2)$  adições e subtrações escalares.

Tal procedimento satisfaz à recorrência (28.14). Tudo o que temos a fazer agora é preencher os detalhes que faltam.

### Determinação dos produtos de submatrizes

Não está muito claro o modo como Strassen descobriu os produtos de submatrizes que são a chave para fazer seu algoritmo funcionar. Aqui, vamos reconstruir um método de descoberta plausível.

Vamos supor que cada produto de matrizes  $P_i$  possa ser escrito na forma

$$\begin{aligned} P_i &= A_i B_i \\ &= (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d) \cdot (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h), \end{aligned} \quad (28.15)$$

onde os coeficientes  $\alpha_{ij}, \beta_{ij}$  são todos extraídos do conjunto  $\{-1, 0, 1\}$ . Isto é, supomos que cada produto é calculado pela adição ou subtração de alguma das submatrizes de  $A$ , adicionando ou subtraindo algumas das submatrizes de  $B$ , e depois multiplicando os dois resultados. Embora estratégias mais gerais sejam possíveis, essa estratégia simples funciona.

Se formarmos todos os nossos produtos dessa maneira, poderemos usar esse método recursivamente, sem supor a comutatividade da multiplicação, pois cada produto tem todas as submatrizes  $A$  à esquerda e todas as submatrizes  $B$  à direita. Essa propriedade é essencial para a aplicação recursiva desse método, pois a multiplicação de matrizes não é comutativa.

Por conveniência, usaremos matrizes  $4 \times 4$  para representar combinações lineares de produtos de submatrizes, onde cada produto combina uma submatriz de  $A$  com uma submatriz de  $B$ , como na equação (28.15). Por exemplo, podemos reescrever a equação (28.9) como

$$\begin{aligned} r &= ae + bg \\ &= (a \ b \ c \ d) \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} \end{aligned}$$

$$= \begin{matrix} & e & f & g & h \\ a & + & \cdot & \cdot & \cdot \\ b & \cdot & \cdot & + & \cdot \\ c & \cdot & \cdot & \cdot & \cdot \\ d & \cdot & \cdot & \cdot & \cdot \end{matrix}.$$

A última expressão utiliza uma notação abreviada na qual “+” representa +1, “·” representa 0, e “−” representa −1. (De agora em diante, omitiremos as identificações de linhas e colunas.) Usando essa notação, temos as equações a seguir para outras submatrizes da matriz resultado  $C$ :

$$s = af + bh$$

$$= \begin{pmatrix} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

$$t = ce + dg$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{pmatrix},$$

$$u = cf + db$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Começamos nossa busca de um algoritmo mais rápido para multiplicação de matrizes observando que a submatriz  $s$  pode ser calculada como  $s = P_1 + P_2$ , onde  $P_1$  e  $P_2$  são calculados usando uma multiplicação de matrizes cada:

$$\begin{aligned} P_1 &= A_1 B_1 \\ &= a \cdot (f - b) \\ &= af - ab \end{aligned}$$

$$= \begin{pmatrix} \cdot & + & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

$$\begin{aligned} P_2 &= A_2 B_2 \\ &= (a - b) \cdot b \\ &= ab + bb \end{aligned}$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

A matriz  $t$  pode ser calculada de maneira semelhante como  $t = P_3 + P_4$ , onde

$$\begin{aligned} P_3 &= A_3 B_3 \\ &= (c + d) \cdot e \\ &= ce + de \end{aligned}$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}.$$

e

$$\begin{aligned} P_4 &= A_4 B_4 \\ &= d \cdot (g - e) \\ &= dg - de \end{aligned}$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{pmatrix}.$$

Vamos definir um **termo essencial** como um dos oito termos que aparecem no lado direito de uma das equações (28.9) a (28.12). Agora, usamos 4 produtos para calcular as duas submatrizes  $s$  e  $t$  cujos termos essenciais são  $af$ ,  $bb$ ,  $ce$  e  $dg$ . Observe que  $P_1$  calcula o termo essencial  $af$ ,  $P_2$  calcula o termo essencial  $bb$ ,  $P_3$  calcula o termo essencial  $ce$  e  $P_4$  calcula o termo essencial  $dg$ . Desse modo, só falta calcular as duas submatrizes  $r$  e  $u$  restantes, cujos termos essenciais são os termos diagonais  $ae$ ,  $bg$ ,  $cf$  e  $db$ , sem utilizar mais de 3 produtos adicionais. Agora, tentaremos a inovação  $P_5$  para calcular dois termos essenciais de uma vez:

$$\begin{aligned} P_5 &= A_5 B_5 \\ &= (a + d) \cdot (e + b) \\ &= ae + ab + de + db \end{aligned}$$

$$= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}.$$

Além de calcular os dois termos essenciais  $ae$  e  $db$ ,  $P_5$  calcula os termos não essenciais  $0ab$  e  $de$ , que precisam ser cancelados de alguma maneira. Podemos usar  $P_4$  e  $P_2$  para cancelá-los, mas dois outros termos não essenciais aparecem então:

$$P_5 + P_4 - P_2 = ae + db + dg - db$$

$$= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \end{pmatrix}.$$

Porém, acrescentando um produto adicional

$$\begin{aligned} P_6 &= A_6 B_6 \\ &= (b - d) \cdot (g + b) \\ &= bg + bb - dg - db \end{aligned}$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{pmatrix},$$

obtemos

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ &= ae + bg \end{aligned}$$

$$= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Podemos obter  $u$  de maneira semelhante a partir de  $P_5$  usando  $P_1$  e  $P_3$  para mover os termos não essenciais de  $P_5$  em uma direção diferente:

$$P_5 + P_1 - P_2 = ae + af - ce + db$$

$$= \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

## Subtraindo um produto adicional

$$\begin{aligned} P_7 &= A_7 B_7 \\ &= (a - c) \cdot (e + f) \\ &= ae + af - ce - cf \end{aligned}$$

$$= \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & - & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

obtemos agora

$$\begin{aligned} u &= P_5 + P_1 - P_3 - P_7 \\ &= cf + db \end{aligned}$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Os 7 produtos de submatrizes  $P_1, P_2, \dots, P_7$  podem desse modo ser usados para calcular o produto  $C = AB$ , o que completa a descrição do método de Strassen.

## Discussão

Do ponto de vista prático, o algoritmo de Strassen freqüentemente não é o método de escolha para multiplicação de matrizes, por quatro razões:

1. O fator constante oculto no tempo de execução do algoritmo de Strassen é maior que o fator constante no método simples  $\Theta(n^3)$ .
2. Quando as matrizes são esparsas, métodos específicos para matrizes esparsas são mais rápidos.
3. O algoritmo de Strassen não é tão estável numericamente quanto o método simples.
4. As submatrizes formadas nos níveis de recursão consomem espaço.

As duas últimas razões foram minimizadas em torno de 1990. Higham [145] demonstrou que a diferença de estabilidade numérica foi superenfatizada; embora o algoritmo de Strassen seja numericamente instável demais para algumas aplicações, ele está dentro de limites aceitáveis para outras. Bailey *et al.* [30] discutem técnicas para reduzir os requisitos de memória para o algoritmo de Strassen.

Na prática, implementações de multiplicação rápida de matrizes para matrizes densas usam o algoritmo de Strassen com tamanhos de matrizes acima de um “ponto de passagem” e trocam para o método simples depois que o tamanho do subproblema se reduz a dimensões menores que o ponto de passagem. O valor exato do ponto de passagem é altamente dependente do sistema. As análises que levam em conta operações mas ignoram efeitos de caches e canalização produziram pontos de passagem baixos como  $n = 8$  (por Higham [145]) ou  $n = 12$  (por Huss-Lederman *et al.* [163]). Medidas empíricas em geral produzem pontos de passagem mais

altos, sendo alguns baixos como  $n = 20$  ou próximos desse valor. Para qualquer sistema dado, normalmente é simples determinar o ponto de passagem por experimentação.

Usando técnicas avançadas além do escopo deste texto, é possível de fato multiplicar matrizes  $n \times n$  em tempo melhor que  $\Theta(n^{\lg 7})$ . O melhor limite superior atual é aproximadamente  $O(n^{2.376})$ . O melhor limite inferior conhecido é apenas o óbvio limite  $\Omega(n^2)$  (óbvio porque temos de preencher  $n^2$  elementos da matriz produto). Desse modo, não sabemos no momento o quanto a multiplicação de matrizes é realmente difícil.

## Exercícios

### 28.2-1

Use o algoritmo de Strassen para calcular o produto de matrizes

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}.$$

Mostre o seu trabalho.

### 28.2-2

Como você modificaria o algoritmo de Strassen para multiplicar matrizes  $n \times n$  nas quais  $n$  não é uma potência exata de 2? Mostre que o algoritmo resultante funciona no tempo  $\Theta(n^{\lg 7})$ .

### 28.2-3

Qual é o maior  $k$  tal que, se você puder multiplicar matrizes  $3 \times 3$  usando  $k$  multiplicações (sem supor a comutatividade da multiplicação), então você poderá multiplicar matrizes  $n \times n$  no tempo  $O(n^{\lg 7})$ ? Qual seria o tempo de execução desse algoritmo?

### 28.2-4

V. Pan descobriu um modo de multiplicar matrizes  $68 \times 68$  usando 132.464 multiplicações, um modo de multiplicar matrizes  $70 \times 70$  usando 143.640 multiplicações e um modo de multiplicar matrizes  $72 \times 72$  usando 155.424 multiplicações. Que método produz o melhor tempo de execução assintótico quando usado em um algoritmo de multiplicação de matrizes de dividir e conquistar? Compare-o com o tempo de execução para o algoritmo de Strassen.

### 28.2-5

Com que rapidez é possível multiplicar uma matriz  $kn \times n$  por uma matriz  $n \times kn$ , usando o algoritmo de Strassen como uma sub-rotina? Responda à mesma pergunta com a ordem das matrizes de entrada invertida.

### 28.2-6

Mostre como multiplicar os números complexos  $a + bi$  e  $c + di$  usando apenas três multiplicações reais. O algoritmo deve tomar  $a, b, c$  e  $d$  como entrada e produzir o componente real  $ac - bd$  e o componente imaginário  $ad + bc$  separadamente.

## 28.3 Resolução de sistemas de equações lineares

Resolver um conjunto de equações lineares simultâneas é um problema fundamental que ocorre em diversas aplicações. Um sistema linear pode ser expresso como uma equação matricial, na qual cada elemento da matriz ou do vetor pertence a um campo, em geral os números reais  $\mathbb{R}$ . Esta seção descreve como resolver um sistema de equações lineares usando um método chamado decomposição LUP.

Começamos com um conjunto de equações lineares em  $n$  incógnitas  $x_1, x_2, \dots, x_n$ :

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2, \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n &= b_n, \end{aligned} \tag{28.16}$$

Um conjunto de valores para  $x_1, x_2, \dots, x_n$  que satisfaz a todas as equações (28.16) simultaneamente é dito uma *solução* para essas equações. Nesta seção, trataremos apenas o caso em que existem exatamente  $n$  equações em  $n$  incógnitas.

Podemos reescrever de forma conveniente as equações (28.16) como a equação de vetores de matrizes

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

ou, de forma equivalente, fazendo  $A = (a_{ij})$ ,  $x = (x_i)$  e  $b = (b_i)$ , como

$$Ax = b. \tag{28.17}$$

Se  $A$  é não singular, ela possui uma inversa  $A^{-1}$ , e

$$x = A^{-1}b \tag{28.18}$$

é o vetor solução. Podemos provar que  $x$  é a única solução para a equação (28.17) da maneira ilustrada a seguir. Se existem duas soluções,  $x$  e  $x'$ , então  $Ax = Ax' = b$  e

$$\begin{aligned} X &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= A^{-1}(A)x' \\ &= x'. \end{aligned}$$

Nesta seção, nossa preocupação predominante será o caso em que  $A$  é não singular ou, de modo equivalente (pelo Teorema 28.1), a ordem de  $A$  é igual ao número  $n$  de incógnitas. Contudo, existem outras possibilidades, as quais merecem uma breve discussão. Se o número de equações é menor que o número  $n$  de incógnitas – ou, de modo mais geral, se a ordem de  $A$  é menor que  $n$  – então o sistema é *subdeterminado*. Em geral, um sistema subdeterminado tem um número infinito de soluções, embora possa não ter nenhuma solução, se as equações forem incompatíveis. Se o número de equações excede o número  $n$  de incógnitas, o sistema é *superdeterminado*, e é possível que não exista nenhuma solução. Encontrar boas soluções aproximadas para sistemas de equações lineares superdeterminados é um problema importante que discutimos na Seção 28.5.

Vamos retornar ao nosso problema de resolver o sistema  $Ax = b$  de  $n$  equações em  $n$  incógnitas. Uma abordagem é calcular  $A^{-1}$ , e depois multiplicar ambos os lados por  $A^{-1}$ , produzindo  $A^{-1}Ax = A^{-1}b$ , ou  $x = A^{-1}b$ . Essa abordagem sofre na prática de instabilidade numérica. Felizmente, existe outra abordagem – a decomposição LUP – que é numericamente estável e tem a vantagem adicional de ser mais rápida na prática.

## Visão geral da decomposição LUP

A idéia por trás da decomposição LUP é encontrar três matrizes  $L$ ,  $U$  e  $P$   $n \times n$  tais que

$$PA = LU, \quad (28.19)$$

onde

- $L$  é uma matriz triangular inferior unitária,
- $U$  é uma matriz triangular superior, e
- $P$  é uma matriz permutação.

Chamamos as matrizes  $L$ ,  $U$  e  $P$  que satisfazem à equação (28.19) uma **decomposição LUP** da matriz  $A$ . Mostraremos que toda matriz não singular  $A$  possui tal decomposição.

A vantagem de calcular uma decomposição LUP para a matriz  $A$  é que sistemas lineares podem ser resolvidos mais prontamente quando são triangulares, como é o caso das matrizes  $L$  e  $U$ . Tendo encontrado uma decomposição LUP para  $A$ , podemos resolver a equação (28.17)  $Ax = b$ , resolvendo apenas sistemas lineares triangulares da maneira mostrada a seguir. A multiplicação de ambos os lados de  $Ax = b$  por  $P$  produz a equação equivalente  $PAx = Pb$  que, pelo Exercício 28.1-5, resulta em permutar as equações (28.16). Usando nossa decomposição (28.19), obtemos

$$LUX = Pb.$$

Podemos agora solucionar essa equação resolvendo dois sistemas lineares triangulares. Definimos  $y = Ux$ , onde  $x$  é o vetor solução desejado. Primeiro, resolvemos o sistema triangular inferior

$$Ly = Pb \quad (28.20)$$

para o vetor incógnita  $y$  por um método chamado “substituição direta”. Tendo encontrado a solução para  $y$ , resolvemos então o sistema triangular superior

$$Ux = y \quad (28.21)$$

para a incógnita  $x$  por um método chamado “substituição inversa”. O vetor  $x$  é nossa solução para  $Ax = b$ , pois a matriz permutação  $P$  é invertível (Exercício 28.1-5):

$$\begin{aligned} Ax &= P^{-1}LUX \\ &= P^{-1}Ly \\ &= P^{-1}Pb \\ &= b. \end{aligned}$$

Nosso próximo passo é mostrar como funcionam a substituição direta e a substituição inversa, e depois atacar o problema de calcular a própria decomposição LUP.

## Substituição direta e inversa

A **substituição direta** pode resolver o sistema triangular inferior (28.20) no tempo  $\Theta(n^2)$ , dados  $L$ ,  $P$  e  $b$ . Por conveniência, representamos a permutação  $P$  de forma compacta por um arranjo  $\pi[1..n]$ . Para  $i = 1, 2, \dots, n$ , a entrada  $\pi[i]$  indica que  $P_{i,\pi[i]} = 1 = 1$  e  $P_{ij} = 0$  para  $j \neq \pi[i]$ . Desse modo,  $PA$  tem  $a_{\pi[i]}$  na linha  $i$  e na coluna  $j$ , e  $Pb$  tem  $b_{\pi[i]}$  como seu  $i$ -ésimo elemento. Tendo em vista que  $L$  é triangular inferior unitária, a equação (28.20) pode ser reescrita como

$$\begin{aligned} y_1 &= b_{\pi[1]}, \\ l_{21}y_1 + y_2 &= b_{\pi[2]}, \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]}, \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + \dots + l_{n3}y_3 + \dots + y_n &= b_{\pi[n]}. \end{aligned}$$

Podemos resolver diretamente para  $y_1$ , pois a primeira equação nos informa que  $y_1 = b_{\pi[1]}$ . Tendo resolvido para  $y_1$ , podemos substituir seu valor na segunda equação, formando

$$y_2 = b_{\pi[2]} - l_{21}y_1.$$

Agora, podemos substituir  $y_1$  e  $y_2$  na terceira equação, obtendo

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2).$$

Em geral, substituímos  $y_1, y_2, \dots, y_{i-1}$  de forma “direta” na  $i$ -ésima equação para obter  $y_i$ :

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j.$$

A **substituição inversa** é semelhante à substituição direta. Dados  $U$  e  $y$ , resolvemos primeiro a  $n$ -ésima equação e trabalhamos em sentido contrário até a primeira equação. Como a substituição direta, esse processo é executado no tempo  $\Theta(n^2)$ . Tendo em vista que  $U$  é triangular superior, podemos reescrever o sistema (28.21) como

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \dots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1, \\ u_{22}x_2 + \dots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2, \\ &\vdots \\ u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2}, \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1}, \\ u_{nn}x_n &= y_n. \end{aligned}$$

Desse modo, podemos resolver para  $x_n, x_{n-1}, \dots, x_1$ , com sucesso, da seguinte maneira:

$$x_n = y_n/u_{nn},$$

$$x_{n-1} = (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1},$$

$$x_{n-2} = (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2},$$

$\vdots$

ou, em geral,

$$x_i = \left( y_i - \sum_{j=i+1}^n u_{ij} x_j \right) / u_{ii}.$$

Dados  $P, L, U$  e  $b$ , o procedimento LUP-SOLVE resolve a equação para  $x$ , combinando a substituição direta e a inversa. O pseudocódigo pressupõe que a dimensão  $n$  aparece no atributo `linhas[L]` e que a matriz permutação  $P$  é representada pelo arranjo  $\pi$ .

**LUP-SOLVE( $L, U, \pi, b$ )**

```

1  $n \leftarrow \text{linhas}[L]$ 
2 for  $i \leftarrow 1$  to  $n$ 
3   do  $y_i \leftarrow b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j$ 
4 for  $i \leftarrow n$  downto 1
5   do  $x_i \leftarrow \left( y_i - \sum_{j=i+1}^n u_{ij} x_j \right) / u_{ii}$ 
6 return  $x$ 
```

O procedimento LUP-SOLVE resolve para  $y$  usando a substituição direta nas linhas 2 e 3, e depois resolve para  $x$  usando a substituição inversa nas linhas 4 e 5. Como existe um loop implícito nos somatórios dentro de cada um dos loops **for**, o tempo de execução é  $\Theta(n^2)$ .

Como um exemplo desses métodos, considere o sistema de equações lineares definido por

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix},$$

onde

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix},$$

$$b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix},$$

e desejamos resolver para a incógnita  $x$ . A decomposição LUP é

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0,2 & 1 & 0 \\ 0,6 & 0,5 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0,8 & -0,6 \\ 0 & 0 & 2,5 \end{pmatrix},$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

(O leitor pode verificar que  $PA = LU$ .) Usando a substituição direta, resolvemos  $Ly = Pb$  para  $y$ :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0,2 & 1 & 0 \\ 0,6 & 0,5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix},$$

e obtemos

$$y = \begin{pmatrix} 8 \\ 1,4 \\ 1,5 \end{pmatrix}$$

calculando primeiro  $y_1$ , depois  $y_2$  e finalmente  $y_3$ . Usando a substituição inversa, resolvemos  $Ux = y$  para  $x$ :

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0,8 & -0,6 \\ 0 & 0 & 2,5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1,4 \\ 1,5 \end{pmatrix},$$

obtendo dessa maneira a resposta desejada

$$x = \begin{pmatrix} -1,4 \\ 2,2 \\ 0,6 \end{pmatrix},$$

calculando primeiro  $x_3$ , depois  $x_2$  e finalmente  $x_1$ .

## Calculando uma decomposição LU

Agora mostramos que, se uma decomposição LUP pode ser calculada para uma matriz não singular  $A$ , a substituição direta e inversa pode ser usada para resolver o sistema  $Ax = b$  de equações lineares. Resta mostrar como uma decomposição LUP para  $A$  pode ser encontrada de forma eficiente. Começamos com o caso em que  $A$  é uma matriz não singular  $n \times n$  e  $P$  está ausente (ou, de forma equivalente,  $P = I_n$ ). Nesse caso, devemos encontrar uma fatoração  $A = LU$ . Chamamos as duas matrizes  $L$  e  $U$  uma **decomposição LU** de  $A$ .

O processo pelo qual executamos a decomposição LU é chamado **eliminação gaussiana**. Começamos subtraíndo das outras equações os múltiplos da primeira equação, de modo que a primeira variável seja removida dessas equações. Em seguida, subtraímos da terceira equação e das equações subsequentes os múltiplos da segunda equação, de modo que agora a primeira e a segunda variáveis sejam removidas dessas equações. Continuamos esse processo até o sistema que resta ter uma forma triangular superior – de fato, ele é a matriz  $U$ . A matriz  $L$  é formada pelos multiplicadores de linhas que fazem as variáveis serem eliminadas.

Nosso algoritmo para implementar essa estratégia é recursivo. Desejamos construir uma decomposição LU para uma matriz não singular  $A$   $n \times n$ . Se  $n = 1$ , então terminamos, pois podemos escolher  $L = I_1$  e  $U = A$ . Para  $n > 1$ , dividimos  $A$  em quatro partes:

$$A = \left( \begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array} \right)$$

$$= \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$$

onde  $v$  é um vetor coluna de tamanho  $(n - 1)$ ,  $w^T$  é um vetor linha de tamanho  $(n - 1)$  e  $A'$  é uma matriz  $(n - 1) \times (n - 1)$ . Então, usando a álgebra de matrizes (verifique as equações simplesmente efetuando as multiplicações), podemos fatorar  $A$  como

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - w^T/a_{11} \end{pmatrix}. \end{aligned} \quad (28.22)$$

Os valores 0 na primeira e na segunda matriz da fatoração são vetores linha e coluna, respectivamente, de tamanho  $n - 1$ . O termo  $vw^T/a_{11}$ , formado tomando-se o produto externo de  $v$  e  $w$  e dividindo-se cada elemento do resultado por  $a_{11}$ , é uma matriz  $(n - 1) \times (n - 1)$ , que corresponde em tamanho à matriz  $A'$  da qual ela é subtraída. A matriz  $(n - 1) \times (n - 1)$  resultante

$$A' - vw^T/a_{11} \quad (28.23)$$

é chamada **complemento de Schur** de  $A$  com relação a  $a_{11}$ .

Afirmamos que, se  $A$  é não singular, então o complemento de Schur também é não singular. Por quê? Suponha que o complemento de Schur, que é  $(n - 1) \times (n - 1)$ , é singular. Então, pelo Teorema 28.1, ele tem ordem de linhas estritamente menor que  $n - 1$ . Como as  $n - 1$  entradas inferiores na primeira coluna da matriz

$$\begin{pmatrix} a_{11} & w^T \\ 0 & A' - w^T/a_{11} \end{pmatrix}$$

são todas 0, as  $n - 1$  linhas inferiores dessa matriz devem ter ordem estritamente menor que  $n - 1$ . A ordem de linhas da matriz inteira é então estritamente menor que  $n$ . Aplicando o Exercício 28.1-10 à equação (28.22),  $A$  tem ordem estritamente menor que  $n$  e, pelo Teorema 28.1, derivamos a contradição de que  $A$  é singular.

Como o complemento de Schur é não singular, agora podemos encontrar recursivamente uma decomposição LU desse complemento. Digamos que

$$A' - vw^T/a_{11} = L'U',$$

onde  $L'$  é triangular inferior unitária e  $U'$  é triangular superior. Então, usando a álgebra de matrizes, temos

$$\begin{aligned}
A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & L'U' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & U' \end{pmatrix} \\
&= LU,
\end{aligned}$$

fornecendo assim nossa decomposição LU. (Observe que, como  $L'$  é uma matriz triangular inferior unitária,  $L$  também é e, como  $U'$  é triangular superior,  $U$  também é.)

É claro que, se  $a_{11} = 0$ , esse método não funciona, porque ele divide por 0. Ele também não funciona se a entrada superior esquerda do complemento de Schur  $A' - vw^T/a_{11}$  é 0, pois dividimos por ela no passo seguinte da recursão. Os elementos pelos quais dividimos durante a decomposição LU são chamados *pivôs*, e eles ocupam os elementos diagonais da matriz  $U$ . A razão para incluirmos uma matriz permutação  $P$  durante a decomposição LUP é que ela nos permite evitar a divisão por elementos zero. O uso de permutações para evitar a divisão por 0 (ou por números pequenos) é chamado *emprego de pivôs*.

Uma classe importante de matrizes para as quais a decomposição LU sempre funciona corretamente é a classe de matrizes simétricas definidas como positivas. Tais matrizes não exigem nenhum emprego de pivôs, e assim a estratégia recursiva descrita antes pode ser empregada sem medo de dividir por 0. Provaremos esse resultado, bem como vários outros, na Seção 28.5.

Nosso código para decomposição LU de uma matriz  $A$  decorre da estratégia recursiva, a não ser pelo fato de que um loop de iteração substitui a recursão. (Essa transformação é uma otimização padrão para um procedimento de “final recursivo” – um procedimento cuja última operação é uma chamada recursiva a ele próprio.) Ele pressupõe que a dimensão de  $A$  é mantida no atributo *linhas*[ $A$ ]. Como sabemos que a matriz de saída  $U$  tem valores 0 abaixo da diagonal, e como LUP-SOLVE não examina essas entradas, o código não se preocupa em preenchê-las. Da mesma forma, como a matriz de saída  $L$  tem valores 1 em sua diagonal e valores 0 acima da diagonal, essas entradas também não são preenchidas. Desse modo, o código calcula apenas as entradas “significativas” de  $L$  e  $U$ .

#### LU-DECOMPOSITION( $A$ )

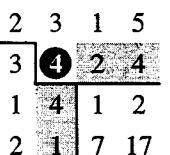
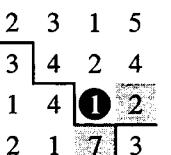
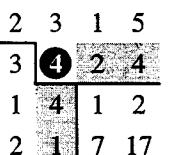
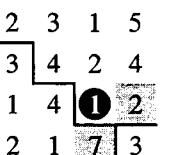
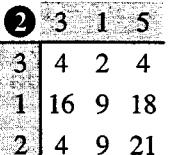
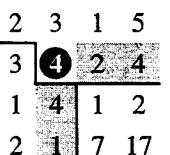
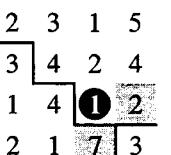
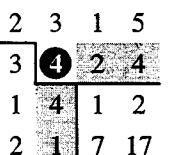
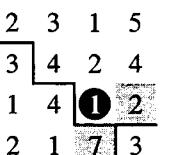
```

1  $n \leftarrow \text{linhas}[A]$ 
2 for  $k \leftarrow 1$  to  $n$ 
3   do  $u_{kk} \leftarrow a_{kk}$ 
4   for  $i \leftarrow k + 1$  to  $n$ 
5     do  $l_{ik} \leftarrow a_{ik}/u_{kk}$      $\triangleright l_{ik}$  contém  $v_i$ 
6      $u_{ki} \leftarrow a_{ki}$             $\triangleright u_{ki}$  contém  $w_i^T$ 
7   for  $i \leftarrow k + 1$  to  $n$ 
8     do for  $j \leftarrow k + 1$  to  $n$ 
9       do  $a_{ij} \leftarrow a_{ij} - l_{ik}u_{kj}$ 
10  return  $L$  e  $U$ 

```

O loop **for** externo que começa na linha 2 itera-se uma vez para cada passo recursivo. Dentro desse loop, o pivô é determinado como  $u_{kk} = a_{kk}$  na linha 3. Dentro do loop **for** nas linhas 4 a 6 (que não é executado quando  $k = n$ ), os vetores  $v$  e  $w^T$  são usados para atualizar  $L$  e  $U$ . Os elemen-

tos do vetor  $v$  são determinados na linha 5, onde  $v_i$  é armazenado em  $l_{ik}$ , e os elementos do vetor  $w^T$  são determinados na linha 6, onde  $w_i^T$  é armazenado em  $u_{ki}$ . Finalmente, os elementos do complemento de Schur são calculados nas linhas 7 a 9 e armazenados de volta na matriz  $A$ . (Não precisamos dividir por  $a_{kk}$  na linha 9, porque já o fizemos quando calculamos  $l_{ik}$  na linha 5.) Pelo fato de a linha 9 ter aninhamento triplo, LU-DECOMPOSITION é executado no tempo  $\Theta(n^3)$ .

|            |                                                                                   |                                                                                   |                                                                                    |
|------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| 2 3 1 5    |  |  |  |
| 6 13 5 19  |  |  |  |
| 2 19 10 23 |  |  |  |
| 4 10 11 31 |  |  |  |
| (a)        | (b)                                                                               | (c)                                                                               | (d)                                                                                |

$$\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

(e)

FIGURA 28.1 A operação de LU-DECOMPOSITION. (a) A matriz  $A$ . (b) O elemento  $a_{11} = 2$  no círculo preto é o pivô, a coluna sombreada é  $v/a_{11}$ , e a linha sombreada é  $w^T$ . Os elementos de  $U$  calculados até agora estão acima da linha horizontal, e os elementos de  $L$  estão à esquerda da linha vertical. A matriz complemento de Schur  $A' - vw^T/a_{11}$  ocupa a parte inferior direita. (c) Agora operamos sobre a matriz complemento de Schur produzida a partir da parte (b). O elemento  $a_{22} = 4$  no círculo preto é o pivô, e a coluna e a linha sombreadas são  $v/a_{22}$  e  $w^T$  (no particionamento do complemento de Schur), respectivamente. As linhas dividem a matriz nos elementos de  $U$  calculados até agora (acima), os elementos de  $L$  calculados até agora (esquerda) e o novo complemento de Schur (direita inferior). (d) O próximo passo completa a fatoração. (O elemento 3 no novo complemento de Schur se torna parte de  $U$  quando a recursão termina.) (e) A fatoração  $A = LU$

A Figura 28.1 ilustra a operação de LU-DECOMPOSITION. Ela mostra uma otimização padrão do procedimento, na qual os elementos significativos de  $L$  e  $U$  são armazenados “no lugar” na matriz  $A$ . Isto é, podemos configurar uma correspondência entre cada elemento  $a_{ij}$  e  $l_{ij}$  (se  $i > j$ ) ou  $u_{ij}$  (se  $i \leq j$ ) e atualizar a matriz  $A$  para que ela contenha  $L$  e  $U$  quando o procedimento terminar. O pseudocódigo para essa otimização é obtido a partir do pseudocódigo anterior, pela simples substituição de cada referência a  $l$  ou  $u$  por  $a$ ; não é difícil verificar que essa transformação preserva a correção.

## Calculando uma decomposição LUP

Em geral, na resolução de um sistema de equações lineares  $Ax = b$ , devemos empregar pivôs de elementos fora da diagonal de  $A$  para evitar a divisão por 0. Não apenas a divisão por 0 é indesejável, mas também a divisão por qualquer valor pequeno, ainda que  $A$  seja não singular, porque instabilidades numéricas podem resultar da computação. Então, tentaremos empregar pivôs sobre um valor grande.

A matemática por trás da decomposição LUP é semelhante à da decomposição LU. Lembre-se de que temos uma matriz não singular  $A n \times n$  e desejamos encontrar uma matriz permutação  $P$ , uma matriz triangular inferior unitária  $L$  e uma matriz triangular superior  $U$ , tais que  $PA = LU$ . Antes de particionarmos a matriz  $A$ , como fizemos para a decomposição LU, movemos um elemento não nulo, digamos  $a_{k1}$ , da primeira coluna até a posição (1, 1) da matriz. (Se a primeira coluna contém apenas valores 0, então  $A$  é singular, porque seu determinante é 0, pelos Teore-

mas 28.4 e 28.5.) Para preservar o conjunto de equações, trocamos a linha 1 pela linha  $k$ , o que é equivalente a multiplicar  $A$  por uma matriz permutação  $Q$  à esquerda (Exercício 28.1-5). Desse modo, podemos escrever  $QA$  como

$$QA = \begin{pmatrix} \alpha_{k1} & w^T \\ v & A' \end{pmatrix},$$

onde  $v = (\alpha_{21}, \alpha_{31}, \dots, \alpha_{n1})$ , exceto pelo fato de que  $\alpha_{11}$  substitui  $\alpha_{k1}$ ;  $w^T = (\alpha_{k2}, \alpha_{k3}, \dots, \alpha_{kn})$ ; e  $A'$  é uma matriz  $(n-1) \times (n-1)$ . Como  $\alpha_{k1} \neq 0$ , podemos executar agora grande parte da mesma álgebra linear que usamos para a decomposição LU, mas desta vez garantindo que não faremos a divisão por 0:

$$\begin{aligned} QA &= \begin{pmatrix} \alpha_{k1} & w^T \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/\alpha_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} \alpha_{k1} & w^T \\ 0 & A' - vw^T/\alpha_{k1} \end{pmatrix}. \end{aligned}$$

Como vimos no caso da decomposição LU, se  $A$  é não singular, então o complemento de Schur  $A' = vw^T/\alpha_{k1}$  também é não singular. Consequentemente, podemos encontrar de forma induutiva uma decomposição LUP para o complemento de Schur, com a matriz triangular inferior unitária  $L'$ , a matriz triangular superior  $U'$  e a matriz permutação  $P'$ , tais que

$$P'(A' = vw^T/\alpha_{k1}) = L'U'.$$

Definimos

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q,$$

que é uma matriz permutação, pois é o produto de duas matrizes permutação (Exercício 28.1-5). Agora, temos

$$\begin{aligned} PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\ &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/\alpha_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} \alpha_{k1} & w^T \\ 0 & A' - vw^T/\alpha_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/\alpha_{k1} & P' \end{pmatrix} \begin{pmatrix} \alpha_{k1} & w^T \\ 0 & A' - vw^T/\alpha_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/\alpha_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} \alpha_{k1} & w^T \\ 0 & P'(A' - vw^T/\alpha_{k1}) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/\alpha_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} \alpha_{k1} & w^T \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/\alpha_{k1} & L' \end{pmatrix} \begin{pmatrix} \alpha_{k1} & w^T \\ 0 & U' \end{pmatrix} \\ &= LU, \end{aligned}$$

gerando a decomposição LUP. Pelo fato de  $L'$  ser triangular inferior unitária,  $L$  também é; como  $U'$  é triangular superior,  $U$  também é.

Observe que, nessa derivação, diferente daquela que ocorre na decomposição LU, tanto o vetor coluna  $v/a_{k1}$  quanto o complemento de Schur  $A' = vw^T/a_{k1}$  devem ser multiplicados pela matriz permutação  $P'$ .

Da mesma forma que LU-DECOMPOSITION, nosso pseudocódigo para decomposição LUP substitui a recursão por um loop de iteração. Como um melhoramento em relação à implementação direta da recursão, mantemos dinamicamente a matriz permutação  $P$  como um arranjo  $\pi$ , onde  $\pi[i] = j$  significa que a  $i$ -ésima linha de  $P$  contém 1 na coluna  $j$ . Também implementamos o código para calcular  $L$  e  $U$  “no lugar” na matriz  $A$ . Portanto, quando o procedimento termina,

$$a_{ij} = \begin{cases} l_{ij} & \text{se } i > j, \\ u_{ij} & \text{se } i \leq j. \end{cases}$$

#### LUP-DECOMPOSITION( $A$ )

```

1   $n \leftarrow \text{linhas}[A]$ 
2  for  $i \leftarrow 1$  to  $n$ 
3    do  $\pi[i] \leftarrow i$ 
4  for  $k \leftarrow 1$  to  $n$ 
5    do  $p \leftarrow 0$ 
6      for  $i \leftarrow k$  to  $n$ 
7        do if  $|a_{ik}| > p$ 
8          then  $p \leftarrow |a_{ik}|$ 
9             $k' \leftarrow i$ 
10       if  $p = 0$ 
11         then error “matriz singular”
12       trocar  $\pi[k] \leftrightarrow \pi[k']$ 
13       for  $i \leftarrow 1$  to  $n$ 
14         do trocar  $a_{ki} \leftrightarrow a_{k'i}$ 
15       for  $i \leftarrow k + 1$  to  $n$ 
16         do  $a_{ik} \leftarrow a_{ik}/a_{kk}$ 
17         for  $j \leftarrow k + 1$  to  $n$ 
18           do  $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$ 
```

A Figura 28.2 ilustra como LUP-DECOMPOSITION fatora uma matriz. O arranjo  $\pi$  é inicializado pelas linhas 2 e 3 para representar a permutação identidade. O loop **for** externo que começa na linha 4 implementa a recursão. Cada vez que passamos pelo loop externo, as linhas 5 a 9 determinam o elemento  $a_{k'k}$  com maior valor absoluto dentre os que estão na primeira coluna atual (a coluna  $k$ ) da matriz  $(n-k+1) \times (n-k+1)$  cuja decomposição LU deve ser encontrada. Se todos os elementos na primeira coluna atual são zero, as linhas 10 e 11 informam que a matriz é singular. Para empregar o pivô, trocamos  $\pi[k']$  por  $\pi[k]$  na linha 12 e trocamos a  $k$ -ésima e a  $k'$ -ésima linha de  $A$  nas linhas 13 e 14, formando assim o elemento pivô  $a_{kk'}$ . (As linhas inteiras são trocadas porque, na derivação do método anterior, não apenas  $A' = vw^T/a_{k1}$  é multiplicado por  $P'$ , mas também  $v/a_{k1}$ .) Finalmente, o complemento de Schur é calculado pelas linhas 15 a 18, de maneira muito semelhante ao modo como ele é calculado pelas linhas 4 a 9 de LU-DECOMPOSITION, exceto pelo fato de que aqui a operação é escrita para funcionar “no lugar”.

Devido à sua estrutura de loop com aninhamento triplo, o tempo de execução de LUP-DECOMPOSITION é  $\Theta(n^3)$ , o mesmo de LU-DECOMPOSITION. Desse modo, o emprego de pivôs nos custa no máximo um fator constante em tempo.

$$\begin{array}{|c|cccc|} \hline & 1 & 2 & 0 & 2 & 0,6 \\ \hline 1 & 2 & 3 & 3 & 4 & -2 \\ 2 & 3 & 5 & 4 & 2 & \\ 3 & 5 & -1 & -2 & 3,4 & -1 \\ 4 & -1 & & & & \\ \hline \end{array}$$

(a)

$$\begin{array}{|c|cccc|} \hline & 3 & 5 & 4 & 2 \\ \hline 1 & 2 & 3 & 3 & 4 & -2 \\ 2 & 1 & 2 & 0 & 2 & 0,6 \\ 3 & 4 & 1 & -2 & 3,4 & -1 \\ 4 & & & & & \\ \hline \end{array}$$

(b)

$$\begin{array}{|c|cccc|} \hline & 3 & 5 & 4 & 2 \\ \hline 1 & 2 & 0,6 & 0 & 1,6 & -3,2 \\ 2 & 1 & 0,4 & -2 & 0,4 & -0,2 \\ 3 & 4 & -0,2 & -1 & 4,2 & -0,6 \\ 4 & & & & & \\ \hline \end{array}$$

(c)

$$\begin{array}{|c|cccc|} \hline & 3 & 5 & 4 & 2 \\ \hline 1 & 2 & 0,6 & 0 & 1,6 & -3,2 \\ 2 & 1 & 0,4 & -2 & 0,4 & -0,2 \\ 3 & 4 & -0,2 & -1 & 4,2 & -0,6 \\ 4 & & & & & \\ \hline \end{array}$$

(d)

$$\begin{array}{|c|cccc|} \hline & 3 & 5 & 4 & 2 \\ \hline 1 & 2 & 0,4 & -2 & 0,4 & -0,2 \\ 2 & 1 & 0,6 & 0 & 1,6 & -3,2 \\ 3 & 4 & -0,2 & -1 & 4,2 & -0,6 \\ 4 & & & & & \\ \hline \end{array}$$

(e)

$$\begin{array}{|c|cccc|} \hline & 3 & 5 & 4 & 2 \\ \hline 1 & 2 & 0,4 & -2 & 0,4 & -0,2 \\ 2 & 1 & 0,6 & 0 & 1,6 & -3,2 \\ 3 & 4 & -0,2 & 0,5 & 4 & -0,5 \\ 4 & & & & & \\ \hline \end{array}$$

(f)

$$\begin{array}{|c|cccc|} \hline & 3 & 5 & 4 & 2 \\ \hline 1 & 2 & 0,4 & -2 & 0,4 & -0,2 \\ 2 & 1 & 0,6 & 0 & 1,6 & -3,2 \\ 3 & 4 & -0,2 & 0,5 & 4 & -0,5 \\ 4 & & & & & \\ \hline \end{array}$$

(g)

$$\begin{array}{|c|cccc|} \hline & 3 & 5 & 4 & 2 \\ \hline 1 & 2 & 0,4 & -2 & 0,4 & -0,2 \\ 2 & 1 & 0,6 & 0 & 1,6 & -3,2 \\ 3 & 4 & -0,2 & 0,5 & 4 & -0,5 \\ 4 & & & & & \\ \hline \end{array}$$

(h)

$$\begin{array}{|c|cccc|} \hline & 3 & 5 & 4 & 2 \\ \hline 1 & 2 & 0,4 & -2 & 0,4 & -0,2 \\ 2 & 1 & 0,6 & 0 & 0,4 & -3 \\ 3 & 4 & -0,2 & 0,5 & 4 & -0,5 \\ 4 & & & & & \\ \hline \end{array}$$

(i)

$$\begin{array}{l} P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \quad A = \begin{pmatrix} 2 & 0 & 2 & 0,6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3,4 & -1 \end{pmatrix}, \quad L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0,4 & 1 & 0 & 0 \\ -0,2 & 0,5 & 1 & 0 \\ 0,6 & 0 & 0,4 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 5 & 5 & 4 & 2 \\ 0 & -2 & 0,4 & -0,2 \\ 0 & 0 & 4 & -0,5 \\ 0 & 0 & 0 & -3 \end{pmatrix} \end{array}$$

(j)

**FIGURA 28.2** A operação de LUP-DECOMPOSITION. (a) A matriz de entrada  $A$  com a permutação identidade das linhas à esquerda. O primeiro passo do algoritmo determina que o elemento 5 em preto na terceira linha é o pivô para a primeira coluna. (b) As linhas 1 e 3 são trocadas e a permutação é atualizada. A coluna e a linha sombreadas representam  $v$  e  $w^T$ . (c) O vetor  $v$  é substituído por  $v/5$ , e a parte inferior direita da matriz é atualizada com o complemento de Schur. As linhas dividem a matriz em três regiões: elementos de  $U$  (acima), elementos de  $L$  (esquerda) e elementos do complemento de Schur (inferior direita). (d)–(f) O segundo passo. (g)–(i) O terceiro passo. Nenhuma outra mudança ocorre no quarto e último passo. (j) A decomposição LUP  $PA = LU$

## Exercícios

### 28.3-1

Resolva a equação

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

usando substituição direta.

### 28.3-2

Encontre uma decomposição LU da matriz

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}.$$

### 28.3-3

Resolva a equação

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

usando uma decomposição LUP.

### 28.3-4

Descreva a decomposição LUP de uma matriz diagonal.

### 28.3-5

Descreva a decomposição LUP de uma matriz permutação  $A$ , e prove que ela é única.

### 28.3-6

Mostre que, para todo  $n \geq 1$ , existe uma matriz singular  $n \times n$  que tem uma decomposição LU.

### 28.3-7

Em LU-DECOMPOSITION, é necessário executar a iteração do loop for mais externo quando  $k = n$ ? E em LUP-DECOMPOSITION?

## 28.4 Inversão de matrizes

Embora na prática as inversas de matrizes não sejam utilizadas de modo geral para resolver sistemas de equações lineares, preferindo-se em vez disso usar técnicas numericamente mais estáveis como a decomposição LUP, às vezes é necessário calcular a inversa de uma matriz. Nesta seção, mostraremos como a decomposição LUP pode ser usada para calcular a inversa de uma matriz. Também provaremos que a multiplicação de matrizes e o cálculo da inversa de uma matriz são problemas de dificuldade equivalente, no sentido de que (sujeitos a condições técnicas) podemos usar um algoritmo para um com o objetivo de resolver o outro no mesmo tempo de execução assintótico. Desse modo, podemos usar o algoritmo de Strassen para multiplicação de matrizes com a finalidade de inverter uma matriz. De fato, o artigo original de Strassen foi motivado pelo problema de mostrar que um conjunto de equações lineares poderia ser resolvido com maior rapidez que pela utilização do método habitual.

### Como calcular a inversa de uma matriz a partir de uma decomposição LUP

Vamos supor que temos uma decomposição LUP de uma matriz  $A$  na forma de três matrizes  $L$ ,  $U$  e  $P$  tais que  $PA = LU$ . Usando LUP-SOLVE, podemos resolver uma equação da forma  $Ax = b$  no tempo  $\Theta(n^2)$ . Tendo em vista que a decomposição LUP depende de  $A$ , mas não de  $b$ , podemos executar LUP-SOLVE sobre um segundo conjunto de equações da forma  $Ax = b'$  no tempo adicional  $\Theta(n^2)$ . Em geral, uma vez que tenhamos a decomposição LUP de  $A$ , podemos resolver, no tempo  $\Theta(kn^2)$ ,  $k$  versões da equação  $Ax = b$  que diferem apenas em  $b$ .

A equação

$$AX = I_n \tag{28.24}$$

pode ser vista como um conjunto de  $n$  equações distintas da forma  $Ax = b$ . Essas equações definem a matriz  $X$  como a inversa de  $A$ . Para ser preciso, seja  $X_i$  a  $i$ -ésima coluna de  $X$ , e lembre-se de que o vetor unitário  $e_i$  é a  $i$ -ésima coluna de  $I_n$ . A equação (28.24) pode então ser resolvida para  $X$ , usando-se a decomposição LUP de  $A$  para resolver cada equação

$$AX_i = e_i$$

separadamente para  $X_i$ . Cada uma das  $n$  colunas  $X_i$  pode ser encontrada no tempo  $\Theta(n^2)$ , e assim a computação de  $X$  a partir da decomposição LUP de  $A$  demora o tempo  $\Theta(n^3)$ . Como a decomposição LUP de  $A$  pode ser calculada no tempo  $\Theta(n^3)$ , a inversa  $A^{-1}$  de uma matriz  $A$  pode ser determinada no tempo  $\Theta(n^3)$ .

## Multiplicação de matrizes e inversão de matrizes

Agora, mostraremos que as acelerações teóricas obtidas para multiplicação de matrizes se traduzem em acelerações na inversão de matrizes. De fato, provamos algo mais forte: a inversão de matrizes é equivalente à multiplicação de matrizes, no sentido a seguir. Se  $M(n)$  denota o tempo para multiplicar duas matrizes  $n \times n$ , então existe um modo de inverter uma matriz  $n \times n$  no tempo  $O(M(n))$ . Além disso, se  $I(n)$  denota o tempo para inverter uma matriz não singular  $n \times n$ , então existe um modo de multiplicar duas matrizes  $n \times n$  no tempo  $O(I(n))$ . Provaremos esses resultados em dois teoremas separados.

### **Teorema 28.7 (A multiplicação não é mais difícil que a inversão)**

Se podemos inverter uma matriz  $n \times n$  no tempo  $I(n)$ , onde  $\Omega(n^2)$  e  $I(n)$  satisfaz à condição de regularidade  $I(3n) = O(I(n))$ , então podemos multiplicar duas matrizes  $n \times n$  no tempo  $O(I(n))$ .

**Prova** Sejam  $A$  e  $B$  matrizes  $n \times n$  cujo produto de matrizes  $C$  desejamos calcular. Definimos a matriz  $D$   $3n \times 3n$  por

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

A inversa de  $D$  é

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix},$$

e portanto podemos calcular o produto  $AB$  tomado a submatriz superior direita  $n \times n$  de  $D^{-1}$ .

Podemos construir a matriz  $D$  no tempo  $\Theta(n^2) = O(I(n))$ , e podemos inverter  $D$  no tempo  $O(I(3n)) = O(I(n))$ , pela condição de regularidade sobre  $I(n)$ . Temos então  $M(n) = O(I(n))$ .

Observe que  $I(n)$  satisfaz à condição de regularidade sempre que  $I(n) = \Theta(n^c \lg^d n)$  para quaisquer constantes  $c > 0$ ,  $d \geq 0$ .

A prova de que a inversão de matrizes não é mais difícil que a multiplicação de matrizes se baseia em algumas propriedades de matrizes simétricas definidas como positivas que serão provadas na Seção 28.5.

### **Teorema 28.8 (A inversão não é mais difícil que a multiplicação)**

Suponha que podemos multiplicar duas matrizes reais  $n \times n$  no tempo  $M(n)$ , onde  $M(n) = \Omega(n^2)$  e  $M(n)$  satisfaz às duas condições de regularidade  $M(n+k) = O(M(n))$  para qualquer  $k$  no intervalo  $0 \leq k \leq n$  e  $M(n/2) \leq cM(n)$  para alguma constante  $c < 1/2$ . Então, podemos calcular a inversa de qualquer matriz real não singular  $n \times n$  no tempo  $O(M(n))$ .

**Prova** Podemos supor que  $n$  é uma potência exata de 2, pois temos

$$598 \quad \left( \begin{array}{cc} A & 0 \\ 0 & I_k \end{array} \right)^{-1} = \left( \begin{array}{cc} A^{-1} & 0 \\ 0 & I_k \end{array} \right)$$

para qualquer  $k > 0$ . Desse modo, escolhendo  $k$  tal que  $n + k$  seja uma potência de 2, aumentamos a matriz até um tamanho que seja a próxima potência de 2 e obtemos a resposta desejada  $A^{-1}$  a partir da resposta ao problema aumentado. A primeira condição de regularidade sobre  $M(n)$  assegura que esse aumento não provoca o aumento do tempo de execução por mais que um fator constante.

Por enquanto, vamos supor que a matriz  $A$   $n \times n$  é simétrica e definida como positiva. Particionamos  $A$  em quatro submatrizes  $n/2 \times n/2$ :

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix}. \quad (28.25)$$

Em seguida, se fizermos

$$S = D - CB^{-1}C^T \quad (28.26)$$

ser o complemento de Schur de  $A$  com relação a  $B$  (veremos mais detalhes sobre essa forma do complemento de Schur na Seção 28.5), teremos

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^T S^{-1}CB^{-1} & -B^{-1}C^T S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}, \quad (28.27)$$

pois  $AA^{-1} = I_n$ , como podemos verificar executando a multiplicação de matrizes. As matrizes  $B^{-1}$  e  $S^{-1}$  existem se  $A$  é simétrica e definida como positiva, pelos Lemas 28.9, 28.10 e 28.11 da Seção 28.5, porque tanto  $B$  quanto  $S$  são simétricas e definidas como positivas. Pelo Exercício 28.1-2,  $B^{-1}C^T = (CB^{-1})^T$  e  $B^{-1}C^T S^{-1} = (S^{-1}CB^{-1})^T$ . As equações (28.26) e (28.27) podem então ser usadas para especificar um algoritmo recursivo envolvendo 4 multiplicações de matrizes  $n/2 \times n/2$ :

$$\begin{aligned} & C \cdot B^{-1}, \\ & (CB^{-1}) \cdot C^T, \\ & S^{-1} \cdot (CB^{-1}), \\ & (CB^{-1})^T \cdot (S^{-1}CB^{-1}). \end{aligned}$$

Desse modo, podemos inverter uma matriz  $n \times n$  simétrica definida como positiva invertendo duas matrizes  $n/2 \times n/2$  ( $B$  e  $S$ ), executando essas quatro multiplicações de matrizes  $n/2 \times n/2$  (o que podemos fazer com um algoritmo para matrizes  $n \times n$ ), mais um custo adicional  $O(n^2)$  para extrair submatrizes de  $A$  e executar um número constante de adições e subtrações sobre essas matrizes  $n/2 \times n/2$ . Obtemos a recorrência

$$\begin{aligned} I(n) & \leq 2I(n/2) + 4M(n) + O(n^2) \\ & = 2I(n/2) + \Theta(M(n)) \\ & = O(M(n)). \end{aligned}$$

A segunda linha é válida porque  $M(n) = \Omega(n^2)$ , e a terceira linha se segue porque a segunda condição de regularidade na declaração do teorema nos permite aplicar o caso 3 do teorema mestre (Teorema 4.1).

Resta provar que o tempo de execução assintótico da multiplicação de matrizes pode ser obtido para inversão de matrizes quando  $A$  é invertível, mas não simétrica e definida como positiva. A idéia básica é que, para qualquer matriz não singular  $A$ , a matriz  $A^T A$  é simétrica (pelo Exercício 28.1-2) e definida como positiva (pelo Teorema 28.6). Então, o artifício é reduzir o problema de inverter  $A$  ao problema de inverter  $A^T A$ .

A redução é baseada na observação de que, quando  $A$  é uma matriz não singular  $n \times n$ , temos

$$A^{-1} = (A^T A)^{-1} A^T,$$

pois  $((A^T A)^{-1} A^T)A = (A^T A)^{-1} (A^T A) = I_n$  e uma matriz inversa é única. Então, podemos calcular  $A^{-1}$  primeiro multiplicando  $A^T$  por  $A$  para obter  $A^T A$ , e depois inverter a matriz simétrica definida como positiva  $A^T A$  empregando o algoritmo de dividir e conquistar anterior e, finalmente, multiplicando o resultado por  $A^T$ . Cada um desses três passos demora o tempo  $O(M(n))$  e, desse modo, qualquer matriz não singular com entradas reais pode ser invertida no tempo  $O(M(n))$ . ■

A prova do Teorema 28.8 sugere um meio de resolver a equação  $Ax = b$  sem empregar pivôs, desde que  $A$  seja não singular. Multiplicamos ambos os lados da equação por  $A^T$ , produzindo  $(A^T A)x = A^T b$ . Essa transformação não afeta a solução  $x$ , pois  $A^T$  é invertível, e assim podemos fatorar a matriz simétrica definida como positiva  $A^T A$  calculando uma decomposição LU. Usamos então a substituição direta e inversa, a fim de resolver para  $x$  com o lado direito  $A^T b$ . Embora esse método esteja teoricamente correto, na prática o procedimento LUP-DECOMPOSITION funciona muito melhor. A decomposição LUP exige menos operações aritméticas por um fator constante, e tem propriedades numéricas um pouco melhores.

## Exercícios

### 28.4-1

Seja  $M(n)$  o tempo para multiplicar matrizes  $n \times n$ , e seja  $S(n)$  o tempo necessário para elevar ao quadrado uma matriz  $n \times n$ . Mostre que a multiplicação e a elevação de matrizes ao quadrado têm essencialmente a mesma dificuldade: um algoritmo de multiplicação de matrizes de tempo  $M(n)$  implica um algoritmo de elevação ao quadrado de tempo  $O(M(n))$ , e um algoritmo de elevação ao quadrado de tempo  $S(n)$  implica um algoritmo de multiplicação de matrizes de tempo  $O(S(n))$ .

### 28.4-2

Seja  $M(n)$  o tempo para multiplicar matrizes  $n \times n$ , e seja  $L(n)$  o tempo para calcular a decomposição LUP de uma matriz  $n \times n$ . Mostre que a multiplicação de matrizes e o cálculo de decomposições LUP de matrizes têm essencialmente a mesma dificuldade: um algoritmo de multiplicação de matrizes de tempo  $M(n)$  implica um algoritmo de decomposição LUP de tempo  $O(M(n))$ , e um algoritmo de decomposição LUP de tempo  $L(n)$  implica um algoritmo de multiplicação de matrizes de tempo  $O(L(n))$ .

### 28.4-3

Seja  $M(n)$  o tempo para multiplicar matrizes  $n \times n$ , e seja  $D(n)$  o tempo necessário para encontrar o determinante de uma matriz  $n \times n$ . Mostre que a multiplicação de matrizes e o cálculo do determinante têm essencialmente a mesma dificuldade: um algoritmo de multiplicação de matrizes de tempo  $M(n)$  implica um algoritmo de determinante de tempo  $O(M(n))$ , e um algoritmo de determinante de tempo  $D(n)$  implica um algoritmo de multiplicação de matrizes de tempo  $O(D(n))$ .

### 28.4-4

Seja  $M(n)$  o tempo para multiplicar matrizes booleanas  $n \times n$ , e seja  $T(n)$  o tempo para encontrar o fecho transitivo de matrizes booleanas  $n \times n$ . (Veja a Seção 25.2.) Mostre que um algoritmo de multiplicação de matrizes booleanas de tempo  $M(n)$  implica um algoritmo de fecho transitivo de tempo  $O(M(n) \lg n)$ , e um algoritmo de fecho transitivo de tempo  $T(n)$  implica um algoritmo de multiplicação de matrizes booleanas de tempo  $O(T(n))$ .

#### 28.4-5

O algoritmo de inversão de matrizes baseado no Teorema 28.8 funciona quando elementos de matrizes são retirados do campo de inteiros módulo 2? Explique.

#### 28.4-6 \*

Generalize o algoritmo de inversão de matrizes do Teorema 28.8 para tratar matrizes de números complexos, e prove que sua generalização funciona corretamente. (*Sugestão:* Em vez da transposta de  $A$ , use a **transposta conjugada**  $A^*$ , que é obtida a partir da transposta de  $A$  pela substituição de toda entrada por seu conjugado complexo. Em vez de matrizes simétricas, considere matrizes **hermitianas**, que são matrizes  $A$  tais que  $A = A^*$ .)

## 28.5 Matrizes simétricas definidas como positivas e aproximação de mínimos quadrados

As matrizes simétricas definidas como positivas têm muitas propriedades interessantes e desejáveis. Por exemplo, elas são não singulares, e a decomposição LU pode ser executada sobre elas sem que tenhamos de nos preocupar com a divisão por 0. Nesta seção, provaremos várias outras propriedades importantes de matrizes simétricas definidas como positivas e mostraremos uma aplicação interessante para a adaptação de curvas por uma aproximação de mínimos quadrados.

A primeira propriedade que provaremos talvez seja a mais básica.

### Lema 28.9

Qualquer matriz simétrica definida como positiva é não singular.

**Prova** Suponha que uma matriz  $A$  seja singular. Então, pelo Corolário 28.3, existe um vetor não nulo  $x$  tal que  $Ax = 0$ . Conseqüentemente,  $x^T Ax = 0$ , e  $A$  não pode ser definida como positiva. ■

A prova de que podemos executar a decomposição LU sobre uma matriz simétrica definida como positiva  $A$  sem dividir por 0 é mais complicada. Começamos provando propriedades sobre certas submatrizes de  $A$ . Defina a  $k$ -ésima **submatriz inicial** de  $A$  como a matriz  $A_k$  que consiste na interseção das primeiras  $k$  linhas e das primeiras  $k$  colunas de  $A$ .

### Lema 28.10

Se  $A$  é uma matriz simétrica definida como positiva, então toda submatriz inicial de  $A$  é simétrica e definida como positiva.

**Prova** É óbvio que cada submatriz inicial  $A_k$  é simétrica. Para provar que  $A_k$  é definida como positiva, supomos que ela não é e derivamos uma contradição. Se  $A_k$  não é definida como positiva, então existe um vetor  $x_k \neq 0$  de tamanho  $k$  tal que  $x_k^T A_k x_k \leq 0$ . Sendo  $A$   $n \times n$ , definimos o vetor  $x = (x_k^T 0)^T$  de tamanho  $n$ , onde existem  $n - k$  valores 0 seguindo  $x_k$ . Então, temos

$$\begin{aligned} x_k^T A x &= (x_k^T \quad 0) \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \begin{pmatrix} x_k \\ 0 \end{pmatrix} \\ &= (x_k^T \quad 0) \begin{pmatrix} A_k x_k \\ B x_k \end{pmatrix} \\ &= x_k^T A_k x_k \\ &\leq 0, \end{aligned}$$

o que contradiz o fato de  $A$  ser definida como positiva. ■

Agora, voltamos a algumas propriedades essenciais do complemento de Schur. Seja  $A$  uma matriz simétrica definida como positiva, e seja  $A_k$  uma submatriz inicial  $k \times k$  de  $A$ . Particionamos  $A$  como

$$A = \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}. \quad (28.28)$$

Generalizamos a definição (28.23) para definir o **complemento de Schur** de  $A$  com relação a  $A_k$  como

$$S = C - B A_k^{-1} B^T \quad (28.29)$$

(Pelo Lema 28.10,  $A_k$  é simétrica e definida como positiva; portanto,  $A_k^{-1}$  existe pelo Lema 28.9, e  $S$  é bem definida.) Observe que nossa primeira definição (28.23) do complemento de Schur é coerente com a definição (28.29), fazendo-se  $k = 1$ .

O próximo lema mostra que as matrizes de complemento de Schur de matrizes simétricas definidas como positivas são elas próprias simétricas e definidas como positivas. Esse resultado foi usado no Teorema 28.8, e seu corolário é necessário para provar a correção da decomposição LU no caso de matrizes simétricas definidas como positivas.

### Lema 28.11 (Lema do complemento de Schur)

Se  $A$  é uma matriz simétrica definida como positiva e  $A_k$  é uma submatriz inicial  $k \times k$  de  $A$ , então o complemento de Schur de  $A$  com relação a  $A_k$  é simétrica e definida como positiva.

**Prova** Como  $A$  é simétrica, então a submatriz  $C$  também é. Pelo Exercício 28.1-8, o produto  $B A_k^{-1} B^T$  é simétrico e, pelo Exercício 28.1-1,  $S$  é simétrica.

Resta mostrar que  $S$  é definida como positiva. Considere a partição de  $A$  dada na equação (28.28). Para qualquer vetor não nulo  $x$ , temos  $x^T A x > 0$  pela hipótese de que  $A$  é definida como positiva. Vamos dividir  $x$  em dois subvetores  $y$  e  $z$  compatíveis com  $A_k$  e  $C$ , respectivamente. Como  $A_k^{-1}$  existe, temos

$$\begin{aligned} x^T A x &= (y^T \ z^T) \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\ &= (y^T \ z^T) \begin{pmatrix} A_k y + B^T z \\ B y + C z \end{pmatrix} \\ &= y^T A_k y + y^T B^T z + z^T B y + z^T C z \\ &= (y + A_k^{-1} B^T z)^T A_k (y + A_k^{-1} B^T z) + z^T (C - B A_k^{-1} B^T) z, \end{aligned} \quad (28.30)$$

por mágica matricial. (Verifique efetuando multiplicações.) Essa última equação serve para “completar o quadrado” da forma quadrática. (Ver Exercício 28.5-2.)

Tendo em vista que  $x^T A x > 0$  é válida para qualquer  $x$  não nulo, vamos escolher qualquer  $z$  não nulo e depois escolher  $y = -A_k^{-1} B^T z$ , o que faz o primeiro termo da equação (28.30) desaparecer, restando

$$z^T (C - B A_k^{-1} B^T) z = z^T S z$$

como o valor da expressão. Para qualquer  $z \neq 0$ , temos então  $z^T S z = x^T A x > 0$ , e portanto  $S$  é definida como positiva. ■

### Corolário 28.12

A decomposição LU de uma matriz simétrica definida como positiva nunca provoca uma divisão por 0.

**Prova** Seja  $A$  uma matriz simétrica definida como positiva. Provaremos algo mais forte que o enunciado do corolário: todo pivô é estritamente positivo. O primeiro pivô é  $a_{11}$ . Seja  $e_1$  o primeiro vetor unitário, do qual obtemos  $a_{11} = e_1^T A e_1 > 0$ . Tendo em vista que o primeiro passo da decomposição LU produz o complemento de Schur de  $A$  com relação a  $A_1 = (a_{11})$ , o Lema 28.11 implica que todos os pivôs são positivos por indução. ■

### Aproximação de mínimos quadrados

O ajuste de curvas a determinados conjuntos de pontos de dados é uma aplicação importante de matrizes simétricas definidas como positivas. Suponha que temos um conjunto de  $m$  pontos de dados

$$(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m),$$

onde os valores  $y_i$  estão reconhecidamente sujeitos a erros de medição. Gostaríamos de determinar uma função  $F(x)$  tal que

$$y_i = F(x_i) + n_i, \quad (28.31)$$

para  $i = 1, 2, \dots, m$ , onde os erros de aproximação  $n_i$  são pequenos. A forma da função  $F$  depende do problema em questão. Aqui, vamos supor que ela tem a forma de uma soma linearmente ponderada,

$$F(x) = \sum_{j=1}^n c_j f_j(x),$$

onde o número de termos (somandos)  $n$  e as *funções de base*  $f_j$  específicas são escolhidas com base no conhecimento do problema em questão. Uma opção comum é  $f_j(x) = x^{j-1}$ , o que significa que

$$F(x) = c_1 + c_2 x + c_3 x^2 + \dots + c_n x^{n-1}$$

é um polinômio de grau  $n - 1$  em  $x$ .

Escolhendo  $n = m$ , podemos calcular cada  $y_i$  exatamente na equação (28.31). Porém, essa função  $F$  de grau elevado “ajusta o ruído” bem como os dados, e em geral fornece resultados ruins quando usada com a finalidade de predizer  $y$  para valores de  $x$  não vistos anteriormente. Em geral, é melhor escolher  $n$  significativamente menor que  $m$  e esperar que, escolhendo bem os coeficientes  $c_j$ , possamos obter uma função  $F$  que encontre os padrões significativos nos pontos de dados sem prestar atenção indevida ao ruído. Existem alguns princípios teóricos para escolher  $n$ , mas eles estão além do escopo deste texto. Em todo caso, uma vez que  $n$  é escolhido, terminamos com um conjunto superdeterminado de equações cuja solução desejamos aproximar. Mostraremos agora como isso pode ser feito.

Seja

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix}$$

a matriz de valores das funções de base nos pontos dados; isto é,  $\alpha_{ij} = f_j(x_i)$ . Seja  $c = (c_k)$  o vetor de coeficientes de tamanho  $n$  desejado. Então,

$$Ac = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$$

$$= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

é o vetor de tamanho  $m$  de “valores previstos” para  $y$ . Desse modo,

$$\eta = Ac - y$$

é o vetor de tamanho  $m$  de **erros de aproximação**.

Para minimizar os erros de aproximação, optamos por minimizar a norma do vetor de erro  $\eta$ , o que nos dá uma solução de **mínimos quadrados**, pois

$$\|\eta\| = \left( \sum_{i=1}^m \eta_i^2 \right)^{1/2}.$$

Tendo em vista que

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^m \left( \sum_{j=1}^n \alpha_{ij} c_j - y_i \right)^2,$$

podemos minimizar  $\|\eta\|$  diferenciando  $\|\eta\|^2$  com relação a cada  $c_k$ , e então definindo o resultado como 0:

$$\frac{d\|\eta\|^2}{dc_k} = \sum_{i=1}^m 2 \left( \sum_{j=1}^n \alpha_{ij} c_j - y_i \right) \alpha_{ik} = 0. \quad (28.32)$$

As  $n$  equações (28.32) para  $k = 1, 2, \dots, n$  são equivalentes à única equação matricial

$$(Ac - y)^T A = 0$$

ou, de modo equivalente (usando o Exercício 28.1-2), a

$$A^T (Ac - y) = 0$$

que implica

$$604 \boxed{A^T A c = A^T y}. \quad (28.33)$$

Em estatística, isso se chama **equação normal**. A matriz  $A^T A$  é simétrica pelo Exercício 28.1-2 e, se  $A$  tem ordem total de colunas então, pelo Teorema 28.6,  $A^T A$  também é definida como positiva. Conseqüentemente,  $(A^T A)^{-1}$  existe, e a solução para a equação (28.33) é

$$\begin{aligned} c &= ((A^T A)^{-1} A^T y \\ &= A^+ y, \end{aligned} \tag{28.34}$$

onde a matriz  $A^+ = ((A^T A)^{-1} A^T)$  é chamada a **pseudo-inversa** da matriz  $A$ . A pseudo-inversa é uma generalização natural da noção de uma matriz inversa para o caso em que  $A$  é não quadrada. (Compare a equação (28.34) como a solução aproximada para  $Ac = y$  com a solução  $A^{-1}b$  como a solução exata para  $Ax = b$ .)

Como exemplo da produção de um ajuste de mínimos quadrados, vamos supor que temos 5 pontos de dados

$$(x_1, y_1) = (-1, 2),$$

$$(x_2, y_2) = (1, 1),$$

$$(x_3, y_3) = (2, 1),$$

$$(x_4, y_4) = (3, 0),$$

$$(x_5, y_5) = (5, 3),$$

mostrados como pontos pretos na Figura 28.3. Desejamos ajustar esses pontos com um polinômio quadrático

$$F(x) = c_1 + c_2 x + c_3 x^2.$$

Começamos com a matriz de valores de funções de base

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix},$$

cuja pseudo-inversa é

$$A^+ = \begin{pmatrix} 0,500 & 0,300 & 0,200 & 0,100 & -0,100 \\ -0,388 & 0,093 & 0,190 & 0,193 & 0,088 \\ 0,060 & -0,036 & -0,048 & -0,036 & 0,060 \end{pmatrix}$$

Multiplicando  $y$  por  $A^+$ , obtemos o vetor de coeficientes

$$c = \begin{pmatrix} 1,200 \\ -0,757 \\ 0,214 \end{pmatrix},$$

que corresponde ao polinômio quadrático

$$F(x) = 1,200 - 0,757x + 0,214x^2$$

como o ajuste quadrático mais próximo para os dados especificados, em um sentido de mínimos quadrados.

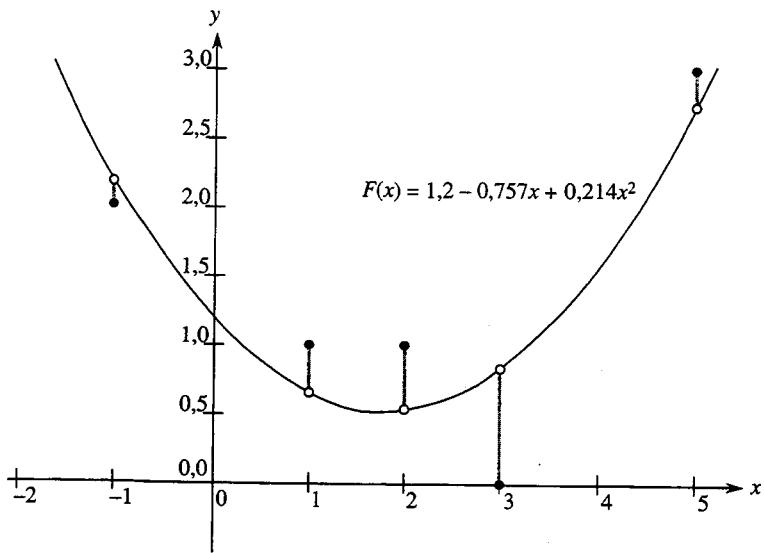


FIGURA 28.3 O ajuste de mínimos quadrados de um polinômio quadrático ao conjunto de pontos de dados  $\{(-1, 2), (1, 1), (2, 1), (3, 0), (5, 3)\}$ . Os pontos pretos são os pontos de dados, e os pontos brancos são seus valores estimados previstos pelo polinômio  $F(x) = 1,2 - 0,757x + 0,214x^2$ , o polinômio quadrático que minimiza a soma dos erros quadrados. O erro para cada ponto de dados é mostrado como uma linha sombreada

Como uma questão prática, resolvemos a equação normal (28.33) multiplicando  $y$  por  $A^T$ , e depois encontrando uma decomposição LU de  $A^T A$ . Se  $A$  tem ordem total, a matriz  $A^T A$  apresenta a garantia de ser não singular, porque é simétrica e definida como positiva. (Ver Exercício 28.1-2 e Teorema 28.6.)

## Exercícios

### 28.5-1

Prove que todo elemento da diagonal de uma matriz simétrica definida como positiva é positivo.

### 28.5-2

Seja  $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$  uma matriz  $2 \times 2$  simétrica definida como positiva. Prove que seu determinante  $ac - b^2$  é positivo “completando o quadrado” de maneira semelhante à que é usada na prova do Lema 28.11.

### 28.5-3

Prove que o elemento máximo em uma matriz simétrica definida como positiva reside na diagonal.

### 28.5-4

Prove que o determinante de cada submatriz inicial de uma matriz simétrica definida como positiva é positivo.

### 28.5-5

Seja  $A_k$  a  $k$ -ésima submatriz inicial de uma matriz simétrica definida como positiva  $A$ . Prove que  $\det(A_k)/\det(A_{k-1})$  é o  $k$ -ésimo pivô durante a decomposição LU onde, por convenção,  $\det(A_0) = 1$ .

### 28.5-6

Encontre a função da forma

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

que é o melhor ajuste de mínimos quadrados para os pontos de dados

### 28.5-7

Mostre que a pseudo-inversa  $A^+$  satisfaz às quatro equações a seguir:

$$AA^+A = A,$$

$$A^+AA^+ = A^+,$$

$$(AA^+)^T = AA^+,$$

$$(A^+A)^T = A^+A.$$

## Problemas

### 28-1 Sistemas tridiagonais de equações lineares

Considere a matriz tridiagonal

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

- a. Encontre uma decomposição LU de  $A$ .
- b. Resolva a equação  $Ax = (1 \ 1 \ 1 \ 1 \ 1)^T$ , usando substituição direta e inversa.
- c. Encontre a inversa de  $A$ .
- d. Mostre que, para qualquer matriz tridiagonal  $n \times n$  simétrica definida como positiva  $A$  e qualquer vetor  $b$  de  $n$  elementos, a equação  $Ax = b$  pode ser resolvida no tempo  $O(n)$ , executando-se uma decomposição LU. Demonstre que qualquer método baseado na formação de  $A^{-1}$  é assintoticamente mais dispendioso no pior caso.
- e. Mostre que, para qualquer matriz tridiagonal  $n \times n$  não singular  $A$  e qualquer vetor  $b$  de  $n$  elementos, a equação  $Ax = b$  pode ser resolvida no tempo  $O(n)$ , pela execução de uma decomposição LUP.

### 28-2 Curvas

Um método prático para interpolar um conjunto de pontos com uma curva é usar **curvas cúbicas**. Temos um conjunto  $\{(x_i, y_i) : i = 0, 1, \dots, n\}$  de  $n + 1$  pares de valores de pontos, onde  $x_0 < x_1 < \dots < x_n$ . Desejamos ajustar uma curva de forma cúbica  $f(x)$  aos pontos. Isto é, a curva  $f(x)$  é formada por  $n$  polinômios cúbicos  $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$  para  $i = 0, 1, \dots, n - 1$  onde, se  $x$  cair no intervalo  $x_i \leq x \leq x_{i+1}$ , então o valor da curva será dado por  $f(x) = f_i(x - x_i)$ . Os pontos  $x_i$ , nos quais os polinômios cúbicos são “colados” entre si são chamados **nós**. Por simplicidade, vamos supor que  $x_i = i$  para  $i = 0, 1, \dots, n$ .

Para assegurar a continuidade de  $f(x)$ , é necessário que

$$f(x_i) = f(0) = y_i,$$

$$f(x_{i+1}) = f(1) = y_{i+1}$$

para  $i = 0, 1, \dots, n - 1$ . Para assegurar que  $f(x)$  é suficientemente suave, também insistimos que deve haver continuidade da primeira derivada em cada nó:

$$f'(x_{i+1}) = f'_i(1) = f'_{i+1}(0)$$

para  $i = 0, 1, \dots, n - 1$ .

- a. Suponha que, para  $i = 0, 1, \dots, n$ , temos não apenas os pares de valores de pontos  $\{(x_i, y_i)\}$ , mas também as primeiras derivadas  $D_i = f'(x_i)$  em cada nó. Expressa cada coeficiente  $a_i, b_i, c_i$  e  $d_i$  em termos dos valores  $y_i, y_{i+1}, D_i$  e  $D_{i+1}$ . (Lembre-se de que  $x_i = i$ .) Com que rapidez os  $4n$  coeficientes podem ser calculados a partir dos pares de valores de pontos e das primeiras derivadas?

Ainda resta a dúvida sobre como escolher as primeiras derivadas de  $f(x)$  nos nós. Um método é exigir que as segundas derivadas sejam contínuas nos nós:

$$f''(x_{i+1}) = f''_i(1) = f''_{i+1}(0)$$

para  $i = 0, 1, \dots, n - 1$ . No primeiro e no último nó, supomos que  $f''(x_0) = f''_0(0)$  e  $f''(x_n) = f''_{n-1}(1)$ ; essas hipóteses fazem de  $f(x)$  uma curva cúbica **natural**.

- b. Use as restrições de continuidade sobre a segunda derivada para mostrar que, para  $i = 1, 2, \dots, n - 1$ ,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) \quad (28.35)$$

- c. Mostre que

$$2D_0 + D_1 = 3(y_1 - y_0), \quad (28.36)$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}). \quad (28.37)$$

- d. Reescreva as equações (28.35) a (28.37) como uma equação matricial envolvendo o vetor  $D = \langle D_0, D_1, \dots, D_n \rangle$  de incógnitas. Que atributos tem a matriz em sua equação?  
e. Demonstre que um conjunto de  $n + 1$  pares de valores de pontos pode ser interpolado com uma curva cúbica natural no tempo  $O(n)$  (ver Problema 28-1).  
f. Mostre como determinar uma curva cúbica natural que interpole um conjunto de  $n + 1$  pontos  $(x_i, y_i)$  satisfazendo a  $x_0 < i_1 < \dots < x_n$ , mesmo quando  $x_i$  não é necessariamente igual a  $i$ . Que equação matricial deve ser resolvida, e com que rapidez seu algoritmo é executado?

## Notas de capítulo

Existem muitos textos excelentes disponíveis que descrevem a computação numérica e científica com muito mais detalhes do que o espaço nos permite aqui. Os textos a seguir são especialmente interessantes: George e Liu [113], Golub e Van Loan [125], Press, Flannery, Teukolsky e Vetterling [248, 249], e ainda Strang [285, 286].

Golub e Van Loan [125] discutem a estabilidade numérica. Eles mostram por que  $\det(A)$  não é necessariamente um bom indicador da estabilidade de uma matriz  $A$ , propondo em vez disso o uso de  $\|A\|_\infty \|A^{-1}\|_\infty$ , onde  $\|A\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|$ . Eles também examinam a questão de como calcular esse valor sem na realidade calcular  $A^{-1}$ .

A publicação do algoritmo de Strassen em 1969 [287] provocou muito interesse. Antes disso, era difícil imaginar que o algoritmo simples pudesse ser melhorado. O limite superior assintótico sobre a dificuldade da multiplicação de matrizes foi consideravelmente melhorado desde então. O algoritmo assintoticamente mais eficiente que existe até hoje para multiplicar matrizes  $n \times n$ , devido a Coppersmith e Winograd [70], tem um tempo de execução  $O(n^{2.376})$ . A apresentação gráfica do algoritmo de Strassen se deve a Paterson [238].

A eliminação gaussiana, na qual se baseiam as decomposições LU e LUP, foi o primeiro método sistemático para resolução de sistemas de equações lineares. Ele também foi um dos primeiros algoritmos numéricos. Embora fosse conhecido antes, sua descoberta é comumente atribuída a C. F. Gauss (1777-1855). Em seu famoso artigo [287], Strassen também mostrou que uma matriz  $n \times n$  pode ser invertida no tempo  $O(n^{\lg 7})$ . Winograd [317] provou originalmente que a multiplicação de matrizes não é mais difícil que a inversão de matrizes, e a recíproca se deve a Aho, Hopcroft e Ullman [5].

Outra decomposição de matrizes importante é a *decomposição de valor singular*, ou SVD (singular value decomposition). Na SVD, uma matriz  $A$   $m \times n$  é fatorada em  $A = Q_1 \Sigma Q_2^T$ , onde  $\Sigma$  é uma matriz  $m \times n$  com valores não nulos apenas na diagonal,  $Q_1$  é  $m \times m$  com colunas mutuamente ortonormais, e  $Q_2$  é  $n \times n$ , também com colunas mutuamente ortonormais. Dois vetores são *ortonormais* se seu produto interno é 0 e cada vetor tem a norma 1. Os livros de Strang [285, 285] e de Golub e Van Loan [125] contêm bons tratamentos da SVD.

Strang [286] tem uma apresentação excelente de matrizes simétricas definidas como positivas e sobre álgebra linear em geral.

## *Capítulo 29*

# *Programação linear*

Muitos problemas podem ser formulados como a ação de maximizar ou minimizar um objetivo, sendo dados recursos limitados e restrições concorrentes. Se pudermos especificar o objetivo como uma função linear de certas variáveis e se pudermos especificar as restrições sobre recursos como igualdades ou desigualdades sobre essas variáveis, então teremos um **problema de programação linear**. Os programas lineares surgem em uma variedade de aplicações práticas. Começamos estudando uma aplicação em política eleitoral.

### **Um problema político**

Suponha que você seja um político tentando vencer uma eleição. Seu distrito tem três tipos diferentes de áreas – urbana, suburbana e rural. Essas áreas têm, respectivamente, 100.000, 200.000 e 50.000 eleitores registrados. Para governar de forma efetiva, você gostaria de ter a maioria dos votos em cada uma das três regiões. Você é honrado e nunca consideraria aceitar normas nas quais não acredita. Porém, percebe que certas questões podem ser mais influentes na obtenção de votos em certos lugares. Suas principais questões são a construção de mais estradas, o controle de armas, subsídios agrícolas e um imposto sobre combustíveis dedicado à melhoria do trânsito. De acordo com as pesquisas da suas equipes de campanha, é possível estimar quantos votos você ganhará ou perderá em cada segmento da população gastando \$1.000 na publicidade de cada questão. Essa informação aparece na tabela da Figura 29.1. Nessa tabela, cada entrada descreve o número de milhares de eleitores urbanos, suburbanos ou rurais que poderiam ser conquistados gastando-se \$1.000 em publicidade em defesa de uma questão específica. Entradas negativas denotam votos que seriam perdidos. Sua tarefa é descobrir a quantidade mínima de dinheiro que seria preciso gastar para obter 50.000 votos urbanos, 100.000 votos suburbanos e 25.000 votos rurais.

| política                   | urbana | suburbana | rural |
|----------------------------|--------|-----------|-------|
| construir estradas         | -2     | 5         | 3     |
| controle de armas          | 8      | 2         | -5    |
| subsídios agrícolas        | 0      | 0         | 10    |
| imposto sobre combustíveis | 10     | 0         | -2    |

FIGURA 29.1 Os efeitos de políticas sobre os eleitores. Cada entrada descreve o número de milhares de eleitores urbanos, suburbanos ou rurais que poderiam ser conquistados gastando-se \$1.000 em publicidade em apoio a uma política sobre uma questão específica. Entradas negativas denotam votos que seriam perdidos

Por tentativa e erro, é possível apresentar uma estratégia que obterá o número necessário de votos, mas tal estratégia pode não ser a menos dispendiosa. Por exemplo, você poderia dedicar \$20.000 de publicidade para construção de estradas, \$0 para controle de armas, \$4.000 para subsídios agrícolas e \$9.000 para um imposto de combustíveis. Nesse caso, você ganharia  $20(-2) + 0(8) + 4(0) + 9(10) = 50$  mil votos urbanos,  $20(5) + 0(2) + 4(0) + 9(0) = 100$  mil votos suburbanos e  $20(3) + 0(-5) + 4(10) + 9(-2) = 82$  mil votos rurais. Você obteria o número exato de votos desejados nas áreas urbanas e suburbanas e mais que o número de votos suficientes na área rural. (De fato, na área rural, você conseguiria mais votos que o número de eleitores existentes!) Para armazenar esses votos, teria sido pago um total de  $20 + 0 + 4 + 9 = 33$  mil dólares em publicidade.

Naturalmente, você poderia imaginar se a sua estratégia foi a melhor possível. Isto é, teria sido possível alcançar as metas gastando menos em publicidade? Um processo adicional de tentativa e erro pode ajudá-lo a responder a essa pergunta, mas você teria um método sistemático para responder a tais perguntas. Para fazê-lo, formulamos essa questão em termos matemáticos. Introduzimos 4 variáveis:

- $x_1$  é o número de milhares de dólares gastos em publicidade sobre construção de estradas.
- $x_2$  é o número de milhares de dólares gastos em publicidade sobre controle de armas.
- $x_3$  é o número de milhares de dólares gastos em publicidade sobre subsídios agrícolas.
- $x_4$  é o número de milhares de dólares gastos em publicidade sobre um imposto de combustíveis.

Podemos escrever a exigência de ganhar pelo menos 50.000 votos urbanos como

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 . \quad (29.1)$$

De modo semelhante, podemos escrever os requisitos de ganhar pelo menos 100.000 votos suburbanos e 25.000 votos rurais como

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \quad (29.2)$$

e

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 . \quad (29.3)$$

Qualquer configuração das variáveis  $x_1, x_2, x_3, x_4$  para que as desigualdades (29.1) a (29.3) sejam satisfeitas é uma estratégia que obterá um número suficiente de cada tipo de voto. Para manter os custos tão baixos quanto possível, gostaríamos de minimizar a quantia gasta em publicidade. Isto é, gostaríamos de minimizar a expressão

$$x_1 + x_2 + x_3 + x_4 . \quad (29.4)$$

Embora a publicidade negativa seja uma ocorrência comum em campanhas políticas, não há nada como publicidade de custo negativo. Conseqüentemente, exigimos que

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0 \text{ e } x_4 \geq 0 . \quad (29.5) \quad |_{611}$$

Combinando as desigualdades (29.1) a (29.3) e (29.5) com o objetivo de minimizar (29.4), obtemos o que é conhecido como um “programa linear”. Formatamos esse problema como

$$\text{minimizar} \quad x_1 + x_2 + x_3 + x_4 \quad (29.6)$$

sujeito a

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \quad (29.7)$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \quad (29.8)$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \quad (29.9)$$

$$x_1, x_2, x_3, x_4 \geq 0. \quad (29.10)$$

A solução desse programa linear produzirá uma estratégia ótima para o político.

### Programas lineares gerais

No problema geral de programação linear, desejamos otimizar uma função linear de acordo com um conjunto de desigualdades lineares. Dado um conjunto de números reais  $a_1, a_2, \dots, a_n$  e um conjunto de variáveis  $x_1, x_2, \dots, x_n$ , uma *função linear*  $f$  sobre essas variáveis é definida por

$$f(x_1, x_2, \dots, x_n) = a_1x_1 + a_2x_2 + \dots + a_nx_n = \sum_{j=1}^n a_jx_j.$$

Se  $b$  é um número real e  $f$  é uma função linear, então a equação

$$f(x_1, x_2, \dots, x_n) = b$$

é uma *igualdade linear* e as desigualdades

$$f(x_1, x_2, \dots, x_n) \leq b$$

e

$$f(x_1, x_2, \dots, x_n) \geq b$$

são *desigualdades lineares*. Usamos a expressão *restrições lineares* para denotar igualdades lineares ou desigualdades lineares. Em programação linear, não permitimos desigualdades estritas. Formalmente, um *problema de programação linear* é o problema de minimizar ou maximizar uma função linear de acordo com um conjunto finito de restrições lineares. Se formos minimizar, então chamamos o programa linear um *programa linear de minimização* e, se formos maximizar, então chamamos o programa linear um *programa linear de maximização*.

O restante deste capítulo cobrirá a formulação e a solução de programas lineares. Embora existam vários algoritmos de tempo polinomial para programação linear, não os estudaremos neste capítulo. Em vez disso, estudaremos o algoritmo simplex, que é o algoritmo de programação linear mais antigo. O algoritmo simplex não é executado em tempo polinomial no pior caso, mas é bastante eficiente e extensamente utilizado na prática.

## Uma visão geral de programação linear

Para descrever propriedades e algoritmos relacionados a programas lineares, é conveniente ter formas canônicas para expressá-los. Utilizaremos duas formas, **padrão** e **relaxada**, neste capítulo. Elas serão definidas com exatidão na Seção 29.1. Informalmente, um programa linear em forma padrão é a maximização de uma função linear sujeita a *desigualdades* lineares, enquanto um programa linear em forma relaxada é a maximização de uma função linear sujeita a *igualdades* lineares. Em geral, utilizaremos a forma padrão para expressar programas lineares, mas é mais conveniente usar a forma relaxada quando descrevemos os detalhes do algoritmo simplex. No momento, restringimos nossa atenção a maximizar uma função linear em  $n$  variáveis sujeita a um conjunto de  $m$  desigualdades lineares.

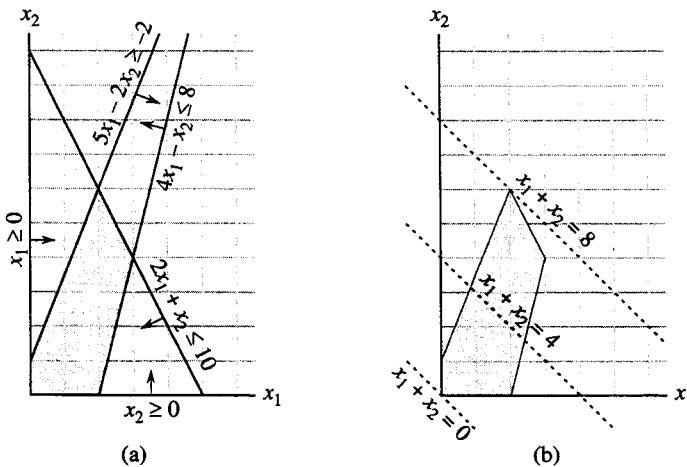


FIGURA 29.2 (a) O programa linear dado em (29.12) a (29.15). Cada restrição é representada por uma linha e uma direção. A interseção das restrições, que é a região possível, está sombreada. (b) As linhas pontilhadas mostram, respectivamente, os pontos para os quais o valor de objetivo é 0, 4 e 8. A solução ótima para o programa linear é  $x_1 = 2$  e  $x_2 = 6$  com valor de objetivo 8

Vamos considerar primeiro o seguinte programa linear com duas variáveis:

$$\text{maximizar } x_1 + x_2 \quad (29.11)$$

sujeito a

$$4x_1 - x_2 \leq 8 \quad (29.12)$$

$$2x_1 + x_2 \leq 10 \quad (29.13)$$

$$5x_1 - 2x_2 \geq -2 \quad (29.14)$$

$$x_1, x_2 \geq 0 \quad (29.15)$$

Chamamos qualquer configuração das variáveis  $x_1$  e  $x_2$  que satisfaz a todas as restrições (29.12) a (29.15) uma **solução possível** para o programa linear. Se representarmos em um grafo as restrições no sistema de coordenadas cartesianas  $(x_1, x_2)$ , como na Figura 29.2(a), veremos que o conjunto de soluções possíveis (sombreado na figura) forma uma região conve-

<sup>1</sup> Uma definição intuitiva de região convexa é que ela satisfaz ao requisito de que, para dois pontos quaisquer na região, todos os pontos em um segmento de linha entre eles também estão na região.

xa<sup>1</sup> no espaço bidimensional. Chamamos essa região convexa **região possível**. A função que desejamos maximizar é chamada **função de objetivo**. Conceitualmente, poderíamos avaliar a função de objetivo  $x_1 + x_2$  em cada ponto da região possível; chamamos o valor da função de objetivo em um determinado ponto **valor de objetivo**. Poderíamos então identificar um ponto que tem o valor de objetivo máximo como uma solução ótima. Para esse exemplo (e para a maioria dos programas lineares), a região possível contém um número infinito de pontos, e assim desejamos determinar um modo eficiente de encontrar um ponto que alcance o valor de objetivo máximo sem avaliar explicitamente a função de objetivo em cada ponto na região possível.

Em duas dimensões, podemos otimizar por meio de um procedimento gráfico. O conjunto de pontos para os quais  $x_1 + x_2 = z$ , para qualquer  $z$ , é uma linha com a inclinação  $-1$ . Se plotarmos  $x_1 + x_2 = 0$ , obteremos a linha com inclinação  $-1$  pela origem, como na Figura 29.2(b). A interseção dessa linha com a região possível é o conjunto de soluções possíveis que têm um valor de objetivo  $0$ . Nesse caso, essa interseção da linha com a região possível é o ponto  $(0, 0)$ . De modo mais geral, para qualquer  $z$ , a interseção da linha  $x_1 + x_2 = z$  com a região possível é o conjunto de soluções possíveis que têm valor de objetivo  $z$ . A Figura 29.2(b) mostra as linhas  $x_1 + x_2 = 0$ ,  $x_1 + x_2 = 4$  e  $x_1 + x_2 = 8$ . Como a região possível da Figura 29.2 é limitada tem de haver algum valor máximo  $z$  para o qual a interseção da linha  $x_1 + x_2 = z$  com a região possível é não vazia. Qualquer ponto em que isso ocorre é uma solução ótima para o programa linear que, nesse caso, é o ponto  $x_1 = 2$  e  $x_2 = 6$  com valor de objetivo  $8$ . Não é acidental o fato de uma solução ótima para o programa linear ter ocorrido em um vértice da região possível. O valor máximo de  $z$  para o qual a linha  $x_1 + x_2 = z$  intercepta a região possível deve estar no limite da região possível, e assim a interseção dessa linha com o limite da região possível é um vértice ou um segmento de linha. Se a interseção é um vértice, então existe apenas uma solução ótima, e ela é um vértice. Se a interseção é um segmento de linha, todo ponto nesse segmento de linha deve ter o mesmo valor de objetivo; em particular, ambas as extremidades do segmento de linha são soluções ótimas. Tendo em vista que cada extremidade de um segmento de linha é um vértice, também existe uma solução ótima em um vértice nesse caso.

Embora não possamos representar facilmente em grafos programas lineares com mais de duas variáveis, a mesma intuição é válida. Se temos três variáveis, então cada restrição é descrita por um semi-espacô no espaço tridimensional. A interseção desses semi-espacos forma a região possível. O conjunto de pontos para os quais a função de objetivo obtém um valor  $z$  é agora um plano. Se todos os coeficientes da função de objetivo são não negativos e se a origem é uma solução possível para o programa linear, então, à medida que afastamos esse plano da origem, encontramos pontos de valor de objetivo crescente. (Se a origem não é possível ou se alguns coeficientes na função de objetivo são negativos, o quadro intuitivo se torna um pouco mais complicado.) Como em duas dimensões, pelo fato de a região possível ser convexa, o conjunto de pontos que alcançam o valor de objetivo ótimo deve incluir um vértice da região possível. De modo semelhante, se temos  $n$  variáveis, cada restrição define um semi-espacô no espaço  $n$  dimensional. A região possível formada pela interseção desses semi-espacos é chamada **simplex**. A função de objetivo é agora um hiperplano e, devido à convexidade, uma solução ótima ainda ocorrerá em um vértice do simplex.

O **algoritmo simplex** toma como entrada um programa linear e retorna uma solução ótima. Ele começa em algum vértice do simplex e executa uma seqüência de iterações. Em cada iteração, ele se move ao longo de uma aresta do simplex a partir de um vértice atual até um vértice vizinho cujo valor de objetivo não é menor que o do vértice atual (e normalmente é maior). O algoritmo simplex termina quando alcança um máximo local, que é um vértice a partir do qual todos os vértices vizinhos têm um valor de objetivo menor. Como a região possível é convexa e a função de objetivo é linear, esse valor ótimo local é realmente um valor ótimo global. Na Seção 29.4, usaremos um conceito denominado “dualidade” para mostrar que a solução retornada pelo algoritmo simplex é de fato ótima.

Embora a visão geométrica ofereça uma boa visão intuitiva das operações do algoritmo simplex, não faremos referência explícita a ela quando desenvolvemos os detalhes do algoritmo simplex na Seção 29.3. Em vez disso, adotaremos uma visão algébrica. Primeiro, escrevemos o programa linear dado em forma relaxada, que é um conjunto de igualdades lineares. Essas igualdades lineares expressarão algumas das variáveis, chamadas “variáveis básicas”, em termos de outras variáveis, chamadas “variáveis não básicas”. A passagem de um vértice para outro será realizada fazendo-se uma variável básica se tornar não básica e fazendo-se uma variável não básica se tornar básica. Essa operação é chamada “pivô” e, vista algebraicamente, não é nada além de reescrever o programa linear em uma forma relaxada equivalente.

O exemplo de duas variáveis descrito antes foi particularmente simples. Precisaremos examinar muitos outros detalhes neste capítulo. Essas questões incluem a identificação de programas lineares que não têm nenhuma solução, programas lineares que não têm nenhuma solução ótima finita e programas lineares para os quais a origem não é uma solução possível.

## Aplicações de programação linear

A programação linear tem um grande número de aplicações. Qualquer livro-texto sobre pesquisa operacional está repleto de exemplos de programação linear, e ela é agora uma ferramenta padrão ensinada aos alunos na maioria das escolas. O cenário eleitoral é um exemplo típico. Dois outros exemplos de programação linear são:

- Uma empresa aérea deseja programar as tripulações de seus vôos. A Administração de Aviação Federal impõe muitas restrições, como limitar o número de horas consecutivas que cada membro da tripulação pode trabalhar e insistir que uma determinada tripulação trabalhe apenas em um modelo de aeronave durante cada mês. A empresa aérea quer programar tripulações em todos os seus vôos usando o menor número possível de tripulantes.
- Uma empresa petrolífera quer decidir onde realizar perfurações em busca de petróleo. A instalação de uma perfuração em um determinado local tem um custo associado e, com base em pesquisas geológicas, uma compensação esperada de um certo número de barris de petróleo. A empresa tem um orçamento limitado para localizar novos poços e quer maximizar a quantidade de petróleo que espera encontrar, dado esse orçamento.

Os programas lineares também são úteis para modelar e resolver problemas de grafos e combinatórios, como os que aparecem neste livro. Já vimos um caso especial de programação linear usado para resolver sistemas de restrições de diferenças na Seção 24.4. Na Seção 29.2 estudaremos como formular diversos problemas de grafos e fluxo em rede como programas lineares. Na Seção 35.4, utilizaremos a programação linear como uma ferramenta para encontrar uma solução aproximada para outro problema de grafo.

## Algoritmos para programação linear

Este capítulo estuda o algoritmo simplex. Esse algoritmo, quando implementado cuidadosamente, com freqüência resolve programas lineares gerais com bastante rapidez na prática. Porem, com algumas entradas criadas com cuidado, o algoritmo simplex pode exigir tempo exponencial. O primeiro algoritmo de tempo polinomial para programação linear foi o *algoritmo de elipsóide*, que é executado lentamente na prática.

Uma segunda classe de algoritmos de tempo polinomial é conhecida como *métodos de ponto interior*. Em contraste com o algoritmo simplex, que se move ao longo do exterior da região possível e mantém uma solução possível que é um vértice do simplex em cada iteração, esses algoritmos se movem pelo interior da região possível. As soluções intermediárias, embora possíveis, não são necessariamente vértices do simplex, mas a solução final é um vértice. O primeiro algoritmo desse tipo foi descoberto por Karmarkar. Para entradas grandes, o desempe-

nho dos algoritmos de ponto interior pode competir com o algoritmo simples, e às vezes ser mais rápido que o deste último.

Se acrescentarmos a um programa linear o requisito adicional de que todas as variáveis têm valores inteiros, teremos um **programa linear inteiro**. O Exercício 34.5-3 lhe pede para mostrar que apenas encontrar uma solução possível para esse problema é NP-difícil; como não há nenhum algoritmo de tempo polinomial conhecido para qualquer problema NP-difícil, não existe nenhum algoritmo de tempo polinomial conhecido para programação linear inteira. Em contraste, um problema geral de programação linear pode ser resolvido em tempo polinomial.

Neste capítulo, se tivermos um programa linear com variáveis  $x = (x_1, x_2, \dots, x_n)$  e desejarmos nos referir a uma configuração específica das variáveis, usaremos a notação  $\bar{x} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$ .

## 29.1 Formas padrão e relaxada

Esta seção descreve dois formatos, a forma padrão e a forma relaxada, que serão úteis na especificação e na utilização de programas lineares. Na forma padrão, todas as restrições são desigualdades, enquanto na forma relaxada, as restrições são igualdades.

### Forma padrão

Na **forma padrão**, recebemos  $n$  números reais  $c_1, c_2, \dots, c_n$ ,  $m$  números reais  $b_1, b_2, \dots, b_m$  e  $mn$  números reais  $a_{ij}$  para  $i = 1, 2, \dots, m$  e  $j = 1, 2, \dots, n$ . Desejamos encontrar  $n$  números reais  $x_1, x_2, \dots, x_n$  que

$$\text{maximizem } \sum_{j=1}^n c_j x_j \quad (29.16)$$

$$\text{sujeito a } \sum_{j=1}^n a_{ij} x_j \leq b_i \text{ para } i = 1, 2, \dots, m \quad (29.17)$$

$$x_j \geq 0 \text{ para } j = 1, 2, \dots, n. \quad (29.18)$$

Generalizando a terminologia que introduzimos para o programa linear de duas variáveis, chamamos a expressão (29.16) de **função de objetivo** e as  $n + m$  desigualdades nas linhas (29.17) e (29.18) de **restrições**. As  $n$  restrições na linha (29.18) são chamadas **restrições de não negatividade**. Um programa linear arbitrário não precisa ter restrições de não negatividade, mas a forma padrão as exige. Às vezes, consideramos conveniente expressar um programa linear em uma forma mais compacta. Se criarmos uma matriz  $m \times n A = (a_{ij})$ , um vetor  $m$ -dimensional  $b = (b_i)$ , um vetor  $n$ -dimensional  $c = (c_j)$  e um vetor  $n$ -dimensional  $x = (x_j)$ , poderemos reescrever o programa linear definido em (29.16) a (29.18) como

$$\text{maximizar } c^T x \quad (29.19)$$

sujeito a

$$Ax \leq b \quad (29.20)$$

$$x \geq 0. \quad (29.21)$$

Na linha (29.19),  $c^T x$  é o produto interno de dois vetores. Na linha (29.20),  $Ax$  é um produto de vetores de matrizes e, na linha (29.21),  $x \geq 0$  significa que cada entrada do vetor  $x$  deve ser não negativa. Observamos que é possível especificar um programa linear em forma padrão por uma tupla  $(A, b, c)$  e adotaremos a convenção de que  $A$ ,  $b$  e  $c$  sempre têm as dimensões dadas anteriormente.

Agora introduzimos a terminologia para descrever soluções de programas lineares. Uma parte dessa terminologia foi usada no exemplo anterior de um programa linear de duas variáveis.

Chamamos uma configuração das variáveis  $\bar{x}$  que satisfazem a todas as restrições uma **solução**

**possível**, enquanto uma configuração das variáveis  $\bar{x}$  que deixa de satisfazer a pelo menos uma restrição é chamada uma **solução impossível**. Dizemos que uma solução  $\bar{x}$  tem **valor de objetivo**  $c^T \bar{x}$ . Uma solução possível  $\bar{x}$  cujo valor de objetivo é máximo sobre todas as soluções possíveis é uma **solução ótima**, e chamamos seu valor de objetivo  $c^T \bar{x}$  de **valor de objetivo ótimo**. Se um programa linear não tem soluções possíveis, dizemos que o programa linear é **impossível**; caso contrário, ele é **possível**. Se um programa linear tem algumas soluções possíveis, mas não tem um valor de objetivo ótimo finito, dizemos que o programa linear é **ilimitado**. O Exercício 29.1-9 lhe pede para mostrar que um programa linear pode ter um valor de objetivo ótimo finito mesmo se a região possível não é limitada.

## Conversão de programas lineares para a forma padrão

Sempre é possível converter um programa linear, dado como a minimização ou maximização de uma função linear sujeita a restrições lineares, para a forma padrão. Um programa linear pode não estar em forma padrão por uma entre quatro razões possíveis:

1. A função de objetivo pode ser uma minimização em vez de uma maximização.
2. Pode haver variáveis sem restrições de não negatividade.
3. Pode haver **restrições de igualdade**, que têm um sinal de igualdade em vez de um sinal de menor que ou igual a.
4. Pode haver **restrições de desigualdade** mas, em vez de terem um sinal de menor que ou igual a, elas têm um sinal de maior que ou igual a.

Ao converter um programa linear  $L$  em outro programa linear  $L'$ , seria interessante a propriedade de uma solução ótima para  $L'$  produzir uma solução ótima para  $L$ . Para captar essa idéia, dizemos que dois programas lineares de maximização  $L$  e  $L'$  são **equivalentes** se, para cada solução possível  $\bar{x}$  para  $L$  com valor de objetivo  $z$ , existe uma solução possível correspondente  $\bar{x}'$  para  $L'$  com valor de objetivo  $z$  e, para cada solução possível  $\bar{x}'$  para  $L'$  com valor de objetivo  $z$ , existe uma solução possível correspondente  $\bar{x}$  para  $L$  com valor de objetivo  $z$ . (Essa definição não implica uma correspondência de um para um entre soluções possíveis.) Um programa linear de minimização  $L$  e um programa linear de maximização  $L'$  são equivalentes se, para cada solução possível  $\bar{x}'$  para  $L$  com valor objetivo  $z$ , existe uma solução possível correspondente  $\bar{x}'$  para  $L'$  com valor de objetivo  $-z$  e, para cada solução possível para  $L'$  com valor de objetivo  $z$ , existe uma solução possível correspondente  $\bar{x}$  para  $L$  com valor de objetivo  $-z$ .

Mostramos agora como remover, um a um, cada um dos problemas possíveis na lista anterior. Depois de remover cada um deles, demonstraremos que o novo programa linear é equivalente ao antigo.

Para converter um programa linear de minimização  $L$  em um programa linear de maximização  $L'$  equivalente, basta negar os coeficientes na função de objetivo. Tendo em vista que  $L$  e  $L'$  têm conjuntos idênticos de soluções possíveis e, para qualquer solução possível, o valor de objetivo em  $L$  é o negativo do valor de objetivo em  $L'$ , esses dois programas lineares são equivalentes. Por exemplo, se temos o programa linear

$$\text{minimizar } -2x_1 + 3x_2$$

sujeito a

$$x_1 + x_2 = 7$$

$$x_1 - 2x_2 \leq 4$$

$$x_1 \geq 0$$

e negarmos os coeficientes da função de objetivo, obtemos

$$\text{maximizar } 2x_1 - 3x_2$$

sujeito a

$$x_1 + x_2 = 7$$

$$x_1 - 2x_2 \leq 4$$

$$x_1 \geq 0.$$

Em seguida, mostramos como converter um programa linear em que algumas das variáveis não têm restrições de não negatividade em um programa em que cada variável tem uma restrição de não negatividade. Suponha que alguma variável  $x_j$  não tem uma restrição de não negatividade. Então, substituímos cada ocorrência de  $x_j$  por  $x'_j - x''_j$  e adicionamos as restrições de não negatividade  $x'_j \geq 0$  e  $x''_j \geq 0$ . Desse modo, se a função de objetivo tem um termo  $c_j x_j$ , ela é substituída por  $c_j x'_j - c_j x''_j$  e, se a restrição  $i$  tem um termo  $a_{ij} x_j$ , ela é substituída por  $a_{ij} x'_j - a_{ij} x''_j$ . Qualquer solução possível  $\bar{x}$  para o novo programa linear corresponde a uma solução possível para o programa linear original com  $\bar{x}_j = x'_j - x''_j$  e com o mesmo valor de objetivo, e portanto os dois programas lineares são equivalentes. Aplicamos esse esquema de conversão a cada variável que não tem uma restrição de não negatividade para produzir um programa linear equivalente em que todas as variáveis têm restrições de não negatividade.

Continuando com o exemplo, queremos assegurar que cada variável tem uma restrição de não negatividade correspondente. A variável  $x_1$  tem tal restrição, mas a variável  $x_2$  não tem. Então, substituímos  $x_2$  por duas variáveis  $x'_2$  e  $x''_2$ , e modificamos o programa linear para obter

$$\text{maximizar } 2x_1 - 3x'_2 + 3x''_2$$

sujeito a

$$x_1 + x'_2 - x''_2 = 7 \quad (29.22)$$

$$x_1 - 2x'_2 + 2x''_2 \leq 4$$

$$x_1, x'_2, x''_2 \geq 0.$$

Em seguida, convertemos as restrições de igualdade em restrições de desigualdade. Suponha que um programa linear tem uma restrição de igualdade  $f(x_1, x_2, \dots, x_n) = b$ . Tendo em vista que  $x = y$  se e somente se  $x \geq y$  e  $x \leq y$ , podemos substituir essa restrição de igualdade pelo par de restrições de desigualdade  $f(x_1, x_2, \dots, x_n) \leq b$  e  $f(x_1, x_2, \dots, x_n) \geq b$ . Repetindo essa conversão para cada restrição de igualdade, temos um programa linear em que todas as restrições são desigualdades.

Finalmente, podemos converter as restrições de maior que ou igual a em restrições de menor que ou igual a, multiplicando essas restrições por  $-1$ . Isto é, qualquer desigualdade da forma

$$\sum_{j=1}^n a_{ij} x_j \geq b_i$$

é equivalente a

$$\sum_{j=1}^n -a_{ij} x_j \leq -b_i$$

Desse modo, substituindo cada coeficiente  $a_{ij}$  por  $-a_{ij}$  e cada valor  $b_i$  por  $-b_i$ , obtemos uma restrição de menor que ou igual a equivalente.

Concluindo nosso exemplo, substituímos a igualdade na restrição (29.22) por duas desigualdades, obtendo

$$\text{maximizar} \quad 2x_1 - 3x'_2 + 3x''_2$$

sujeito a

$$x_1 + x'_2 - x''_2 \leq 7$$

$$x_1 + x'_2 - x''_2 \geq 7$$

$$x_1 - 2x'_2 + 2x''_2 \leq 4$$

$$x_1, x'_2, x''_2 \geq 0.$$

Finalmente, negamos a restrição (29.23). Por consistência nos nomes de variáveis, renomeamos  $x'_2$  como  $x_2$  e  $x''_2$  como  $x_3$ , obtendo a forma padrão

$$\text{maximizar} \quad 2x_1 - 3x_2 + 3x_3$$

(29.24)

sujeito a

$$x_1 + x_2 - x_3 \leq 7$$

(29.25)

$$-x_1 - x_2 + x_3 \leq -7$$

(29.26)

$$x_1 - 2x_2 + 2x_3 \leq 4$$

(29.27)

$$x_1, x_2, x_3 \geq 0.$$

(29.28)

### Conversão de programas lineares para a forma relaxada

Para resolver de maneira eficiente um programa linear com o algoritmo simplex, preferimos expressá-lo em uma forma na qual algumas das restrições são restrições de igualdade. Mais precisamente, vamos convertê-lo para uma forma em que as restrições de não negatividade sejam as únicas restrições de desigualdade, e as restrições restantes sejam igualdades. Seja

$$\sum_{j=1}^n a_{ij}x_j \leq b_i \quad (29.29)$$

uma restrição de desigualdade. Introduzimos uma nova variável  $s$  e reescrevemos a desigualdade (29.29) como as duas restrições

$$s = b_i - \sum_{j=1}^n a_{ij}x_j, \quad (29.30)$$

$$s \geq 0. \quad (29.31)$$

Chamamos  $s$  uma **variável relaxada** porque ela mede a **folga**, ou diferença, entre os lados esquerdo e direito da equação (29.29). Como a desigualdade (29.29) é verdadeira se e somente se a equação (29.30) e a desigualdade (29.31) são ambas verdadeiras, podemos aplicar essa conversão a cada restrição de desigualdade de um programa linear, obtendo um programa linear equivalente em que as únicas restrições de desigualdade são as restrições de não negatividade. Quando convertermos da forma padrão para a forma relaxada, usaremos  $x_{n+i}$  (em vez de  $s$ ) para denotar a variável relaxada associada com a  $i$ -ésima desigualdade. A  $i$ -ésima restrição é então

$$x_{n+i} = b_i - \sum_{j=1}^n a_{ij}x_j, \quad (29.32)$$

juntamente com a restrição de não negatividade  $x_{n+i} \geq 0$ .

Aplicando essa conversão a cada restrição de um programa linear em forma padrão, obtemos um programa linear em uma forma diferente. Por exemplo, para o programa linear descrito em (29.24) a (29.28), introduzimos variáveis relaxadas  $x_4$ ,  $x_5$  e  $x_6$ , obtendo

$$\text{maximizar} \quad 2x_1 - 3x_2 + 3x_3 \quad (29.33)$$

sujeito a

$$x_4 = 7 - x_1 - x_2 + x_3 \quad (29.34)$$

$$x_5 = -7 + x_1 + x_2 - x_3 \quad (29.35)$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \quad (29.36)$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0. \quad (29.37)$$

Nesse programa linear, todas as restrições com exceção das restrições de não negatividade são igualdades, e cada variável está sujeita a uma restrição de não negatividade. Escrevemos cada restrição de igualdade com uma das variáveis no lado esquerdo da igualdade, e todas as outras no lado direito. Além disso, cada equação tem o mesmo conjunto de variáveis no lado direito, e essas variáveis também são as únicas que aparecem na função de objetivo. As variáveis no lado esquerdo das igualdades são chamadas *variáveis básicas*, e as do lado direito são chamadas *variáveis não básicas*.

Para programas lineares que satisfazem a essas condições, às vezes omitiremos as palavras “maximizar” e “sujeito a”, e omitiremos também as restrições explícitas de não negatividade. Utilizaremos ainda a variável  $z$  para denotar o valor da função de objetivo. Chamamos o formato resultante *formas relaxada*. Se escrevermos o programa linear dado em (29.33) a (29.37) em forma relaxada, obtemos

$$z = 2x_1 - 3x_2 + 3x_3 \quad (29.38)$$

$$x_4 = 7 - x_1 - x_2 + x_3 \quad (29.39)$$

$$x_5 = -7 + x_1 + x_2 - x_3 \quad (29.40)$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \quad (29.41)$$

Como ocorre com a forma padrão, será conveniente ter uma notação mais concisa para descrever uma forma relaxada. Como veremos na Seção 29.3, os conjuntos de variáveis básicas e não básicas mudarão à medida que o algoritmo simplex for executado. Usamos  $N$  para denotar o conjunto de índices das variáveis não básicas e  $B$  para denotar o conjunto de índices das variáveis básicas. Sempre teremos  $|N| = n$ ,  $|B| = m$  e  $N \cup B = \{1, 2, \dots, n+m\}$ . As equações serão indexadas pelas entradas de  $B$  e as variáveis no lado direito serão indexadas pelas entradas de  $N$ . Como na forma padrão, usamos  $b_i$ ,  $c_j$  e  $a_{ij}$  para denotar termos constantes e coeficientes. Também usamos  $v$  para denotar um termo constante opcional na função de objetivo. Portanto, podemos definir de modo conciso uma forma relaxada por uma tupla  $(N, B, A, b, c, v)$ , denotando a forma relaxada

$$z = v + \sum_{j \in N} c_j x_j \quad (29.42)$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{para } i \in B, \quad (29.43)$$

em que todas as variáveis  $x$  são restritas a serem não negativas. Como subtraímos a soma  $\sum_{j \in N} a_{ij}x_j$  em (29.43), os valores  $a_{ij}$  são realmente os negativos dos coeficientes que “aparecem” na forma relaxada.

Por exemplo, na forma relaxada

$$\begin{aligned} z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\ x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\ x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3}, \\ x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}, \end{aligned}$$

temos  $B = \{1, 2, 4\}$ ,  $N = \{3, 5, 6\}$ ,

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix},$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix}$$

$c = (c_3 \ c_5 \ c_6)^T = (-1/6 \ -1/6 \ -2/3)^T$  e  $v = 28$ . Observe que os índices em  $A$ ,  $b$  e  $c$  não são necessariamente conjuntos de inteiros contíguos; eles dependem dos conjuntos de índices  $B$  e  $N$ . Como um exemplo do fato de as entradas de  $A$  serem os negativos dos coeficientes conforme eles aparecem na forma relaxada, observe que a equação para  $x_1$  inclui o termo  $x_3/6$ , ainda que o coeficiente  $a_{13}$  realmente seja  $-1/6$  em vez de  $+1/6$ .

## Exercícios

### 29.1-1

Se expressarmos o programa linear em (29.24) a (29.28) na notação compacta de (29.19) a (29.21), quais são os valores  $n$ ,  $m$ ,  $A$ ,  $b$  e  $c$ ?

### 29.1-2

Forneça três soluções possíveis para o programa linear em (29.24) a (29.28). Qual é o valor de objetivo de cada uma?

### 29.1-3

Para a forma relaxada em (29.38) a (29.41), quais são os valores de  $N$ ,  $B$ ,  $A$ ,  $b$ ,  $c$  e  $v$ ?

### 29.1-4

Converta o seguinte programa linear para a forma padrão:

minimizar  $2x_1 + 7x_2$

sujeito a

$$\begin{aligned} x_1 &= 7 \\ 3x_1 + x_2 &\geq 24 \\ x_2 &\geq 0 \\ x_3 &\leq 0. \end{aligned}$$

**29.1-5**

Converta o seguinte programa linear para a forma relaxada:

maximizar  $2x_1 - 6x_3$

sujeito a

$$x_1 + x_2 - x_3 \leq 7$$

$$3x_1 - x_2 \geq 8$$

$$-x_1 + 2x_2 + 2x_3 \geq 0$$

$$x_1, x_2, x_3 \geq 0.$$

Quais são as variáveis básicas e não básicas?

**29.1-6**

Mostre que o seguinte programa linear é impossível:

maximizar  $3x_1 - 2x_2$

sujeito a

$$x_1 + x_2 \leq 2$$

$$-2x_1 - 2x_2 \leq -10$$

$$x_1, x_2 \geq 0.$$

**29.1-7**

Mostre que o seguinte programa linear é ilimitado:

maximizar  $x_1 - x_2$

sujeito a

$$-2x_1 + x_2 \leq -1$$

$$-x_1 - 2x_2 \leq -2$$

$$x_1, x_2 \geq 0.$$

**29.1-8**

Suponha que temos um programa linear geral com  $n$  variáveis e  $m$  restrições, e suponha que o convertemos para a forma padrão. Forneça um limite superior sobre o número de variáveis e restrições no programa linear resultante.

**29.1-9**

Dê um exemplo de um programa linear para o qual a região possível não é limitada, mas o valor de objetivo ótimo é finito.

## 29.2 Formulação de problemas como programas lineares

Embora o algoritmo simplex seja focalizado neste capítulo, também é importante ser capaz de reconhecer quando um problema pode ser formulado como um programa linear. Uma vez que um problema é formulado como um programa linear de tamanho polinomial, ele pode ser resolvido em tempo polinomial pelos algoritmos de elipsóide ou de ponto interior. Vários pacotes de software de programação linear podem resolver problemas de modo eficiente; assim, uma vez que o problema tenha sido expresso como um programa linear, ele pode ser resolvido na prática por tal pacote.

Examinaremos diversos exemplos concretos de problemas de programação linear. Começaremos com dois problemas que já estudamos: o problema de caminhos mais curtos de única ori-

gem (veja o Capítulo 24) e o problema de fluxo máximo (veja o Capítulo 26). Em seguida, descreveremos o problema de fluxo de custo mínimo. Existe um algoritmo de tempo polinomial que não se baseia em programação linear para o problema de fluxo de custo mínimo, mas não o examinaremos. Finalmente, descrevemos o problema de fluxo de várias mercadorias, para o qual o único algoritmo de tempo polinomial conhecido se baseia em programação linear.

## Caminhos mais curtos

O problema de caminhos mais curtos de única origem, descrito no Capítulo 24, pode ser formulado como um programa linear. Nesta seção, focalizaremos a formulação do problema de caminhos mais curtos de um único par, deixando a extensão para o problema mais geral de caminhos mais curtos de única origem como o Exercício 29.2-3.

No problema de caminhos mais curtos de um único par, temos um grafo ponderado orientado  $G = (V, E)$ , com função peso  $w : E \rightarrow \mathbb{R}$  mapeando arestas para pesos de valor real, um vértice de origem  $s$  e um vértice de destino  $t$ . Desejamos calcular o valor  $d[t]$ , que é o peso de um caminho mais curto de  $s$  para  $t$ . Para expressar esse problema como um programa linear, precisamos determinar um conjunto de variáveis e restrições que definem quando temos um caminho mais curto desde  $s$  até  $t$ . Felizmente, o algoritmo de Bellman-Ford faz exatamente isso. Quando termina, o algoritmo de Bellman-Ford calculou, para cada vértice  $v$ , um valor  $d[v]$  tal que para cada aresta  $(u, v) \in E$ , temos  $d[v] \leq d[u] + w(u, v)$ . Inicialmente, o vértice de origem recebe um valor  $d[s] = 0$ , que nunca é alterado. Desse modo, obtemos o seguinte programa linear para calcular o peso do caminho mais curto de  $s$  até  $t$ :

$$\text{maximizar} \quad d[t] \quad (29.44)$$

sujeito a

$$d[v] \leq d[u] + w(u, v) \quad \text{para cada aresta } (u, v) \in E, \quad (29.45)$$

$$d[s] = 0. \quad (29.46)$$

Nesse programa linear, existem  $|V|$  variáveis  $d[v]$ , uma para cada vértice  $v \in V$ . Existem  $|E| + 1$  restrições, uma para cada aresta, além da restrição adicional de que o vértice de origem sempre tem o valor 0.

## Fluxo máximo

O problema de fluxo máximo também pode ser expresso como um programa linear. Lembre-se de que temos um grafo orientado  $G = (V, E)$  em que cada aresta  $(u, v) \in E$  tem uma capacidade não negativa  $c(u, v) \geq 0$ , e dois vértices distintos, uma origem  $s$  e um sorvedor  $t$ . Como definimos na Seção 26.1, um fluxo é uma função de valor real  $f : V \times V \rightarrow \mathbb{R}$  que satisfaz a três propriedades: restrições de capacidade, anti-simetria e conservação de fluxo. Um fluxo máximo é um fluxo que satisfaz a essas restrições e maximiza o valor de fluxo, que é o fluxo total de saída da origem. Então, um fluxo satisfaz a restrições lineares, e o valor de um fluxo é uma função linear. Lembrando também que supomos que  $c(u, v) = 0$  se  $(u, v) \notin E$ , podemos expressar o problema de fluxo máximo como um programa linear:

$$\text{maximizar} \quad \sum_{v \in V} f(s, v) \quad (29.47)$$

sujeito a

$$f(u, v) \leq c(u, v) \quad \text{para cada } u, v \in V, \quad (29.48) \quad | 623$$

$$f(u, v) = -f(v, u) \quad \text{para cada } u, v \in V, \quad (29.49)$$

$$\sum_{v \in V} f(s, v) = 0 \quad \text{para cada } u \in V - \{s, t\}. \quad (29.50)$$

Esse programa linear tem  $|V|^2$  variáveis, correspondendo ao fluxo entre cada par de vértices, e tem  $2|V|^2 + |V| - 2$  restrições.

Normalmente é mais eficiente resolver um problema linear de tamanho menor. O problema linear de (29.47) a (29.50) tem, por facilidade de notação, um fluxo e uma capacidade 0 para cada par de vértices  $u, v$  com  $(u, v) \notin E$ . Seria mais eficiente reescrever o programa linear de forma que ele tivesse  $O(V + E)$  restrições. O Exercício (29.2-5) lhe pede para fazê-lo.

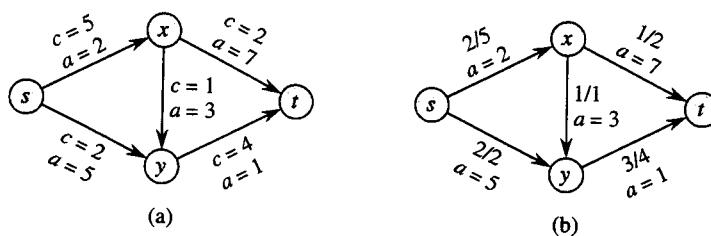


FIGURA 29.3 (a) Um exemplo de problema de fluxo de custo mínimo. Denotamos as capacidades por  $c$  e os custos por  $a$ . O vértice  $s$  é a origem e vértice  $t$  é o sorvedor, e desejamos enviar 4 unidades de fluxo de  $s$  para  $t$ . (b) Uma solução para o problema de fluxo de custo mínimo em que 4 unidades de fluxo são enviadas de  $s$  para  $t$ . Para cada aresta, o fluxo e a capacidade são escritos como fluxo/capacidade

## Fluxo de custo mínimo

Nesta seção, usamos a programação linear para resolver problemas para os quais já conhecemos algoritmos eficientes. De fato, um algoritmo eficiente projetado especificamente para um problema, como o algoritmo de Dijkstra para o problema de caminhos mais curtos de única origem, ou o método de push-relabel para fluxo máximo, com frequência será mais eficiente que a programação linear, tanto em teoria quanto na prática.

A potência real da programação linear vem da capacidade de resolver novos problemas. Lembre-se do problema enfrentado pelo político no início deste capítulo. O problema de obter um número suficiente de votos e, ao mesmo tempo, não gastar muito dinheiro, não é resolvido por qualquer dos algoritmos que estudamos neste livro, ainda que seja resolvido por programação linear. Há muitos livros que descrevem tais problemas reais que a programação linear pode resolver. A programação linear também é particularmente útil para resolver variantes de problemas para os quais talvez ainda não seja conhecido um algoritmo eficiente.

Por exemplo, considere a generalização a seguir do problema de fluxo máximo. Suponha que cada aresta  $(u, v)$  tem, além de uma capacidade  $c(u, v)$ , um custo de valor real  $a(u, v)$ . Se enviarmos  $f(u, v)$  unidades de fluxo pela aresta  $(u, v)$ , incorreremos em um custo  $a(u, v)f(u, v)$ . Também temos um destino de fluxo  $d$ . Desejamos enviar  $d$  unidades de fluxo de  $s$  para  $t$  de tal modo que o custo total gerado pelo fluxo,  $\sum_{(u,v) \in E} a(u, v)f(u, v)$ , seja minimizado. Esse problema é conhecido como **problema de fluxo de custo mínimo**.

A Figura 29.3(a) mostra um exemplo do problema de fluxo de custo mínimo. Desejamos enviar 4 unidades de fluxo de  $s$  para  $t$ , incorrendo no custo total mínimo. Qualquer fluxo válido específico, isto é, uma função  $f$  que satisfaz às restrições (29.48) a (29.50), incorre em um custo total  $\sum_{(u,v) \in E} a(u, v)f(u, v)$ . Desejamos encontrar o fluxo específico de 4 unidades que minimiza esse custo. Uma solução ótima é dada na Figura 29.3(b), e ela tem o custo total  $\sum_{(u,v) \in E} a(u, v)f(u, v) = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$ .

Existem algoritmos de tempo polinomial projetados especificamente para o problema de fluxo de custo mínimo, mas eles estão além do escopo deste livro. Entretanto, podemos expressar o problema de fluxo de custo mínimo como um programa linear. O programa linear é semelhante ao do problema de fluxo máximo, com a restrição adicional de que o valor do fluxo deve ser exatamente  $d$  unidades, e com a nova função de objetivo de minimizar o custo:

$$\text{minimizar} \quad \sum_{(u,v) \in E} a(u,v) f(u,v) \quad (29.51)$$

sujeito a

$$f(u, v) \leq c(u, v) \quad \text{para cada } u, v \in V, \quad (29.52)$$

$$f(u, v) = -f(v, u) \quad \text{para cada } u, v \in V, \quad (29.53)$$

$$\sum_{v \in E} f(u, v) = 0 \quad \text{para cada } u \in V - \{s, t\}, \quad (29.54)$$

$$\sum_{v \in E} f(s, v) = d. \quad (29.55)$$

## Fluxo de várias mercadorias

Como um exemplo final, consideramos outro problema de fluxo. Suponha que a empresa Lucky Puck da Seção 26.1 decida diversificar sua linha de produtos e não comercializar apenas discos de hóquei, mas também bastões e capacetes de hóquei. Cada item de equipamento é manufaturado em sua própria fábrica, tem seu próprio armazém e deve ser transportado todo dia, desde a fábrica até o armazém. Os bastões são fabricados em Vancouver e devem ser transportados para Saskatoon, e os capacetes são fabricados em Edmonton e devem ser transportados para Regina. No entanto, a capacidade da rede de transporte não muda e os diferentes itens – ou **mercadorias** – devem compartilhar a mesma rede.

Esse exemplo é uma instância de um **problema de fluxo de várias mercadorias**. Nesse problema, temos novamente um grafo orientado  $G = (V, E)$  em que cada aresta  $(u, v) \in E$  tem uma capacidade não negativa  $c(u, v) \geq 0$ . Como no problema de fluxo máximo, supomos implicitamente que  $c(u, v) = 0$  para  $(u, v) \notin E$ . Além disso, recebemos  $k$  artigos diferentes,  $K_1, K_2, \dots, K_k$ , onde o artigo  $i$  é especificado pela tripla  $K_i = (s_i, t_i, d_i)$ . Aqui,  $s_i$  é a origem do artigo  $i$ ,  $t_i$  é o sorvedor do artigo  $i$  e  $d_i$  é a demanda, o valor de fluxo desejado para a mercadoria  $i$  desde  $s_i$  até  $t_i$ . Definimos um fluxo para a mercadoria  $i$ , denotada por  $f_i$  (de forma que  $f_i(u, v)$  é o fluxo da mercadoria  $i$  desde o vértice  $u$  até o vértice  $v$ ) como uma função de valor real que satisfaz às restrições de conservação de fluxo, anti-simetria e capacidade. Agora definimos  $f(u, v)$ , o **fluxo agregado**, como a soma dos vários fluxos de mercadorias, de modo que [ver símbolo]  $f_i(u, v)$ . O fluxo agregado sobre a aresta  $(u, v)$  não deve ser maior que a capacidade da aresta  $(u, v)$ . Essa restrição agrupa as restrições de capacidade para as mercadorias individuais. Do modo como esse problema é descrito, não há nada para minimizar; só precisamos determinar se é possível encontrar tal fluxo. Desse modo, escrevemos um programa linear com uma função de objetivo “nula”:

$$\text{minimizar} \quad 0$$

sujeito a

$$\begin{aligned} \sum_{i=1}^k f_i(u, v) &\leq c(v, u) && \text{para cada } u, v \in V, \\ f_i(u, v) &\leq -f_i(v, u) && \text{para cada } i = 1, 2, \dots, k \text{ e} \end{aligned}$$

$$\text{para cada } u, v \in V,$$

$$\begin{aligned} \sum_{v \in V} f_i(u, v) &= 0 && \text{para cada } i = 1, 2, \dots, k \text{ e} \\ &&& \text{para cada } u \in V - \{s_i, t_i\}, \end{aligned}$$

$$\begin{aligned} \sum_{v \in V} f_i(s_i, v) &= d_i && \text{para cada } i = 1, 2, \dots, k. \end{aligned}$$

O único algoritmo de tempo polinomial conhecido para esse problema é expressá-lo como um programa linear, e depois resolvê-lo com um algoritmo de programação linear de tempo polinomial.

## Exercícios

### 29.2-1

Coloque o programa linear de caminhos mais curtos de um único par de (29.44) a (29.46) em forma padrão.

### 29.2-2

Escreva explicitamente o programa linear correspondente a encontrar o caminho mais curto desde o nó  $s$  até o nó  $y$  na Figura 24.2(a).

### 29.2-3

No problema de caminhos mais curtos de única origem, queremos encontrar os pesos de caminhos mais curtos desde um vértice de origem  $s$  até todos os vértices  $v \in V$ . Dado um grafo  $G$ , escreva um programa linear para o qual a solução tenha a propriedade de que  $d[v]$  é o peso do caminho mais curto desde  $s$  até  $v$  para cada vértice  $v \in V$ .

### 29.2-4

Escreva explicitamente o programa linear correspondente a encontrar o fluxo máximo na Figura 26.1(a).

### 29.2-5

Reescreva o programa linear para fluxo máximo (29.47) a (29.50) de modo que ele use apenas  $O(V + E)$  restrições.

### 29.2-6

Escreva um programa linear que, dado um grafo bipartido  $G = (V, E)$ , resolva o problema de emparelhamento bipartido máximo.

### 29.2-7

No **problema de fluxo de várias mercadorias de custo mínimo**, temos um grafo orientado  $G = (V, E)$  no qual cada aresta  $(u, v) \in E$  tem uma capacidade não negativa  $c(u, v) \geq 0$  e um custo  $a(u, v)$ . Como no problema de fluxo de várias mercadorias, temos  $k$  artigos diferentes,  $K_1, K_2, \dots, K_k$ , onde a mercadoria  $i$  é especificada pela tripla  $K_i = (s_i, t_i, d_i)$ . Definimos o fluxo  $f_i$  para a mercadoria  $i$  e o fluxo agregado  $f(u, v)$  sobre a aresta  $(u, v)$  como no problema de fluxo de várias mercadorias. Um fluxo possível é aquele em que o fluxo agregado em cada aresta  $(u, v)$  não é maior que a capacidade da aresta  $(u, v)$ . O custo de um fluxo é  $\sum_{u, v \in V} a(u, v) f(u, v)$ , e a meta é encontrar o fluxo possível de custo mínimo. Expresse esse problema como um programa linear.

## 29.3 O algoritmo simplex

O algoritmo simplex é o método clássico para resolver programas lineares. Em contraste com a maioria dos outros algoritmos neste livro, seu tempo de execução não é polinomial no pior caso. Porém, ele gera idéias para programas lineares e com freqüência é bastante rápido na prática.

Além de ter uma interpretação geométrica, descrita antes neste capítulo, o algoritmo simplex guarda alguma semelhança com a eliminação gaussiana, discutida na Seção 28.3. A eliminação gaussiana começa com um sistema de igualdades lineares cuja solução é desconhecida. Em cada iteração, reescrivemos esse sistema em uma forma equivalente que tem alguma estrutura adicional. Após um certo número de iterações, reescrivemos o sistema de modo que a solução seja simples de obter. O algoritmo simplex prossegue de maneira semelhante, e podemos vê-lo como uma eliminação gaussiana para desigualdades.

Agora, descrevemos a idéia principal por trás de uma iteração do algoritmo simplex. Associada a cada iteração, haverá “uma solução básica” facilmente obtida a partir da forma relaxada do programa linear: definir cada variável não básica como 0 e calcular os valores das variáveis básicas a partir das restrições de igualdade. Uma solução básica sempre corresponderá a um vértice do simplex. Algebricamente, uma iteração converte uma forma relaxada em uma forma relaxada equivalente. O valor de objetivo da solução básica possível associada não será menor que o da iteração anterior (e, em geral, será maior). Para alcançar esse aumento no valor de objetivo, escolemos uma variável não básica tal que, se fôssemos aumentar o valor dessa variável desde 0, então o valor de objetivo também aumentaria. A quantidade pela qual aumentamos a variável é limitada pelas outras restrições. Em particular, nós a aumentamos até alguma variável básica se tornar 0. Em seguida, reescrevemos a forma relaxada, trocando as funções dessa variável básica e da variável não básica escolhida. Apesar de termos usado uma configuração específica das variáveis para orientar o algoritmo e de empregarmos essa configuração em nossas provas, o algoritmo não mantém de forma explícita essa solução. Ele simplesmente reescreve o programa linear até a solução ótima se tornar “óbvia”.

## Um exemplo do algoritmo simplex

Começamos com um exemplo estendido. Considere o seguinte programa linear em forma padrão:

$$\text{maximizar} \quad 3x_1 + x_2 + 2x_3 \quad (29.56)$$

sujeito a

$$x_1 + x_2 + 3x_3 \leq 30 \quad (29.57)$$

$$2x_1 + 2x_2 + 5x_3 \leq 24 \quad (29.58)$$

$$4x_1 + x_2 + 2x_3 \leq 36 \quad (29.59)$$

$$x_1, x_2, x_3 \geq 0. \quad (29.60)$$

Para usar o algoritmo simplex, devemos converter o programa linear em forma relaxada; vimos como fazê-lo na Seção 29.1. Além de ser uma manipulação algébrica, o algoritmo relaxado é um conceito útil. Na Seção 29.1, observamos que cada variável tem uma restrição de não negatividade correspondente; dizemos que uma restrição de igualdade é **restrita** para uma configuração particular de suas variáveis não básicas se elas fazem a variável básica da restrição se tornar 0. De modo semelhante, uma configuração de variáveis não básicas que faria uma variável básica se tornar negativa **viola** aquela restrição. Desse modo, as variáveis relaxadas mantêm explicitamente a informação sobre a distância a que cada restrição está de ser restrita, e então ajudam a definir o quanto podemos aumentar os valores de variáveis não básicas sem violar quaisquer restrições.

Associando as variáveis relaxadas  $x_4$ ,  $x_5$  e  $x_6$  respectivamente às desigualdades (29.57) a (29.59) e pondo o programa linear em forma relaxada, obtemos

$$z = 3x_1 + x_2 + 2x_3 \quad (29.61)$$

$$x_4 = 30 - x_1 - x_2 - 3x_3 \quad (29.62)$$

$$x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \quad (29.63)$$

$$x_6 = 36 - 4x_1 - x_2 - 2x_3 \quad (29.64)$$

O sistema de restrições (29.62) a (29.64) tem 3 equações e 6 variáveis. Qualquer configuração das variáveis  $x_1$ ,  $x_2$  e  $x_3$  define valores para  $x_4$ ,  $x_5$  e  $x_6$ ; existe então um número infinito de soluções para esse sistema de equações. Uma solução é possível se todos os valores  $x_1, x_2, \dots, x_6$  são não negativos, e também pode haver um número infinito de soluções possíveis. O número infi-

nito de soluções possíveis para um sistema como esse será útil em provas posteriores. Vamos nos concentrar na **solução básica**: definiremos todas as variáveis (não básicas) no lado direito como 0, e então calcularemos os valores das variáveis (básicas) no lado esquerdo. Nesse exemplo, a solução básica é  $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$ , que tem valor de objetivo  $z = (3 \cdot 0) + (1 \cdot 0) + (2 \cdot 0) = 0$ . Observe que essa solução básica define  $\bar{x}_i = b_i$  para cada  $i \in B$ . Uma iteração do algoritmo simplex reescreverá o conjunto de equações e a função de objetivo para pôr um conjunto diferente de variáveis no lado direito. Desse modo, existirá uma solução básica diferente associada ao problema reescrito. Enfatizamos que a ação de reescrever não altera de nenhuma forma o problema de programação linear subjacente; o problema em uma iteração tem um conjunto de soluções possíveis idêntico ao do problema na iteração anterior. Contudo, o problema tem realmente uma solução básica diferente da solução na iteração anterior.

Se uma solução básica também é possível, nós a denominamos **solução básica possível**. Durante a execução do algoritmo simplex, a solução básica quase sempre será uma solução básica possível. Porém, veremos na Seção 29.5 que, para as primeiras iterações do algoritmo simplex, a solução básica pode não ser possível.

Nossa meta, em cada iteração, é reformular o programa linear de forma que a solução básica tenha um valor de objetivo maior. Selecionamos uma variável não básica  $x_e$  cujo coeficiente na função de objetivo é positivo e acrescentamos o valor de  $x_e$  tanto quanto possível sem violar quaisquer das restrições. A variável  $x_e$  se torna básica, e alguma outra variável  $x_i$  se torna não básica. Os valores de outras variáveis básicas e da função de objetivo também podem mudar.

Para continuar o exemplo, vamos pensar em aumentar o valor de  $x_1$ . À medida que aumentamos  $x_1$ , os valores de  $x_4$ ,  $x_5$  e  $x_6$  diminuem. Como temos uma restrição de não negatividade para cada variável, não podemos permitir que qualquer delas se torne negativa. Se  $x_1$  aumenta acima de 30, então  $x_4$  se torna negativa, enquanto  $x_5$  e  $x_6$  se tornam negativas quando  $x_1$  aumenta acima de 12 e 9, respectivamente. A terceira restrição (29.64) é a restrição mais rígida, e ela limita o quanto podemos aumentar  $x_1$ . Então, trocaremos as funções de  $x_1$  e  $x_6$ . Resolvemos a equação (29.64) para  $x_1$  e obtemos

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}. \quad (29.65)$$

Para reescrever as outras equações com  $x_6$  no lado direito, substituímos  $x_1$  usando a equação (29.65). Fazendo isso para a equação (29.62), obtemos

$$\begin{aligned} x_4 &= 30 - x_1 - x_2 - x_3 \\ &= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 \\ &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4}. \end{aligned} \quad (29.66)$$

De modo semelhante, podemos combinar a equação (29.65) com a restrição (29.63) e com a função de objetivo (29.61) para reescrever nosso programa linear da seguinte forma:

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \quad (29.67)$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \quad (29.68)$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \quad (29.69)$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}. \quad (29.70)$$

Chamamos essa operação uma **criação de pivô**. Como demonstramos antes, a criação de pivô escolhe uma variável não básica  $x_e$ , chamada **variável de entrada** e uma variável básica  $x_b$ , chamada **variável de saída** e troca suas funções.

O programa linear descrito em (29.67) a (29.70) é equivalente ao programa linear descrito nas equações (29.61) a (29.64). As operações que executamos no algoritmo simplex são reescrever equações de modo que as variáveis se desloquem entre o lado esquerdo e o lado direito e substituir uma equação em outra. A primeira operação cria de modo trivial um problema equivalente e a segunda, por álgebra linear elementar, também cria um problema equivalente.

Para demonstrar essa equivalência, observe que nossa solução básica original  $(0, 0, 0, 30, 24, 36)$  satisfaz às novas equações (29.68) a (29.70) e tem valor de objetivo  $27 + (1/4) \cdot 0 + (1/2) \cdot 0 - (3/4) \cdot 36 = 0$ . A solução básica associada ao novo programa linear define os valores não básicos como 0, e é  $(9, 0, 0, 21, 6, 0)$ , com valor de objetivo  $z = 27$ . A aritmética simples verifica que essa solução também satisfaz às equações (29.62) a (29.64) e, quando conectada à função de objetivo (29.61), tem valor de objetivo  $(3 \cdot 9) + (1 \cdot 0) + (2 \cdot 0) = 27$ .

Continuando com o exemplo, desejamos encontrar uma nova variável cujo valor queremos aumentar. Não queremos aumentar  $x_6$  pois, à medida que seu valor aumenta, valor de objetivo diminui. Podemos tentar aumentar  $x_2$  ou  $x_3$ ; escolheremos  $x_3$ . Até que ponto podemos aumentar  $x_3$  sem violar qualquer das restrições? A restrição (29.68) a limita a 18, a restrição (29.69) a limita a  $42/5$ , e a restrição (29.70) a limita a  $3/2$ . A terceira restrição é de novo a mais rígida, e então reescreveremos a terceira restrição de forma que  $x_3$  fique no lado esquerdo e  $x_5$  no lado direito. Em seguida, substituímos essa nova equação nas equações (29.67) a (29.69) e obtemos o sistema novo, mas equivalente

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \quad (29.71)$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \quad (29.72)$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \quad (29.73)$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}. \quad (29.74)$$

Esse sistema tem a solução básica associada  $(33/4, 0, 3/2, 69/4, 0, 0)$ , com valor de objetivo  $111/4$ . Agora, a única maneira de aumentar o valor de objetivo é aumentar  $x_2$ . As três restrições fornecem limites superiores iguais a 132, 4 e  $\infty$ , respectivamente. (O limite superior  $\infty$  da restrição (29.74) ocorre porque, à medida que aumentamos  $x_2$ , o valor da variável básica  $x_4$  também aumenta. Então, essa restrição não limita o quanto  $x_2$  pode ser aumentada.) Aumentamos  $x_2$  até 4, e ela se torna não básica. Depois, resolvemos a equação (29.73) para  $x_2$  e a substituímos nas outras equações para obter

$$z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \quad (29.75)$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \quad (29.76)$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \quad (29.77)$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2}. \quad (29.78)$$

Nesse ponto, todos os coeficientes na função de objetivo são negativos. Como veremos mais adiante neste capítulo, essa situação só ocorre quando reescrevemos o programa linear de forma que a solução básica seja uma solução ótima. Assim, para esse problema, a solução  $(8, 4, 0,$

$(18, 0, 0)$ , com valor de objetivo 28, é ótima. Podemos agora retornar ao nosso programa linear original dado em (29.56) a (29.60). As únicas variáveis no programa linear original são  $x_1$ ,  $x_2$  e  $x_3$ , e assim nossa solução é  $x_1 = 8$ ,  $x_2 = 4$  e  $x_3 = 0$ , com valor de objetivo  $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$ . Observe que os valores das variáveis relaxadas na solução final medem a folga em cada desigualdade. A variável relaxada  $x_4$  é 18 e, na desigualdade (29.57), o lado esquerdo, com valor  $8 + 4 + 0 = 12$ , é 18 unidades menor que o lado direito 30. As variáveis relaxadas  $x_5$  e  $x_6$  são 0 e, de fato, nas desigualdades (29.58) e (29.59), o lado esquerdo e o lado direito são iguais. Observe também que, embora os coeficientes na forma relaxada original sejam inteiros, os coeficientes nos outros programas lineares não são necessariamente inteiros, e as soluções intermediárias não são necessariamente inteiras. Além disso, a solução final para um programa linear não precisa ser inteira; é simples coincidência que esse exemplo tenha uma solução inteira.

## Criação de pivô

Agora formalizamos o procedimento para criação de pivô. O procedimento PIVOT toma como entrada uma forma relaxada, dada pela tupla  $(N, B, A, b, c, v)$ , o índice  $l$  da variável de saída  $x_l$ , e o índice  $e$  da variável de entrada  $x_e$ . Ele retorna a tupla  $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$  que descreve a nova forma relaxada. (Lembre-se mais uma vez de que as entradas das matrizes  $A$  e  $\hat{A}$  são na realidade a negação dos coeficientes que aparecem na forma relaxada.)

**PIVOT( $N, B, A, b, c, v, l, e$ )**

- 1  $\triangleright$  Calcular os coeficientes da equação para a nova variável básica  $x_e$ .
- 2  $\hat{b}_e \leftarrow b_l/a_{le}$
- 3 **for** cada  $j \in N - \{e\}$
- 4   **do**  $\hat{a}_{ej} \leftarrow a_{lj}/a_{le}$
- 5    $\hat{a}_{el} \leftarrow 1/a_{le}$
- 6  $\triangleright$  Calcular os coeficientes das restrições restantes.
- 7 **for** cada  $i \leftarrow B - \{l\}$
- 8   **do**  $\hat{b}_i \leftarrow b_i - a_{ie}\hat{b}_e$
- 9     **for** cada  $j \in N - \{e\}$
- 10       **do**  $\hat{a}_{ij} \leftarrow a_{ij} - a_{ie}\hat{a}_{ej}$
- 11        $\hat{a}_{ij} \leftarrow a_{ij} - a_{ie}\hat{a}_{el}$
- 12  $\triangleright$  Calcular a função de objetivo.
- 13  $\hat{v} \leftarrow v + c_e\hat{b}_e$
- 14 **for** cada  $j \in N - \{e\}$
- 15   **do**  $\hat{c}_j \leftarrow c_j - c_e\hat{a}_{ej}$
- 16    $\hat{c}_l \leftarrow -c_e\hat{a}_{el}$
- 17  $\triangleright$  Calcular novos conjuntos de variáveis básicas e não básicas.
- 18  $\hat{N} = N - \{e\} \cup \{l\}$
- 19  $\hat{B} = B - \{l\} \cup \{e\}$
- 20 **return**  $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$

PIVOT funciona da maneira descrita a seguir. As linhas 2 a 5 calculam os coeficientes na nova equação para  $x_e$ , reescrevendo a equação que tem  $x_l$  no lado esquerdo para, em vez disso, ter  $x_e$  no lado esquerdo. As linhas 7 a 11 atualizam as equações restantes substituindo o lado direito dessa nova equação em cada ocorrência de  $x_e$ . As linhas 13 a 16 fazem a mesma substituição para a função de objetivo, e as linhas 18 e 19 atualizam os conjuntos de variáveis não básicas e básicas. A linha 20 retorna a nova forma relaxada. Como foi dado, se  $a_{le} = 0$ , PIVOT causaria um erro de divisão por 0 mas, como veremos nas provas dos Lemas 29.2 e 29.12, PIVOT é chamado apenas quando  $a_{le} \neq 0$ .

Agora vamos resumir o efeito que PIVOT tem sobre os valores das variáveis na solução

### Lema 29.1

Considere uma chamada  $\text{PIVOT}(N, B, A, b, c, v, I, e)$  na qual  $a_{le} \neq 0$ . Sejam  $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$  os valores retornados pela chamada, e seja  $\bar{x}$  a solução básica depois da chamada. Então

1.  $\bar{x}_j = 0$  para cada  $j \in \hat{N}$ .
2.  $\bar{x}_e = b_l/a_{le}$ .
3.  $\bar{x}_i = b_i - a_{ie}\hat{b}_e$  para cada  $i \in \hat{B} - \{e\}$ .

**Prova** A primeira declaração é verdadeira, porque a solução básica sempre define todas as variáveis não básicas como 0. Quando definimos cada variável não básica como 0 em uma restrição

$$\bar{x}_i = \hat{b}_i - \sum_{j \in N} \hat{a}_{ij} \bar{x}_j ,$$

temos  $\bar{x}_i = \hat{b}_i$  para cada  $i \in \hat{B}$ . Tendo em vista que  $e \in \hat{B}$ , pela linha 2 de PIVOT, temos

$$\bar{x}_e = \hat{b}_e = b_l/a_{le} ,$$

o que prova a segunda declaração. De modo semelhante, usando a linha 8 para cada  $i \in \hat{B} - e$ , temos

$$\bar{x}_i = \hat{b}_i = b_i - a_{ie}\hat{b}_e ,$$

o que prova a terceira declaração. ■

## O algoritmo simplex formal

Agora estamos prontos para formalizar o algoritmo simplex, que demonstramos por meio de um exemplo. Esse exemplo foi particularmente interessante, e poderíamos ter encontrado várias outras questões:

- Como determinar se um programa linear é possível?
- O que fazer se o programa linear for possível, mas a solução básica inicial não for possível?
- Como determinar se um programa linear é ilimitado?
- Como escolher as variáveis de entrada e de saída?

Na Seção 29.5 mostraremos como determinar se um problema é possível e, se for o caso, como encontrar uma forma relaxada na qual a solução básica inicial seja possível. Então, supomos que temos um procedimento  $\text{INITIALIZE-SIMPLEX}(A, b, c)$  que toma como entrada um programa linear em forma padrão, ou seja, uma matriz  $m \times n$   $A = (a_{ij})$ , um vetor  $m$ -dimensional  $b = (b_i)$  e um vetor  $n$ -dimensional  $c = (c_j)$ . Se o problema for impossível, ele retorna uma mensagem de que o programa é impossível e em seguida termina. Caso contrário, ele retorna uma forma relaxada para a qual a solução básica inicial é possível.

O procedimento SIMPLEX toma como entrada um programa linear em forma padrão, da maneira descrita. Ele retorna um vetor de  $n$  elementos  $\bar{x} = (\bar{x}_j)$  que é uma solução ótima para o programa linear descrito em (29.19) a (29.21).

$\text{SIMPLEX}(A, b, c)$

- 1  $(N, B, A, b, c, v) \leftarrow \text{INITIALIZE-SIMPLEX}(A, b, c)$
- 2 **while** algum índice  $j \in N$  tem  $c_j > 0$
- 3     **do** escolher um índice  $e \in N$  para o qual  $c_e > 0$
- 4         **for** cada índice  $i \in B$

```

5      do if  $a_{ie} > 0$ 
6          then  $\Delta_i \leftarrow b_i/a_{ie}$ 
7          else  $\Delta_i \leftarrow \infty$ 
8      escolher um índice  $l \in B$  que minimize  $\Delta_l$ 
9      if  $\Delta_l = \infty$ 
10         then return "ilimitado"
11     else  $(N, B, A, b, c, v) \leftarrow \text{PIVOT}(N, B, A, b, c, v, l, e)$ 
12   for  $i \leftarrow 1$  to  $n$ 
13     do if  $i \in B$ 
14       then  $\bar{x}_i \leftarrow b_i$ 
15       else  $\bar{x}_i \leftarrow 0$ 
16   return  $(\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$ 

```

O procedimento SIMPLEX funciona como descrevemos a seguir. Na linha 1, ele chama o procedimento INITIALIZE-SIMPLEX( $A, b, c$ ), descrito anteriormente, que determina que o programa linear é impossível ou retorna uma forma relaxada, para a qual a solução básica seja possível. A parte principal do algoritmo é dada no loop **while** das linhas 2 a 11. Se todos os coeficientes na função de objetivo são negativos, então o loop **while** termina. Caso contrário, na linha 3, selecionamos uma variável  $x_e$  cujo coeficiente na função de objetivo seja positivo como a variável de entrada. Apesar de termos liberdade para escolher qualquer variável como a variável de entrada, supomos o uso de alguma regra determinística previamente especificada. Em seguida, nas linhas 4 a 8, verificamos cada restrição e escolhemos a que limita mais severamente a quantidade pela qual podemos aumentar  $x_e$  sem violar quaisquer das restrições de não negatividade; a variável básica associada a essa restrição é  $x_l$ . Novamente, podemos ter a liberdade de escolher uma dentre diversas variáveis como a variável de saída, mas supomos o uso de alguma regra determinística previamente especificada. Se nenhuma das restrições limitar a quantidade pela qual a variável de entrada pode aumentar, o algoritmo retorna "ilimitado" na linha 10. Caso contrário, a linha 11 troca as funções das variáveis de entrada e saída, chamando a sub-rotina PIVOT( $N, B, A, b, c, v, l, e$ ), descrita anteriormente. As linhas 12 a 15 calculam uma solução para as variáveis de programação linear originais  $\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n$ , definindo todas as variáveis não básicas como 0 e cada variável básica  $i$  como  $\bar{x}_i$  para  $b_i$ . No Teorema 29.10, veremos que essa solução é uma solução ótima para o programa linear. Por fim a linha 16 retorna os valores calculados dessas variáveis de programação linear originais.

Para mostrar que SIMPLEX é correto, primeiro mostramos que, se SIMPLEX tem uma solução inicial possível e eventualmente termina, então ele retorna uma solução possível ou determina que o programa linear é ilimitado. Então, mostramos que SIMPLEX termina. Finalmente, na Seção 29.4, mostramos que a solução retornada é ótima.

### Lema 29.2

Dado um programa linear  $(A, b, c)$ , suponha que a chamada a INITIALIZE-SIMPLEX na linha 1 de SIMPLEX retorne uma forma relaxada para a qual a solução básica é possível. Então, se SIMPLEX retorna uma solução na linha 16, essa solução é uma solução possível para o programa linear. Se SIMPLEX retorna "ilimitado" na linha 10, o programa linear é ilimitado.

**Prova** Usamos o seguinte loop invariante de três partes:

No início de cada iteração do loop **while** das linhas 2 a 11,

1. A forma relaxada é equivalente à forma relaxada retornada pela chamada de INITIALIZE-SIMPLEX.
2. Para cada  $i \in B$ , temos  $b_i \geq 0$ .
3. A solução básica associada à forma relaxada é possível.

**Inicialização:** A equivalência das formas relaxadas é trivial para a primeira iteração. Supomos, no enunciado do lema, que a chamada a INITIALIZE-SIMPLEX na linha 1 de SIMPLEX retorna uma forma relaxada para a qual a solução básica é possível. Desse modo, a terceira parte do invariante é verdadeira. Além disso, tendo em vista que cada variável básica  $x_i$  é definida como  $b_i$  na solução básica, e como a viabilidade da solução básica implica que cada variável básica  $x_i$  é não negativa, temos que  $b_i \geq 0$ . Assim, a segunda parte do invariante é válida.

**Manutenção:** Mostraremos que o loop invariante é mantido, supondo que a instrução **return** na linha 10 não é executada. Trataremos o caso em que a linha 10 é executada quando discutirmos o término.

Uma iteração do loop **while** permuta a função de uma variável básica e uma variável não básica. As únicas operações executadas envolvem a resolução de equações e a substituição de uma equação em outra, e então a forma relaxada é equivalente à da iteração anterior que, pelo loop invariante, é equivalente à forma inicial relaxada.

Agora demonstramos a segunda parte do loop invariante. Supomos que, no início de cada iteração do loop **while**,  $b_i \geq 0$  para cada  $i \in B$ , e mostraremos que essas desigualdades permanecem verdadeiras depois da chamada a PIVOT na linha 11. Tendo em vista que as únicas mudanças para as variáveis  $b_i$  e o conjunto  $B$  de variáveis básicas ocorrem nessa atribuição, basta mostrar que a linha 11 mantém essa parte do invariante. Fazemos  $b_i$ ,  $a_{ij}$  e  $B$  se referirem a valores antes da chamada de PIVOT, e  $\hat{b}_i$  se refere a valores retornados de PIVOT.

Primeiro, observamos que  $\hat{b}_e \geq 0$ , porque  $b_l \geq 0$  pelo loop invariante,  $a_{le} > 0$  pela linha 5 de SIMPLEX, e  $\hat{b}_e = b_l/a_{le}$  pela linha 2 de PIVOT.

Para os índices restantes  $i \in B - l$ , temos que

$$\begin{aligned}\hat{b}_i &= b_i - a_{ie}\hat{b}_e && \text{(pela linha 8 de PIVOT)} \\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(pela linha 2 de PIVOT)} .\end{aligned}\tag{29.79}$$

Temos dois casos a considerar, dependendo do fato de  $a_{ie} > 0$  ou  $a_{ie} \leq 0$ . Se  $a_{ie} > 0$  então, como escolhemos  $l$  tal que

$$b_l/a_{le} \leq b_i/a_{ie} \quad \text{para todo } i \in B ,\tag{29.80}$$

temos

$$\begin{aligned}\hat{b}_i &= b_i - a_{ie}(b_l/a_{le}) && \text{(pela equação (29.79))} \\ &\geq b_i - a_{ie}(b_i/a_{ie}) && \text{(pela desigualdade (29.80))} \\ &= b_i - b_i \\ &= 0 ,\end{aligned}$$

e, desse modo,  $\hat{b}_i \geq 0$ . Se  $a_{ie} \leq 0$ , então como  $a_{le}$ ,  $b_i$  e  $b_l$  são todos não negativos, a equação (29.79) implica que  $\hat{b}_i$  também deve ser não negativo.

Agora demonstramos que a solução básica é possível, isto é, que todas as variáveis têm valores não negativos. As variáveis não básicas são definidas como 0 e portanto são não negativas. Cada variável básica  $x_i$  é definida pela equação

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j .$$

A solução básica define  $\bar{x}_i = b_i$ . Usando a segunda parte do loop invariante, concluímos que cada variável básica  $\bar{x}_i$  é não negativa.

**Término:** O loop **while** pode terminar de dois modos. Se ele terminar devido à condição na linha 2, então a solução básica atual é possível, e essa solução é retornada na linha 16. O outro modo de terminar é retornar “ilimitado” na linha 10. Nesse caso, para cada iteração do loop **for** nas linhas 4 a 7, quando a linha 5 é executada, descobrimos que  $a_{ie} \leq 0$ . Seja  $\bar{x}$  a solução básica associada à forma relaxada no início da iteração que retornou “ilimitado”. Considere a solução  $\bar{x}$  definida como

$$\bar{x}_i = \begin{cases} \infty & \text{se } i = e, \\ 0 & \text{se } i \in N - \{e\}, \\ b_i - \sum_{j \in N} a_{ij} \bar{x}_j & \text{se } i \in B. \end{cases}$$

Agora mostramos que essa solução é possível, isto é, que todas as variáveis são não negativas. As variáveis não básicas diferentes de  $\bar{x}_e$  são iguais a 0, e  $\bar{x}_e$  é positiva; desse modo, todas as variáveis não básicas são não negativas. Para cada variável básica  $\bar{x}_i$ , temos

$$\begin{aligned} \bar{x}_i &= b_i - \sum_{j \in N} a_{ij} \bar{x}_j \\ &= b_i - a_{ie} \bar{x}_e. \end{aligned}$$

O loop invariante implica que  $b_i \geq 0$ , e temos  $a_{ie} \leq 0$  e  $\bar{x}_e = \infty > 0$ . Portanto,  $\bar{x}_i \geq 0$ .

Mostramos agora que o valor de objetivo para a solução  $\bar{x}$  é ilimitado. O valor de objetivo é

$$\begin{aligned} z &= v + \sum_{j \in N} c_j \bar{x}_j \\ &= v + c_e \bar{x}_e. \end{aligned}$$

Tendo em vista que  $c_e > 0$  (pela linha 3) e  $\bar{x}_e = \infty$ , o valor de objetivo é  $\infty$ , e assim o programa linear é ilimitado. ■

Em cada iteração, SIMPLEX mantém  $A$ ,  $b$ ,  $c$  e  $v$ , além dos conjuntos  $N$  e  $B$ . Embora manter explicitamente  $A$ ,  $b$ ,  $c$  e  $v$  seja essencial para a implementação eficiente do algoritmo simplex, isso não é estritamente necessário. Em outras palavras, a forma relaxada é determinada de modo exclusivo pelos conjuntos de variáveis básicas e não básicas. Antes de provar esse fato, vamos provar um lema algébrico útil.

### Lema 29.3

Seja  $I$  um conjunto de índices. Para cada  $i \in I$ , sejam  $\alpha_i$  e  $\beta_i$  números reais e seja  $x_i$  uma variável de valor real. Seja  $\gamma$  qualquer número real. Suponha que, para quaisquer configurações de  $x_i$ , temos

$$\sum_{i \in I} \alpha_i x_i = \gamma + \sum_{i \in I} \beta_i x_i. \quad (29.81)$$

Então,  $\alpha_i = \beta_i$  para cada  $i \in I$ , e  $\gamma = 0$ .

**Prova** Tendo em vista que a equação (29.81) é válida para quaisquer valores de  $x_i$ , podemos usar valores específicos para tirar conclusões sobre  $\alpha$ ,  $\beta$  e  $\gamma$ . Se fizermos  $x_i = 0$  para cada  $i \in I$ , concluímos que  $\gamma = 0$ . Agora, escolha um índice arbitrário  $i \in I$ , e sejam  $x_i = 1$  e  $x_k = 0$  para todo  $k \neq i$ . Então, devemos ter  $\alpha_i = \beta_i$ . Tendo em vista que escolhemos  $i$  como qualquer índice em  $I$ , concluímos que  $\alpha_i = \beta_i$  para cada  $i \in I$ . ■

Mostramos agora que a forma relaxada de um programa linear é determinada exclusivamente pelo conjunto de variáveis básicas.

### Lema 29.4

Seja  $(A, b, c)$  um programa linear em forma padrão. Dado um conjunto  $B$  de variáveis básicas, a forma relaxada associada é determinada de modo exclusivo.

**Prova** Suponha para fins de contradição que existem duas formas relaxadas diferentes com o mesmo conjunto  $B$  de variáveis básicas. As formas relaxadas também devem ter conjuntos idênticos  $N = \{1, 2, \dots, n + m\} - B$  de variáveis não básicas. Escrevemos a primeira forma relaxada como

$$z = v + \sum_{j \in N} c_j x_j \quad (29.82)$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{para } i \in B, \quad (29.83)$$

E a segunda como

$$z = v' + \sum_{j \in N} c'_j x_j \quad (29.84)$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{para } i \in B. \quad (29.85)$$

Considere o sistema de equações formado pela subtração de cada equação na linha (29.85) da equação correspondente na linha (29.83). O sistema resultante é

$$0 = (b_i - b'_i) - \sum_{j \in N} (a_{ij} - a'_{ij}) x_j \quad \text{para } i \in B$$

ou, de modo equivalente,

$$\sum_{j \in N} a_{ij} x_j = (b_i - b'_i) - \sum_{j \in N} a'_{ij} x_j \quad \text{para } i \in B.$$

Agora, para cada  $i \in B$ , aplicamos o Lema 29.3 com  $\alpha_i = a_{ij}$ ,  $\beta_i = a'_{ij}$  e  $\gamma = b_i - b'_i$ . Tendo em vista que  $\alpha_i = \beta_i$ , temos que  $a_{ij} = a'_{ij}$  para cada  $j \in N$  e, como  $\gamma = 0$ , temos  $b_i = b'_i$ . Portanto, para as duas formas relaxadas,  $A$  e  $b$  são idênticos a  $A'$  e  $b'$ . Usando um argumento semelhante, o Exercício 29.3-1 mostra que também deve ocorrer  $c = c'$  e  $v = v'$ , e consequentemente, as formas relaxadas devem ser idênticas. ■

Resta mostrar que SIMPLEX termina e que, quando ele termina, a solução retornada é ótima. A Seção 29.4 tratará do caráter ótimo. Agora, vamos discutir o término.

### Término

No exemplo dado no início desta seção, cada iteração do algoritmo simplex aumentou o valor de objetivo associado à solução básica. Como o Exercício 29.3-2 lhe pede para mostrar, nenhuma iteração de SIMPLEX pode diminuir o valor de objetivo associado à solução básica. Infelizmente, é possível que uma iteração deixe o valor de objetivo inalterado. Esse fenômeno é chamado **degenerescência** e agora o estudaremos em detalhes.

O valor de objetivo é alterado pela atribuição  $\hat{v} \leftarrow v + c_e \hat{b}_e$  na linha 13 de PIVOT. Tendo em vista que SIMPLEX chama PIVOT somente quando  $c_e > 0$ , o único modo para o valor de objetivo | 635

permanecer inalterado (isto é,  $\hat{v} = v$ ) é  $\hat{b}_e$  ser 0. Esse valor é atribuído como  $\hat{b}_e \leftarrow b_l/a_{le}$  na linha 2 de PIVOT. Como sempre chamamos PIVOT com  $a_{le} \neq 0$ , vemos que, para  $\hat{b}_e$  ser igual a 0 e consequentemente o valor de objetivo ser inalterado, devemos ter  $b_l = 0$ .

Na realidade, essa situação pode ocorrer. Considere o programa linear

$$\begin{aligned} z &= x_1 + x_2 + x_3 \\ x_4 &= 8 - x_1 - x_2 \\ x_5 &= x_2 - x_3. \end{aligned}$$

Vamos supor que escolhemos  $x_1$  como a variável de entrada e  $x_4$  como a variável de saída. Após a ação de pivô, obtemos

$$\begin{aligned} z &= 8 + x_3 + x_4 \\ x_1 &= 8 - x_2 - x_4 \\ x_5 &= x_2 - x_3. \end{aligned}$$

Nesse ponto, nossa única escolha é fazer o pivô com  $x_3$  entrando e  $x_5$  saindo. Tendo em vista que  $b_5 = 0$ , o valor de objetivo de 8 permanece inalterado após a criação de pivô:

$$\begin{aligned} z &= 8 + x_2 - x_4 - x_5 \\ x_1 &= 8 - x_2 - x_4 \\ x_3 &= x_2 - x_5. \end{aligned}$$

O valor de objetivo não se alterou, mas nossa representação sim. Felizmente, se fizermos de novo o pivô, com  $x_2$  entrando e  $x_1$  saindo, o valor de objetivo aumentará, e o algoritmo simplex poderá continuar.

Agora, mostramos que a degenerescência é o único fato que poderia impedir o algoritmo simplex de terminar. Lembre-se de nossa hipótese de que SIMPLEX escolha índices  $e$  e  $l$ , nas linhas 3 e 8 respectivamente, de acordo com alguma regra determinística. Dizemos que SIMPLEX **circula** se as formas relaxadas em duas iterações diferentes são idênticas; nesse caso, como SIMPLEX é um algoritmo determinístico, ele circulará indefinidamente pela mesma série de formas relaxadas.

### Lema 29.5

Se SIMPLEX deixar de terminar em no máximo  $\binom{n+m}{m}$  iterações, ele circulará.

**Prova** Pelo Lema 29.4, o conjunto  $B$  de variáveis básicas determina de modo exclusivo uma forma relaxada. Existem  $n+m$  variáveis e  $|B| = m$  e assim existem  $\binom{n+m}{m}$  maneiras de escolher  $B$ . Portanto, existem apenas  $\binom{n+m}{m}$  formas relaxadas exclusivas. Então, se SIMPLEX for executado por mais de  $\binom{n+m}{m}$  iterações, ele deverá entrar em um ciclo. ■

A entrada em um ciclo é teoricamente possível, mas muito rara. Ela pode ser evitada escolhendo-se as variáveis de entrada e saída de forma um pouco mais cuidadosa. Uma opção é perturbar ligeiramente a entrada de modo que seja impossível ter duas soluções com o mesmo valor de objetivo. Uma segunda opção é romper as ligações lexicograficamente, e uma terceira é romper as ligações sempre escolhendo a variável com o menor índice. Essa última estratégia é conhecida como regra de Bland. Omitimos a prova de que essas estratégias evitam a entrada em um ciclo.

**Lema 29.6**

Se, nas linhas 3 e 8 de SIMPLEX, as ligações sempre são rompidas escolhendo-se a variável com o menor índice, então SIMPLEX deve terminar. ■

Concluímos esta seção com o lema a seguir.

**Lema 29.7**

Supondo que INITIALIZE-SIMPLEX retorne uma forma relaxada para a qual a solução básica é possível, SIMPLEX informa que um programa linear é ilimitado, ou então termina com uma solução possível em no máximo  $\binom{n+m}{m}$  iterações.

**Prova** Os Lemas 29.2 e 29.6 mostram que, se INITIALIZE-SIMPLEX retorna uma forma relaxada para a qual a solução básica é possível, SIMPLEX informa que um programa linear é ilimitado, ou então termina com uma solução possível. Pela contraposição do Lema 29.5, se SIMPLEX termina com uma solução possível, então ele termina em no máximo  $\binom{n+m}{m}$  iterações. ■

**Exercícios****29.3-1**

Complete a prova do Lema 29.4, mostrando que deve ocorrer  $c = c'$  e  $v = v'$ .

**29.3-2**

Mostre que a chamada a PIVOT na linha 11 de SIMPLEX nunca diminuirá o valor de  $v$ .

**29.3-3**

Suponha que convertemos um programa linear  $(A, b, c)$  em forma padrão para a forma relaxada. Mostre que a solução básica é possível se e somente se  $b_i \geq 0$  para  $i = 1, 2, \dots, m$ .

**29.3-4**

Resolva o seguinte programa linear usando SIMPLEX:

$$\text{maximizar} \quad 18x_1 + 12,5x_2$$

sujeito a

$$x_1 + x_2 \leq 20$$

$$x_1 \leq 12$$

$$x_2 \leq 16$$

$$x_1, x_2 \geq 0.$$

**29.3-5**

Resolva o seguinte programa linear usando SIMPLEX:

$$\text{maximizar} \quad -5x_1 - 3x_2$$

sujeito a

$$x_1 - x_2 \leq 1$$

$$2x_1 + x_2 \leq 2$$

$$x_1, x_2 \geq 0.$$

**29.3-6**

Resolva o seguinte programa linear usando SIMPLEX:

$$\begin{aligned}
& \text{minimizar} && x_1 + x_2 + x_3 \\
& \text{sujeito a} && \\
& && 2x_1 + 7,5x_2 + 3x_3 \geq 10000 \\
& && 20x_1 + 5x_2 + 10x_3 \geq 30000 \\
& && x_1, x_2, x_3 \geq 0 .
\end{aligned}$$

## 29.4 Dualidade

Provamos que, sob certas hipóteses, SIMPLEX terminará. Contudo, ainda não mostramos que ele realmente encontra a solução ótima para um programa linear. Para fazê-lo, introduzimos um conceito importante, chamado **dualidade de programação linear**.

A dualidade é uma propriedade muito importante. Em um problema de otimização, a identificação de um problema dual está quase sempre relacionada à descoberta de um algoritmo de tempo polinomial. A dualidade também é eficiente em sua habilidade para fornecer uma prova de que uma solução é de fato ótima.

Por exemplo, suponha que, dada uma instância de um problema de fluxo máximo, encontramos um fluxo  $f$  de valor  $|f|$ . Como saber se  $f$  é um fluxo máximo? Pelo teorema de corte mínimo de fluxo máximo (Teorema 26.7), se pudermos encontrar um corte cujo valor também seja  $|f|$ , então teremos verificado que  $f$  é de fato um fluxo máximo. Esse é um exemplo de dualidade: dado um problema de maximização, definimos um problema de minimização relacionado, tal que os dois problemas tenham os mesmos valores de objetivo ótimos.

Dado um programa linear em que o objetivo é maximizar, descreveremos como formular um programa linear **dual** em que o objetivo é minimizar, e cujo valor ótimo é idêntico ao do programa linear original. Quando nos referimos a programas lineares duais, chamamos o programa linear original de **primitivo**.

Dado um programa linear primitivo em forma padrão, como em (29.16) a (29.18), definimos o programa linear dual como

$$\text{minimizar} \quad \sum_{i=1}^m b_i y_i \quad (29.86)$$

$$\text{sujeito a} \quad \sum_{i=1}^m a_{ij} y_i \geq c_j \quad \text{para } j = 1, 2, n, \dots, \quad (29.87)$$

$$y_i \geq 0 \quad \text{para } i = 1, 2, \dots, m . \quad (29.88)$$

Para formar o dual, alteramos a maximização para uma minimização, trocamos as funções do lado direito e os coeficientes da função de objetivo, e substituímos o sinal de menor que ou igual a por um sinal de maior que ou igual a. Cada uma das  $m$  restrições no primitivo tem uma variável associada  $y_i$  no dual, e cada uma das  $n$  restrições no dual tem uma variável  $x_j$  associada no primitivo. Por exemplo, considere o programa linear dado em (29.56) a (29.60). O dual desse programa linear é

$$\text{minimizar} \quad 30y_1 + 24y_2 + 36y_3 \quad (29.89)$$

$$\text{sujeito a} \quad \begin{aligned} y_1 + 2y_2 + 4y_3 &\geq 3 \\ y_1 + 2y_2 + y_3 &\geq 1 \end{aligned} \quad (29.90) \quad (29.91)$$

$$3y_1 + 5y_2 + 2y_3 \geq 2 \quad (29.92)$$

$$y_1, y_2, y_3 \geq 0 . \quad (29.93)$$

Mostraremos, no Teorema 29.10, que o valor ótimo do programa linear dual é sempre igual ao valor ótimo do programa linear primitivo. Além disso, o algoritmo simplex na realidade resolve implicitamente tanto o programa linear primitivo quanto o dual ao mesmo tempo, fornecendo assim uma prova do caráter ótimo.

Começamos demonstrando a **dualidade fraca**, que declara que qualquer solução possível para o programa linear primitivo tem um valor não maior que o de qualquer solução possível para o programa linear dual.

**Lema 29.8 (Dualidade fraca de programação linear)**

Seja  $\bar{x}$  qualquer solução possível para o programa linear primitivo em (29.16) a (29.18), e seja  $\bar{y}$  qualquer solução possível para o programa linear dual em (29.86) a (29.88). Então

$$\sum_{j=1}^n c_j \bar{x}_j \leq \sum_{i=1}^m b_i \bar{y}_i .$$

**Prova** Temos

$$\begin{aligned} \sum_{j=1}^n c_j \bar{x}_j &\leq \sum_{j=1}^n \left( \sum_{i=1}^m a_{ij} \bar{y}_i \right) \bar{x}_j && \text{(por (29.87))} \\ &= \sum_{i=1}^m \left( \sum_{j=1}^n a_{ij} \bar{x}_j \right) \bar{y}_i \\ &= \sum_{i=1}^m b_i \bar{y}_i && \text{(por (29.17)).} \end{aligned}$$

**Corolário 29.9**

Seja  $\bar{x}$  uma solução possível para um programa linear primitivo  $(A, b, c)$  e seja  $\bar{y}$  uma solução possível para o programa linear dual correspondente. Se

$$\sum_{j=1}^n c_j \bar{x}_j = \sum_{i=1}^m b_i \bar{y}_i$$

Então,  $\bar{x}$  e  $\bar{y}$  são soluções ótimas para os programas lineares primitivo e dual, respectivamente.

**Prova** Pelo Lema 29.8, o valor de objetivo de uma solução possível para o primitivo não pode exceder o de uma solução possível para o dual. O programa linear primitivo é um problema de maximização, e o dual é um problema de minimização. Desse modo, se as soluções possíveis  $\bar{x}$  e  $\bar{y}$  têm o mesmo valor de objetivo, nenhuma delas pode ser melhorada. ■

Antes de provar que sempre existe uma solução dual cujo valor é igual ao de uma solução primitiva ótima, descreveremos como encontrar tal solução. Quando executamos o algoritmo simplex no programa linear em (29.56) a (29.60), a última iteração produziu a forma relaxada (29.75) a (29.78) com  $B = \{1, 2, 4\}$  e  $N = \{3, 5, 6\}$ . Como veremos a seguir, a solução básica associada à forma relaxada final é uma solução ótima para o programa linear; uma solução ótima para o programa linear (29.56) a (29.60) é então  $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (8, 4, 0)$ , com valor de objetivo  $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$ . Como também veremos a seguir, podemos obter uma solução dual ótima: os negativos dos coeficientes da função de objetivo primitiva são os valores das variáveis duais. Mais precisamente, suponha que a última forma relaxada da primitiva seja

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ x_i &= b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{para } i \in B . \end{aligned}$$

Então, uma solução dual ótima é definir

$$\bar{y}_i = \begin{cases} -c'_{n+i} & \text{se } (n+i) \in N, \\ 0 & \text{caso contrário.} \end{cases} \quad (29.94)$$

Desse modo, uma solução ótima para o programa linear dual definido em (29.89) a (29.93) é  $\bar{y}_1 = 0$  (pois  $n+1 = 4 \in B$ ),  $\bar{y}_2 = -c'_5 = 1/6$  e  $\bar{y}_3 = -c'_6 = 2/3$ . Avaliando a função de objetivo dual (29.89), obtemos um valor de objetivo igual a  $(30 \cdot 0) + (24 \cdot (1/6)) + (36 \cdot (2/3)) = 28$ , o que confirma que o valor de objetivo do primitivo realmente é igual ao valor de objetivo do dual. Combinando esses cálculos com o Lema 29.8, temos uma prova de que o valor de objetivo ótimo do programa linear primitivo é 28. Agora, mostramos que, em geral, uma solução ótima para o dual e uma prova do caráter ótimo de uma solução para o primitivo podem ser obtidas dessa maneira.

#### **Teorema 29.10 (Dualidade de programação linear)**

Suponha que SIMPLEX retorne valores  $\bar{x} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$  no programa linear primitivo ( $A, b, c$ ). Sejam  $N$  e  $B$  as variáveis não básicas e básicas para a forma relaxada final, seja  $c'$  a representação dos coeficientes na forma relaxada final, e seja  $\bar{y} = (\bar{y}_1, \bar{y}_2, \dots, \bar{y}_m)$  definido pela equação (29.94). Então,  $\bar{x}$  é uma solução ótima para o programa linear primitivo,  $\bar{y}$  é uma solução ótima para o programa linear dual e

$$\sum_{j=1}^n c_j \bar{x}_j = \sum_{i=1}^m b_i \bar{y}_i. \quad (29.95)$$

**Prova** Pelo Corolário 29.9, se pudermos encontrar soluções possíveis  $\bar{x}$  e  $\bar{y}$  que satisfazem à equação (29.95), então  $\bar{x}$  e  $\bar{y}$  devem ser soluções ótimas primitivas e duais. Agora, mostraremos que as soluções  $\bar{x}$  e  $\bar{y}$  descritas no enunciado do teorema satisfazem à equação (29.95).

Vamos supor que executamos SIMPLEX sobre um programa linear primitivo, como mostram as linhas (29.16) a (29.18). O algoritmo prossegue por uma série de formas relaxadas até terminar com uma forma relaxada final com função de objetivo

$$z = v' + \sum_{j \in N} c'_j x_j. \quad (29.96)$$

Tendo em vista que SIMPLEX terminou com uma solução, pela condição da linha 2, sabemos que

$$c'_j \leq 0 \quad \text{para todo } j \in N. \quad (29.97)$$

Se definirmos

$$c'_j = 0 \quad \text{para todo } j \in B, \quad (29.98)$$

podemos reescrever equação (29.96) como

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ &= v' + \sum_{j \in N} c'_j x_j + \sum_{j \in B} c'_j x_j \quad (\text{porque } c'_j = 0 \text{ se } j \in B) \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j \quad (\text{porque } N \cup B = \{1, 2, \dots, n+m\}). \end{aligned} \quad (29.99)$$

Para a solução básica  $\bar{x}$  associada a essa forma relaxada final,  $\bar{x}_j = 0$  para todo  $j \in N$ , e  $z = v'$ . Tendo em vista que todas as formas relaxadas são equivalentes, se avaliarmos a função de objetivo original em  $\bar{x}$ , devemos obter o mesmo valor de objetivo, isto é,

$$\sum_{j=1}^n c_j \bar{x}_j = v' + \sum_{j=1}^{n+m} c'_j \bar{x}_j \quad (29.100)$$

$$\begin{aligned} &= v' + \sum_{j \in N} c'_j \bar{x}_j + \sum_{j \in B} c'_j \bar{x}_j \\ &= v' + \sum_{j \in N} (c'_j \cdot 0) + \sum_{j \in B} (0 \cdot \bar{x}_j). \end{aligned} \quad (29.101)$$

Mostraremos agora que  $\bar{y}$ , definido pela equação (29.94), é possível para o programa linear dual e que seu valor de objetivo  $\sum_{i=1}^m b_i \bar{y}_i$  é igual a  $\sum_{j=1}^n c_j \bar{x}_j$ . A equação (29.100) afirma que a primeira e a última formas relaxadas, avaliadas em  $\bar{x}$ , são iguais. De modo mais geral, a equivalência de todas as formas relaxadas implica que, para *qualquer* conjunto de valores  $x = (x_1, x_2, \dots, x_n)$ , temos

$$\sum_{j=1}^n c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j .$$

Conseqüentemente, para qualquer conjunto de valores  $x$ , temos

$$\begin{aligned} &\sum_{j=1}^n c_j x_j \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j \\ &= v' + \sum_{j=1}^n c'_j x_j + \sum_{j=n+1}^{n+m} c'_j x_j \\ &= v' + \sum_{j=1}^n c'_j x_j + \sum_{i=1}^m c'_{n+i} x_{n+i} \\ &= v' + \sum_{j=1}^n c'_j x_j + \sum_{i=1}^m (-\bar{y}_i) x_{n+i} \quad (\text{pela equação (29.94)}) \\ &= v' + \sum_{j=1}^n c'_j x_j + \sum_{i=1}^m -\bar{y}_i \left( b_i - \sum_{j=1}^n a_{ij} x_j \right) \quad (\text{pela equação (29.32)}) \\ &= v' + \sum_{j=1}^n c'_j x_j + \sum_{i=1}^m b_i \bar{y}_i + \sum_{i=1}^m \sum_{j=1}^n (a_{ij} x_j) \bar{y}_i \\ &= v' + \sum_{j=1}^n c'_j x_j - \sum_{i=1}^m b_i \bar{y}_i + \sum_{j=1}^n \sum_{i=1}^m (a_{ij} \bar{y}_i) x_j \\ &= \left( v' + \sum_{i=1}^m b_i \bar{y}_i \right) + \sum_{j=1}^n \left( c'_j + \sum_{i=1}^m a_{ij} \bar{y}_i \right) x_j , \end{aligned}$$

de forma que

$$\sum_{j=1}^n c_j x_j = \left( v' + \sum_{i=1}^m b_i \bar{y}_i \right) + \sum_{j=1}^n \left( c'_j + \sum_{i=1}^m a_{ij} \bar{y}_i \right) x_j. \quad (29.102)$$

Aplicando o Lema 29.3 à equação (29.102), obtemos

$$v' + \sum_{i=1}^m b_i \bar{y}_i = 0, \quad (29.103)$$

$$c'_j + \sum_{i=1}^m a_{ij} \bar{y}_i = c_j \quad \text{para } j = 1, 2, \dots, n. \quad (29.104)$$

Pela equação (29.103), temos que  $\sum_{i=1}^m b_i \bar{y}_i = v'$ , e consequentemente o valor de objetivo do dual  $\left( \sum_{i=1}^m b_i \bar{y}_i \right)$  é igual ao do primitivo ( $v'$ ). Resta mostrar que a solução  $\bar{y}$  é possível para o problema dual. De (29.97) e (29.98), temos que  $c'_j \leq 0$  para todo  $j = 1, 2, \dots, n + m$ . Assim, para qualquer  $i = 1, 2, \dots, m$ , (29.104) implica que

$$c_j = c'_j + \sum_{i=1}^m a_{ij} \bar{y}_i,$$

que satisfaz às restrições (29.87) do dual. Finalmente, tendo em vista que  $c'_j \leq 0$  para cada  $j \in N \cup B$ , quando definimos  $\bar{y}$  de conjunto com a equação (29.94), temos que cada  $\bar{y}_i \geq 0$ , e assim as restrições de não negatividade também são satisfeitas. ■

Mostramos que, dado um programa linear possível, se INITIALIZE-SIMPLEX retorna uma solução possível, e se SIMPLEX termina sem retornar “ilimitado”, então a solução retornada é realmente uma solução ótima. Também mostramos como elaborar uma solução ótima para o programa linear dual.

## Exercícios

### 29.4-1

Formule o programa linear dual dado no Exercício 29.3-4.

### 29.4-2

Suponha que temos um programa linear que não está em forma padrão. Poderíamos produzir o dual convertendo-o primeiro para a forma padrão, e depois criando o dual. Porém, seria mais conveniente ser capaz de produzir o dual diretamente. Explique como, dado um programa linear arbitrário, podemos obter diretamente o dual desse programa linear.

### 29.4-3

Escreva o dual do programa linear de fluxo máximo, conforme as linhas (29.47) a (29.50). Explique como interpretar essa formulação como um problema de corte mínimo.

### 29.4-4

Escreva o dual do programa linear de custo mínimo, conforme as linhas (29.51) a (29.55). Explique como interpretar esse problema em termos de grafos e fluxos.

### 29.4-5

Mostre que o dual do dual de um programa linear é o programa linear primitivo.

### 29.4-6

Qual resultado do Capítulo 26 pode ser interpretado como uma dualidade fraca para o problema de fluxo máximo?

## 29.5 A solução básica inicial possível

Nesta seção, primeiro descrevemos como testar se um programa linear é possível e, se for, como produzir uma forma relaxada para a qual a solução básica seja possível. Concluímos com a prova do teorema fundamental de programação linear, que informa que o procedimento SIMPLEX sempre produz o resultado correto.

### Como encontrar uma solução inicial

Na Seção 29.3, supomos que tínhamos um procedimento INITIALIZE-SIMPLEX que determina se um programa linear tem soluções possíveis e, nesse caso, fornece uma forma relaxada para a qual a solução básica é possível. Descrevemos esse procedimento aqui.

Um programa linear pode ser possível, ainda que a solução básica inicial possa não ser possível. Por exemplo, considere o seguinte programa linear:

$$\text{maximizar} \quad 2x_1 - x_2 \quad (29.105)$$

sujeito a

$$2x_1 - x_2 \leq 2 \quad (29.106)$$

$$x_1 - 5x_2 \leq -4 \quad (29.107)$$

$$x_1, x_2 \geq 0. \quad (29.108)$$

Se fôssemos converter esse programa linear para a forma relaxada, a solução básica seria definir  $x_1 = 0$  e  $x_2 = 0$ . Essa solução viola a restrição (29.107) e assim, não é uma solução possível. Desse modo, INITIALIZE-SIMPLEX não pode simplesmente retornar a forma relaxada óbvia. Por inspeção, não está claro nem mesmo se esse programa linear tem quaisquer soluções possíveis. Para determinar se ele as tem, podemos formular um *programa linear auxiliar*. Para esse programa linear auxiliar, seremos capazes de encontrar (com um pouco de trabalho) uma forma relaxada para a qual a solução básica é possível. Além disso, a solução desse programa linear auxiliar determinará se o programa linear inicial é possível e, se for o caso, fornecerá uma solução possível com a qual poderemos inicializar SIMPLEX.

### Lema 29.11

Seja  $L$  um programa linear em forma padrão, dado como em (29.16) a (29.18). Seja  $L_{\text{aux}}$  o programa linear a seguir com  $n + 1$  variáveis:

$$\text{maximizar} \quad -x_0 \quad (29.109)$$

sujeito a

$$\sum_{j=1}^n a_{ij}x_j - x_0 \leq b_i \quad \text{para } i = 1, 2, \dots, m, \quad (29.110)$$

$$x_j \geq 0 \quad \text{para } j = 0, 1, \dots, n. \quad (29.111)$$

Então,  $L$  é possível se e somente se o valor de objetivo ótimo de  $L_{\text{aux}}$  é 0.

**Prova** Suponha que  $L$  tenha uma solução possível  $\bar{x} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$ . Então, a solução  $\bar{x}_0 = 0$  combinada com  $\bar{x}$  é uma solução possível para  $L_{\text{aux}}$  com valor de objetivo 0. Tendo em vista que  $x_0 \geq 0$  é uma restrição de  $L_{\text{aux}}$  e a função de objetivo é maximizar  $-x_0$ , essa solução deve ser ótima para  $L_{\text{aux}}$ .

Reciprocamente, suponha que o valor de objetivo ótimo  $L_{\text{aux}}$  seja 0. Então,  $\bar{x}_0 = 0$ , e os valores das variáveis restantes  $\bar{x}$  satisfazem às restrições de  $L$ . ■

Agora, descrevemos nossa estratégia para encontrar uma solução básica inicial possível para um programa linear  $L$  em forma padrão:

```

INITIALIZE-SIMPLEX( $A, b, c$ )
1 seja  $l$  o índice do mínimo  $b_i$ 
2 if  $b_l \geq 0$             $\triangleright$  A solução básica inicial é possível?
3   then return ( $\{1, 2, \dots, n\}, \{n + 1, n + 2, \dots, n + m\}, A, b, c, 0$ )
4   formar  $L_{aux}$  adicionando  $-x_0$  ao lado esquerdo de cada equação
      e definindo a função de objetivo como  $-x_0$ 
5   seja  $(N, B, A, b, c, v)$  a forma relaxada resultante para  $L_{aux}$ 
6    $\triangleright L_{aux}$  tem  $n + 1$  variáveis não básicas e  $m$  variáveis básicas.
7    $(N, B, A, b, c, v) \leftarrow \text{PIVOT}(N, B, A, b, c, v, l, 0)$ 
8    $\triangleright$  A solução básica é agora possível para  $L_{aux}$ .
9   repetir o loop while das linhas 2 a 11 de SIMPLEX até encontrar uma
      solução ótima para  $L_{aux}$ 
10  if a solução básica define  $\bar{x}_0 = 0$ 
11    then return a forma final relaxada com  $x_0$  removido e
        a função de objetivo original restaurada
12  else return "impossível"

```

INITIALIZE-SIMPLEX funciona da maneira descrita a seguir. Nas linhas 1 a 3, testamos implicitamente a solução básica para a forma relaxada inicial de  $L$  dada por  $N = \{1, 2, \dots, n\}, B = \{n + 1, n + 2, \dots, n + m\}, \bar{x}_i = b_i$  para todo  $i \in B$  e  $\bar{x}_j = 0$  para todo  $j \in N$ . (A criação da forma relaxada não exige nenhum esforço explícito, pois os valores de  $A, b$  e  $c$  são os mesmos tanto na forma relaxada quanto na forma padrão.) Se essa solução básica é possível – isto é,  $\bar{x}_i \geq 0$  para todo  $i \in N \cup B$  – então a forma relaxada é retornada. Caso contrário, na linha 4, formamos o programa linear auxiliar  $L_{aux}$  como no Lema 29.11. Tendo em vista que a solução básica inicial para  $L$  não é possível, a solução básica inicial para a forma relaxada correspondente a  $L_{aux}$  também não será possível. Então, na linha 7, executamos uma chamada de PIVOT, com  $x_0$  entrando e  $x_l$  saindo, onde o índice  $l$  é escolhido na linha 1 como o índice do  $b_i$  mais negativo. Veremos em breve que a solução básica resultante dessa chamada de PIVOT será possível. Agora que temos uma forma relaxada para a qual a solução básica é possível, podemos na linha 9 chamar repetidamente PIVOT para resolver por completo o programa linear auxiliar. Como o teste da linha 10 demonstra, se encontrarmos uma solução ótima para  $L_{aux}$  com valor de objetivo 0, então na linha 11 criamos uma forma relaxada para  $L$ , para a qual a solução básica é possível. Para isso, eliminamos todos os termos  $x_0$  das restrições e restauramos a função de objetivo original para  $L$ . A função de objetivo original pode conter tanto variáveis básicas quanto não básicas. Portanto, na função de objetivo, substituímos cada variável básica pelo lado direito de sua restrição associada. Por outro lado, se na linha 10 descobrimos que o programa linear original  $L$  é impossível, retornamos essa informação na linha 12.

Agora demonstramos a operação de INITIALIZE-SIMPLEX no programa linear (29.105) a (29.108). Esse programa linear é possível se podemos encontrar valores não negativa para  $x_1$  e  $x_2$  que satisfaçam às desigualdades (29.106) e (29.107). Usando o Lema 29.11, formulamos o programa linear auxiliar

$$\text{maximizar} \quad -x_0 \tag{29.112}$$

sujeito a

$$2x_1 - x_2 - x_0 \leq 2 \tag{29.113}$$

$$x_1 - 5x_2 - x_0 \leq -4 \tag{29.114}$$

Pelo Lema 29.11, se o valor de objetivo ótimo desse programa linear auxiliar é 0, então o programa linear original tem uma solução possível. Se o valor de objetivo ótimo desse programa linear auxiliar é positivo, então o programa linear original não tem uma solução possível.

Escrevemos esse programa linear em forma relaxada, obtendo

$$\begin{aligned} z &= -x_0 \\ x_3 &= 2 - 2x_1 + x_2 + x_0 \\ x_4 &= -4 - x_1 + 5x_2 + x_0. \end{aligned}$$

Ainda não terminamos, porque a solução básica, que seria definida como  $x_4 = -4$ , não é possível para esse programa linear auxiliar. Contudo, podemos com uma chamada a PIVOT, converter essa forma relaxada em uma forma na qual a solução básica é possível. Como indica a linha 7, escolhemos  $x_0$  como a variável de entrada. Na linha 1, escolhemos como variável de saída  $x_4$ , que é a variável básica cujo valor na solução básica é mais negativo. Depois da criação de pivô, temos a forma relaxada

$$\begin{aligned} z &= -4 - x_1 + 5x_2 - x_4 \\ x_0 &= 4 + x_1 - 5x_2 + x_4 \\ x_3 &= 6 - x_1 - 4x_2 + x_4. \end{aligned}$$

A solução básica associada é  $(x_0, x_1, x_2, x_3, x_4) = (4, 0, 0, 6, 0)$ , que é possível. Agora chamamos repetidamente PIVOT até obtermos uma solução ótima para  $L_{\text{aux}}$ . Nesse caso, uma chamada a PIVOT com  $x_2$  entrando e  $x_0$  saindo produz

$$\begin{aligned} z &= -x_0 \\ x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5}. \end{aligned}$$

Essa forma relaxada é a solução final para o problema auxiliar. Tendo em vista que essa solução tem  $x_0 = 0$ , sabemos que nosso problema inicial era possível. Além disso, tendo em vista que  $x_0 = 0$ , podemos simplesmente removê-la do conjunto de restrições. Podemos então usar a função de objetivo original, com substituições apropriadas feitas para incluir apenas variáveis não básicas. Em nosso exemplo, obtemos a função de objetivo

$$2x_1 - x_2 = 2x_1 - \left( \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \right).$$

Definindo  $x_0 = 0$  e simplificando, obtemos a função de objetivo

$$\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5},$$

e a forma relaxada

$$\begin{aligned} z &= \frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ x_2 &= \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{9x_1}{5} - \frac{x_4}{5}. \end{aligned}$$

Essa forma relaxada tem uma solução básica possível, e podemos retorná-la ao procedimento SIMPLEX.

Agora mostramos formalmente a correção de INITIALIZE-SIMPLEX.

### Lema 29.12

Se um programa linear  $L$  não tem nenhuma solução possível, então INITIALIZE-SIMPLEX retorna “impossível”. Caso contrário, ele retorna uma forma relaxada válida para a qual a solução básica é possível.

**Prova** Primeiro, suponha que o programa linear  $L$  não tem nenhuma solução possível. Então, pelo Lema 29.11, o valor de objetivo ótimo de  $L_{\text{aux}}$ , definido em (29.109) a (29.111) é não nulo e, pela restrição de não negatividade em  $x_0$ , uma solução ótima deve ter um valor de objetivo negativo. Além disso, esse valor de objetivo deve ser finito, tendo em vista que a configuração  $x_i = 0$ , para  $i = 1, 2, \dots, n$  e  $x_0 = |\min_{i=1}^m \{b_i\}|$  é possível, e essa solução tem valor de objetivo  $-\lfloor \min_{i=1}^m \{b_i\} \rfloor$ . Então, a linha 9 de INITIALIZE-SIMPLEX encontrará uma solução com um valor de objetivo negativo. Seja  $\bar{x}$  a solução básica associada à forma relaxada final. Não podemos ter  $\bar{x}_0 = 0$ , porque então  $L_{\text{aux}}$  teria valor de objetivo 0, contradizendo o fato de que o valor de objetivo é negativo. Desse modo, o teste na linha 10 resulta no retorno de “impossível” na linha 12.

Suponha agora que o programa linear  $L$  tenha uma solução possível. Do Exercício 29.3-3, sabemos que, se  $b_i \geq 0$  para  $i = 1, 2, \dots, m$ , então a solução básica associada à forma relaxada inicial é possível. Nesse caso, as linhas 2 e 3 retornarão a forma relaxada associada à entrada. (Não existe muito a ser feito para converter a forma padrão em forma relaxada, pois  $A$ ,  $b$  e  $c$  são iguais em ambas.)

No restante da prova, trataremos o caso em que o programa linear é possível, mas não retornamos na linha 3. Demonstramos que, nesse caso, as linhas 4 a 9 encontram uma solução possível para  $L_{\text{aux}}$  com valor de objetivo 0. Primeiro, pelas linhas 1 e 2, devemos ter

$$b_l < 0$$

e

$$b_l \leq b_i \text{ para cada } i \in B. \quad (29.115)$$

Na linha 7, executamos uma operação de pivô em que a variável de saída  $x_e$  é o lado esquerdo da equação com  $b_e$  mínimo, e a variável de entrada é  $x_0$ , a variável extra adicionada. Agora mostramos que, depois desse pivô, todas as entradas de  $b$  são não negativas e, consequentemente, a solução básica para  $L_{\text{aux}}$  é possível. Sendo  $\bar{x}$  a solução básica depois da chamada a PIVOT, e sendo  $\hat{b}$  e  $\hat{B}$  valores retornados por PIVOT, o Lema 29.1 implica que

$$\bar{x}_i = \begin{cases} b_i - a_{ie} \hat{b}_e & \text{se } i \in \hat{B} - \{e\}, \\ b_i/a_{ie} & \text{se } i = e. \end{cases} \quad (29.116)$$

A chamada a PIVOT na linha 7 tem  $e = 0$  e, por (29.110), temos que

$$646 \mid a_{i0} = a_{ie} = -1 \text{ para cada } i \in B. \quad (29.117)$$

(Observe que  $a_{i0}$  é o coeficiente de  $x_0$  que aparece em (29.110), e não a negação do coeficiente, porque  $L_{\text{aux}}$  está em forma padrão, em vez de estar em forma relaxada.) Tendo em vista que  $l \in B$ , também temos que  $a_{le} = -1$ . Desse modo,  $b_l/a_{le} > 0$ , e assim  $\bar{x}_e > 0$ . Para as variáveis básicas restantes, temos

$$\begin{aligned}\bar{x}_i &= b_i - a_{ie}\hat{b}_e && (\text{pela equação (29.116)}) \\ &= b_i - a_{ie}(b_l/a_{le}) && (\text{pela linha 2 de PIVOT}) \\ &= b_i - b_l && (\text{pela equação (29.117) e } a_{le} = -1) \\ &\geq 0 && (\text{pela desigualdade (29.115)}),\end{aligned}$$

que implica que cada variável básica é agora não negativa. Conseqüentemente, a solução básica depois da chamada a PIVOT na linha 7 é possível. Em seguida, executamos a linha 9, o que resolve  $L_{\text{aux}}$ . Tendo em vista que supomos que  $L$  tem uma solução possível, o Lema 29.11 implica que  $L_{\text{aux}}$  tem uma solução ótima com valor de objetivo 0. Considerando que todas as formas relaxadas são equivalentes, a solução básica final para  $L_{\text{aux}}$  deve ter  $\bar{x}_0 = 0$  e, depois de remover  $x_0$  do programa linear, obtemos uma forma relaxada que é possível para  $L$ . Essa forma relaxada é então retornada na linha 10. ■

## Teorema fundamental de programação linear

Concluímos este capítulo mostrando que o procedimento SIMPLEX funciona. Em particular, qualquer programa linear é impossível, é ilimitado ou tem uma solução ótima com um valor de objetivo finito e, em cada caso, SIMPLEX agirá de forma apropriada.

### *Teorema 29.13 (Teorema fundamental de programação linear)*

Qualquer programa linear  $L$ , dado em forma padrão,

1. tem uma solução ótima com um valor de objetivo finito,
2. é impossível, ou
3. é ilimitado.

Se  $L$  é impossível, SIMPLEX retorna “impossível”. Se  $L$  é ilimitado, SIMPLEX retorna “ilimitado”. Caso contrário, SIMPLEX retorna uma solução ótima com um valor de objetivo finito.

**Prova** Pelo Lema 29.12, se o programa linear  $L$  é impossível, então SIMPLEX retorna “impossível”. Agora, suponha que o programa linear  $L$  é possível. Pelo Lema 29.12, INITIALIZE-SIMPLEX retorna uma forma relaxada para a qual a solução básica é possível. Então, pelo Lema 29.7, SIMPLEX retorna “ilimitado” ou termina com uma solução possível. Se ele terminar com uma solução finita, então o Teorema 29.10 nos diz que essa solução é ótima. Por outro lado, se SIMPLEX retornar “ilimitado”, o Lema 29.2 nos diz que o programa linear  $L$  é realmente ilimitado. Tendo em vista que SIMPLEX sempre termina de um desses modos, a prova está completa. ■

## Exercícios

### 29.5-1

Forneça pseudocódigo detalhado para implementar as linhas 5 e 11 de INITIALIZE-SIMPLEX.

### 29.5-2

Mostre que, quando INITIALIZE-SIMPLEX executar o loop principal de SIMPLEX, “ilimitado” nunca será retornado.

### 29.5-3

Suponha que recebemos um programa linear  $L$  em forma padrão e suponha que, tanto para  $L$  quanto para o dual de  $L$ , as soluções básicas associadas com as formas relaxadas iniciais são possíveis. Mostre que o valor de objetivo ótimo de  $L$  é 0.

### 29.5-4

Suponha que permitimos desigualdades estritas em um programa linear. Mostre que, nesse caso, o teorema fundamental de programação linear não é válido.

### 29.5-5

Resolva o seguinte programa linear usando SIMPLEX:

$$\text{maximizar} \quad x_1 + 3x_2$$

sujeito a

$$-x_1 + x_2 \leq -1$$

$$-2x_1 - 2x_2 \leq -6$$

$$-x_1 + 4x_2 \leq 2$$

$$x_1, x_2 \geq 0.$$

### 29.5-6

Resolva o programa linear dado em (29.6) a (29.10).

### 29.5-7

Considere o seguinte programa linear de 1 variável, que chamamos  $P$ :

$$\text{maximizar} \quad tx$$

sujeito a

$$rx \leq s$$

$$x \geq 0,$$

onde  $r, s$  e  $t$  são números reais arbitrários. Seja  $D$  o dual de  $P$ .

Enuncie para quais valores de  $r, s$  e  $t$  você pode afirmar que

1. Tanto  $P$  quanto  $D$  têm soluções ótimas com valores de objetivo finitos.
2.  $P$  é possível, mas  $D$  é impossível.
3.  $D$  é possível, mas  $P$  é impossível.
4. Nem  $P$  nem  $D$  é possível.

## Problemas

### 29-1 Possibilidade de desigualdade linear

Dado um conjunto de  $m$  desigualdades lineares em  $n$  variáveis  $x_1, x_2, \dots, x_n$ , o **problema de possibilidade de desigualdade linear** pergunta se existe uma configuração das variáveis que satisfaça simultaneamente a cada uma das desigualdades.

- a. Mostre que, se tivermos um algoritmo para programação linear, podemos usá-lo para resolver o problema de possibilidade de desigualdade linear. O número de variáveis e restrições que você utilizará no problema de programação linear deve ser polinomial em  $n$  e  $m$ .

- b.** Mostre que, se tivermos um algoritmo para problema de possibilidade de desigualdade linear, podemos usá-lo para resolver um problema de programação linear. O número de variáveis e desigualdades lineares que você utilizará no problema de possibilidade de desigualdade linear deve ser polinomial em  $n$  e  $m$ , o número de variáveis e restrições no programa linear.

### 29-2 Relaxamento complementar

O relaxamento complementar descreve um relacionamento entre os valores de variáveis primitivas e restrições duais, e entre os valores de variáveis duais e restrições primitivas. Seja  $\bar{x}$  uma solução ótima para um programa linear primitivo dado em (29.16) a (29.18), e seja  $\bar{y}$  a solução ótima para o programa linear dual dado em (29.86) a (29.88). O relaxamento complementar declara que as seguintes condições são necessárias e suficientes para  $\bar{x}$  e  $\bar{y}$  serem ótimas:

$$\sum_{i=1}^m a_{ij} \bar{y}_i = c_j \text{ ou } \bar{x}_j = 0 \quad \text{para } j = 1, 2, \dots, n$$

e

$$\sum_{i=1}^m a_{ij} \bar{x}_i = b_j \text{ ou } \bar{y}_j = 0 \quad \text{para } i = 1, 2, \dots, m.$$

- a.** Verifique se o relaxamento complementar é válido para o programa linear das linhas (29.56) a (29.60).
- b.** Prove que o relaxamento complementar é válido para qualquer programa linear primitivo e seu dual correspondente.
- c.** Prove que uma solução possível  $\bar{x}$  para um programa linear primitivo dado nas linhas (29.16) a (29.18) é ótima se e somente se existem valores  $\bar{y} = (\bar{y}_1, \bar{y}_2, \dots, \bar{y}_m)$  tais que
  1.  $\bar{y}$  é uma solução possível para o programa linear dual dado em (29.86) a (29.88),
  2.  $\sum_{i=1}^m a_{ij} \bar{y}_i = c_j$  sempre que  $\bar{x}_j > 0$ , e
  3.  $\bar{y}_i = 0$  sempre  $\sum_{j=1}^m a_{ij} \bar{x}_j = b_i$ .

### 29-3 Programação linear de inteiros

Um problema de programação linear de inteiros é um problema de programação linear com a restrição adicional de que as variáveis  $x$  devem assumir valores inteiros. O Exercício 34.5-3 mostra que simplesmente determinar se um programa linear de inteiros tem uma solução possível é NP-difícil, o que significa que é improvável que exista um algoritmo de tempo polinomial para esse problema.

- a.** Mostre que a dualidade fraca (Lema 29.8) é válida para um programa linear de inteiros.
- b.** Mostre que a dualidade (Teorema 29.10) nem sempre é válida para um programa linear de inteiros.
- c.** Dado um programa linear primitivo em forma padrão, vamos definir  $P$  como o valor de objetivo ótimo para o programa linear primitivo,  $D$  como o valor de objetivo ótimo para seu dual,  $IP$  como o valor de objetivo ótimo para a versão de inteiros do primitivo (ou seja, o primitivo com a restrição adicional de que as variáveis assumem valores inteiros) e  $ID$  como o valor de objetivo ótimo para a versão de inteiros do dual. Mostre que  $IP \leq P = D \leq ID$ .

### 29-4 Lema de Farkas

Seja  $A$  uma matriz  $m \times n$  e  $b$  um vetor de  $m$  elementos. Então, o lema de Farkas declara que exatamente um dos sistemas

$$Ax \leq 0,$$

$$bx > 0$$

e

$$yA = b,$$

$$y \geq 0$$

pode ser resolvido, onde  $x$  é um vetor de  $n$  elementos e  $y$  é um vetor de  $m$  elementos. Prove o lema de Farkas.

## Notas do capítulo

Este capítulo só iniciou o estudo do amplo campo da programação linear. Vários livros se dedicam exclusivamente à programação linear, inclusive os de Chvátal [62], Gass [111], Karloff [171], Schrijver [266] e Vanderbei [304]. Muitos outros livros apresentam uma boa cobertura de programação linear, incluindo os de Papadimitriou e Steiglitz [237], e Ahuja, Magnanti e Orlin [7]. A cobertura deste capítulo se baseia na abordagem adotada por Chvátal.

O algoritmo simplex para programação linear foi inventado por G. Dantzig em 1947. Logo depois, foi descoberto que diversos problemas em uma variedade de campos podiam ser formulados como programas lineares e resolvidos com o algoritmo simplex. Essa realização levou ao florescimento de usos de programação linear, juntamente com vários algoritmos. As variantes do algoritmo simplex continuam a ser os métodos mais populares para resolução de problemas de programação linear. Essa história é descrita em vários lugares, inclusive nas notas de [62] e [171].

O algoritmo de elipsóide foi o primeiro algoritmo de tempo polinomial para programação linear, e se deve a L. G. Khachian em 1979; ele se baseou no trabalho anterior de N. Z. Shor, D. B. Judin e A. S. Nemirovskii. O uso do algoritmo de elipsóide para resolver uma variedade de problemas em otimização combinatória descrito no trabalho de Grötschel, Lovász e Schrijver [134]. Até hoje, o algoritmo de elipsóide não parece ser competidor para o algoritmo simplex na prática.

O artigo de Karmarkar [172] inclui uma descrição de seu algoritmo de ponto interior. Muitos pesquisadores subsequentes projetaram algoritmos de ponto interior. Boas pesquisas podem ser encontradas no artigo de Goldfarb e Todd [122] e no livro de Ye [319].

A análise do algoritmo simplex é uma área ativa de pesquisa. Klee e Minty construíram um exemplo em que o algoritmo simplex executa  $2^n - 1$  iterações. O algoritmo simplex normalmente funciona muito bem na prática, e muitos pesquisadores tentaram dar justificativa teórica a essa observação empírica. Uma linha de pesquisa iniciada por Borgwardt e seguida por muitos outros mostra que, sob certas hipóteses probabilísticas em relação à entrada, o algoritmo simplex converge em tempo polinomial esperado. O progresso recente nessa área se deve a Spielman e Teng [284], que apresentam a “análise suavizada de algoritmos” e a aplicam ao algoritmo simplex.

O algoritmo simplex é conhecido por funcionar de modo mais eficiente em certos casos especiais. Em particular, vale a pena notar o algoritmo simplex de rede, que é o algoritmo simplex especializado para problemas de fluxo em rede. Para certos problemas de rede, inclusive os problemas de caminhos mais curtos, fluxo máximo e fluxo de custo mínimo, as variantes do algoritmo simplex em rede funcionam em tempo polinomial. Por exemplo, consulte o artigo de Orlin [234] e as citações que ele contém.

---

## Capítulo 30

# Polinômios e a FFT

O método direto para somar dois polinômios de grau  $n$  demora o tempo  $\Theta(n)$ , mas o método direto para multiplicá-los demora o tempo  $\Theta(n^2)$ . Neste capítulo, mostraremos como a transformação rápida de Fourier (FFT – Fast Fourier Transform) pode reduzir o tempo para multiplicar polinômios a  $\Theta(n \lg n)$ .

O uso mais comum das transformações de Fourier, e consequentemente da FFT, é no processamento de sinais. Um sinal é dado no **domínio de tempo**: como uma função de mapeamento de tempo para amplitude. A análise de Fourier nos permite expressar o sinal como uma soma ponderada de senóides de freqüências variadas deslocadas em fase. Os pesos e as fases associados com as freqüências caracterizam o sinal no **domínio de freqüência**. O processamento de sinais é uma área rica, para a qual existem vários livros de boa qualidade; as notas do capítulo fazem referência a alguns deles.

### Polinômios

Um **polinômio** na variável  $x$  sobre um campo algébrico  $F$  é a representação de uma função  $A(x)$  como uma soma formal:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j .$$

Chamamos os valores  $a_0, a_1, \dots, a_{n-1}$  de **coeficientes** do polinômio. Os coeficientes são tirados do campo  $F$ , em geral do conjunto  $\mathbb{C}$  de números complexos. Um polinômio  $A(x)$  é dito de **grau**  $k$  se seu coeficiente mais alto diferente de zero é  $a_k$ . Qualquer inteiro estritamente maior que o grau de um polinômio é um **limite de grau** desse polinômio. Desse modo, o grau de um polinômio de limite de grau  $n$  pode ser qualquer inteiro entre 0 e  $n - 1$ , inclusive.

Existe uma variedade de operações que poderíamos pretender definir para polinômios. No caso da **adição de polinômios**, se  $A(x)$  e  $B(x)$  são polinômios de limite de grau  $n$ , dizemos que sua **soma** é um polinômio  $C(x)$ , também de limite de grau  $n$ , tal que  $C(x) = A(x) + B(x)$  para todo  $x$  no campo subjacente. Isto é, se

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

e

$$B(x) = \sum_{j=0}^{n-1} b_j x^j ,$$

então

$$C(x) = \sum_{j=0}^{n-1} c_j x^j ,$$

onde  $c_j = a_j + b_j$  para  $j = 0, 1, \dots, n - 1$ . Por exemplo, se  $A(x) = A(x) = 6x^3 + 7x^2 - 10x + 9$  e  $B(x) = -2x^3 + 4x - 5$ , então  $C(x) = 4x^3 + 7x^2 - 6x + 4$ .

No caso da **multiplicação de polinômios**, se  $A(x)$  e  $B(x)$  são polinômios de limite de grau  $n$ , dizemos que seu **produto**  $C(x)$  é um polinômio de limite de grau  $2n - 1$  tal que  $C(x) = A(x)B(x)$  para todo  $x$  no campo subjacente. É provável que você tenha multiplicado polinômios antes, multiplicando cada termo de  $A(x)$  por cada termo de  $B(x)$  e combinando os termos com potências iguais. Por exemplo, podemos multiplicar  $A(x) = 6x^3 + 7x^2 - 10x + 9$  e  $B(x) = -2x^3 + 4x - 5$  desta forma:

$$\begin{array}{r} 6x^3 + 7x^2 - 10x + 9 \\ - 2x^3 \\ \hline - 30x^3 - 35x^2 + 50x - 45 \\ 24x^4 + 28x^3 - 40x^2 + 36x \\ \hline - 12x^6 - 14x^5 + 20x^4 - 18x^3 \\ \hline - 12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45 \end{array}$$

Outra maneira de expressar o produto  $C(x)$  é

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \quad (30.1)$$

onde

$$c_j = \sum_{k=0}^j a_k b_{j-k} . \quad (30.2)$$

Observe que  $\text{grau}(C) = \text{grau}(A) + \text{grau}(B)$ , implicando

$$\begin{aligned} \text{limite de } \text{grau}(C) &= \text{limite de } \text{grau}(A) + \text{limite de } \text{grau}(B) - 1 \\ &= \text{limite de } \text{grau}(A) + \text{limite de } \text{grau}(B) . \end{aligned}$$

Apesar disso, falaremos no limite de grau de  $C$  como sendo a soma dos limites de graus de  $A$  e  $B$  pois, se um polinômio tem limite de grau  $k$  ele também tem limite de grau  $k + 1$ .

## Esboço do capítulo

A Seção 30.1 apresenta dois modos de representar polinômios: a representação de coeficientes e a representação de valores de pontos. Os métodos diretos para multiplicar polinômios – equações (30.1) e (30.2) – demoram o tempo  $\Theta(n^2)$  quando os polinômios são representados em forma de coeficientes, mas apenas o tempo  $\Theta(n)$  quando eles são representados em forma de valores de pontos. Porém, podemos multiplicar polinômios usando a representação de coeficientes somente no tempo  $\Theta(n \lg n)$ , fazendo a conversão entre as duas representações. Para verificar por que isso funciona, primeiro devemos estudar raízes complexas da unidade, o que faremos na Seção 30.2. Em seguida, usaremos a FFT e sua inversa, também descrita na Seção 30.2, para

executar as conversões. A Seção 30.3 mostra como implementar a FFT com rapidez, tanto em modelos seriais quanto paralelos.

Este capítulo usa extensivamente números complexos, e o símbolo  $i$  será usado exclusivamente para denotar  $\sqrt{-1}$ .

## 30.1 Representação de polinômios

As representações de coeficientes e valores de pontos de polinômios são em certo sentido equivalentes; isto é, um polinômio na forma de valores de pontos tem uma contrapartida exclusiva em forma de coeficientes. Nesta seção, vamos introduzir as duas representações e mostrar como elas podem ser combinadas para permitir a multiplicação de dois polinômios de limite de grau  $n$  no tempo  $\Theta(n \lg n)$ .

### Representação de coeficientes

Uma *representação de coeficientes* de um polinômio  $A(x) = \sum_{j=0}^{n-1} a_j x^j$  de limite de grau  $n$  é um vetor de coeficientes  $a = (a_0, a_1, \dots, a_{n-1})$ . Nas equações matriciais deste capítulo, em geral trataremos vetores como vetores coluna.

A representação de coeficientes é conveniente para certas operações sobre polinômios. Por exemplo, a operação de *avaliar* o polinômio  $A(x)$  em um determinado ponto  $x_0$  consiste em calcular o valor de  $A(x_0)$ . A avaliação demora o tempo  $\Theta(n)$  usando-se a *regra de Horner*:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \dots + x_0(a_{n-2} + x_0(a_{n-1}) \dots))) .$$

De modo semelhante, a adição de dois polinômios representados pelos vetores de coeficientes  $a = (a_0, a_1, \dots, a_{n-1})$  e  $b = (b_0, b_1, \dots, b_{n-1})$  demora o tempo  $\Theta(n)$ : simplesmente damos saída ao vetor de coeficientes  $c = (c_0, c_1, \dots, c_{n-1})$ , onde  $c^j = a_j + b_j$  para  $j = 0, 1, \dots, n-1$ .

Agora, considere a multiplicação de dois polinômios de limite de grau  $n$   $A(x)$  e  $B(x)$ , representados em forma de coeficientes. Se usarmos o método descrito pelas equações (30.1) e (30.2), a multiplicação de polinômios irá demorar o tempo  $\Theta(n^2)$ , pois cada coeficiente no vetor  $a$  deverá ser multiplicado por cada coeficiente no vetor  $b$ . A operação de multiplicar polinômios em forma de coeficientes parece ser consideravelmente mais difícil que a de avaliar um polinômio ou somar dois polinômios. O vetor de coeficientes resultante  $c$ , dado pela equação (30.2), também é chamado *convolução* dos vetores de entrada  $a$  e  $b$ , sendo denotado por  $c = a \otimes b$ . Tendo em vista que a multiplicação de polinômios e o cálculo de convoluções são problemas computacionais fundamentais de considerável importância prática, este capítulo se concentra em algoritmos eficientes para eles.

### Representação de valores de pontos

Uma *representação de valores de pontos* de um polinômio  $A(x)$  de limite de grau  $n$  é um conjunto de  $n$  pares de valores de pontos

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$$

tal que todos os valores  $x_k$  são distintos e

$$y_k = A(xk) \tag{30.3}$$

para  $k = 0, 1, \dots, n-1$ . Um polinômio tem muitas representações de valores de pontos diferentes, pois qualquer conjunto de  $n$  pontos distintos  $x_0, x_1, \dots, x_{n-1}$  pode ser usado como base para a representação.

O cálculo de uma representação de valores de pontos para um dado polinômio em forma de coeficientes é em princípio direto, pois tudo que temos de fazer é selecionar  $n$  pontos distintos  $x_0, x_1, \dots, x_{n-1}$ , e depois avaliar  $A(x_k)$  para  $k = 0, 1, \dots, n-1$ . Com o método de Horner, essa avaliação de  $n$  pontos demora o tempo  $\Theta(n^2)$ . Veremos mais tarde que, se escolhermos  $x_k$  de modo inteligente, esse cálculo poderá ser acelerado para execução no tempo  $\Theta(n \lg n)$ .

O inverso da avaliação – a determinação da forma de coeficientes de um polinômio a partir de uma representação de valores de pontos – é chamada **interpolação**. O teorema a seguir mostra que a interpolação é bem definida, supondo-se que o limite de grau do polinômio de interpolação seja igual ao número de pares de valores de pontos dados.

### **Teorema 30.1 (Unicidade de polinômio de interpolação)**

Para qualquer conjunto  $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$  de  $n$  pares de valores de pontos, existe um único polinômio  $A(x)$  de limite de grau  $n$  tal que  $y_k = A(x_k)$  para  $k = 0, 1, \dots, n-1$ .

**Prova** A prova é baseada na existência da inversa de uma certa matriz. A equação (30.3) é equivalente à equação matricial

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}. \quad (30.4)$$

A matriz no lado esquerdo é denotada por  $V(x_0, x_1, \dots, x_{n-1})$  e é conhecida como uma matriz de Vandermonde. Pelo Exercício 28.1-11, essa matriz tem determinante

$$\prod_{0 \leq j < k \leq n-1} (x_k - x_j).$$

e, portanto, pelo Teorema 28.5, ela é invertível (isto é, não singular) se os valores  $x_k$  são distintos. Desse modo, os coeficientes  $a_j$  podem ser resolvidos por exclusividade, dada a representação de valores de pontos:

$$a = V(x_0, x_1, \dots, x_{n-1})^{-1}y.$$

A prova do Teorema 30.1 descreve um algoritmo para interpolação baseado na resolução do conjunto (30.4) de equações lineares. Usando os algoritmos de decomposição LU do Capítulo 28, podemos resolver essas equações no tempo  $O(n^3)$ .

Um algoritmo mais rápido para a interpolação de  $n$  pontos se baseia na **fórmula de Lagrange**:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}. \quad (30.5)$$

Talvez você queira verificar se o lado direito da equação (30.5) é um polinômio de limite de grau  $n$  que satisfaz a  $A(x_i) = y_i$  para todo  $i$ . O Exercício 30.1-5 lhe pede para mostrar como calcular os coeficientes de  $A$  usando a fórmula de Lagrange no tempo  $\Theta(n^2)$ .

Desse modo, a avaliação de  $n$  pontos e a interpolação são operações inversas bem definidas que fazem a transformação entre a representação de coeficientes de um polinômio e uma representação de valores de pontos.<sup>1</sup> Os algoritmos descritos anteriormente para esses problemas demoram o tempo  $\Theta(n^2)$ .

---

<sup>1</sup>A interpolação é um problema notoriamente complicado do ponto de vista da estabilidade numérica. Embora as abordagens descritas aqui sejam matematicamente corretas, pequenas diferenças nas entradas ou erros de arredondamento durante os cálculos podem gerar grandes diferenças no resultado.

A representação de valores de pontos é bastante conveniente para muitas operações sobre polinômios. No caso da adição, se  $C(x) = A(x) + B(x)$ , então  $C(x_k) = A(x_k) + B(x_k)$  para qualquer ponto  $x_k$ . Mais precisamente, se temos uma representação de valores de pontos para  $A$ ,

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\},$$

e para  $B$ ,

$$\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{n-1}, y'_{n-1})\}$$

(observe que  $A$  e  $B$  são avaliados nos *mesmos*  $n$  pontos), então uma representação de valores de pontos para  $C$  é

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \dots, (x_{n-1}, y_{n-1} + y'_{n-1})\}.$$

Portanto, o tempo para somar dois polinômios de limite de grau  $n$  na forma de valores de pontos é  $\Theta(n)$ .

De modo semelhante, a representação de valores de pontos é conveniente para multiplicação de polinômios. Se  $C(x) = A(x)B(x)$ , então  $C(x_k) = A(x_k)B(x_k)$  para qualquer ponto  $x_k$ , e podemos multiplicar ponto a ponto uma representação de valores de pontos para  $A$  por uma representação de valores de pontos para  $B$ , a fim de obter uma representação de valores de pontos para  $C$ . Contudo, devemos encarar o problema de que o limite de grau de  $C$  é a soma dos limites de graus para  $A$  e  $B$ . Uma representação padrão de valores de pontos para  $A$  e  $B$  consiste em  $n$  pares de valores de pontos para cada polinômio. A multiplicação desses valores nos dá  $n$  pares de valores de pontos para  $C$  mas, como o limite de grau de  $C$  é  $2n$ , precisamos de  $2n$  pares de valores de pontos para uma representação de valores de pontos de  $C$ . (Veja o Exercício 30.1-4.) Desse modo, devemos começar com representações de valores de pontos “estendidas” para  $A$  e  $B$ , consistindo em  $2n$  pares de valores de pontos cada uma. Dada uma representação de valores de pontos estendida para  $A$ ,

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{2n-1}, y_{2n-1})\},$$

e uma representação de valores de pontos estendida correspondente para  $B$ ,

$$\{(x_0, y'_0), (x_1, y'_1), \dots, (x_{n-1}, y'_{2n-1})\},$$

então uma representação de valores de pontos para  $C$  é

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \dots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\},$$

Dados dois polinômios de entrada em forma de valores de pontos estendida, vemos que o tempo para multiplicá-los com a finalidade de obter a forma de valores de pontos do resultado é  $\Theta(n)$ , muito menor que o tempo necessário para multiplicar polinômios em forma de coeficientes.

Finalmente, vamos considerar como avaliar um polinômio dado em forma de valores de pontos em um novo ponto. Para esse problema, não existe aparentemente nenhuma abordagem mais simples que converter o polinômio primeiro em forma de coeficientes, e depois avaliá-lo no novo ponto.

## Multiplicação rápida de polinômios em forma de coeficientes

Podemos usar o método de multiplicação de tempo linear para polinômios em forma de valores de pontos para acelerar a multiplicação de polinômios em forma de coeficiente? A resposta depende de nossa capacidade de converter um polinômio rapidamente da forma de coeficientes para a forma de valores de pontos (avaliar) e vice-versa (interpolar).

Podemos usar quaisquer pontos que desejarmos como pontos de avaliação mas, escolhendo os pontos de avaliação cuidadosamente, será possível fazer a conversão entre representações somente no tempo  $\Theta(n \lg n)$ . Como veremos na Seção 30.2, se escolhermos “raízes complexas da unidade” como pontos de avaliação, poderemos produzir uma representação de valores de pontos tomando a transformação discreta de Fourier (DFT – Discrete Fourier Transform) de um vetor de coeficientes. A operação inversa, a interpolação, pode ser executada tomando-se a “DFT inversa” de pares de valores de pontos, produzindo um vetor de coeficientes. A Seção 30.2 mostrará como a FFT executa as operações DFT e DFT inversa no tempo  $\Theta(n \lg n)$ .

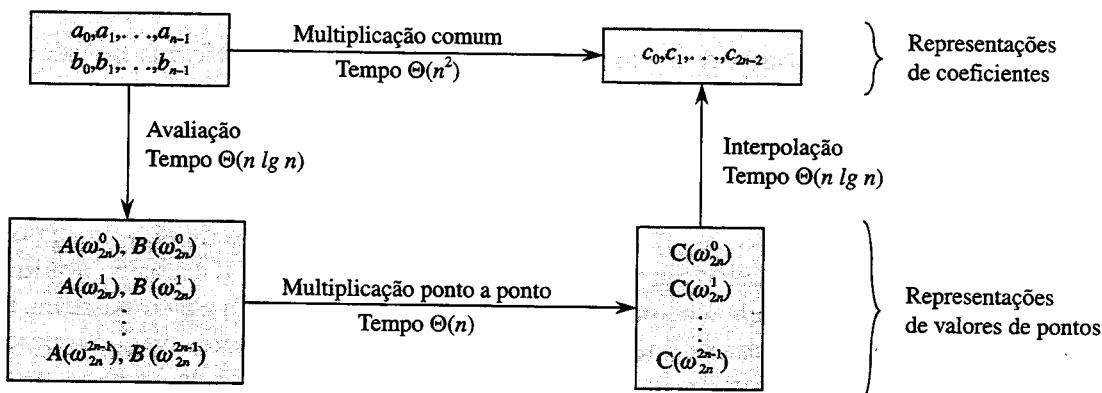


FIGURA 30.1 Um esboço gráfico de um processo de multiplicação eficiente de polinômios. As representações na parte superior estão em forma de coeficientes, enquanto as da parte inferior estão em forma de valores de pontos. As setas da esquerda para a direita correspondem à operação de multiplicação. Os  $\omega_{2n}$  termos são raízes  $(2n)$ -ésimas complexas da unidade

A Figura 30.1 representa graficamente essa estratégia. Um detalhe secundário se refere aos limites de graus. O produto de dois polinômios de limite de grau  $n$  é um polinômio de limite de grau  $2n$ . Assim, antes de avaliar os polinômios de entrada  $A$  e  $B$ , primeiro devemos duplicar seus limites de graus para  $2n$ , somando  $n$  coeficientes de alta ordem iguais a 0. Como os vetores têm  $2n$  elementos, usamos “raízes  $(2n)$ -ésimas complexas da unidade”, que são denotados pelos  $\omega_{2n}$  termos na Figura 30.1.

Dada a FFT, temos o procedimento de tempo  $\Theta(n \lg n)$  a seguir para multiplicar dois polinômios  $A(x)$  e  $B(x)$  de limite de grau  $n$ , onde as representações de entrada e saída estão em forma de coeficientes. Supomos que  $n$  é uma potência de 2; esse requisito sempre pode ser satisfeito adicionando-se coeficientes zero de alta ordem.

- 1. Duplicar o limite de grau:** Criar representações de coeficientes de  $A(x)$  e  $B(x)$  como polinômios de limite de grau  $2n$ , adicionando  $n$  coeficientes zero de alta ordem a cada um.
- 2. Avaliar:** Calcular representações de valores de pontos de  $A(x)$  e  $B(x)$  de comprimento  $2n$  através de duas aplicações da FFT de ordem  $2n$ . Essas representações contêm os valores dos dois polinômios nas raízes  $(2n)$ -ésimas da unidade.
- 3. Multiplicação ponto a ponto:** Calcular uma representação de valores de pontos para o polinômio  $C(x) = A(x)B(x)$  multiplicando esses valores ponto a ponto. Essa representação contém o valor de  $C(x)$  em cada raiz  $(2n)$ -ésima da unidade.

4. *Interpolar.* Criar a representação de coeficientes do polinômio  $C(x)$  através de uma única aplicação de uma FFT sobre  $2n$  pares de valores de pontos para calcular a DFT inversa.

Os passos (1) e (3) demoram o tempo  $\Theta(n)$ , e os passos (2) e (4) demoram o tempo  $\Theta(n \lg n)$ . Desse modo, uma vez que mostrarmos como usar a FFT, teremos demonstrado o seguinte.

### **Teorema 30.2**

O produto de dois polinômios de limite de grau  $n$  pode ser calculado no tempo  $\Theta(n \lg n)$ , com ambas as representações de entrada e saída em forma de coeficientes. ■

## **Exercícios**

### **30.1-1**

Multiplique os polinômios  $A(x) = 7x^3 - x^2 + x - 10$  e  $B(x) = 8x^3 - 6x + 3$ , usando as equações (30.1) e (30.2).

### **30.1-2**

A avaliação de um polinômio  $A(x)$  de limite de grau  $n$  em um dado ponto  $x_0$  também pode ser feita dividindo-se  $A(x)$  pelo polinômio  $(x - x_0)$  para obter um polinômio quociente  $q(x)$  de limite de grau  $n - 1$  e um resto  $r$ , tais que

$$A(x) = q(x)(x - x_0) + r.$$

Claramente,  $A(x_0) = r$ . Mostre como calcular o resto  $r$  e os coeficientes de  $q(x)$  no tempo  $\Theta(n)$  a partir de  $x_0$  e dos coeficientes de  $A$ .

### **30.1-3**

Derive uma representação de valores de pontos para  $A^{\text{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$  a partir de uma representação de valores de pontos para  $A(x) = \sum_{j=0}^{n-1} a_j x^j$ , supondo que nenhum dos pontos seja 0.

### **30.1-4**

Prove que  $n$  pares de valores de pontos distintos são necessários para especificar de forma exclusiva um polinômio de limite de grau  $n$ , ou seja, se menos de  $n$  pares de valores de pontos distintos são dados, eles deixam de especificar um único polinômio de limite de grau  $n$ . (*Sugestão:* Usando o Teorema 30.1, o que você poderia dizer sobre um conjunto de  $n - 1$  pares de valores de pontos ao qual é adicionado mais um par de valores de pontos escolhido arbitrariamente?)

### **30.1-5**

Mostre como usar a equação (30.5) para realizar a interpolação no tempo  $\Theta(n^2)$ . (*Sugestão:* Primeiro calcule a representação de coeficientes do polinômio [ver símbolo] $_{j}(x - x_j)$ , e depois divida por  $(x - x_k)$  conforme necessário para o numerador de cada termo. Veja o Exercício 30.1-2. Cada um dos  $n$  denominadores pode ser calculado no tempo  $O(n)$ .)

### **30.1-6**

Explique o que está errado na abordagem “óbvia” para divisão de polinômios, usando uma representação de valores de pontos, isto é, dividindo os valores  $y$  correspondentes. Discuta separadamente o caso no qual a divisão tem resultado exato e o caso em que ela não o tem.

### **30.1-7**

Considere dois conjuntos  $A$  e  $B$ , cada um com  $n$  inteiros no intervalo de 0 até  $10n$ . Desejamos calcular a **soma cartesiana** de  $A$  e  $B$ , definida por

$$C = \{x + y : x \in A \text{ e } y \in B\}$$

Observe que os inteiros em  $C$  estão no intervalo de 0 até  $20n$ . Queremos encontrar os elementos de  $C$  e o número de vezes que cada elemento de  $C$  é percebido como uma soma de elementos em  $A$  e  $B$ . Mostre que o problema pode ser resolvido no tempo  $O(n \lg n)$ . (*Sugestão:* Represente  $A$  e  $B$  como polinômios de grau  $10n$  no máximo.)

## 30.2 A DFT e a FFT

Na Seção 30.1 afirmamos que, se usarmos raízes complexas da unidade, poderemos avaliar e interpolar polinômios no tempo  $\Theta(n \lg n)$ . Nesta seção, definiremos as raízes complexas da unidade e estudaremos suas propriedades, definiremos a DFT, e então mostraremos como a FFT calcula a DFT e sua inversa apenas no tempo  $\Theta(n \lg n)$ .

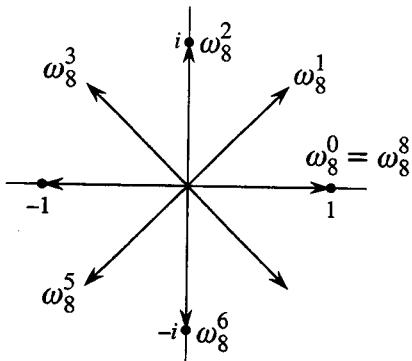


FIGURA 30.2 Os valores de  $\omega_8^0, \omega_8^1, \dots, \omega_8^7$  no plano complexo, onde  $\omega_8 = e^{2\pi i/8}$  é a raiz oitava principal da unidade

### Raízes complexas da unidade

Uma **raiz  $n$ -ésima complexa da unidade** é um número complexo  $\omega$  tal que

$$\omega^n = 1.$$

Existem exatamente  $n$  raízes  $n$ -ésimas complexas da unidade; essas são  $e^{2\pi i k/n}$  para  $k = 0, 1, \dots, n-1$ . Para interpretar essa fórmula, usamos a definição da exponencial de um número complexo:

$$e^{iu} = \cos(u) + i \sin(u).$$

A Figura 30.2 mostra que as  $n$  raízes complexas da unidade estão igualmente espaçadas em torno do círculo de raio unitário com centro na origem do plano complexo. O valor

$$\omega_n = e^{2\pi i/n} \tag{30.6}$$

é chamado **raiz  $n$ -ésima principal da unidade**; todas as outras raízes  $n$ -ésimas complexas da unidade são potências de  $\omega_n$ <sup>2</sup>

As  $n$  raízes  $n$ -ésimas complexas da unidade,

$$\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1},$$

formam um grupo sob a multiplicação (consulte a Seção 31.3). Esse grupo tem a mesma estrutura que o grupo aditivo  $(\mathbb{Z}_n, +)$  módulo  $n$ , pois  $\omega_n^n = \omega_n^0 = 1$  implica que  $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$ . De modo semelhante,  $\omega_n^{-1} = \omega_n^{n-1}$ . As propriedades essenciais das raízes  $n$ -ésimas complexas da unidade são dadas nos lemas a seguir.

<sup>2</sup> Muitos outros autores definem  $\omega_n$  de forma diferente:  $\omega_n = e^{2\pi i/n}$ . Essa definição alternativa tende a ser usada para aplicações de processamento de sinais. A matemática subjacente é substancialmente a mesma com uma ou outra definição de  $\omega_n$ .

**Lema 30.3 (Lema de cancelamento)**

Para quaisquer inteiros  $n \geq 0$ ,  $k \geq 0$  e  $d > 0$ ,

$$\omega_{dn}^{dk} = \omega_n^k. \quad (30.7)$$

**Prova** O lema decorre diretamente da equação (30.6), pois

$$\begin{aligned}\omega_{dn}^{dk} &= (e^{2\pi i/dn})^{dk} \\ &= (e^{2\pi i/n})^k \\ &= \omega_n^k.\end{aligned}$$

**Corolário 30.4**

Para qualquer inteiro par  $n > 0$ ,

$$\omega_n^{n/2} = \omega_2 = -1.$$

**Prova** A prova fica para o Exercício 30.2-1.

**Lema 30.5 (Lema da divisão em metades)**

Se  $n > 0$  é par, então os quadrados das  $n$  raízes  $n$ -ésimas complexas da unidade são as  $n/2$  raízes  $(n/2)$ -ésimas complexas da unidade.

**Prova** Pelo lema de cancelamento, temos  $(\omega_n^k)^2 = \omega_{n/2}^k$  para qualquer inteiro não negativo  $k$ . Observe que, se elevarmos ao quadrado todas as raízes  $n$ -ésimas complexas da unidade, então cada raiz  $(n/2)$ -ésima da unidade será obtida exatamente duas vezes, pois

$$(\omega_n^{k+n/2})^2 = \omega_n^{2k+n}$$

$$= \omega_n^{2k} \omega_n^n$$

$$= (\omega_n^k)^2.$$

Desse modo,  $\omega_n^k$  e  $\omega_n^{k+n/2}$  têm o mesmo quadrado. Essa propriedade também pode ser provada usando-se o Corolário 30.4, pois  $\omega_n^{n/2} = -1$  implica  $\omega_n^{k+n/2} = -\omega_n^k$  e, portanto,  $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$ .

Como veremos, o lema da divisão em metades é essencial para nossa abordagem de dividir e conquistar, por realizar a conversão entre representações de coeficientes e valores de pontos de polinômios, pois ele garante que os subproblemas recursivos terão somente metade da extensão.

**Lema 30.6 (Lema do somatório)**

Para qualquer inteiro  $n \geq 1$  e inteiro não negativo  $k$  não divisível por  $n$ ,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.$$

**Prova** A equação (A.5) se aplica a valores complexos, bem como a reais, e então temos

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1}$$

$$= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1}$$

$$= \frac{(1)^k - 1}{\omega_n^k - 1}$$

$$= 0.$$

Exigir que  $k$  não seja divisível por  $n$  assegura que o denominador não é 0, pois  $\omega_n^k$  somente quando  $k$  é divisível por  $n$ . ■

## A DFT

Vamos lembrar que desejamos avaliar um polinômio

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

de limite de grau  $n$  em  $\omega_n^0, \omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}$  (isto é, nas  $n$  raízes  $n$ -ésimas complexas da unidade).<sup>3</sup> Sem perda de generalidade, supomos que  $n$  é uma potência de 2, pois um dado limite de grau sempre pode ser elevado – sempre podemos adicionar novos coeficientes zero de alta ordem conforme necessário.<sup>4</sup> Assim, presumimos que  $A$  é dado em forma de coeficientes:  $a = (a_0, a_1, \dots, a_{n-1})$ . Vamos então definir os resultados  $y_k$ , para  $k = 0, 1, \dots, n-1$ , por

$$y_k = A(\omega_n^k)$$

$$= \sum_{j=0}^{n-1} a_j \omega_n^{kj}. \quad (30.8)$$

O vetor  $y = (y_0, y_1, \dots, y_{n-1})$  é a **transformação discreta de Fourier (DFT – Discrete Fourier Transform)** do vetor de coeficientes  $a = (a_0, a_1, \dots, a_{n-1})$ . Também escrevemos  $y = \text{DFT}_n(a)$ .

---

<sup>3</sup> O comprimento  $n$  é de fato o que chamamos  $2n$  na Seção 30.1, pois duplicamos o limite de grau dos polinômios dados antes da avaliação. Então, no contexto da multiplicação de polinômios, estamos trabalhando realmente com raízes ( $2n$ )-ésimas complexas da unidade.

<sup>4</sup> Quando usando a FFT para processamento de sinais, o preenchimento com coeficientes zero para chegar a um tamanho de potência de 2 em geral é desaconselhável, pois tende a introduzir artefatos de alta frequência. Uma técnica usada para preencher até o tamanho de uma potência de 2 em processamento de sinais é o **espelhamento**. Sendo  $n'$  a menor potência inteira de 2 maior que  $n$ , um modo de espelhar define  $a_{n+j} = a_{n-j-2}$  para  $j = 0, 1, \dots, n'-n-1$ .

## A FFT

Usando um método conhecido como *transformação rápida de Fourier (FFT – Fast Fourier Transform)*, que tira proveito das propriedades especiais das raízes complexas da unidade, podemos calcular  $DFT_n(a)$  no tempo  $\Theta(n \lg n)$ , em lugar do tempo  $\Theta(n^2)$  do método direto.

O método da FFT utiliza uma estratégia de dividir e conquistar, empregando os coeficientes de índice par e os coeficientes de índice ímpar de  $A(x)$  separadamente para definir os dois novos polinômios de limite de grau  $n/2$   $A^{[0]}(x)$  e  $A^{[1]}(x)$ :

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1},$$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}.$$

Observe que  $A^{[0]}$  contém todos os coeficientes de índice par de  $A$  (a representação binária do índice termina em 0) e  $A^{[1]}$  contém todos os coeficientes de índice ímpar (a representação binária do índice termina em 1). Segue-se que

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2), \quad (30.9)$$

de forma que o problema de avaliar  $A(x)$  em  $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$  se reduz a

1. avaliar os polinômios de limite de grau  $n/2$   $A^{[0]}(x)$  e  $A^{[1]}(x)$  nos pontos

$$(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2, \quad (30.10)$$

e então

2. combinar os resultados de acordo com equação (30.9).

Pelo lema da divisão em metades, a lista de valores (30.10) não consiste em  $n$  valores distintos, mas somente nas  $n/2$  raízes ( $n/2$ )-ésimas complexas da unidade, com cada raiz ocorrendo exatamente duas vezes. Assim, os polinômios  $A^{[0]}$  e  $A^{[1]}$  de limite de grau  $n/2$  são avaliados recursivamente nas  $n/2$  raízes ( $n/2$ )-ésimas complexas da unidade. Esses subproblemas apresentam exatamente a mesma forma do problema original, mas têm metade do tamanho. Agora, dividimos com sucesso o cálculo de uma  $DFT_n$  de  $n$  elementos em dois cálculos de  $DFT_{n/2}$  de  $n/2$  elementos. Essa decomposição é a base para o algoritmo da FFT recursiva a seguir, que calcula a DFT de um vetor de  $n$  elementos  $a = (a_0, a_1, \dots, a_{n-1})$ , onde  $n$  é uma potência de 2.

**RECURSIVE-FFT( $a$ )**

```

1  $n \leftarrow \text{comprimento}[a]$   $\triangleright n$  é uma potência de 2.
2 if  $n = 1$ 
3   then return  $a$ 
4  $\omega n \leftarrow e^{2\pi i/n}$ 
5  $\omega \leftarrow 1$ 
6  $a^{[0]} \leftarrow (a_0, a_2, \dots, a_{n-2})$ 
7  $a^{[1]} \leftarrow (a_1, a_3, \dots, a_{n-1})$ 
8  $y^{[0]} \leftarrow \text{RECURSIVE-FFT}(a^{[0]})$ 
9  $y^{[1]} \leftarrow \text{RECURSIVE-FFT}(a^{[1]})$ 
10 for  $k \leftarrow 0$  to  $n/2 - 1$ 
11   do  $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$ 
12    $y_{k+(n/2)} \leftarrow y_k^{[0]} + \omega y_k^{[1]}$ 
13    $\omega \leftarrow \omega \omega n$ 
14 return  $y$        $\triangleright y$  é considerado um vetor coluna.

```

O procedimento RECURSIVE-FFT funciona da maneira descrita a seguir. As linhas 2 e 3 representam a base da recursão; a DFT de um elemento é o próprio elemento pois, nesse caso,

$$y_0 = \alpha_0 \omega_1^0$$

$$= \alpha_0 \cdot 1$$

$$= \alpha_0.$$

As linhas 6 e 7 definem os vetores de coeficientes para os polinômios  $A^{[0]}$  e  $A^{[1]}$ . As linhas 4, 5 e 13 garantem que  $\omega$  será atualizado corretamente, de tal forma que, sempre que as linhas 11 e 12 forem executadas,  $\omega = \omega_n^k$ . (Manter um valor contínuo  $\omega$  de uma iteração até outra poupa tempo em relação ao cálculo de  $\omega_n^k$  desde o início a cada passagem pelo loop **for**.) As linhas 8 e 9 executam os  $DFT_{n/2}$  cálculos recursivos, configurando, para  $k = 0, 1, \dots, n/2 - 1$ ,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k), \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k), \end{aligned}$$

ou, tendo em vista que  $\omega_{n/2}^k = \omega_n^{2k}$  pelo lema de cancelamento,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_n^{2k}), \\ y_k^{[1]} &= A^{[1]}(\omega_n^{2k}). \end{aligned}$$

As linhas 11 e 12 combinam os resultados dos  $DFT_{n/2}$  cálculos recursivos. Para  $y_0, y_1, \dots, y_{n/2-1}$ , a linha 11 produz

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \quad (\text{pela equação (30.9)}). \end{aligned}$$

Para  $y_{n/2}, y_{n/2+1}, \dots, y_{n-1}$ , fazendo-se  $k = 0, 1, \dots, n/2 - 1$ , a linha 12 produz

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} - \omega_n^{k+(n/2)} y_k^{[1]} \quad (\text{pois } \omega_n^{k+(n/2)} = -\omega_n^k) \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \quad (\text{pois } \omega_n^{2k+n} = \omega_n^{2k}) \\ &= A(\omega_n^{k+(n/2)}) \quad (\text{pela equação 30.9})). \end{aligned}$$

Desse modo, o vetor  $y$  retornado por RECURSIVE-FFT é de fato a DFT do vetor de entrada  $\alpha$ .

Dentro do loop **for** das linhas 10 a 13, cada valor  $y_k^{[1]}$  é multiplicado por  $\omega_n^k$ , para  $k = 0, 1, \dots, n/2 - 1$ . O produto é adicionado e subtraído de  $y_k^{[0]}$ . Como cada fator  $\omega_n^k$  é usado tanto em sua forma positiva quanto negativa, os fatores  $\omega_n^k$  são conhecidos como *fatores de giro*.

Para determinar o tempo de execução do procedimento RECURSIVE-FFT, observamos que, com exceção das chamadas recursivas, cada invocação demora o tempo  $\Theta(n)$ , onde  $n$  é o comprimento do vetor de entrada. A recorrência para o tempo de execução é então

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

$$= \Theta(n \lg n).$$

Desse modo, podemos avaliar um polinômio de limite de grau  $n$  nas raízes  $n$ -ésimas complexas da unidade no tempo  $\Theta(n \lg n)$ , usando a transformação rápida de Fourier.

## Interpolação nas raízes complexas da unidade

Agora completamos o esquema de multiplicação de polinômios, mostrando como interpolar as raízes complexas da unidade por um polinômio, o que nos permite realizar a conversão de volta da forma de valores de pontos para a forma de coeficientes. Fazemos a interpolação escrevendo a DFT como uma equação matricial e então observando a forma da inversa da matriz.

A partir da equação (30.4), podemos escrever a DFT como o produto matricial  $y = V_n \alpha$ , onde  $V_n$  é uma matriz de Vandermonde que contém as potências apropriadas de  $\omega_n$ :

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \vdots \\ \alpha_{n-1} \end{pmatrix}.$$

A entrada  $(k, j)$  de  $V_n$  é  $\omega_n^{kj}$  para  $j = 0, 1, \dots, n-1$ , e os expoentes das entradas de  $V_n$  formam uma tabela de multiplicação.

Para a operação inversa, que escrevemos como  $\alpha = \text{DFT}_n^{-1}(y)$ , prosseguimos multiplicando  $y$  pela matriz  $V_n^{-1}$ , a inversa de  $V_n$ .

### Teorema 30.7

Para  $j, k = 0, 1, \dots, n-1$ , a entrada  $(j, k)$  de  $V_n^{-1}$  é  $\omega_n^{-kj}$ .

**Prova** Mostramos que  $V_n = I_n$ , a matriz identidade  $n \times n$ . Considere a entrada  $(j, j')$  de  $V_n^{-1}V_n$ :

$$\begin{aligned} [V_n^{-1}V_n]_{jj'} &= \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) \\ &= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n. \end{aligned}$$

Esse somatório é igual a 1 se  $j' = j$ , e é 0 caso contrário, de acordo com o lema de somatório (Lema 30.6). Observe que nos baseamos em  $-(n-1) < j' - j < n-1$ , de modo que  $j' - j$  não é divisível por  $n$ , para que o lema de somatório se aplique. ■

Dada a matriz inversa  $V_n^{-1}$ , temos que  $\text{DFT}_n^{-1}(y)$  é dada por

$$\alpha_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{30.11}$$

para  $j = 0, 1, \dots, n-1$ . Comparando as equações (30.8) e (30.11), vemos que a modificação do algoritmo da FFT para trocar os papéis de  $\alpha$  e  $y$ , substituir  $\omega n$  por  $\omega_n^{-1}$  e dividir cada elemento do resultado por  $n$ , nos permite calcular a DFT inversa (ver Exercício 30.2-4). Desse modo,  $\text{DFT}_n^{-1}$  também pode ser calculada no tempo  $\Theta(n \lg n)$ .

Assim, usando a FFT e a FFT inversa, podemos transformar um polinômio de limite de grau  $n$  entre sua representação de coeficientes e uma representação de valores de pontos no tempo  $\Theta(n \lg n)$ . No contexto da multiplicação de polinômios, mostramos o seguinte.

### **Teorema 30.8 (Teorema de convolução)**

Para dois vetores quaisquer  $a$  e  $b$  de comprimento  $n$ , onde  $n$  é uma potência de 2,

$$a \otimes b = \text{DFT}_{2n}^{-1} \left( (\text{DFT}_{2n}(a) \cdot \text{DFT}_{2n}(b)) \right),$$

onde os vetores  $a$  e  $b$  são preenchidos com 0 até o comprimento  $2n$ , e  $\cdot$  denota o produto em nível de componentes de dois vetores de  $2n$  elementos. ■

## **Exercícios**

### **30.2-1**

Prove o Corolário 30.4.

### **30.2-2**

Calcule a DFT do vetor  $(0, 1, 2, 3)$ .

### **30.2-3**

Faça o Exercício 30.1-1 usando o esquema de tempo  $\Theta(n \lg n)$ .

### **30.2-4**

Escreva pseudocódigo para calcular  $\text{DFT}_n^{-1}$  no tempo  $\Theta(n \lg n)$ .

### **30.2-5**

Descreva a generalização do procedimento FFT para o caso em que  $n$  é uma potência de 3. Forneça uma recorrência para o tempo de execução, e resolva a recorrência.

### **30.2-6 \***

Suponha que em vez de executar uma FFT de  $n$  elementos sobre o campo de números complexos (onde  $n$  é par), usamos o anel  $\mathbb{Z}_m$  de inteiros módulo  $m$ , onde  $m = 2^{tn/2}$  e  $t$  é um inteiro positivo arbitrário. Utilize  $\omega = 2_t$  em lugar de  $\omega_n$  como raiz  $n$ -ésima principal da unidade, módulo  $m$ . Prove que a DFT e a DFT inversa estão bem definidas nesse sistema.

### **30.2-7**

Dada uma lista de valores  $z_0, z_1, \dots, z_{n-1}$  (possivelmente com repetições), mostre como encontrar os coeficientes do polinômio  $P(x)$  de limite de grau  $n$  que tem zeros apenas em  $z_0, z_1, \dots, z_{n-1}$  (possivelmente com repetições). Seu procedimento deve ser executado no tempo  $O(n \lg^2 n)$ . (Sugestão: O polinômio  $P(x)$  tem um zero em  $z_j$  se e somente se  $P(x)$  é um múltiplo de  $(x - z_j)$ .)

### **30.2-8 \***

A *transformação de trinado* de um vetor  $a = (a_0, a_1, \dots, a_{n-1})$  é o vetor  $y = (y_0, y_1, \dots, y_{n-1})$ , onde  $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$  e  $z$  é qualquer número complexo. Assim, a DFT é um caso especial da transformação de trinado, obtido tomando-se  $z = \omega_n$ . Prove que a transformação de trinado pode ser avaliada no tempo  $O(n \lg n)$  para qualquer número complexo  $z$ . (Sugestão: Use a equação

$$y_k = 2^{k^2/2} \sum_{j=0}^{n-1} (a_j z^{j^2/2}) (z^{-(k-j)^2/2})$$

para ver a transformação de trinado como um convolução.)

## **30.3 Implementações eficientes de FFT**

Considerando-se que as aplicações práticas da DFT, como o processamento de sinais, exigem o máximo em velocidade, esta seção examina duas implementações eficientes de FFT. Primeiro, vamos examinar uma versão iterativa do algoritmo da FFT que é executada no tempo  $\Theta(n \lg n)$ , mas que tem uma constante oculta mais baixa na notação de  $\Theta$  que a implementação recursiva da Seção 30.2. Em seguida, usaremos as idéias que nos levaram à implementação iterativa para projetar um circuito paralelo eficiente de FFT.

## Uma implementação iterativa de FFT

Observamos em primeiro lugar que o loop **for** das linhas 10 a 13 de RECURSIVE-FFT envolve o cálculo do valor  $\omega_n^k y_k^{[1]}$  duas vezes. Na terminologia dos compiladores, esse valor é conhecido como uma **subexpressão comum**. Podemos mudar o loop para calculá-lo apenas uma vez, armazenando-o em uma variável temporária  $t$ .

```

for  $k \leftarrow 0$  to  $n/2 - 1$ 
  do  $t \leftarrow \omega_n^k y_k^{[1]}$ 
     $y_k \leftarrow y_k^{[0]} + t$ 
     $y_{k+(n/2)} \leftarrow y_k^{[0]} - t$ 
     $\omega \leftarrow \omega \omega_n$ 
  
```

A operação nesse loop, multiplicando o fato de giro  $\omega = \omega_n^k$  por  $y_k^{[1]}$ , armazenando o produto em  $t$ , e adicionando e subtraindo  $t$  de  $y_k^{[0]}$ , é conhecida como uma **operação de borboleta** e é mostrada esquematicamente na Figura 30.3.

Agora mostramos como tornar o algoritmo FFT iterativo, em vez de ter uma estrutura recursiva. Na Figura 30.4, organizamos os vetores de entrada para as chamadas recursivas em uma invocação de RECURSIVE-FFT em uma estrutura de árvore, onde a chamada inicial é para  $n = 8$ . A árvore tem um nó para cada chamada do procedimento, identificado pelo vetor de entrada correspondente. Cada invocação de RECURSIVE-FFT faz duas chamadas recursivas, a menos que tenha recebido um vetor de 1 elemento. Fazemos a primeira chamada ao filho esquerdo e a segunda chamada ao filho direito.

Examinando a árvore, observamos que, se pudéssemos organizar os elementos do vetor inicial  $a$  na ordem em que eles aparecem nas folhas, poderíamos imitar a execução do procedimento RECURSIVE-FFT da maneira ilustrada a seguir. Primeiro, tomamos os elementos aos pares, calculamos a DFT de cada par usando uma operação de borboleta e substituímos o par por sua DFT. Em seguida, o vetor contém as  $n/2$  DFTs de 2 elementos. Então, tomamos essas  $n/2$  DFTs aos pares e calculamos a DFT dos quatro elementos de vetores de onde elas vieram, executando duas operações de borboleta, substituindo duas DFTs de 2 elementos por uma DFT de 4 elementos. Assim, o vetor contém  $n/4$  DFTs de 4 elementos. Continuamos dessa maneira até o vetor conter duas DFTs de  $(n/2)$  elementos, as quais podem ser combinadas com o uso de  $n/2$  operações de borboleta na DFT final de  $n$  elementos.

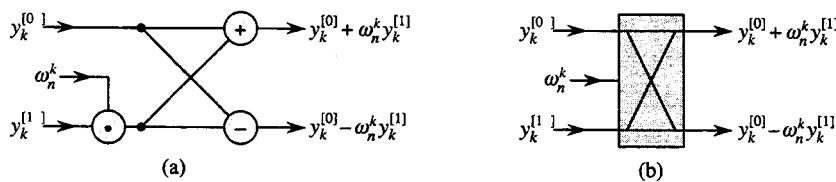


FIGURA 30.3 Uma operação de borboleta. (a) Os dois valores de entrada são introduzidos a partir da esquerda, o fator de giro  $\omega_n^k$  é multiplicado por  $y_k^{[1]}$ , e a soma e a diferença são fornecidas como saída à direita. (b) Um desenho simplificado de uma operação borboleta. Usaremos essa representação em um circuito paralelo de FFT.

Para transformar essa observação em código, usamos um arranjo  $A[0..n-1]$  que contém inicialmente os elementos do vetor de entrada  $a$  na ordem em que eles aparecem nas folhas da árvore da Figura 30.4. (Mostraremos adiante como determinar essa ordem, conhecida como uma permutação com inversão de bits.) Pelo fato de a combinação ter de ser feita em cada nível da árvore, introduzimos uma variável  $s$  para contar os níveis, variando desde 1 (na parte inferior, quando estamos combinando pares para formar DFTs de 2 elementos) até  $\lg n$  (na parte superior, quando estamos combinando duas DFTs de  $(n/2)$  elementos para produzir o resultado final). O algoritmo tem então a seguinte estrutura:

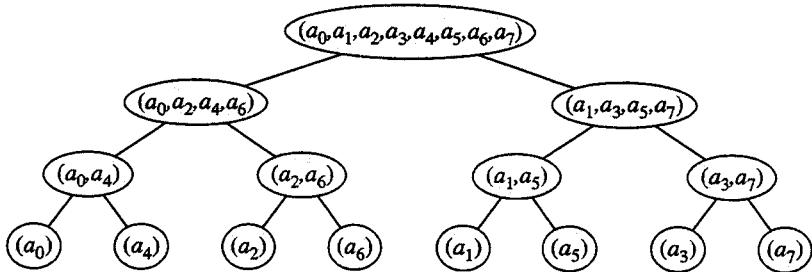


FIGURA 30.4 A árvore de vetores de entrada para as chamadas recursivas do procedimento RECURSIVE-FFT. A invocação inicial é para  $n = 8$

```

1 for  $s \leftarrow 1$  to  $\lg n$ 
2 do for  $k \leftarrow 0$  to  $n - 1$  by  $2^s$ 
3   do combinar as duas DFTs de  $2^{s-1}$  elementos em
       $A[k..k + 2^{s-1} - 1]$  e  $A[k + 2^{s-1}..k + 2^s - 1]$ 
      em uma DFT de  $2^s$  elementos em  $A[k .. k + 2^s - 1]$ 

```

Podemos expressar o corpo do loop (linha 3) como um pseudocódigo mais preciso. Copiamos o loop **for** do procedimento RECURSIVE-FFT, identificando  $y^{[0]}$  com  $A[k .. k + 2^{s-1} - 1]$  e  $y^{[1]}$  com  $A[k + 2^{s-1} .. k + 2^s - 1]$ . O valor de giro usado em cada operação de borboleta depende do valor de  $s$ ; ele é uma potência de  $\omega_m$ , onde  $m = 2^s$ . (Introduzimos a variável  $m$  unicamente para fins de legibilidade.) Introduzimos outra variável temporária  $u$  que nos permite executar a operação de borboleta no local. Quando substituímos a linha 3 da estrutura global pelo corpo do loop, obtemos o pseudocódigo a seguir, que forma a base da implementação paralela que apresentaremos mais adiante. Primeiro, o código chama o procedimento auxiliar BIT-REVERSE-COPY para copiar o vetor  $a$  no arranjo  $A$ , na ordem inicial em que precisamos dos valores.

#### ITERATIVE-FFT( $a$ )

```

1 BIT-REVERSE-COPY( $a, A$ )
2  $n \leftarrow \text{comprimento}[a]$      $\triangleright n$  é uma potência de 2.
3 for  $s \leftarrow 1$  to  $\lg n$ 
4   do  $m \leftarrow 2^s$ 
5      $\omega \leftarrow e^{2\pi i/m}$ 
6     for  $k \leftarrow 0$  to  $n - 1$  by  $m$ 
7       do  $w \leftarrow 1$ 
8         for  $j \leftarrow 0$  to  $m/2 - 1$ 
9           do  $t \leftarrow \omega A[k + j + m/2] \cdot$ 
10             $u \leftarrow A[k + j]$ 
11             $A[k + j] \leftarrow u + t$ 
12             $A[k + j + m/2] \leftarrow u - t$ 
13            $\omega \leftarrow \omega \omega_m$ 

```

De que modo BIT-REVERSE-COPY obtém os elementos do vetor de entrada  $a$  na ordem desejada no arranjo  $A$ ? A ordem em que as folhas aparecem na Figura 30.4 é uma **permutação com inversão de bits**. Isto é, se fizermos de  $\text{rev}(k)$  o inteiro de  $\lg n$  bits formado pela inversão dos bits da representação binária de  $k$ , então queremos inserir o elemento de vetor  $a_k$  na posição de arranjo  $A[\text{rev}(k)]$ . Por exemplo, na Figura 30.4, as folhas aparecem na ordem 0, 4, 2, 6, 1, 5, 3, 7; em binário, essa seqüência é 000, 100, 010, 110, 001, 101, 011, 111 e, em binário com inversão de bits, obtemos a seqüência 000, 001, 010, 011, 100, 101, 110, 111. Para ver que, em geral, queremos a ordem binária com inversão de bits, observamos que no nível superior da árvore, os índices cujo bit de baixa ordem é 0 são inseridos na subárvore da esquerda, e os índices cujo bit de baixa ordem é 1 são inseridos na subárvore da direita. Extrairindo o bit de baixa ordem

em cada nível, continuamos com esse processo descendo a árvore, até obtermos a ordem binária com inversão de bits nas folhas.

Como a função  $\text{rev}(k)$  é facilmente calculada, o procedimento BIT-REVERSE-COPY pode ser escrito como a seguir.

```
BIT-REVERSE-COPY( $a, A$ )
1  $n \leftarrow \text{comprimento}[a]$ 
2 for  $k \leftarrow 0$  to  $n - 1$ 
3 do  $A[\text{rev}(k)] \leftarrow a_k$ 
```

A implementação iterativa de FFT é executada no tempo  $\Theta(n \lg n)$ . A chamada a BIT-REVERSE-COPY( $a, A$ ) certamente é executada no tempo  $O(n \lg n)$ , pois fazemos a iteração  $n$  vezes e podemos inverter um inteiro entre 0 e  $n - 1$ , com  $\lg n$  bits, no tempo  $O(\lg n)$ . (Na prática, em geral conhecemos o valor inicial de  $n$  com antecedência; assim, provavelmente codificariam um mapeamento de tabela  $k$  para  $\text{rev}(k)$ , fazendo BIT-REVERSE-COPY ser executado no tempo  $\Theta(n)$  com uma constante oculta baixa. Outra alternativa é usar o inteligente esquema de contador binário reverso amortizado descrito no Problema 17-1.) Para completar a prova de que ITERATIVE-FFT é executado no tempo  $\Theta(n \lg n)$ , mostramos que  $L(n)$ , o número de vezes que o corpo do loop mais interno (linhas 8 a 13) é executado, é  $\Theta(n \lg n)$ . O loop **for** das linhas 6 a 13 itera  $n/m = n/2^s$  vezes para cada valor de  $s$ , e o loop mais interno das linhas 8 a 13 itera  $m/2 = 2^{s-1}$  vezes. Desse modo,

$$\begin{aligned} L(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\ &= \sum_{s=1}^{\lg n} \frac{n}{2} \\ &= \Theta(n \lg n). \end{aligned}$$

## Um circuito paralelo de FFT

Podemos explorar muitas das propriedades que nos permitiram implementar um algoritmo iterativo eficiente de FFT para produzir um algoritmo paralelo eficiente para a FFT. Expressaremos o algoritmo paralelo de FFT como um circuito de aparência muito semelhante às redes de comparação do Capítulo 27. Em vez de comparadores, o circuito de FFT utiliza operações de borboleta, como mostra a Figura 30.3(b). A noção de profundidade que desenvolvemos no Capítulo 27 também se aplica aqui. O circuito PARALLEL-FFT que calcula a FFT sobre  $n$  entradas é mostrado na Figura 30.5 para  $n = 8$ . O circuito começa com uma permutação com inversão de bits das entradas, seguida por  $\lg n$  estágios, cada um consistindo em  $n/2$  operações de borboleta executadas em paralelo. A profundidade do circuito é então  $\Theta(\lg n)$ .

A parte mais à esquerda do circuito PARALLEL-FFT executa a permutação com inversão de bits, e o restante imita o procedimento iterativo ITERATIVE-FFT. Aproveitamos o fato de que cada iteração do loop **for** mais externo executa  $n/2$  operações de borboleta independentes que podem ser executadas em paralelo. O valor de  $s$  em cada iteração dentro de ITERATIVE-FFT corresponde a um estágio de operações de borboleta, como mostra a Figura 30.5. Dentro do estágio  $s$ , para  $s = 1, 2, \dots, \lg n$ , existem  $n/2^s$  grupos de borboletas (correspondendo a cada valor de  $k$  em ITERATIVE-FFT), com  $2^{s-1}$  borboletas por grupo (correspondendo a cada valor de  $j$  em ITERATIVE-FFT). As operações de borboleta mostradas na Figura 30.5 correspondem às operações de borboleta do loop mais interno (linhas 9 a 12 de ITERATIVE-FFT). Observe também que os fatores de giro usados nas operações de borboleta correspondem aos que são utilizados em ITERATIVE-FFT: no estágio  $s$ , usamos  $\omega_m^0, \omega_m^1, \dots, \omega_m^{m/2-1}$ , onde  $m = 2^s$ .

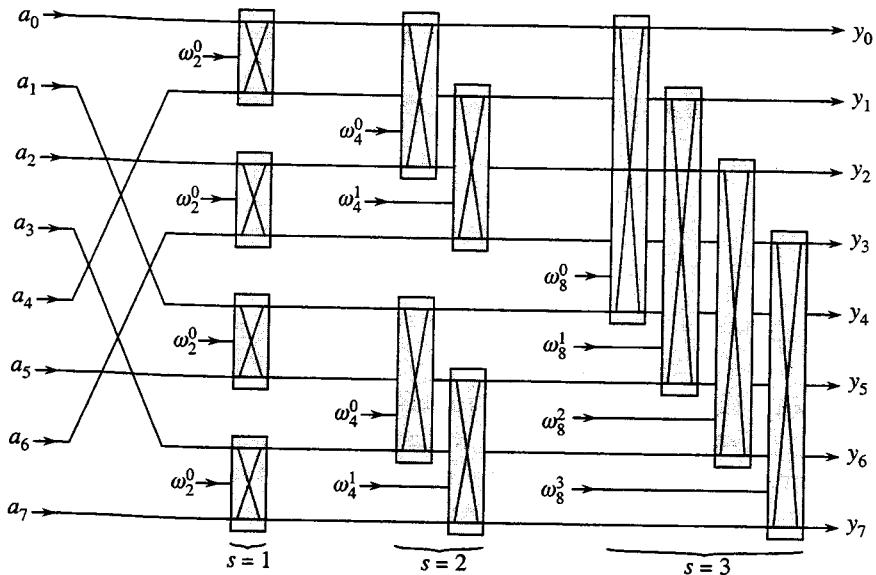


FIGURA 30.5 Um circuito PARALLEL-FFT que calcula a FFT, mostrada aqui em  $n = 8$  entradas. Cada operação de borboleta toma como entrada os valores em dois fios, juntamente com um fator de giro, e produz como saídas os valores em dois fios. Os estágios de operações de borboleta são identificados de modo a corresponder a iterações do loop externo do procedimento ITERATIVE-FFT. Somente os fios superior e inferior que passam por uma borboleta interagem com ela; fios que passam pelo meio de uma borboleta não afetam essa borboleta, nem seus valores são alterados por essa borboleta. Por exemplo, a borboleta superior no estágio 2 não tem nenhuma relação com o fio 1 (o fio cuja saída é identificada por  $y_1$ ); suas entradas e saídas estão apenas nos fios 0 e 2 (identificadas por  $y_0$  e  $y_2$ , respectivamente). Uma FFT sobre  $n$  entradas pode ser calculada na profundidade  $\Theta(\lg n)$  com  $\Theta(n \lg n)$  operações de borboleta.

## Exercícios

### 30.3-1

Mostre como ITERATIVE-FFT calcula a DFT do vetor de entrada  $(0, 2, 3, -1, 4, 5, 7, 9)$ .

### 30.3-2

Mostre como implementar um algoritmo de FFT com a permutação de inversão de bits ocorrendo no final, e não no início do cálculo. (Sugestão: Considere a DFT inversa.)

### 30.3-3

Quantas vezes ITERATIVE-FFT calcula fatores de giro em cada fase? Reescreva ITERATIVE-FFT para calcular fatores de giro somente  $2^{s-1}$  vezes na fase  $s$ .

### 30.3-4 \*

Suponha que os somadores dentro das operações de borboleta no circuito FFT às vezes falham de tal maneira que sempre produzem uma saída zero, independente de suas entradas. Suponha que exatamente um somador tenha falhado, mas que você não saiba qual. Descreva como é possível identificar o somador que falhou fornecendo entradas para o circuito FFT global e observando as saídas. Qual é a eficiência do seu método?

## Problemas

### 30-1 Multiplicação de dividir e conquistar

- a. Mostre como multiplicar dois polinômios lineares  $ax + b$  e  $cx + d$  usando apenas três multiplicações. (Sugestão: Uma das multiplicações é  $(a + b) \cdot (c + d)$ .)

- b. Forneça dois algoritmos de dividir e conquistar para multiplicar dois polinômios de limite de grau  $n$  que sejam executados no tempo  $\Theta(n \lg^3)$ . O primeiro algoritmo deve dividir os coeficientes dos polinômios de entrada em uma metade alta e uma metade baixa, e o segundo algoritmo deve dividi-los de acordo com o fato do índice ser ímpar ou par.
- c. Mostre que dois inteiros de  $n$  bits podem ser multiplicados em  $\Theta(n \lg^3)$  passos, onde cada passo opera sobre no máximo um número constante de valores de 1 bit.

### 30-2 Matrizes de Toeplitz

Uma **matriz de Toeplitz** é uma matriz  $n \times n$  matrix  $A = (a_{ij})$  tal que  $a_{ij} = a_{i-1,j-1}$  para  $i = 2, 3, \dots, n$  e  $j = 2, 3, \dots, n$ .

- a. A soma de duas matrizes de Toeplitz é necessariamente de Toeplitz? E o produto?
- b. Descreva como representar uma matriz de Toeplitz de forma que duas matrizes de Toeplitz  $n \times n$  possam ser somados no tempo  $O(n)$ .
- c. Forneça um algoritmo de tempo  $O(n \lg n)$  para multiplicar uma matriz de Toeplitz  $n \times n$  por um vetor de comprimento  $n$ . Use sua representação da parte (b).
- d. Forneça um algoritmo eficiente para multiplicar duas matrizes de Toeplitz  $n \times n$ . Analise seu tempo de execução.

### 30-3 Transformação rápida de Fourier multidimensional

Podemos generalizar a transformação discreta de Fourier unidimensional definida pela equação (30.8) para  $d$  dimensões. Nossa entrada é um arranjo  $d$ -dimensional  $A = (a_{i_1, i_2, \dots, i_d})$  cujas dimensões são  $n_1, n_2, \dots, n_d$ , onde  $n_1 n_2 \cdots n_d = n$ . Definimos a transformação discreta de Fourier  $d$ -dimensional pela equação

$$y_{k_1, k_2, \dots, k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1, j_2, \dots, j_d} \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d}$$

para  $0 \leq k_1 < n_1, 0 \leq k_2 < n_2, \dots, 0 \leq k_d < n_d$ .

- a. Mostre que podemos calcular uma DFT  $d$ -dimensional calculando DFTs unidimensionais em cada dimensão por sua vez. Isto é, primeiro calcule  $n/n_1$  DFTs unidimensionais separadas ao longo da dimensão 1. Em seguida, usando o resultado das DFTs ao longo da dimensão 1 como entrada, calcule  $n/n_2$  DFTs unidimensionais separadas ao longo da dimensão 2. Usando esse resultado como a entrada, calcule  $n/n_3$  DFTs unidimensionais separadas ao longo da dimensão 3 e assim por diante, até a dimensão  $d$ .
- b. Mostre que a ordenação de dimensões não importa, e assim podemos calcular uma DFT  $d$ -dimensional calculando as DFTs unidimensionais em qualquer ordem das  $d$  dimensões.
- c. Mostre que, se calcularmos cada DFT unidimensional calculando a transformação rápida de Fourier, o tempo total para calcular uma DFT  $d$ -dimensional é  $O(n \lg n)$ , independente de  $d$ .

### 30-4 Avaliando todas as derivadas de um polinômio em um ponto

Dado um polinômio  $A(x)$  de limite de grau  $n$ , sua  $t$ -ésima derivada é definida por

$$A^{(t)}(x) = \begin{cases} A(x) & \text{se } t = 0, \\ \frac{d}{dx} A^{(t-1)}(x) & \text{se } 1 \leq t \leq n-1, \\ 0 & \text{se } t \geq n. \end{cases}$$

A partir da representação de coeficientes  $(a_0, a_1, \dots, a_{n-1})$  de  $A(x)$  e de um dado ponto  $x_0$ , desejamos determinar  $A^{(t)}(x_0)$  para  $t = 0, 1, \dots, n-1$ .

- a. Dados os coeficientes  $b_0, b_1, \dots, b_{n-1}$  tais que

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j,$$

mostre como calcular  $A^{(t)}(x_0)$  para  $t = 0, 1, \dots, n - 1$ , no tempo  $O(n)$ .

- b.** Explique como encontrar  $b_0, b_1, \dots, b_{n-1}$  no tempo  $O(n \lg n)$ , dado  $A(x_0 + \omega_n^k)$  para  $k = 0, 1, \dots, n - 1$ .
- c.** Prove que

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left( \frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right),$$

onde  $f(j) = a_j \cdot j!$  e

$$g(l) = \begin{cases} x_0^{-1}/(-l)! & \text{se } -(n-1) \leq l \leq 0 \\ 0 & \text{se } 1 \leq l \leq (n-1). \end{cases}$$

- d.** Explique como avaliar  $A(x_0 + \omega_n^k)$  para  $k = 0, 1, \dots, n - 1$  no tempo  $O(n \lg n)$ . Conclua que todas as derivadas não triviais de  $A(x)$  podem ser avaliadas em  $x_0$  no tempo  $O(n \lg n)$ .

### 30-5 Avaliação de polinômios em vários pontos

Observamos que o problema de avaliar um polinômio de limite de grau  $n - 1$  em um ponto único pode ser resolvido no tempo  $O(n)$  usando-se a regra de Horner. Também descobrimos que tal polinômio pode ser avaliado em todas as  $n$  raízes complexas da unidade no tempo  $O(n \lg n)$  com a utilização da FFT. Mostraremos agora como avaliar um polinômio de limite de grau  $n$  em  $n$  pontos arbitrários no tempo  $O(n \lg^2 n)$ .

Para isso, usaremos o fato de que podemos calcular o resto do polinômio quando um desses polinômios é dividido por outro no tempo  $O(n \lg n)$ , um resultado que presumimos sem prova. Por exemplo, o resto de  $3x^3 + x^2 - 3x + 1$  quando dividido por  $x^2 + x + 2$  é

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = -7x + 5.$$

Dada a representação de coeficientes de um polinômio  $A(x) = \sum_{k=0}^{n-1} a_k x^k$  e  $n$  pontos  $x_0, x_1, \dots, x_{n-1}$ , desejamos calcular os  $n$  valores  $A(x_0), A(x_1), \dots, A(x_{n-1})$ . Para  $0 \leq i \leq j \leq n - 1$ , definimos os polinômios  $P_{ij}(x) = \prod_{k=i}^j (x - x_k)$  e  $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$ . Observe que  $Q_{ij}(x)$  tem limite de grau no máximo  $j - i$ .

- a.** Prove que  $A(x) \bmod (x - z) = A(z)$  para qualquer ponto  $z$ .
- b.** Prove que  $Q_{kk}(x) = A(x_k)$  e que  $Q_{0, n-1}(x) = A(x)$ .
- c.** Prove que, para  $i \leq k \leq j$ , temos  $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$  e  $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$ .
- d.** Forneça um algoritmo de tempo  $O(n \lg^2 n)$  para avaliar  $A(x_0), A(x_1), \dots, A(x_{n-1})$ .

### 30-6 FFT com a utilização da aritmética modular

Conforme foi definida, a transformação discreta de Fourier exige o uso de números complexos, o que pode resultar em uma perda de precisão devido a erros de arredondamento. Para alguns problemas, sabe-se que a resposta contém apenas inteiros, e é desejável utilizar uma variação da FFT baseada em aritmética modular, a fim de garantir que a resposta será calculada com exatidão. Um exemplo de tal problema é o de multiplicar dois polinômios com coeficientes inteiros. O Exercício 30.2-6 oferece uma abordagem, usando um módulo de comprimento  $\Omega(n)$  bits para tratar uma DFT sobre  $n$  pontos. Este problema apresenta outra abordagem que usa um módulo

com o comprimento mais razoável  $O(\lg n)$ ; ele exige que você entenda o assunto do Capítulo 31. Seja  $n$  uma potência de 2.

- a. Vamos supor que procuramos pelo menor  $k$  tal que  $p = kn + 1$  seja primo. Forneça um argumento heurístico simples pelo qual se poderia esperar que  $k$  fosse aproximadamente  $\lg n$ . (O valor de  $k$  poderia ser muito maior ou menor, mas podemos esperar de forma razoável examinar  $O(\lg n)$  valores candidatos de  $k$  em média.) De que modo o comprimento esperado de  $p$  se compara ao comprimento de  $n$ ?

Seja  $g$  um gerador de  $Z_p^*$  e seja  $w = g^k \bmod p$ .

- b. Demonstre que a DFT e a DFT inversa são operações inversas bem definidas de módulo  $p$ , onde  $w$  é usado como uma raiz  $n$ -ésima principal da unidade.
- c. Demonstre que a FFT e sua inversa podem ser levadas a funcionar em módulo  $p$  no tempo  $O(n \lg n)$ , onde operações sobre palavras de  $O(\lg n)$  bits demoram um tempo unitário. Suponha que o algoritmo receba  $p$  e  $w$ .
- d. Calcule a DFT módulo  $p = 17$  do vetor  $(0, 5, 3, 7, 7, 2, 1, 6)$ . Observe que  $g = 3$  é um gerador de  $Z_{17}^*$ .

## Notas do capítulo

O livro de Van Loan [303] é um tratamento da transformação rápida de Fourier. Press, Flannery, Teukolsky e Vetterling [248, 249] apresentam uma boa descrição da transformação rápida de Fourier e de suas aplicações. Para ver uma excelente introdução ao processamento de sinais, uma área de aplicação de FFT bastante popular, consulte os textos de Oppenheim e Schafer [232] e de Oppenheim e Willsky [233]. O livro de Oppenheim e Schafer também mostra como tratar casos em que  $n$  não é uma potência inteira de 2.

A análise de Fourier não é limitada a dados unidimensionais. Ela é amplamente usada em processamento de imagens para analisar dados em duas ou mais dimensões. Os livros de Gonzalez e Woods [127] e Pratt [246] discutem transformações de Fourier multidimensionais e seu uso em processamento de imagens, e os livros de Tolimieri, An e Lu [300] e de Van Loan [303] discutem os fundamentos matemáticos de transformações rápidas de Fourier.

Cooley e Tukey [68] são amplamente admitidos como os criadores da FFT na década de 1960. A FFT foi de fato descoberta muito antes dessa época, mas sua importância não foi completamente percebida antes do advento dos computadores digitais modernos. Embora Press, Flannery, Teukolsky e Vetterling atribuam as origens do método a Runge e König em 1924, um artigo de Heideman, Johnson, e Burrus [141] relata o histórico da FFT desde a época de C. F. Gauss, em 1805.

---

## *Capítulo 31*

# *Algoritmos de teoria dos números*

A teoria dos números já foi vista como um belo assunto, mas em grande parte inútil na matemática pura. Hoje em dia, os algoritmos da teoria dos números são amplamente utilizados, devido em parte à criação de esquemas de criptografia baseados em grandes números primos. A viabilidade desses esquemas se baseia em nossa habilidade para encontrar facilmente grandes primos, enquanto sua segurança depende de nossa incapacidade para fatorar o produto de grandes primos. Este capítulo apresenta alguns algoritmos da teoria dos números e algoritmos associados que estão subjacentes a tais aplicações.

A Seção 31.1 introduz conceitos básicos da teoria dos números, como a divisibilidade, a equivalência modular e fatoração única. A Seção 31.2 estuda um dos algoritmos mais antigos do mundo: o algoritmo de Euclides para calcular o máximo divisor comum de dois inteiros. A Seção 31.3 revê conceitos da aritmética modular. A Seção 31.4 estuda então o conjunto de múltiplos de um dado número  $a$ , módulo  $n$  e mostra como encontrar todas as soluções para a equação  $a \equiv b$  usando o algoritmo de Euclides. O teorema chinês do resto é apresentado na Seção 31.5. A Seção 31.6 considera as potências de um dado número  $a$ , módulo  $n$  e apresenta um algoritmo de elevação ao quadrado repetida para calcular de modo eficiente  $a^b \bmod n$ , dados  $a$ ,  $b$  e  $n$ . Essa operação está no núcleo do teste eficiente de números primos e da maior parte da criptografia moderna. Em seguida, a Seção 31.7 descreve o sistema de criptografia de chave pública RSA. A Seção 31.8 descreve um teste de números primos aleatórios que pode ser usado para encontrar primos maiores de modo eficiente, uma tarefa essencial na criação de chaves para o sistema de criptografia RSA. Finalmente, a Seção 31.9 revê uma heurística simples mas eficaz para fatorar inteiros menores. É curioso que a fatoração seja um problema que as pessoas talvez vejam como intratável, pois a segurança do RSA depende da dificuldade de fatorar inteiros maiores.

### **Tamanho de entradas e custo de cálculos aritméticos**

Tendo em vista que trabalharemos com inteiros maiores, precisamos ajustar nosso modo de pensar sobre o tamanho de uma entrada e sobre o custo de operações aritméticas elementares.

Neste capítulo, uma “entrada grande” em geral significa uma entrada contendo “inteiros maiores” em lugar de uma entrada contendo “muitos inteiros” (como no caso da ordenação). Desse modo, mediremos o tamanho de uma entrada em termos do *número de bits* exigidos para representar aquela entrada, não apenas em termos do número de inteiros na entrada. Um algorit-

mo com entradas inteiras  $a_1, a_2, \dots, a_k$  é um **algoritmo de tempo polinomial** se ele é executado em tempo polinomial em  $\lg a_1, \lg a_2, \dots, \lg a_k$ , ou seja, polinomial nos comprimentos de suas entradas codificadas em binário.

Na maior parte deste livro, descobrimos que é conveniente imaginar as operações aritméticas elementares (multiplicações, divisões ou cálculo de restos) como operações primitivas que demoram uma unidade de tempo. Contando o número de tais operações aritméticas que um algoritmo efetua, temos uma base para fazer uma estimativa razoável do tempo de execução real do algoritmo em um computador. Porém, as operações elementares podem ser demoradas, quando suas entradas são grandes. Desse modo, torna-se conveniente medir quantas **operações de bits** um algoritmo da teoria dos números exige. Nesse modelo, uma multiplicação de dois inteiros de  $\beta$  bits pelo método comum utiliza  $\Theta(\beta^2)$  operações de bits. De modo semelhante, a operação de dividir um inteiro de  $\beta$  bits por um inteiro mais curto, ou a operação de tomar o resto da divisão de um inteiro de  $\beta$  bits por um inteiro mais curto, pode ser executada no tempo  $\Theta(\beta^2)$  por algoritmos simples. (Ver Exercício 31.1-11.) São conhecidos métodos mais rápidos. Por exemplo, um método simples de dividir e conquistar para multiplicar dois inteiros de  $\beta$  bits tem um tempo de execução  $\Theta(\beta \lg^2 \beta)$  e o método mais rápido conhecido tem um tempo de execução  $\Theta(\beta \lg \beta \lg \lg \beta)$ . Porém, para finalidades práticas, o algoritmo  $\Theta(\beta^2)$  freqüentemente é melhor e utilizaremos esse limite como uma base para nossas análises.

Neste capítulo, os algoritmos são geralmente analisados tanto em termos do número de operações aritméticas quanto do número de operações de bits que eles exigem.

## 31.1 Noções de teoria elementar dos números

Esta seção fornece uma breve revisão de noções da teoria elementar dos números relacionadas com o conjunto  $\mathbf{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$  de inteiros e com o conjunto  $\mathbf{N} = \{0, 1, 2, \dots\}$  de números naturais.

### Divisibilidade e divisores

A noção de que um inteiro é divisível por outro é fundamental na teoria dos números. A notação  $d | a$  (leia “ $d$  divide  $a$ ”) significa que  $a = kd$  para algum inteiro  $k$ . Todo inteiro divide 0. Se  $a > 0$  e  $d | a$ , então  $|d| \leq |a|$ . Se  $d | a$ , então também dizemos que  $a$  é um **múltiplo** de  $d$ . Se  $d$  não divide  $a$ , escrevemos  $d \nmid a$ .

Se  $d | a$  e  $d \geq 0$ , dizemos que  $d$  é um **divisor** de  $a$ . Note que  $d | a$  se e somente se  $-d | a$ , de forma que nenhuma generalidade é perdida definindo-se os divisores como não negativos, com a compreensão de que o negativo de qualquer divisor de  $a$  também divide  $a$ . Um divisor de um inteiro não nulo  $a$  é no mínimo 1, mas não maior que  $|a|$ . Por exemplo, os divisores de 24 são 1, 2, 3, 4, 6, 8, 12 e 24.

Todo inteiro  $a$  é divisível pelos **divisores triviais** 1 e  $a$ . Os divisores não triviais de  $a$  também são chamados **fatores** de  $a$ . Por exemplo, os fatores de 20 são 2, 4, 5 e 10.

### Números primos e múltiplos

Um inteiro  $a > 1$  cujos únicos divisores são os divisores triviais 1 e  $a$  é chamado **número primo** (ou, mais simplesmente, um **primo**). Os primos têm muitas propriedades especiais e desempenham um papel fundamental na teoria dos números. Os primeiros 20 primos, em ordem crescente, são

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71.

O Exercício 31.1-1 lhe pede para provar que existem infinitamente muitos primos. Um inteiro  $a > 1$  que não é primo é chamado **número múltiplo** (ou, mais simplesmente, um **múltiplo**). | 673

Por exemplo, 39 é múltiplo, porque  $3 | 39$ . O inteiro 1 é chamado **unidade** e não é nem primo nem múltiplo. De modo semelhante, o inteiro 0 e todos os inteiros negativos não são primos nem múltiplos.

## O teorema da divisão, restos e equivalência modular

Dado um inteiro  $n$ , os inteiros podem ser particionados entre os que são múltiplos de  $n$  e aqueles que não são múltiplos de  $n$ . Grande parte da teoria dos números é baseada em um refinamento dessa partição obtido pela classificação dos não múltiplos de  $n$  de acordo com seus restos quando divididos por  $n$ . O teorema a seguir é a base para esse refinamento. A prova desse teorema não será dada aqui (ver, por exemplo, Niven e Zuckerman [231]).

### **Teorema 31.1 (Teorema da divisão)**

Para qualquer inteiro  $a$  e qualquer inteiro positivo  $n$ , existem inteiros únicos  $q$  e  $r$  tais que  $0 \leq r < n$  e  $a = qn + r$ .

O valor  $q = \lfloor a/n \rfloor$  é o **quociente** da divisão. O valor  $r = a \bmod n$  é o **resto** (ou **resíduo**) da divisão. Temos que  $n | a$  se e somente se  $a \bmod n = 0$ .

Os inteiros podem ser divididos em  $n$  classes de equivalência, de acordo com seus restos módulo  $n$ . A **classe de equivalência módulo  $n$**  que contém um inteiro  $a$  é

$$[a]_n = \{a + kn : k \in \mathbb{Z}\}.$$

Por exemplo,  $[3]_7 = \{\dots, -11, -4, 3, 10, 17, \dots\}$ ; outras representações para esse conjunto são  $[-4]_7$  e  $[10]_7$ . Usando a notação definida na Seção 3.2, podemos dizer que escrever  $a \in [b]_n$  é o mesmo que escrever  $a \equiv b$ . O conjunto de todas essas classes de equivalência é

$$\mathbb{Z}_n = \{[a]_n : 0 \leq a \leq n-1\} \quad (31.1)$$

Com freqüência se vê a definição

$$\mathbb{Z}_n = \{0, 1, \dots, n-1\}, \quad (31.2)$$

que deve ser lida como equivalente à equação (31.1), com a compreensão de que 0 representa  $[0]_n$ , 1 representa  $[1]_n$  e assim por diante; cada classe é representada pelo seu elemento não negativo mínimo. Contudo, as classes de equivalência subjacentes devem ser lembradas. Por exemplo, uma referência a  $-1$  como elemento de  $\mathbb{Z}_n$  é uma referência a  $[n-1]_n$ , pois  $-1 \equiv n-1$ .

## Divisores comuns e máximo divisor comum

Se  $d$  é um divisor de  $a$  e também um divisor de  $b$ , então  $d$  é um **divisor comum** de  $a$  e  $b$ . Por exemplo, os divisores de 30 são 1, 2, 3, 5, 6, 10, 15 e 30, e assim os divisores comuns de 24 e 30 são 1, 2, 3 e 6. Observe que 1 é um divisor comum de dois inteiros quaisquer.

Uma propriedade importante dos divisores comuns é que

$$d | a \text{ e } d | b \text{ implica } d | (a+b) \text{ e } d | (a-b). \quad (31.3)$$

De modo mais geral, temos que

$$d | a \text{ e } d | b \text{ implica } d | (ax+by) \quad (31.4)$$

para quaisquer inteiros  $x$  e  $y$ . Além disso, se  $a | b$ , então  $|a| \leq |b|$  ou  $b = 0$ , o que implica que

$$674 \mid a | b \text{ e } b | a \text{ implica } a = \pm b. \quad (31.5)$$

O **máximo divisor comum** de dois inteiros  $a$  e  $b$ , que não sejam ambos zero, é o maior dos divisores comuns de  $a$  e  $b$ ; ele é denotado por  $\text{mdc}(a, b)$ . Por exemplo,  $\text{mdc}(24, 30) = 6$ ,  $\text{mdc}(5, 7) = 1$  e  $\text{mdc}(0, 9) = 9$ . Se  $a$  e  $b$  são ambos não nulos, então  $\text{mdc}(a, b)$  é um inteiro entre 1 e  $\min(|a|, |b|)$ . Definimos  $\text{mdc}(0, 0)$  como 0; essa definição é necessária para tornar as propriedades padrão da função mdc (como a equação (31.9) a seguir) universalmente válidas.

Temos a seguir as propriedades elementares da função mdc:

$$\text{mdc}(a, b) = \text{mdc}(b, a), \quad (31.6)$$

$$\text{mdc}(a, b) = \text{mdc}(-a, b), \quad (31.7)$$

$$\text{mdc}(a, b) = \text{mdc}(|a|, |b|), \quad (31.8)$$

$$\text{mdc}(a, 0) = |a|, \quad (31.9)$$

$$\text{mdc}(a, ka) = |a| \text{ para qualquer } k \in \mathbb{Z}. \quad (31.10)$$

O teorema seguinte fornece uma caracterização alternativa e útil de  $\text{mdc}(a, b)$ .

### **Teorema 31.2**

Se  $a$  e  $b$  são inteiros quaisquer, e não são ambos zero, então  $\text{mdc}(a, b)$  é o menor elemento positivo do conjunto  $\{ax + by : x, y \in \mathbb{Z}\}$  de combinações lineares de  $a$  e  $b$ .

**Prova** Seja  $s$  a menor dessas combinações lineares positivas de  $a$  e  $b$ , e seja  $s = ax + by$  para algum  $x, y \in \mathbb{Z}$ . Seja  $q = \lfloor a/s \rfloor$ . A equação (3.8) implica então

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a(1 - qx) + b(-qy). \end{aligned}$$

e desse modo  $a \bmod s$  também é uma combinação linear de  $a$  e  $b$ . Porém, como  $0 \leq a \bmod s$ , temos que  $a \bmod s = 0$ , porque  $s$  é a menor dessas combinações lineares positivas. Então,  $s \mid a$  e, por raciocínio análogo,  $s \mid b$ . Assim,  $s$  é um divisor comum de  $a$  e  $b$ , e então  $\text{mdc}(a, b) \geq s$ . A equação (31.4) implica que  $\text{mdc}(a, b) \mid s$ , pois  $\text{mdc}(a, b)$  divide tanto  $a$  quanto  $b$ , e  $s$  é uma combinação linear de  $a$  e  $b$ . Porém,  $\text{mdc}(a, b) \mid s$  e  $s > 0$  implicam que  $\text{mdc}(a, b) \leq s$ . A combinação de  $\text{mdc}(a, b) \geq s$  e  $\text{mdc}(a, b) \leq s$  produz  $\text{mdc}(a, b) = s$ ; concluímos que  $s$  é o máximo divisor comum de  $a$  e  $b$ . ■

### **Corolário 31.3**

Para quaisquer inteiros  $a$  e  $b$ , se  $d \mid a$  e  $d \mid b$ , então  $d \mid \text{mdc}(a, b)$ .

**Prova** Esse corolário decorre da equação (31.4), porque  $\text{mdc}(a, b)$  é uma combinação linear de  $a$  e  $b$  pelo Teorema 31.2. ■

### **Corolário 31.4**

Para todos os inteiros  $a$  e  $b$  e qualquer inteiro não negativo  $n$ ,

$$\text{mdc}(an, bn) = n \text{ mdc}(a, b)$$

**Prova** Se  $n = 0$ , o corolário é trivial. Se  $n > 0$ , então  $\text{mdc}(an, bn)$  é o menor elemento positivo do conjunto  $\{anx + bny\}$ , que é  $n$  vezes o menor elemento positivo do conjunto  $\{ax + by\}$ . ■

### **Corolário 31.5**

Para todos os inteiros positivos  $n$ ,  $a$  e  $b$ , se  $n \mid ab$  e  $\text{mdc}(a, n) = 1$ , então  $n \mid b$ .

**Prova** A prova fica como o Exercício 31.1-4.

## Inteiros primos entre si

Dois inteiros  $a, b$  são ditos **primos entre si** se seu único divisor comum é 1, isto é, se  $\text{mdc}(a, b) = 1$ . Por exemplo, 8 e 15 são primos entre si, pois os divisores de 8 são 1, 2, 4 e 8, enquanto os divisores de 15 são 1, 3, 5 e 15. O teorema a seguir afirma que, se cada um de dois inteiros e um inteiro  $p$  são primos entre si, então seu produto  $ab$  e  $p$  são primos entre si.

### Teorema 31.6

Para quaisquer inteiros  $a, b$  e  $p$ , se  $\text{mdc}(a, p) = 1$  e  $\text{mdc}(b, p) = 1$ , então  $\text{mdc}(ab, p) = 1$ .

**Prova** Segue-se do Teorema 31.2 que existem inteiros  $x, y, x'$  e  $y'$  tais que

$$ax + py = 1,$$

$$bx' + py' = 1.$$

Multiplicando essas equações e reorganizando, temos

$$ab(xx') + p(ybx' + y'ax + pyy') = 1.$$

Desse modo, tendo em vista que 1 é uma combinação linear positiva de  $ab$  e  $p$ , usamos o Teorema 31.2 para completar a prova. ■

Dizemos que inteiros  $n_1, n_2, \dots, n_k$  são **primos entre si dois a dois** se, sempre que  $i \neq j$ , temos  $\text{mdc}(n_i, n_j) = 1$ .

## Fatoração única

Um fato elementar mas importante sobre a divisibilidade por primos é dado a seguir.

### Teorema 31.7

Para todos os primos  $p$  e todos os inteiros  $a, b$ , se  $p \mid ab$ , então  $p \mid a$  ou  $p \mid b$  (ou ambos).

**Prova** Suponha para fins de contradição que  $p \mid ab$ , mas que  $p \nmid a$  e  $p \nmid b$ . Desse modo,  $\text{mdc}(a, p) = 1$  e  $\text{mdc}(b, p) = 1$ , pois os únicos divisores de  $p$  são 1 e  $p$  e, por hipótese,  $p$  não divide nem  $a$  nem  $b$ . Então, o Teorema 31.6 implica que  $\text{mdc}(ab, p) = 1$ , contradizendo nossa suposição de que  $p \mid ab$ , pois  $p \mid ab$  implica  $\text{mdc}(ab, p) = p$ . Essa contradição completa a prova. ■

Uma consequência do Teorema 31.7 é que um inteiro tem uma fatoração única em primos.

### Teorema 31.8 (Fatoração única)

Um inteiro múltiplo  $a$  pode ser escrito exatamente de um modo como um produto da forma

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

onde os  $p_i$  são primos,  $p_1 < p_2 < \dots < p_r$ , e os  $e_i$  são inteiros positivos.

**Prova** A prova fica como o Exercício 31.1-10.

Como exemplo, o número 6000 pode ser fatorado unicamente como  $2^4 \cdot 3 \cdot 5^3$ .

## Exercícios

### 31.1-1

Prove que existem infinitamente muitos primos. (*Sugestão:* Mostre que nenhum dos primos  $p_1, p_2, \dots, p_k$  divide  $(p_1 p_2 \dots p_k) + 1$ .)

### 31.1-2

Prove que, se  $a | b$  e  $b | c$ , então  $a | c$ .

### 31.1-3

Prove que, se  $p$  é primo e  $0 < k < p$ , então  $\text{mdc}(k, p) = 1$ .

### 31.1-4

Prove o Corolário 31.5.

### 31.1-5

Prove que, se  $p$  é primo e  $0 < k < p$ , então  $p \mid \binom{p}{q}$ . Conclua que, para todos os inteiros  $a, b$  e primos  $p$ ,

$$(a + b)^p \equiv a^p + b^p \pmod{p}.$$

### 31.1-6

Prove que, se  $a$  e  $b$  são inteiros positivos quaisquer tais que  $a | b$  e  $b > 0$ , então

$$(x \bmod b) \bmod a = x \bmod a$$

para qualquer  $x$ . Prove, sob as mesmas hipóteses, que

$$x \equiv y \pmod{b} \text{ implica } x \equiv y \pmod{a}$$

para quaisquer inteiros  $x$  e  $y$ .

### 31.1-7

Para qualquer inteiro  $k > 0$ , dizemos que um inteiro  $n$  é uma  $k$ -ésima potência se existe um inteiro  $a$  tal que  $a^k = n$ . Dizemos que  $n > 1$  é uma **potência não trivial** se ela é uma  $k$ -ésima potência para algum inteiro  $k > 1$ . Mostre como determinar se um dado inteiro de  $\beta$  bits  $n$  é uma potência não trivial em tempo polinomial em  $\beta$ .

### 31.1-8

Prove as equações (31.6) a (31.10).

### 31.1-9

Mostre que o operador  $\text{mdc}$  é associativo. Isto é, prove que, para todos os inteiros  $a, b$  e  $c$ ,

$$\text{mdc}(a, \text{mdc}(b, c)) = \text{mdc}(\text{mdc}(a, b), c)$$

### 31.1-10 \*

Prove o Teorema 31.8.

### 31.1-11

Forneça algoritmos eficientes para as operações de dividir um inteiro de  $\beta$  bits por um inteiro menor e de tomar o resto da divisão de um inteiro de  $\beta$  bits por um inteiro menor. Seus algoritmos devem ser executados no tempo  $O(\beta^2)$ .

### 31.1-12

Forneça um algoritmo eficiente para converter um dado inteiro (binário) de  $\beta$  bits para uma representação decimal. Demonstre que, se a multiplicação ou divisão de inteiros cujo comprimento é no máximo  $\beta$  demorar o tempo  $M(\beta)$ , então a conversão de binário para decimal poderá ser executada no tempo  $\Theta(M(\beta) \lg \beta)$ . (Sugestão: Use uma abordagem de dividir e conquistar, obtendo as metades superior e inferior do resultado com recursões separadas.)

## 31.2 Máximo divisor comum

Nesta seção, descrevemos algoritmo de Euclides para calcular o máximo divisor comum de dois inteiros de modo eficiente. A análise de tempo de execução mostra uma conexão surpreendente com os números de Fibonacci, o que produz uma entrada de pior caso para o algoritmo de Euclides.

Vamos nos limitar nesta seção aos inteiros não negativos. Essa restrição é justificada pela equação (31.8), que declara que  $\text{mdc}(a, b) = \text{mdc}(|a|, |b|)$ .

Em princípio, podemos calcular  $\text{mdc}(a, b)$  para inteiros positivos  $a$  e  $b$  a partir dos fatores primos de  $a$  e  $b$ . De fato, se

$$a = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}, \quad (31.11)$$

$$b = p_1^{f_1} p_2^{f_2} \dots p_r^{f_r} \quad (31.12)$$

com zero expoentes sendo usados para tornar o conjunto de primos  $p_1, p_2, \dots, p_r$  igual para  $a$  e  $b$ , então, como o Exercício 31.2-1 lhe pede para mostrar,

$$\text{mdc}(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \dots p_r^{\min(e_r, f_r)}. \quad (31.13)$$

Porém, como mostraremos na Seção 31.9, os melhores algoritmos existentes para fatoração não são executados em tempo polinomial. Desse modo, essa abordagem para calcular o máximo divisor comum parece improvável para produzir um algoritmo eficiente.

O algoritmo de Euclides para calcular o máximo divisor comum é baseado no teorema a seguir.

### **Teorema 31.9 (Teorema de recursão MDC)**

Para qualquer inteiro não negativo  $a$  e qualquer inteiro positivo  $b$ ,  $\text{mdc}(a, b) = \text{mdc}(b, a \bmod b)$ .

**Prova** Mostraremos que  $\text{mdc}(a, b) = \text{mdc}(b, a \bmod b)$  dividem um ao outro, de forma que, pela equação (31.5), eles devem ser iguais (pois ambos são não negativos).

Primeiro, mostramos que  $\text{mdc}(a, b) \mid \text{mdc}(b, a \bmod b)$ . Se fizermos  $d = \text{mdc}(a, b)$ , então  $d \mid a$  e  $d \mid b$ . Pela equação (3.8),  $(a \bmod b) = a - qb$ , onde  $q = \lfloor a/b \rfloor$ . Tendo em vista que  $(a \bmod b)$  é então uma combinação linear de  $a$  e  $b$ , a equação (31.4) implica que  $d \mid (a \bmod b)$ . Assim, como  $d \mid b$  e  $d \mid (a \bmod b)$ , o Corolário 31.3 implica que  $d \mid \text{mdc}(b, a \bmod b)$  ou, de forma equivalente, que

$$\text{mdc}(a, b) \mid \text{mdc}(b, a \bmod b). \quad (31.14)$$

Mostrar que  $\text{mdc}(b, a \bmod b) \mid \text{mdc}(a, b)$  é quase o mesmo. Se agora fizermos  $d = \text{mdc}(b, a \bmod b)$ , então  $d \mid b$  e  $d \mid (a \bmod b)$ . Tendo em vista que  $a = qb + (a \bmod b)$ , onde  $q = \lfloor a/b \rfloor$ , temos que  $a$  é uma combinação linear de  $b$  e  $(a \bmod b)$ . Pela equação (31.4), concluímos que  $d \mid a$ . Como  $d \mid b$  e  $d \mid a$ , temos que  $d \mid \text{mdc}(a, b)$  pelo Corolário 31.3 ou, de forma equivalente, que

$$\text{mdc}(b, a \bmod b) \mid \text{mdc}(a, b). \quad (31.15)$$

## Algoritmo de Euclides

O algoritmo de mdc a seguir é descrito no livro *Elementos* de Euclides (cerca de 300 a.C.), embora possa ter origem até mesmo anterior. Ele foi escrito como um programa recursivo baseado diretamente no Teorema 31.9. As entradas  $a$  e  $b$  são inteiros não negativos arbitrários.

```
EUCLID( $a, b$ )
1 if  $b = 0$ 
2   then return  $a$ 
3   else return EUCLID( $b, a \bmod b$ )
```

Como exemplo da execução de EUCLID, considere o cálculo de  $\text{mdc}(30, 21)$ :

```
EUCLID(30, 21) = EUCLID(21, 9)
                  = EUCLID(9, 3)
                  = EUCLID(3, 0)
                  = 3 .
```

Nesse cálculo, há três chamadas recursivas de EUCLID.

A correção de EUCLID decorre do Teorema 31.9 e do fato de que, se o algoritmo retorna  $a$  na linha 2, então  $b = 0$ , e assim a equação (31.9) implica que  $\text{mdc}(a, b) = \text{mdc}(a, 0) = a$ . O algoritmo não pode executar recursões indefinidamente, pois o segundo argumento diminui estritamente em cada chamada recursiva e é sempre não negativo. Assim, EUCLID sempre termina com a resposta correta.

## O tempo de execução do algoritmo de Euclides

Analisamos o tempo de execução do pior caso de EUCLID como uma função do tamanho de  $a$  e  $b$ . Supomos, sem qualquer perda de generalidade, que  $a > b \geq 0$ . Essa hipótese pode ser justificada pela observação de que, se  $b > a \geq 0$ , então EUCLID( $a, b$ ) faz imediatamente a chamada recursiva EUCLID( $b, a$ ). Ou seja, se o primeiro argumento é menor que o segundo argumento, EUCLID gasta uma chamada recursiva trocando seus argumentos, e depois prossegue. De modo semelhante, se  $b = a > 0$ , o procedimento termina depois de uma chamada recursiva, pois  $a \bmod b = 0$ .

O tempo de execução global de EUCLID é proporcional ao número de chamadas recursivas que ele realiza. Nossa análise faz uso dos números de Fibonacci  $F_k$ , definidos pela recorrência (3.21).

### Lema 31.10

Se  $a > b \geq 1$  e a invocação EUCLID( $a, b$ ) executa  $k \geq 1$  chamadas recursivas, então  $a \geq F_{k+2}$  e  $b \geq F_{k+1}$ .

**Prova** A prova é por indução sobre  $k$ . Como base da indução, seja  $k = 1$ . Desse modo,  $b \geq 1 = F_2$  e, como  $a > b$ , devemos ter  $a \geq 1 = F_3$ . Tendo em vista que  $b > (a \bmod b)$ , em cada chamada recursiva o primeiro argumento é estritamente maior que o segundo; a hipótese de que  $a > b$  é então válida para cada chamada recursiva.

Suponha indutivamente que o lema seja verdadeiro se  $k - 1$  chamadas recursivas são realizadas; provaremos então que ele é verdadeiro para  $k$  chamadas recursivas. Tendo em vista que  $k > 0$ , temos  $b > 0$ , e EUCLID( $a, b$ ) chama EUCLID( $b, a \bmod b$ ) recursivamente, e este, por sua vez, efetua  $k - 1$  chamadas recursivas. Então, a hipótese indutiva implica que  $b \geq F_{k+1}$  (provando assim uma parte do lema) e  $a \geq F_k$ . Temos

$$b + (a \bmod b) = b + (a - \lfloor a/b \rfloor b)$$

$$\leq a ,$$

pois  $a > b > 0$  implica  $\lfloor a/b \rfloor \geq 1$ . Desse modo,

$$a \geq b + (a \bmod b)$$

$$\geq F_{k+1} + F_k$$

$$= F_{k+2}$$

O teorema a seguir é um corolário imediato desse lema.

**Teorema 31.11 (Teorema de Lamé)**

Para qualquer inteiro  $k \geq 1$ , se  $a > b \geq 1$  e  $b < F_{k+1}$ , então a invocação EUCLID( $a, b$ ) faz menos de  $k$  chamadas recursivas. ■

Podemos mostrar que o limite superior do Teorema 31.11 é o melhor possível. Números de Fibonacci consecutivos são uma entrada do pior caso para EUCLID. Considerando que EUCLID( $F_3, F_2$ ) faz exatamente uma chamada recursiva e que, para  $k > 2$  temos  $F_{k+1} \bmod F_k = F_{k-1}$ , também temos

$$\begin{aligned} \text{mdc}(F_{k+1}, F_k) &= \text{mdc}(F_k, (F_{k+1} \bmod F_k)) \\ &= \text{mdc}(F_k, F_{k-1}) . \end{aligned}$$

Desse modo, EUCLID( $F_{k+1}, F_k$ ) efetua recursões *exatamente*  $k - 1$  vezes, de acordo com o limite superior do Teorema 31.11.

Tendo em vista que  $F_k$  é aproximadamente  $\phi^5/\sqrt{5}$ , onde  $\phi$  é a razão áurea  $(1 + \sqrt{5})/2$  definida pela equação (3.22), o número de chamadas recursivas em EUCLID é  $O(\lg b)$ . (Veja no Exercício 31.2-5 um limite mais restrito.) Segue-se que, se EUCLID for aplicado a dois números de  $\beta$  bits, então ele executará  $O(\beta)$  operações aritméticas e  $O(\beta^2)$  operações de bits (supondo-se que a multiplicação e a divisão de números de  $\beta$  bits tomem  $O(\beta^3)$  operações de bits. O Problema 31-2 lhe pede para mostrar um limite  $O(\beta^2)$  sobre o número de operações de bits.

### A forma estendida do algoritmo de Euclides

Agora vamos reescrever o algoritmo de Euclides para calcular informações úteis adicionais. Especificamente, estendemos o algoritmo para calcular os coeficientes inteiros  $x$  e  $y$  tais que

$$d = \text{mdc}(a, b) = ax + by . \quad (31.16)$$

Observe que  $x$  e  $y$  podem ser zero ou negativos. Descobriremos mais adiante que esses coeficientes são úteis para o cálculo de inversos multiplicativos modulares. O procedimento EXTENDED-EUCLID toma como entrada um par arbitrário de inteiros e retorna uma tripla da forma  $(d, x, y)$  que satisfaz à equação (31.16).

EXTENDED-EUCLID( $a, b$ )

- 1 **if**  $b = 0$
- 2   **then return**  $(a, 1, 0)$
- 3    $(d', x', y') \leftarrow \text{EXTENDED-EUCLID}(b, a, \bmod b)$
- 4    $(d, x, y) \leftarrow (d', y', x' - \lfloor a/b \rfloor y')$
- 5 **return**  $(d, x, y)$

| $a$ | $b$ | $\lfloor a/b \rfloor$ | $d$   | $x$ | $y$ |
|-----|-----|-----------------------|-------|-----|-----|
| 99  | 78  | 1                     | 3 -11 | 14  |     |
| 78  | 21  | 3                     | 3     | 3   | -11 |
| 21  | 15  | 1                     | 3     | -2  | 3   |
| 15  | 6   | 2                     | 3     | 1   | -2  |
| 6   | 3   | 2                     | 3     | 0   | 1   |
| 3   | 0   | -                     | 3     | 1   | 0   |

FIGURA 31.1 Um exemplo da operação de EXTENDED-EUCLID sobre as entradas 99 e 78. Cada linha mostra um nível da recursão: os valores das entradas  $a$  e  $b$ , o valor calculado  $\lfloor a/b \rfloor$  e os valores  $d$ ,  $x$  e  $y$  retornados. A tripla  $(d, x, y)$  retornada se torna a tripla  $(d', x', y')$  usada no cálculo do nível mais alto de recursão seguinte. A chamada EXTENDED-EUCLID(99, 78) retorna  $(3, -11, 14)$ , e assim  $\text{mdc}(99, 78) = 3$  e  $\text{mdc}(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$

A Figura 31.1 ilustra a execução de EXTENDED-EUCLID com o cálculo de  $\text{mdc}(99, 78)$ .

O procedimento EXTENDED-EUCLID é uma variação do procedimento EUCLID. A linha 1 é equivalente ao teste “ $b = 0$ ” na linha 1 de EUCLID. Se  $b = 0$ , então EXTENDED-EUCLID retorna não apenas  $d = a$  na linha 2, mas também os coeficientes  $x = 1$  e  $y = 0$ , de modo que  $a = ax + by$ . Se  $b \neq 0$ , EXTENDED-EUCLID primeiro calcula  $(d', y', x')$  tal que  $d' = \text{mdc}(b, a \bmod b)$  e

$$d' = bx' + (a \bmod b)y'. \quad (31.17)$$

Como ocorre com EUCLID, temos nesse caso  $d = \text{mdc}(a, b) = d' = \text{mdc}(b, a \bmod b)$ . Para obter  $x$  e  $y$  tais que  $d = ax + by$ , começamos reescrevendo a equação (31.17) usando a equação  $d = d'$  e a equação (3.8):

$$\begin{aligned} d &= bx' + (a \bmod b)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y'). \end{aligned}$$

Desse modo, a escolha de  $x = y'$  e  $y = x' - \lfloor a/b \rfloor y'$  satisfaz à equação  $d = ax + by$ , provando a correção de EXTENDED-EUCLID.

Tendo em vista que o número de chamadas recursivas feitas em EUCLID é igual ao número de chamadas recursivas feitas em EXTENDED-EUCLID, os tempos de execução de EUCLID e EXTENDED-EUCLID são iguais, até dentro de um fator constante. Isto é, para  $a > b > 0$ , o número de chamadas recursivas é  $O(\lg b)$ .

## Exercícios

### 31.2-1

Prove que as equações (31.11) e (31.12) implicam a equação (31.13).

### 31.2-2

Calcule os valores  $(d, x, y)$  que a chamada EXTENDED-EUCLID(899, 493) retorna.

### 31.2-3

Prove que, para todos os inteiros  $a, k$  e  $n$ ,

$$\text{mdc}(a, n) = \text{mdc}(a + kn, n)$$

### 31.2-4

Reescreva EUCLID em uma forma iterativa que use apenas uma quantidade constante de memória (isto é, armazene somente um número constante de valores inteiros).

### 31.2-5

Se  $a > b \geq 0$ , mostre que a invocação  $\text{EUCLID}(a, b)$  faz no máximo  $1 + \log_{\phi} b$  chamadas recursivas. Melhore esse limite para  $1 + \log_{\phi} (b / \text{mdc}(a, b))$ .

### 31.2-6

O que  $\text{EXTEND-EUCLID}(F_{k+1}, F_k)$  retorna? Prove a correção da sua resposta.

### 31.2-7

Defina a função mdc para mais de dois argumentos pela equação recursiva  $\text{mdc}(a_0, a_1, \dots, a_n) = \text{mdc}(a_0, \text{mdc}(a_0, a_1, \dots, a_n))$ . Mostre que a função mdc retorna a mesma resposta, independente da ordem na qual seus argumentos são especificados. Mostre também como encontrar inteiros  $x_0, x_1, \dots, x_n$  tais que  $\text{mdc}(a_0, a_1, \dots, a_n) = a_0x_0, a_1x_1, \dots, a_nx_n$ . Mostre que o número de divisões executadas por seu algoritmo é  $O(n + \lg(\max \{a_0, a_1, \dots, a_n\}))$ .

### 31.2-8

Defina mmc( $a_0, a_1, \dots, a_n$ ) como o **mínimo múltiplo comum** dos  $n$  inteiros  $a_0, a_1, \dots, a_n$ , isto é, o menor inteiro não negativo que é um múltiplo de cada  $a_i$ . Mostre como calcular mmc( $a_0, a_1, \dots, a_n$ ) de forma eficiente usando a operação mdc (de dois argumentos) como uma sub-rotina.

### 31.2-9

Prove que  $n_1, n_2, n_3$  e  $n_4$  são primos entre si dois a dois se e somente se  $\text{mdc}(n_1n_2, n_3n_4) = \text{mdc}(n_1n_3, n_2n_4) = 1$ . Mostre, de modo mais geral, que  $n_1, n_2, \dots, n_k$  são primos entre si dois a dois se e somente se um conjunto de pares  $\lceil \lg k \rceil$  de números derivados de  $n_i$  são primos entre si.

## 31.3 Aritmética modular

Informalmente, podemos imaginar a aritmética modular como a aritmética usual sobre os inteiros, exceto pelo fato de que, se estamos trabalhando em módulo  $n$ , então todo resultado  $x$  é substituído pelo elemento de  $\{0, 1, \dots, n - 1\}$  equivalente a  $x$ , módulo  $n$  (isto é,  $x$  é substituído por  $x \bmod n$ ). Esse modelo informal é suficiente se nos limitarmos às operações de adição, subtração e multiplicação. Um modelo mais formal para aritmética modular, que damos agora, é melhor descrito dentro da estrutura da teoria de grupos.

## Grupos finitos

Um **grupo**  $(S, \oplus)$  é um conjunto  $S$  unido a uma operação binária  $\oplus$  definida sobre  $S$  para a qual são válidas as seguintes propriedades.

1. **Fechamento:** Para todo  $a, b \in S$ , temos  $a \oplus b \in S$ .
2. **Identidade:** Existe um elemento  $e \in S$ , chamado **identidade** do grupo, tal que  $e \oplus a = a \oplus e = a$  para todo  $a \in S$ .
3. **Associatividade:** Para todo  $a, b, c \in S$ , temos  $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ .
4. **Inversos:** Para cada  $a \in S$ , existe um elemento único  $b \in S$ , chamado **inverso** de  $a$ , tal que  $a \oplus b = b \oplus a = e$ .

Como exemplo, considere o grupo familiar  $(\mathbb{Z}, +)$  dos inteiros  $\mathbb{Z}$  sob a operação de adição: 0 é a identidade, e o inverso de  $a$  é  $-a$ . Se um grupo  $(S, \oplus)$  satisfaz à **lei comutativa**  $a \oplus b = b \oplus a$  para todo  $a, b \in S$ , então ele é um **grupo abeliano**. Se um grupo  $(S, \oplus)$  satisfaz a  $|S| < \infty$ , então ele é um **grupo finito**.

## Os grupos definidos por adição e multiplicação modulares

Podemos formar dois grupos abelianos finitos usando a adição e a multiplicação módulo  $n$ , onde  $n$  é um inteiro positivo. Esses grupos são baseados nas classes de equivalência dos inteiros módulo  $n$ , definidas na Seção 31.1.

Para definir um grupo em  $\mathbb{Z}_n$ , precisamos ter operações binárias adequadas, que obtemos redefinindo as operações comuns de adição e multiplicação. É fácil definir operações de adição e multiplicação para  $\mathbb{Z}_n$ , porque a classe de equivalência de dois inteiros determina exclusivamente a classe de equivalência de sua soma ou seu produto. Isto é, se  $a \equiv a' \pmod{n}$  e  $b \equiv b' \pmod{n}$ , então

$$a + b \equiv a' + b' \pmod{n}$$

$$ab \equiv a'b' \pmod{n}$$

| $+_6$ | 0 | 1 | 2 | 3 | 4 | 5 | $\cdot_{15}$ | 1  | 2  | 4  | 7  | 8  | 11 | 13 | 14 |
|-------|---|---|---|---|---|---|--------------|----|----|----|----|----|----|----|----|
| 0     | 0 | 1 | 2 | 3 | 4 | 5 | 1            | 1  | 2  | 4  | 7  | 8  | 11 | 13 | 14 |
| 1     | 1 | 2 | 3 | 4 | 5 | 0 | 2            | 2  | 4  | 8  | 14 | 1  | 7  | 11 | 13 |
| 2     | 2 | 3 | 4 | 5 | 0 | 1 | 4            | 4  | 8  | 1  | 13 | 2  | 14 | 7  | 11 |
| 3     | 3 | 4 | 5 | 0 | 1 | 2 | 7            | 7  | 14 | 13 | 4  | 11 | 2  | 1  | 8  |
| 4     | 4 | 5 | 0 | 1 | 2 | 3 | 8            | 8  | 1  | 2  | 11 | 4  | 13 | 14 | 7  |
| 5     | 5 | 0 | 1 | 2 | 3 | 4 | 11           | 11 | 7  | 14 | 2  | 13 | 1  | 8  | 4  |
|       |   |   |   |   |   |   | 13           | 13 | 11 | 7  | 1  | 14 | 8  | 4  | 2  |
|       |   |   |   |   |   |   | 14           | 14 | 13 | 11 | 8  | 7  | 4  | 2  | 1  |

(a)

(b)

FIGURA 31.2 Dois grupos finitos. As classes de equivalência são denotadas por seus elementos representantes. (a) O grupo  $(\mathbb{Z}_6, +_6)$ . (b) O grupo  $(\mathbb{Z}_{15}^*, \cdot_{15})$

Desse modo, definimos a adição e a multiplicação módulo  $n$ , denotadas por  $+_n$  e  $\cdot_n$ , como a seguir:

$$[a]_n +_n [b]_n = [a + b]_n, \quad (31.18)$$

$$[a]_n \cdot_n [b]_n = [ab]_n.$$

(A subtração pode ser definida de modo semelhante sobre  $\mathbb{Z}_n$  por  $[a]_n -_n [b]_n = [a - b]_n$ , mas a divisão é mais complicada, como veremos.) Esses fatos justificam a prática comum e conveniente de usar o menor elemento não negativo de cada classe de equivalência como seu representante ao executar cálculos em  $\mathbb{Z}_n$ . A adição, a subtração e a multiplicação são executadas da maneira usual sobre os representantes, mas cada resultado  $x$  é substituído pelo representante de sua classe (isto é, por  $x \bmod n$ ).

Usando essa definição de adição módulo  $n$ , definimos o **grupo aditivo módulo  $n$**  como  $(\mathbb{Z}_n, +_n)$ . Esse tamanho do grupo aditivo módulo  $n$  é  $|\mathbb{Z}_n| = n$ . A Figura 31.2(a) fornece a tabela de operação para o grupo  $(\mathbb{Z}_6, +_6)$ .

**Teorema 31.12**

O sistema  $(\mathbb{Z}_n, +_n)$  é um grupo abeliano finito.

**Prova** A equação (31.18) mostra que  $(\mathbb{Z}_n, +_n)$  é fechado. A associatividade e a comutatividade de  $+_n$  decorrem da associatividade e da comutatividade de  $+$ :

$$\begin{aligned} ([a]_n +_n [b]_n) +_n [c]_n &= [a + b]_n +_n [c]_n \\ &= [(a + b) + c]_n \\ &= [a + (b + c)]_n \\ &= [a]_n +_n [b + c]_n \\ &= [a]_n +_n ([b]_n +_n [c]_n), \\ [a]_n +_n [b]_n &= [a + b]_n \\ &= [b + a]_n \\ &= [b]_n +_n [a]_n. \end{aligned}$$

O elemento identidade de  $(\mathbb{Z}_n, +_n)$  é 0 (isto é,  $[0]_n$ ). O inverso (aditivo) de um elemento  $a$  (ou seja,  $[a]_n$ ) é o elemento  $-a$  (isto é,  $[-a]_n$  ou  $[n-a]_n$ ), pois  $[a]_n +_n [-a]_n = [a - a]_n = [0]_n$ . ■

Usando a definição de multiplicação módulo  $n$ , definimos o **grupo multiplicativo módulo  $n$**  como  $(\mathbb{Z}_n^*, \cdot_n)$ . Os elementos desse grupo são o conjunto  $\mathbb{Z}_n^*$  de elementos em  $\mathbb{Z}_n$  que são primos entre si para  $n$ :

$$\mathbb{Z}_n^* = \{[a]_n \in \mathbb{Z}_n : \text{mdc}(a, n) = 1\}.$$

Para ver que  $\mathbb{Z}_n^*$  é bem definido, observe que, para  $0 \leq a < n$ , temos  $a \equiv (a + kn) \pmod{n}$  para todos os inteiros  $k$ . Então, pelo Exercício 31.2-3,  $\text{mdc}(a, n) = 1$  implica  $\text{mdc}(a + kn, n) = 1$  para todos os inteiros  $k$ . Tendo em vista que  $[a]_n = \{a + kn : k \in \mathbb{Z}_n\}$ , o conjunto  $\mathbb{Z}_n^*$  é bem definido. Um exemplo de tal grupo é

$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\},$$

onde a operação de grupo é a multiplicação módulo 15. (Aqui, denotamos um elemento  $[a]_{15}$  como  $a$ ; por exemplo, denotamos  $[7]_{15}$  como 7.) A Figura 31.2(b) mostra o grupo  $(\mathbb{Z}_{15}^*, \cdot_{15})$ . Por exemplo,  $8 \cdot 11 \equiv 13$ , funcionando em  $\mathbb{Z}_{15}^*$ . A identidade para esse grupo é 1.

**Teorema 31.13**

O sistema  $(\mathbb{Z}_n^*, \cdot_n)$  é um grupo abeliano finito.

**Prova** O Teorema 31.6 implica que  $(\mathbb{Z}_n^*, \cdot_n)$  é fechado. A associatividade e a comutatividade podem ser provadas para  $\cdot_n$  como foram para  $+_n$  na prova do Teorema 31.12. O elemento identidade é  $[1]_n$ . Para mostrar a existência de inversos, seja  $a$  um elemento de  $\mathbb{Z}_n^*$ , e seja  $(d, x, y)$  a saída de EXTENDED-EUCLID( $a, n$ ). Então  $d = 1$ , pois  $a \in \mathbb{Z}_n^*$  e

$$ax + ny = 1$$

ou, de modo equivalente,

$$684 \mid ax \equiv 1 \pmod{n}.$$

Desse modo,  $[x]_n$  é um inverso multiplicativo de  $[\alpha]_n$ , módulo  $n$ . A prova de que inversos estão definidos de modo exclusivo será adiada até o Corolário 31.26.

Como exemplo do cálculo de inversos multiplicativos, suponha que  $\alpha = 5$  e  $n = 11$ . Então, EXTENDED-EUCLID( $\alpha, n$ ) retorna  $(d, x, y) = (1, -2, 1)$ , de forma que  $1 = 5 \cdot (-2) + 11 \cdot 1$ . Desse modo,  $-2$  (isto é,  $9 \bmod 11$ ) é um inverso multiplicativo de  $5$  módulo  $11$ .

Ao trabalharmos com os grupos  $(\mathbb{Z}_n, +_n)$  e  $(\mathbb{Z}_n^*, \cdot_n)$  no restante deste capítulo, seguiremos a prática conveniente de denotar classes de equivalência por seus elementos representantes e denotar as operações  $+_n$  e  $\cdot_n$  pelas notações aritmética usuais  $+$  e  $\cdot$  (ou justaposição), respectivamente. Além disso, as equivalências módulo  $n$  também podem ser interpretadas como equações em  $\mathbb{Z}_n$ . Por exemplo, as duas declarações a seguir são equivalentes:

$$ax \equiv b \pmod{n} .$$

$$[\alpha]_n \cdot_n [x]_n = [b]_n .$$

Como uma conveniência adicional, às vezes nos referimos a um grupo  $(S, \oplus)$  somente como  $S$ , quando a operação é entendida a partir do contexto. Desse modo, podemos fazer referência aos grupos  $(\mathbb{Z}_n, +_n)$  e  $(\mathbb{Z}_n^*, \cdot_n)$  como  $\mathbb{Z}_n$  e  $\mathbb{Z}_n^*$ , respectivamente.

O inverso (multiplicativo) de um elemento  $\alpha$  é denotado por  $(\alpha^{-1} \bmod n)$ . A divisão em  $\mathbb{Z}_n^*$  é definida pela equação  $a/b \equiv ab^{-1}$ . Por exemplo, em  $\mathbb{Z}_{15}^*$  temos que  $7^{-1} \equiv 13 \pmod{15}$ , pois  $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$ , de modo que  $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$ .

O tamanho de  $\mathbb{Z}_n^*$  é denotado por  $\phi(n)$ . Essa função, conhecida como *função phi de Euler*, satisfaz à equação

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right), \quad (31.19)$$

onde  $p$  é executada sobre todos os primos que dividem  $n$  (inclusive o próprio  $n$ , se  $n$  é primo). Não provaremos essa fórmula aqui. Intuitivamente, começamos com uma lista dos  $n$  restos  $\{0, 1, \dots, n-1\}$  e depois, para cada primo  $p$  que divide  $n$ , cancelamos todo múltiplo de  $p$  na lista. Por exemplo, como os divisores primos de  $45$  são  $3$  e  $5$ ,

$$\begin{aligned} \phi(45) &= 45 \left(1 - \frac{1}{3}\right) \left(1 - \frac{1}{5}\right) \\ &= 45 \left(\frac{2}{3}\right) \left(\frac{4}{5}\right) \\ &= 24 . \end{aligned}$$

Se  $p$  é primo, então  $\mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$ , e

$$\phi(p) = p - 1 \quad (31.20)$$

Se  $n$  é múltiplo, então  $\phi(n) < n - 1$ .

## Subgrupos

Se  $(S, \oplus)$  é um grupo,  $S' \subseteq S$ , e  $(S', \oplus)$  também é um grupo, então  $(S', \oplus)$  é chamado **subgrupo** de  $(S, \oplus)$ . Por exemplo, os inteiros pares formam um subgrupo dos inteiros sob a operação de adição. O teorema a seguir fornece uma ferramenta útil para reconhecer subgrupos.

**Teorema 31.14 (Um subconjunto fechado não vazio de um grupo finito é um subgrupo)**

Se  $(S, \oplus)$  é um grupo finito e  $S'$  é qualquer subconjunto não vazio de  $S$  tal que  $a \oplus b \in S'$  para todo  $a, b \in S'$ , então  $(S', \oplus)$  é um subgrupo de  $(S, \oplus)$ .

**Prova** A prova fica para o Exercício 31.3-2. ■

Por exemplo, o conjunto  $\{0, 2, 4, 6\}$  forma um subgrupo de  $\mathbf{Z}_8$ , pois ele é não vazio e fechado sob a operação  $+$  (isto é, é fechado sob  $+_8$ ).

O teorema a seguir fornece uma restrição extremamente útil sobre o tamanho de um subgrupo; omitimos a prova.

**Teorema 31.15 (Teorema de Lagrange)**

Se  $(S, \oplus)$  é um grupo finito e  $(S', \oplus)$  é um subgrupo de  $(S, \oplus)$ , então  $|S'|$  é um divisor de  $|S|$ .

Um subgrupo  $S'$  de um grupo  $S$  é dito um subgrupo **próprio** se  $S' \neq S$ . O corolário a seguir será usado em nossa análise do procedimento de teste do caráter primo de Miller-Rabin na Seção 31.8.

**Corolário 31.16**

Se  $S'$  é um subgrupo próprio de um grupo finito  $S$ , então  $|S'| \leq |S|/2$ . ■

## Subgrupos gerados por um elemento

O Teorema 31.14 fornece um caminho interessante para produzir um subgrupo de um grupo finito  $(S, \oplus)$ : escolha um elemento  $a$  e tome todos os elementos que podem ser gerados a partir de  $a$  usando a operação de grupo. Especificamente, defina  $a^{(k)}$  para  $k \geq 1$  por

$$a^{(k)} = \underbrace{\oplus_{i=1}^k a}_{k} = a \oplus \underbrace{a \oplus \dots \oplus a}_{k}.$$

Por exemplo, se tomarmos  $a = 2$  no grupo  $\mathbf{Z}_6$ , a sequência  $a^{(1)}, a^{(2)}, \dots$  será  $2, 4, 0, 2, 4, 0, 2, 4, 0, \dots$ .

No grupo  $\mathbf{Z}_n$ , temos  $a^{(k)} = ka$  e, no grupo  $\mathbf{Z}_n^*$ , temos  $a^{(k)} = a^k$ . O **subgrupo gerado por a**, denotado por  $\langle a \rangle$  ou  $(\langle a \rangle, \oplus)$ , é definido por

$$\langle a \rangle = \{a^{(k)} : k \geq 1\}.$$

Dizemos que  $a$  **gera** o subgrupo  $\langle a \rangle$ , ou que  $a$  é um **gerador** de  $\langle a \rangle$ . Como  $S$  é finito,  $\langle a \rangle$  é um subconjunto finito de  $S$ , incluindo possivelmente todo o conjunto  $S$ . Como a associatividade de  $\oplus$  implica

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)},$$

$\langle a \rangle$  é fechado e, portanto, pelo Teorema 31.14,  $\langle a \rangle$  é um subgrupo de  $S$ . Por exemplo, em  $\mathbf{Z}_6$ , temos

$$\langle 0 \rangle = \{0\},$$

$$\langle 1 \rangle = \{0, 1, 2, 3, 4, 5\},$$

$$686 \quad \langle 2 \rangle = \{0, 2, 4\}.$$

De modo semelhante, em  $Z_7^*$ , temos

$$\langle 1 \rangle = \{1\},$$

$$\langle 2 \rangle = \{1, 2, 4\},$$

$$\langle 3 \rangle = \{1, 2, 3, 4, 5, 6\}.$$

A **ordem** de  $a$  (no grupo  $S$ ), denotada por  $\text{ord}(a)$ , é definida como o menor inteiro positivo  $t$  tal que  $a^{(t)} = e$ .

### Teorema 31.17

Para qualquer grupo finito  $(S, \oplus)$  e qualquer  $a \in S$ , a ordem de um elemento é igual ao tamanho do subgrupo que ele gera, ou  $\text{ord}(a) = |\langle a \rangle|$ .

**Prova** Seja  $t = \text{ord}(a)$ . Como  $a^{(t)} = e$  e  $a^{(t+k)} = a^{(t)} \oplus a^{(k)}$  para  $k \geq 1$ , se  $i > t$ , então  $a^{(i)} = a^{(j)}$  para algum  $j < i$ . Desse modo, nenhum elemento novo é visto depois de  $a^{(t)}$ , de modo que  $\langle a \rangle = \{a^{(1)}, a^{(2)}, \dots, a^{(t)}\}$  e  $|\langle a \rangle| \leq t$ . Para mostrar que  $|\langle a \rangle| \geq t$ , suponha para fins de contradição que  $a^{(i)} = a^{(j)}$  para algum  $i, j$  que satisfaça a  $1 \leq i < j \leq t$ . Então,  $a^{(i+k)} = a^{(j+k)}$  para  $k \geq 0$ . Porém, isso implica que  $a^{(i+(t-j))} = a^{(j+(t-j))}$ , uma contradição, pois  $i + (t-j) < t$ , mas  $t$  é o menor valor positivo tal que  $a^{(t)} = e$ . Então, cada elemento da seqüência  $a^{(1)}, a^{(2)}, \dots, a^{(t)}$  é distinto, e . Concluímos que  $\text{ord}(a) = |\langle a \rangle|$ . ■

### Corolário 31.18

A seqüência  $a^{(1)}, a^{(2)}, \dots$  é periódica com período  $t = \text{ord}(a)$ ; isto é,  $a^{(i)} = a^{(j)}$  se e somente se  $i \equiv j \pmod{t}$ .

É coerente com o corolário anterior definir  $a^{(0)}$  como  $e$  e  $a^{(i)}$  como  $a^{(i \bmod t)}$ , onde  $t = \text{ord}(a)$ , para todos os inteiros  $i$ .

### Corolário 31.19

Se  $(S, \oplus)$  é um grupo finito com identidade  $e$ , então para todo  $a \in S$ ,

$$a^{(|S|)} = e.$$

**Prova** O teorema de Lagrange implica que  $\text{ord}(a) \mid |S|$ , e assim  $|S| \equiv 0$ , onde  $t = \text{ord}(a)$ . Então,  $a^{(|S|)} = a^{(0)} = e$ . ■

## Exercícios

### 31.3-1

Desenhe as tabelas de operação de grupo para os grupos  $(Z_4, +_4)$  e  $(Z_5^*, \cdot_5)$ . Mostre que esses grupos são isomórficos, exibindo uma correspondência de um para um  $\alpha$  entre seus elementos, tal que  $a + b \equiv c \pmod{4}$  se e somente se  $\alpha(a) \cdot \alpha(b) \equiv \alpha(c)$ .

### 31.3-2

Prove o Teorema 31.14.

### 31.3-3

Mostre que, se  $p$  é primo e  $e$  é um inteiro positivo, então

$$\phi(p^e) = p^{e-1}(p-1).$$

### 31.3-4

Mostre que, para qualquer  $n > 1$  e para qualquer  $a \in Z_n^*$ , a função  $f_a : Z_n^* \rightarrow Z_n^*$  definida por  $f_a(x) = ax \bmod n$  é uma permutação de  $Z_n^*$ .

### 31.3-5

Liste todos os subgrupos de  $Z_9$  e de  $Z_{13}^*$ .

## 31.4 Resolução de equações lineares modulares

Agora, vamos considerar o problema de encontrar soluções para a equação

$$ax \equiv b \pmod{n}, \quad (31.21)$$

onde  $a > 0$  e  $n > 0$ . Existem diversas aplicações para esse problema; por exemplo, nós o empregaremos como parte do procedimento para encontrar chaves no sistema de criptografia de chave pública RSA, na Seção 31.7. Supomos que  $a$ ,  $b$  e  $n$  são dados, e que devemos encontrar todos os valores de  $x$ , módulo  $n$ , que satisfazem à equação (31.21). Pode haver zero, uma ou mais de uma solução.

Seja  $\langle a \rangle$  a representação do subgrupo de  $\mathbb{Z}_n$  gerado por  $a$ . Tendo em vista que  $\langle a \rangle = \{a(x) : x > 0\} = \{ax \pmod{n} : x > 0\}$ , a equação (31.21) tem uma solução se e somente se  $b \in \langle a \rangle$ . O teorema de Lagrange (Teorema 31.15) nos informa que  $|\langle a \rangle|$  deve ser um divisor de  $n$ . O teorema a seguir nos dá uma caracterização precisa de  $\langle a \rangle$ .

### Teorema 31.20

Para quaisquer inteiros positivos  $a$  e  $n$ , se  $d = \text{mdc}(a, n)$ , então

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \dots, ((n/d) - 1)d\}, \quad (31.22)$$

em  $\mathbb{Z}_n$  e, desse modo,

$$|\langle a \rangle| = n/d.$$

**Prova** Começamos mostrando que  $d \in \langle a \rangle$ . Lembre-se de que EXTENDED-EUCLID( $a, n$ ) produz inteiros  $x'$  e  $y'$  tais que  $ax' + ny' = d$ . Desse modo,  $ax' \equiv d$ , de forma que  $d \in \langle a \rangle$ .

Tendo em vista que  $d \in \langle a \rangle$ , segue-se que todo múltiplo de  $d$  pertence a  $\langle a \rangle$ , porque qualquer múltiplo de um múltiplo de  $a$  é ele próprio um múltiplo de  $a$ . Então,  $\langle a \rangle$  contém todo elemento em  $\{0, d, 2d, \dots, ((n/d) - 1)d\}$ . Isto é,  $\langle d \rangle \subseteq \langle a \rangle$ .

Mostramos agora que  $\langle a \rangle \subseteq \langle d \rangle$ . Se  $m \in \langle a \rangle$ , então  $m = ax \pmod{n}$  para algum inteiro  $x$ , e assim  $m = ax + ny$  para algum inteiro  $y$ . Contudo,  $d | a$  e  $d | n$ , e então  $d | m$  pela equação (31.4). Portanto,  $m \in \langle d \rangle$ .

Combinando esses resultados, temos que  $\langle a \rangle = \langle d \rangle$ . Para ver que  $|\langle a \rangle| = n/d$ , observe que existem exatamente  $n/d$  múltiplos de  $d$  entre 0 e  $n - 1$ , inclusive. ■

### Corolário 31.21

A equação  $ax \equiv b$  pode ser resolvida para a incógnita  $x$  se e somente se  $\text{mdc}(a, n) | b$ . ■

### Corolário 31.22

A equação  $ax \equiv b$  tem  $d$  soluções distintas módulo  $n$ , onde  $d = \text{mdc}(a, n)$ , ou então não tem nenhuma solução.

**Prova** Se  $ax \equiv b$  tem uma solução, então  $b \in \langle a \rangle$ . Pelo Teorema 31.17,  $\text{ord}(a) = |\langle a \rangle|$ , e assim o Corolário 31.18 e o Teorema 31.20 implicam que a seqüência  $ai \pmod{n}$ , para  $i = 0, 1, \dots$  é periódica com período  $|\langle a \rangle| = n/d$ . Se  $b \in \langle a \rangle$ , então  $b$  aparece exatamente  $d$  vezes na seqüência  $ai \pmod{n}$ , para  $i = 0, 1, \dots, n - 1$ , pois o bloco de comprimento  $(n/d)$  de valores  $\langle a \rangle$  é repetido exatamente  $d$  vezes à medida que  $i$  cresce desde 0 até  $n - 1$ . Os índices  $x$  dessas  $d$  posições são as soluções da equação  $ax \equiv b \pmod{n}$ . ■

### **Teorema 31.23**

Seja  $d = \text{mdc}(a, n)$  e suponha que  $d = ax' + ny'$  para alguns inteiros  $x'$  e  $y'$  (por exemplo, como calculado por EXTENDED-EUCLID). Se  $d | b$ , então a equação  $ax \equiv b \pmod{n}$  tem como uma de suas soluções o valor  $x_0$ , onde

$$x_0 = x'(b/d) \pmod{n}.$$

**Prova** Temos

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod{n} && (\text{porque } ax' \equiv d \pmod{n}) \\ &\equiv d(b/d) \pmod{n} \\ &\equiv b \end{aligned}$$

e desse modo  $x_0$  é uma solução para  $ax \equiv b \pmod{n}$ . ■

### **Teorema 31.24**

Suponha que a equação  $ax \equiv b \pmod{n}$  tenha solução (isto é,  $d | b$ , onde  $d = \text{mdc}(a, n)$ ), e que  $x_0$  é qualquer solução para essa equação. Então, essa equação tem exatamente  $d$  soluções distintas, módulo  $n$ , dadas por  $x_i = x_0 + i(n/d)$  para  $i = 0, 1, \dots, d - 1$ .

**Prova** Tendo em vista que  $n/d > 0$  e  $0 \leq i(n/d) < n$  para  $i = 0, 1, \dots, d - 1$ , os valores  $x_0, x_1, \dots, x_{d-1}$  são todos distintos, módulo  $n$ . Como  $x_0$  é uma solução de  $ax \equiv d \pmod{n}$ , temos  $ax_0 \pmod{n} = b$ . Desse modo, para  $i = 0, 1, \dots, d - 1$ , temos

$$\begin{aligned} ax_i \pmod{n} &= a(x_0 + in/d) \pmod{n} \\ &= a(x_0 + ain/d) \pmod{n} \\ &= ax_0 \pmod{n} && (\text{porque } d | a) \\ &= b, \end{aligned}$$

e então  $x_i$  também é uma solução. Pelo Corolário 31.22, existem exatamente  $d$  soluções, de forma que  $x_0, x_1, \dots, x_{d-1}$  deve ser todas elas.

Agora, desenvolvemos a matemática necessária para resolver a equação  $ax \equiv b \pmod{n}$ ; o algoritmo a seguir imprime todas as soluções para essa equação. As entradas  $a$  e  $n$  são inteiros positivos arbitrários, e  $b$  é um inteiro arbitrário.

```
MODULAR-LINEAR-EQUATION-SOLVER( $a, b, n$ )
1  $(d, x', y') \leftarrow \text{EXTENDED-EUCLID}(a, n)$ 
2 if  $d | b$ 
3   then  $x_0 \leftarrow x'(b/d) \pmod{n}$ 
4   for  $i \leftarrow 0$  to  $d - 1$ 
5     do imprimir  $(x_0 + i(n/d)) \pmod{n}$ 
6   else imprimir "nenhuma solução"
```

Como um exemplo da operação desse procedimento, considere a equação  $14x = 30 \pmod{100}$  (aqui,  $a = 14$ ,  $b = 30$  e  $n = 100$ ). Chamando EXTENDED-EUCLID na linha 1, obtemos  $(d, x, y) = (2, -7, 1)$ . Como  $2 | 30$ , as linhas 3 a 5 são executadas. Na linha 3, calculamos  $x_0 = (-7)(15) \pmod{100} = 95$ . O loop nas linhas 4 e 5 imprime as duas soluções: 95 e 45.

O procedimento MODULAR-LINEAR-EQUATION-SOLVER funciona da maneira descrita a seguir. A linha 1 calcula  $d = \text{mdc}(a, n)$  bem como dois valores  $x'$  e  $y'$  tais que  $d = ax' + ny'$ , demonstrando que  $x'$  é uma solução para a equação  $ax \equiv d \pmod{n}$ . Se  $d$  não divide  $b$ , então a equação  $ax \equiv b \pmod{n}$  não tem nenhuma solução, pelo Corolário 31.21. A linha 2 verifica se  $d \mid b$ ; se não, a linha 6 informa que não existe nenhuma solução. Caso contrário, a linha 3 calcula uma solução  $x_0$  para  $ax \equiv b \pmod{n}$ , de acordo com o Teorema 31.23. Dada uma solução, o Teorema 31.24 informa que as outras  $d - 1$  soluções podem ser obtidas adicionando-se múltiplos de  $(n/d)$ , módulo  $n$ . O loop for das linhas 4 e 5 imprime todas as  $d$  soluções, começando com  $x_0$  e espaçadas entre si  $(n/d)$ , módulo  $n$ .

MODULAR-LINEAR-EQUATION-SOLVER executa  $O(\lg n + \text{mdc}(a, n))$  operações aritméticas, pois EXTENDED-EUCLID demora  $O(\lg n)$  operações aritméticas, e cada iteração do loop for das linhas 4 e 5 demora um número constante de operações aritméticas.

Os corolários do Teorema 31.24 a seguir fornecem especializações de interesse particular.

### **Corolário 31.25**

Para qualquer  $n > 1$ , se  $\text{mdc}(a, n) = 1$ , então a equação  $ax \equiv b \pmod{n}$  tem uma única solução módulo  $n$ . ■

Se  $b = 1$ , um caso comum de interesse considerável, o  $x$  que estamos procurando é um *inverso multiplicativo* de  $a$ , módulo  $n$ .

### **Corolário 31.26**

Para qualquer  $n > 1$ , se  $\text{mdc}(a, n) = 1$ , então a equação  $ax \equiv 1 \pmod{n}$  tem uma solução única, módulo  $n$ . Caso contrário, ela não tem nenhuma solução. ■

O Corolário 31.26 nos permite usar a notação  $(a^{-1} \pmod{n})$  para referenciar o inverso multiplicativo de  $a$ , módulo  $n$ , quando  $a$  e  $n$  são primos entre si. Se  $\text{mdc}(a, n) = 1$ , então uma solução para a equação  $ax \equiv 1 \pmod{n}$  é o inteiro  $x$  retornado por EXTENDED-EUCLID, pois a equação

$$\text{mdc}(a, n) = 1 = ax + ny$$

implica  $ax \equiv 1 \pmod{n}$ . Desse modo,  $(a^{-1} \pmod{n})$  pode ser calculado de forma eficiente com o uso de EXTENDED-EUCLID.

## **Exercícios**

### **31.4-1**

Encontre todas as soluções para a equação  $35x \equiv 10 \pmod{11}$ .

### **31.4-2**

Prove que a equação  $ax \equiv ay \pmod{n}$  implica  $x \equiv y$  sempre que  $\text{mdc}(a, n) = 1$ . Mostre que a condição  $\text{mdc}(a, n) = 1$  é necessária, fornecendo um contra-exemplo com  $\text{mdc}(a, n) > 1$ .

### **31.4-3**

Considere a seguinte mudança na linha 3 do procedimento MODULAR-LINEAR-EQUATION-SOLVER:

3   **then**  $x_0 \leftarrow x'(b/d) \pmod{(n/d)}$

Isso funcionará? Explique por que ou por que não.

### **31.4-4 \***

Seja  $f(x) = f_0 + f_1x + \dots + f_tx^t$  um polinômio de grau  $t$ , com coeficientes  $f_i$  tirados de  $\mathbb{Z}_p$ , onde  $p$  é primo. Dizemos que  $\mathbb{Z}_p$  é um *zero* de  $f$  se  $a \in \mathbb{Z}_p$ . Prove que, se  $a$  é um zero de  $f$ , então  $f(a) \equiv 0 \pmod{p}$  para algum polinômio  $g(x)$  de grau  $t - 1$ . Prove por indução sobre  $t$  que um polinômio

690 |  $f(x)$  de grau  $t$  pode ter no máximo  $t$  zeros distintos com um módulo  $p$  primo.

## 31.5 O teorema chinês do resto

Por volta de 100 d.C., o matemático chinês Sun-Tsu resolveu o problema de encontrar os inteiros  $x$  que deixam restos 2, 3 e 2 quando divididos por 3, 5 e 7, respectivamente. Uma dessas soluções é  $x = 23$ ; todas as soluções têm a forma  $23 + 105k$  para inteiros arbitrários  $k$ . O “teorema chinês do resto” fornece uma correspondência entre um sistema de equações de módulo igual a um conjunto de módulos primos entre si dois a dois (por exemplo, 3, 5 e 7) e uma equação de módulo igual a seu produto (por exemplo, 105).

O teorema chinês do resto tem dois usos importantes. Seja o inteiro  $n$  fatorado como  $n = n_1 n_2 \dots n_k$ , onde os fatores  $n_i$  são primos entre si dois a dois. Primeiro, o teorema chinês do resto é um “teorema de estrutura” descritivo que mostra a estrutura de  $\mathbb{Z}_n$  como algo idêntico à estrutura do produto cartesiano  $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \dots \times \mathbb{Z}_{n_k}$ , com adição e multiplicação de componentes módulo  $n_i$  no  $i$ -ésimo componente. Em segundo lugar, essa descrição pode freqüentemente ser usada para produzir algoritmos eficientes, pois o trabalho em cada um dos sistemas  $\mathbb{Z}_{n_i}$  pode ser mais eficiente (em termos de operações de bits) que o trabalho de módulo  $n$ .

### **Teorema 31.27 (Teorema chinês do resto)**

Seja  $n = n_1 n_2 \dots n_k$ , onde os  $n_i$  são primos entre si dois a dois. Considere a correspondência

$$\alpha \leftrightarrow (\alpha_1, \alpha_2, \dots, \alpha_k) \quad (31.23)$$

onde  $\alpha \in \mathbb{Z}_n$ ,  $\alpha_i \in \mathbb{Z}_{n_i}$ , e

$$\alpha_i = \alpha \bmod n_i$$

para  $i = 1, 2, \dots, k$ . Então, o mapeamento de (31.23) é uma correspondência de um para um (bi-jecção) entre  $\mathbb{Z}_n$  e o produto cartesiano  $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \dots \times \mathbb{Z}_{n_k}$ . Operações executadas sobre os elementos de  $\mathbb{Z}_n$  podem ser executadas de forma equivalente sobre as  $k$ -tuplas correspondentes, executando-se as operações de forma independente em cada posição de coordenada no sistema apropriado. Ou seja, se

$$\alpha \leftrightarrow (\alpha_1, \alpha_2, \dots, \alpha_k),$$

$$b \leftrightarrow (b_1, b_2, \dots, b_k),$$

então

$$(\alpha + b) \bmod n \leftrightarrow ((\alpha_1 + b_1) \bmod n_1, \dots, (\alpha_k + b_k) \bmod n_k). \quad (31.24)$$

$$(\alpha - b) \bmod n \leftrightarrow ((\alpha_1 - b_1) \bmod n_1, \dots, (\alpha_k - b_k) \bmod n_k). \quad (31.25)$$

$$(ab) \bmod n \leftrightarrow ((\alpha_1 b_1 \bmod n_1, \dots, \alpha_k b_k) \bmod n_k). \quad (31.26)$$

**Prova** A transformação entre as duas representações é bastante simples. Ir de  $\alpha$  para  $(\alpha_1, \alpha_2, \dots, \alpha_k)$  exige apenas  $k$  divisões. Calcular  $\alpha$  a partir das entradas  $(\alpha_1, \alpha_2, \dots, \alpha_k)$  é um pouco mais complicado; isso pode ser feito como a seguir. Começamos definindo  $m_i = n/n_i$  para  $i = 1, 2, \dots, k$ ; portanto,  $m_i$  é o produto de todos os valores  $n_j$  diferentes de  $n_i$ :  $m_i = n_1 n_2 \dots n_{i-1} n_{i+1} \dots n_k$ . Em seguida, definimos

$$c_i = m_i(m_i^{-1} \bmod n_i) \quad (31.27)$$

para  $i = 1, 2, \dots, k$ . A equação 31.27 é sempre bem definida: como  $m_i$  e  $n_i$  são primos entre si (pelo Teorema 31.6), o Corolário 31.26 garante que  $(m_i^{-1} \bmod n_i)$  existe. Finalmente, podemos calcular  $\alpha$  como uma função de  $\alpha_1, \alpha_2, \dots, \alpha_k$  assim:

$$\alpha \equiv (\alpha_1 c_1, \alpha_2 c_2, \dots, \alpha_k c_k) \pmod{n_i} \quad (31.28)$$

Agora mostramos que a equação (31.28) assegura que  $\alpha \equiv a_i \pmod{n_i}$  para  $i = 1, 2, \dots, k$ . Observe que, se  $j \neq i$ , então  $m_j \equiv 0 \pmod{n_i}$ , o que implica que  $c_j \equiv m_j \equiv 0 \pmod{n_i}$ . Observe também que  $c_i \equiv 1 \pmod{n_i}$ , a partir da equação (31.27). Desse modo, temos a interessante e útil correspondência

$$c_j \leftrightarrow (0, 0, \dots, 0, 1, 0, \dots, 0)$$

um vetor que tem valores 0 em todo lugar, exceto na  $i$ -ésima coordenada, onde tem o valor 1; desse modo, em certo sentido,  $c_i$  forma uma “base” para a representação. Então, para cada  $i$ , temos

$$\alpha \equiv a_i c_i \pmod{n_i}$$

$$\equiv a_i m_i (m_i^{-1} \pmod{n_i}) \pmod{n_i}$$

$$\equiv a_i \pmod{n_i}$$

o que desejávamos mostrar: nosso método para calcular  $\alpha$  a partir dos valores  $a_i$  produz um resultado  $\alpha$  que satisfaz às restrições  $\alpha \equiv a_i \pmod{n_i}$  para  $i = 1, 2, \dots, k$ . A correspondência é de um para um, pois podemos realizar transformações em ambos os sentidos. Finalmente, as equações (31.24) a (31.26) decorrem diretamente do Exercício 31.1-6, pois  $x \pmod{n_i} = (x \pmod{n}) \pmod{n_i}$  para qualquer  $x$  e  $i = 1, 2, \dots, k$ . ■

Os corolários a seguir serão usados mais adiante neste capítulo.

### **Corolário 31.28**

Se  $n_1, n_2, \dots, n_k$  são primos entre si dois a dois e  $n = n_1, n_2, \dots, n_k$  então, para quaisquer inteiros  $a_1, a_2, \dots, a_k$ , o conjunto de equações simultâneas

$$x \equiv a_i \pmod{n_i},$$

para  $i = 1, 2, \dots, k$ , tem uma solução única módulo  $n$  para a incógnita  $x$ .

### **Corolário 31.29**

Se  $n_1, n_2, \dots, n_k$  são primos entre si dois a dois e  $n = n_1, n_2, \dots, n_k$  então, para todos os inteiros  $x$  e  $\alpha$ ,

$$x \equiv \alpha \pmod{n_i},$$

para  $i = 1, 2, \dots, k$  se e somente se

$$x \equiv \alpha \pmod{n},$$

Como um exemplo de aplicação do teorema chinês do resto, vamos supor que temos as duas equações

$$\alpha \equiv 2 \pmod{5},$$

$$\alpha \equiv 3 \pmod{13},$$

de modo que  $a_1 = 2, n_1 = m_2 = 5, a_2 = 3$  e  $n_2 = m_1 = 13$ , e desejamos calcular  $\alpha \pmod{65}$ , pois  $n = 65$ . Pelo fato de  $13^{-1} \equiv 2 \pmod{5}$  e  $5^{-1} \equiv 8$ , temos

$$c_1 \equiv 13 (2 \pmod{5}) = 26,$$

$$c_2 \equiv 5 (8 \pmod{13}) = 40,$$

e

$$\alpha \equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65}$$

$$\equiv 52 + 120 \pmod{65}$$

$$\equiv 42 \pmod{65}.$$

|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 0  | 40 | 15 | 55 | 30 | 5  | 45 | 20 | 60 | 35 | 10 | 50 | 25 |
| 1 | 26 | 1  | 41 | 16 | 56 | 31 | 6  | 46 | 21 | 61 | 36 | 11 | 51 |
| 2 | 52 | 27 | 2  | 42 | 17 | 57 | 32 | 7  | 47 | 22 | 62 | 37 | 12 |
| 3 | 13 | 53 | 28 | 3  | 43 | 18 | 58 | 33 | 8  | 48 | 23 | 63 | 38 |
| 4 | 39 | 14 | 54 | 29 | 4  | 44 | 19 | 59 | 34 | 9  | 49 | 24 | 64 |

FIGURA 31.3 Uma ilustração do teorema chinês do resto para  $n_1 = 5$  e  $n_2 = 13$ . Nesse exemplo,  $c_1 = 26$  e  $c_2 = 40$ . Na linha  $i$ , coluna  $j$  é mostrado o valor de  $\alpha$ , módulo 65, tal que  $(\alpha \bmod 5) = i$  e  $(\alpha \bmod 13) = j$ . Observe que a linha 0, coluna 0 contém um 0. De modo semelhante, a linha 4, coluna 12 contém o valor 64 (equivalente a -1). Tendo em vista que  $c_1 = 26$ , descer uma linha aumenta  $\alpha$  em 26. Da mesma forma,  $c_2 = 40$  significa que mover uma coluna para a direita aumenta  $\alpha$  em 40. O aumento de  $\alpha$  em 1 corresponde ao movimento em diagonal para baixo e para a direita, passando-se da parte inferior para a superior e da direita para a esquerda.

Veja na Figura 31.3 uma ilustração do teorema chinês do resto, módulo 65.

Desse modo, podemos trabalhar em módulo  $n$ , seja trabalhando diretamente em módulo  $n$  ou na representação transformada com o uso de cálculos separados de módulo  $n_i$ , conforme for conveniente. Os cálculos são totalmente equivalentes.

## Exercícios

### 31.5-1

Encontre todas as soluções para as equações  $x \equiv 4 \pmod{5}$  e  $x \equiv 5 \pmod{11}$ .

### 31.5-2

Encontre todos os inteiros  $x$  que deixam restos 1, 2, 3 quando divididos por 9, 8, 7, respectivamente.

### 31.5-3

Demonstre que, de acordo com as definições do Teorema 31.27, se  $\text{mdc}(\alpha, n) = 1$ , então

$$(\alpha^{-1} \bmod n) \leftrightarrow ((\alpha_1^{-1} \bmod n_1), (\alpha_2^{-1} \bmod n_2), \dots, (\alpha_k^{-1} \bmod n_k)).$$

### 31.5-4

De acordo com as definições do Teorema 31.27, prove que, para qualquer polinômio  $f$ , o número de raízes da equação  $f(x) \equiv 0 \pmod{n}$  é igual ao produto do número de raízes de cada uma das equações  $f(x) \equiv 0 \pmod{n_1}, f(x) \equiv 0 \pmod{n_2}, \dots, f(x) \equiv 0 \pmod{n_k}$ .

## 31.6 Potências de um elemento

Assim como é natural considerar os múltiplos de um dado elemento  $\alpha$ , módulo  $n$ , muitas vezes é natural considerar a seqüência de potências de  $\alpha$ , módulo  $n$ , onde  $\mathbb{Z}_n^*$ :

$$\alpha^0, \alpha^1, \alpha^2, \alpha^3, \dots, \tag{31.29}$$

módulo  $n$ . Indexando-se a partir de 0, o 0-ésimo valor nessa seqüência é  $\alpha^0 \bmod n = 1$  e o  $i$ -ésimo valor é  $\alpha^i \bmod n$ . Por exemplo, as potências de 3 módulo 7 são

| $i$           | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ... |
|---------------|---|---|---|---|---|---|---|---|---|---|----|----|-----|
| $3^i \bmod 7$ | 1 | 3 | 2 | 6 | 4 | 5 | 1 | 3 | 2 | 6 | 4  | 5  | ... |

enquanto as potências de 2 módulo 7 são

| $i$           | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ... |
|---------------|---|---|---|---|---|---|---|---|---|---|----|----|-----|
| $2^i \bmod 7$ | 1 | 2 | 4 | 1 | 2 | 4 | 1 | 2 | 4 | 1 | 2  | 4  | ... |

Nesta seção, seja  $\langle a \rangle$  o subgrupo de  $Z_n^*$  gerado por  $a$  por meio de multiplicação repetida, e seja  $\text{ord}_n(a)$  ( $a$  “ordem de  $a$ , módulo  $n$ ”) um valor que denota a ordem de  $a$  em  $Z_n^*$ . Por exemplo,  $\langle 2 \rangle = \{1, 2, 4\}$  em  $Z_7^*$ , e  $\text{ord}_7(2) = 3$ . Usando a definição da função phi de Euler  $\phi(n)$  como o tamanho de  $Z_n^*$  (consulte a Seção 31.3), agora traduzimos o Corolário 31.19 para a notação de  $Z_n^*$ , a fim de obter o teorema de Euler e especializá-lo para  $Z_p^*$ , onde  $p$  é primo, com o objetivo de obter o teorema de Fermat.

### Teorema 31.30 (Teorema de Euler)

Para qualquer inteiro  $n > 1$ ,

$$a^{\phi(n)} \equiv 1 \pmod{n} \text{ para todo } a \in Z_n^*.$$

### Teorema 31.31 (Teorema de Fermat)

Se  $p$  é primo, então

$$a^{p-1} \equiv 1 \pmod{p} \text{ para todo } a \in Z_p^*.$$

**Prova** Pela equação (31.20),  $\phi(n) = p - 1$  se  $p$  é primo.

Esse corolário se aplica a todo elemento em  $Z_p$  com exceção de 0, pois  $0 \notin Z_p^*$ . Porém, para todo  $a \in Z_p$ , temos  $a^p \equiv a$  se  $p$  é primo.

Se  $\text{ord}_n(g) = |Z_n^*|$ , então todo elemento em  $Z_n^*$  é uma potência de  $g$ , módulo  $n$ , e dizemos que  $g$  é uma **raiz primitiva** ou um **gerador** de  $Z_n^*$ . Por exemplo, 3 é uma raiz primitiva, módulo 7, mas 2 não é uma raiz primitiva, módulo 7. Se  $Z_n^*$  possui uma raiz primitiva, dizemos que o grupo  $Z_n^*$  é **cíclico**. Omitimos a prova do teorema a seguir, que foi provado por Niven e Zuckerman [231].

### Teorema 31.32

Os valores de  $n > 1$  para os quais  $Z_n^*$  é cíclico são  $2, 4, p^e$  e  $2p^e$ , para todos os primos  $p > 2$  e todos os inteiros positivos  $e$ .

Se  $g$  é uma raiz primitiva de  $Z_n^*$  e  $a$  é qualquer elemento de  $Z_n^*$ , então existe um  $z$  tal que  $g^z \equiv a \pmod{n}$ . Esse  $z$  é chamado **logaritmo discreto** ou **índice** de  $a$ , módulo  $n$ , para a base  $g$ ; denotamos esse valor como  $\text{ind}_{n,g}(a)$ .

### Teorema 31.33 (Teorema do logaritmo discreto)

Se  $g$  é uma raiz primitiva de  $Z_n^*$ , então a equação  $g^x \equiv g^y \pmod{n}$  é válida se e somente se a equação  $x \equiv y \pmod{\phi(n)}$  é válida.

**Prova** Primeiro, suponha que  $x \equiv y \pmod{\phi(n)}$ . Então,  $x = y + k\phi(n)$  para algum inteiro  $k$ . Assim,

$$694 \mid g^x \equiv g^{y+k\phi(n)} \pmod{n}$$

$$\begin{aligned}
&\equiv g^y \cdot (g^{\phi(n)})^k \pmod{n} \\
&\equiv g^y \cdot 1^k \pmod{n} \quad (\text{pelo teorema de Euler}) \\
&\equiv g^y \pmod{n}.
\end{aligned}$$

Reciprocamente, suponha que  $g^x \equiv g^y \pmod{n}$ . Como a seqüência de potências de  $g$  gera todo elemento de  $\langle g \rangle$  e  $|\langle g \rangle| = \phi(n)$ , o Corolário 31.18 implica que a seqüência de potências de  $g$  é periódica com período  $\phi(n)$ . Portanto, se  $g^x \equiv g^y \pmod{n}$ , então devemos ter  $x \equiv y \pmod{\phi(n)}$ . ■

Algumas vezes, o uso de logaritmos discretos pode simplificar o raciocínio sobre uma equação modular, como ilustra a prova do teorema a seguir.

### **Teorema 31.34**

Se  $p$  é um primo ímpar e  $e \geq 1$ , então a equação

$$x^2 \equiv 1 \pmod{p^e} \quad (31.30)$$

tem somente duas soluções, ou seja,  $x = 1$  e  $x = -1$ .

**Prova** Seja  $n = p^e$ . O Teorema 31.32 implica que  $Z_n^*$  tem uma raiz primitiva  $g$ . A equação (31.30) pode ser escrita como

$$(g^{\text{ind}_{n,g}(x)})^2 \equiv g^{\text{ind}_{n,g}(1)} \pmod{n}. \quad (31.31)$$

Depois de verificar que  $\text{ind}_{n,g}(1) = 0$ , observamos que o Teorema 31.33 implica que a equação (31.31) é equivalente a

$$2 \cdot \text{ind}_{n,g}(x) \equiv 0 \pmod{\phi(n)}. \quad (31.32)$$

Para resolver essa equação para a incógnita  $\text{ind}_{n,g}(x)$ , aplicamos os métodos da Seção 31.4. Pela equação (31.19), temos  $\phi(n) = p^e(1 - 1/p) = (p - 1)p^{e-1}$ . Sendo  $d = ???(2, \phi(n)) = ??(2, (p - 1)p^{e-1}) = 2$ , e observando-se que  $d \mid 0$ , descobrimos a partir do Teorema 31.24 que a equação (31.32) tem exatamente  $d = 2$  soluções. Então, a equação (31.30) tem exatamente 2 soluções, que são  $x = 1$  e  $x = -1$ , por inspeção.

Um número  $x$  é uma **raiz quadrada não trivial de 1, módulo n**, se ele satisfaz à equação  $x^2 \equiv 1 \pmod{n}$ , mas  $x$  não é equivalente a nenhuma das duas raízes quadradas “triviais”: 1 ou  $-1$ , módulo  $n$ . Por exemplo, 6 é uma raiz quadrada não trivial de 1, módulo 35. O corolário a seguir para o Teorema 31.34 será usado na prova de correção do procedimento de teste do caráter primo de Miller-Rabin na Seção 31.8.

### **Corolário 31.35**

Se existe uma raiz quadrada não trivial de 1, módulo  $n$ , então  $n$  é múltiplo.

**Prova** Pela contraposição do Teorema 31.34, se existe uma raiz quadrada não trivial de 1, módulo  $n$ , então  $n$  não pode ser um primo ímpar ou uma potência de um primo ímpar. Se  $x^2 \equiv 1 \pmod{2}$ , então  $x \equiv 1 \pmod{2}$  e assim todas as raízes quadradas de 1, módulo 2, são triviais. Portanto,  $n$  não pode ser primo. finalmente, devemos ter  $n > 1$  para que uma raiz quadrada não trivial de 1 exista. Então,  $n$  tem de ser múltiplo. ■

## Elevação a potências com elevação ao quadrado repetida

Uma operação que ocorre com freqüência em cálculos da teoria dos números é a elevação de um número a uma potência módulo outro número, também conhecida como **exponenciação modular**. Mais precisamente, gostaríamos de encontrar um modo eficiente para calcular  $a^b \text{ mod } n$ , onde  $a$  e  $b$  são inteiros não negativos e  $n$  é um inteiro positivo. A exponenciação modular é uma operação essencial em muitas rotinas de teste do caráter primo e no sistema de criptografia de chave pública RSA. O método de **elevação ao quadrado repetida** resolve esse problema de forma eficiente, utilizando a representação binária de  $b$ .

Seja  $\langle b_k, b_{k-1}, \dots, b_1, b_0 \rangle$  a representação binária de  $b$ . (Isto é, a representação binária tem  $k+1$  bits de comprimento,  $b_k$  é o bit mais significativo e  $b_0$  é o bit menos significativo.) O procedimento a seguir calcula  $a^c \text{ mod } n$  à medida que  $c$  é aumentado por duplicações e incrementos, desde 0 até  $b$ .

**MODULAR-EXPONENTIATION( $a, b, n$ )**

```

1  $c \leftarrow 0$ 
2  $d \leftarrow 1$ 
3 seja  $\langle b_k, b_{k-1}, \dots, b_1, b_0 \rangle$  a representação binária de  $b$ 
4 for  $i \leftarrow k$  downto 0
5   do  $c \leftarrow 2c$ 
6    $d \leftarrow (d \cdot d) \text{ mod } n$ 
7   if  $b_i = 1$ 
8     then  $c \leftarrow c + 1$ 
9      $d \leftarrow (d \cdot a) \text{ mod } n$ 
10 return  $d$ 

```

O uso essencial da elevação ao quadrado em cada iteração explica o nome “elevação ao quadrado repetida”. Como exemplo, para  $a = 7$ ,  $b = 560$  e  $n = 561$ , o algoritmo calcula a seqüência de valores módulo 561 mostrada na Figura 31.4; a seqüência de expoentes empregados é mostrada na linha da tabela identificada por  $c$ .

A variável  $c$  não é realmente exigida pelo algoritmo, mas foi incluída para fins explicativos; o algoritmo preserva o loop invariante de duas partes a seguir:

| $i$   | 9 | 8  | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-------|---|----|-----|-----|-----|-----|-----|-----|-----|-----|
| $b_i$ | 1 | 0  | 0   | 0   | 1   | 1   | 0   | 0   | 0   | 0   |
| $c$   | 1 | 2  | 4   | 8   | 17  | 35  | 70  | 140 | 280 | 560 |
| $d$   | 7 | 49 | 157 | 526 | 160 | 241 | 298 | 166 | 67  | 1   |

FIGURA 31.4 Os resultados de MODULAR-EXPONENTIATION quando se calcula  $a^b \text{ (mod } n)$ , onde  $a = 7$ ,  $b = 560 = \langle 1000110000 \rangle$ , e  $n = 561$ . Os valores são mostrados após cada execução do loop **for**. O resultado final é 1

Imediatamente antes de cada iteração do loop **for** das linhas 4 a 9:

1. O valor de  $c$  é igual ao prefixo  $\langle b_k, b_{k-1}, \dots, b_{i+1} \rangle$  da representação binária de  $b$ .
2.  $d = a^c \text{ mod } n$ .

Usamos este loop invariante como a seguir:

**Inicialização:** Inicialmente,  $i = k$ , de forma que o prefixo  $\langle b_k, b_{k-1}, \dots, b_{i+1} \rangle$  está vazio, o que corresponde a  $c = 0$ . Além disso,  $d = 1 = a^0 \text{ mod } n$ .

**Manutenção:** Sejam  $c'$  e  $d'$  os valores de  $c$  e  $d$  no fim de uma iteração do loop for, e portanto os valores antes da próxima iteração. Cada iteração atualiza  $c' \leftarrow 2c$  (se  $b_i = 0$ ) ou  $c' \leftarrow 2c + 1$  (se  $b_i = 1$ ), de forma que  $c$  estará correto antes da próxima iteração. Se  $b_i = 0$ , então  $d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{2c} \bmod n = \text{mod } n = a^c \bmod n$ . Se  $b_i = 1$ , então  $d' = d^2 a \bmod n = (a^c)^2 a \bmod n = a^{2c+1} \bmod n = a^c \bmod n$ . Em qualquer caso,  $d = a^c \bmod n$  antes da próxima iteração.

**Término:** No término,  $i = -1$ . Desse modo,  $c = b$ , pois  $c$  tem o valor do prefixo  $\langle b_k, b_{k-1}, \dots, b_0 \rangle$  da representação binária de  $b$ . Conseqüentemente  $d = a^c \bmod n = a^b \bmod n$ .

Se as entradas  $a$ ,  $b$  e  $n$  são números de  $\beta$  bits, então o número total de operações aritméticas exigidas é  $O(\beta)$ , e o número total de operações de bits exigidas é  $O(\beta^3)$ .

## Exercícios

### 31.6-1

Desenhe uma tabela mostrando a ordem de todo elemento em  $Z_{11}^*$ . Escolha a menor raiz primitiva  $g$  e calcule uma tabela fornecendo  $\text{ind}_{n,g}(x)$  para todo  $x \in Z_{11}^*$ .

### 31.6-2

Forneça um algoritmo de exponenciação modular que examine os bits de  $b$  da direita para a esquerda, em vez de examiná-los da esquerda para a direita.

### 31.6-3

Supondo que você conheça  $\phi(n)$ , explique como calcular  $a^{-1} \bmod n$  para qualquer  $a \in Z_n^*$  usando o procedimento MODULAR-EXPONENTIATION.

## 31.7 O sistema de criptografia de chave pública RSA

Um sistema de criptografia de chave pública pode ser usado para codificar mensagens enviadas entre dois participantes de uma comunicação, de forma que um intruso que escute as mensagens codificadas não possa decodificá-las. Um sistema de criptografia de chave pública também permite que um dos participantes acrescente uma “assinatura digital” impossível de forjar ao final de uma mensagem eletrônica. Tal assinatura é a versão eletrônica de uma assinatura manuscrita em um documento de papel. Ela pode ser conferida com facilidade por qualquer pessoa, não pode ser forjada por ninguém, e ainda perde sua validade se qualquer bit da mensagem for alterado. Portanto, ela fornece autenticação, tanto da identidade do signatário quanto do conteúdo da mensagem assinada. É a ferramenta perfeita para assinar eletronicamente contratos de negócios, cheques eletrônicos, pedidos eletrônicos de compras e outras comunicações eletrônicas que devam ser autenticadas.

O sistema de criptografia de chave pública RSA se baseia na diferença drástica entre a facilidade de encontrar números primos grandes e a dificuldade de fatorar o produto de dois números primos grandes. A Seção 31.8 descreve um procedimento eficiente para encontrar números primos grandes, e a Seção 31.9 discute o problema de fatorar inteiros maiores.

## Sistemas de criptografia de chave pública

Em um sistema de criptografia de chave pública, cada participante tem ao mesmo tempo uma **chave pública** e uma **chave secreta**. Cada chave é um item de informações. Por exemplo, no sistema de criptografia RSA, cada chave consiste em um par de inteiros. Os participantes “Alice” e “Bob” são tradicionalmente usados em exemplos de criptografia; denotamos suas chaves públicas e secretas como  $P_A, S_A$  para Alice e  $P_B, S_B$  para Bob.

Cada participante cria suas próprias chaves pública e secreta. Cada um mantém em segredo sua chave secreta, mas pode revelar sua chave pública a qualquer pessoa, ou até mesmo publi-

cá-la. De fato, com freqüência é conveniente supor que a chave pública de qualquer pessoa está disponível em um diretório público, de forma que qualquer participante possa obter facilmente a chave pública de qualquer outro participante.

As chaves pública e secreta especificam funções que podem ser aplicadas a qualquer mensagem. Seja  $\mathcal{D}$  o conjunto de mensagens permitidas. Por exemplo,  $\mathcal{D}$  poderia ser o conjunto de todas as seqüências de bits de comprimento finito. Exigimos que as chaves pública e secreta especifiquem funções de um para um de  $\mathcal{D}$  para ele próprio. A função correspondente à chave pública de Alice,  $P_A()$ , é denotada por  $P_A()$  e a função correspondente à sua chave secreta,  $S_A$ , é denotada por  $S_A()$ . As funções  $P_A()$  e  $S_A()$  são portanto permutações de  $\mathcal{D}$ . Supomos que as funções  $P_A()$  e  $S_A()$  possam ser calculadas de forma eficiente, sendo dada a chave  $P_A$  ou  $S_A$  correspondente.

As chaves pública e secreta para qualquer participante formam um “par coincidente”, pois elas especificam funções que são inversas uma da outra. Isto é,

$$M = S_A(P_A(M)), \quad (31.33)$$

$$M = P_A(S_A(M)) \quad (31.34)$$

para qualquer mensagem  $M \in \mathcal{D}$ . A transformação de  $M$  com as duas chaves  $P_A$  e  $S_A$ , sucessivamente, em qualquer ordem, gera de novo a mensagem  $M$ .

Em um sistema de criptografia de chave pública, é essencial que ninguém, exceto Alice, possa calcular a função  $S_A()$  em qualquer período de tempo prático. A privacidade da correspondência que é codificada e enviada a Alice e a autenticidade das assinaturas digitais de Alice se baseiam na hipótese de que apenas Alice é capaz de calcular  $S_A()$ . Essa exigência é o motivo pelo qual Alice mantém  $S_A$  secreta; se não o fizer, ela perderá sua exclusividade, e o sistema de criptografia não poderá proporcionar a ela recursos exclusivos. A hipótese de que apenas Alice possa calcular  $S_A()$  deve se manter válida mesmo que todo mundo conheça  $P_A$  e possa calcular  $P_A()$ , a função inversa de  $S_A()$ , de modo eficiente. A principal dificuldade em projetar um sistema de criptografia de chave pública funcional está em compreender como criar um sistema no qual possamos revelar uma transformação  $P_A()$ , sem revelar através dela o modo de calcular a transformação inversa correspondente  $S_A()$ .

Em um sistema de criptografia de chave pública, a codificação funciona da maneira mostrada na Figura 31.5. Suponha que Bob deseje enviar a Alice uma mensagem  $M$  codificada de tal forma que ela pareça ininteligível para um intruso. O cenário para o envio da mensagem é dado a seguir.

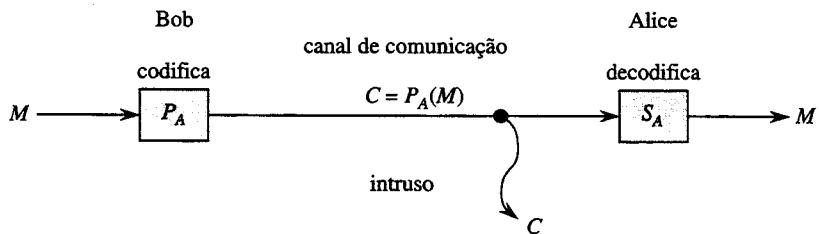
- Bob obtém a chave pública de Alice,  $P_A$  (de um diretório público ou diretamente de Alice).
- Bob calcula o **texto cifrado**  $C = P_A(M)$  correspondente à mensagem  $M$  e envia  $C$  a Alice.
- Quando Alice recebe o texto cifrado  $C$ , ela aplica sua chave secreta  $S_A$ , a fim de recuperar a mensagem original:  $M = S_A(C)$ .

Pelo fato de  $S_A()$  e  $P_A()$  serem funções inversas, Alice pode calcular  $M$  a partir de  $C$ . Como apenas Alice é capaz de calcular  $S_A()$ , somente ela pode calcular  $M$  a partir de  $C$ . A codificação de  $M$  usando  $P_A()$  protegeu  $M$  contra a revelação por qualquer pessoa, exceto Alice.

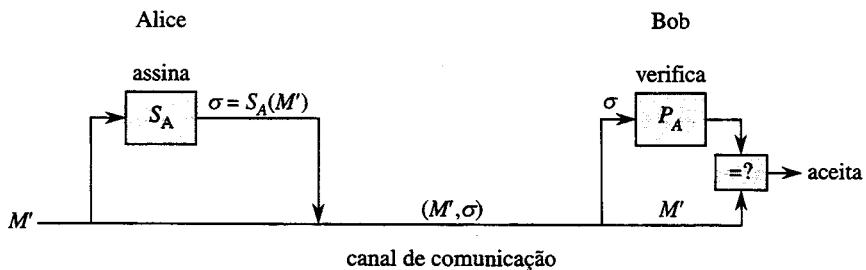
As assinaturas digitais oferecem uma facilidade de implementação semelhante dentro de nossa formulação de um sistema de criptografia de chave pública. (Observamos que existem outras maneiras de abordar o problema de construir assinaturas digitais, o que não examinaremos aqui.) Suponha agora que Alice deseje enviar a Bob uma mensagem de resposta  $M'$  com assinatura digital. O cenário da assinatura digital tem as características mostradas na Figura 31.6.

- Alice calcula sua **assinatura digital**  $\sigma$  para a mensagem  $M'$  usando sua chave secreta  $S_A$  e a equação  $\sigma = S_A(M')$ .

- Alice envia o par mensagem/assinatura  $(M', \sigma)$  a Bob.
- Ao receber  $(M', \sigma)$ , Bob pode confirmar que ela foi enviada por Alice com a utilização da chave pública de Alice, verificando a equação  $M' = P_A(\sigma)$ . (Presumivelmente,  $M'$  contém o nome de Alice, e assim Bob sabe qual chave pública deve usar.) Se a equação é válida, então Bob conclui que a mensagem  $M'$  foi realmente assinada por Alice. Se a equação não é válida, Bob conclui que a mensagem  $M'$  ou a assinatura digital  $\sigma$  foi danificada por erros de transmissão, ou então que o par  $(M', \sigma)$  é uma tentativa de falsificação.



**FIGURA 31.5** Codificação em um sistema de chave pública. Bob codifica a mensagem  $M$  usando a chave pública  $P_A$  de Alice e transmite o texto cifrado resultante  $C = P_A(M)$  a Alice. Um intruso que capturar o texto cifrado transmitido não obterá nenhuma informação sobre  $M$ . Alice recebe  $C$  e a decodifica usando sua chave secreta para obter a mensagem original  $M = S_A(C)$



**FIGURA 31.6** Assinaturas digitais em um sistema de chave pública. Alice assina a mensagem  $M'$  anexando sua assinatura digital  $\sigma$  à mensagem. Ela transmite o par mensagem/assinatura  $(M', \sigma)$  a Bob, que a verifica, conferindo a equação  $\sigma$ . Se a equação é válida, ele aceita  $(M', \sigma)$  como uma mensagem que foi assinada por Alice

Pelo fato de uma assinatura digital fornecer tanto autenticação da identidade do signatário quanto a autenticação do conteúdo da mensagem assinada, ela é análoga a uma assinatura manuscrita no final de um documento escrito.

Uma propriedade importante de uma assinatura digital é que ela pode ser verificada por qualquer pessoa que tenha acesso à chave pública do signatário. Uma mensagem assinada pode ser verificada por uma parte, e depois repassada a outras partes que também têm a possibilidade de verificar a assinatura. Por exemplo, a mensagem poderia ser um cheque eletrônico de Alice para Bob. Depois de verificar a assinatura de Alice no cheque, Bob pode entregar o cheque a seu banco, o qual poderá então verificar também a assinatura e efetuar a transferência de fundos apropriada.

Observamos que uma mensagem assinada não está codificada; a mensagem está “às claras” e não é protegida contra revelação. Pela composição dos protocolos anteriores para codificação e para assinaturas, podemos criar mensagens que estejam ao mesmo tempo assinadas e codificadas. Primeiro, o signatário anexa sua assinatura digital à mensagem, e depois codifica o par mensagem/assinatura resultante com a chave pública do destinatário pretendido. O destinatário decodifica a mensagem recebida com sua chave secreta, a fim de obter tanto a mensagem original quanto sua assinatura digital. Assim, ele pode verificar a assinatura usando a chave pública do

signatário. O processo combinado correspondente com o uso de sistemas de papel é assinar o documento em papel, e depois lacrar o documento dentro de um envelope de papel que será aberto apenas pelo destinatário pretendido.

## O sistema de criptografia RSA

No sistema de criptografia de chave pública RSA, um participante cria suas chaves pública e secreta com o procedimento a seguir.

1. Selecione ao acaso dois números primos grandes  $p$  e  $q$ , tais que  $p \neq q$ . Os primos  $p$  e  $q$  podem ter, digamos, 512 bits cada um.
2. Calcule  $n$  pela equação  $n = pq$ .
3. Selecione um inteiro ímpar pequeno  $e$  tal que ele seja primo em relação a  $\phi(n)$  que, pela equação (31.19), é igual a  $(p - 1)(q - 1)$ .
4. Calcule  $d$  como o inverso multiplicativo de  $e$ , módulo  $\phi(n)$ . O Corolário 31.26 garante que  $d$  existe e é definido de forma exclusiva. Podemos usar a técnica da Seção 31.4 para calcular  $d$ , dados  $e$  e  $\phi(n)$ .)
5. Publique o par  $P = (e, n)$  como sua **chave pública RSA**.
6. Guarde em segredo o par  $S = (d, n)$  como sua **chave secreta RSA**.

Por esse esquema, o domínio  $\mathcal{D}$  é o conjunto  $\mathbf{Z}_n$ . A transformação de uma mensagem  $M$  associada a uma chave pública  $P = (e, n)$  é

$$P(M) = M^e \pmod{n} . \quad (31.35)$$

A transformação de um texto cifrado  $C$  associado a uma chave secreta  $S = (d, n)$  é

$$S(C) = C^d \pmod{n} . \quad (31.36)$$

Essas equações se aplicam tanto à codificação quanto a assinaturas. Para criar uma assinatura, o signatário aplica sua chave secreta à mensagem a ser assinada, em lugar de um texto cifrado. Para verificar uma assinatura, a chave pública do signatário é aplicada a ela, em lugar de ser aplicada a uma mensagem que deverá ser codificada.

As operações de chave pública e chave secreta podem ser implementadas usando-se o procedimento MODULAR-EXPONENTIATION descrito na Seção 31.6. Para analisar o tempo de execução dessas operações, suponha que a chave pública  $(e, n)$  e a chave secreta  $(d, n)$  satisfaçam a  $\lg e = O(1)$ ,  $\lg d \leq \beta$  e  $\lg n \leq \beta$ . Então, a aplicação de uma chave pública exige  $O(1)$  multiplicações modulares e usa  $O(\beta^2)$  operações de bits. A aplicação de uma chave secreta exige  $O(\beta)$  multiplicações modulares, empregando-se  $O(\beta^3)$  operações de bits.

### **Teorema 31.36 (Correção do RSA)**

As equações do RSA (31.35) e (31.36) definem transformações inversas de  $\mathbf{Z}_n$  que satisfazem às equações (31.33) e (31.34).

**Prova** A partir das equações (31.35) e (31.36) temos que, para qualquer  $M \in \mathbf{Z}_n$ ,

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n} .$$

Tendo em vista que  $e$  e  $d$  são inversos multiplicativos módulo  $\phi(n) = (p - 1)(q - 1)$ ,

$$ed = 1 + k(p - 1)(q - 1)$$

$$\begin{aligned}
 M^{ed} &\equiv M (M^{p-1})^{k(q-1)} \pmod{p} \\
 &\equiv M (1)^{k(q-1)} \pmod{p} \quad (\text{pelo Teorema 31.31}) \\
 &\equiv M \pmod{p}
 \end{aligned}$$

Além disso,  $M^{ed} \equiv M \pmod{p}$  se  $M \equiv 0 \pmod{p}$ . Desse modo,

$$M^{ed} \equiv M \pmod{p}$$

para todo  $M$ . De modo semelhante,

$$M^{ed} \equiv M \pmod{q}$$

para todo  $M$ . Portanto, de acordo com o Corolário 31.29 para o teorema chinês do resto,

$$M^{ed} \equiv M \pmod{n}$$

para todo  $M$ .

A segurança do sistema de criptografia RSA se baseia em grande parte na dificuldade de fatorar inteiros maiores. Se um adversário pode fatorar o módulo  $n$  em uma chave pública, então ele pode derivar a chave secreta a partir da chave pública, usando o conhecimento dos fatores  $p$  e  $q$  da mesma forma que o criador da chave pública os utilizou. Assim, se é fácil fatorar inteiros maiores, então é fácil romper o sistema de criptografia RSA. A declaração recíproca de que, se é difícil fatorar inteiros maiores, então é difícil romper o RSA, não foi provada. Porém, depois de uma década de pesquisas, não foi encontrado nenhum método mais fácil para romper o sistema de criptografia de chave pública RSA do que para fatorar o módulo  $n$ . Além disso, como veremos na Seção 31.9, a fatoração de inteiros maiores é surpreendentemente difícil. Selecionando aleatoriamente e multiplicando dois primos de 512 bits, é possível criar uma chave pública que não poderá ser “rompida” em qualquer período viável de tempo com tecnologia atual. Na ausência de uma inovação fundamental no projeto de algoritmos da teoria dos números, e quando implementado com cuidado de acordo com padrões recomendados, o sistema de criptografia RSA é capaz de fornecer um alto grau de segurança em aplicações.

Contudo, a fim de alcançar segurança com o sistema de criptografia RSA, é aconselhável trabalhar com inteiros que têm várias centenas de bits de comprimento, a fim de resistir a possíveis avanços na arte da fatoração. No momento em que escrevemos (em 2001), os módulos RSA estavam comumente na faixa de 768 a 2048 bits. Para criar módulos de tais tamanhos, devemos ser capazes de encontrar primos grandes de modo eficiente. Esse problema é examinado na Seção 31.8.

Por eficiência, o RSA é usado com freqüência em um modo “híbrido” ou de “gerenciamento de chaves”, com sistemas de criptografia rápidos de chaves não públicas. Com um sistema desse tipo, as chaves de codificação e decodificação são idênticas. Se Alice desejar enviar uma longa mensagem  $M$  a Bob em particular, ela selecionará uma chave aleatória  $K$  para o sistema de criptografia de chave não pública rápido e codificará  $M$  usando  $K$ , obtendo assim o texto cifrado  $C$ . Nesse caso,  $C$  é tão longo quanto  $M$ , mas  $K$  é bastante curto. Em seguida, ela codifica  $K$  usando a chave pública RSA de Bob. Tendo em vista que  $K$  é curta, o cálculo de  $P_B(K)$  é rápido (muito mais rápido que o cálculo de  $P_B(M)$ ). Depois, ela transmite  $(C, P_B(K))$  a Bob, que decodifica  $P_B(K)$  para obter  $K$  e então usa  $K$  para decodificar  $C$ , obtendo  $M$ .

Uma abordagem híbrida semelhante é usada com freqüência para criar de forma eficiente assinaturas digitais. Nessa abordagem, o RSA é combinado com uma **função hash resistente a colisões**  $b$  – uma função fácil de calcular, mas para a qual é inviável em termos computacionais encontrar duas mensagens  $M$  e  $M'$  tais que  $b(M) = b(M')$ . O valor  $b(M)$  é uma “impressão digital”

curta (digamos, de 160 bits) da mensagem  $M$ . Se Alice desejar assinar uma mensagem  $M$ , primeiro ela aplicará  $b$  a  $M$  para obter a impressão digital  $b(M)$ , e depois a usará para assinar com sua chave secreta. Ela envia  $(M, S_A(b(M)))$  a Bob como sua versão assinada de  $M$ . Bob pode verificar a assinatura, calculando  $b(M)$  e verificando que  $P_A$  aplicada a  $S_A(b(M))$  da forma como foi recebida é igual a  $b(M)$ . Pelo fato de ninguém poder criar duas mensagens com a mesma impressão digital, é impossível em termos computacionais alterar uma mensagem assinada e preservar a validade da assinatura.

Finalmente, notamos que o uso de *certificados* torna muito mais fácil a distribuição de chaves públicas. Por exemplo, suponha que exista uma “autoridade confiável”  $T$  cuja chave pública seja conhecida por todos. Alice pode obter de  $T$  uma mensagem assinada (seu certificado) declarando que “a chave pública de Alice é  $P_A$ ”. Esse certificado tem “autenticação própria”, pois todo mundo conhece  $P_T$ . Alice pode incluir seu certificado em suas mensagens assinadas, de forma que o destinatário tenha a chave pública de Alice imediatamente disponível, e assim possa verificar sua assinatura. Tendo em vista que a chave pública de Alice foi assinada por  $T$ , o destinatário sabe que a chave de Alice é realmente de Alice.

## Exercícios

### 31.7-1

Considere um conjunto de chaves RSA com  $p = 11$ ,  $q = 29$ ,  $n = 319$  e  $e = 3$ . Que valor de  $d$  deve ser usado na chave secreta? Qual é a codificação da mensagem  $M = 100$ ?

### 31.7-2

Prove que, se o expoente público  $e$  de Alice é 3 e um adversário obtém o expoente secreto  $d$  de Alice, então o adversário pode fatorar o módulo  $n$  de Alice em tempo polinomial no número de bits em  $n$ . (Embora não seja convidado a prová-lo, talvez você esteja interessado em saber que esse resultado permanece verdadeiro mesmo-se a condição  $e = 3$  é removida. Consulte Miller [221].)

### 31.7-3 \*

Prove que o RSA é multiplicativo, no sentido de que

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n}.$$

Use esse fato para provar que, se um adversário tivesse um procedimento que fosse capaz de decodificar de forma eficiente 1% das mensagens escolhidas aleatoriamente a partir de  $\mathbb{Z}_n$  e codificadas com  $P_A$ , então ele poderia empregar um algoritmo probabilístico para decodificar toda mensagem codificada com  $P_A$ , com probabilidade alta.

## \*31.8 Como testar o caráter primo

Nesta seção, consideraremos o problema de encontrar primos grandes. Começamos com uma discussão da densidade dos primos, prosseguindo até examinarmos uma abordagem plausível (embora incompleta) para testar o caráter primo, e depois apresentaremos um teste aleatório efetivo do caráter primo, devido a Miller e Rabin.

### A densidade de números primos

Para muitas aplicações (como a criptografia), precisamos encontrar primos grandes “aleatórios”. Felizmente, primos grandes não são muito raros, de forma que não é muito demorado testar inteiros aleatórios do tamanho apropriado até encontrar um primo. A função de distribuição de primos  $\pi(n)$  especifica o número de primos menores que ou iguais a  $n$ . Por exemplo,  $\pi(10) = 4$ , pois existem 4 números primos menores que ou iguais a 10, ou seja, 2, 3, 5 e 7. O teorema dos números primos nos dá uma aproximação útil para  $\pi(n)$ .

### Teorema 31.37 (Teorema dos números primos)

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n/\ln n} = 1.$$

A aproximação  $n/\ln n$  fornece estimativas razoavelmente precisas de  $\pi(n)$ , mesmo para  $n$  pequeno. Por exemplo, ela erra por menos de 6% em  $n = 10^9$ , onde  $\pi(n) = 50.847.478$  e  $n/\ln n \approx 48.254.942$ . (Para alguém que segue a teoria dos números,  $10^9$  é um número pequeno.)

Podemos usar o teorema dos números primos para fazer uma estimativa da probabilidade de um inteiro  $n$  escolhido aleatoriamente ser primo como  $1/\ln n$ . Desse modo, poderia ser necessário examinar aproximadamente  $\ln n$  inteiros escolhidos de forma aleatória perto de  $n$  para encontrar um primo que tivesse o mesmo comprimento de  $n$ . Por exemplo, encontrar um primo de 512 bits poderia exigir testar o caráter primo de aproximadamente  $\ln 2^{512} \approx 355$  números de 512 bits escolhidos aleatoriamente. (Esse valor pode ser reduzido à metade, escolhendo-se apenas inteiros ímpares.)

No restante desta seção, vamos considerar o problema de determinar se um inteiro ímpar grande  $n$  é ou não um primo. Por conveniência de notação, vamos supor que  $n$  tenha a decomposição em fatores primos

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}, \quad (31.37)$$

onde  $r \geq 1$ ,  $p_1, p_2, \dots, p_r$  são os fatores primos de  $n$ , e  $e_1, e_2, \dots, e_r$  são inteiros positivos. É claro que  $n$  é primo se e somente se  $r = 1$  e  $e_1 = 1$ .

Uma abordagem simples para o problema de testar o caráter primo é a *divisão experimental*. Tentamos dividir  $n$  por cada inteiro  $2, 3, \dots, \lfloor \sqrt{n} \rfloor$ . (Mais uma vez, inteiros pares maiores que 2 podem ser ignorados.) É fácil ver que  $n$  é primo se e somente se nenhum dos divisores experimentais divide  $n$ . Supondo que cada divisão experimental demore um tempo constante, o tempo de execução do pior caso é  $\Theta(\sqrt{n})$ , que é exponencial no comprimento de  $n$ . (Lembre-se de que, se  $n$  for codificado em binário com o uso de  $\beta$  bits, então  $\beta = \sqrt{n}$ , e assim  $\Theta(2^{\beta/2})$ .) Desse modo, a divisão experimental funciona bem apenas se  $n$  é muito pequeno ou se por acaso ele tem um fator primo pequeno. Quando funciona, a divisão experimental tem a vantagem não apenas de determinar se  $n$  é primo ou múltiplo mas também de determinar um dos fatores primos de  $n$ , caso  $n$  seja múltiplo.

Nesta seção, estamos interessados apenas em descobrir se um dado número  $n$  é primo; se  $n$  for múltiplo, não nos preocuparemos em encontrar seus fatores primos. Como veremos na Seção 31.9, o cálculo dos fatores primos de um número é dispendioso em termos computacionais. Talvez seja surpreendente o fato de ser muito mais fácil saber se um dado número é primo que descobrir os fatores primos do número se ele não for primo.

### Teste do caráter pseudoprímo

Agora, vamos considerar um método para testar o caráter primo que “quase funciona” e de fato é bom o bastante para muitas aplicações práticas. Um refinamento desse método que remove o pequeno defeito será apresentado mais adiante. Seja  $Z_n^+$  a representação dos elementos não nulos de  $Z_n$ :

$$Z_n^+ = \{1, 2, \dots, n - 1\}.$$

Se  $n$  é primo, então  $Z_n^+ = Z_n^*$ .

Dizemos que  $n$  é um **pseudoprímo de base  $a$**  se  $n$  é múltiplo e

$$a^{n-1} \equiv 1 \pmod{n}.$$

O teorema de Fermat (Teorema 31.31) implica que, se  $n$  é primo, então  $n$  satisfaz à equação (31.38) para todo  $a$  em  $\mathbb{Z}_n^+$ . Desse modo, se pudermos encontrar qualquer  $a \in \mathbb{Z}_n^+$  tal que  $n$  não satisfaça à equação (31.38), então  $n$  certamente será múltiplo. De modo surpreendente, a recíproca *quase* é válida, e assim esse critério forma um teste quase perfeito do caráter primo. Efetuamos testes para ver se  $n$  satisfaz à equação (31.38) para  $a = 2$ . Se não, declaramos  $n$  como múltiplo. Caso contrário, fornecemos como saída um palpite de que  $n$  é primo (quando, de fato, tudo o que sabemos é que  $n$  é primo ou um pseudoprímo de base 2).

O procedimento a seguir pretende dessa maneira verificar o caráter primo de  $n$ . Ele utiliza o procedimento MODULAR-EXPONENTIATION da Seção 31.6. A entrada  $n$  é considerada um inteiro ímpar maior que 2.

#### PSEUDOPRIME( $n$ )

```

1 if MODULAR-EXPONENTIATION(2,  $n - 1$ ,  $n$ )  $\not\equiv 1 \pmod{n}$ 
2 then return COMPOSITE           ▷ Definitivamente.
3 else return PRIME              ▷ Esperamos!

```

Esse procedimento pode produzir erros, mas apenas de um tipo. Isto é, se ele afirma que  $n$  é múltiplo, então ele sempre está correto. Porém, se afirma que  $n$  é primo, então ele comete um erro apenas se  $n$  é um pseudoprímo de base 2.

Com que freqüência esse procedimento erra? De forma surpreendente, apenas raras vezes. Existem somente 22 valores de  $n$  menores que 10.000 para os quais ele erra; os quatro primeiros desses valores são 341, 561, 645 e 1105. Podemos mostrar que a probabilidade de que esse programa cometa um erro em um número de  $\beta$  bits escolhido aleatoriamente chegue a zero, à medida que  $\beta \rightarrow \infty$ . Utilizando estimativas mais precisas criadas por Pomerance [244] do número de pseudoprimos de base 2 de um dado tamanho, podemos avaliar que um número de 512 bits escolhido aleatoriamente que seja considerado primo pelo procedimento anterior tem menos de uma chance em  $10^{20}$  de ser um pseudoprímo de base 2, e um número de 1024 bits escolhido aleatoriamente que é chamado primo tem menos de uma chance em  $10^{41}$  de ser um pseudoprímo de base 2. Então, se estiver apenas tentando encontrar um número primo grande para alguma aplicação, para todos os propósitos práticos você quase nunca estará errado se escolher números grandes ao acaso até um deles fazer PSEUDOPRIME fornecer a saída PRIME. Porém, quando os números cujo caráter primo está sendo testado não são escolhidos aleatoriamente, precisamos de uma abordagem melhor para testar o caráter primo. Como veremos, um pouco mais de inteligência e alguma aleatoriedade produzirá uma rotina de teste do caráter primo que funcionará bem sobre todas as entradas.

Infelizmente, não podemos eliminar todos os erros apenas verificando a equação (31.38) para um segundo número base, digamos  $a = 3$ , porque existem inteiros múltiplos  $n$  que satisfazem à equação (31.38) para todo  $a \in \mathbb{Z}_n^*$ . Esses inteiros são conhecidos como *números de Carmichael*. Os três primeiros números de Carmichael são 561, 1105 e 1729. Os números de Carmichael são extremamente raros; por exemplo, existem apenas 255 deles menores que 100.000.000. O Exercício 31.8-2 ajuda a explicar por que eles são tão raros.

Em seguida, mostraremos como melhorar nosso teste do caráter primo de forma que ele não seja enganado por números de Carmichael.

### O teste aleatório do caráter primo de Miller-Rabin

O teste do caráter primo de Miller-Rabin supera os problemas do teste simples PSEUDOPRIME com duas modificações:

- Ele experimenta diversos valores base a escolhidos aleatoriamente, em vez de apenas um valor base.
- Enquanto calcula cada exponenciação modular, ele observa se uma raiz quadrada não trivial de 1, módulo  $n$ , é descoberta. Nesse caso, ele pára e fornece COMPOSITE como saída. O Corolário 31.35 justifica a detecção de múltiplos feita dessa maneira.

O pseudocódigo para o teste do caráter primo de Miller-Rabin é dado a seguir. A entrada  $n > 2$  é o número ímpar cujo caráter primo será testado, e  $s$  é o número de valores base escolhidos aleatoriamente a partir de  $Z_n^+$  a serem experimentados. O código utiliza o gerador de número aleatórios RANDOM da Seção 5.1: RANDOM(1,  $n - 1$ ) retorna um inteiro  $a$  escolhido aleatoriamente que satisfaz  $1 \leq a \leq n - 1$ . O código emprega um procedimento auxiliar WITNESS tal que WITNESS( $a, n$ ) é TRUE se e somente se  $a$  é uma “testemunha” do caráter múltiplo de  $n$  – ou seja, se é possível usar  $a$  para provar (de um modo que veremos em breve) que  $n$  é múltiplo. O teste WITNESS( $a, n$ ) é uma extensão, embora mais eficiente, do teste

$$a^{n-1} \not\equiv 1 \pmod{n}$$

que formava a base (usando-se  $a = 2$ ) para PSEUDOPRIME. Primeiro, apresentaremos e justificaremos a construção de WITNESS, e depois mostraremos como ele é utilizado no teste do caráter primo de Miller-Rabin. Seja  $n - 1 = 2^t u$ , onde  $t \geq 1$  e  $u$  é ímpar; isto é, a representação binária de  $n - 1$  é a representação binária do inteiro ímpar  $u$  seguido por exatamente  $t$  zeros. Então,  $a^{n-1} \equiv (a^u)^{2^t}$ , de forma que podemos calcular  $a^{n-1} \pmod{n}$  calculando primeiro  $a^u \pmod{n}$ , e depois elevando o resultado ao quadrado  $t$  vezes sucessivamente.

#### WITNESS( $a, n$ )

```

1 seja  $n - 1 = 2^t u$ , onde  $t \geq 1$  e  $u$  é ímpar
2  $x_0 \leftarrow \text{MODULAR-EXPONENTIATION}(a, u, n)$ 
3 for  $i \leftarrow 1$  to  $t$ 
4   do  $x_i \leftarrow x_{i-1}^2$ 
5   if  $x_i = 1$  e  $x_{i-1} \neq 1$  e  $x_{i-1} \neq n - 1$ 
6     then return TRUE
7 if  $x_t \neq 1$ 
8 then return TRUE
9 return FALSE

```

Esse pseudocódigo para WITNESS calcula  $a^{n-1} \pmod{n}$ , calculando primeiro o valor  $x_0 = a^u \pmod{n}$  na linha 2, e depois elevando o resultado ao quadrado  $t$  vezes em uma linha no loop for das linhas 3 a 6. Por indução em  $i$ , a seqüência  $x_0, x_1, \dots, x_t$  de valores calculados satisfaz à equação  $x_i \equiv a^{2^i u}$  para  $i = 0, 1, \dots, t$ , de forma que, em particular,  $x_t \equiv a^{n-1} \pmod{n}$ . Porém, sempre que uma etapa de elevação ao quadrado é executada na linha 4, o loop pode terminar prematuramente, se as linhas 5 e 6 verificarem se acabou de ser descoberta uma raiz quadrada não trivial de 1. Nesse caso, o algoritmo pára e retorna TRUE. As linhas 7 e 8 retornam TRUE se o valor calculado para  $x_t \equiv a^{n-1} \pmod{n}$  não é igual a 1, da mesma maneira que o procedimento PSEUDOPRIME retorna COMPOSITE nesse caso. A linha 9 retorna FALSE se não tivermos retornado TRUE na linha 6 ou 8.

Agora, vamos demonstrar que, se WITNESS( $a, n$ ) retorna TRUE, então é possível construir uma prova de que  $n$  é múltiplo usando-se  $a$ .

Se WITNESS retorna TRUE a partir da linha 8, então ele descobriu que  $x_t = a^{n-1} \pmod{n} \neq 1$ . Contudo, se  $n$  é primo temos, pelo teorema de Fermat (Teorema 31.31) que  $a^{n-1} \equiv 1 \pmod{n}$  para todo  $a \in Z_n^+$ . Então,  $n$  não pode ser primo, e a equação  $a^{n-1} \pmod{n} \neq 1$  é uma prova desse fato.

Se WITNESS retorna TRUE a partir da linha 6, então ele descobriu que  $x_{i-1}$  é uma raiz quadrada não trivial de  $x_i = 1$ , módulo  $n$ , pois temos  $x_{i-1} \neq \pm 1 \pmod{n}$  ainda que  $x_i \equiv x_{i-1}^2 \equiv 1 \pmod{n}$ . O Corolário 31.35 afirma que somente se  $n$  é múltiplo pode existir uma raiz quadrada não trivial de 1 módulo  $n$ , de forma que uma demonstração de que  $x$  é uma raiz quadrada não trivial de 1 módulo  $n$  é uma prova de que  $n$  é múltiplo.

Isso completa nossa prova da correção de WITNESS. Se a invocação WITNESS( $a, n$ ) tiver TRUE como saída, então  $n$  certamente será múltiplo, e uma prova de que  $n$  é múltiplo poderá ser determinada com facilidade a partir de  $a$  e  $n$ .

Neste momento apresentamos resumidamente uma descrição alternativa do comportamento de WITNESS como uma função da seqüência  $X = \langle x_0, x_1, \dots, x_t \rangle$ , que você poderá considerar útil mais tarde, quando analisarmos a eficiência do teste do caráter primo de MILLER-RABIN. Observe que, se  $x_i = 1$  para algum  $0 \leq i < t$ , WITNESS não poderia calcular o restante da seqüência. Porém, se ele o fizesse, cada valor  $x_{i+1}, x_{i+2}, \dots, x_t$  seria 1, e consideraríamos essas posições na seqüência  $X$  como sendo todas iguais a 1. Temos quatro casos:

1.  $X = \langle \dots, d \rangle$ , onde  $d \neq 1$ : a seqüência  $X$  não termina em 1. Retorne True;  $a$  é uma testemunha do caráter múltiplo de  $n$  (pelo Teorema de Fermat).
2.  $X = \langle 1, 1, \dots, 1 \rangle$ : a seqüência  $X$  é toda formada por valores 1. Retorne False;  $a$  não é uma testemunha do caráter múltiplo de  $n$ .
3.  $X = \langle \dots, -1, 1, \dots, 1 \rangle$ : a seqüência  $X$  termina em 1, e o último valor diferente de 1 é igual a  $-1$ . Retorne FALSE;  $a$  não é uma testemunha do caráter múltiplo de  $n$ .
4.  $X = \langle \dots, d, 1, \dots, 1 \rangle$ , onde  $d \neq \pm 1$ : a seqüência  $X$  termina em 1, mas o último valor diferente de 1 não é igual a  $-1$ . Retorne True;  $a$  é uma testemunha do caráter múltiplo de  $n$ , pois  $d$  é uma raiz quadrada não trivial de 1.

Agora, examinamos o teste do caráter primo de Miller-Rabin com base no uso de WITNESS. Novamente, supomos que  $n$  é um inteiro ímpar maior que 2.

**MILLER-RABIN( $n, s$ )**

```

1 for  $j \leftarrow 1$  to  $s$ 
2   do  $\leftarrow \text{RANDOM}(1, n - 1)$ 
3     if  $\text{WITNESS}(a, n)$ 
4       then return COMPOSITE       $\triangleright$  Definitivamente.
5 return PRIME                   $\triangleright$  Quase certamente.

```

O procedimento MILLER-RABIN é uma procura probabilística de uma prova de que  $n$  é múltiplo. O loop principal (começando na linha 1) escolhe  $s$  valores aleatórios de  $a$  de  $\mathbb{Z}_n^+$  (linha 2). Se um dos valores de  $a$  escolhidos for uma testemunha do caráter múltiplo de  $n$ , então a saída de MILLER-RABIN será COMPOSITE na linha 4. Tal saída é sempre correta, pela correção de WITNESS. Se nenhuma testemunha puder ser encontrada em  $s$  tentativas, MILLER-RABIN presumirá que isso ocorreu porque não existe nenhuma testemunha a ser encontrada, e portanto  $n$  é primo. Veremos que essa saída provavelmente estará correta se  $s$  for grande o bastante, mas que existe uma pequena chance de que o procedimento talvez não tenha tido sorte em sua escolha dos valores de  $a$ , e que as testemunhas existem, ainda que não se tenha encontrado nenhuma.

Para ilustrar a operação de MILLER-RABIN, seja  $n$  o número de Carmichael 561, de forma que  $n - 1 = 560 = 2^4 \cdot 35$ . Supondo que  $a = 7$  seja escolhido como uma base, a Figura 31.4 mostra que WITNESS calcula  $x_0 \equiv a^{35} \equiv 241 \pmod{561}$  e portanto calcula a seqüência  $X = \langle 241, 298, 166, 67, 1 \rangle$ . Desse modo, WITNESS descobre uma raiz quadrada não trivial de 1 na última etapa de elevação ao quadrado, pois  $a^{280} \equiv 67 \pmod{n}$  e  $a^{560} \equiv 1 \pmod{n}$ . Então,  $a = 7$  é uma testemunha do caráter múltiplo de  $n$ , WITNESS(7, n) retorna TRUE e MILLER-RABIN retorna COMPOSITE.

Se  $n$  é um número de  $\beta$  bits, MILLER-RABIN exige  $O(s\beta)$  operações aritméticas e  $O(s\beta^3)$  operações de bits, pois ele não exige assintoticamente mais trabalho que  $s$  exponenciações modulares.

## Taxa de erros do teste do caráter primo de Miller-Rabin

Se a saída de MILLER-RABIN é PRIME, então existe uma pequena chance de que ele tenha cometido um erro. Porém, diferente do que ocorre em PSEUDOPRIME, a chance de erro não depende de  $n$ ; não existe nenhuma entrada ruim para esse procedimento. Em vez disso, ele depende do tamanho de  $s$  e da “sorte no jogo” ao escolher valores básicos  $a$ . Além disso, tendo em vista que

cada teste é mais estrito que uma simples verificação da equação (31.38), podemos esperar, em princípios gerais, que a taxa de erro seja pequena para inteiros  $n$  escolhidos aleatoriamente. O teorema a seguir apresenta um argumento mais preciso.

### Teorema 31.38

Se  $n$  é um número múltiplo ímpar, então o número de testemunhas do caráter múltiplo de  $n$  é pelo menos  $(n - 1)/2$ .

**Prova** A prova mostra que o número de não testemunhas é no máximo  $(n - 1)/2$ , o que implica o teorema.

Começamos afirmando que qualquer não testemunha deve ser um membro de  $Z_n^*$ . Por quê? Considere qualquer não testemunha  $a$ . Ela deve satisfazer  $a^{n-1} \not\equiv 1 \pmod{n}$  ou, de modo equivalente,  $a \cdot a^{n-2} \not\equiv 1 \pmod{n}$ . Portanto, existe uma solução para a equação  $ax \equiv 1 \pmod{n}$ , ou seja,  $a^{n-2}$ . Pelo Corolário 31.21,  $\text{mdc}(a, n) \mid 1$  que, por sua vez, implica que  $\text{mdc}(a, n) = 1$ . Assim  $a$  é um membro de  $Z_n^*$ ; todas as não testemunhas pertencem a  $Z_n^*$ .

Para completar a prova, mostramos que não apenas todas as não testemunhas estão contidas em  $Z_n^*$ , mas todas elas estão contidas em um subgrupo próprio  $B$  de  $Z_n^*$  (lembre-se de que dizemos que  $B$  é um subgrupo *próprio* de  $Z_n^*$  quando  $B$  é um subgrupo de  $Z_n^*$ , mas  $B$  não é igual a  $Z_n^*$ ). Pelo Corolário 31.16, temos então  $|B| \leq |Z_n^*|/2$ . Tendo em vista que  $Z_n^* \leq n - 1$ , obtemos  $|B| \leq (n - 1)/2$ . Então, o número de não testemunhas é no máximo  $(n - 1)/2$ , e assim o número de testemunhas deve ser pelo menos  $(n - 1)/2$ .

Agora, mostraremos como encontrar um subgrupo próprio  $B$  de  $Z_n^*$ , que contém todas as não testemunhas. Dividiremos a prova em dois casos.

*Caso 1:* Existe um  $x \in Z_n^*$  tal que

$$x^{n-1} \not\equiv 1 \pmod{n}.$$

Em outras palavras,  $n$  não é um número de Carmichael. Pelo fato de que, como observamos antes, os números de Carmichael são extremamente raros, o Caso 1 é o caso principal que surge “na prática” (por exemplo, quando  $n$  é escolhido aleatoriamente e seu caráter primo está sendo testado).

Seja  $B = \{n \in Z_n^* : b^{n-1} \equiv 1 \pmod{n}\}$ . É claro que  $B$  é não vazio, pois  $1 \in B$ . Tendo em vista que  $B$  é fechado sob a multiplicação módulo  $n$ , temos que  $B$  é um subgrupo de  $Z_n^*$  pelo Teorema 31.14. Observe que toda não testemunha pertence a  $B$ , pois uma não testemunha  $a$  satisfaz a  $a^{n-1} \not\equiv 1 \pmod{n}$ . Como  $x \in Z_n^*$ , temos que  $B$  é um subgrupo próprio de  $Z_n^*$ .

*Caso 2:* Para todo  $Z_n^*$ ,

$$x^{n-1} \equiv 1 \pmod{n} \tag{31.39}$$

Em outras palavras,  $n$  é um número de Carmichael. Esse caso é extremamente raro na prática. Porém, o teste de Miller-Rabin (diferente de um teste de caráter pseudoprímo) pode determinar de modo eficiente o caráter múltiplo de números de Carmichael, como mostramos agora.

Nesse caso,  $n$  não pode ser uma potência de um primo. Para ver por que, vamos supor a hipótese contrária de que  $n = p^e$ , onde  $p$  é um primo e  $e > 1$ . Derivamos uma contradição como a seguir. Tendo em vista que supomos que  $n$  é ímpar,  $p$  também deve ser ímpar. O Teorema 31.32 implica que  $Z_n^*$  é um grupo cíclico: ele contém um gerador  $g$  tal que  $\text{ord}_n(g) = |Z_n^*| = \phi(n) = p^e(1 - 1/p) = (p - 1)p^{e-1}$ . Pela equação (31.39), temos  $g^{n-1} \equiv 1 \pmod{n}$ . Então, o teorema do logaritmo discreto (Teorema 31.33, tomando-se  $y = 0$ ) implica que  $n - 1 \equiv 0 \pmod{\phi(n)}$ , ou

$$(p - 1)p^{e-1} \mid p^e - 1.$$

Isso é uma contradição para  $e > 1$ , pois  $(p - 1)p^{e-1}$  é divisível pelo primo  $p$ , mas  $p^e - 1$  não é. Desse modo,  $n$  não é uma potência de um primo.

Tendo em vista que o número múltiplo ímpar  $n$  não é uma potência de um primo, decomponemos esse número em um produto  $n_1 n_2$ , onde  $n_1$  e  $n_2$  são números ímpares maiores que 1 e primos entre si. (Podem existir várias maneiras para fazer isso, e não importa a maneira que escolhemos. Por exemplo, se  $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$ , então podemos escolher  $n_1 = p_1^{e_1}$  e  $n_2 = p_2^{e_2} p_3^{e_3} \dots p_r^{e_r}$ .)

Lembre-se de que definimos  $t$  e  $u$  de modo que  $n - 1 = 2^t u$ , onde  $t \geq 1$  e  $u$  é ímpar, e que, para uma entrada  $a$ , o procedimento WITNESS calcula a seqüência

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \dots, a^{2^t u} \rangle$$

(todos os cálculos são executados em módulo  $n$ ).

Vamos dizer que um par de inteiros  $(v, j)$  é **aceitável** se  $v \in Z_n^*$ ,  $j \in \{0, 1, \dots, t\}$  e  $v^{2^j u} \equiv -1 \pmod{n}$ .

Pares aceitáveis certamente existem, pois  $u$  é ímpar; podemos escolher  $v = n - 1$  e  $j = 0$ , de forma que  $(n - 1, 0)$  seja um par aceitável. Agora, escolha o maior  $j$  possível tal que exista um par aceitável  $(v, j)$  e fixe  $v$  de modo que  $(v, j)$  seja um par aceitável. Seja

$$B = \{x \in Z_n^* : x^{2^j u} \equiv \pm 1 \pmod{n}\}.$$

Como  $B$  é fechado sob a multiplicação módulo  $n$ , ele é um subgrupo de  $Z_n^*$ . Portanto, pelo Corolário 31.16,  $|B|$  divide  $|Z_n^*|$ . Toda não testemunha deve ser um elemento de  $B$ , pois a seqüência  $X$  produzida por uma não testemunha deve ser formada toda por valores 1, ou então conter um valor  $-1$  não além da  $j$ -ésima posição, pelo caráter máximo de  $j$ . (Se  $(a, j')$  é aceitável, onde  $a$  é uma não testemunha, devemos ter  $j' \leq j$ , pelo modo como escolhemos  $j$ .)

Agora, usaremos a existência de  $v$  para demonstrar que existe um  $w \in Z_n^* - B$ . Como  $v^{2^j u} \in 1 \pmod{n}$ , temos  $v^{2^j u} \equiv -1 \pmod{n_1}$  pelo Corolário 31.29 para o teorema chinês do resto. Pelo Corolário 31.28, existe um  $w$  que satisfaz simultaneamente às equações

$$w \equiv v \pmod{n_1},$$

$$w \equiv v \pmod{n_2}.$$

Então,

$$w^{2^j u} \equiv -1 \pmod{n_1},$$

$$w^{2^j u} \equiv 1 \pmod{n_2}.$$

Pelo Corolário 31.29,  $w^{2^j u} \not\equiv 1 \pmod{n_1}$  implica  $w^{2^j u} \not\equiv 1 \pmod{n}$ , e  $w^{2^j u} \not\equiv -1 \pmod{n_2}$  implica  $w^{2^j u} \not\equiv -1 \pmod{n}$ . Conseqüentemente,  $w^{2^j u} \not\equiv \pm 1 \pmod{n}$ , e então  $w \notin B$ .

Resta mostrar que  $w \in Z_n^*$ , o que fazemos trabalhando primeiramente separadamente em módulo  $n_1$  e em módulo  $n_2$ . Trabalhando em módulo  $n_1$ , observamos que, tendo em vista que  $v \in Z_n^*$ , temos que  $\text{mdc}(v, n) = 1$ , e assim também  $\text{mdc}(v, n_1) = 1$ ; se  $v$  não tem quaisquer divisores comuns com  $n$ , certamente ele não tem quaisquer divisores comuns com  $n_1$ . Considerando que  $w \equiv v \pmod{n_1}$ , vemos que  $\text{mdc}(w, n_1) = 1$ . Trabalhando em módulo  $n_2$ , observamos que  $w \equiv 1 \pmod{n_2}$  implica  $\text{mdc}(w, n_2) = 1$ . Para combinar esses resultados, usamos o Teorema 31.6, que implica que  $\text{mdc}(w, n_1 n_2) = \text{mdc}(w, n) = 1$ . Isto é,  $w \in Z_n^*$ .

Assim,  $w \in Z_n^* - B$ , e finalizamos o Caso 2 com a conclusão de que  $B$  é um subgrupo próprio

Em qualquer caso, vemos que o número de testemunhas para o caráter múltiplo de  $n$  é pelo menos  $(n - 1)/2$ . ■

### **Teorema 31.39**

Para qualquer inteiro ímpar  $n > 2$  e inteiro positivo  $s$ , a probabilidade de MILLER-RABIN( $n, s$ ) errar é no máximo  $2^{-s}$ .

**Prova** Usando o Teorema 31.38 vemos que, se  $n$  é múltiplo, então cada execução do loop for das linhas 1 a 4 tem uma probabilidade de pelo menos  $1/2$  de descobrir uma testemunha  $x$  para o caráter múltiplo de  $n$ . MILLER-RABIN só cometerá um erro se for tão sem sorte que deixe de descobrir uma testemunha para o caráter múltiplo de  $n$  em cada uma das  $s$  iterações do loop principal. A probabilidade de haver uma tal seqüência de perdas é no máximo  $2^{-s}$ . ■

Desse modo, a escolha de  $s = 50$  deve bastar para quase qualquer aplicação imaginável. Se estivermos tentando encontrar primos grandes aplicando MILLER-RABIN a inteiros maiores *escolhidos aleatoriamente*, então é possível demonstrar (embora não o façamos aqui) que a escolha de um valor pequeno de  $s$  (digamos 3) tem bem pouca probabilidade de conduzir a resultados errôneos. Isto é, para um inteiro múltiplo ímpar  $n$  escolhido aleatoriamente, o número esperado de não testemunhas do caráter múltiplo de  $n$  provável é muito menor que  $(n - 1)/2$ . Porém, se o inteiro  $n$  não for escolhido de forma aleatória, o melhor que se poderá provar é que o número de não testemunhas é no máximo  $(n - 1)/4$ , usando uma versão melhorada do Teorema 31.38. Além disso, existem inteiros  $n$  para os quais o número de não testemunhas é  $(n - 1)/4$ .

## **Exercícios**

### **31.8-1**

Prove que, se um inteiro  $n > 1$  não é um primo ou uma potência de um primo, então existe uma raiz quadrada não trivial de 1 módulo  $n$ .

### **31.8-2 \***

É possível aumentar ligeiramente o poder do teorema de Euler para a forma

$$a^{\lambda(n)} \equiv 1 \pmod{n} \text{ para todos } a \in \mathbb{Z}_n^*,$$

onde  $n = p_1^{e_1} \cdots p_r^{e_r}$ , e  $\lambda(n)$  é definido por

$$\lambda(n) = \text{mmc}(\phi(p_1^{e_1}), \dots, \phi(p_r^{e_r})). \quad (31.40)$$

Prove que  $\lambda(n) \mid (\phi(n))$ . Um número múltiplo  $n$  é um número de Carmichael se  $\lambda(n) \mid n - 1$ . O menor número de Carmichael é  $561 = 3 \cdot 11 \cdot 17$ ; no caso,  $\lambda(561) = \text{mmc}(2, 10, 16) = 80$ , que divide 560. Prove que os números de Carmichael devem ser ambos “livres de quadrados” (não divisíveis pelo quadrado de qualquer primo) e o produto de pelo menos três primos. Por essa razão, eles não são muito comuns.

### **31.8-3**

Prove que, se  $x$  é uma raiz quadrada não trivial de 1, módulo  $n$ , então  $\text{mdc}(x - 1, n)$  e  $\text{mdc}(x + 1, n)$  são ambos divisores não triviais de  $n$ .

## **★ 31.9 Fatoração de inteiros**

Suponha que temos um inteiro  $n$  que desejamos *fatorar*, ou seja, decompor esse inteiro em um produto de primos. O teste do caráter primo da seção anterior nos informaria que  $n$  é múltiplo, mas em geral ele não nos informa os fatores primos de  $n$ . A fatoração de um inteiro grande  $n$  pa-

rece ser muito mais difícil que simplesmente determinar se  $n$  é primo ou múltiplo. É inviável com os supercomputadores de hoje e os melhores algoritmos já criados fatorar um número arbitrário de 1024 bits.

## Heurística rho de Pollard

A divisão experimental por todos os inteiros até  $B$  oferece a garantia de fatorar completamente qualquer número até  $B^2$ . Para a mesma quantidade de trabalho, o procedimento a seguir irá fatorar qualquer número até  $B^4$  (a menos que estejamos sem sorte). Como o procedimento é apenas uma heurística, nem seu tempo de execução nem seu sucesso é garantido, embora o procedimento seja bastante eficiente na prática. Outra vantagem do procedimento POLLARD-RHO é que usa apenas um número constante de posições de memória. (Você pode implementar facilmente POLLARD-RHO em uma calculadora de bolso programável para encontrar fatores de números pequenos.)

### POLLARD-RHO( $n$ )

```

1   $i \leftarrow 1$ 
2   $x_1 \leftarrow \text{RANDOM}(0, n - 1)$ 
3   $y \leftarrow x_1$ 
4   $k \leftarrow 2$ 
5  while TRUE
6    do  $i \leftarrow i + 1$ 
7     $x_i \leftarrow (x_{i-1}^2 - 1) \bmod n$ 
8     $d \leftarrow \text{mdc}(y - x_i, n)$ 
9    if  $d \neq 1$  e  $d \neq n$ 
10      then imprimir  $d$ 
11    if  $i = k$ 
12      then  $y \leftarrow x_i$ 
13       $k \leftarrow 2k$ 
```

O procedimento funciona da maneira descrita a seguir. As linhas 1 e 2 inicializam  $i$  como 1 e  $x_1$  como um valor escolhido aleatoriamente em  $\mathbb{Z}_n$ . O loop **while** na linha 5 iteração para sempre, procurando por fatores de  $n$ . Durante cada iteração do loop **while**, a recorrência

$$x_i \leftarrow (x_{i-1}^2 - 1) \bmod n \quad (31.41)$$

é usada na linha 7 para produzir o próximo valor de  $x_i$  na seqüência infinita

$$x_1, x_2, x_3, x_4, \dots ; \quad (31.42)$$

o valor de  $i$  é incrementado de forma correspondente na linha 6. O código é escrito com o uso de variáveis subscritas  $x_i$  por clareza, mas o programa funciona do mesmo modo se todos os subscritos forem retirados, pois somente o valor mais recente de  $x_i$  precisa ser mantido. Com essa modificação, o procedimento só utiliza um número constante de posições de memória.

De vez em quando, o programa grava na variável  $y$  o valor  $x_i$  gerado mais recentemente. De maneira específica, os valores que são gravados são aqueles cujos subscritos são potências de 2:

$$x_1, x_2, x_4, x_8, x_{16}, \dots ;$$

A linha 3 grava o valor  $x_i$  e a linha 12 grava  $x_k$ , sempre que  $i$  é igual a  $k$ . A variável  $k$  é inicializada como 2 na linha 4, e  $k$  é duplicado na linha 13 sempre que  $y$  é atualizado. Então,  $k$  segue a seqüência 1, 2, 4, 8, ... e sempre fornece o subscrito do próximo valor  $x_k$  a ser gravado em  $y$ .

As linhas 8 a 10 tentam encontrar um fator de  $n$ , usando o valor gravado de  $y$  e o valor atual de  $x_i$ . Especificamente, a linha 8 calcula o máximo divisor comum  $d = \text{mdc}(y - x_i, n)$ . Se  $d$  é um divisor não trivial de  $n$  (verificado na linha 9), então a linha 10 imprime  $d$ .

Esse procedimento para encontrar um fator pode parecer um tanto misterioso a princípio. Contudo, observe que POLLARD-RHO nunca imprime uma resposta incorreta; qualquer número que ele imprime é um divisor não trivial de  $n$ . Entretanto, POLLARD-RHO pode não imprimir nada; não há nenhuma garantia de que ele produzirá algum resultado. Porém, veremos que existe uma boa razão para esperar que POLLARD-RHO imprima um fator  $p$  de  $n$  depois de  $\Theta(\sqrt{p})$  iterações do loop `while`. Desse modo, se  $n$  é múltiplo, podemos esperar que esse procedimento descubra divisores suficientes para fatorar  $n$  completamente depois de cerca de  $n^{1/4}$  atualizações, pois todo fator primo  $p$  de  $n$ , exceto possivelmente o maior deles, é menor que  $\sqrt{n}$ .

Iniciamos nossa análise do comportamento desse procedimento estudando o tempo que demora para uma seqüência aleatória de módulo  $n$  repetir um valor. Tendo em vista que  $Z_n$  é finito, e que cada valor na seqüência (31.42) depende apenas do valor anterior, a seqüência (31.42) eventualmente se repete. Uma vez que alcançamos um  $x_i$  tal que  $x_i = x_j$  para algum  $j < i$ , estamos em um ciclo, pois  $x + 1 = x_{j+1}, x_{j+2} = x_{j+3}$  e assim por diante. A razão para o nome “heurística rho” é que, como mostra a Figura 31.7, a seqüência  $x_1, x_2 = x_{j-1}$  pode ser desenhada como a “cauda” da letra grega rô, e o ciclo  $x_j, x_{j+1}, \dots, x_i$  como o “corpo” da letra rô.

Vamos considerar a questão do tempo necessário para que a seqüência de  $x_i$  se repita. Isso não é exatamente o que precisamos, mas veremos depois como modificar o argumento.

Para a finalidade dessa estimativa, vamos supor que a função

$$f_n(x) = (x^2 - 1) \bmod n$$

se comporte como uma função “aleatória”. É claro que ela não é realmente aleatória, mas essa hipótese produz resultados coerentes com o comportamento observado de POLLARD-RHO. Podemos então considerar cada  $x_i$  como sendo traçado independentemente a partir de  $Z_n$ , de acordo com uma distribuição uniforme sobre  $Z_n$ . Pela análise do paradoxo do aniversário da Seção 5.4.1, o número esperado de passos executados antes da entrada da seqüência em um ciclo é  $\Theta(\sqrt{n})$ .

Agora, vamos à modificação necessária. Seja  $p$  um fator não trivial de  $n$  tal que  $\text{mdc}(p, n/p) = 1$ . Por exemplo, se  $n$  tem a fatoração  $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$ , então podemos considerar  $p = p_1^{e_1}$ . (Se  $e_1 = 1$ , então  $p$  é apenas o menor fator primo de  $n$ , um bom exemplo para ter em mente.)

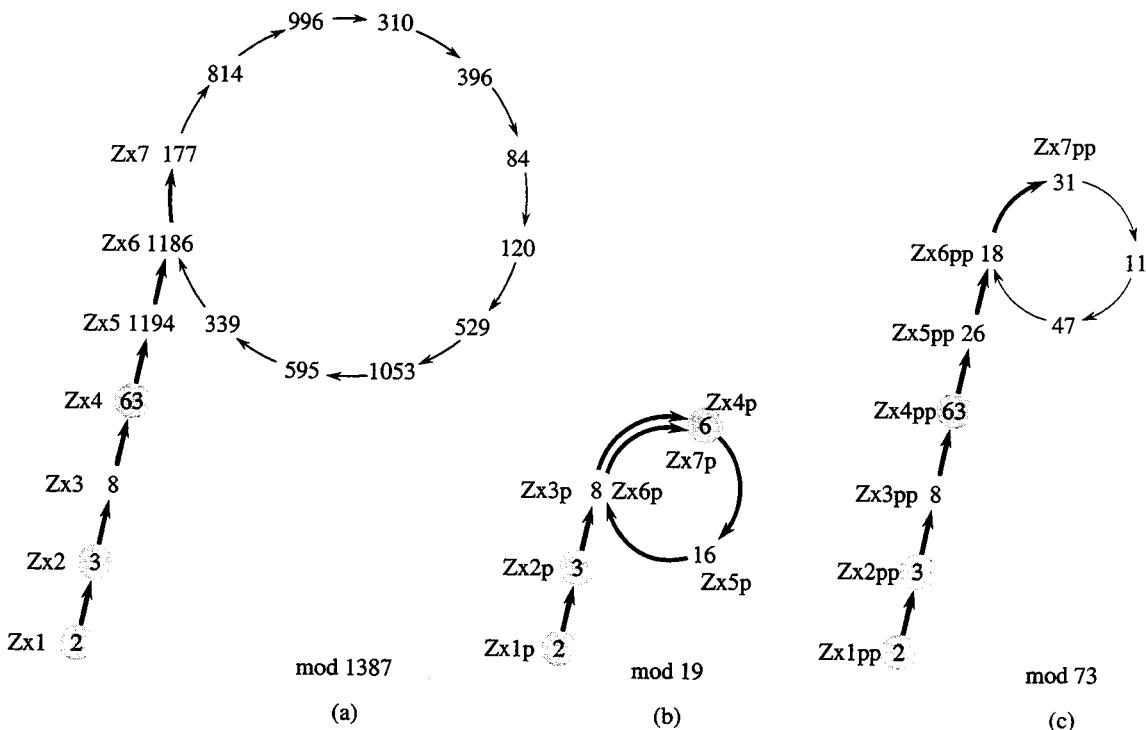
A seqüência  $\langle x_i \rangle$  induz uma seqüência correspondente  $\langle x'_i \rangle$  módulo  $p$ , onde

$$x'_i = x_i \bmod p$$

para todo  $i$ .

Além disso, como  $f_n$  é definido usando-se apenas operações aritméticas (elevação ao quadrado e subtração) em módulo  $n$ , veremos que é possível calcular  $x'_{i+1}$  a partir de  $x'_i$ ; a visão de “módulo  $p$ ” da seqüência é uma versão menor do que está acontecendo em módulo  $n$ :

$$\begin{aligned} x'_{i+1} &= x_{i+1} \bmod p \\ &= f_n(x_i) \bmod p \\ &= ((x_i^2) \bmod p) \bmod p \\ &= (x_i^2) \bmod p \quad (\text{pelo Exercício 31.1-6}) \\ &= ((x_i \bmod p)^2 - 1) \bmod p \\ &= ((x'_i)^2 - 1) \bmod p \\ &= f_p(x'_i). \end{aligned}$$



**FIGURA 31.7** A heurística rho de Pollard. (a) Os valores produzidos pela recorrência  $x_{i+1} \leftarrow (x_i^2 - 1)$ , começando com  $x_1 = 2$ . A decomposição em fatores primos de 1387 é  $19 \cdot 73$ . As setas grossas indicam os passos da iteração que são executados antes de ser descoberto o fator 19. As setas finas apontam para valores não alcançados na iteração, a fim de ilustrar a forma “rho” (rô). Os valores sombreados são os y valores armazenados por POLLARD-RHO. O fator 19 é descoberto depois de  $x_7 = 177$  ser alcançado, quando  $\text{mdc}(63 - 177, 1387) = 19$  é calculado. O primeiro valor de  $x$  que se repetiria é 1186, mas o fator 19 é descoberto antes de ser alcançado esse valor. (b) Os valores produzidos pela mesma recorrência, módulo 19. Todo valor  $x_i$  dado na parte (a) é equivalente, em módulo 19, ao valor  $x'_i$  mostrado aqui. Por exemplo, tanto  $x_4 = 63$  quanto  $x_7 = 177$  são equivalentes a 6, módulo 19. (c) Os valores produzidos pela mesma recorrência, módulo 73. Todo valor  $x_i$  dado na parte (a) é equivalente, em módulo 73, ao valor  $x''_i$  mostrado aqui. Pelo teorema chinês do resto, cada nó na parte (a) corresponde a um par de nós, um deles proveniente da parte (b) e outro da parte (c).

Desse modo, embora não estejamos calculando explicitamente a seqüência  $\langle x'_i \rangle$ , essa seqüência é bem definida e obedece à mesma recorrência que a seqüência  $\langle x_i \rangle$ .

Pelo mesmo raciocínio de antes, descobrimos que o número esperado de passos antes de se repetir a seqüência  $\langle x'_i \rangle$  é  $\Theta(\sqrt{p})$ . Se  $p$  é pequeno em comparação com  $n$ , a seqüência  $\langle x'_i \rangle$  pode se repetir muito mais depressa que a seqüência  $\langle x_i \rangle$ . Na realidade, a seqüência  $\langle x'_i \rangle$  se repete assim que dois elementos da seqüência  $\langle x_i \rangle$  são equivalentes apenas em módulo  $p$ , em lugar de serem equivalentes em módulo  $n$ . A Figura 31.7, partes (b) e (c), apresenta uma ilustração desse fato.

Seja  $t$  o índice do primeiro valor repetido na seqüência  $\langle x'_i \rangle$  e seja  $u > 0$  o comprimento do ciclo assim produzido. Isto é,  $t & u > 0$  são os menores valores tais que  $x'_{t+i} = x'_{t+u+i}$  para todo  $i \geq 0$ . Pelos argumentos anteriores, os valores esperados de  $t$  e  $u$  são ambos  $\Theta(\sqrt{p})$ . Observe que, se  $x'_{t+i} = x'_{t+u+i}$ , então  $p \mid (x_{t+u+i} - x_{t+i})$ . Desse modo,  $(x_{t+u+i} - x_{t+i}, n) > 1$ .

Portanto, uma vez que POLLARD-RHO tenha gravado como  $y$  qualquer valor  $x_k$  tal que  $k > t$ , então  $y \bmod p$  estará sempre no ciclo módulo  $p$ . (Se um novo valor for gravado como  $y$ , esse valor também estará no ciclo módulo  $p$ .) Eventualmente,  $k$  é definido como um valor maior que  $u$ , e então o procedimento executará um loop inteiro em torno do ciclo módulo  $p$ , sem mudar o valor de  $y$ . Um fator  $n$  será então descoberto quando  $x_i$  “colidir com” o valor previamente armazenado de  $y$ , módulo  $p$ , isto é, quando  $x_i \equiv y \pmod{p}$ .

Pode-se presumir que o fator encontrado é o fator  $p$ , embora ocasionalmente possa ocorrer de um múltiplo de  $p$  ser descoberto. Como os valores esperados de  $t$  e  $u$  são  $\Theta(\sqrt{p})$ , o número esperado de passos necessários para produzir o fator  $p$  é  $\Theta(\sqrt{p})$ .

Há duas razões pelas quais esse algoritmo pode não funcionar como esperado. Primeiro, a análise heurística do tempo de execução não é rigorosa, e é possível que o ciclo de valores, módulo  $p$ , seja muito maior que  $\sqrt{p}$ . Nesse caso, o algoritmo será executado corretamente, embora com lentidão muito maior que a desejada. Na prática, isso parece não ser uma questão importante. Em segundo lugar, os divisores de  $n$  produzidos por esse algoritmo sempre poderiam ser um dos fatores triviais 1 ou  $n$ . Por exemplo, suponha que  $n = pq$ , onde  $p$  e  $q$  são primos. Pode acontecer de os valores de  $t$  e  $u$  para  $p$  serem idênticos aos valores de  $t$  e  $u$  para  $q$ , e portanto o fator  $p$  ser sempre revelado na mesma operação mdc que revela o fator  $q$ . Tendo em vista que ambos os fatores são revelados ao mesmo tempo, o fator trivial  $pq = n$  é revelado, o que é inútil. Mais uma vez, isso não parece ser um problema real na prática. Se necessário, a heurística pode ser reiniciada com uma recorrência diferente da forma  $x_{i+1} \leftarrow (x_i^2 - c)$ . (Os valores  $c = 0$  e  $c = 2$  devem ser evitados por razões que não nos interessam aqui, mas outros valores servem.)

É claro que essa análise é heurística e não rigorosa, pois a recorrência não é realmente “aleatória”. Todavia, o procedimento funciona bem na prática, e parece ser tão eficiente como sua análise heurística indica. Ele é o método preferido para encontrar pequenos fatores primos de um número grande. Para fatorar completamente um número múltiplo  $n$  de  $\beta$  bits, só precisamos encontrar todos os fatores primos menores que  $\lfloor n^{1/2} \rfloor$ , e assim esperamos que POLLARD-RHO exija no máximo  $n^{1/2} = 2^{\beta/4}$  operações aritméticas, e no máximo  $n^{1/2} \beta^2 = 2^{\beta/4} \beta^2$  operações de bits. A capacidade de POLLARD-RHO para encontrar um fator pequeno  $p$  de  $n$  com um número esperado  $\Theta(\sqrt{p})$  de operações aritméticas freqüentemente é sua característica mais atraente.

## Exercícios

### 31.9-1

Consultando o histórico de execução mostrado na Figura 31.7(a), quando POLLARD-RHO imprime o fator 73 de 1387?

### 31.9-2

Suponha que temos uma função  $f: \mathbb{Z}_n \rightarrow \mathbb{Z}_n$  e um valor inicial  $x_0 \in \mathbb{Z}$ . Defina  $x_i = f(x_{i-1})$  para  $i = 1, 2, \dots$ . Sejam  $t$  e  $u > 0$  os menores valores tais que  $x_{t+u} = x_t$  para  $i = 0, 1, \dots$ . Na terminologia do algoritmo rho de Pollard,  $t$  é o comprimento da cauda e  $u$  é o comprimento do ciclo do rho. Forneça um algoritmo eficiente para determinar com exatidão  $t$  e  $u$ , e analise seu tempo de execução.

### 31.9-3

Quantos passos você esperaria que POLLARD-RHO exigisse para descobrir um fator da forma  $p^e$ , onde  $p$  é primo e  $e > 1$ ?

### 31.9-4 \*

Uma desvantagem de POLLARD-RHO na forma como foi escrito é que ele exige um cálculo de mdc para cada passo da recorrência. Foi sugerido que poderíamos criar um lote de cálculos do mdc, acumulando o produto de vários  $x_i$  em uma linha, e depois usando esse produto em lugar de  $x_i$  no cálculo do mdc. Descreva cuidadosamente como você implementaria essa idéia, por que ela funciona e que tamanho de lote você escolheria como o mais eficiente ao trabalhar com um número  $n$  de  $\beta$  bits.

## Problemas

### 31-1 Algoritmo de mdc binário

Na maioria dos computadores, as operações de subtração, teste de paridade (ímpar ou par) de um inteiro binário e divisão em metades podem ser executadas com maior rapidez que o cálculo | 713

de restos. Este problema investiga o **algoritmo de mdc binário**, que evita os cálculos de resto usados no algoritmo de Euclides.

- Prove que, se  $a$  e  $b$  são ambos pares, então  $\text{mdc}(a, b) = 2 \text{mdc}(a/2, b/2)$ .
- Prove que, se  $a$  é ímpar e  $b$  é par, então  $\text{mdc}(a, b) = \text{mdc}(a, b/2)$ .
- Prove que, se  $a$  e  $b$  são ambos ímpares, então  $\text{mdc}(a, b) = \text{mdc}((a - b)/2, b)$ .
- Projete um algoritmo de mdc binário eficiente para inteiros de entrada  $a$  e  $b$ , onde  $a \geq b$ , que funcione no tempo  $O(\lg a)$ . Suponha que cada subtração, teste de paridade e divisão em metades possa ser executada em tempo unitário.

### 31-2 Análise de operações de bits no algoritmo de Euclides

- Considere o algoritmo comum “de lápis e papel” para a divisão longa; dividir  $a$  por  $b$ , gerando um quociente  $q$  e um resto  $r$ . Mostre que esse método exige  $O((1 + \lg q) \lg b)$  operações de bits.
- Defina  $\mu(a, b) = (1 + \lg a)(1 + \lg b)$ . Mostre que o número de operações de bits executadas por EUCLID na redução do problema de calcular  $\text{mdc}(a, b)$  ao problema de calcular  $\text{mdc}(b, a \bmod b)$  é no máximo  $c(\mu(a, b) - \mu(b, a \bmod b))$  para alguma constante suficientemente grande  $c > 0$ .
- Mostre que  $\text{EUCLID}(a, b)$  exige  $O(\mu(a, b))$  operações de bits em geral e  $O(\beta^2)$  operações de bits quando aplicado a duas entradas de  $\beta$  bits.

### 31-3 Três algoritmos para números de Fibonacci

Este problema compara a eficiência de três métodos para calcular o  $n$ -ésimo número de Fibonacci  $F_n$ , dado  $n$ . Suponha que o custo de adicionar, subtrair ou multiplicar dois números seja  $O(1)$ , independente do tamanho dos números.

- Mostre que o tempo de execução do método recursivo direto para calcular  $F_n$  baseado na recorrência (3.21) é exponencial em  $n$ .
- Mostre como calcular  $F_n$  no tempo  $O(n)$  usando memoização.
- Mostre como calcular  $F_n$  no tempo  $O(\lg n)$  usando apenas adição e multiplicação de inteiros. (Sugestão: Considere a matriz

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

e suas potências.)

- Suponha agora que a soma de dois números de  $\beta$  bits demore o tempo  $\Theta(\beta)$  e que a multiplicação de dois números de  $\beta$  bits demore o tempo  $\Theta(\beta^2)$ . Qual é o tempo de execução desses três métodos sob essa medida de custo mais razoável para as operações aritméticas elementares?

### 31-4 Resíduos quadráticos

Seja  $p$  um primo ímpar. Um número  $a \in \mathbb{Z}_p^*$  é um **resíduo quadrático** se a equação  $x^2 = a \pmod{p}$  tem uma solução para a incógnita  $x$ .

- Mostre que existem exatamente  $(p - 1)/2$  resíduos quadráticos, módulo  $p$ .
- Se  $p$  é primo, definimos o **símbolo de Legendre**  $(\frac{a}{p})$ , para  $a \in \mathbb{Z}_p^*$ , como 1 se  $a$  é um resíduo quadrático módulo  $p$  e -1 em caso contrário. Prove que, se  $a \in \mathbb{Z}_p^*$ , então

$$\left( \frac{1}{p} \right) \equiv a^{(p-1)/2} \pmod{p} .$$

Forneça um algoritmo eficiente para descobrir se um dado número  $a$  é ou não um resíduo quadrático módulo  $p$ . Analise a eficiência de seu algoritmo.

- c. Prove que, se  $p$  é um primo da forma  $4k + 3$  e  $a$  é um resíduo quadrático em  $\mathbb{Z}_p^*$ , então  $a^{k+1} \pmod p$  é uma raiz quadrada de  $a$ , módulo  $p$ . Quanto tempo é necessário para se encontrar a raiz quadrada de um resíduo quadrático  $a$  módulo  $p$ ?
- d. Descreva um algoritmo aleatório eficiente para encontrar um resíduo não quadrático, com módulo igual a um primo  $p$  arbitrário, isto é, um elemento de  $\mathbb{Z}_p^*$  que não é um resíduo quadrático. Quantas operações aritméticas seu algoritmo exige em média?

## Notas do capítulo

Niven e Zuckerman [231] fornecem uma excelente introdução à teoria elementar dos números. Knuth [183] contém uma boa descrição de algoritmos para encontrar o máximo divisor comum, bem como outros algoritmos básicos da teoria dos números. Bach [28] e Riesel [258] fornecem pesquisas mais recentes sobre a teoria dos números computacionais. Dixon [78] apresenta uma visão geral da fatoração e do teste do caráter primo. Os anais de conferências editados por Pomerance [245] contêm vários artigos de pesquisa interessantes. Mais recentemente, Bach e Shallit [29] forneceram uma avaliação excepcional dos conceitos básicos da teoria computacional dos números.

Knuth [183] discute a origem do algoritmo de Euclides. Ele aparece no Livro 7, Proposições 1 e 2, do livro *Elementos* do matemático grego Euclides, que foi escrito em torno de 300 a.C. A descrição de Euclides pode ter sido derivada de um algoritmo criado por Eudoxus, por volta de 375 a.C. O algoritmo de Euclides pode ostentar a honra de ser o mais antigo algoritmo não trivial; ele só encontra rival em um algoritmo para multiplicação conhecido pelos antigos egípcios. Shallit [274] escreve a história da análise do algoritmo de Euclides.

Knuth atribui um caso especial do teorema chinês do resto (Teorema 31.27) ao matemático chinês Sun-Tsū, que viveu em alguma época entre 200 a.C. e 200 d.C. – a data é bastante incerta. O mesmo caso especial foi apresentado pelo matemático grego Nichomachus por volta de 100 d.C. Ele foi generalizado por Chin Chiu-Shao em 1247. O teorema chinês do resto foi finalmente enunciado e provado em sua forma mais geral por L. Euler, em 1734.

O algoritmo aleatório do teste do caráter primo apresentado aqui se deve a Miller [221] e Rabin [254]; ele é o algoritmo aleatório mais rápido que se conhece para teste do caráter primo, até em fatores constantes. A prova do Teorema 31.39 é uma ligeira adaptação de uma demonstração sugerida por Bach [27]. Uma prova de um resultado mais forte para MILLER-RABIN foi dada por Monier [224, 225]. A aleatoriedade parece ser necessária para se obter um algoritmo de teste do caráter primo de tempo polinomial. O algoritmo determinístico mais rápido conhecido para teste do caráter primo é a versão de Cohen-Lenstra [65] do teste do caráter primo de Adleman, Pomerance e Rumely [3]. Ao testar o caráter primo de um número  $n$  de comprimento  $\lceil \lg(n+1) \rceil$ , ele é executado no tempo  $(\lg n)^{O(\lg \lg \lg n)}$ , que é apenas ligeiramente superpolinomial.

O problema de encontrar primos grandes “aleatórios” é discutido de forma interessante em um artigo de Beauchemin, Brassard, Crépeau, Gutier e Pomerance [33].

O conceito de um sistema de criptografia de chave pública se deve a Diffie e Hellman [74]. O sistema de criptografia RSA foi proposto em 1977 por Rivest, Shamir e Adleman [259]. Desde então, o campo da criptografia floresceu. Nossa compreensão do sistema de criptografia RSA se aprofundou, e implementações modernas utilizam aprimoramentos significativos das técnicas básicas apresentadas aqui. Além disso, muitas novas técnicas foram desenvolvidas para demonstrar que os sistemas de criptografia são seguros. Por exemplo, Goldwasser e Micali [123] mostram que a aleatoriedade poder ser uma ferramenta eficaz no projeto de esquemas confiáveis de criptografia de chave pública. Para esquemas de assinaturas, Goldwasser, Micali e Rivest [124] apresentam um esquema de assinatura digital para o qual todo tipo concebível de falsificação é

comprovadamente tão difícil quanto a decomposição em fatores primos. Menezes *et al.* [220] fornecem uma avaliação da criptografia aplicada.

A heurística rho para fatoração de inteiros foi criada por Pollard [242]. A versão apresentada aqui é uma variação proposta por Brent [48].

Os melhores algoritmos para fatoração de números grandes têm um tempo de execução que cresce de forma aproximadamente exponencial com a raiz cúbica do comprimento do número  $n$  a ser fatorado. O algoritmo geral de fatoração de peneira de campo numérico foi desenvolvido por Buhler *et al.* [51] como uma extensão das idéias sobre o algoritmo de fatoração de peneira de campo numérico criado por Pollard [243] e Lenstra *et al.* [201] e aprimorado por Coppersmith [69] e outros; esse talvez seja o algoritmo mais eficiente de modo geral para grandes entradas. Embora seja difícil apresentar uma análise rigorosa desse algoritmo, sob hipóteses razoáveis podemos derivar uma estimativa de tempo de execução igual a  $L(1/3, n)^{1,902 + o(1)}$ , onde  $L(\alpha, n) = e^{(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$ .

O método de curva elíptica criado por Lenstra [202] pode ser mais eficaz para algumas entradas que o método de peneira de campo numérico pois, como o método rho de Pollard, ele pode encontrar um pequeno fator primo  $p$  com bastante rapidez. Usando-se esse método, o tempo para encontrar  $p$  é estimado em  $L(1/2, p)^{\sqrt{2}+o(1)}$ .

## Capítulo 32

# Emparelhamento de cadeias

Encontrar todas as ocorrências de um padrão em um texto é um problema que surge com freqüência nos programas de edição de textos. Em geral, o texto é um documento que está sendo editado, e o padrão procurado é uma palavra específica fornecida pelo usuário. Os algoritmos eficientes para esse problema podem ajudar muito o nível de resposta do programa de edição de textos. Os algoritmos de emparelhamento de cadeias também são usados, por exemplo, para procurar por padrões específicos em seqüências de DNA.

Formalizamos o **problema de emparelhamento de cadeias** da maneira mostrada a seguir. Supomos que o texto é um arranjo  $T[1..n]$  de comprimento  $n$  e que o padrão é um arranjo  $P[1..m]$  de comprimento  $m \leq n$ . Supomos ainda que os elementos de  $P$  e  $T$  são caracteres tirados de um alfabeto finito  $\Sigma$ . Por exemplo, podemos ter  $\Sigma = \{0, 1\}$  ou  $\Sigma = \{a, b, \dots, z\}$ . Os arranjos de caracteres  $P$  e  $T$  são chamados com freqüência **cadeias** de caracteres.

Dizemos que o padrão  $P$  **ocorre com deslocamento  $s$**  no texto  $T$  (ou, de forma equivalente, que o padrão  $P$  **ocorre a partir da posição  $s + 1$**  no texto  $T$ ) se  $0 \leq s \leq n - m$  e  $T[s + 1..s + m] = P[1..m]$  (isto é, se  $T[s + j] = P[j]$ , para  $1 \leq j \leq m$ ). Se  $P$  ocorre com deslocamento  $s$  em  $T$ , então chamamos  $s$  um **deslocamento válido**; caso contrário, chamamos  $s$  um **deslocamento não válido**. O problema de emparelhamento de cadeias é o problema de encontrar todos os deslocamentos válidos com os quais um determinado padrão  $P$  ocorre em um dado texto  $T$ . A Figura 32.1 ilustra essas definições.

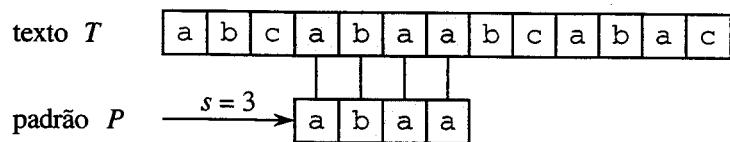


FIGURA 32.1 O problema de emparelhamento de cadeias. A meta é encontrar todas as ocorrências do padrão  $P = abaa$  no texto  $T$  abcabaabcabac. O padrão ocorre apenas uma vez no texto, no deslocamento  $s = 3$ . O deslocamento  $s = 3$  é dito um deslocamento válido. Cada caractere do padrão é conectado por uma linha vertical ao caractere correspondente no texto, e todos os caracteres que correspondem aparecem sombreados

Exceto para o algoritmo de força bruta simples, que examinamos na Seção 32.1, cada algoritmo de emparelhamento de cadeias deste capítulo executa algum pré-processamento baseado no padrão, e depois encontra todos os deslocamentos válidos; chamaremos essa fase posterior

de “emparelhamento”. A Figura 32.2 mostra os tempos de pré-processamento e correspondência para cada um dos algoritmos deste capítulo. O tempo de execução total de cada algoritmo é a soma dos tempos de pré-processamento e emparelhamento. A Seção 32.2 apresenta um interessante algoritmo de emparelhamentos de cadeias, devido a Rabin e Karp. Embora o tempo de execução no pior caso  $\Theta((n - m + 1)m)$  desse algoritmo não seja melhor que o do método simples, ele funciona muito melhor em média e na prática. Ele também é generalizado de forma interessante para outros problemas de emparelhamento de padrões. Depois disso, a Seção 32.3 descreve um algoritmo de emparelhamento de cadeias que começa pela construção de um autômato finito projetado especificamente para procurar por ocorrências do padrão  $P$  específico em um texto. Esse algoritmo é executado no tempo de pré-processamento  $O(m|\Sigma|)$ , mas apenas no tempo de emparelhamento  $\Theta(n)$ . O algoritmo similar – mas muito mais inteligente – de Knuth-Morris-Pratt (ou KMP) é apresentado na Seção 32.4; o algoritmo KMP tem o mesmo tempo de emparelhamento  $\Theta(n)$  e reduz o tempo de pré-processamento a somente  $\Theta(m)$ .

| Algoritmo          | Tempo de pré-processamento | Tempo de emparelhamento |
|--------------------|----------------------------|-------------------------|
| Simples            | 0                          | $\Theta((n - m + 1)m)$  |
| Rabin-Karp         | $\Theta(m)$                | $\Theta((n - m + 1)m)$  |
| Autômato finito    | $O(m \Sigma )$             | $\Theta(n)$             |
| Knuth-Morris-Pratt | $\Theta(m)$                | $\Theta(n)$             |

FIGURA 32.2 Algoritmos de emparelhamento de cadeias neste capítulo e seus tempos de pré-processamento e emparelhamento

## Notação e terminologia

Faremos  $\Sigma^*$  (que é lido como “sigma-asterisco”) denotar o conjunto de todos as cadeias de comprimento finito formadas usando-se caracteres do alfabeto  $\Sigma$ . Neste capítulo, consideraremos apenas cadeias de comprimento finito. A **cadeia vazia** de comprimento zero, denotada por  $\varepsilon$ , também pertence a  $\Sigma^*$ . O comprimento de uma cadeia  $x$  é denotado por  $|x|$ . A **concatenação** de duas cadeias  $x$  e  $y$ , representada por  $xy$ , tem comprimento  $|x| + |y|$  e consiste nos caracteres de  $x$  seguidos pelos caracteres de  $y$ .

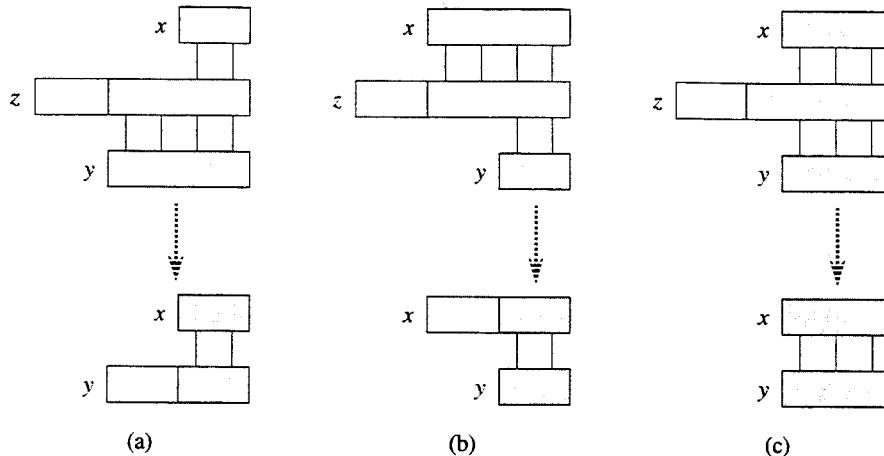
Dizemos que uma cadeia  $w$  é um **prefixo** de uma cadeia  $x$ , denotada por  $w \sqsubset x$ , se  $x = wy$  para alguma cadeia  $y \in \Sigma^*$ . Observe que, se  $w \sqsubset x$  então  $|w| \leq |x|$ . De modo semelhante, dizemos que uma cadeia  $w$  é um **sufixo** de uma cadeia  $x$ , representada por  $\sqsupset w$ , se  $x = yw$  para algum  $y \in \Sigma^*$ . Decorre de  $\sqsupset$  que  $|w| \leq |x|$ . A cadeia vazia  $\varepsilon$  é ao mesmo tempo um sufixo e um prefixo de toda cadeia. Por exemplo, temos  $ab \sqsubset abcca$  e  $c当地 cca \sqsupset abcca$ . É útil observar que, para quaisquer cadeias  $x$  e  $y$  e qualquer caractere  $a$ , temos  $x \sqsubset y$  se e somente se  $xa \sqsubset ya$ . Observe também que  $\sqsubset$  e  $\sqsupset$  são relações transitivas. O lema a seguir será útil mais adiante.

### Lema 32.1 (Lema de sufixos superpostos)

Suponha que  $x$ ,  $y$  e  $z$  sejam cadeias tais que  $x \sqsubset z$  e  $y \sqsupset z$ . Se  $|x| \leq |y|$ , então  $x \sqsubset y$ . Se  $|x| \geq |y|$ , então  $y \sqsubset x$ . Se  $|x| = |y|$ , então  $x = y$ .

**Prova** Consulte a Figura 32.2 para ver uma prova gráfica. ■

Para abreviar a notação, denotaremos o prefixo de  $k$  caracteres  $P[1..k]$  do padrão  $P[1..m]$  por  $P_k$ . Desse modo,  $P_0 = \varepsilon$  e  $P_m = P = P[1..m]$ . De modo semelhante, denotamos o prefixo de  $k$  caracteres do texto  $T$  como  $T_k$ . Usando essa notação, podemos enunciar o problema de emparelhamento de cadeias como o de encontrar todos os deslocamentos  $s$  no intervalo  $0 \leq s \leq n - m$  tais que  $P \sqsubset T_{s+m}$ .



**FIGURA 32.3** Uma prova gráfica do Lema 32.1. Supomos que  $x \sqsupset z$  e  $y \sqsupset z$ . As três partes da figura ilustram os três casos do lema. Linhas verticais conectam regiões correspondentes (que aparecem sombreadas) das cadeias. (a) Se  $|x| \leq |y|$ , então  $x \sqsupset y$ . (b) Se  $|x| \geq |y|$ , então  $y \sqsupset x$ . (c) Se  $|x| = |y|$ , então  $x = y$ .

Em nosso pseudocódigo, permitimos que duas cadeias de igual comprimento sejam comparadas para igualdade como uma operação primitiva. Se as cadeias são comparadas da esquerda para a direita e a comparação parar quando for descoberta uma incompatibilidade, faremos a suposição de que o tempo despendido por tal teste é uma função linear do número de caracteres correspondentes descobertos. Para sermos precisos, consideramos que o teste " $x = y$ " demora o tempo  $\Theta(t + 1)$ , onde  $t$  é o comprimento da mais longa cadeia  $z$  tal que  $z \sqsubset x$  e  $z \sqsubset y$ . (Escrevemos  $\Theta(t + 1)$  em vez de  $\Theta(t)$  para tratar o caso em que  $t = 0$ ; os primeiros caracteres comparados não correspondem, mas demora um período de tempo positivo executar essa comparação.)

### 32.1 O algoritmo simples de emparelhamento de cadeias

O algoritmo simples encontra todos os deslocamentos válidos usando um loop que verifica a condição  $P[1 .. m] = T[s + 1 .. s + m]$  para cada um dos  $n - m + 1$  valores possíveis de  $s$ .

## NAIVE-STRING-MATCHER( $T, P$ )

```

1  $n \leftarrow \text{comprimento}[T]$ 
2  $m \leftarrow \text{comprimento}[P]$ 
3 for  $s \leftarrow 0$  to  $n - m$ 
4   do if  $P[1..m] = T[s + 1..s + m]$ 
5     then imprimir "Padrão ocorre com deslocamento"  $s$ 

```

O procedimento simples de emparelhamento de cadeias pode ser interpretado graficamente como o deslizamento de um “gabarito” contendo o padrão sobre o texto, observando para quais deslocamentos todos os caracteres no gabarito igualam os caracteres correspondentes no texto, como ilustra a Figura 32.4. O loop **for** que começa na linha 3 considera cada deslocamento possível explicitamente. O teste na linha 4 determina se o deslocamento atual é válido ou não; esse teste envolve um loop implícito para verificar posições de caracteres correspondentes, até todas as posições corresponderem com sucesso, ou até ser encontrada uma incompatibilidade. A linha 5 imprime cada deslocamento *s* válido.

O procedimento NAIVE-STRING-MATCHER demora o tempo  $O((n - m + 1)m)$ , e esse limite é restrito no pior caso. Por exemplo, considere a cadeia de texto  $a^n$  (uma cadeia de  $n$  caracteres  $a$ ) e o padrão  $a^m$ . Para cada um dos  $n - m + 1$  valores possíveis do deslocamento  $s$ , o loop implícito na linha 4 para comparar caracteres correspondentes deve ser executado  $m$  vezes para validar o deslocamento. Desse modo, o tempo de execução do pior caso é  $\Theta((n - m + 1)m)$ , que é  $\Theta(n^2)$ .

se  $m = \lfloor n/2 \rfloor$ . O tempo de execução de NAIVE-STRING-MATCHER é igual a seu tempo de emparelhamento, pois não há nenhum pré-processamento.

Como veremos, NAIVE-STRING-MATCHER não é um procedimento ótimo para esse problema. Na realidade, neste capítulo mostraremos um algoritmo com um tempo de pré-processamento no pior caso  $\Theta(m)$  e um tempo de emparelhamento no pior caso  $\Theta(n)$ . O combinador de cadeias simples é ineficiente, porque as informações obtidas sobre o texto para um valor de  $s$  são totalmente ignoradas na consideração de outros valores de  $s$ . Porém, tais informações podem ser muito valiosas. Por exemplo, se  $P = aaab$  e descobrirmos que  $s = 0$  é válido, então nenhum dos deslocamentos 1, 2 ou 3 será válido, pois  $T[4] = b$ . Nas próximas seções, examinaremos diversas maneiras de fazer uso efetivo dessa espécie de informações.

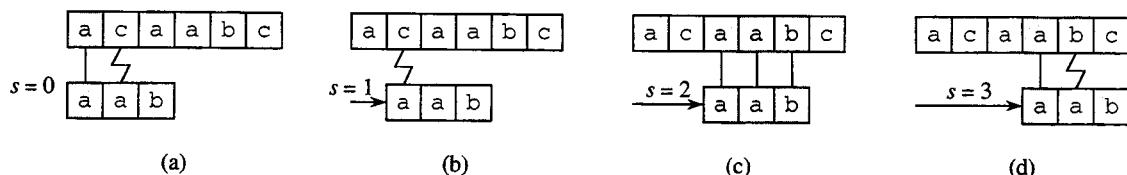


FIGURA 32.4 A operação do combinador de cadeias simples para o padrão  $P = aab$  e o texto  $T = acaabc$ . Podemos imaginar o padrão  $P$  como um “gabarito” que deslizamos próximo ao texto. (a)–(d) Os quatro alinhamentos sucessivos experimentados pelo combinador de cadeias simples. Em cada parte, linhas verticais conectam regiões correspondentes nas quais se encontrou uma coincidência (que aparece sombreada), e uma linha irregular conecta o primeiro caractere incompatível, se houver. Uma ocorrência do padrão é encontrada no deslocamento  $s = 2$ , como mostra a parte (c)

## Exercícios

### 32.1-1

Mostre as comparações que o combinador de cadeias simples realiza para o padrão  $P = 0001$  no texto  $T = 000010001010001$ .

### 32.1-2

Suponha que todos os caracteres no padrão  $P$  sejam diferentes. Mostre como acelerar NAIVE-STRING-MATCHER para ser executado no tempo  $O(n)$  sobre um texto  $T$  de  $n$  caracteres.

### 32.1-3

Suponha que o padrão  $P$  e o texto  $T$  sejam cadeias de comprimento  $m$  e  $n$ , respectivamente, escolhidas aleatoriamente a partir do alfabeto  $d$ -ário  $\Sigma_d = \{0, 1, \dots, d-1\}$ , onde  $d \geq 2$ . Mostre que o número esperado de comparações de caractere para caractere feitas pelo loop implícito na linha 4 do algoritmo simples é

$$(n - m + 1) \frac{1 - d^{-m}}{1 - d^{-1}} \leq 2((n - m + 1)$$

sobre todas as execuções desse loop. (Suponha que o algoritmo simples interrompa a comparação de caracteres para um dado deslocamento depois de ser encontrada uma incompatibilidade, ou se o padrão inteiro coincidir.) Desse modo, para cadeias escolhidos de forma aleatória, o algoritmo simples é bastante eficiente.

### 32.1-4

Vamos supor que permitimos que o padrão  $P$  contenha ocorrências de um **caractere lacuna**  $\diamond$  que pode corresponder a uma cadeia de caracteres *arbitrária* (mesmo uma cadeia de comprimento zero). Por exemplo, o padrão  $ab\diamond ba\diamond c$  ocorre no texto  $cabccbacbacab$  como

$\begin{array}{c} \text{cabccbacba} \\ \text{ab} \diamond \text{ba} \diamond \text{c} \end{array}$

e como

$\begin{array}{c} \text{cabccbacba} \\ \text{ab} \diamond \text{ba} \diamond \text{c} \end{array} \text{ ab .}$

Observe que o caractere lacuna pode ocorrer um número arbitrário de vezes no padrão, mas partimos do princípio de que ele não ocorre de modo algum no texto. Forneça um algoritmo de tempo polinomial para descobrir se tal padrão  $P$  ocorre em um dado texto  $T$ , e analise o tempo de execução de seu algoritmo.

## 32.2 O algoritmo de Rabin-Karp

Rabin e Karp propuseram um algoritmo de emparelhamento de cadeias que funciona bem na prática e que também é generalizado para outros algoritmos de problemas relacionados, como o emparelhamento de padrões bidimensionais. O algoritmo de Rabin-Karp usa o tempo de pré-processamento  $\Theta(m)$ , e seu tempo de execução no pior caso é  $\Theta((n - m + 1)m)$ . Porém, com base em certas hipóteses, seu tempo de execução no caso médio é melhor.

Esse algoritmo faz uso de noções elementares da teoria dos números, como a equivalência de dois números módulo em um terceiro número. Talvez você queira consultar a Seção 31.1, pois ela apresenta as definições relevantes.

Para fins explicativos, vamos supor que  $\Sigma = \{0, 1, 2, \dots, 9\}$ , de forma que cada caractere seja um dígito decimal. (No caso geral, podemos supor que cada caractere é um dígito na notação de base  $d$ , onde  $d = |\Sigma|$ .) Então, podemos ver uma cadeia de  $k$  caracteres consecutivos como a representação de um número decimal de comprimento  $k$ . Assim, a cadeia de caracteres 31415 corresponde ao número decimal 31.415. Dada a interpretação dupla dos caracteres de entrada como símbolos gráficos e dígitos, achamos conveniente nesta seção denotá-los como se fossem dígitos, em nossa fonte de texto padrão.

Dado um padrão  $P[1..m]$ , seja  $p$  seu valor decimal correspondente. De modo semelhante, dado um texto  $T[1..n]$ , seja  $t_s$  o valor decimal da subcadeia de comprimento  $m$   $T[s + 1..s + m]$ , para  $s = 0, 1, \dots, n - m$ . Certamente,  $t_s = p$  se e somente se  $T[s + 1..s + m] = P[1..m]$ ; desse modo,  $s$  é um deslocamento válido se e somente se  $t_s = p$ . Se pudéssemos calcular  $p$  no tempo  $\Theta(m)$ , e todos os  $t_s$  valores em um tempo total  $\Theta(n - m + 1)$ ,<sup>1</sup> então poderíamos determinar todos os deslocamentos válidos  $s$  no tempo  $\Theta(m) + \Theta(n - m + 1) = \Theta(n)$ , comparando  $p$  com cada um dos valores  $t_s$ . (Por enquanto, não vamos nos preocupar com a possibilidade de que  $p$  e os valores de  $t_s$  possam ser números muito grandes.)

Podemos calcular  $p$  no tempo  $\Theta(m)$  usando a regra de Horner (ver Seção 30.1):

$$P = p[m] + 10(P[m - 1] + 10(P[m - 2] + \dots + 10(P[2] + 10P[1]) \dots)) .$$

O valor  $t_0$  pode ser calculado de modo semelhante a partir de  $T[1..m]$  no tempo  $\Theta(m)$ .

Para calcular os valores restantes  $t_1, t_2, \dots, t_{n-m}$  no tempo  $\Theta(n - m)$ , basta observar que  $t_{s+1}$  pode ser calculado a partir de  $t_s$  em tempo constante, pois

---

<sup>1</sup> Escrevemos  $\Theta(n - m + 1)$  em vez de  $\Theta(n - m)$  porque existem  $n - m + 1$  valores diferentes que  $s$  adota. A parte “+1” é significativa em um sentido assintótico porque, quando  $m = n$ , o cálculo do único valor  $t_s$  demora o tempo  $\Theta(1)$ , e não o tempo  $\Theta(0)$ .

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]. \quad (32.1)$$

Por exemplo, se  $m = 5$  e  $t_s = 31415$ , então desejamos remover o dígito de mais alta ordem  $T[s+1] = 3$  e introduzir o novo dígito de baixa ordem (suponha que ele seja  $T[s+5+1] = 2$ ) para obter

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\ &= 14152. \end{aligned}$$

A subtração de  $10^{m-1}T[s+1]$  remove o dígito de mais alta ordem de  $t_s$ , a multiplicação do resultado por 10 desloca o número uma posição à esquerda e a adição de  $T[s+m+1]$  introduz o dígito de baixa ordem apropriado. Se a constante  $10^{m-1}$  for pré-calculada (o que pode ser feito no tempo  $O(\lg m)$  usando-se as técnicas da Seção 31.6, embora para essa aplicação um método direto de  $O(m)$  seja bastante adequado), então cada execução da equação (32.1) toma um número constante de operações aritméticas. Desse modo, podemos calcular  $p$  no tempo  $\Theta(m)$  e calcular  $t_0, t_1, \dots, t_{n-m}$  no tempo  $\Theta(n-m+1)$ , e podemos encontrar todas as ocorrências do padrão  $P[1..m]$  no texto  $T[1..n]$  com o tempo de pré-processamento  $\Theta(m)$  e o tempo de emparelhamento  $\Theta(n-m+1)$ .

A única dificuldade com esse procedimento é que  $p$  e  $t_s$  podem ser grandes demais para serem usados de forma conveniente. Se  $P$  contém  $m$  caracteres, então a hipótese de que cada operação aritmética sobre  $p$  (que tem o comprimento de  $m$  dígitos) demora um “tempo constante” é irracional. Felizmente, existe um remédio simples para esse problema, como mostra a Figura 32.5: calcule  $p$  e os módulos de  $t_s$  com um valor adequado de módulo  $q$ . Tendo em vista que o cálculo de  $p, p_0$  e da recorrência (32.1) podem todos ser executados em módulo  $q$ , vemos que  $p$  módulo  $q$  pode ser calculado em tempo  $\Theta(m)$  e todos os  $t_s$  podem ser calculados em módulo  $q$  no tempo  $\Theta(n-m+1)$ . Em geral, o módulo  $q$  é escolhido como um primo tal que  $10q$  se encaixe em uma palavra de computador, o que permite a execução de todos os cálculos necessários com aritmética de precisão simples. Normalmente, com um alfabeto  $d$ -ário  $\{0, 1, \dots, d-1\}$ , escolhemos  $q$  de modo que  $dq$  caiba em uma palavra de computador e ajustamos a equação de recorrência (32.1) para funcionar em módulo  $q$ ; assim, ela se torna

$$t_{s+1} = (d(t_s - T[s+1]b) + T[s+m+1]) \bmod q. \quad (32.2)$$

onde  $b \equiv d^{m-1}$  é o valor do dígito “1” na posição de alta ordem de uma janela de texto de  $m$  dígitos.

Contudo, agora a facilidade do trabalho em módulo  $q$  apresenta um contratempo, pois  $t_s \equiv p \pmod{q}$  não implica que  $t_s = p$ . Por outro lado, se  $t_s \not\equiv p \pmod{q}$ , então definitivamente temos que  $t_s \neq p$ , de forma que o deslocamento  $s$  não é válido. Portanto, podemos usar o teste  $t_s \equiv p \pmod{q}$  como um teste heurístico rápido para eliminar deslocamentos  $s$  não válidos. Qualquer deslocamento  $s$  para o qual  $t_s \equiv p \pmod{q}$ , deve passar por um teste adicional para ver se  $s$  é realmente válido ou apenas temos um **acerto espúrio**. Esse teste pode ser feito verificando-se explicitamente a condição  $P[1..m] = T[s+1..s+m]$ . Se  $q$  é grande o bastante, então podemos esperar que ocorram acertos espúrios com frequência pequena o suficiente para que o custo da verificação extra seja baixo.

O procedimento a seguir torna essas idéias precisas. As entradas para o procedimento são o texto  $T$ , o padrão  $P$ , a base  $d$  a utilizar (que em geral é considerada  $|\Sigma|$ ) e o primo  $q$  a empregar.

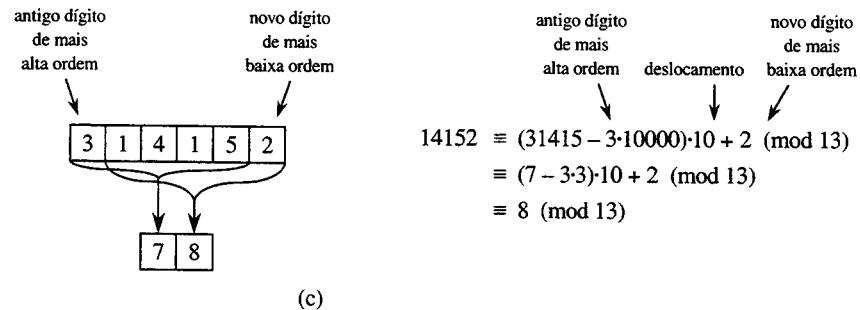
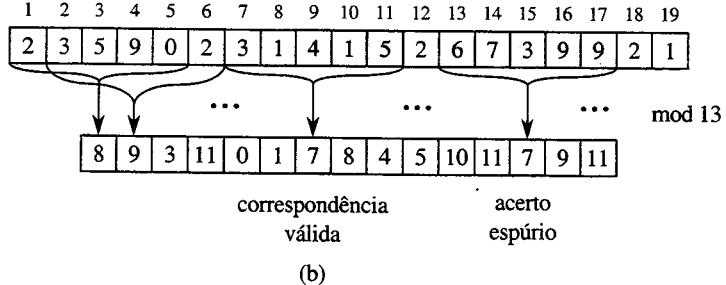
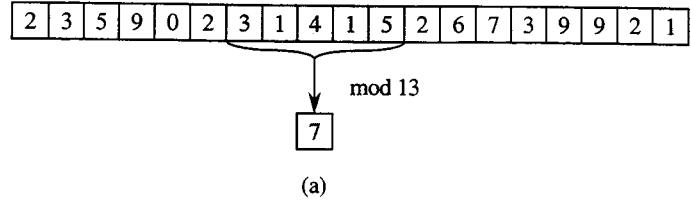


FIGURA 32.5 O algoritmo de Rabin-Karp. Cada caractere é um dígito decimal, e calculamos valores em módulo 13. (a) Uma cadeia de texto. Uma janela de comprimento 5 é mostrada com sombreada. O valor numérico do número sombreado é calculado em módulo 13, produzindo o valor 7. (b) A mesma cadeia de texto com valores calculados em módulo 13 para cada posição possível de uma janela de comprimento 5. Supondo o padrão  $P = 31415$ , procuramos por janelas cujo valor em módulo 13 seja 7, pois  $31415 \equiv 7 \pmod{13}$ . Duas dessas janelas são encontradas e elas aparecem sombreadas na figura. A primeira, começando na posição de texto 7, é de fato uma ocorrência do padrão, enquanto a segunda, que começa na posição de texto 13, é um acerto espúrio. (c) O cálculo do valor para uma janela em tempo constante, dado o valor para a janela anterior. A primeira janela tem valor 31415. Abandonando-se o dígito 3 de mais alta ordem, deslocando-se à esquerda (multiplicando-se por 10) e depois somando-se o dígito 2 de baixa ordem, resulta o novo valor 14152. Porém, todos os cálculos são executados em módulo 13, e assim o valor para a primeira janela é 7, e o valor calculado para a nova janela é 8

RABIN-KARP-MATCHER( $T, P, d, q$ )

- 1  $n \leftarrow \text{comprimento}[T]$
- 2  $m \leftarrow \text{comprimento}[P]$
- 3  $b \leftarrow d^{m-1} \pmod{q}$
- 4  $p \leftarrow 0$
- 5  $t_0 \leftarrow 0$
- 6 **for**  $i \leftarrow 1$  **to**  $m$  **do**  $\triangleright$  Pré-processamento.
- 7   **do**  $p \leftarrow (dp + P[i]) \pmod{q}$
- 8   **do**  $t_0 \leftarrow (dt_0 + T[i]) \pmod{q}$
- 9 **for**  $s \leftarrow 0$  **to**  $n - m$  **do**  $\triangleright$  Emparelhamento.
- 10   **do if**  $p = t_s$
- 11      **then if**  $P[1..m] = T[s + 1..s + m]$

```

12      then "Padrão ocorre com deslocamento"  $s$ 
13      if  $s < n - m$ 
14          then  $t_{s+1} \leftarrow (d(t_s - T[s+1]b) + T[s+m+1]) \bmod q$ 

```

O procedimento RABIN-KARP-MATCHER funciona da maneira descrita a seguir. Todos os caracteres são interpretados como dígitos de base  $d$ . Os subscritos em  $t$  são fornecidos apenas por questão de clareza; o programa funciona corretamente se todos os subscritos forem retirados. A linha 3 inicializa  $b$  com o valor da posição de dígito de mais alta ordem de uma janela de  $m$  dígitos. As linhas 4 a 8 calculam  $p$  como o valor de  $P[1..m] \bmod q$  e  $t_0$  como o valor de  $T[1..m] \bmod q$ . O loop **for** das linhas 9 a 14 iteração através de todos os deslocamentos  $s$  possíveis, mantendo o seguinte invariante:

Sempre que a linha 10 é executada,  $t_s = T[s+1..s+m] \bmod q$ .

Se  $p = t_s$  na linha 10 (um “acerto”), então verificamos se  $P[..m] = T[s+1..s+m]$  na linha 11 para eliminar a possibilidade de um acerto espúrio. Quaisquer deslocamentos válidos encontrados são impressos na linha 12. Se  $s < n - m$  (verificado na linha 13), então o loop **for** deve ser executado no mínimo mais uma vez, e assim a linha 14 é executada primeiro para assegurar que o loop invariante será válido quando a linha 10 for novamente alcançada. A linha 14 calcula o valor de  $t_{s+1} \bmod q$  a partir do valor de  $t_s \bmod q$  em tempo constante, usando a equação (32.2) diretamente.

RABIN-KARP-MATCHER demora o tempo de processamento  $\Theta(m)$  e o tempo de emparelhamento é  $\Theta((n-m+1)m)$  no pior caso, pois (como no algoritmo simples de emparelhamento de cadeias) o algoritmo de Rabin-Karp verifica explicitamente todo deslocamento válido. Se  $P = a^m$  e  $T = a^n$ , então as verificações demoram o tempo  $\Theta((n-m+1)m)$ , pois cada um dos  $n-m+1$  deslocamentos possíveis é válido.

Em muitas aplicações, esperamos alguns deslocamentos válidos (talvez alguma constante  $c$  deles); nessas aplicações, o tempo de emparelhamento esperado do algoritmo é apenas  $O((n-m+1) + cm) = O(n+m)$ , mais o tempo necessário para processar acertos espúrios. Podemos basear uma análise heurística na hipótese de que a redução de valores em módulo  $q$  atua como um mapeamento aleatório de  $\Sigma^*$  para  $Z_q$ . (Veja a discussão sobre o uso da divisão para hash na Seção 11.3.1. É difícil formalizar e provar tal hipótese, embora uma abordagem viável seja supor que  $q$  é escolhido de forma aleatória a partir de inteiros de tamanho apropriado. Não procuraremos essa formalização aqui.) Então, podemos esperar que o número de acertos espúrios seja  $O(n/q)$ , pois a chance de que um  $t_s$ , arbitrário seja equivalente a  $p$ , módulo  $q$ , pode ser estimada como  $1/q$ . Tendo em vista que existem  $O(n)$  posições em que o teste da linha 10 falha e gastamos o tempo  $O(m)$  para cada acerto, o tempo de emparelhamento esperado do algoritmo de Rabin-Karp é

$$O(n) + O(m(v + n/q)) ,$$

onde  $v$  é o número de deslocamentos válidos. Esse tempo de execução é  $O(n)$  se  $v = O(1)$  escolhemos  $q \geq m$ . Isto é, se o número esperado de deslocamentos válidos é pequeno ( $O(1)$ ) e o primo  $q$  é escolhido com tamanho maior que o comprimento do padrão, então podemos esperar que o procedimento de Rabin-Karp utilize apenas o tempo de emparelhamento  $O(n+m)$ . Como  $m \leq n$ , esse tempo de emparelhamento esperado é  $O(n)$ .

## Exercícios

### 32.2-1

Trabalhando em módulo  $q = 11$ , quantos acertos espúrios o combinador de Rabin-Karp encontra no texto  $T = 3141592653589793$  ao procurar pelo padrão  $P = 26$ ? 724

### 32.2-2

Como você estenderia o método de Rabin-Karp ao problema de pesquisar uma cadeia de texto em busca de uma ocorrência de qualquer padrão de um dado conjunto de  $k$  padrões? Comece supondo que todos os  $k$  padrões têm o mesmo comprimento. Em seguida, generalize sua solução para permitir que os padrões tenham comprimentos diferentes.

### 32.2-3

Mostre como estender o método de Rabin-Karp para tratar o problema de procurar por um dado padrão  $m \times m$  em um arranjo de caracteres  $n \times n$ . (O padrão pode ser deslocado vertical e horizontalmente, mas não pode ser girado.)

### 32.2-4

Alice tem uma cópia de um longo arquivo de  $n$  bits  $A = \langle a_{n-1}, a_{n-2}, \dots, a_0 \rangle$ , e Bob tem um arquivo semelhante de  $n$  bits  $B = \langle b_{n-1}, b_{n-2}, \dots, b_0 \rangle$ . Alice e Bob desejam saber se seus arquivos são idênticos. Para evitar transmitir todo o arquivo  $A$  ou  $B$ , eles usam a verificação probabilística rápida a seguir. Juntos, eles selecionam um primo  $q > 1000n$  e selecionam aleatoriamente um inteiro  $x$  a partir de  $\{0, 1, \dots, q - 1\}$ . Em seguida, Alice avalia

$$A(x) = \left( \sum_{i=0}^n a_i x^i \right) \bmod q,$$

e Bob avalia de modo semelhante  $B(x)$ . Prove que, se  $A \neq B$ , existe no máximo uma chance em 1000 de que  $A(x) = B(x)$ ; por outro lado, se os dois arquivos forem iguais,  $A(x)$  será necessariamente igual a  $B(x)$ . (Sugestão: Veja o Exercício 31.4-4.)

## 32.3 Emparelhamento de cadeias com autômatos finitos

Muitos algoritmos de emparelhamento de cadeias constroem um autômato finito que busca na cadeia de texto  $T$  todas as ocorrências do padrão  $P$ . Esta seção apresenta um método para construir tal autômato. Esses autômatos de emparelhamento de cadeias são muito eficientes: eles examinam cada caractere de texto *exatamente uma vez*, levando um tempo constante por caractere de texto. O tempo de emparelhamento usado – depois do pré-processamento do padrão para construir o autômato – é então  $\Theta(n)$ . Porém, o tempo para construir o autômato pode ser grande, se  $\Sigma$  é grande. A Seção 32.4 descreve um modo inteligente de contornar esse problema.

Começamos esta seção com a definição de um autômato finito. Em seguida, examinaremos um autômato especial de emparelhamento de cadeias e mostraremos como ele pode ser usado para encontrar ocorrências de um padrão em um texto. Essa discussão inclui detalhes sobre como simular o comportamento de um autômato de emparelhamento de cadeias sobre um dado texto. Finalmente, mostraremos como construir o autômato de emparelhamento de cadeias para um determinado padrão de entrada.

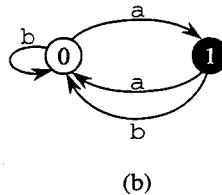
### Autômatos finitos

Um **autômato finito**  $M$  é uma 5-tupla  $(Q, q_0, A, \Sigma, \delta)$ , onde

- $Q$  é um conjunto finito de **estados**,
- $q_0 \in Q$  é o **estado inicial**,
- $A \subseteq Q$  é um conjunto distinto de **estados aceitáveis**,
- $\Sigma$  é um **alfabeto de entrada** finito,
- $\delta$  é uma função de  $Q \times \Sigma$  em  $Q$ , chamada **função de transição** de  $M$ .

| estado | a | b |
|--------|---|---|
| 0      | 1 | 0 |
| 1      | 0 | 0 |

(a)



(b)

FIGURA 32.6 Um autômato finito simples de dois estados com o conjunto de estados  $Q = \{0, 1\}$ , estado inicial  $q_0 = 0$  e alfabeto de entrada  $\Sigma = \{a, b\}$ . (a) Uma representação tabular da função de transição  $\delta$ . (b) Um diagrama de transição de estados equivalente. O estado 1 é o único estado aceitável (mostrado escurecido). Arestras orientadas representam transições. Por exemplo, a aresta do estado 1 para o estado 0 identificada por b indica  $\delta(1, b) = 0$ . Esse autômato aceita as cadeias que terminam em um número ímpar de valores a. Mais precisamente, uma cadeia  $x$  é aceita se e somente se  $x = yz$ , onde  $y = \varepsilon$  ou  $y$  termina com um b, e  $z = a^k$ , onde  $k$  é ímpar. Por exemplo, a seqüência de estados que esse autômato introduz para a entrada abaaa (incluindo o estado inicial) é  $\langle 0, 1, 0, 1, 0, 1 \rangle$ , e assim ele aceita essa entrada. Para a entrada abbaa, a seqüência de estados é  $\langle 0, 1, 0, 0, 1, 0 \rangle$ , e portanto ele rejeita essa entrada

O autômato finito começa no estado  $q_0$  e lê os caracteres de sua cadeia de entrada um de cada vez. Se o autômato estiver no estado  $q$  e ler o caractere de entrada  $a$ , ele se moverá (“fará uma transição”) do estado  $q$  para o estado  $\delta(q, a)$ . Sempre que seu estado atual  $q$  é um elemento de  $A$ , dizemos que a máquina *M aceitou* a cadeia lida até agora. Uma entrada que não seja aceita é dita *rejeitada*. A Figura 32.6 ilustra essas definições com um autômato simples de dois estados.

Um autômato finito  $M$  induz uma função  $\phi$ , chamada *função de estado final*, desde  $\Sigma^*$  até  $Q$ , tal que  $\phi(w)$  é o estado em que  $M$  termina depois de examinar a cadeia  $w$ . Desse modo,  $M$  aceita uma cadeia  $w$  se e somente se  $\phi(w) \in A$ . A função  $\phi$  é definida pela relação recursiva

$$\begin{aligned}\phi(\varepsilon) &= q_0, \\ \phi(wa) &= \delta(\phi(w), a) \text{ para } w \in \Sigma^*, a \in \Sigma.\end{aligned}$$

## Autômatos de emparelhamento de cadeias

Existe um autômato de emparelhamento de cadeias para cada padrão  $P$ ; esse autômato deve ser construído a partir do padrão em uma etapa de pré-processamento, antes de poder ser usado para pesquisar a cadeia de texto. A Figura 32.7 ilustra essa construção para o padrão  $P$  ababaca. De agora em diante, faremos a suposição de que  $P$  é um determinada cadeia de padrão fixo; por brevidade, não indicaremos a dependência de  $P$  em nossa notação.

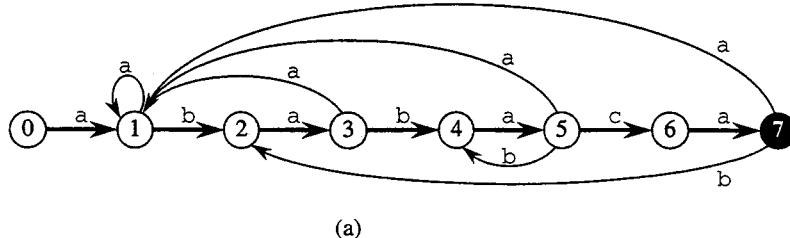
A fim de especificar o autômato de emparelhamento de cadeias relacionado a um determinado padrão  $P[1..m]$ , primeiro definimos uma função auxiliar  $\sigma$ , chamada *função sufixo* que corresponde a  $P$ . A função  $\sigma$  é um mapeamento de  $\Sigma^*$  para  $\{0, 1, \dots, m\}$  tal que  $\sigma(x)$  é o comprimento do mais longo prefixo de  $P$  que é um sufixo de  $x$ :

$$\sigma(x) = \max \{k : P_k \sqsupseteq x\}.$$

A função sufixo  $\sigma$  é bem definida, pois a cadeia vazia  $P_0 = \varepsilon$  é um sufixo de toda cadeia. Como exemplos, para o padrão  $P = ab$ , temos  $\sigma(\varepsilon) = 0$ ,  $\sigma(ccaca) = 1$  e  $\sigma(ccab) = 2$ . Para um padrão  $P$  de comprimento  $m$ , temos  $\sigma(x) = m$  se e somente se  $\square$ . Segue-se da definição da função sufixo que, se  $\square$ , então  $\sigma(x) \leq \sigma(y)$ .

Definimos o autômato de emparelhamento de cadeias equivalente a um dado padrão  $P[1 .. m]$  como a seguir.

- O conjunto de estados  $Q$  é  $\{0, 1, \dots, m\}$ . O estado inicial  $q_0$  é o estado 0, e estado  $m$  é o único estado aceitável.



(a)

| estado | entrada |   |   | $P$ |
|--------|---------|---|---|-----|
|        | a       | b | c |     |
| 0      | 1       | 0 | 0 | a   |
| 1      | 1       | 2 | 0 | b   |
| 2      | 3       | 0 | 0 | a   |
| 3      | 1       | 4 | 0 | b   |
| 4      | 5       | 0 | 0 | a   |
| 5      | 1       | 4 | 6 | c   |
| 6      | 7       | 0 | 0 | a   |
| 7      | 1       | 2 | 0 |     |

(b)

| $i$         | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|-------------|---|---|---|---|---|---|---|---|---|---|----|----|
| $T[i]$      | - | a | b | a | b | a | b | a | c | a | b  | a  |
| $\phi(T_i)$ | 0 | 1 | 2 | 3 | 4 | 5 | 4 | 5 | 6 | 7 | 2  | 3  |

(c)

FIGURA 32.7 (a) Um diagrama de transição de estados para o autômato de emparelhamento de cadeias que aceita todos as cadeias que terminam com a cadeia ababaca. O estado 0 é o estado inicial, e o estado 7 (mostrado escurecido) é o único estado aceitável. Uma aresta orientada do estado  $i$  para o estado  $j$  identificada por  $a$  representa  $\delta(i, a) = j$ . As arestas que vão para a direita formando a “espinha dorsal” do autômato, mostradas mais grossas na figura, correspondem a comparações bem-sucedidas entre caracteres do padrão e da entrada. As arestas que vão para a esquerda correspondem a comparações malsucedidas. Algumas arestas correspondentes a comparações malsucedidas não são mostradas; por convenção, se um estado  $i$  não tem nenhuma aresta de saída identificada por  $a$  para algum  $a \in \Sigma$ , então  $\delta(i, a) = 0$ . (b) A função de transição  $\delta$  correspondente, e a cadeia de padrão  $P = ababaca$ . As entradas correspondentes a comparações bem-sucedidas entre os caracteres do padrão e de entrada aparecem sombreadas. (c) A operação do autômato sobre o texto  $T = abababacaba$ . Sob cada caractere de texto  $T[i]$  é dado o estado  $\phi(T_i)$  em que o autômato está depois de processar o prefixo  $T_i$ . Uma ocorrência do padrão foi encontrada, terminando na posição 9

- A função de transição  $\delta$  é definida pela seguinte equação, para qualquer estado  $q$  e caractere  $a$ :

$$\delta(q, a) = \delta(P_q a). \quad (32.3)$$

Aqui está uma boa razão intuitiva para a definição de  $\delta(q, a) = \delta(P_q a)$ . A máquina mantém como um invariante de sua operação que

$$\phi(T_i) = \sigma(T_i) \quad (32.4)$$

esse resultado é provado no Teorema 32.4 a seguir. Em outras palavras, isso significa que, depois de examinar os  $i$  primeiros caracteres da cadeia de texto  $T$ , a máquina se encontra no estado  $\phi(T_i) = q$ , onde  $q = \sigma(T_i)$  é o comprimento do mais longo sufixo de  $T_i$  que também é um prefixo do padrão  $P$ . Se o próximo caractere examinado é  $T[i + 1] = a$ , então a máquina deve fazer uma transição para o estado  $\sigma(T_{i+1}) = \sigma(T_i a)$ . A prova do teorema mostra que  $\sigma(T_i a) = \sigma(P_q a)$ . Isto é, para calcular o comprimento do sufixo mais longo de  $T_i a$  que é um prefixo de  $P$ , podemos calcular o sufixo mais longo de  $P_q a$  que é um prefixo de  $P$ . Em cada estado, a máquina só precisa conhecer o comprimento do prefixo mais longo de  $P$  que é um sufixo do que foi lido até agora. Então, a definição de  $\delta(q, a) = \sigma(P_q a)$  mantém o invariante desejado (32.4). Esse argumento informal se tornará rigoroso em breve.

Por exemplo, no autômato de emparelhamento de cadeias da Figura 32.7, temos  $\delta(5, b) = 4$ . Isso decorre do fato de que, se o autômato lê um  $b$  no estado  $q = 5$ , então  $P_q b = ababab$ , e o prefixo mais longo de  $P$  que também é um sufixo de  $ababab$  é  $P_4 = abab$ .

Para esclarecer a operação de um autômato de emparelhamento de cadeias, daremos agora um programa simples e eficiente para simular o comportamento de um autômato desse tipo (representado por sua função de transição  $\delta$ ) ao encontrar ocorrências de um padrão  $P$  de comprimento  $m$  em um texto de entrada  $T[1..n]$ . Como ocorre em qualquer autômato de emparelhamento de cadeias para um padrão de comprimento  $m$ , o conjunto de estados  $Q$  é  $\{0, 1, \dots, m\}$ , o estado inicial é 0 e o único estado aceitável é o estado  $m$ .

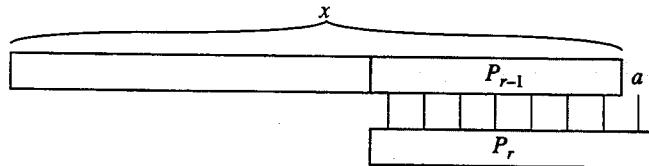


FIGURA 32.8 Uma ilustração para a prova do Lema 32.2. A figura mostra que  $r \leq \sigma(x) + 1$ , onde  $r = \sigma(xa)$

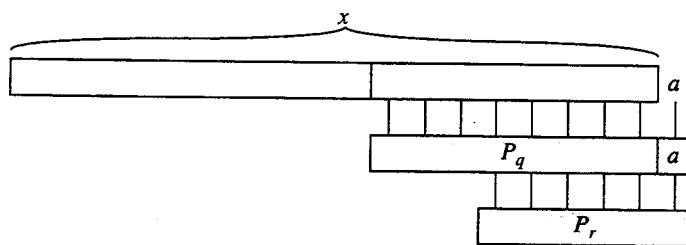


FIGURA 32.9 Uma ilustração para a prova do Lema 32.3. A figura mostra que  $r = \sigma(P_q a)$ , onde  $q = \sigma(x)$  e  $r = \sigma(xa)$

#### FINITE-AUTOMATON-MATCHER( $T, \delta, m$ )

```

1  $n \leftarrow \text{comprimento}[T]$ 
2  $q \leftarrow 0$ 
3 for  $i \leftarrow 1$  to  $n$ 
4   do  $q \leftarrow \delta(q, T[i])$ 
5   if  $q = m$ 
6     then imprimir "Padrão ocorre com deslocamento"  $i - m$ 

```

A estrutura de loop simples de FINITE-AUTOMATON-MATCHER implica que seu tempo de execução sobre uma cadeia de texto de comprimento  $n$  é  $\Theta(n)$ . Porém, esse tempo de emparelhamento não inclui o tempo de pré-processamento necessário para calcular a função de transição  $\delta$ . Trataremos desse problema mais adiante, depois de provar que o procedimento FINITE-AUTOMATON-MATCHER opera corretamente.

Considere a operação do autômato sobre um texto de entrada  $T[1..n]$ . Provaremos que o autômato está no estado  $\sigma(T_i)$  depois de examinar o caractere  $T[i]$ . Tendo em vista que  $\sigma(T_i) = m$  se e somente se  $P \sqsupseteq T_i$ , a máquina está no estado aceitável  $m$  se e somente se o padrão  $P$  acabou de ser examinado. Para provar esse resultado, fazemos uso dos dois lemas a seguir sobre a função sufixo  $\sigma$ .

#### Lema 32.2 (Desigualdade da função sufixo)

Para qualquer cadeia  $x$  e caractere  $a$ , temos  $\sigma(xa) \leq \sigma(x) + 1$ .

**Prova** Em relação à Figura 32.8, seja  $r = \sigma(xa)$ . Se  $r = 0$ , então a conclusão  $\sigma(xa) = r \leq \sigma(x) + 1$  é trivialmente satisfeita, graças ao caráter não negativo de  $\sigma(x)$ . Assim, suponha que  $r > 0$ . Agora,

$P_r \sqsupseteq xa$ , pela definição de  $\sigma$ . Desse modo,  $P_{r-1} \sqsupseteq x$ , retirando-se  $a$  do final de  $P_r$  e do final de  $xa$ . Então,  $r-1 \leq \sigma(x)$ , pois  $\sigma(x)$  é o maior  $k$  tal que  $\sqsupseteq$ , e  $\sigma(xa) = r \leq \sigma(x) + 1$ . ■

### Lema 32.3 (Lema de recursão da função sufixo)

Para qualquer cadeia  $x$  e caractere  $a$ , se  $q = \sigma(x)$ , então  $\sigma(xa) = \sigma(P_q a)$ .

**Prova** Da definição de  $\sigma$ , temos  $P_q \sqsupseteq x$ . Como mostra a Figura 32.9, também temos  $P_q a \sqsupseteq xa$ . Se fizermos  $r = \sigma(xa)$ , então  $r \leq q + 1$  pelo Lema 32.2. Tendo em vista que  $P_q a \sqsupseteq xa$ ,  $P_r \sqsupseteq xa$  e  $|P_r| \leq |P_q a|$ , o Lema 32.1 implica  $P_r \sqsupseteq P_q a$ . Então,  $r \leq \sigma(P_q a)$ , ou seja,  $\sigma(xa) \leq \sigma(P_q a)$ . Porém, também temos  $\sigma(P_q a) \leq \sigma(P_q a)$ , pois  $P_q a \sqsupseteq xa$ . Assim,  $\sigma(xa) = \sigma(P_q a)$ .

Agora estamos prontos para provar nosso principal teorema que caracteriza o comportamento de um autômato de emparelhamento de cadeias sobre um dado texto de entrada. Como observamos antes, esse teorema mostra que o autômato está simplesmente controlando, em cada passo, o prefixo mais longo do padrão que é um sufixo do que foi lido até o momento. Em outras palavras, o autômato mantém o invariante (32.4).

### Teorema 32.4

Se  $\phi$  é a função do estado final de um autômato de emparelhamento de cadeias para um dado padrão  $P$ , e  $T[1 .. n]$  é um texto de entrada para o autômato, então

$$\phi(T_i) = \sigma(T_i)$$

para  $i = 0, 1, \dots, n$ .

**Prova** A prova é por indução sobre  $i$ . Para  $i = 0$ , o teorema é trivialmente verdadeiro, pois  $T_0 = \varepsilon$ . Portanto,  $\phi(T_0) = 0 = \sigma(T_0)$ .

Agora, supomos que  $\phi(T_i) = \sigma(T_i)$  e provamos que  $\phi(T_{i+1}) = \sigma(T_{i+1})$ . Seja  $q$  a representação de  $\phi(T_i)$ , e seja  $a$  a representação de  $T[i+1]$ . Então,

$$\begin{aligned} \phi(T_{i+1}) &= \phi(T_i a) && (\text{pelas definições de } T_{i+1} \text{ e } a) \\ &= \delta(\phi(T_i), a) && (\text{pela definição de } \phi) \\ &= \delta(q, a) && (\text{pela definição de } q) \\ &= \sigma(P_q a) && (\text{pela definição (32.3) de } \delta) \\ &= \sigma(T_i a) && (\text{pelo Lema 32.3 e por indução}) \\ &= \sigma(T_{i+1}) && (\text{pela definição de } T_{i+1}). \end{aligned}$$

Pelo Teorema 32.4, se a máquina entra no estado  $q$  na linha 4, então  $q$  é o maior valor tal que  $P_q \sqsupseteq T_i$ . Desse modo, temos  $q = m$  na linha 5 se e somente se uma ocorrência do padrão  $P$  acabou de ser examinada. Concluímos que FINITE-AUTOMATON-MATCHER opera corretamente.

### Cálculo da função de transição

O procedimento a seguir calcula a função de transição  $\delta$  a partir de um determinado padrão  $P[1 .. m]$ .

### COMPUTE-TRANSITION-FUNCTION( $P, \Sigma$ )

```
1  $m \leftarrow \text{comprimento}[P]$ 
2 for  $q \leftarrow 0$  to  $m$ 
3   do for cada caractere  $a \in \Sigma$ 
4     do  $k \leftarrow \min(m + 1, q + 2)$ 
5     repeat  $k \leftarrow k - 1$ 
6     until  $P_k \sqsupseteq P_q a$ 
7      $\delta(q, a) \leftarrow k$ 
8 return  $\delta$ 
```

Esse procedimento calcula  $\delta(q, a)$  de maneira direta, de acordo com sua definição. Os loops aninhados que começam nas linhas 2 e 3 consideram todos os estados  $q$  e caracteres  $a$ , e as linhas 4 a 7 definem  $\delta(q, a)$  como o maior  $k$  tal que  $P_k \sqsupseteq P_q a$ . O código começa com o maior valor concebível de  $k$ , que é  $\min(m, q + 1)$  e diminui  $k$  até  $P_k \sqsupseteq P_q a$ .

O tempo de execução de COMPUTE-TRANSITION-FUNCTION é  $O(m^3 |\Sigma|)$ , porque os loops exteriores contribuem com um fator  $m |\Sigma|$ , o loop **repeat** mais interno pode ser executado no máximo  $m + 1$  vezes, e o teste  $P_k \sqsupseteq P_q a$  da linha 6 pode exigir a comparação de até  $m$  caracteres. Existem procedimentos muito mais rápidos; o tempo necessário para calcular  $\delta$  a partir de  $P$  pode ser melhorado até  $O(m |\Sigma|)$ , utilizando-se algumas informações calculadas de modo inteligente sobre o padrão  $P$  (ver Exercício 32.4-6). Com esse procedimento otimizado para calcular  $\delta$ , podemos encontrar todas as ocorrências de um padrão de comprimento  $m$  em um texto de comprimento  $n$  sobre um alfabeto  $\delta$  com o tempo de pré-processamento  $O(m |\Sigma|)$  e o tempo de emparelhamento  $\Theta(n)$ .

## Exercícios

### 32.3-1

Construa o autômato de emparelhamento de cadeias para o padrão  $P = \text{aabab}$  e ilustre sua operação sobre a cadeia de texto  $T = \text{aaababaabaababaab}$ .

### 32.3-2

Desenhe um diagrama de transição de estados para um autômato de emparelhamento de cadeias referente ao padrão  $\text{ababbabbababbababbabb}$  sobre o alfabeto  $\Sigma = \{\text{a}, \text{b}\}$ .

### 32.3-3

Dizemos que um padrão  $P$  é **não sobreposto** se  $P_k \sqsupseteq P_q$  implica  $k = 0$  ou  $k = q$ . Descreva o diagrama de transição de estados do autômato de emparelhamento de cadeias para um padrão não sobreposto.

### 32.3-4 \*

Dados dois padrões  $P$  e  $P'$ , descreva como construir um autômato finito que determine todas as ocorrências de *um ou outro* padrão. Procure minimizar o número de estados em seu autômato.

### 32.3-5

Dado um padrão  $P$  contendo caracteres lacuna (ver Exercício 32.1-4), mostre como construir um autômato finito que possa encontrar uma ocorrência de  $P$  em um texto  $T$  no tempo de emparelhamento  $O(n)$ , onde  $n = |T|$ .

## ★ 32.4 O algoritmo de Knuth-Morris-Pratt

Agora, apresentaremos um algoritmo de emparelhamento de cadeias de tempo linear criado por Knuth, Morris e Pratt. Seu algoritmo evita por completo o cálculo da função de transição  $\delta$ , e seu tempo de emparelhamento é  $\Theta(n)$  usando apenas uma função auxiliar  $\pi(1..m)$  pré-calculada a

partir do padrão no tempo  $\Theta(m)$ . O arranjo  $\pi$  permite que a função de transição  $\pi$  seja calculada de modo eficiente (em um sentido amortizado) “durante a execução” conforme necessário. Em termos aproximados, para qualquer estado  $q = 0, 1, \dots, m$  e qualquer caractere  $a \in \Sigma$ , o valor  $\pi[q]$  contém as informações que são independentes de  $a$  e são necessárias para calcular  $\delta(q, a)$ . (Essa observação será esclarecida em breve.) Tendo em vista que o arranjo  $\pi$  tem apenas  $m$  entradas, enquanto  $\delta$  tem  $\Theta(m |\Sigma|)$  entradas, economizamos um fator  $|\Sigma|$  no tempo de pré-processamento, calculando  $\pi$  em vez de  $\delta$ .

## A função de prefixo para um padrão

A função de prefixo  $\pi$  para um padrão encapsula o conhecimento sobre o modo como o padrão se compara com os deslocamentos dele próprio. Essa informação pode ser usada para evitar testes de deslocamentos inúteis no algoritmo simples de correspondência de padrões, ou para evitar o cálculo prévio de  $\delta$  para um autômato de emparelhamento de cadeias.

Considere a operação do combinador de cadeias simples. A Figura 32.10(a) mostra um determinado deslocamento  $s$  de um gabarito contendo o padrão  $P = ababaca$  com um texto  $T$ . Para esse exemplo,  $q = 5$  dos caracteres foram comparados com sucesso, mas o sexto caractere do padrão não coincidiu com o caractere de texto correspondente. A informação de que  $q$  caracteres foram comparados com sucesso determina os caracteres de texto correspondentes. Conhecer esses  $q$  caracteres de texto nos permite determinar de imediato que certos deslocamentos não são válidos. No exemplo da figura, o deslocamento  $s + 1$  é necessariamente não-válido, pois o primeiro caractere do padrão (a) estaria alinhado com um caractere de texto que sabemos que coincide com o segundo caractere do padrão (b). Porém, o deslocamento  $s' = s + 2$  mostrado na parte (b) da figura, alinha os três primeiros caracteres do padrão com três caracteres de texto que devem necessariamente coincidir. Em geral, é útil saber a resposta para a seguinte questão:

Dado que caracteres do padrão  $P[1 \dots q]$  correspondem a caracteres de texto  $T[s + 1 \dots s + q]$ , qual é o menor deslocamento  $s' > s$  tal que

$$P[1 \dots k] = T[s' + 1 \dots s' + k], \quad (32.5)$$

onde  $s' + k = s + q$ ?

Tal deslocamento  $s'$  é o primeiro deslocamento maior que  $s$  que não é necessariamente não válido devido a nosso conhecimento de  $T[s + 1 \dots s + q]$ . No melhor caso, temos que  $s' = s + q$ , e deslocamentos  $s + 1, s + 2, \dots, s + q - 1$  são todos eliminados de imediato. Em qualquer caso, no novo deslocamento  $s'$ , não precisamos comparar os  $k$  primeiros caracteres de  $P$  com os caracteres correspondentes de  $T$ , pois temos a garantia de que eles coincidem pela equação (32.5).

As informações necessárias podem ser pré-calculadas comparando-se o padrão com ele próprio, como ilustra a Figura 32.10(c). Tendo em vista que  $T[s' + 1 \dots s' + k]$  faz parte da porção conhecida do texto, ele é um sufixo da cadeia  $P_q$ . A equação (32.5) pode então ser interpretada como a solicitação do maior  $k < q$  tal que  $P_k \sqsupseteq P_q$ . Então,  $s' = s + (q - k)$  é o próximo deslocamento potencialmente válido. Como resultado, é conveniente armazenar o número  $k$  de caracteres coincidentes no novo deslocamento  $s'$ , em vez de armazenar, digamos,  $s' - s$ . Essa informação pode ser usada para acelerar tanto o algoritmo simples de emparelhamento de cadeias quanto o combinador do autômato finito.

Formalizamos a pré-computação necessária da maneira descrita a seguir. Dado um padrão  $P[1..m]$ , a **função prefixo** para o padrão  $P$  é a função  $\pi : \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m - 1\}$  tal que

$$\pi[q] = \max\{k : K < q \text{ e } P_k \sqsupseteq P_q\}.$$

Isto é,  $\pi[q]$  é o comprimento do prefixo mais longo de  $P$  que é um sufixo próprio de  $P_q$ . Como outro exemplo, a Figura 32.11(a) fornece a função prefixo completa  $\pi$  para o padrão ababababca.

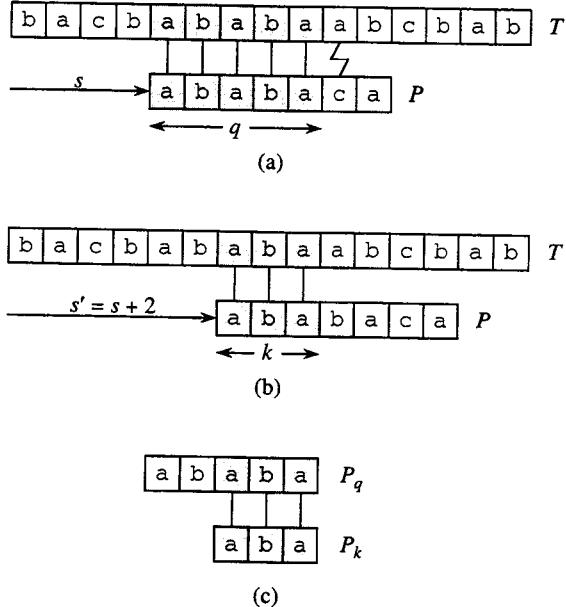


FIGURA 32.10 A função prefixo  $\pi$ . (a) O padrão  $P = ababaca$  está alinhado com um texto  $T$  de modo que os primeiros  $q = 5$  caracteres coincidem. Os caracteres correspondentes, que aparecem sombreados, estão conectados por linhas verticais. (b) Usando apenas nosso conhecimento dos 5 caracteres correspondentes, podemos deduzir que um deslocamento  $s + 1$  é não válido, mas que um deslocamento de  $s' = s + 2$  é coerente com tudo que sabemos sobre o texto, e portanto é potencialmente válido. (c) As informações úteis para tais deduções podem ser pré-calculadas comparando-se o padrão com ele próprio. Aqui, vemos que o prefixo mais longo de  $P$  que também é um sufixo de  $P_5$  é  $P_3$ . Essa informação é pré-calculada e representada no arranjo  $\pi$ , de modo que  $\pi[5] = 3$ . Considerando-se que  $q$  caracteres foram comparados com sucesso no deslocamento  $s$ , o próximo deslocamento potencialmente válido está em  $s' = s + (q - \pi[q])$ .

| $i$      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----------|---|---|---|---|---|---|---|---|---|----|
| $P[i]$   | a | b | a | b | a | b | a | b | c | a  |
| $\pi[i]$ | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 0 | 1  |

(a)

|       |            |                     |              |
|-------|------------|---------------------|--------------|
| $P_8$ | ababababab | c a                 |              |
| $P_6$ | abababab   | a b c a             | $\pi[8] = 6$ |
| $P_4$ | ababab     | a b a b c a         | $\pi[6] = 4$ |
| $P_2$ | ab         | a b a b a b c a     | $\pi[4] = 2$ |
| $P_0$ | Z          | a b a b a b a b c a | $\pi[2] = 0$ |

(b)

FIGURA 32.11 Uma ilustração do Lema 32.5 para o padrão  $P = ababababca$  e  $q = 8$ . (a) A função  $\pi$  para o padrão dado. Tendo em vista que  $\pi[8] = 6$ ,  $\pi[6] = 4$ ,  $\pi[4] = 2$  e  $\pi[2] = 0$ , pela iteração de  $\pi$  obtemos  $\pi^*[8] = \{6, 4, 2, 0\}$ . (b) Deslizamos o gabarito contendo o padrão  $P$  para a direita e observamos quando algum prefixo  $P_k$  de  $P$  coincide com algum sufixo próprio de  $P_8$ ; isso acontece para  $k = 6, 4, 2$  e  $0$ . Na figura, a primeira linha fornece  $P$ , e a linha vertical pontilhada é desenhada logo após  $P_8$ . Linhas sucessivas mostram todos os deslocamentos de  $P$  que fazem algum prefixo  $P_k$  de  $P$  corresponder a algum sufixo de  $P_8$ . Caracteres comparados com sucesso aparecem sombreados. Linhas verticais conectam caracteres correspondentes alinhados. Desse modo,  $\{k : K < q \text{ e } P_k \sqsupseteq P_q\} = \{6, 4, 2, 0\}$ . O lema afirma que  $\pi^*[q] = \{k : K < q \text{ e } P_k \sqsupseteq P_q \text{ para todo } q\}$

O algoritmo de emparelhamento de Knuth-Morris-Pratt é dado no pseudocódigo a seguir como o procedimento KMP-MATCHER. Ele é modelado em sua maior parte com base em FINITE-AUTOMATON-MATCHER, como veremos. KMP-MATCHER chama o procedimento auxiliar COMPUTE-PREFIX-FUNCTION para calcular  $\pi$ . ]

#### KMP-MATCHER( $T, P$ )

```

1  $n \leftarrow \text{comprimento}[T]$ 
2  $m \leftarrow \text{comprimento}[P]$ 
3  $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4  $q \leftarrow 0$                                  $\triangleright$  Número de caracteres correspondentes.
5 for  $i \leftarrow 1$  to  $n$                    $\triangleright$  Varre o texto da esquerda para a direita.
6 do while  $q > 0$  e  $P[q + 1] \neq T[i]$ 
7   do  $q \leftarrow \pi[q]$                        $\triangleright$  O próximo caractere não corresponde.
8   if  $P[q + 1] = T[i]$ 
9     then  $q \leftarrow q + 1$                      $\triangleright$  O próximo caractere corresponde.
10    if  $q = m$                              $\triangleright$  Todo o  $P$  correspondeu?
11    then imprimir "Padrão ocorre com deslocamento"  $i - m$ 
12    $q \leftarrow \pi[q]$                        $\triangleright$  Procura pelo próximo emparelhamento.
```

#### COMPUTE-PREFIX-FUNCTION( $P$ )

```

1  $m \leftarrow \text{comprimento}[P]$ 
2  $\pi[1] \leftarrow 0$ 
3  $k \leftarrow 0$ 
4 for  $q \leftarrow 2$  to  $m$ 
5   do while  $k > 0$  e  $P[k + 1] \neq P[q]$ 
6     do  $k \leftarrow \pi[k]$ 
7     if  $P[k + 1] = P[q]$ 
8       then  $k \leftarrow k + 1$ 
9      $\pi[k] \leftarrow k$ 
10 return  $\pi$ 
```

Começamos com uma análise dos tempos de execução desses procedimentos. Provar que os procedimentos são corretos será mais complicado.

### Análise do tempo de execução

O tempo de execução de COMPUTE-PREFIX-FUNCTION é  $\Theta(m)$ , usando-se o método potencial de análise amortizada (veja a Seção 17.3). Associamos um potencial  $k$  ao estado atual  $k$  do algoritmo. Esse potencial tem um valor inicial 0, pela linha 3. A linha 6 diminui  $k$  sempre que é executada, pois  $\pi[k]$ . Porém, tendo em vista que  $\pi[k] \geq 0$  para todo  $k$ ,  $k$  nunca poderá se tornar negativo. Além dessa, a única linha que afeta  $k$  é a linha 8, que aumenta  $k$  em no máximo uma unidade durante cada execução do corpo do loop **for**. Como  $k < q$  na entrada do loop **for**, e como  $q$  é incrementado em cada iteração do corpo do loop **for**,  $k < q$  é sempre válida. (Isso também justifica a afirmação de que  $\pi[q] < q$ , pela linha 9.) Podemos compensar cada execução do corpo do loop **while** na linha 6 com a diminuição correspondente na função potencial, pois  $\pi[q]$ . A linha 8 aumenta a função potencial em no máximo uma unidade, de forma que o custo amortizado do corpo do loop nas linhas 5 a 9 é  $O(1)$ . Considerando-se que o número de iterações do loop mais externo é  $\Theta(m)$ , e como a função potencial final é no mínimo tão grande quanto a função potencial inicial, o tempo de execução real total no pior caso de COMPUTE-PREFIX-FUNCTION é  $\Theta(m)$ .

Uma análise amortizada semelhante, usando o valor de  $q$  como função potencial, mostra que o tempo de emparelhamento de KMP-MATCHER é  $\Theta(n)$ .

Em comparação com FINITE-AUTOMATON-MATCHER, usando  $\pi$  em lugar de  $\delta$ , reduzimos o tempo para pré-processar o padrão de  $O(m |\Sigma|)$  para  $\Theta(m)$ , mantendo o tempo real do empalhamento limitado por  $\Theta(n)$ .

## Correção do cálculo da função prefixo

Começamos com um lema essencial mostrando que, pela iteração da função prefixo  $\pi$ , podemos enumerar todos os prefixos  $P_k$  que são sufixos próprios de um dado prefixo  $P_q$ . Seja

$$\pi^*[q] = \pi[q] =, \pi^{(2)}[q] =, \pi^{(3)}[q], \dots, \pi^{(t)}[q] \} ,$$

onde  $\pi^{(i)}[q]$  é definida em termos de iteração funcional, de forma que  $\pi^{(0)}[q] = q$  e  $\pi[\pi^{(i+1)}[a]]$  para  $i \geq 1$ , e onde se comprehende que a seqüência em  $\pi^*[q]$  se interrompe quando  $\pi^{(T)}[q] = 0$  é alcançada. O lema a seguir caracteriza  $\pi^*[q]$ , conforme ilustra a Figura 32.11.

### Lema 32.5 (Lema de iteração da função prefixo)

Seja  $P$  um padrão de comprimento  $m$  com função prefixo  $\pi$ . Então, para  $q = 1, 2, \dots, m$ , temos  $\pi^*[q] = \{k : k < q \text{ e } P_k \sqsupseteq P_q\}$ .

**Prova** Primeiro, provaremos que

$$i \in \pi^*[q] \text{ implica } P_i \sqsupseteq P_q. \quad (32.6)$$

Se  $i \in \pi^*[q]$ , então  $i = \pi^{(u)}[q]$  para algum  $u > 0$ . Provamos a equação (32.6) por indução sobre  $u$ . Para  $u = 1$ , temos  $i = \pi[q]$ , e a afirmação se segue, pois  $i < q$  e  $P_{\pi[q]} \sqsupseteq P_q$ . O uso das relações  $\pi[i] < i$  e  $P_{\pi[i]} \sqsupseteq P_i$  e da transitividade de  $<$  e  $\sqsupseteq$  estabelecem a afirmação para todo  $i$  em  $\pi^*[q]$ . Então,  $\pi^*[q] \subseteq \{k : k < q \text{ e } P_k \sqsupseteq P_q\}$ .

Provaremos que  $\{k : k < q \text{ e } P_k \sqsupseteq P_q\} \subseteq \pi^*[q]$  por contradição. Suponha a hipótese contrária de que existe um inteiro no conjunto  $\{k : k < q \text{ e } P_k \sqsupseteq P_q\} - \pi^*[q]$ , e seja  $j$  o maior desses valores. Como  $\pi[q]$  é o maior valor em  $\{k : k < q \text{ e } P_k \sqsupseteq P_q\}$  e  $\pi[q] \in \pi^*[q]$ , devemos ter  $j < \pi[q]$ , e assim azemos  $j'$  denotar o menor inteiro em  $\pi^*[q]$  que é maior que  $j$ . (Podemos escolher  $j' = \pi[q]$ , se não existe nenhum outro número em  $\pi^*[q]$  que seja maior que  $j$ .) Temos  $P_j \sqsupseteq P_q$ , porque  $j \in \{k : k < q \text{ e } P_k \sqsupseteq P_q\}$ , e temos  $P_{j'} \sqsupseteq P_q$  porque  $j' \in \pi^*[q]$ . Desse modo,  $P_j \sqsupseteq P_{j'}$  pelo Lema 32.1 e  $j'$  é o maior valor menor que  $j'$  com essa propriedade. Portanto, devemos ter  $\pi[j'] = j$  e, como  $j' \in \pi^*[q]$ , devemos ter também  $j \in \pi^*[q]$ . Essa contradição prova o lema. ■

O algoritmo COMPUTE-PREFIX-FUNCTION calcula  $\pi[q]$  em ordem para que  $q = 1, 2, \dots, m$ . O cálculo de  $\pi[1] = 0$  na linha 2 de COMPUTE-PREFIX-FUNCTION certamente é correto, pois  $\pi[q] < q$  para todo  $q$ . O lema a seguir e seu corolário serão usados para provar que COMPUTE-PREFIX-FUNCTION calcula  $\pi[q]$  corretamente para  $q > 1$ .

### Lema 32.6

Seja  $P$  um padrão de comprimento  $m$ , e seja  $\pi$  a função prefixo para  $P$ . Para  $q = 1, 2, \dots, m$ , se  $\pi[q] > 0$ , então  $\pi[q] - 1 \in \pi^*[q - 1]$ .

**Prova** Se  $r = \pi[q] > 0$ , então  $r < q$  e  $P_r \sqsupseteq P_q$ ; portanto,  $r - 1 < q - 1$  e  $P_{r-1} \sqsupseteq P_{q-1}$  (eliminando-se o último caractere de  $P_r$  e  $P_q$ ). Então, pelo Lema 32.5,  $\pi[q] - 1 = r - 1 \in \pi^*[q - 1]$ . ■

Para  $q = 2, 3, \dots, m$ , definimos o subconjunto  $E_q - 1 \subseteq \pi^*[q - 1]$  por

$$\begin{aligned} E_q - 1 &= \{k \in \pi^*[q - 1] : P[k + 1] = P[q]\} \\ &= \{k : k < q - 1 \text{ e } P_k \sqsupseteq P_{q-1} \text{ e } P[k + 1] = P[q]\} \\ &= \{k : k < q - 1 \text{ e } P_{k+1} \sqsupseteq P_q\}. \end{aligned}$$

O conjunto  $E_{q-1}$  consiste nos valores  $k < q - 1$  para os quais  $P_k \sqsupset P_{q-1}$  e para os quais  $P_{k+1} \sqsupset P_q$ , porque  $P[k+1] = P[q]$ . Desse modo,  $E_{q-1}$  consiste nos valores  $k \in \pi^*[q-1]$  tais que podemos estender  $P_k$  a  $P_{k+1}$  e obter um sufixo próprio de  $P_q$ .

### Corolário 32.7

Seja  $P$  um padrão de comprimento  $m$ , e seja  $\pi$  a função de prefixo para  $P$ . Para  $q = 2, 3, \dots, m$ ,

$$\pi[q] = \begin{cases} 0 & \text{se } E_{q-1} = \emptyset, \\ 1 + \max\{k \in E_{q-1}\} & \text{se } E_{q-1} \neq \emptyset. \end{cases}$$

**Prova** Se  $E_{q-1}$  é vazio, não existe nenhum  $k \in \pi^*[q-1]$  (incluindo  $k = 0$ ) para o qual podemos estender  $P_k$  a  $P_{k+1}$  e obter um sufixo próprio de  $P_q$ . Então,  $\pi[q] = 0$ .

Se  $E_{q-1}$  é não vazio, então para cada  $k \in E_{q-1}$  temos  $k + 1 < q$  e  $P_{k+1} \sqsupset P_q$ . Então, da definição de  $\pi[q]$ , temos

$$\pi[q] \geq 1 + \max\{k \in E_{q-1}\} \quad (32.7)$$

Observe que  $\pi[q] > 0$ . Seja  $r = \pi[q] - 1$ , de modo que  $r + 1 = \pi[q]$ . Tendo em vista que  $r + 1 > 0$ , temos  $P[r+1] = P[q]$ . Além disso, pelo Lema 32.6, temos  $r \in \pi^*[q-1]$ . Então,  $r \in E_{q-1}$ , e assim  $r \leq \max\{k \in E_{q-1}\}$  ou, de modo equivalente,

$$\pi[q] \leq 1 + \max\{k \in E_{q-1}\} \quad (32.8)$$

A combinação das equações (32.7) e (32.8) completa a prova. ■

Agora, terminamos a prova de que COMPUTE-PREFIX-FUNCTION calcula  $\pi$  corretamente. No procedimento COMPUTE-PREFIX-FUNCTION, no início de cada iteração do loop **for** das linhas 4 a 9, temos que  $k = \pi[q-1]$ . Essa condição é reforçada pelas linhas 2 e 3 quando se entra no loop pela primeira vez e permanece verdadeira em cada iteração sucessiva, por causa da linha 9. As linhas 5 a 8 ajustam  $k$  de modo que ele agora se torne o valor correto de  $\pi[q]$ . O loop nas linhas 5 e 6 pesquisa todos os valores  $k \in \pi^*[q-1]$  até encontrar um para o qual  $P[k+1] = P[q]$ ; nesse ponto, temos que  $k$  é o maior valor no conjunto  $E_{q-1}$ , de forma que, pelo Corolário 32.7, podemos definir  $\pi[q]$  como  $k + 1$ . Se nenhum  $k$  for encontrado,  $k = 0$  na linha 7. Se  $P[k+1] = P[q]$ , então devemos definir tanto  $k$  quanto  $\pi[q]$  como 1; caso contrário, devemos deixar  $k$  como está e definir  $\pi[q]$  como 0. As linhas 7 a 9 definem  $k$  e  $\pi[q]$  corretamente em qualquer caso. Isso completa nossa prova da correção de COMPUTE-PREFIX-FUNCTION.

## Correção do algoritmo KMP

O procedimento KMP-MATCHER pode ser visto como uma reimplementação do procedimento FINITE-AUTOMATON-MATCHER. De modo específico, provaremos que o código nas linhas 6 a 9 de KMP-MATCHER é equivalente à linha 4 de FINITE-AUTOMATON-MATCHER, que define  $q$  como  $\delta(q, T[i])$ . Porém, em vez de usar um valor armazenado de  $\delta(q, T[i])$ , esse valor é recalculado conforme necessário a partir de  $\pi$ . Depois que tivermos demonstrado que KMP-MATCHER simula o comportamento de FINITE-AUTOMATON-MATCHER, a correção de KMP-MATCHER seguirá da correção de FINITE-AUTOMATON-MATCHER (embora seja visto em breve porque a linha 12 em KMP-MATCHER é necessária).

A correção de KMP-MATCHER decorre da afirmação de que devemos ter  $\delta(q, T[i]) = 0$  ou então  $\delta(q, T[i]) - 1 \in \pi^*[q]$ . Para verificar essa afirmação, seja  $\delta(q, T[i])$ . Então,  $P_k \sqsupset P_q T[i]$  pelas definições de  $\delta$  e  $\sigma$ . Assim,  $k = 0$ , ou então  $k \geq 1$  e  $P_{k-1} \sqsupset P_q$ , eliminando-se o último caractere de  $P_k$  e  $P_q T[i]$  (em cujo caso,  $k - 1 \in \pi^*[q]$ ). Portanto,  $k = 0$  ou  $k - 1 \in \pi^*[q]$ , o que prova a afirmação. | 735

A afirmação é usada como a seguir. Seja  $q'$  o valor de  $q$  quando a linha 6 é executada. Usamos a equivalência  $\pi^*[q] = \{k : k < q - 1 \text{ e } P_k = P_q\}$  a partir do Lema 32.5 para justificar a iteração  $q \leftarrow \pi[q]$  que enumera os elementos de  $\{k : P_k = P_q\}$ . As linhas 6 a 9 determinam  $\delta(q', T[i])$  pelo exame dos elementos de  $\pi^*[q']$  em ordem decrescente. O código usa a afirmação para começar com  $q = \phi(T_{i-1}) = \sigma(T_{i-1})$  e executar a iteração  $q \leftarrow \pi[q]$  até ser encontrado um  $q$  tal que  $q = 0$  ou  $P[q + 1] = T[i]$ . No primeiro caso,  $\delta(q', T[i])$ ; no último  $q$  é o elemento máximo em  $E_{q'}$ , de forma que  $\delta(q', T[i]) = q + 1$  pelo Corolário 32.7.

A linha 12 é necessária em KMP-MATCHER para evitar uma possível referência a  $P[m + 1]$  na linha 6 depois de uma ocorrência de  $P$  ter sido encontrada. (O argumento de que  $q = \sigma(T_{i-1})$  na execução seguinte da linha 6 permanece válida pela sugestão dada no Exercício 32.4-6:  $\delta(m, a) = \delta(\pi[m], a)$  ou, de modo equivalente,  $\sigma(P_a) = \sigma(P_{\pi[m]} | a)$  para qualquer  $a \in \Sigma$ .) O argumento restante para a correção do algoritmo de Knuth-Morris-Pratt vem da correção de FINITE-AUTOMATON-MATCHER, pois agora vemos que KMP-MATCHER simula o comportamento de FINITE-AUTOMATON-MATCHER.

## Exercícios

### 32.4-1

Calcule a função prefixo  $\pi$  para o padrão ababbabbabbabbabb quando o alfabeto é  $\Sigma = \{a, b\}$ .

### 32.4-2

Forneça um limite superior sobre o tamanho de  $\pi^*[q]$  como uma função de  $q$ . Forneça um exemplo para mostrar que seu limite é restrito.

### 32.4-3

Explique como determinar as ocorrências do padrão  $P$  no texto  $T$ , examinando a função  $\pi$  em busca da cadeia  $PT$  (a cadeia de comprimento  $m + n$  que é a concatenação de  $P$  e  $T$ ).

### 32.4-4

Mostre como otimizar KMP-MATCHER substituindo a ocorrência de  $\pi$  na linha 7 (mas não na linha 12) por  $\pi'$ , onde  $\pi'$  é definido recursivamente para  $q = 1, 2, \dots, m$  pela equação

$$\pi'[q] = \begin{cases} 0 & \text{se } \pi[q] = 0, \\ \pi'[\pi[q]] & \text{se } \pi[q] \neq 0 \text{ e } P[\pi[q]+1] = P[q+1], \\ \pi[q] & \text{se } \pi[q] \neq 0 \text{ e } P[\pi[q]+1] \neq P[q+1]. \end{cases}$$

Explique por que o algoritmo modificado é correto, e explique em que sentido essa modificação constitui uma otimização.

### 32.4-5

Forneça um algoritmo de tempo linear para descobrir se um texto  $T$  é uma rotação cílica de outra cadeia  $T'$ . Por exemplo, as cadeias arc e car são rotações cílicas uma da outra.

### 32.4-6 \*

Forneça um algoritmo eficiente para calcular a função de transição  $\delta$  para o autômato de emparelhamento de cadeias equivalente a um dado padrão  $P$ . Seu algoritmo deve ser executado no tempo  $O(m |\Sigma|)$ . (Sugestão: Prove que  $\delta(q, a) = \delta(\pi[q], a)$  se  $q = m$  ou  $P[q + 1] \neq a$ .)

## Problemas

### 32-1 Emparelhamento de cadeias baseada em fatores de repetição

Seja  $y^i$  a concatenação da cadeia  $y$  com ela própria  $i$  vezes. Por exemplo,  $(ab)^3 = ababab$ . Dizemos que uma cadeia  $x' \in \Sigma^*$  tem **fator de repetição**  $r$  se  $x = y^r$  para alguma cadeia  $y \in \Sigma^*$  e algum  $r > 0$ . Seja  $\rho(x)$  o maior  $r$  tal que  $x$  tenha fator de repetição  $r$ .

- a. Forneça um algoritmo eficiente que tome como entrada um padrão  $P[1..m]$  e que calcule o valor  $\rho(P_i)$  para  $i = 1, 2, \dots, m$ . Qual é o tempo de execução de seu algoritmo?
- b. Para qualquer padrão  $P[1..m]$ , seja  $\rho^*(P)$  definido como  $\max_{1 \leq i \leq m} \rho(P_i)$ . Prove que, se o padrão  $P$  for escolhido aleatoriamente a partir do conjunto de todos as cadeias binárias de comprimento  $m$ , então o valor esperado de  $\rho^*(P)$  será  $O(1)$ .
- c. Demonstre que o algoritmo de emparelhamento de cadeias a seguir encontra corretamente todas as ocorrências do padrão  $P$  em um texto  $T[1..n]$  no tempo  $O(\rho^*(P)n + m)$ .

REPETITION-MATCHER( $P, T$ )

```

1  $m \leftarrow \text{comprimento}[P]$ 
2  $n \leftarrow \text{comprimento}[T]$ 
3  $k \leftarrow 1 + \rho^*(P)$ 
4  $q \leftarrow 0$ 
5  $s \leftarrow 0$ 
6 while  $s \leftarrow n - m$ 
7   do if  $T[s + q + 1] = P[q + 1]$ 
8     then  $q \leftarrow q + 1$ 
9     if  $q = m$ 
10       then imprimir "Padrão ocorre com deslocamento"  $s$ 
11   if  $q = m$  ou  $T[s + q + 1] \neq P[q + 1]$ 
12     then  $s \leftarrow s + \max(1, \lceil q/k \rceil)$ 
13    $q \leftarrow 0$ 
```

Esse algoritmo foi desenvolvido por Galil e Seiferas. Ampliando bastante essas idéias, eles obtêm um algoritmo de emparelhamento de cadeias de tempo linear que utiliza apenas o espaço de armazenamento  $O(1)$  além do que é necessário para  $P$  e  $T$ .

## Notas do capítulo

A relação entre a emparelhamento de cadeias e a teoria de autômatos finitos é discutida por Aho, Hopcroft e Ullman [5]. O algoritmo de Knuth-Morris-Pratt [187] foi criado independentemente por Knuth e Pratt e por Morris; eles publicaram seu trabalho em conjunto. O algoritmo de Rabin-Karp foi proposto por Rabin e Karp [175]. Galil e Seiferas [107] apresentam um interessante algoritmo determinístico para emparelhamento de cadeias em tempo linear que utiliza apenas o espaço  $O(1)$  além do que é exigido para armazenar o padrão e o texto.

# *Geometria computacional*

A geometria computacional é o ramo da ciência de computação que estuda algoritmos para resolver problemas geométricos. Na engenharia e na matemática modernas, a geometria computacional tem aplicações em, entre outros campos, gráficos de computador, robótica, projeto VLSI, projeto com o auxílio do computador e estatística. A entrada para um problema de geometria computacional é em geral uma descrição de um conjunto de objetos geométricos, como um conjunto de pontos, um conjunto de segmentos de linha ou os vértices de um polígono no sentido anti-horário. A saída é com freqüência uma resposta a uma consulta sobre os objetos como, por exemplo, se quaisquer das linhas se cruzam, ou talvez um novo objeto geométrico, como a envoltória convexa (a menor envoltória do polígono convexo) do conjunto de pontos).

Neste capítulo, observamos alguns algoritmos de geometria computacional em duas dimensões, isto é, no plano. Cada objeto de entrada é representado como um conjunto de pontos  $\{p_1, p_2, p_3, \dots\}$ , onde cada  $p_i = (x_i, y_i)$  e  $x_i, y_i \in \mathbb{R}$ . Por exemplo, um polígono  $P$  de  $n$  vértices é representado por uma seqüência  $\langle p_0, p_1, p_2, \dots, p_{n-1} \rangle$  de seus vértices na ordem de seu aparecimento no perímetro de  $P$ . A geometria computacional também pode ser executada em três dimensões, e até em espaços de dimensões mais altas, mas tais problemas e suas soluções podem ser muito difíceis de visualizar. Porém, mesmo em duas dimensões, podemos ter uma boa amostra das técnicas de geometria computacional.

A Seção 33.1 mostra como responder a perguntas básicas sobre segmentos de linha de modo eficiente e com precisão: se um segmento está à direita ou à esquerda de outro que compartilha uma extremidade, que caminho devemos seguir para atravessar dois segmentos de linha adjacentes e se dois segmentos de linha se cruzam. A Seção 33.2 apresenta uma técnica chamada “varredura” que usamos para desenvolver um algoritmo de tempo  $O(n \lg n)$  para determinar se existe qualquer interseção entre um conjunto de  $n$  segmentos de linha. A Seção 33.3 oferece dois algoritmos de “varredura rotacional” que calculam a envoltória convexa (o menor polígono convexo envoltório) de um conjunto de  $n$  pontos: a varredura de Graham, que é executada no tempo  $O(n \lg n)$ , e a marcha de Jarvis, que demora o tempo  $O(nh)$ , onde  $h$  é o número de vértices da envoltória convexa. Finalmente, a Seção 33.4 fornece um algoritmo de dividir e conquistar de tempo  $O(n \lg n)$  para encontrar o par de pontos mais próximo em um conjunto de  $n$  pontos no plano.

### **33.1 Propriedades de segmentos de linha**

Vários algoritmos da geometria computacional neste capítulo exigirão respostas a perguntas sobre as propriedades de segmentos de linha. Uma **combinação convexa** de dois pontos distin-

tos  $p_1 = (x_1, y_1)$  e  $p_2 = (x_2, y_2)$  é qualquer ponto  $p_3 = (x_3, y_3)$  tal que, para algum  $\alpha$  no intervalo  $0 \leq \alpha \leq 1$ , temos  $x_3 = \alpha x_1 + (1 - \alpha)x_2$  e  $y_3 = \alpha y_1 + (1 - \alpha)y_2$ . Também escrevemos que  $p_3 = \alpha p_1 + (1 - \alpha)p_2$ . Intuitivamente,  $p_3$  é qualquer ponto que esteja na linha que passa por  $p_1$  e  $p_2$  e que está sobre ou entre  $p_1$  e  $p_2$  na linha. Dados dois pontos distintos  $p_1$  e  $p_2$ , o **segmento de linha**  $p_1 p_2$  é o conjunto de combinações convexas de  $p_1$  e  $p_2$ . Chamamos  $p_1$  e  $p_2$  de **extremidades** ou **Pontos extremos** do segmento  $p_1 p_2$ . Às vezes a ordenação de  $p_1$  e  $p_2$  é importante, e falamos do **segmento orientado**  $\overrightarrow{p_1 p_2}$ . Se  $p_1$  é a **origem**  $(0, 0)$ , então podemos tratar o segmento orientado  $\overrightarrow{p_1 p_2}$  como o **vetor**  $p_2$ .

Nesta seção, exploraremos as seguintes questões:

1. Dados dois segmentos orientados  $\overrightarrow{p_0 p_1}$  e  $\overrightarrow{p_0 p_2}$ ,  $\overrightarrow{p_0 p_1}$  está à direita de  $\overrightarrow{p_0 p_2}$  em relação a sua extremidade comum  $p_0$ ?
2. Dados dois segmentos de linha  $\overrightarrow{p_0 p_1}$  e  $\overrightarrow{p_0 p_2}$ , se percorrermos  $\overrightarrow{p_0 p_1}$  e depois  $\overrightarrow{p_0 p_2}$ , vamos nos dirigir para a esquerda no ponto  $p_1$ ?
3. Os segmentos de linha  $\overrightarrow{p_1 p_2}$  e  $\overrightarrow{p_3 p_4}$  se cruzam?

Não há nenhuma restrição sobre os pontos dados.

Podemos responder a cada pergunta no tempo  $O(1)$ , o que não deve ser nenhuma surpresa, pois o tamanho da entrada de cada pergunta é  $O(1)$ . Além disso, nossos métodos usarão apenas adições, subtrações, multiplicações e comparações. Não precisamos nem da divisão nem de funções trigonométricas, pois ambas podem ser dispendiosas em termos computacionais e propensas a problemas com erros de arredondamento. Por exemplo, o método “direto” de determinar se dois segmentos se cruzam – calcular a equação de linha da forma  $y = mx + b$  para cada segmento ( $m$  é a inclinação e  $b$  é a interseção com o eixo  $y$ ), encontrar o ponto de interseção das linhas e verificar se esse ponto está em ambos os segmentos – utiliza a divisão para encontrar o ponto de interseção. Quando os segmentos são quase paralelos, esse método é muito sensível à precisão da operação de divisão em computadores reais. O método desta seção, que evita divisão, é muito mais preciso.

## Produtos cruzados

O cálculo de produtos cruzados está no núcleo de nossos métodos de segmentos de linha. Considere os vetores  $p_1$  e  $p_2$  mostrados na Figura 33.1(a). O **produto cruzado**  $p_1 \times p_2$  pode ser interpretado como a área assinalada do paralelogramo formado pelos pontos  $(0, 0)$ ,  $p_1$ ,  $p_2$  e  $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$ . Uma definição equivalente, embora mais útil, apresenta o produto cruzado como o determinante de uma matriz:<sup>1</sup>

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1. \end{aligned}$$

Se  $p_1 \times p_2$  é positivo, então  $p_1$  está à direita de  $p_2$  com relação à origem  $(0, 0)$ ; se esse produto cruzado é negativo, então  $p_1$  está à esquerda de  $p_2$ . A Figura 33.1(b) mostra as regiões à direita e à esquerda em relação ao vetor  $p$ . Uma condição limite surge se o produto cruzado é zero; nesse caso, os vetores são **colineares**, apontando para o mesmo sentido ou para sentidos opostos.

<sup>1</sup>Na realidade, o produto cruzado é um conceito tridimensional. Ele é um vetor perpendicular a  $p_1$  e  $p_2$ , de acordo com a “regra da mão direita” e cuja magnitude é  $|x_1 y_2 - x_2 y_1|$ . Contudo, neste capítulo, provaremos que é conveniente tratar o produto cruzado apenas como o valor  $x_1 y_2 - x_2 y_1$ .

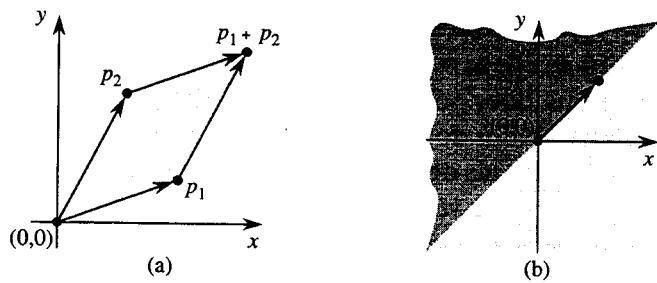


FIGURA 33.1 (a) O produto cruzado de vetores  $p_1$  e  $p_2$  é a área assinalada do paralelogramo. (b) A região ligeiramente sombreada contém vetores que estão à direita de  $p$ . A região com sombreamento mais escuro contém vetores que estão à esquerda de  $p$

Para determinar se um segmento orientado  $\overrightarrow{p_0 p_1}$  está à direita de um segmento orientado  $\overrightarrow{p_0 p_2}$  com relação à sua extremidade comum  $p_0$ , simplesmente fazemos a conversão para usar  $p_0$  como origem. Ou seja, fazemos  $p_1 - p_0$  denotar o vetor  $p'_1 = (x'_1, y'_1)$ , onde  $x'_1 = x_1 - x_0$  e  $y'_1 = y_1 - y_0$ , e definimos  $p_2 - p_0$  de maneira semelhante. Em seguida, calculamos o produto cruzado

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

Se esse produto cruzado é positivo, então  $\overrightarrow{p_0 p_1}$  está à direita de  $\overrightarrow{p_0 p_2}$ ; se é negativo, ele está à esquerda.

### Como determinar se segmentos consecutivos se dirigem para a esquerda ou direita

Nossa próxima pergunta é se dois segmentos de linha consecutivos  $\overrightarrow{p_0 p_1}$  e  $\overrightarrow{p_1 p_2}$  se dirigem para a esquerda ou direita no ponto  $p_1$ . De forma equivalente, queremos um método para determinar o caminho que um dado ângulo  $\angle p_0 p_1 p_2$  segue. Produtos cruzados nos permitem responder a essa pergunta sem calcular o ângulo. Como mostra a Figura 33.2, basta verificar se o segmento orientado  $\overrightarrow{p_0 p_2}$  está à direita ou à esquerda em relação ao segmento orientado  $\overrightarrow{p_0 p_1}$ . Para isso, calculamos o produto cruzado  $(p_2 - p_0) \times (p_1 - p_0)$ . Se o sinal desse produto cruzado é negativo, então  $\overrightarrow{p_0 p_2}$  está à esquerda em relação a  $\overrightarrow{p_0 p_1}$ , e portanto fazemos uma volta à esquerda em  $p_1$ . Um produto cruzado positivo indica uma orientação horária e uma volta à direita. Um produto cruzado igual a 0 significa que os pontos  $p_0$ ,  $p_1$  e  $p_2$  são colineares.

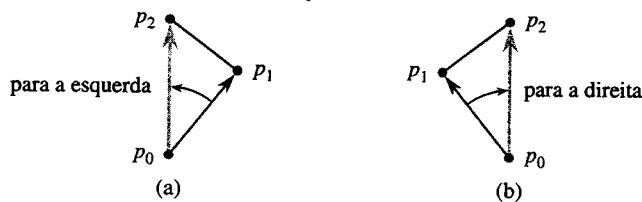


FIGURA 33.2 Usando o produto cruzado para determinar como segmentos de linha consecutivos  $\overrightarrow{p_0 p_1}$  e  $\overrightarrow{p_1 p_2}$  se voltam no ponto  $p_1$ . Verificamos se o segmento orientado  $\overrightarrow{p_0 p_2}$  está à direita ou à esquerda em relação ao segmento orientado  $\overrightarrow{p_0 p_1}$ . (a) Se estiverem à esquerda, os pontos se orientam à esquerda. (b) Se estiverem à direita, eles se voltam para a direita

### Como determinar se dois segmentos de linha se cruzam

Para determinar se dois segmentos de linha se cruzam, verificamos se cada segmento intercepta a linha que contém o outro. Um segmento  $\overrightarrow{p_1 p_2}$  intercepta uma linha se o ponto  $p_1$  reside em

um lado da linha e o ponto  $p_2$  reside no outro lado. Um caso limite surge se  $p_1$  ou  $p_2$  reside diretamente sobre a linha. Dois segmentos de linha se cruzam se e somente se uma (ou ambas) das condições a seguir é válida:

1. Cada segmento intercepta a linha que contém o outro.
2. Uma extremidade de um segmento reside no outro segmento. (Essa condição vem do caso limite.)

Os procedimentos a seguir implementam essa idéia. SEGMENTS-INTERSECT retorna TRUE se os segmentos  $p_1p_2$  e  $p_3p_4$  se cruzam FALSE se eles não se cruzam. Ele chama as sub-rotinas DIRECTION, que calcula orientações relativas usando o método de produto cruzado anterior, e ON-SEGMENT, que determina se um ponto conhecido que se sabe ser colinear com um segmento reside nesse segmento.

**SEGMENTS-INTERSECT( $p_1, p_2, p_3, p_4$ )**

```

1  $d_1 \leftarrow \text{DIRECTION}(p_3, p_4, p_1)$ 
2  $d_2 \leftarrow \text{DIRECTION}(p_3, p_4, p_2)$ 
3  $d_3 \leftarrow \text{DIRECTION}(p_1, p_2, p_3)$ 
4  $d_4 \leftarrow \text{DIRECTION}(p_1, p_2, p_4)$ 
5 if ( $(d_1 > 0 \text{ e } d_2 < 0)$  ou ( $d_1 < 0 \text{ e } d_2 > 0$ )) e
   ( $(d_3 > 0 \text{ e } d_4 < 0)$  ou ( $d_3 < 0 \text{ e } d_4 > 0$ ))
6 then return TRUE
7 elseif  $d_1 = 0$  e ON-SEGMENT( $p_3, p_4, p_1$ )
8 then return TRUE
9 elseif  $d_2 = 0$  e ON-SEGMENT( $p_3, p_4, p_2$ )
10 then return TRUE
11 elseif  $d_3 = 0$  e ON-SEGMENT( $p_1, p_2, p_3$ )
12 then return TRUE
13 elseif  $d_4 = 0$  e ON-SEGMENT( $p_1, p_2, p_4$ )
14 then return TRUE
15 else return FALSE

```

**DIRECTION( $p_i, p_j, p_k$ )**

```
1 return  $(p_k - p_i) \times (p_j - p_i)$ 
```

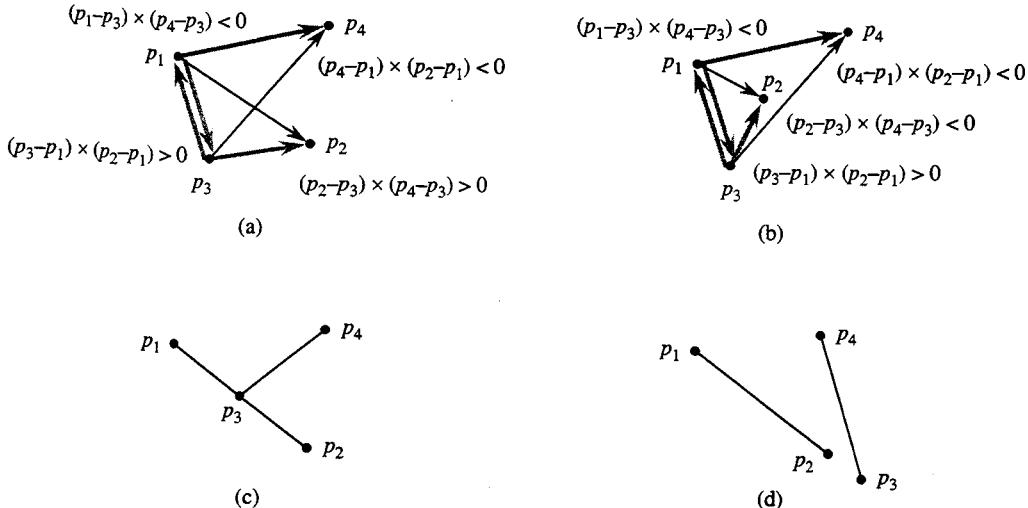
**ON-SEGMENT( $p_i, p_j, p_k$ )**

```

1 if  $\min(x_i, x_j) \leq x_k \leq \max(x_i, x_j)$  e  $\min(y_i, y_j) \leq y_k \leq \max(y_i, y_j)$ 
2 then return TRUE
3 else return FALSE

```

SEGMENTS-INTERSECT funciona da maneira ilustrada a seguir. As linhas 1 a 4 calculam a orientação relativa  $d_i$  de cada extremidade  $p_i$  com relação ao outro segmento. Se todas as orientações relativas forem não nulas, então podemos determinar facilmente se os segmentos  $p_1p_2$  e  $p_3p_4$  se cruzam, como a seguir. O segmento  $p_1p_2$  intercepta a linha que contém o segmento  $p_3p_4$  se os segmentos orientados  $p_3p_1$  e  $p_3p_2$  têm orientações opostas em relação a  $p_3p_4$ . Nesse caso, os sinais de  $d_1$  e  $d_2$  diferem. De forma semelhante, o segmento  $p_3p_4$  intercepta a linha que contém  $p_1p_2$  se os sinais de  $d_3$  e  $d_4$  diferem. Se o teste da linha 5 é verdadeiro, então os segmentos interceptam um ao outro, e SEGMENTS-INTERSECT retorna TRUE. A Figura 33.3(a) mostra esse caso. Do contrário, os segmentos não interceptam as linhas um do outro, embora um caso limite possa se aplicar. Se todas as orientações relativas forem não nulas, nenhum caso limite se aplica. Todos os testes de comparação com 0 nas linhas 7 a 13 falham então, e SEGMENTS-INTERSECT retorna FALSE na linha 15. A Figura 33.3(b) mostra esse caso.



**FIGURA 33.3** Casos no procedimento SEGMENTS-INTERSECT. (a) O segmento  $\overline{p_1p_2}$  e o segmento  $\overline{p_3p_4}$  interceptam as linhas um do outro. Como  $p_3p_4$  intercepta a linha contendo  $p_1p_2$ , os sinais dos produtos cruzados  $(p_3-p_1) \times (p_2-p_1)$  e  $(p_4-p_1) \times (p_2-p_1)$  diferem entre si. Como  $p_1p_2$  intercepta a linha contendo  $p_3p_4$ , os sinais dos produtos cruzados  $(p_1-p_3) \times (p_4-p_3)$  e  $(p_2-p_3) \times (p_4-p_3)$  também diferem. (b) O segmento  $p_3p_4$  intercepta a linha contendo  $p_1p_2$ , mas  $p_1p_2$  não intercepta a linha contendo  $p_3p_4$ . Os sinais dos produtos cruzados  $(p_1-p_3) \times (p_4-p_3)$  e  $(p_2-p_3) \times (p_4-p_3)$  são iguais. (c) O ponto  $p_3$  é colinear em relação a  $p_1p_2$  e está entre  $p_1$  e  $p_2$ . (d) O ponto  $p_3$  é colinear com  $p_1p_2$ , mas não está entre  $p_1$  e  $p_2$ . Os segmentos não se cruzam

Um caso limite ocorre se qualquer orientação relativa  $d_k$  é 0. Aqui, sabemos que  $p_k$  é colinear com o outro segmento. Ele está diretamente sobre o outro segmento se e somente se está entre as extremidades do outro segmento. O procedimento ON-SEGMENT retorna se  $p_k$  está entre as extremidades do segmento  $p_i p_j$ , que será o outro segmento quando chamado nas linhas 7 a 13; o procedimento supõe que  $p_k$  é colinear com o segmento  $p_i p_j$ . As Figuras 33.3(c) e (d) mostram casos com pontos colineares. Na Figura 33.3(c),  $p_3$  está em  $p_1p_2$ , e então SEGMENTS-INTERSECT retorna TRUE na linha 12. Nenhuma extremidade está em outros segmentos na Figura 33.3(d) e assim SEGMENTS-INTERSECT retorna FALSE na linha 15.

## Outras aplicações de produtos cruzados

Seções posteriores deste capítulo introduzirão usos adicionais para produtos cruzados. Na Seção 33.3, precisaremos ordenar um conjunto de pontos de acordo com seus ângulos polares com relação a uma dada origem. Como o Exercício 33.1-3 lhe pede para mostrar, os produtos cruzados podem ser usados para executar as comparações no procedimento de ordenação. Na Seção 33.2, usaremos árvores vermelho-preto para manter a ordenação vertical de um conjunto de segmentos de linha. Em lugar de manter valores de chaves explícitos, substituiremos cada comparação de chave no código da árvore vermelho-preto por um cálculo de produto cruzado para determinar qual dos dois segmentos que cruzam uma dada linha vertical está sobre o outro.

## Exercícios

### 33.1-1

Prove que, se  $p_1 \times p_2$  é positivo, então o vetor  $p_1$  está à direita de vetor  $p_2$  com relação à origem  $(0, 0)$  e que, se seu produto cruzado é negativo, então  $p_1$  está à esquerda de  $p_2$ .

### 33.1-2

O professor Paulo sugere que somente a dimensão  $x$  precisa ser testada na linha 1 de ON-SEGMENT. Mostre por que o professor está errado.

### 33.1-3

O **ângulo polar** de um ponto  $p_1$  com relação a um ponto de origem  $p_0$  é o ângulo do vetor  $p_1 - p_0$  no sistema de coordenadas polares habitual. Por exemplo, o ângulo polar  $(3, 5)$  com relação a  $(2, 4)$  é o ângulo do vetor  $(1, 1)$ , que é 45 graus ou  $\pi/4$  radianos. O ângulo polar  $(3, 3)$  com relação a  $(2, 4)$  é o ângulo do vetor  $(1, -1)$ , que é 315 graus ou  $7\pi/4$  radianos. Escreva pseudocódigo para ordenar uma seqüência  $\langle p_1, p_2, \dots, p_n \rangle$  de  $n$  pontos de acordo com seus ângulos polares com relação a um determinado ponto de origem  $p_0$ . Seu procedimento deve demorar o tempo  $O(n \lg n)$  e usar produtos cruzados para comparar ângulos.

### 33.1-4

Mostre como determinar no tempo  $O(n^2 \lg n)$  se três pontos quaisquer em um conjunto de  $n$  pontos são colineares.

### 33.1-5

Um **polígono** é uma curva fechada formada por segmentos lineares no plano. Ou seja, é uma curva que se fecha sobre si mesma e que é formada por uma seqüência de segmentos de reta, chamados **lados** do polígono. Um ponto que une dois lados consecutivos é chamado **vértice** do polígono. Se o polígono é **simples**, como geralmente devemos supor, seus lados não se cruzam. O conjunto de pontos no plano delimitados por um polígono simples forma o **interior** do polígono, o conjunto de pontos no polígono propriamente dito forma seu **limite** e o conjunto de pontos que cercam o polígono forma seu **exterior**. Um polígono simples é **convexo** se, dados dois pontos quaisquer em seu limite ou em seu interior, todos os pontos no segmento de linha traçado entre eles estão contidos no limite ou no interior do polígono.

O professor Armando propõe o método a seguir para determinar se uma seqüência  $\langle p_1, p_2, \dots, p_{n-1} \rangle$  de  $n$  pontos forma os vértices consecutivos de um polígono convexo. Mostre a saída “sim” se o conjunto  $\{[p_i p_{i+1} p_{i+2}] : i = 0, 1, \dots, n-1\}$ , onde a adição de subscritos é executada em módulo  $n$ , não contém ao mesmo tempo voltas para a esquerda e para a direita; caso contrário, mostre a saída “não”. Mostre que, embora esse método seja executado em tempo linear, ele nem sempre produz a resposta correta. Modifique o método do professor de forma que ele sempre produza a resposta correta em tempo linear.

### 33.1-6

Dado um ponto  $p_0 = (x_0, y_0)$ , o **raio horizontal direito** de  $p_0$  é o conjunto de pontos  $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ e } y_i = y_0\}$ , ou seja, é o conjunto de pontos situados à direita de  $p_0$ , juntamente com o próprio  $p_0$ . Mostre como determinar se um dado raio horizontal direito de  $p_0$  cruza um segmento de linha  $p_1 p_2$  no tempo  $O(1)$ , reduzindo o problema ao de determinar se dois segmentos de linha se cruzam.

### 33.1-7

Um modo de determinar se um ponto  $p_0$  está no interior de um polígono  $P$  simples, mas não necessariamente convexo, é observar qualquer raio de  $p_0$  e verificar se o raio intercepta o limite de  $P$  um número ímpar de vezes, mas se o próprio  $p_0$  não está no limite de  $P$ . Mostre como calcular no tempo  $\Theta(n)$  se um ponto  $p_0$  está no interior de um polígono  $P$  de  $n$  vértices. (Sugestão: Use o Exercício 33.1-6. Certifique-se de que seu algoritmo está correto quando o raio cruza o limite do polígono em um vértice, e quando o raio se sobrepõe a um lado do polígono.)

### 33.1-8

Mostre como calcular a área de um polígono de  $n$  vértices simples, mas não necessariamente convexo, no tempo  $\Theta(n)$ . (Veja no Exercício 33.1-5 definições pertinentes a polígonos.)

## 33.2 Como determinar se dois segmentos quaisquer se cruzam

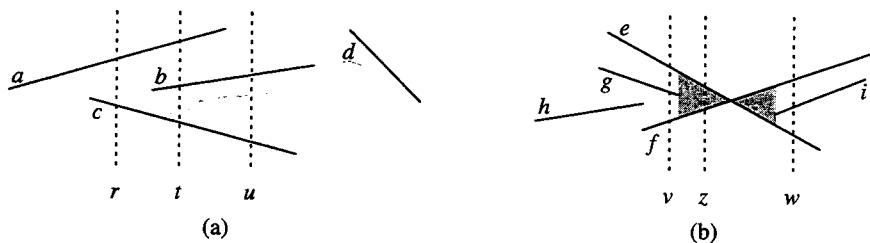
Esta seção apresenta um algoritmo para determinar se dois segmentos de linha quaisquer em um conjunto de segmentos se cruzam. O algoritmo utiliza uma técnica conhecida como “varre-

dura”, comum a muitos algoritmos de geometria computacional. Além disso, como mostram os exercícios no final desta seção, esse algoritmo, ou variações simples dele, pode ser usado para resolver outros problemas de geometria computacional.

O algoritmo é executado no tempo  $O(n \lg n)$ , onde  $n$  é o número de segmentos dados. Ele determina apenas se existe ou não alguma interseção; o algoritmo não imprime todas as interseções. (Pelo Exercício 33.2-1, ele demora o tempo  $\Omega(n^2)$  no pior caso para encontrar *todas* as interseções em um conjunto de  $n$  segmentos de linha.)

Na **varredura**, uma **linha de varredura** vertical imaginária passa pelo conjunto de objetos geométricos dado, em geral da esquerda para a direita. A dimensão espacial pela qual a linha de varredura se move, nesse caso a dimensão  $x$ , é tratada como uma dimensão de tempo. A varredura fornece um método para ordenar objetos geométricos, normalmente inserindo-os em uma estrutura de dados dinâmica, e para tirar proveito de relacionamentos entre eles. O algoritmo de interseção de segmentos de linha nesta seção considera todas as extremidades de segmentos de linha na ordem da esquerda para a direita e procura por uma interseção toda vez que encontra uma extremidade.

Com o objetivo de descrever e provar a correção de nosso algoritmo para determinar se dois quaisquer entre  $n$  segmentos de linha se cruzam, adotamos duas hipóteses simplificadoras. Primeiro, supomos que nenhum segmento de entrada é vertical. Em segundo lugar, supomos que não há três segmentos de entrada que se cruzem em um único ponto. Os Exercícios 33.2-8 e 33.2-9 lhe pedem para mostrar que o algoritmo é resistente o bastante para só precisar de uma leve modificação para funcionar mesmo quando essas hipóteses não são válidas. Na realidade, remover tais hipóteses simplificadoras e lidar com condições limites freqüentemente é a parte mais difícil da programação de algoritmos de geometria computacional e de provar sua correção.



**FIGURA 33.4** A ordenação entre segmentos de linha em diversas linhas de varredura verticais. (a) Temos  $a >, c$ ,  $a >, b$ ,  $b >, c$ ,  $a >, c$ , e  $b >, c$ . O segmento  $d$  não é comparável a nenhum outro segmento mostrado. (b) Quando os segmentos  $e$  e  $f$  se cruzam, sua ordem é invertida: temos  $e >, v$ , mas  $f >, w$ . Qualquer linha de varredura (por exemplo,  $z$ ) que passe pela região sombreada tem  $e$  e  $f$  consecutivos em sua ordem total

## Ordenação de segmentos

Tendo em vista que supomos que não existem segmentos verticais, qualquer segmento de entrada que cruze uma determinada linha de varredura vertical a intercepta em um único ponto. Podemos então ordenar os segmentos que cruzam uma linha de varredura vertical de acordo com as coordenadas  $y$  dos pontos de interseção.

Para ser mais preciso, considere dois segmentos  $s_1$  e  $s_2$ . Dizemos que esses segmentos são **comparáveis** em  $x$  se a linha de varredura vertical com coordenada  $x$  igual a  $x$  intercepta ambos os segmentos. Dizemos que  $s_1$  está **acima** de  $s_2$  em  $x$ , o que se indica por  $s_1 >, s_2$ , se  $s_1$  e  $s_2$  são comparáveis em  $x$  e a interseção de  $s_1$  com a linha de varredura em  $x$  é mais alta que a interseção de  $s_2$  com a mesma linha de varredura. Na Figura 33.4(a), por exemplo, temos os relacionamentos  $a >, c$ ,  $a >, b$ ,  $b >, c$ ,  $a >, c$ , e  $b >, c$ . O segmento  $d$  não é comparável a qualquer outro segmento.

Para qualquer  $x$  dado, a relação “ $>, x$ ” é uma ordem total (ver Seção B.2) sobre segmentos que cruzam a linha de varredura em  $x$ . Porém, a ordem pode ser diferente para valores distintos de  $x$ ,

à medida que os segmentos entram ou saem da ordenação. Um segmento entra na ordenação quando sua extremidade esquerda é encontrada pela varredura, e sai da ordenação quando sua extremidade direita é encontrada.

O que acontece quando a linha de varredura passa pela interseção de dois segmentos? Como mostra a Figura 33.4(b), suas posições na ordem total são invertidas. As linhas de varredura  $v$  e  $w$  estão à esquerda e à direita, respectivamente, do ponto de interseção dos segmentos  $e$  e  $f$ , e temos  $e >_v f >_w$ . Observe que, pelo fato de termos suposto que não há três segmentos que se cruzem no mesmo ponto, deve existir alguma linha de varredura vertical  $x$  para a qual os segmentos  $e$  e  $f$  que se cruzam são *consecutivos* na ordem total  $>_x$ . Qualquer linha de varredura que passe pela região sombreada da Figura 33.4(b), como  $z$ , tem  $e$  e  $f$  consecutivos em sua ordem total.

## Como mover a linha de varredura

Em geral, os algoritmos de varredura administram dois conjuntos de dados:

1. O *status da linha de varredura* fornece os relacionamentos entre os objetos interceptados pela linha de varredura.
2. A *programação de pontos de eventos* é uma sequência de coordenadas  $x$ , ordenadas da esquerda para a direita, que define as posições de parada da linha de varredura. Chamaremos cada posição de parada de um *ponto de evento*. As mudanças no status da linha de varredura ocorrem apenas nos pontos de eventos.

Para alguns algoritmos (por exemplo, o algoritmo solicitado no Exercício 33.2-7), a programação de pontos de eventos é determinada dinamicamente à medida que o algoritmo progride. Entretanto, o algoritmo em estudo determina os pontos de eventos estaticamente, com base apenas em propriedades simples dos dados de entrada. Em particular, cada extremidade de um segmento é um ponto de evento. Ordenamos as extremidades de segmentos aumentando a coordenada  $x$  e seguimos da esquerda para a direita. Inserimos um segmento no status da linha de varredura quando sua extremidade esquerda é encontrada, e o eliminamos do status da linha de varredura quando sua extremidade direita é encontrada. Sempre que dois segmentos se tornam consecutivos pela primeira vez na ordem total, verificamos se eles se cruzam.

O status da linha de varredura é uma ordem total  $T$ , para a qual exigimos as seguintes operações:

- $\text{INSERT}(T, s)$ : insere segmento  $s$  em  $T$ .
- $\text{DELETE}(T, s)$ : elimina o segmento  $s$  de  $T$ .
- $\text{ABOVE}(T, s)$ : retorna o segmento imediatamente acima do segmento  $s$  em  $T$ .
- $\text{BELOW}(T, s)$ : retorna o segmento imediatamente abaixo do segmento  $s$  em  $T$ .

Se existem  $n$  segmentos na entrada, podemos executar cada uma dessas operações no tempo  $O(\lg n)$ , usando árvores vermelho-preto. Lembre-se de que as operações em árvores vermelho-preto vistas no Capítulo 13 envolvem a comparação de chaves. Podemos substituir as comparações de chaves por comparações de produtos cruzados que determinam a ordenação relativa de dois segmentos (ver Exercício 33.2-2).

## Pseudocódigo de interseção de segmentos

O algoritmo a seguir toma como entrada um conjunto  $S$  de  $n$  segmentos de linha, retornando o valor booleano TRUE se qualquer par de segmentos em  $S$  se cruza, e FALSE em caso contrário. A ordem total  $T$  é implementada por uma árvore vermelho-preto.

ANY-SEGMENTS-INTERSECT( $S$ )

```

1  $T \leftarrow \emptyset$ 
2 ordenar as extremidades dos segmentos em  $S$  da esquerda para a direita,
   rompendo as ligações pela inserção das extremidades esquerdas
   antes das extremidades direitas e rompendo outras ligações
   inserindo primeiro os pontos com coordenadas  $y$  mais baixas
3 for cada ponto  $p$  na lista ordenada de extremidades
4   do if  $p$  é a extremidade esquerda de um segmento  $s$ 
5     then INSERT( $T, s$ )
6       if (ABOVE( $T, s$ ) existe e cruza  $s$ )
          ou (BELOW( $T, s$ ) existe e cruza  $s$ )
7         then return TRUE
8       if  $p$  é a extremidade direita de um segmento  $s$ 
9         then if tanto ABOVE( $T, s$ ) quanto BELOW( $T, s$ ) existem
            e ABOVE( $T, s$ ) cruza BELOW( $T, s$ )
10        then return TRUE
11        DELETE( $T, s$ )
12    return FALSE

```

A Figura 33.5 ilustra a execução do algoritmo. A linha 1 inicializa a ordem total como vazia. A linha 2 determina a programação de pontos de eventos, ordenando as  $2n$  extremidades de segmentos da esquerda para a direita, rompendo as ligações da maneira descrita. Observe que a linha 2 pode ser executada pela ordenação lexicográfica das extremidades sobre  $(x, e, y)$ , onde  $x$  e  $y$  são as coordenadas habituais e  $e = 0$  para uma extremidade esquerda e  $e = 1$  para uma extremidade direita.

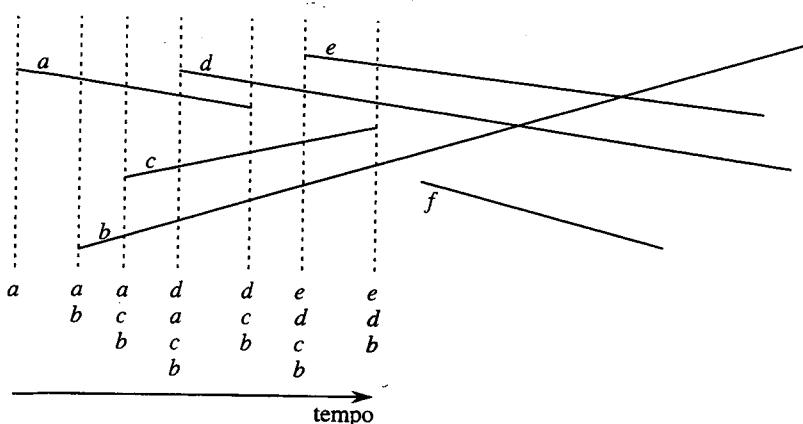


FIGURA 33.5 A execução de ANY-SEGMENTS-INTERSECT. Cada linha tracejada é a linha de varredura em um ponto de evento, e a ordenação de nomes de segmentos abaixo de cada linha de varredura é a ordem total  $T$  no final do loop for em que o ponto de evento correspondente é processado. A interseção de segmentos  $d$  e  $b$  é encontrada quando o segmento  $c$  é eliminado

Cada iteração do loop for das linhas 3 a 11 processa um ponto de evento  $p$ . Se  $p$  é a extremidade esquerda de um segmento  $s$ , a linha 5 acrescenta  $s$  à ordem total, e as linhas 6 e 7 retornam TRUE se  $s$  cruza qualquer um dos segmentos consecutivos a ele na ordem total definida pela linha de varredura que passa por  $p$ . (Uma condição limite ocorre se  $p$  reside em outro segmento  $s'$ . Nesse caso, exigimos apenas que  $s$  e  $s'$  sejam inseridos consecutivamente em  $T$ .) Se  $p$  é a extremidade direita de um segmento  $s$ , então  $s$  deve ser eliminado da ordem total. As linhas 9 e 10 retornam TRUE se existe uma interseção entre os segmentos que circundam  $s$  na ordem total definida pela linha de varredura que passa por  $p$ ; esses segmentos se tornarão consecutivos na or-

dem total quando  $s$  for eliminado. Se esses segmentos não se cruzam, a linha 11 elimina o segmento  $s$  da ordem total. Finalmente, se nenhuma interseção é encontrada no processamento de todos os  $2n$  pontos de eventos, a linha 12 retorna FALSE.

## Correção

Para mostrar que ANY-SEGMENTS-INTERSECT é correto, provaremos que a chamada ANY-SEGMENTS-INTERSECT( $S$ ) retorna TRUE se e somente se existe uma interseção entre os segmentos em  $S$ .

É fácil ver que ANY-SEGMENTS-INTERSECT retorna TRUE (nas linhas 7 e 10) apenas se encontra uma interseção entre dois dos segmentos de entrada. Conseqüentemente, se ele retorna TRUE, existe uma interseção.

Também precisamos mostrar a recíproca: se existe uma interseção, então ANY-SEGMENTS-INTERSECT retorna TRUE. Vamos supor que existe pelo menos uma interseção. Seja  $p$  o ponto de interseção mais à esquerda, rompendo as ligações pela escolha daquele que tem a menor ordenada  $y$ , e sejam  $a$  e  $b$  os segmentos que se cruzam em  $p$ . Como não ocorre nenhuma interseção à esquerda de  $p$ , a ordem dada por  $T$  é correta em todos os pontos à esquerda de  $p$ . Considerando-se que três segmentos não se cruzam no mesmo ponto, existe uma linha de varredura  $z$  na qual  $a$  e  $b$  se tornam consecutivos na ordem total.<sup>2</sup> Além disso,  $z$  está à esquerda de  $p$  ou passa por  $p$ . Existe uma extremidade de segmento  $q$  na linha de varredura  $z$  que é o ponto de evento no qual  $a$  e  $b$  se tornam consecutivos na ordem total. Se  $p$  está na linha de varredura  $z$ , então  $q = p$ . Se  $p$  não está na linha de varredura  $z$ , então  $q$  está à esquerda de  $p$ . Em qualquer caso, a ordem dada por  $T$  é correta imediatamente antes de  $q$  ser encontrado. (Aqui utilizamos a ordem lexicográfica em que o algoritmo processa os pontos de evento. Como  $p$  é o mais baixo dos pontos de interseção mais à esquerda, mesmo que  $p$  esteja na linha de varredura  $z$  e exista outro ponto de interseção  $p'$  em  $z$ , o ponto de evento  $q = p$  é processado antes da outra interseção  $p'$  poder interferir na ordem total  $T$ . Além disso, ainda que  $p$  seja a extremidade esquerda de um segmento, digamos  $a$ , e a extremidade direita do outro segmento, digamos  $b$ , como os eventos da extremidade esquerda ocorrem antes de eventos da extremidade direita, o segmento  $b$  está em  $T$  quando o segmento  $a$  é encontrado primeiro.) Ou o ponto de evento  $q$  é processado por ANY-SEGMENTS-INTERSECT ou ele não é processado.

Se  $q$  é processado por ANY-SEGMENTS-INTERSECT, existem apenas duas possibilidades para a ação tomada:

1. Ou  $a$  ou  $b$  é inserido em  $T$ , e o outro segmento está acima ou abaixo dele na ordem total. As linhas 4 a 7 detectam esse caso.
2. Os segmentos  $a$  e  $b$  já estão em  $T$ , e um segmento entre eles na ordem total é eliminado, fazendo  $a$  e  $b$  se tornarem consecutivos. As linhas 8 a 11 detectam esse caso.

Em qualquer caso, a interseção  $p$  é encontrada, e ANY-SEGMENTS-INTERSECT retorna TRUE.

Se o ponto de evento  $q$  não é processado por ANY-SEGMENTS-INTERSECT, o procedimento deve ter retornado antes de processar todos os pontos de eventos. Essa situação só poderia ter ocorrido se ANY-SEGMENTS-INTERSECT já tivesse encontrado uma interseção e retornado TRUE.

Desse modo, se existe uma interseção, ANY-SEGMENTS-INTERSECT retorna TRUE. Como já vimos, se ANY-SEGMENTS-INTERSECT retorna TRUE, existe uma interseção. Assim, ANY-SEGMENTS-INTERSECT sempre retorna uma resposta correta.

---

<sup>2</sup>Se permitirmos que três segmentos se cruzem no mesmo ponto, poderá haver um segmento  $c$  interveniente que cruze tanto  $a$  quanto  $b$  no ponto  $p$ . Ou seja, podemos ter  $a <_w c$  e  $c <_w b$  para todas as linhas de varredura  $w$  à esquerda de  $p$  para as quais  $a <_w b$ . O Exercício 33.2-8 lhe pede para mostrar que ANY-SEGMENTS-INTERSECT é correto ainda que três segmentos se cruzem no mesmo ponto.

## Tempo de execução

Se existem  $n$  segmentos no conjunto  $S$ , então ANY-SEGMENTS-INTERSECT é executado no tempo  $O(n \lg n)$ . A linha 1 demora o tempo  $O(1)$ . A linha 2 demora o tempo  $O(n \lg n)$ , usando a ordenação por intercalação ou heapsort. Tendo em vista que existem  $2n$  pontos de eventos, o loop **for** das linhas 3 a 11 itera no máximo  $2n$  vezes. Cada iteração demora o tempo  $O(\lg n)$ , pois cada operação da árvore vermelho-preto demora o tempo  $O(\lg n)$  e, usando o método da Seção 33.1, cada teste de interseção demora o tempo  $O(1)$ . Desse modo, o tempo total é  $O(n \lg n)$ .

## Exercícios

### 33.2-1

Mostre que podem existir  $\Theta(n^2)$  interseções em um conjunto de  $n$  segmentos de linha.

### 33.2-2

Dados dois segmentos  $a$  e  $b$  que são comparáveis em  $x$ , mostre como determinar no tempo  $O(1)$  qual dentre  $a >_x b$  ou  $b >_x a$  é válida. Suponha que nenhum segmento é vertical. (*Sugestão:* Se  $a$  e  $b$  não se cruzam, você pode simplesmente usar produtos cruzados. Se  $a$  e  $b$  se cruzam – o que evidentemente se pode determinar usando apenas produtos cruzados – ainda é possível usar somente operações de adição, subtração e multiplicação, evitando a divisão. É claro que, na aplicação da relação  $>_x$  utilizada aqui, se  $a$  e  $b$  se cruzam, podemos apenas parar e declarar que encontramos uma interseção.)

### 33.2-3

O professor Mariano sugere que modifiquemos ANY-SEGMENTS-INTERSECT de forma que, em vez de retornar ao encontrar uma interseção, ele imprima os segmentos que se cruzam e continue ativo na próxima iteração do loop **for**. O professor chama o procedimento resultante PRINT-INTERSECTING-SEGMENTS e afirma que ele imprime todas as interseções, da esquerda para a direita, à medida que elas ocorrem no conjunto de segmentos de linha. Mostre de duas maneiras que o professor está errado, fornecendo um conjunto de segmentos para o qual a primeira interseção encontrada por PRINT-INTERSECTING-SEGMENTS não é a interseção mais à esquerda, e um conjunto de segmentos para o qual PRINT-INTERSECTING-SEGMENTS deixa de encontrar todas as interseções.

### 33.2-4

Forneça um algoritmo de tempo  $O(n \lg n)$  para determinar se um polígono de  $n$  vértices é simples.

### 33.2-5

Forneça um algoritmo de tempo  $O(n \lg n)$  para determinar se dois polígonos simples com um total de  $n$  vértices se cruzam.

### 33.2-6

Um **disco** consiste em uma circunferência e seu interior, e é representado por seu centro e pelo raio. Dois discos se cruzam se têm qualquer ponto em comum. Forneça um algoritmo de tempo  $O(n \lg n)$  para determinar se dois discos quaisquer em um conjunto de  $n$  discos se cruzam.

### 33.2-7

Dado um conjunto de  $n$  segmentos de linha contendo um total de  $k$  interseções, mostre como dar saída a todas as  $k$  interseções no tempo  $O((n + k) \lg n)$ .

### 33.2-8

Mostre que ANY-SEGMENTS-INTERSECT funciona corretamente, mesmo que três ou mais segmentos se cruzem no mesmo ponto.

### 33.2-9

Mostre que ANY-SEGMENTS-INTERSECT funciona corretamente na presença de segmentos verticais se a extremidade inferior de um segmento vertical é processada como se fosse uma extremidade esquerda e a extremidade superior é processada como se fosse uma extremidade direita. Como sua resposta ao Exercício 33.2-2 se altera no caso de serem permitidos segmentos verticais?

## 33.3 Como encontrar a envoltória convexa

A **envoltória convexa** de um conjunto  $Q$  de pontos é o menor polígono convexo  $P$  para o qual cada ponto em  $Q$  está no limite de  $P$  ou em seu interior (veja no Exercício 33.1-5 uma definição precisa de um polígono convexo.) Denotamos a envoltória convexa de  $Q$  por  $\text{CH}(Q)$ . Intuitivamente, podemos imaginar cada ponto em  $Q$  como sendo um prego que se projeta para fora de uma tábua. A envoltória convexa é então a forma produzida por um elástico de borracha que envolvesse todas os pregos. A Figura 33.6 mostra um conjunto de pontos e sua envoltória convexa.

Nesta seção, apresentaremos dois algoritmos que calculam a envoltória convexa de um conjunto de  $n$  pontos. Ambos os algoritmos dão saída aos vértices da envoltória convexa da direita para a esquerda em ordem. O primeiro, conhecido como a varredura de Graham, é executado no tempo  $O(n \lg n)$ . O segundo, chamado marcha de Jarvis, é executado no tempo  $O(nb)$ , onde  $b$  é o número de vértices da envoltória convexa. Como podemos ver na Figura 33.6, todo vértice de  $\text{CH}(Q)$  é um ponto em  $Q$ . Ambos os algoritmos exploram essa propriedade, decidindo quais vértices em  $Q$  serão mantidos como vértices da envoltória convexa e quais vértices em  $Q$  serão descartados.

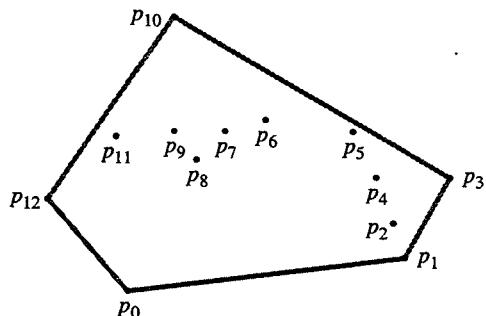


FIGURA 33.6 Um conjunto de pontos  $Q = \{p_0, p_1, \dots, p_{12}\}$  com sua envoltória convexa  $\text{CH}(Q)$  traçada em cor cinza

De fato, existem diversos métodos que calculam envoltórias convexas no tempo  $O(n \lg n)$ . Tanto a varredura de Graham quanto a marcha de Jarvis usam uma técnica chamada “varredura rotacional”, processando vértices na ordem dos ângulos polares que eles formam com um vértice de referência. Outros métodos incluem os seguintes.

- No **método incremental**, os pontos são ordenados da esquerda para a direita, formando uma seqüência  $\langle p_1, p_2, \dots, p_n \rangle$ . Na  $i$ -ésima fase, a envoltória convexa  $\text{CH}(\{p_1, p_2, \dots, p_{i-1}\})$  dos  $i-1$  pontos mais à esquerda é atualizada de acordo com o  $i$ -ésimo ponto a partir da esquerda, formando assim  $\text{CH}(\{p_1, p_2, \dots, p_i\})$ . Conforme o Exercício 33.3-6 lhe pede para mostrar, esse método pode ser implementado para demorar ao todo o tempo  $O(n \lg n)$ .
- No **método de dividir e conquistar**, no tempo  $\Theta(n)$  o conjunto de  $n$  pontos é dividido em dois subconjuntos, um contendo os  $\lceil n/2 \rceil$  pontos mais à esquerda e outro com os  $\lfloor n/2 \rfloor$  pontos mais à direita; as envoltórias convexas dos subconjuntos são calculadas recursivamente, e então um método inteligente é usado para combinar as envoltórias no

tempo  $O(n)$ . O tempo de execução é descrito pela recorrência familiar  $T(n) = 2T(n/2) + O(n)$ , e assim o método de dividir e conquistar é executado no tempo  $O(n \lg n)$ .

- O **método de podar e pesquisar** é semelhante ao algoritmo da mediana de tempo linear do pior caso da Seção 9.3. Ele encontra a parte superior (ou “cadeia superior”) da envoltória convexa, descartando repetidamente uma fração constante dos pontos restantes, até só restar a cadeia superior da envoltória convexa. Em seguida, ele faz o mesmo para a cadeia inferior. Esse método é assintoticamente o mais rápido: se a envoltória convexa contém  $b$  vértices, ele é executado apenas no tempo  $O(n \lg b)$ .

O cálculo da envoltória convexa de um conjunto de pontos é em si mesmo um problema interessante. Além disso, os algoritmos para alguns outros problemas de geometria computacional começam pelo cálculo de uma envoltória convexa. Por exemplo, considere o **problema do par mais afastado** bidimensional: temos um conjunto de  $n$  pontos no plano e desejamos encontrar os dois pontos cuja distância entre eles seja máxima. Conforme o Exercício 33.3-3 lhe pede para provar, esses dois pontos devem ser vértices da envoltória convexa. Embora isso não seja provado aqui, o par de vértices mais afastados de um polígono convexo de  $n$  vértices pode ser encontrado no tempo  $O(n)$ . Desse modo, calculando a envoltória convexa dos  $n$  pontos de entrada no tempo  $O(n \lg n)$ , e depois encontrando o par mais afastado dos vértices do polígono convexo resultante, podemos localizar o par de pontos mais afastados em qualquer conjunto de  $n$  pontos no tempo  $O(n \lg n)$ .

## Varredura de Graham

A **varredura de Graham** resolve o problema da envoltória convexa, mantendo uma pilha  $S$  de pontos candidatos. Cada ponto do conjunto de entrada  $Q$  é empurrado uma vez para a pilha, e os pontos que não são vértices de  $\text{CH}(Q)$  são eventualmente extraídos da pilha. Quando o algoritmo termina, a pilha  $S$  contém exatamente os vértices de  $\text{CH}(Q)$ , na ordem da direita para a esquerda de seu aparecimento no limite.

O procedimento GRAHAM-SCAN toma como entrada um conjunto  $Q$  de pontos, onde  $|Q| \geq 3$ . Ele chama as funções  $\text{TOP}(S)$ , que retorna o ponto no topo da pilha  $S$  sem alterar  $S$ , e  $\text{NEXT-TO-TOP}(S)$ , que retorna o ponto uma entrada abaixo do topo da pilha  $S$ , sem alterar  $S$ . Como demonstraremos em breve, a pilha  $S$  retornada por GRAHAM-SCAN contém, de baixo para cima, exatamente os vértices de  $\text{CH}(Q)$  na ordem da direita para a esquerda.

### GRAHAM-SCAN( $Q$ )

- 1 seja  $p_0$  o ponto em  $Q$  com a coordenada  $y$  mínima,  
ou o ponto mais à esquerda no caso de uma ligação
- 2 sejam  $\langle p_1, p_2, \dots, p_m \rangle$  os pontos restantes em  $Q$ ,  
ordenados por ângulo polar em ordem da direita para a esquerda em torno de  $p_0$   
(se mais de um ponto tiver o mesmo ângulo, remova todos eles, exceto o mais afastado  
de  $p_0$ )
- 3  $\text{PUSH}(p_0, S)$
- 4  $\text{PUSH}(p_1, S)$
- 5  $\text{PUSH}(p_2, S)$
- 6 **for**  $i \leftarrow 3$  **to**  $m$
- 7   **do while** o ângulo formado pelos pontos  $\text{NEXT-TO-TOP}(S)$ ,  
 $\text{TOP}(S)$  e  $p_i$  formar uma volta não à esquerda
- 8     **do POP**( $S$ )
- 9     **PUSH**( $p_i, S$ )
- 10 **return**  $S$

A Figura 33.7 ilustra o progresso de GRAHAM-SCAN. A linha 1 escolhe o ponto  $p_0$  com a coordenada  $y$  mais baixa, escolhendo o ponto mais à esquerda no caso de uma ligação. Tendo em vista que não existe nenhum ponto em  $Q$  que esteja abaixo de  $p_0$  e todos os outros pontos com a mesma coordenada  $y$  estão à sua direita,  $p_0$  é um vértice de  $\text{CH}(Q)$ . A linha 2 ordena os pontos restantes de  $Q$  por ângulo polar em relação a  $p_0$ , usando o mesmo método – comparação de produtos cruzados – como no Exercício 33.1-2. Se dois ou mais pontos têm o mesmo ângulo polar em relação a  $p_0$ , todos exceto o mais afastado desses pontos são combinações convexas de  $p_0$  e do ponto mais afastado, e assim nós os removemos inteiramente de consideração. Fazemos  $m$  denotar o número de pontos que restam além de  $p_0$ . O ângulo polar, medido em radianos, de cada ponto de  $Q$  em relação a  $p_0$  está no intervalo meio aberto  $[0, \pi)$ . Como os pontos são ordenados de acordo com ângulos polares, eles estão ordenados da direita para a esquerda em relação a  $p_0$ . Designamos essa seqüência ordenada de pontos por  $\langle p_1, p_2, \dots, p_m \rangle$ . Observe que os pontos  $p_1$  e  $p_m$  são vértices de  $\text{CH}(Q)$  (ver Exercício 33.3-1). A Figura 33.7(a) mostra os pontos da Figura 33.6 numerados sequencialmente em ordem de ângulo polar crescente em relação a  $p_0$ .

O restante do procedimento utiliza a pilha  $S$ . As linhas 3 a 5 inicializam a pilha para conter, de baixo para cima, os três primeiros pontos  $p_0, p_1$  e  $p_2$ . A Figura 33.7(a) mostra a pilha inicial  $S$ . O loop **for** das linhas 6 a 9 itera uma vez para cada ponto na subseqüência  $\langle p_3, p_4, \dots, p_m \rangle$ . A intenção é que, depois do processamento do ponto  $p_i$ , a pilha  $S$  contenha, de baixo para cima, os vértices de  $\langle p_0, p_1, \dots, p_i \rangle$  em ordem da direita para a esquerda. O loop **while** das linhas 7 e 8 remove pontos da pilha se for descoberto que eles não são vértices da envoltória convexa. Quando percorremos a envoltória convexa da direita para a esquerda, devemos fazer uma volta à esquerda em cada vértice. Desse modo, toda vez que o loop **while** encontra um vértice no qual fazemos uma volta não à esquerda, o vértice é extraído da pilha. (Verificando uma volta não à esquerda, em vez de verificar apenas uma volta à direita, esse teste impede a possibilidade de um ângulo reto em um vértice da envoltória convexa resultante. Não queremos nenhum ângulo reto, pois nenhum vértice de um polígono convexo pode ser uma combinação convexa de outros vértices do polígono.) Depois de extraímos todos os vértices que têm voltas não à esquerda quando seguimos em direção ao ponto  $p_i$ , empurramos  $p_i$  para a pilha. Os itens (b)–(k) da Figura 33.7 mostram o estado da pilha  $S$  depois de cada iteração do loop **for**. Finalmente, GRAHAM-SCAN retorna a pilha  $S$  na linha 10. A Figura 33.7(l) mostra a envoltória convexa correspondente.

O teorema a seguir prova formalmente a correção de GRAHAM-SCAN.

### **Teorema 33.1 (Correção da varredura de Graham)**

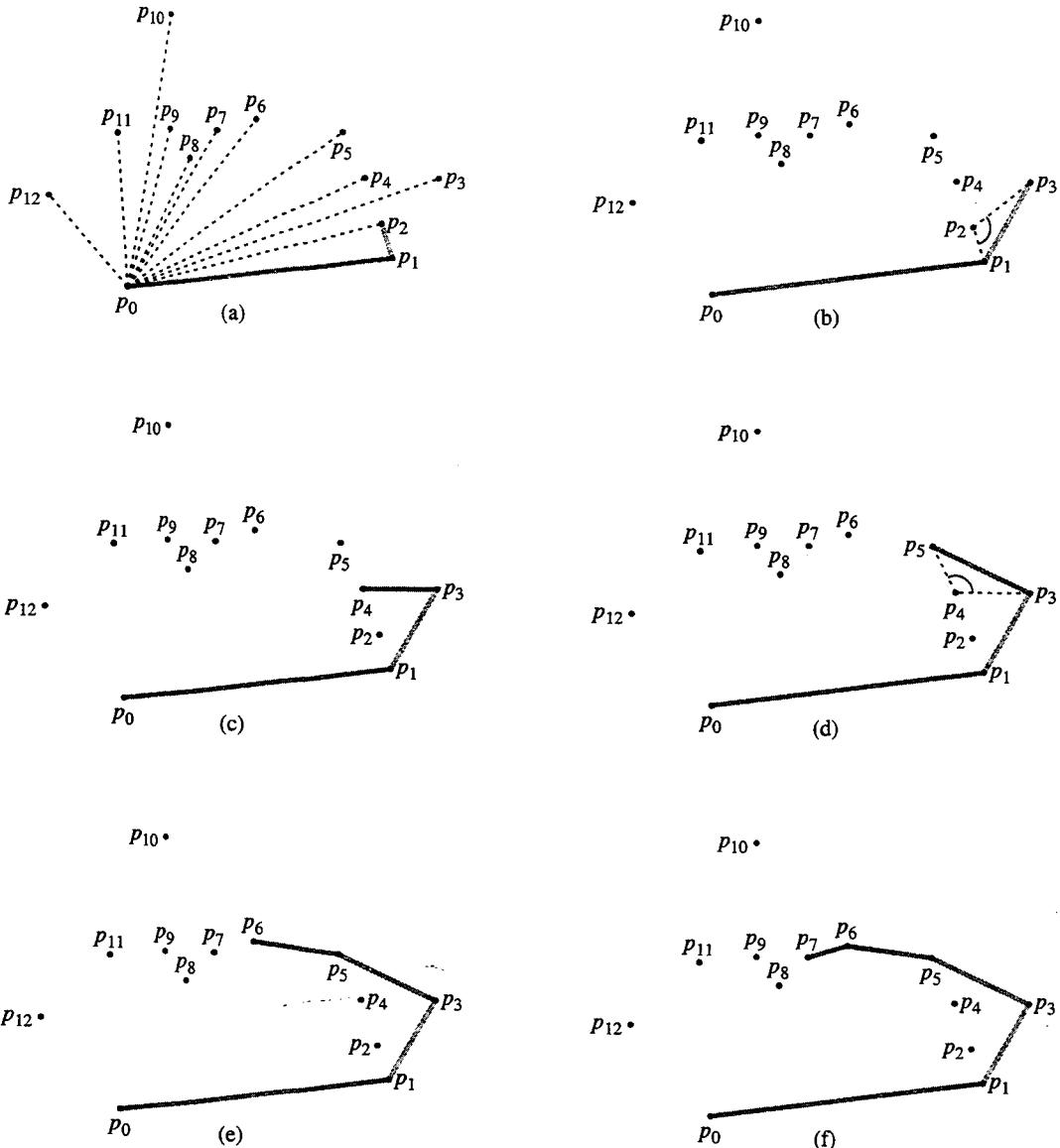
Se GRAHAM-SCAN é executado em um conjunto  $Q$  de pontos, onde  $|Q| \geq 3$ , então, no término, a pilha  $S$  consiste exatamente, de baixo para cima, nos vértices de  $\text{CH}(Q)$  da direita para a esquerda.

**Prova** Depois da linha 2, temos a seqüência de pontos  $\langle p_1, p_2, \dots, p_m \rangle$ . Vamos definir, para  $i = 2, 3, \dots, m$ , o subconjunto de pontos  $Q_i = \{p_0, p_1, \dots, p_i\}$ . Os pontos em  $Q - Q_m$  são aqueles que foram removidos porque tinham o mesmo ângulo polar em relação a  $p_0$  como algum ponto em  $Q_m$ ; esses pontos não estão em  $\text{CH}(Q)$ , e assim  $\text{CH}(Q_m) = \text{CH}(Q)$ . Desse modo, basta mostrar que, quando GRAHAM-SCAN termina, a pilha  $S$  consiste nos vértices de  $\text{CH}(Q_m)$  em ordem da direita para a esquerda e de baixo para cima. Observe que, da mesma maneira que  $p_0, p_1$  e  $p_m$  são vértices de  $\text{CH}(Q)$ , os pontos  $p_0, p_1$  e  $p_i$  são todos vértices de  $\text{CH}(Q_i)$ .

A prova utiliza o seguinte loop invariante:

No início de cada iteração do loop **for** das linhas 6 a 9, a pilha  $S$  consiste, de baixo para cima, exatamente nos vértices de  $\text{CH}(Q_{i-1})$  da direita para a esquerda.

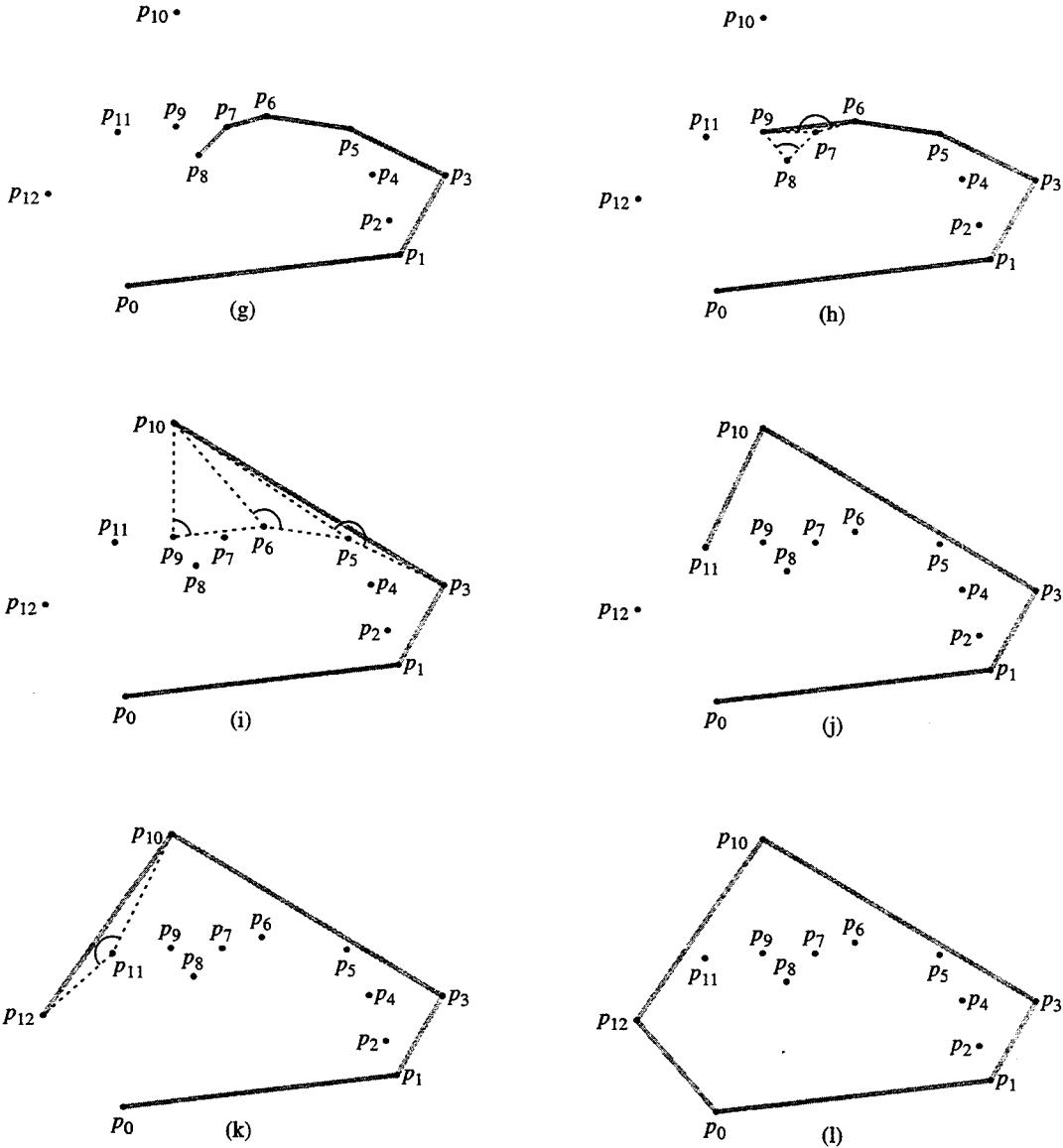
**Inicialização:** O invariante é válido na primeira vez que executamos a linha 6 pois, nesse momento, a pilha  $S$  consiste exatamente nos vértices de  $Q_2 = Q_{i-1}$ , e esse conjunto de três vértices forma sua própria envoltória convexa. Além disso, eles aparecem da direita para a esquerda e de baixo para cima.



**FIGURA 33.7** A execução de GRAHAM-SCAN sobre o conjunto  $Q$  da Figura 33.6. A envoltória convexa atual contida na pilha  $S$  é mostrada em cinza a cada passo. (a) A seqüência  $\langle p_1, p_2, \dots, p_{12} \rangle$  de pontos numerados em ordem de ângulo polar crescente em relação a  $p_0$ , e a pilha inicial  $S$  contendo  $p_0, p_1$  e  $p_2$ . (b)-(k) A pilha  $S$  depois de cada iteração do loop **for** das linhas 6 a 9. Linhas tracejadas mostram voltas não à esquerda, que provocam a extração de pontos da pilha. Por exemplo, na parte (h) a volta para a direita no ângulo  $\angle p_7 p_8 p_9$ , faz  $p_8$  ser extraído, e depois a volta para a direita no ângulo  $\angle p_6 p_7 p_9$ , faz  $p_7$  ser extraído. (l) A envoltória convexa retornada pelo procedimento, que corresponde à da Figura 33.6

**Manutenção:** Na entrada de uma iteração do loop **for**, o ponto superior na pilha  $S$  é  $p_{i-1}$ , que foi empurrado no fim da iteração anterior (ou antes da primeira iteração, quando  $i = 3$ ). Seja  $p_j$  o ponto superior em  $S$  depois do loop **while** das linhas 7 e 8 ser executado, mas antes da linha 9 empurrar  $p_i$ , e seja  $p_k$  o ponto imediatamente abaixo de  $p_j$  em  $S$ . No momento em que  $p_j$  é o ponto superior em  $S$  e ainda não empurramos  $p_i$ , a pilha  $S$  contém exatamente os mesmos pontos que continha depois da iteração  $j$  do loop **for**. Então, pelo loop invariante,  $S$  contém exatamente os vértices de  $\text{CH}(Q_j)$  naquele momento, e eles aparecem em ordem da direita para a esquerda e de baixo para cima.

Vamos continuar a nos concentrar nesse momento imediatamente antes de  $p_i$  ser empurrado. Voltando à Figura 33.8(a), como o ângulo polar de  $p_i$  em relação a  $p_0$  é maior que o ângulo polar de  $p_j$ , e como o ângulo  $\angle p_k p_j p_i$  faz uma volta à esquerda (do contrário, teríamos extraído

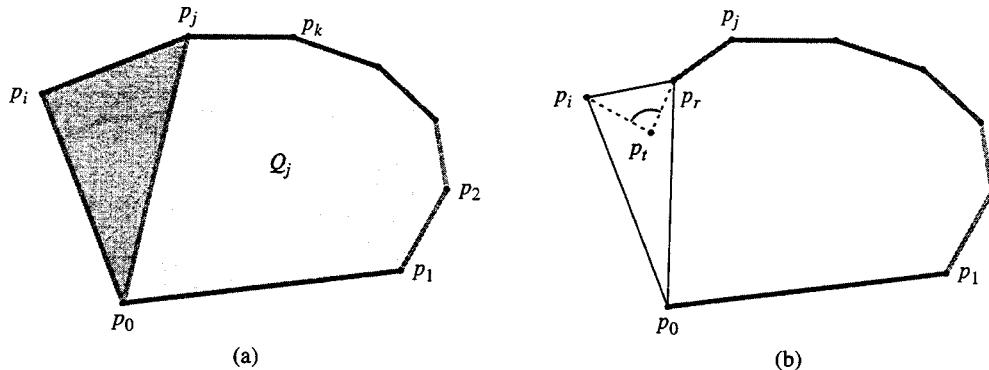


$p_j$ ), vemos que pelo fato de  $S$  conter exatamente os vértices de  $\text{CH}(Q_j)$ , uma vez que empurramos  $p_i$ , a pilha  $S$  conterá exatamente os vértices de  $\text{CH}(Q_j \cup \{p_i\})$ , ainda em ordem da direita para a esquerda e baixo para cima.

Agora, mostramos que  $\text{CH}(Q_j \cup \{p_i\})$  é o mesmo conjunto de pontos que  $\text{CH}(Q_i)$ . Considere qualquer ponto  $p_r$  que tenha sido extraído durante a iteração  $i$  do loop **for**, e seja  $p_t$  o ponto imediatamente abaixo de  $p_r$  na pilha  $S$  no momento em que  $p_r$  foi extraído ( $p_r$  poderia ser  $p_j$ ). O ângulo  $\angle p_r p_t p_i$  faz uma volta não à esquerda, e o ângulo polar de  $p_t$  em relação a  $p_0$  é maior que o ângulo polar de  $p_r$ . Como mostra a Figura 33.8(b),  $p_t$  deve estar no interior do triângulo formado por  $p_0$ ,  $p_r$  e  $p_i$  ou em um lado desse triângulo (mas ele não é um vértice do triângulo). É claro que, tendo em vista que  $p_t$  está dentro de um triângulo formado por três outros pontos de  $Q_i$ , ele não pode ser um vértice de  $\text{CH}(Q_i)$ . Tendo em vista que  $p_t$  não é um vértice de  $\text{CH}(Q_i)$ , temos que

$$\text{CH}(Q_i - \{p_t\}) = \text{CH}(Q_i) . \quad (33.1)$$

Seja  $P_i$  ser o conjunto de pontos que foram extraídos durante a iteração  $i$  do loop **for**. Tendo em vista que a igualdade (33.1) se aplica a todos os pontos em  $P_i$ , podemos aplicá-la repetidamente para mostrar que  $\text{CH}(Q_i - P_i) = \text{CH}(Q_i)$ . Porém,  $Q_i - P_i = Q_j - \{p_i\}$ , e assim concluímos que  $\text{CH}(Q_j - \{p_i\}) = \text{CH}(Q_j - P_i) = \text{CH}(Q_j)$ .



**FIGURA 33.8** A prova da correção de GRAHAM-SCAN. (a) Como o ângulo polar de  $p_i$ , em relação a  $p_0$ , é maior que o ângulo polar de  $p_j$ , e como o ângulo  $\angle p_k p_j p_i$  faz uma volta à esquerda, a adição de  $p_i$  a  $\text{CH}(Q_j)$  fornece exatamente os vértices de  $\text{CH}(Q_j \cup \{p_i\})$ . Se o ângulo  $\angle p_t p_i p_j$  faz uma volta não à esquerda, então  $p_t$  está no interior do triângulo formado por  $p_0, p_t$  e  $p_i$ , ou em um lado do triângulo, e não pode ser um vértice de  $\text{CH}(Q_i)$ .

Mostramos que, depois de empurrarmos  $p_i$ , a pilha  $S$  contém exatamente os vértices de  $\text{CH}(Q_i)$  em ordem da direita para a esquerda e de baixo para cima. Então, incrementar  $i$  tornará o loop invariante válido para a próxima iteração.

**Término:** Quando o loop termina, temos  $i = m + 1$ , e assim o loop invariante implica que a pilha  $S$  consiste exatamente nos vértices de  $\text{CH}(Q_m)$ , que é  $\text{CH}(Q)$ , em ordem da direita para a esquerda e de baixo para cima. Isso completa a prova. ■

Agora, mostraremos que o tempo de execução de GRAHAM-SCAN é  $O(n \lg n)$ , onde  $n = |Q|$ . A linha 1 demora o tempo  $\Theta(n)$ . A linha 2 demora o tempo  $O(n \lg n)$ , usando a ordenação por intercalação ou heapsort para ordenar os ângulos polares, e o método de produtos cruzados da Seção 33.1 para comparar ângulos. (A remoção de todos os pontos exceto o mais afastado com o mesmo ângulo polar pode ser feita em um tempo total  $O(n)$ .) As linhas 3 a 5 demoram o tempo  $O(1)$ . Como  $m \leq n - 1$ , o loop **for** das linhas 6 a 9 é executado no máximo  $n - 3$  vezes. Tendo em vista que PUSH demora o tempo  $O(1)$ , cada iteração demora o tempo  $O(1)$  excluindo-se o tempo gasto no loop **while** das linhas 7 e 8; assim, o tempo global para loop **for** é  $O(n)$ , sem contar o loop **while** aninhado.

Usamos a análise agregada para mostrar que o loop **while** demora o tempo total  $O(n)$ . Para  $i = 0, 1, \dots, m$ , cada ponto  $p_i$  é empurrado sobre a pilha  $S$  exatamente uma vez. Como na análise do procedimento MULTIPOP da Seção 17.1, observamos que existe no máximo uma operação POP para cada operação PUSH. Pelo menos três pontos –  $p_0, p_1$  e  $p_m$  – nunca são extraídos da pilha; assim, no máximo  $m - 2$  operações POP são de fato executadas no total. Cada iteração do loop **while** executa uma operação POP, e assim existem no máximo  $m - 2$  iterações do loop **while** no total. Tendo em vista que o teste da linha 7 demora o tempo  $O(1)$ , cada chamada de POP demora o tempo  $O(1)$  e, como  $m \leq n - 1$ , o tempo total tomado pelo loop **while** é  $O(n)$ . Desse modo, o tempo de execução de GRAHAM-SCAN é  $O(n \lg n)$ .

## Marcha de Jarvis

A **Marcha de Jarvis** calcula a envoltória convexa de um conjunto  $Q$  de pontos por uma técnica conhecida como **embrulhar pacote** (ou **embrulha presente**). O algoritmo é executado no tempo  $O(nb)$ , onde  $b$  é o número de vértices de  $\text{CH}(Q)$ . Quando  $b$  é  $O(\lg n)$ , a marcha de Jarvis é assintoticamente mais rápida que a varredura de Graham.

Intuitivamente, a marcha de Jarvis simula a ação de embrulhar com uma folha de papel o conjunto  $Q$ . Começamos colocando a extremidade do papel no ponto mais baixo do conjunto,

isto é, no mesmo ponto  $p_0$  com o qual começamos a varredura de Graham. Esse ponto é um vértice da envoltória convexa. Puxamos o papel para a direita a fim de esticá-lo, e depois o puxamos para cima até tocar um ponto. Esse ponto também deve ser um vértice da envoltória convexa. Mantendo o papel esticado, continuamos desse modo em torno do conjunto de vértices, até voltarmos ao nosso ponto original  $p_0$ .

Mais formalmente, a marcha de Jarvis constrói uma seqüência  $H = \langle p_1, p_2, \dots, p_{b-1} \rangle$  dos vértices de  $\text{CH}(Q)$ . Começamos com  $p_0$ . Como mostra a Figura 33.9, o próximo vértice  $p_1$  da envoltória convexa tem o menor ângulo polar com relação a  $p_0$ . (No caso de ligações, escolhemos o ponto mais afastado de  $p_0$ .) De modo semelhante,  $p_2$  tem o menor ângulo polar com relação a  $p_1$  e assim por diante. Quando alcançamos o vértice mais alto, digamos  $p_k$  (rompendo as ligações pela escolha do vértice mais afastado), construímos, como mostra a Figura 33.9, a **cadeia da direita** de  $\text{CH}(Q)$ . Para construir a **cadeia da esquerda**, começamos em  $p_k$  e escolhemos  $p_{k+1}$  como o ponto com o menor ângulo polar em relação a  $p_k$ , mas *a partir do eixo x negativo*. Continuamos assim, formando a cadeia da esquerda pelo uso de ângulos polares a partir do eixo x negativo, até voltarmos ao nosso vértice original  $p_0$ .

Poderíamos implementar a marcha de Jarvis em uma varredura conceitual em torno da envoltória convexa, ou seja, sem construir separadamente as cadeias da direita e da esquerda. Em geral, tais implementações controlam o ângulo do último lado da envoltória convexa escolhido e exigem que a seqüência de ângulos de lados da envoltória seja estritamente crescente (no intervalo de 0 a  $2\pi$  radianos). A vantagem de construir cadeias separadas é que não precisamos calcular ângulos de forma explícita; as técnicas da Seção 33.1 bastam para comparar ângulos.

Se implementada corretamente, a marcha de Jarvis tem um tempo de execução  $O(nb)$ . Para cada um dos  $b$  vértices de  $\text{CH}(Q)$ , encontramos o vértice com o ângulo polar mínimo. Cada comparação entre ângulos polares demora o tempo  $O(1)$ , usando as técnicas da Seção 33.1. Como mostra a Seção 9.1, podemos calcular no mínimo  $n$  valores no tempo  $O(n)$ , se cada comparação demorar o tempo  $O(1)$ . Assim, a marcha de Jarvis demora o tempo  $O(nb)$ .

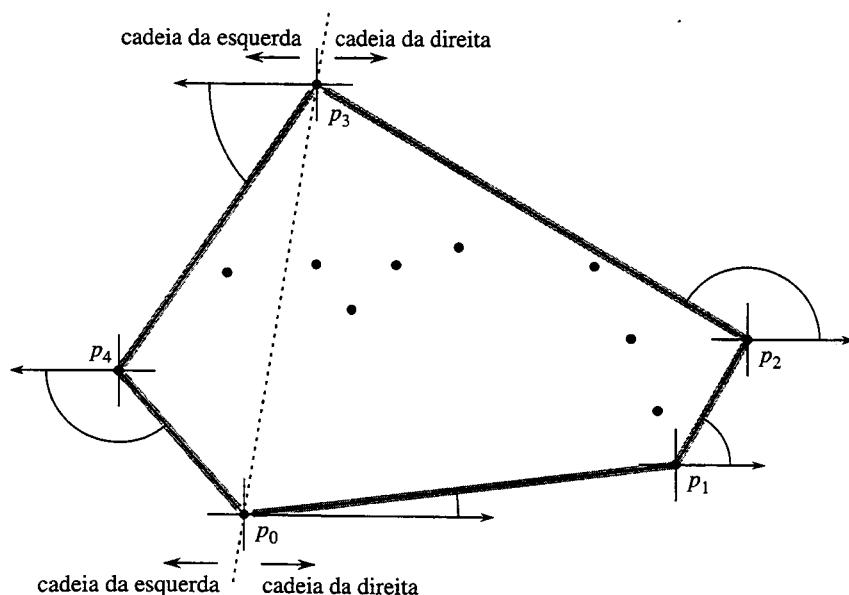


FIGURA 33.9 A operação da marcha de Jarvis. O primeiro vértice escolhido é o ponto mais baixo  $p_0$ . O vértice seguinte,  $p_1$ , tem o menor ângulo polar de qualquer ponto em relação a  $p_0$ . Em seguida,  $p_2$  tem o menor ângulo polar em relação a  $p_1$ . A cadeia da direita chega até o ponto mais alto  $p_3$ . Então, a cadeia da esquerda é construída, encontrando-se ângulos polares mínimos com relação ao eixo x negativo

## Exercícios

### 33.3-1

Prove que, no procedimento GRAHAM-SCAN, os pontos  $p_1$  e  $p_m$  devem ser vértices de  $\text{CH}(Q)$ .

### 33.3-2

Considere um modelo de computação que admite adição, comparação e multiplicação, e para o qual exista um limite inferior  $\Omega(n \lg n)$  para ordenar  $n$  números. Prove que  $\Omega(n \lg n)$  é um limite inferior para calcular, na ordem, os vértices da envoltória convexa de um conjunto de  $n$  pontos em tal modelo.

### 33.3-3

Dado um conjunto de pontos  $Q$ , prove que os pares de pontos mais afastados entre si devem ser vértices de  $\text{CH}(Q)$ .

### 33.3-4

Para um dado polígono  $P$  e um ponto  $q$  em seu limite, a *sombra* de  $q$  é o conjunto de pontos  $r$  tal que o segmento  $qr$  está inteiramente no limite ou no interior de  $P$ . Um polígono  $P$  é *estrelado* se existe um ponto  $p$  no interior de  $P$  que está na sombra de todo ponto no limite de  $P$ . O conjunto de todos esses pontos  $p$  é chamado *núcleo* de  $P$ . (Ver Figura 33.10.) Dado um polígono estrelado  $P$  de  $n$  vértices especificado por seus vértices em ordem da direita para a esquerda, mostre como calcular  $\text{CH}(P)$  no tempo  $O(n)$ .

### 33.3-5

No *problema da envoltória convexa on-line*, temos o conjunto  $Q$  de  $n$  pontos, um ponto de cada vez. Depois de receber cada ponto, temos de calcular a envoltória convexa dos pontos vistos até o momento. É óbvio que poderíamos executar a varredura de Graham uma vez para cada ponto, com o tempo de execução total  $O(n^2 \lg n)$ . Mostre como resolver o problema da envoltória convexa on-line no tempo total  $O(n^2)$ .

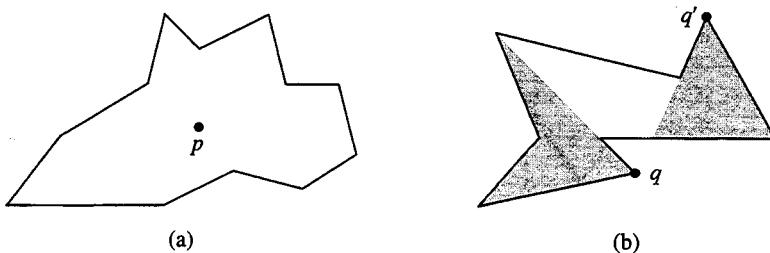


FIGURA 33.10 A definição de um polígono estrelado para uso no Exercício 33.3-4. (a) Um polígono estrelado. O segmento desde o ponto  $p$  até qualquer ponto  $q$  no limite cruza o limite apenas em  $q$ . (b) Um polígono não estrelado. A região sombreada à esquerda é a sombra de  $q$ , e a região sombreada à direita é a sombra de  $q'$ . Como essas regiões são disjuntas, o núcleo está vazio

### 33.3-6 \*

Mostre como implementar o método incremental para calcular a envoltória convexa de  $n$  pontos, de modo que ele seja executado no tempo  $O(n \lg n)$ .

## 33.4 Localização do par de pontos mais próximos

Agora, vamos considerar o problema de encontrar o par de pontos mais próximos em um conjunto  $Q$  de  $n \geq 2$  pontos. A expressão “mais próximos” se refere à distância euclidiana habitual: a distância entre os pontos  $p_1 = (x_1, y_1)$  e  $p_2 = (x_2, y_2)$  é  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ . Dois pontos no conjunto  $Q$  podem ser coincidentes e, nesse caso, a distância entre eles é zero. Esse problema

tem aplicação, por exemplo, em sistemas de controle de tráfego. Um sistema para controlar o tráfego aéreo ou marítimo talvez precise saber quais são os dois veículos mais próximos entre si, a fim de detectar colisões potenciais.

Um algoritmo de pares mais próximos por força bruta simplesmente examina todos os  $\binom{n}{2}$  pares de pontos. Nesta seção, descreveremos um algoritmo de dividir e conquistar para esse problema, cujo tempo de execução é descrito pela recorrência familiar  $T(n) = 2T(n/2) + O(n)$ . Desse modo, o algoritmo usa apenas o tempo  $O(n \lg n)$ .

## O algoritmo de dividir e conquistar

Cada chamada recursiva do algoritmo toma como entrada um subconjunto  $P \subseteq Q$  e os arranjos  $X$  e  $Y$ , cada um contendo todos os pontos do subconjunto de entrada  $P$ . Os pontos no arranjo  $X$  estão ordenados de tal forma que suas coordenadas  $x$  são monotonicamente crescentes. De modo semelhante, o arranjo  $Y$  é ordenado por coordenada  $y$  monotonicamente crescente. Observe que, para nos restringirmos ao limite de tempo  $O(n \lg n)$ , não podemos nos permitir efetuar a ordenação em cada chamada recursiva; se o fizéssemos, a recorrência para o tempo de execução seria  $T(n) = 2T(n/2) + O(n \lg n)$ , cuja solução é  $T(n) = O(n \lg^2 n)$ . (Use a versão do método mestre dada no Exercício 4.4-2.) Veremos um pouco mais adiante como usar a “pré-ordenação” para manter essa propriedade ordenada sem efetuar realmente uma ordenação em cada chamada recursiva.

Uma dada chamada recursiva com entradas  $P, X$  e  $Y$  verifica primeiro se  $|P| \leq 3$ . Nesse caso, a chamada simplesmente executa o método de força bruta descrito anteriormente: experimentar todos os  $\binom{|P|}{2}$  pares de pontos e retornar o par de pontos mais próximos. Se  $|P| > 3$ , a chamada recursiva executa o paradigma de dividir e conquistar descrito a seguir.

**Dividir:** Ela encontra uma linha vertical  $l$  que divide o conjunto de pontos  $P$  em dois conjuntos  $P_L$  e  $P_R$  tais que  $|P_L| = \lceil |P|/2 \rceil$ ,  $|P_R| = \lfloor |P|/2 \rfloor$ , todos os pontos em  $P_L$  estão sobre ou à esquerda da linha  $l$ , e todos os pontos em  $P_R$  estão sobre ou à direita de  $l$ . O arranjo  $X$  é dividido em arranjos  $X_L$  e  $X_R$ , que contêm os pontos de  $P_L$  e  $P_R$  respectivamente, ordenados por coordenada  $x$  monotonicamente crescente. De modo semelhante, o arranjo  $Y$  é dividido nos arranjos  $Y_L$  e  $Y_R$ , que contêm os pontos de  $P_L$  e  $P_R$  respectivamente, ordenados por coordenada  $y$  monotonicamente crescente.

**Conquistar:** Tendo  $P$  dividido em  $P_L$  e  $P_R$ , ele faz duas chamadas recursivas, uma para encontrar o par de pontos mais próximos em  $P_L$  e a outra para encontrar o par de pontos mais próximos em  $P_R$ . As entradas para a primeira chamada são o subconjunto  $P_L$  e os arranjos  $X_L$  e  $Y_L$ ; a segunda chamada recebe as entradas  $P_R, X_R$  e  $Y_R$ . Sejam  $\delta_L$  e  $\delta_R$  as distâncias de pares mais próximos retornadas por  $P_L$  e  $P_R$ , respectivamente, e seja  $\delta = \min(\delta_L, \delta_R)$ .

**Combinar:** O par de pontos mais próximos é qualquer par com distância  $\delta$  encontrado por uma das chamadas recursivas, ou é um par de pontos com um ponto em  $P_L$  e o outro em  $P_R$ . O algoritmo determina se existe um tal par cuja distância é menor que  $\delta$ . Observe que, se existe um par de pontos com distância menor que  $\delta$ , ambos os pontos do par devem estar dentro de  $\delta$  unidades da linha  $l$ . Desse modo, como mostra a Figura 33.11(a) ambos devem residir na faixa vertical de largura  $2\delta$  com centro na linha  $l$ . Para encontrar tal par, se existe algum, o algoritmo faz o seguinte.

1. Cria um arranjo  $Y'$ , que é o arranjo  $Y$  com todos os pontos que não estão na faixa vertical de largura  $2\delta$  removidos. O arranjo  $Y'$  é ordenado por coordenada  $y$ , exatamente como  $Y$ .
2. Para cada ponto  $p$  no arranjo  $Y'$ , o algoritmo tenta encontrar pontos em  $Y'$  que estejam dentro de  $\delta$  unidades de  $p$ . Como veremos em breve, apenas os 7 pontos em  $Y'$  que seguem  $p$  precisam ser considerados. O algoritmo calcula a distância desde  $p$  até cada um desses 7 pontos e controla a distância  $\delta'$  do par de pontos mais próximos encontrada sobre todos os pares de pontos em  $Y'$ .

3. Se  $\delta' < \delta$ , então a faixa vertical de fato contém um par de pontos mais próximos do que aquele que foi encontrado pelas chamadas recursivas. Esse par e sua distância  $\delta'$  são retornados. Caso contrário, o par de pontos mais próximos e sua distância  $\delta$  encontrada pelas chamadas recursivas são retornados.

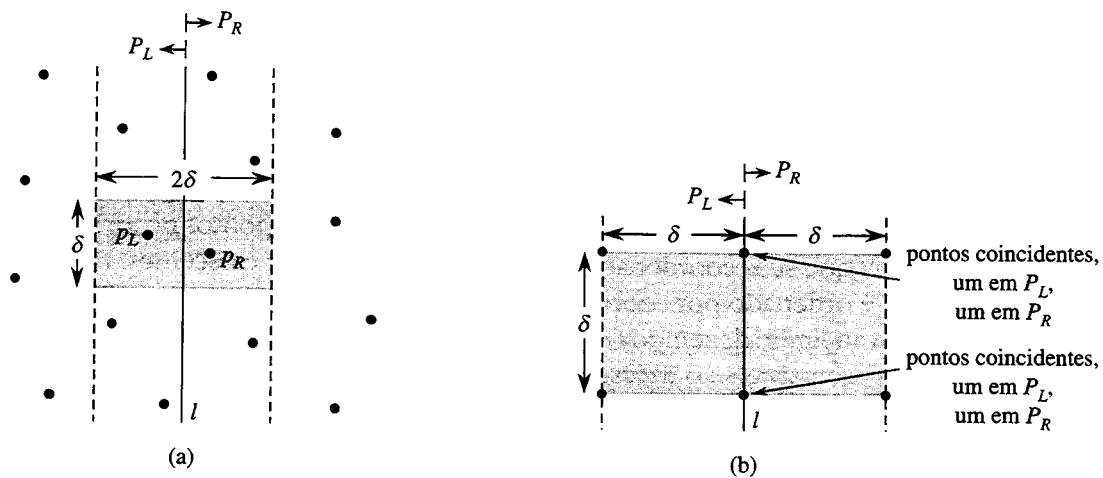


FIGURA 33.11 Conceitos fundamentais na prova de que o algoritmo de pares de pontos mais próximos só precisa verificar 7 pontos seguintes a cada ponto no arranjo  $Y$ . (a) Se  $p_L \in P_L$  e  $p_R \in P_R$  estão afastados entre si menos de  $\delta$  unidades, eles devem residir dentro de um retângulo  $\delta \times 2\delta$  com centro na linha  $l$ . (b) O modo como 4 pontos que estão afastados aos pares pelo menos  $\delta$  unidades podem residir dentro de um quadrado  $\delta \times \delta$ . À esquerda estão 4 pontos em  $P_L$ , e à direita estão 4 pontos em  $P_R$ . Podem existir 8 pontos no retângulo  $\delta \times 2\delta$ , se os pontos mostrados na linha  $l$  são na realidade pares de pontos coincidentes com um ponto em  $P_L$  e um em  $P_R$ .

A descrição anterior omite alguns detalhes de implementação que são necessários para se alcançar o tempo de execução  $O(n \lg n)$ . Depois de provar a correção do algoritmo, mostraremos como implementá-lo para alcançar o limite de tempo desejado.

## Correção

A correção desse algoritmo de pares mais próximos é óbvia, exceto por dois aspectos. Primeiro, interrompendo a recursão quando  $|P| \leq 3$ , garantimos que nunca tentaremos resolver um subproblema consistindo em apenas um ponto. O segundo aspecto é que só precisamos verificar os 7 pontos que seguem cada ponto  $p$  no arranjo  $Y$ ; provaremos agora essa propriedade.

Suponha que, em algum nível da recursão, o par de pontos mais próximos seja  $p_L \in P_L$  e  $p_R \in P_R$ . Desse modo, a distância  $\delta'$  entre  $p_L$  e  $p_R$  é estritamente menor que  $\delta$ . O ponto  $p_L$  deve estar sobre ou à esquerda da linha  $l$  e afastado menos de  $\delta$  unidades. De modo semelhante,  $p_R$  está sobre ou à direita de  $l$  e afastado menos de  $\delta$  unidades. Além disso,  $p_L$  e  $p_R$  estão dentro de  $\delta$  unidades um do outro na direção vertical. Portanto, como mostra a Figura 33.11(a),  $p_L$  e  $p_R$  estão dentro de um retângulo  $\delta \times 2\delta$  com centro na linha  $l$ . (Também é possível que existam outros pontos dentro desse retângulo.)

Mostraremos em seguida que no máximo 8 pontos de  $P$  podem residir dentro desse retângulo  $\delta \times 2\delta$ . Considere o quadrado  $\delta \times \delta$  que forma a metade esquerda desse retângulo. Tendo em vista que todos os pontos dentro de  $P_L$  estão afastados pelo menos  $\delta$  unidades, no máximo 4 pontos podem residir dentro desse quadrado; a Figura 33.11(b) mostra como. De modo semelhante, no máximo 4 pontos em  $P_R$  podem residir dentro do quadrado  $\delta \times \delta$  que forma a metade direita do retângulo. Desse modo, no máximo 8 pontos de  $P$  podem residir no interior do retângulo  $\delta \times 2\delta$ . (Observe que, como os pontos na linha  $l$  podem estar em  $P_L$  ou  $P_R$ , é possível que

existam até 4 pontos em  $l$ . Esse limite é alcançado se existem dois pares de pontos coincidentes, cada par consistindo em um ponto de  $P_L$  e um ponto de  $P_R$ ; um par está na interseção de  $l$  com a parte superior do retângulo, e o outro par está onde  $l$  cruza a parte inferior do retângulo.)

Tendo mostrado que no máximo 8 pontos de  $P$  podem residir no interior do retângulo, é fácil ver que precisamos verificar apenas os 7 pontos que seguem cada ponto no arranjo  $Y'$ . Ainda supondo que o par de pontos mais próximos seja  $p_L$  e  $p_R$ , vamos supor sem perda de generalidade de que  $p_L$  preceda  $p_R$  no arranjo  $Y'$ . Então, mesmo que  $p_L$  ocorra o mais cedo possível em  $Y'$  e  $p_R$  ocorra o mais tarde possível,  $p_R$  estará em uma das sete posições que seguem  $p_L$ . Portanto, mostramos a correção do algoritmo de pares mais próximos.

## Implementação e tempo de execução

Como observamos, nossa meta é fazer a recorrência para o tempo de execução ser  $T(n) = 2T(n/2) + O(n)$ , onde  $T(n)$  é o tempo de execução para um conjunto de  $n$  pontos. A principal dificuldade está em assegurar que os arranjos  $X_L, X_R, Y_L$  e  $Y_R$ , que são repassados a chamadas recursivas, estão ordenados pela coordenada apropriada, e também que o arranjo  $Y'$  está ordenado por coordenada  $y$ . (Observe que, se o arranjo  $X$  que é recebido por uma chamada recursiva já estiver ordenado, então a divisão do conjunto  $P$  em  $P_L$  e  $P_R$  será realizada com facilidade em tempo linear.)

A observação fundamental é que, em cada chamada, desejamos formar um subconjunto ordenado de um arranjo ordenado. Por exemplo, uma chamada particular recebe o subconjunto  $P$  e o arranjo  $Y$ , ordenado pela coordenada  $y$ . Tendo particionado  $P$  em  $P_L$  e  $P_R$ , ela precisa formar os arranjos  $Y_L$  e  $Y_R$ , os quais são ordenados por coordenada  $y$ . Além disso, esses arranjos devem ser formados em tempo linear. O método pode ser visto como o oposto do procedimento MERGE da ordenação por intercalação na Seção 2.3.1: estamos dividindo um arranjo ordenado em dois arranjos ordenados. O pseudocódigo a seguir mostra a idéia.

```

1 comprimento[ $Y_L$ ]  $\leftarrow$  comprimento[ $Y_R$ ] ? 0
2 for  $i \leftarrow 1$  to comprimento[ $Y$ ]
3   do if  $Y[i] \in P_L$ 
4     then comprimento[ $Y_L$ ]  $\leftarrow$  comprimento[ $Y_L$ ] + 1
5      $Y_L[\text{comprimento}[Y_L]] \leftarrow Y[i]$ 
6   else comprimento[ $Y_R$ ]  $\leftarrow$  comprimento[ $Y_R$ ] + 1
7      $Y_R[\text{comprimento}[Y_R]] \leftarrow Y[i]$ 
```

Simplesmente examinamos os pontos do arranjo  $Y$  em ordem. Se um ponto  $Y[i]$  está em  $P_L$ , anexamos o ponto ao final do arranjo  $Y_L$ ; caso contrário, nós o anexamos ao final do arranjo  $Y_R$ . Um pseudocódigo semelhante funciona para formar os arranjos  $X_L, X_R$  e  $Y'$ .

A única pergunta que resta é como obter os pontos iniciais ordenados. Fazemos isso simplesmente executando a **pré-ordenação** dos pontos; isto é, ordenamos os pontos de uma vez por todas *antes* da primeira chamada recursiva. Esses arranjos ordenados são repassados para a primeira chamada recursiva, e daí são reduzidos gradualmente através das chamadas recursivas, conforme necessário. A pré-ordenação acrescenta um valor  $O(n \lg n)$  adicional ao tempo de execução, mas agora cada passo da recursão demora um tempo linear com exceção das chamadas recursivas. Desse modo, se  $T(n)$  for o tempo de execução de cada passo recursivo e  $T'(n)$  for o tempo de execução do algoritmo inteiro, obteremos  $T'(n) = T(n) + O(n \lg n)$ , e

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{se } n > 3 , \\ O(1) & \text{se } n \leq 3 . \end{cases}$$

Desse modo,  $T(n) = O(n \lg n)$  e  $T'(n) = O(n \lg n)$ .

## Exercícios

### 33.4-1

O professor Samuel apresenta um esquema que permite ao algoritmo de pares de pontos mais próximos verificar apenas 5 pontos que seguem cada ponto no arranjo  $Y'$ . A idéia é sempre inserir os pontos sobre a linha  $l$  no conjunto  $P_L$ . Então, não podem existir pares de pontos coincidentes na linha  $l$  com um ponto em  $P_L$  e um em  $P_R$ . Assim, no máximo 6 pontos podem residir no retângulo  $\delta \times 2\delta$ . Qual é a falha no esquema do professor?

### 33.4-2

Sem aumentar o tempo de execução assintótico do algoritmo, mostre como assegurar que o conjunto de pontos repassados à primeira chamada recursiva não contém nenhum ponto coincidente. Prove então que basta verificar os pontos nas 5 posições do arranjo que seguem cada ponto no arranjo  $Y'$ .

### 33.4-3

A distância entre dois pontos pode ser definida de outras maneiras além da euclidiana. No plano, a **distância de  $L_m$**  entre os pontos  $p_1$  e  $p_2$  é dada pela expressão  $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$ . Então, a distância euclidiana é a distância de  $L_2$ . Modifique o algoritmo de pares mais próximos para usar a distância de  $L_1$ , que também é conhecida como **distância de Manhattan**.

### 33.4-4

Dados dois pontos  $p_1$  e  $p_2$  no plano, a distância de  $L_\infty$  entre eles é  $(|x_1 - x_2|, |y_1 - y_2|)$ . Modifique o algoritmo de pares de pontos mais próximos para utilizar a distância de  $L_\infty$ .

### 33.4-5

Sugira uma mudança no algoritmo de pares mais próximos que evite a pré-ordenação do arranjo  $Y$ , mas deixe o tempo de execução como  $O(n \lg n)$ . (Sugestão: Faça a intercalação dos arranjos ordenados  $Y_L$  e  $Y_R$  para formar o arranjo ordenado  $Y$ .)

## Problemas

### 33-1 Camadas convexas

Dado um conjunto  $Q$  de pontos no plano, definimos as **camadas convexas** de  $Q$  de forma induativa. A primeira camada convexa de  $Q$  consiste nos pontos de  $Q$  que são vértices de  $\text{CH}(Q)$ . Para  $i > 1$ , definimos  $Q_i$  consistindo nos pontos de  $Q$ , com todos os pontos em camadas convexas  $1, 2, \dots, i-1$  removidos. Então, a  $i$ -ésima camada convexa de  $Q$  é  $\text{CH}(Q_i)$  se  $Q \neq \emptyset$  e é indefinida em caso contrário.

- Forneça um algoritmo de tempo  $O(n^2)$  para encontrar as camadas convexas de um conjunto de  $n$  pontos.
- Prove que o tempo  $\Omega(n \lg n)$  é necessário para calcular as camadas convexas de um conjunto de  $n$  pontos em qualquer modelo de computação que exija o tempo  $\Omega(n \lg n)$  para ordenar  $n$  números reais.

### 33-2 Camadas máximas

Seja  $Q$  um conjunto de  $n$  pontos no plano. Dizemos que o ponto  $(x, y)$  **domina** o ponto  $(x', y')$  se  $x \geq x'$  e  $y \geq y'$ . Um ponto em  $Q$  que não é dominado por nenhum outro ponto em  $Q$  é chamado **máximo**. Observe que  $Q$  pode conter muitos pontos máximos, os quais podem estar organizados em **camadas máximas** como a seguir. A primeira camada máxima  $L_1$  é o conjunto de pontos máximos de  $Q$ . Para  $i > 1$ , a  $i$ -ésima camada máxima  $L_i$  é o conjunto de pontos máximos em  $Q - \bigcup_{j=1}^{i-1} L_j$ .

Suponha que  $Q$  tenha  $k$  camadas máximas não vazias, e seja  $y_i$  a coordenada  $y$  do ponto mais à esquerda em  $L_i$ , para  $i = 1, 2, \dots, k$ . Por enquanto, supomos que não há dois pontos em  $Q$  com a mesma coordenada  $x$  ou  $y$ .

- Mostre que  $y_1 > y_2 > \dots > y_k$ .

Considere um ponto  $(x, y)$  que está à esquerda de qualquer ponto em  $Q$  e para o qual  $y$  é distinto da coordenada  $y$  de qualquer ponto em  $Q$ . Seja  $Q' = Q \cup [(x, y)]$ .

- Seja  $j$  o índice mínimo tal que  $y_j < y$ , a menos que  $y < y_k$ , em cujo caso fazemos  $j = k + 1$ . Mostre que as camadas máximas de  $Q'$  são as seguintes.

- Se  $j \leq k$ , então as camadas máximas de  $Q'$  são iguais às camadas máximas de  $Q$ , exceto por  $L_j$  também incluir  $(x, y)$  como seu novo ponto mais à esquerda.
- Se  $j = k + 1$ , então as primeiras  $k$  camadas máximas de  $Q'$  são iguais às de  $Q$  mas, além disso,  $Q'$  tem uma  $(k + 1)$ -ésima camada máxima não vazia:  $L_{k+1} = [(x, y)]$ .
- c. Descreva um algoritmo de tempo  $O(n \lg n)$  para calcular as camadas máximas de um conjunto  $Q$  de  $n$  pontos. (Sugestão: Mova uma linha de varredura da direita para a esquerda.)
- d. Surgirá alguma dificuldade se agora permitirmos que pontos de entrada tenham a mesma coordenada  $x$  ou  $y$ ? Sugira um modo de solucionar tais problemas.

### 33-3 Caça-fantasmas e fantasmas

Um grupo de  $n$  Caça-fantasmas está perseguindo  $n$  fantasmas. Cada Caça-fantasma está armado com um pacote de prótons, que dispara um feixe em um fantasma, erradicando-o. Um feixe segue em linha reta e termina quando atinge o fantasma. Os Caça-fantasmas decidem adotar a estratégia a seguir. Eles formarão pares com os fantasmas, constituindo  $n$  pares Caça-fantasmas-fantasma, e depois simultaneamente cada Caça-fantasma irá disparar um feixe em seu fantasma escolhido. Como sabemos, é *muito* perigoso permitir que os feixes se cruzem, e assim os Caça-fantasmas devem escolher pares para os quais nenhum feixe cruze outro feixe.

Suponha que a posição de cada Caça-fantasma e cada fantasma seja um ponto fixo no plano, e que não existam três posições colineares.

- Demonstre que existe uma linha que passa através de um Caça-fantasma e um fantasma, tal que o número de Caça-fantasmas em um lado da linha é igual ao número de fantasmas no mesmo lado. Descreva como encontrar tal linha no tempo  $O(n \lg n)$ .
- Forneça um algoritmo de tempo  $O(n^2 \lg n)$  para emparelhar os Caça-fantasmas com os fantasmas de tal modo que nenhum feixe cruze outro.

### 33-4 Pegando varetas

O professor Charon tem um conjunto de  $n$  varetas, dispostas umas sobre as outras em alguma configuração. Cada vareta é especificada por suas extremidades, e cada extremidade é uma tripla ordenada que dá suas coordenadas  $(x, y, z)$ . Nenhuma vareta é vertical. Ele deseja retirar todas as varetas, uma de cada vez, sujeito à condição de que ele só pode retirar uma vareta se não houver outra vareta sobre ela.

- Forneça um procedimento que tome duas varetas  $a$  e  $b$  e informe se  $a$  está acima, abaixo ou não relacionada a  $b$ .
- Descreva um algoritmo eficiente que determine se é possível retirar todas as varas e, se for o caso, forneça uma seqüência válida de extrações de varetas para fazê-lo.

### 33-5 Distribuições de envoltórias esparsas

Considere o problema de calcular a envoltória convexa de um conjunto de pontos no plano que tenham sido traçados de acordo com alguma distribuição aleatória conhecida. Às vezes, a envoltória convexa de  $n$  pontos traçados a partir de tal distribuição tem o tamanho esperado  $O(n^{1-\epsilon})$

para alguma constante  $\epsilon > 0$ . Chamamos tal distribuição de *envoltórias esparsas*. As distribuições de envoltórias esparsas incluem:

- Pontos traçados de modo uniforme a partir de um disco de raio unitário. A envoltória convexa tem tamanho esperado  $\Theta(n^{1/3})$ .
  - Pontos traçados de modo uniforme a partir do interior de um polígono convexo com  $k$  lados, para qualquer constante  $k$ . A envoltória convexa tem o tamanho esperado  $\Theta(\lg n)$ .
  - Pontos traçados de acordo com uma distribuição bidimensional normal. A envoltória convexa tem o tamanho esperado  $\Theta(\sqrt{\lg n})$ .
- a. Dados dois polígonos convexos com  $n_1$  e  $n_2$  vértices respectivamente, mostre como calcular a envoltória convexa de todos os  $n_1 + n_2$  pontos no tempo  $O(n_1 + n_2)$ . (Os polígonos podem se sobrepor.)
  - b. Mostre que a envoltória convexa de um conjunto de  $n$  pontos traçados independentemente de acordo com uma distribuição de envoltórias esparsas pode ser calculada no tempo esperado  $O(n)$ . (*Sugestão:* Encontre recursivamente as envoltórias convexas dos primeiros  $n/2$  pontos e dos  $n/2$  pontos seguintes, e depois combine os resultados.)

## Notas do capítulo

Este capítulo apenas toca a superfície dos algoritmos e técnicas de geometria computacional. Os livros sobre geometria computacional incluem os de Preparata e Shamos [247], Edelsbrunner [83] e O'Rourke [235].

Embora a geometria tenha sido estudada desde a antigüidade, o desenvolvimento de algoritmos para problemas geométricos é relativamente novo. Preparata e Shamos observam que a primeira noção da complexidade de um problema foi dada por E. Lemoine em 1902. Ele estava estudando construções euclidianas – aquelas que usam uma régua e um compasso – e criou um conjunto de cinco primitivas: colocar uma perna do compasso sobre um dado ponto, colocar uma perna do compasso sobre uma dada linha, desenhar um círculo, passar a borda da régua por um dado ponto e desenhar uma linha. Lemoine estava interessado no número de primitivas necessárias para efetuar uma determinada construção; ele chamou esse valor “simplicidade” da construção.

O algoritmo da Seção 33.2, que determina se quaisquer segmentos se cruzam, se deve a Shamos e Hoey [275].

A versão original da varredura de Graham é dada por Graham [130]. O algoritmo de embrulho de pacote se deve a Jarvis [165]. Usando um modelo de computação de árvore de decisão, Yao [318] provou um limite inferior  $\Omega(n \lg n)$  para o tempo de execução de qualquer algoritmo de envoltórias convexas. Quando o número de vértices  $b$  da envoltória convexa é levado em conta, o algoritmo de podar e pesquisar de Kirkpatrick e Seidel [180], que demora o tempo  $O(n \lg b)$ , é assintoticamente ótimo.

O algoritmo de dividir e conquistar de tempo  $O(n \lg n)$  para encontrar o par de pontos mais próximos foi criado por Shamos e aparece em Preparata e Shamos [247]. Preparata e Shamos também mostram que o algoritmo é assintoticamente ótimo em um modelo de árvore de decisão.

---

## Capítulo 34

# Problemas NP-completos

Quase todos os algoritmos que estudamos até agora foram *algoritmos de tempo polinomial*: sobre entradas de tamanho  $n$ , seu tempo de execução do pior caso é  $O(n^k)$  para alguma constante  $k$ . É natural imaginar se *todos* os problemas podem ser resolvidos em tempo polinomial. A resposta é não. Por exemplo, existem problemas, como o famoso “Problema de parada (halting)” de Turing, que não podem ser resolvidos por qualquer computador, não importa quanto tempo seja fornecido. Também existem problemas que podem ser resolvidos, mas não no tempo  $O(n^k)$  para qualquer constante  $k$ . Em geral, imaginamos problemas que podem ser resolvidos por algoritmo de tempo polinomial como sendo tratáveis, e os problemas que exigem tempo superpolinomial como intratáveis, ou difíceis.

Porém, o assunto deste capítulo é uma classe interessante de problemas, chamados problemas “NP-completos”, cujo status é desconhecido. Ainda não foi descoberto nenhum algoritmo de tempo polinomial para um problema NP-completo, nem ninguém ainda foi capaz de provar que não pode existir nenhum algoritmo de tempo polinomial para qualquer deles. Essa questão chamada  $P \neq NP$  foi um dos mais profundos e surpreendentes problemas de pesquisa abertos na ciência da computação teórica desde que foi proposto pela primeira vez em 1971.

Um aspecto particularmente torturante dos problemas NP-completos é que vários deles parecem a princípio serem semelhantes a problemas que têm algoritmos de tempo polinomial. Em cada um dos pares de problemas a seguir, um deles pode ser resolvido em tempo polinomial e o outro é NP-completo, mas a diferença entre os problemas parece ser tênue:

**Caminhos simples mais curtos e mais longos:** No Capítulo 24, vimos que até mesmo com pesos de arestas negativos, podemos encontrar caminhos *mais curtos* a partir de uma única origem em um grafo orientado  $G = (V, E)$  no tempo  $O(VE)$ . Porém, encontrar o caminho simples *mais longo* entre dois vértices é NP-completo. De fato, ele é NP-completo até mesmo se todos os pesos de arestas forem iguais a 1.

**Viagem de Euler e ciclo hamiltoniano:** Uma *viagem de Euler* de um grafo orientado conectado  $G = (V, E)$  é um ciclo que percorre cada *aresta* de  $G$  exatamente uma vez, embora possa visitar um vértice mais de uma vez. Pelo Problema 22-3, podemos determinar se um grafo tem uma viagem de Euler somente no tempo  $O(E)$  e, de fato, podemos encontrar as arestas da viagem de Euler no tempo  $O(E)$ . Um *ciclo hamiltoniano* de um grafo orientado  $G = (V, E)$  é um ciclo simples que contém cada *vértice* em  $V$ . Determinar se um grafo orientado tem um ciclo hamiltoniano é NP-completo. (Mais adiante neste capítulo, provaremos que determinar se um grafo *não orientado* tem um ciclo hamiltoniano é NP-completo.)

**Satisfabilidade 2-CNF e satisfabilidade 3-CNF:** Uma fórmula booleana contém variáveis cujos valores são 0 ou 1; conectivos booleanos como  $\wedge$  (AND),  $\vee$  (OR) e  $\neg$  (NOT); e parênteses. Uma fórmula booleana é *capaz de satisfação* se existe alguma atribuição dos valores 0 e 1 para suas variáveis que faça com que ela seja avaliada como 1. Definiremos os termos de modo mais formal mais adiante neste capítulo; porém, informalmente, uma fórmula booleana está em *forma normal k-conjuntiva*, ou  $k$ -CNF, se for o AND de cláusulas de OR de exatamente  $k$  variáveis ou suas negações. Por exemplo, a fórmula booleana  $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3)$  está em 2-CNF. (Elas tem a atribuição satisfatória  $x_1 = 1, x_2 = 0, x_3 = 1$ .) Existe um algoritmo de tempo polinomial para determinar se uma fórmula de 2-CNF é capaz de satisfação, mas como veremos mais adiante neste capítulo, determinar se uma fórmula 3-CNF é capaz de satisfação é NP-completo.

## Caráter NP-completo e as classes P e NP

Ao longo deste capítulo, faremos referência a três classes de problemas: P, NP e NPC, sendo essa última classe a de problemas NP-completos. Aqui, vamos descrevê-las de modo informal; mais adiante, vamos defini-las mais formalmente.

A classe P consiste nos problemas que podem ser resolvidos em tempo polinomial. Mais especificamente, são problemas que podem ser resolvidos no tempo  $O(n^k)$  para alguma constante  $k$ , onde  $n$  é o tamanho da entrada para o problema. A maioria dos problemas examinados em capítulos anteriores é da classe P.

A classe NP consiste nos problemas que são “verificáveis” em tempo polinomial. O que queremos dizer nesse caso é que, se tivéssemos de algum modo um “certificado” de uma solução, poderíamos verificar se o certificado é correto em tempo polinomial no tamanho da entrada para o problema. Por exemplo, no problema do ciclo hamiltoniano, dado um grafo orientado  $G = (V, E)$ , um certificado seria uma seqüência  $\langle v_1, v_2, v_3, \dots, v_{|V|} \rangle$  de  $|V|$  vértices. É fácil verificar em tempo polinomial que  $(v_i, v_{i+1}) \in E$  para  $i = 1, 2, 3, \dots, |V| - 1$  e também que  $(v_{|V|}, v_1) \in E$ . Como outro exemplo de satisfabilidade 3-CNF, um certificado seria uma atribuição de valores a variáveis. Podemos verificar facilmente em tempo polinomial que essa atribuição satisfaz à fórmula booleana.

Qualquer problema em P também está em NP, tendo em vista que, se um problema está em P então podemos resolvê-lo em tempo polinomial sem sequer receber um certificado. Vamos formalizar essa noção mais adiante neste capítulo mas, no momento, podemos acreditar que  $P \subseteq NP$ . A questão aberta é se P é ou não um subconjunto próprio de NP.

Informalmente, um problema está na classe NPC – e vamos nos referir a ele como um problema *NP-completo* – se ele está em NP e é tão “difícil” quanto qualquer problema em NP. Definiremos formalmente o que significa ser tão difícil quanto qualquer problema em NP mais adiante neste capítulo. Enquanto isso, vamos declarar sem provar que, se *qualquer* problema NP-completo pode ser resolvido em tempo polinomial, então *todo* problema NP-completo tem um algoritmo de tempo polinomial. A maioria dos teóricos da ciência da computação acredita que os problemas NP-completos são intratáveis pois, dada a ampla faixa de problemas NP-completos que foram estudados até hoje, sem qualquer progresso em direção a uma solução de tempo polinomial, seria verdadeiramente espantoso se todos eles pudessem ser resolvidos em tempo polinomial. Ainda assim, dado o esforço dedicado há tanto tempo para provar que os problemas NP-completos são intratáveis – sem um resultado conclusivo – não podemos eliminar a possibilidade de que os problemas NP-completos possam de fato ser resolvidos em tempo polinomial.

Para se tornar um bom projetista de algoritmos, você deve entender os rudimentos da teoria dos problemas NP-completos. Se puder estabelecer um problema como NP-completo, você fornecerá uma boa evidência de sua intratabilidade. Como um engenheiro, você faria melhor gastando seu tempo no desenvolvimento de um algoritmo de aproximação (ver Capítulo 35) ou resolvendo um caso especial tratável, em vez de procurar por um algoritmo rápido que resolva o problema exatamente. Além disso, muitos problemas naturais e interessantes que na superfície não parecem mais difíceis que a ordenação, a pesquisa de grafos ou o fluxo de rede são de fato NP-completos. Desse modo, é importante se familiarizar com essa classe especial de problemas.

## Visão geral das técnicas para mostrar que um problema é NP-completo

As técnicas que empregamos para mostrar que um determinado problema é NP-completo difere das técnicas usadas ao longo da maior parte deste livro para projetar e analisar algoritmos. Existe uma razão fundamental para tal diferença: mostrando que um problema é NP-completo, estamos fazendo uma declaração sobre a dificuldade para resolver esse problema (ou, pelo menos o quanto o consideramos difícil), não sobre o quanto ele é fácil. Não estamos tentando provar a existência de um algoritmo eficiente, mas sim que provavelmente não existe nenhum algoritmo eficiente. Desse modo, as provas do caráter NP-completo são um pouco parecidas com a prova da Seção 8.1 de um limite inferior de tempo  $\Omega(n \lg n)$  para qualquer algoritmo de ordenação por comparação; contudo, as técnicas específicas usadas para mostrar o caráter NP-completo diferem do método de árvore de decisão usado na Seção 8.1.

Contamos com três conceitos fundamentais para mostrar que um problema é NP-completo:

### Problemas de decisão *versus* problemas de otimização

Muitos problemas de interesse são *problemas de otimização*, em que cada solução possível (isto é, “válida”) tem um valor associado, e desejamos encontrar a solução possível com o melhor valor.

Por exemplo, em um problema que chamamos SHORTEST-PATH, temos um grafo não orientado  $G$  e vértices  $u$  e  $v$ , e desejamos encontrar o caminho de  $u$  até  $v$  que utiliza o menor número de arestas. (Em outras palavras, SHORTEST-PATH é o problema de caminho mais curto de um único par em um grafo não orientado e não ponderado.) Porém, o caráter NP-completo não se aplica diretamente a problemas de otimização, mas a *problemas de decisão*, em que a resposta é simplesmente “sim” ou “não” (ou, de modo mais formal, “1” ou “0”).

Embora mostrar que um problema é NP-completo nos limite ao âmbito de problemas de decisão, existe um relacionamento conveniente entre problemas de otimização e problemas de decisão. Normalmente, podemos formular um determinado problema de otimização como um problema de decisão relacionado impondo um limite sobre o valor a ser otimizado. Por exemplo, no caso de SHORTEST-PATH, um problema de decisão relacionado, que chamamos PATH, é se, dado um grafo orientado  $G$ , vértices  $u$  e  $v$ , e um inteiro  $k$ , existe um caminho de  $u$  até  $v$  consistindo em no máximo  $k$  arestas.

O relacionamento entre um problema de otimização e seu problema de decisão relacionado atua a nosso favor quando tentamos mostrar que o problema de otimização é “difícil”. Isso ocorre porque o problema de decisão é de certo modo “mais fácil” ou, pelo menos, “não mais difícil”. Como um exemplo específico, podemos resolver PATH resolvendo SHORTEST-PATH, e depois comparando o número de arestas no caminho mais curto encontrado ao valor do parâmetro de problema de decisão  $k$ . Em outras palavras, se um problema de otimização é fácil, seu problema de decisão relacionado também é fácil. Declarado de um modo que tem maior relevância para o caráter NP-completo, se pudermos fornecer evidências de que um problema de decisão é difícil, também forneceremos evidências de que seu problema de otimização relacionado é difícil. Desse modo, embora restrinja a atenção a problemas de decisão, a teoria de problemas NP-completos freqüentemente também tem implicações para problemas de otimização.

### Reduções

A noção anterior de mostrar que um problema não é mais difícil ou não é mais fácil que outro se aplica até mesmo quando ambos os problemas são problemas de decisão. tiramos proveito dessa idéia em quase toda prova do caráter NP-completo, como a seguir. Vamos considerar um problema de decisão, digamos  $A$ , que gostaríamos de resolver em tempo polinomial. Chamamos a entrada para um determinado problema de *instância* desse problema; por exemplo, em PATH, uma instância seria um dado grafo  $G$ , vértices específicos  $u$  e  $v$  de  $G$ , e um certo inteiro  $k$ . Agora, suponha que exista um problema de decisão diferente, digamos  $B$ , que já sabemos como resol-

ver em tempo polinomial. Finalmente, suponha que temos um procedimento que transforma qualquer instância  $\alpha$  de  $A$  em alguma instância  $\beta$  de  $B$  com as seguintes características:

1. A transformação demora tempo polinomial.
2. As respostas são as mesmas. Isto é, a resposta para  $\alpha$  é “sim” se e somente se a resposta para  $\beta$  também é “sim”.

Chamamos tal procedimento um **algoritmo de redução** de tempo polinomial e, como mostra a Figura 34.1, ele nos oferece um meio para resolver o problema  $A$  em tempo polinomial:

1. Dada uma instância  $\alpha$  do problema  $A$ , use um algoritmo de redução de tempo polinomial para transformá-la em uma instância  $\beta$  do problema  $B$ .
2. Execute o algoritmo de decisão de tempo polinomial para  $B$  sobre a instância  $\beta$ .
3. Use a resposta de  $\beta$  como a resposta para  $\alpha$ .

Como cada uma dessas etapas demora tempo polinomial, as três juntas também demoram um tempo polinomial, e assim temos um modo de decidir sobre  $\alpha$  em tempo polinomial. Em outras palavras, “reduzindo” a solução do problema  $A$  à solução do problema  $B$ , usamos a “facilidade” de  $B$  para provar a “facilidade” de  $A$ .

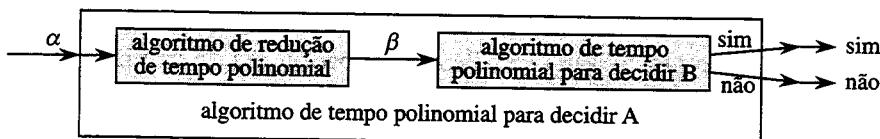


FIGURA 34.1 O uso de um algoritmo de redução de tempo polinomial para resolver um problema de decisão  $A$  em tempo polinomial, dado um algoritmo de decisão de tempo polinomial para outro problema  $B$ . Em tempo polinomial, transformamos uma instância  $\alpha$  de  $A$  em uma instância  $\beta$  de  $B$ , resolvemos  $B$  em tempo polinomial e usamos a resposta de  $\beta$  como a resposta para  $\alpha$

Lembrando que o caráter NP-completo consiste em mostrar o quanto um problema é difícil, em vez de mostrar o quanto ele é fácil, usamos reduções de tempo polinomial da maneira oposta para mostrar que um problema é NP-completo. Vamos levar a idéia um passo adiante, e mostrar como poderíamos usar reduções de tempo polinomial para demonstrar que não pode existir nenhum algoritmo de tempo polinomial para um determinado problema  $B$ . Suponha que temos um problema de decisão  $A$  para o qual já sabemos que não pode existir nenhum algoritmo de tempo polinomial. (Não vamos nos preocupar por enquanto com a maneira de encontrar tal problema  $A$ .) Suponha ainda que temos uma redução de tempo polinomial transformando instâncias de  $A$  em instâncias de  $B$ . Agora, podemos usar uma prova simples por contradição para mostrar que não é possível existir nenhum algoritmo de tempo polinomial para  $B$ . Por outro lado, vamos supor que  $B$  tenha um algoritmo de tempo polinomial. Então, usando o método mostrado na Figura 34.1 teríamos um modo de resolver o problema  $A$  em tempo polinomial, o que contradiz nossa hipótese de que não existe nenhum algoritmo de tempo polinomial para  $A$ .

No caso do caráter NP-completo, não podemos supor que não exista absolutamente nenhum algoritmo de tempo polinomial para o problema  $A$ . Porém, a metodologia da prova é semelhante, no sentido de que provamos que o problema  $B$  é NP-completo na hipótese de que o problema  $A$  também seja NP-completo.

### Um primeiro problema NP-completo

Como a técnica de redução se baseia em ter um problema já reconhecido como NP-completo para provar que um problema diferente é NP-completo, precisamos de um “primeiro” problema NP-completo. O problema que usaremos é o da satisfatibilidade de circuitos, no qual temos um

círculo combinacional booleano composto por portas AND, OR e NOT, e desejamos saber se existe algum conjunto de entradas booleanas para esse círculo que faça sua saída ser 1. Provaremos que esse primeiro problema é NP-completo na Seção 34.3.

## Esboço do capítulo

Este capítulo estuda os aspectos dos problemas NP-completos que surgem mais diretamente na análise de algoritmos. Na Seção 34.1, formalizamos nossa noção de “problema” e definimos a classe de complexidade P de problemas de decisão que podem ser resolvidos em tempo polinomial. Também veremos como essas noções se encaixam na estrutura da teoria da linguagem formal. A Seção 34.2 define a classe NP de problemas de decisão cujas soluções podem ser verificadas em tempo polinomial. Ela também propõe formalmente a questão de  $P \neq NP$ .

A Seção 34.3 mostra como os relacionamentos entre problemas podem ser estudados por meio de “reduções” de tempo polinomial. Ela define os problemas NP-completos e esboça uma prova de que um problema, chamado “satisfabilidade de circuitos”, é NP-completo. Tendo encontrado um problema NP-completo, mostramos na Seção 34.4 como é possível provar que outros problemas são NP-completos, de modo muito mais simples, pela metodologia de reduções. A metodologia é ilustrada, mostrando que dois problemas de satisfabilidade de fórmulas são NP-completos. Na Seção 34.5, mostramos que vários outros problemas também são NP-completos.

## 34.1 Tempo polinomial

Começamos nosso estudo de problemas NP-completos formalizando nossa noção de problemas que podem ser resolvidos em tempo polinomial. Em geral, esses problemas são considerados tratáveis, mas por razões filosóficas, e não matemáticas. Podemos oferecer três argumentos sustentáveis.

Primeiro, embora seja razoável considerar um problema que exige o tempo  $\Theta(n^{100})$  intratável, existem bem poucos problemas práticos que exigem tempo na ordem de um polinômio de tão alto grau. Os problemas calculáveis em tempo polinomial encontrados na prática normalmente exigem muito menos tempo. A experiência mostrou que, uma vez que um algoritmo de tempo polinomial para um problema é descoberto, algoritmos mais eficientes freqüentemente o seguem. Ainda que o melhor algoritmo atual para um problema tenha um tempo de execução igual a  $\Theta(n^{100})$ , é provável que um algoritmo com um tempo de execução muito melhor logo seja descoberto.

Em segundo lugar, para muitos modelos razoáveis de computação, um problema que pode ser resolvido em tempo polinomial em um modelo pode ser resolvido em tempo polinomial em outro modelo. Por exemplo, a classe de problemas que podem ser resolvidos em tempo polinomial pela máquina de acesso aleatório serial usada na maior parte deste livro é igual à classe de problemas que podem ser resolvidos em tempo polinomial em máquinas abstratas de Turing.<sup>1</sup> Ela também é igual à classe de problemas que podem ser resolvidos em tempo polinomial em um computador paralelo, quando o número de processadores cresce polinomialmente com o tamanho da entrada.

Terceiro, a classe de problemas de tempo polinomial que podem ser resolvidos tem propriedades de fechamento interessantes, pois os polinômios são fechados sob a adição, a multiplicação e a composição. Por exemplo, se a saída de um algoritmo de tempo polinomial é alimentada na entrada de outro, o algoritmo composto é polinomial. Caso contrário, se o algoritmo de tempo polinomial faz um número constante de chamadas a sub-rotinas de tempo polinomial, o tempo de execução do algoritmo composto é polinomial.

---

<sup>1</sup>Ver Hopcroft e Ullman [156] ou Lewis e Papadimitriou [204] para examinar um tratamento completo do modelo da máquina de Turing.

## Problemas abstratos

Para entender a classe de problemas de tempo polinomial que podem ser resolvidos, primeiro devemos ter uma noção formal do que é um “problema”. Definimos um **problema abstrato**  $Q$  como uma relação binária sobre um conjunto  $I$  de **instâncias** de problemas e um conjunto  $S$  de **soluções** de problemas. Por exemplo, uma instância de SHORTEST-PATH é uma tripla que consiste em um grafo e dois vértices. Uma solução é uma seqüência de vértices no grafo, talvez com a seqüência vazia denotando que não existe nenhum caminho. O problema SHORTEST-PATH em si é a relação que associa cada instância de um grafo e dois vértices com um caminho mais curto no grafo que conecta os dois vértices. Tendo em vista que caminhos mais curtos não são necessariamente únicos, uma dada instância de problema pode ter mais de uma solução.

Essa formulação de um problema abstrato é mais geral que o necessário para nossos propósitos. Como vimos antes, a teoria de problemas NP-completos restringe a atenção a **problemas de decisão**: aqueles que têm uma solução sim/não. Nesse caso, podemos ver um problema de decisão abstrato como uma função que mapeia o conjunto de instâncias  $I$  para o conjunto solução  $\{0, 1\}$ . Por exemplo, um problema de decisão relacionado a SHORTEST-PATH é o problema PATH que vimos antes. Se  $i = G, u, v, k$  é uma instância do problema de decisão PATH, então  $\text{PATH}(i) = 1$  (sim) se um caminho mais curto de  $u$  até  $v$  tem no máximo  $k$  arestas, e  $\text{PATH}(i) = 0$  (não) em caso contrário. Muitos problemas abstratos não são problemas de decisão, mas sim **problemas de otimização**, nos quais algum valor deve ser minimizado ou maximizado. Porém, como vimos anteriormente, em geral é simples reformular um problema de otimização como um problema de decisão não mais difícil que o primeiro.

## Codificações

Se um programa de computador deve resolver um problema abstrato, as instâncias de problemas devem ser representadas de um modo que o programa reconheça. Uma **codificação** de um conjunto  $S$  de objetos abstratos é um mapeamento  $e$  de  $S$  para o conjunto de cadeias binárias.<sup>2</sup> Por exemplo, estamos todos familiarizados com a codificação dos números naturais  $N = \{0, 1, 2, 3, 4, \dots\}$  como as cadeias  $\{0, 1, 10, 11, 100, \dots\}$ . Usando essa codificação,  $e(17) = 10001$ . Qualquer pessoa que tenha observado no computador representações de caracteres do teclado está familiarizado com os códigos ASCII ou EBCDIC. No código ASCII, a codificação de A é 1000001. Mesmo um objeto composto pode ser codificado como uma cadeia binária pela combinação das representações de suas partes constituintes. Polígonos, grafos, funções, pares ordenados, programas – todos podem ser codificados como cadeias binárias.

Desse modo, um algoritmo de computador que “resolve” algum problema de decisão abstrato na realidade toma uma codificação de uma instância de problema como entrada. Chamamos um problema cujo conjunto de instâncias é o conjunto de cadeias binárias um **problema concreto**. Dizemos que um algoritmo **resolve** um problema concreto no tempo  $O(T(n))$  se, quando é fornecida uma instância de problema  $i$  de comprimento  $n = |i|$ , o algoritmo pode produzir a solução no tempo máximo  $O(T(n))$ .<sup>3</sup> Então, um problema concreto **pode ser resolvido em tempo polinomial** se existe um algoritmo para resolvê-lo no tempo  $O(n^k)$  para alguma constante  $k$ .

Agora podemos definir formalmente a **classe de complexidade P** como o conjunto de problemas de decisão concretos que podem ser resolvidos em tempo polinomial.

<sup>2</sup>O contradomínio de  $e$  não precisa ser formado por cadeias *binárias*; qualquer conjunto de cadeias sobre um alfabeto finito que tenha pelo menos dois símbolos servirá.

<sup>3</sup>Supomos que a saída do algoritmo está isolada de sua entrada. Como demora pelo menos uma etapa para produzir cada bit da saída e existem  $O(T(n))$  etapas de tempo, o tamanho da saída é  $O(T(n))$ .

Podemos usar codificações para mapear problemas abstratos em problemas concretos. Dado um problema de decisão abstrato  $Q$  que mapeia um conjunto de instâncias  $I$  para  $\{0, 1\}$ , uma codificação  $e : I \rightarrow \{0, 1\}^*$  pode ser usada para induzir um problema de decisão concreto relacionado, o qual denotamos por  $e(Q)$ .<sup>4</sup> Se a solução para uma instância de problema abstrato  $i \in I$  é  $Q(i) \in \{0, 1\}$ , então a solução para a instância de problema concreto  $e(i) \in \{0, 1\}^*$  também é  $Q(i)$ . Como um detalhe técnico, podem existir algumas cadeias binárias que não representam nenhuma instância significativa de problema abstrato. Por conveniência, iremos supor que qualquer cadeia desse tipo será mapeada arbitrariamente como 0. Desse modo, o problema concreto produz as mesmas soluções que o problema abstrato sobre instâncias de cadeias binárias que representam as codificações de instâncias do problema abstrato.

Gostaríamos de estender a definição de possibilidade de solução em tempo polinomial de problemas concretos para problemas abstratos, usando codificações como ponte, mas gostaríamos também que a definição fosse independente de qualquer codificação específica. Isto é, a eficiência da solução de um problema não deve depender do modo como o problema está codificado. Infelizmente, ela depende bastante disso. Por exemplo, suponha que um inteiro  $k$  seja fornecido como a única entrada para um algoritmo, e suponha que o tempo de execução do algoritmo seja  $\Theta(k)$ . Se o inteiro  $k$  é fornecido em *unário* – uma cadeia de  $k$  valores 1 – então o tempo de execução do algoritmo é  $O(n)$  sobre entradas de comprimento  $n$ , que é tempo polinomial. Porém, se usarmos a representação binária mais natural do inteiro  $k$ , então o comprimento da entrada será  $n = \lfloor \lg k \rfloor$ . Nesse caso, o tempo de execução do algoritmo será  $\Theta(k) = \Theta(2^n)$ , que é exponencial no tamanho da entrada. Assim, dependendo da codificação, o algoritmo será executado em tempo polinomial ou superpolinomial.

A codificação de um problema abstrato é então muito importante para nossa compreensão do tempo polinomial. Na realidade, não podemos falar sobre a resolução de um problema abstrato sem primeiro especificar uma codificação. Apesar disso, na prática, se eliminarmos codificações “dispendiosas” como as unárias, a codificação real de um problema fará pouca diferença na hora de decidir se o problema pode ser resolvido em tempo polinomial. Por exemplo, a representação de inteiros na base 3 em vez de binário não tem nenhum efeito sobre o fato de um problema poder ser resolvido em tempo polinomial, pois um inteiro representado na base 3 pode ser convertido em um inteiro representado na base 2 em tempo polinomial.

Dizemos que uma função  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  é **calculável em tempo polinomial** se existe um algoritmo de tempo polinomial  $A$  que, dada qualquer entrada  $x \in \{0, 1\}^*$ , produz como saída  $f(x)$ . Para algum conjunto  $I$  de instâncias de problemas, dizemos que duas codificações  $e_1$  e  $e_2$  são **polinomialmente relacionadas** se existem duas funções calculáveis em tempo polinomial  $f_{12}$  e  $f_{21}$  tais que, para qualquer  $i \in I$ , temos  $f_{12}(e_1(i)) = e_2(i)$  e  $f_{21}(e_2(i)) = e_1(i)$ .<sup>5</sup> Isto é, a codificação  $e_2(i)$  pode ser calculada a partir da codificação  $e_1(i)$  por um algoritmo de tempo polinomial, e vice-versa. Se duas codificações  $e_1$  e  $e_2$  de um problema abstrato são polinomialmente relacionadas, o fato de um problema poder ou não ser resolvido em tempo polinomial é independente da codificação que usamos, como mostra o lema a seguir.

### Lema 34.1

Seja  $Q$  um problema de decisão abstrato em um conjunto de instâncias  $I$ , e sejam  $e_1$  e  $e_2$  codificações polinomialmente relacionadas em  $I$ . Então,  $e_1(Q) \in P$  se e somente se  $e_2(Q) \in P$ .

<sup>4</sup> Como veremos em breve,  $\{0, 1\}^*$  denota o conjunto de todas as cadeias compostas de símbolos do conjunto  $\{0, 1\}$ .

<sup>5</sup> Tecnicamente, também exigimos que as funções  $f_{12}$  e  $f_{21}$  “mapeiam não instâncias para não instâncias”. Uma **não instância** de uma codificação  $e$  é uma cadeia  $x \in \{0, 1\}^*$  tal que não existe nenhuma instância  $i$  para a qual  $e(i) = x$ . Exigimos que  $f_{12}(x) = y$  para toda não instância  $x$  da codificação  $e_1$ , onde  $y$  é alguma não instância de  $e_2$ , e que  $f_{21}(x') = y'$  para toda não instância  $x'$  de  $e_2$ , onde  $y'$  é alguma não instância de  $e_1$ .

**Prova** Só precisamos provar o sentido direto, pois o sentido inverso é simétrico. Então, suponha que  $e_1(Q)$  pode ser resolvida no tempo  $O(n^k)$  para alguma constante  $k$ . Além disso, suponha que, para qualquer instância de problema  $i$ , a codificação  $e_1(i)$  pode ser calculada a partir da codificação  $e_2(i)$  no tempo  $O(n^c)$  para alguma constante  $c$ , onde  $n = |e_2(i)|$ . Para resolver o problema  $e_2(Q)$ , sobre a entrada  $e_2(i)$ , primeiro calculamos  $e_1(i)$ , e depois executamos o algoritmo para  $e_1(Q)$  sobre  $e_1(i)$ . Quanto tempo isso demora? A conversão de codificações demora o tempo  $O(n^c)$ , e então  $|e_2(i)| = O(n^c)$ , pois a saída de um computador serial não pode ser mais longa que seu tempo de execução. Resolver o problema em  $e_1(i)$  demora o tempo  $O(|e_2(i)|k) = O(n^{ck})$ , que é polinomial, pois tanto  $c$  quanto  $k$  são constantes.

Desse modo, o fato de um problema abstrato ter suas instâncias codificadas em binário ou em base 3, não afeta sua “complexidade”, isto é, o fato de ele poder ser resolvido em tempo polinomial ou não; porém, se as instâncias forem codificadas em unário, sua complexidade pode mudar. Para ser capaz de converter de uma forma independente da codificação, geralmente iremos supor que instâncias de problemas estão codificadas em qualquer forma razoável e concisa, a menos que seja dito especificamente algo em contrário. Para sermos exatos, vamos supor que a codificação de um inteiro esteja polinomialmente relacionada à sua representação binária, e que a codificação de um conjunto finito esteja polinomialmente relacionada à sua codificação como uma lista de seus elementos, colocados entre chaves e separados por vírgulas. (ASCII é um desses esquemas de codificação.) Com tal codificação “padrão” em mãos, podemos derivar codificações razoáveis de outros objetos matemáticos, como tuplas, grafos e fórmulas. Para denotar a codificação padrão de um objeto, colocaremos o objeto entre colchetes angulares. Desse modo,  $\langle G \rangle$  denota a codificação padrão de um grafo  $G$ .

Desde que utilizemos de forma implícita uma codificação que esteja polinomialmente relacionada a essa codificação padrão, podemos falar diretamente sobre problemas abstratos sem fazer referência a qualquer codificação particular, sabendo que a escolha da codificação não tem nenhum efeito sobre o fato do problema abstrato poder ser resolvido em tempo polinomial. Daí em diante, em geral iremos supor que todas as instâncias de problemas são cadeias binárias codificadas com o uso da codificação padrão, a menos que especifiquemos o contrário de forma explícita. Normalmente, também iremos negligenciar a distinção entre problemas abstratos e concretos. Contudo, o leitor deve ficar atento aos problemas que surgem na prática, nos quais uma codificação padrão não é óbvia e a codificação faz diferença.

## Uma estrutura de linguagem formal

Um dos aspectos convenientes de se concentrar em problemas de decisão é que eles tornam fácil o uso do mecanismo da teoria de linguagem formal. Nesse momento, vale a pena rever algumas definições dessa teoria. Um **alfabeto**  $\Sigma$  é um conjunto finito de símbolos. Uma **linguagem**  $L$  sobre  $\Sigma$  é qualquer conjunto de cadeias formadas por símbolos de  $\Sigma$ . Por exemplo, se  $\Sigma = \{0, 1\}$ , o conjunto  $L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$  é a linguagem de representações binárias de números primos. Denotamos a cadeia **vazia** por  $\varepsilon$ , e a **linguagem vazia** por  $\emptyset$ . A linguagem de todas as cadeias sobre  $\Sigma$  é denotada por  $\Sigma^*$ . Por exemplo, se  $\Sigma = \{0, 1\}$ , então  $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$  é o conjunto de todas as cadeias binárias. Toda linguagem  $L$  sobre  $\Sigma$  é um subconjunto de  $\Sigma^*$ .

Existe uma variedade de operações sobre linguagens. Operações da teoria dos conjuntos, como **união** e **interseção**, decorrem diretamente das definições da teoria dos conjuntos. Definimos o **complemento** de  $L$  por  $\bar{L} = \Sigma^* - L$ . A **concatenação** de duas linguagens  $L_1$  e  $L_2$  é a linguagem

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ e } x_2 \in L_2\}.$$

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots,$$

onde  $L^k$  é a linguagem obtida pela concatenação de  $L$  com ela própria  $k$  vezes.

Do ponto de vista da teoria da linguagem, o conjunto de instâncias para qualquer problema de decisão  $Q$  é simplesmente o conjunto  $\Sigma^*$ , onde  $\Sigma = \{0, 1\}$ . Tendo em vista que  $Q$  é completamente caracterizado pelas instâncias de problema que produzem uma resposta 1 (sim), podemos ver  $Q$  como uma linguagem  $L$  sobre  $\Sigma = \{0, 1\}$ , onde

$$L = \{x \in \Sigma^* : Q(x) = 1\}.$$

Por exemplo, o problema de decisão PATH tem a linguagem correspondente

$$\begin{aligned} \text{PATH} = & \{\langle G, u, v, k \rangle : G = (V, E) \text{ é um grafo não orientado}, \\ & u, v \in V, \\ & k \geq 0 \text{ é um inteiro, e} \\ & \text{existe um caminho de } u \text{ até } v \text{ em } G \\ & \text{que consiste em no máximo } k \text{ arestas}\}. \end{aligned}$$

(Onde for conveniente, às vezes usaremos o mesmo nome – PATH nesse caso – para fazer referência a um problema de decisão e à sua linguagem correspondente.)

A estrutura da linguagem formal permite que se expresse a relação entre problemas de decisão e algoritmos que os resolvem de forma concisa. Dizemos que um algoritmo  $A$  **aceita** uma cadeia  $x \in \{0, 1\}^*$  se, dada a entrada  $x$ , a saída do algoritmo  $A(x)$  é 1. A linguagem **aceita** por um algoritmo  $A$  é o conjunto  $L = \{x \in \{0, 1\}^* : A(x) = 1\}$ , isto é, o conjunto de cadeias que o algoritmo aceita. Um algoritmo  $A$  **rejeita** uma cadeia  $x$  se  $A(x) = 0$ .

Ainda que a linguagem  $L$  seja aceita por um algoritmo  $A$ , o algoritmo não rejeitará necessariamente uma cadeia  $x \notin L$  fornecida como entrada para ele. Por exemplo, o algoritmo pode entrar em loop infinito. Uma linguagem  $L$  é **decidida** por um algoritmo  $A$ , se toda cadeia binária em  $L$  é aceita por  $A$  e toda cadeia binária não pertencente a  $L$  é rejeitada por  $A$ . Uma linguagem  $L$  é **aceita em tempo polinomial** por um algoritmo  $A$  se ela é aceita por  $A$  e se, além disso, existe uma constante  $k$  tal que, para qualquer cadeia de comprimento  $n$   $x \in L$ , o algoritmo  $A$  aceita  $x$  no tempo  $O(n^k)$ . Uma linguagem  $L$  é **decidida em tempo polinomial** por um algoritmo  $A$  se existe uma constante  $k$  tal que, para qualquer cadeia de comprimento  $n$   $x \in \{0, 1\}^*$ , o algoritmo decide corretamente se  $x \in L$  no tempo  $O(n^k)$ . Desse modo, para aceitar uma linguagem, um algoritmo só precisa se preocupar com as cadeias em  $L$  mas, para decidir uma linguagem, ele deve aceitar ou rejeitar corretamente toda cadeia em  $\{0, 1\}^*$ .

Como exemplo, a linguagem PATH pode ser aceita em tempo polinomial. Um algoritmo de aceitação de tempo polinomial verifica se  $G$  codifica um grafo não orientado, verifica  $u$  e  $v$  são vértices em  $G$ , usa a busca em largura para calcular o caminho mais curto de  $u$  até  $v$  em  $G$ , e depois compara o número de arestas no caminho mais curto obtido com  $k$ . Se  $G$  codifica um grafo não orientado e o caminho de  $u$  até  $v$  tem no máximo  $k$  arestas, o algoritmo dá saída a 1 e pára. Caso contrário, o algoritmo continua a ser executado para sempre. Porém, esse algoritmo não decide PATH, pois ele não dá saída a 0 explicitamente para instâncias em que o caminho mais curto tem mais de  $k$  arestas. Um algoritmo de decisão para PATH deve rejeitar explicitamente cadeias binárias que não pertençam a PATH. Para um algoritmo de decisão como PATH, tal algoritmo de decisão é fácil de projetar: em vez de executar para sempre quando não há um caminho de  $u$  até  $v$  com no máximo  $k$  arestas, ele dá saída a 0 e pára. No caso de outros problemas, como o Problema de parada de Turing, existe um algoritmo de aceitação, mas nenhum algoritmo de decisão.

Podemos definir informalmente uma **classe de complexidade** como um conjunto de linguagens, cuja pertinência é determinada por uma **medida de complexidade**, como o tempo

de execução, de um algoritmo que determina se um dada cadeia  $x$  pertence à linguagem  $L$ . A definição real de uma classe de complexidade é um pouco mais técnica – o leitor interessado deve consultar o importante artigo de Hartmanis e Stearns [140].

Usando essa estrutura de teoria da linguagem, podemos fornecer uma definição alternativa da classe de complexidade P:

$$P = \{L, \subseteq \{0, 1\}^* : \text{existe um algoritmo } A \text{ que decide } L \\ \text{em tempo polinomial}\}.$$

De fato, P também é a classe de linguagens que podem ser aceitas em tempo polinomial.

### **Teorema 34.2**

$$P = \{L : L \text{ é aceita por um algoritmo de tempo polinomial}\}.$$

**Prova** Tendo em vista que a classe de linguagens decididas por algoritmos de tempo polinomial é um subconjunto da classe de linguagens aceitas por algoritmos de tempo polinomial, só precisamos mostrar que, se  $L$  é aceita por um algoritmo de tempo polinomial, ela é decidida por um algoritmo de tempo polinomial. Seja  $L$  a linguagem aceita por algum algoritmo de tempo polinomial  $A$ . Usaremos um argumento clássico de “simulação” para construir outro algoritmo de tempo polinomial  $A'$  que decide  $L$ . Como  $A$  aceita  $L$  no tempo  $O(n^k)$  para alguma constante  $k$ , também existe uma constante  $c$  tal que  $A$  aceita  $L$  em no máximo  $T = cn^k$  passos. Para qualquer cadeia de entrada  $x$ , o algoritmo  $A'$  simula a ação de  $A$  para o tempo  $T$ . Ao final do tempo  $T$ , o algoritmo  $A'$  examina o comportamento de  $A$ . Se  $A$  aceitou  $x$ , então  $A'$  aceita  $x$ , dando saída a 1. Se  $A$  não aceitou  $x$ , então  $A'$  rejeita  $x$ , dando saída a 0. A sobrecarga de  $A'$  simular  $A$  não aumenta o tempo de execução em mais de um fator polinomial, e assim  $A'$  é um algoritmo de tempo polinomial que decide  $L$ . ■

Observe que a prova do Teorema 34.2 é não construtiva. Para uma dada linguagem  $L \in P$ , podemos não conhecer realmente um limite sobre o tempo de execução para o algoritmo  $A$  que aceita  $L$ . Apesar disso, sabemos que tal limite existe, e então, que existe um algoritmo  $A'$  que pode verificar o limite, embora possamos não ser capazes de encontrar o algoritmo  $A'$  com facilidade.

## **Exercícios**

### **34.1-1**

Defina o problema de otimização LONGEST-PATH-LENGTH como a relação que associa cada instância de um grafo não orientado e dois vértices com o número de arestas no caminho simples mais longo entre os dois vértices. Defina o problema de decisão LONGEST-PATH =  $\{\langle G, u, v, k \rangle : G = (V, E) \text{ é um grafo não orientado}, u, v \in V, k \geq 0 \text{ é um inteiro, e existe um caminho simples desde } u \text{ até } v \text{ em } G \text{ que consiste em pelo menos } k \text{ arestas}\}$ . Mostre que o problema de otimização LONGEST-PATH-LENGTH pode ser resolvido em tempo polinomial se e somente se LONGEST-PATH  $\in P$ .

### **34.1-2**

Dê uma definição formal para o problema de encontrar o ciclo simples mais longo em um grafo não orientado. Forneça um problema de decisão relacionado. Forneça a linguagem correspondente ao problema de decisão.

### **34.1-3**

Forneça uma codificação formal de grafos orientados como cadeias binárias, usando uma representação de matriz de adjacências. Faça o mesmo usando uma representação de lista de adjacências. Demonstre que as duas representações estão polinomialmente relacionadas.

#### 34.1-4

O algoritmo de programação dinâmica para o problema da mochila 0-1 que é apresentado no Exercício 16.2-2 é um algoritmo de tempo polinomial? Explique sua resposta.

#### 34.1-5

Mostre que um algoritmo que efetua no máximo um número constante de chamadas a sub-rotinas de tempo polinomial é executado em tempo polinomial, mas que um número polinomial de chamadas a sub-rotinas de tempo polinomial pode resultar em um algoritmo de tempo exponencial.

#### 34.1-6

Mostre que a classe P, vista como um conjunto de linguagens, é fechada sob a união, a interseção, a concatenação, o complemento e a estrela de Kleene. Isto é, se  $L_1, L_2 \in P$ , então  $L_1 \cup L_2 \in P$  etc.

## 34.2 Verificação de tempo polinomial

Agora, examinaremos algoritmos que “verificam” a pertinência em linguagens. Por exemplo, suponha que, para uma dada instância  $\langle G, u, v, k \rangle$  do problema de decisão PATH, também recebemos um caminho  $p$  desde  $u$  até  $v$ . Podemos verificar facilmente se o comprimento de  $p$  é no máximo  $k$  e, se for, podemos visualizar  $p$  como um “certificado” de que a instância de fato pertence a PATH. Para o problema de decisão PATH, esse certificado não parece nos importar muito. Afinal, PATH pertence a P – de fato, PATH pode ser resolvido em tempo linear – e assim, a verificação da pertinência de um dado certificado demora tanto tempo quanto a resolução do problema desde o início. Examinaremos agora um problema para o qual ainda não sabemos de nenhum algoritmo de decisão de tempo polinomial; dado um certificado, a verificação é fácil.

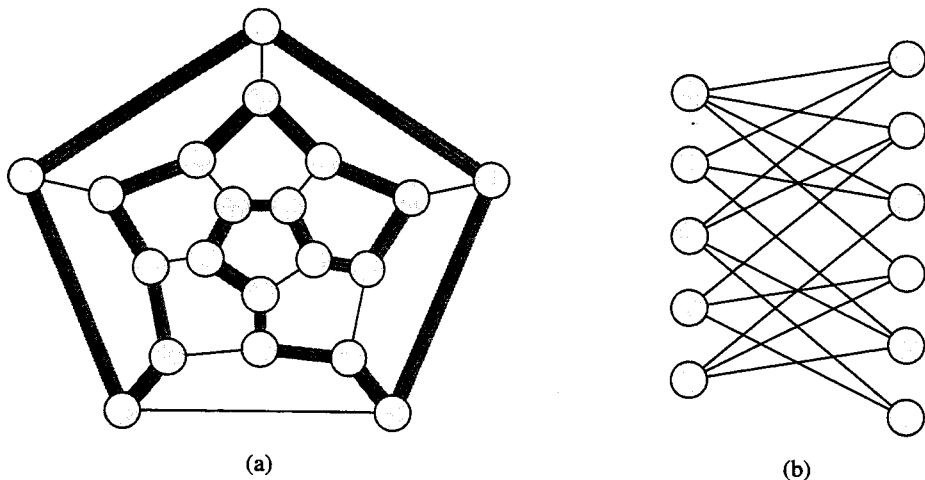


FIGURA 34.2 (a) Um grafo representando os vértices, as arestas e as faces de um dodecaedro, com um ciclo hamiltoniano mostrado por arestas sombreadas. (b) Um grafo bipartido com um número ímpar de vértices. Qualquer grafo desse tipo é não hamiltoniano

### Ciclos hamiltonianos

O problema de encontrar um ciclo hamiltoniano em um grafo não orientado foi estudado por mais de cem anos. Formalmente, um **ciclo hamiltoniano** de um grafo não orientado  $G = (V, E)$  é um ciclo simples que contém cada vértice em  $V$ . Um grafo que contém um ciclo hamiltoniano é chamado **hamiltoniano**; caso contrário, ele é **não hamiltoniano**. Bondy e Murty [45] citam uma carta de W. R. Hamilton descrevendo um jogo matemático sobre o dodecaedro (Figura 34.2(a)) no qual um jogador pega cinco alfinetes em cinco vértices consecutivos quaisquer e o outro jogador deve completar o caminho para formar um ciclo contendo todos os vértices. O

dodecaedro é hamiltoniano, e a Figura 34.2(a) mostra um ciclo hamiltoniano. Porém, nem todos os grafos são hamiltonianos. Por exemplo, a Figura 34.2(b) mostra um grafo bipartido com um número ímpar de vértices. (O Exercício 34.2-2 lhe pede para mostrar que todos esses grafos são não hamiltonianos.)

Podemos definir o **problema do ciclo hamiltoniano**, “Um grafo  $G$  tem um ciclo hamiltoniano?”, como uma linguagem formal:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ é um grafo hamiltoniano}\}.$$

Como poderia um algoritmo decidir a linguagem HAM-CYCLE? Dada uma instância de problema  $\langle G \rangle$ , um algoritmo de decisão possível lista todas as permutações dos vértices de  $G$ , e depois verifica cada permutação para ver se ela é um caminho hamiltoniano. Qual é o tempo de execução desse algoritmo? Se usarmos a codificação “razoável” de um grafo como sua matriz de adjacências, o número  $m$  de vértices no grafo será  $\Omega(\sqrt{n})$ , onde  $n = |\langle G \rangle|$  é o comprimento da codificação de  $G$ . Existem  $m!$  permutações possíveis dos vértices, e portanto o tempo de execução é  $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$ , que não é  $O(n^k)$  para qualquer constante  $k$ . Desse modo, esse algoritmo simples não é executado em tempo polinomial e, de fato, o problema do ciclo hamiltoniano é NP-completo, como provaremos na Seção 34.5.

## Algoritmos de verificação

Considere um problema ligeiramente mais fácil. Suponha que um amigo lhe diga que um dado grafo  $G$  é hamiltoniano, e depois se ofereça para provar isso dando a você os vértices em ordem ao longo do ciclo hamiltoniano. Certamente seria bastante fácil verificar a prova: basta confirmar que o ciclo fornecido é hamiltoniano, conferindo se ele é uma permutação dos vértices de  $V$  e se cada uma das arestas consecutivas ao longo do ciclo existe realmente no grafo. Esse algoritmo de verificação certamente pode ser implementado para execução no tempo  $O(n^2)$ , onde  $n$  é o comprimento da codificação de  $G$ . Portanto, uma prova de que um ciclo hamiltoniano existe em um grafo pode ser verificada em tempo polinomial.

Definimos um **algoritmo de verificação** como sendo um algoritmo de dois argumentos  $A$ , onde um argumento é uma cadeia de entrada comum  $x$  e o outro é uma cadeia binária  $y$  chamada **certificado**. Um algoritmo  $A$  de dois argumentos **verifica** uma cadeia de entrada  $x$  se existe um certificado  $y$  tal que  $A(x, y) = 1$ . A **linguagem verificada** por um algoritmo de verificação  $A$  é

$$L = \{x \in \{0, 1\}^* : \text{existe } y \in \{0, 1\}^* \text{ tal que } A(x, y) = 1\}.$$

Intuitivamente, um algoritmo  $A$  verifica uma linguagem  $L$  se, para qualquer cadeia  $x \in L$ , existe um certificado  $y$  que  $A$  pode utilizar para provar que  $x \in L$ . Além disso, para qualquer cadeia  $x \notin L$ , não deve haver nenhum certificado provando que  $x \in L$ . Por exemplo, no problema do ciclo hamiltoniano, o certificado é a lista de vértices no ciclo hamiltoniano. Se um grafo é hamiltoniano, o próprio ciclo hamiltoniano oferece informações suficientes para verificar esse fato. Reciprocamente, se um grafo não é hamiltoniano, não existe nenhuma lista de vértices que possa enganar o algoritmo de verificação fazendo-o acreditar que o grafo é hamiltoniano, pois o algoritmo de verificação examina cuidadosamente o “ciclo” proposto para ter certeza.

## A classe de complexidade NP

A **classe de complexidade NP** é a classe de linguagens que podem ser verificadas por um algoritmo de tempo polinomial.<sup>6</sup> Mais precisamente, uma linguagem  $L$  pertence a NP se e somente se existe um algoritmo de tempo polinomial de duas entradas  $A$  e uma constante  $c$  tal que

---

<sup>6</sup> O nome “NP” significa, em inglês, “tempo polinomial não determinístico” (nondeterministic polynomial time). A classe NP foi estudada originalmente no contexto de não determinismo, mas este livro usa a noção um pouco mais simples, ainda que equivalente, de verificação. Hopcroft e Ullman [156] fornecem uma boa apresentação do caráter NP-completo em termos de modelos não determinísticos de computação.

$$L = \{x \in \{0, 1\}^*: \text{existe um certificado } y \text{ com } |y| = O(|x|^\gamma) \text{ tal que } A(x, y) = 1\}.$$

Dizemos que o algoritmo  $A$  *verifica* a linguagem  $L$  em tempo polinomial.

De nossa discussão anterior sobre o problema do ciclo hamiltoniano, segue-se que HAM-CYCLE  $\in$  NP. (É sempre agradável saber que um conjunto importante é não vazio.) Além disso, se  $L \in P$ , então  $L \in NP$  pois, se existe um algoritmo de tempo polinomial para decidir  $L$ , o algoritmo pode ser facilmente convertido em um algoritmo de verificação de dois argumentos que simplesmente ignore qualquer certificado e aceite exatamente as cadeias de entrada que ele determina para estar em  $L$ . Logo,  $P \subseteq NP$ .

Não se sabe se  $P = NP$ , mas a maioria dos pesquisadores acredita que  $P$  e  $NP$  não são a mesma classe. Intuitivamente, a classe  $P$  consiste em problemas que podem ser resolvidos com rapidez. A classe  $NP$  consiste em problemas para os quais uma solução pode ser verificada rapidamente. Você deve ter aprendido com a experiência que freqüentemente é mais difícil resolver um problema a partir do início que verificar uma solução apresentada com clareza, em especial quando se trabalha sob restrições de tempo. Os cientistas da computação teórica em geral acreditam que essa analogia se estende às classes  $P$  e  $NP$  e, desse modo, que  $NP$  inclui linguagens que não estão em  $P$ .

Existe uma evidência mais forte de que  $P \neq NP$  – a existência de linguagens “NP-completas”. Estudaremos essa classe na Seção 34.3.

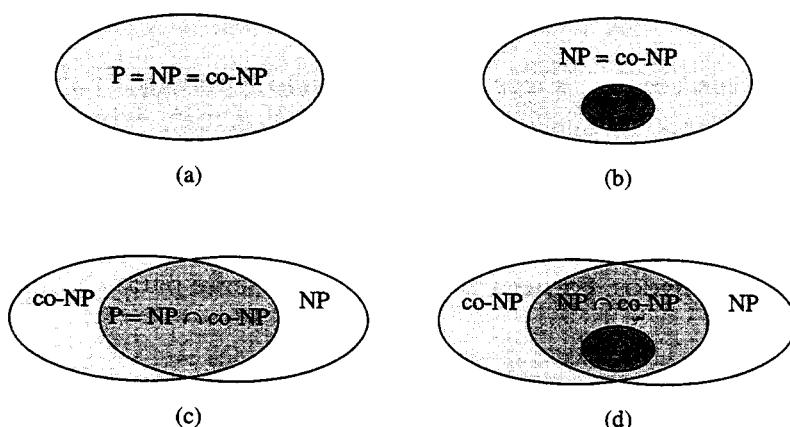


FIGURA 34.3 Quatro possibilidades de relacionamentos entre classes de complexidade. Em cada diagrama, uma região envolvendo outra indica uma relação de subconjunto próprio. (a)  $P = NP = co-NP$ . A maioria dos pesquisadores considera essa possibilidade a mais improvável. (b) Se  $NP$  é fechada sob o complemento, então  $NP = co-NP$ , mas não é preciso haver  $P = NP$ . (c)  $NP \cap co-NP$ , mas  $NP$  não é fechada sob o complemento. (d)  $NP \neq co-NP$  e  $NP \cap co-NP$ . A maioria dos pesquisadores considera essa possibilidade a mais provável

Muitas outras perguntas fundamentais além da pergunta de  $P \neq NP$  permanecem não resolvidas. Apesar do grande trabalho de muitos pesquisadores, ninguém sabe sequer se a classe  $NP$  é fechada sob o complemento. Isto é,  $L \in NP$  implica  $\bar{L} \in NP$ ? Podemos definir a *classe de complexidade co-NP* como o conjunto de linguagens  $L$  tais que  $\bar{L} \in NP$ . A questão de saber se  $NP$  é fechada sob o complemento pode ser redefinida como se  $NP = co-NP$ . Tendo em vista que  $P$  é fechada sob o complemento (Exercício 34.1-6), decorre que  $P \subseteq NP \cap co-NP$ . Porém, mais uma vez não se sabe se  $P = NP \cap co-NP$  ou se existe alguma linguagem em  $NP \cap co-NP - P$ . A Figura 34.3 mostra os quatro cenários possíveis.

Desse modo, nossa compreensão da relação precisa entre  $P$  e  $NP$  é terrivelmente incompleta. Apesar disso, explorando a teoria do caráter NP-completo, descobriremos que nossa desvantagem na demonstração de problemas como intratáveis está, de um ponto de vista prático, longe de ser tão grande quanto poderíamos supor.

## Exercícios

### 34.2-1

Considere a linguagem GRAPH-ISOMORPHISM = { $\langle G, u, v, k \rangle : G_1$  e  $G_2$  são grafos isomórficos}. Prove que GRAPH-ISOMORPHISM  $\in$  NP, descrevendo um algoritmo de tempo polinomial para verificar a linguagem.

### 34.2-2

Prove que, se  $G$  é um grafo bipartido não orientado com um número ímpar de vértices, então  $G$  é não hamiltoniano.

### 34.2-3

Mostre que, se HAM-CYCLE  $\in$  P, então o problema de listar os vértices de um ciclo hamiltoniano, em ordem, pode ser resolvido em tempo polinomial.

### 34.2-4

Prove que a classe NP de linguagens é fechada sob a união, a interseção, a concatenação e a estrela de Kleene. Discuta o fechamento de NP sob o complemento.

### 34.2-5

Mostre que qualquer linguagem em NP pode ser decidida por um algoritmo que é executado no tempo  $2^{O(n^k)}$  para alguma constante  $k$ .

### 34.2-6

Um **caminho hamiltoniano** em um grafo é um caminho simples que visita todo vértice exatamente uma vez. Mostre que a linguagem HAM-PATH = { $\langle G, u, v, k \rangle : \text{existe um caminho hamiltoniano desde } u \text{ até } v \text{ no grafo } G$ } pertence a NP.

### 34.2-7

Mostre que o problema do caminho hamiltoniano pode ser resolvido em tempo polinomial sobre grafos acíclicos orientados. Forneça um algoritmo eficiente para o problema.

### 34.2-8

Seja  $\phi$  uma fórmula booleana construída a partir das variáveis de entrada booleanas  $x_1, x_2, \dots, x_k$ , de negações ( $\neg$ ), AND ( $\wedge$ ), OR ( $\vee$ ) e parênteses. A fórmula [ver símbolo] é uma **tautologia** se ela tem o valor 1 para toda atribuição de 1 e 0 às variáveis de entrada. Defina TAUTOLOGY como a linguagem de fórmulas booleanas que são tautologias. Mostre que TAUTOLOGY  $\in$  co-NP.

### 34.2-9

Prove que  $P \subseteq \text{co-NP}$ .

### 34.2-10

Prove que, se  $\text{NP} \neq \text{co-NP}$ , então  $P \neq \text{NP}$ .

### 34.2-11

Seja  $G$  um grafo conectado não orientado com pelo menos 3 vértices, e seja  $G^3$  o grafo obtido pela conexão de todos os pares de vértices que estão conectados por um caminho em  $G$  de comprimento máximo 3. Prove que  $G^3$  é hamiltoniano. (Sugestão: Construa uma árvore de amplitude para  $G$  e use um argumento induutivo.)

## 34.3 Caráter NP-completo e redutibilidade

Talvez a razão mais forte pela qual os cientistas da computação teórica acreditam que  $P \neq \text{NP}$  seja a existência da classe de problemas “NP-completos”. Essa classe tem a surpreendente propriedade de que, se *qualquer* problema NP-completo pode ser resolvido em tempo polinomial, então *todo* problema em NP tem uma solução em tempo polinomial, isto é,  $P = \text{NP}$ . Entretanto, apesar

de anos de estudo, nenhum algoritmo de tempo polinomial jamais foi descoberto para qualquer problema NP-completo.

A linguagem HAM-CYCLE é um problema NP-completo. Se pudéssemos decidir HAM-CYCLE em tempo polinomial, então poderíamos resolver todo problema em NP em tempo polinomial. De fato, se  $NP - P$  fosse não vazia, poderíamos dizer com certeza que  $HAM-CYCLE \in NP - P$ .

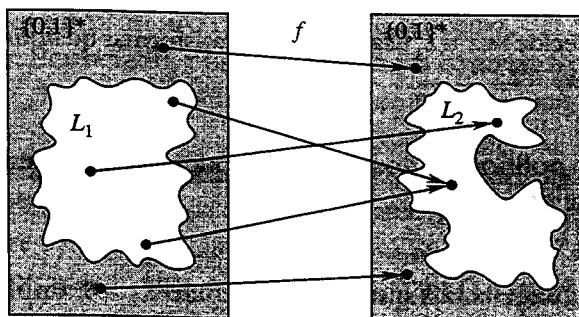


FIGURA 34.4 Uma ilustração de uma redução de tempo polinomial de uma linguagem  $L_1$  a uma linguagem  $L_2$  por meio de uma função de redução  $f$ . Para qualquer entrada  $x \in \{0, 1\}^*$ , a questão de saber se  $x \in L_1$  tem a mesma resposta que a questão de saber se  $f(x) \in L_2$

As linguagens NP-completas são, em certo sentido, as linguagens “mais difíceis” em NP. Nesta seção, mostraremos como comparar a “dificuldade” relativa de linguagens usando uma noção precisa chamada “redutibilidade em tempo polinomial”. Primeiro, definimos formalmente as linguagens NP-completas, e depois esboçamos uma prova de que uma dessas linguagens, chamada CIRCUIT-SAT, é NP-completa. Nas Seções 34.4 e 34.5, usaremos a noção de redutibilidade para mostrar que muitos outros problemas são NP-completos.

## Redutibilidade

Intuitivamente, um problema  $Q$  pode ser reduzido a outro problema  $Q'$  se qualquer instância de  $Q$  pode ser “facilmente reformulada” como uma instância de  $Q'$ , cuja solução fornece uma solução para a instância de  $Q$ . Por exemplo, o problema de resolver equações lineares em um  $x$  indeterminado se reduz ao problema de resolver equações quadráticas. Dada uma instância  $ax + b = 0$ , transformamos essa instância em  $0x^2 + ax + b = 0$ , cuja solução fornece uma solução para  $ax + b = 0$ . Desse modo, se um problema  $Q$  se reduz a outro problema  $Q'$ , então  $Q$  não é, em certo sentido, “mais difícil de resolver” que  $Q'$ .

Retornando à nossa estrutura de linguagem formal para problemas de decisão, dizemos que uma linguagem  $L_1$  é **redutível em tempo polinomial** a uma linguagem  $L_2$ , o que se escreve como  $L_1 \leq_p L_2$ , se existe uma função calculável de tempo polinomial  $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$  tal que, para todo  $x \in \{0, 1\}^*$ ,

$$x \in L_1 \text{ se e somente se } f(x) \in L_2. \quad (34.1)$$

Chamamos a função  $f$  **função de redução**, e um algoritmo de tempo polinomial  $F$  que calcula  $f$  é chamado **algoritmo de redução**.

A Figura 34.4 ilustra a idéia de uma redução de tempo polinomial de uma linguagem  $L_1$  a outra linguagem  $L_2$ . Cada linguagem é um subconjunto de  $\{0, 1\}^*$ . A função de redução  $f$  fornece um mapeamento em tempo polinomial tal que, se  $x \in L_1$ , então  $f(x) \in L_2$ . Além disso, se  $x \notin L_1$ , então  $f(x) \notin L_2$ . Desse modo, a função de redução mapeia qualquer instância  $x$  do problema de decisão representado pela linguagem  $L_1$  para uma instância  $f(x)$  do problema representado por  $L_2$ . Fornecer uma resposta à questão se  $f(x) \in L_2$  fornece diretamente a resposta à questão de  $x \in L_1$ .

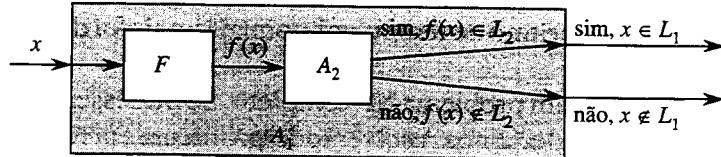


FIGURA 34.5 A prova do Lema 34.3. O algoritmo  $F$  é um algoritmo de redução que calcula a função de redução  $f$  de  $L_1$  a  $L_2$  em tempo polinomial, e  $A_2$  é um algoritmo de tempo polinomial que decide  $L_2$ . Está ilustrado um algoritmo  $A_1$  que decide se  $x \in L_1$  usando  $F$  para transformar qualquer entrada  $x$  em  $f(x)$ , e então usando  $A_2$  para decidir se  $f(x) \in L_2$ .

As reduções de tempo polinomial nos dão uma poderosa ferramenta para provar que diversas linguagens pertencem a P.

### **Lema 34.3**

Se  $L_1, L_2 \subseteq \{0, 1\}^*$  são linguagens tais que  $L_1 \leq_p L_2$ , então  $L_2 \in P$  implica  $L_1 \in P$ .

**Prova** Seja  $A_2$  um algoritmo de tempo polinomial que decide  $L_2$ , e seja  $F$  um algoritmo de redução de tempo polinomial que calcula a função de redução  $f$ . Construiremos um algoritmo de tempo polinomial  $A_1$  que decide  $L_1$ .

A Figura 34.5 ilustra a construção de  $A_1$ . Para uma dada entrada  $x \in \{0, 1\}^*$ , o algoritmo  $A_1$  usa  $F$  para transformar  $x$  em  $f(x)$ , e depois usa  $A_2$  para testar se  $f(x) \in L_2$ . A saída de  $A_2$  é o valor fornecido como a saída de  $A_1$ .

A correção de  $A_1$  decorre da condição (34.1). O algoritmo é executado em tempo polinomial, pois tanto  $F$  quanto  $A_2$  são executados em tempo polinomial (ver Exercício 34.1-5). ■

## Problemas NP-completos

As reduções de tempo polinomial proporcionam um meio formal de mostrar que um problema é pelo menos tão difícil quanto outro, até dentro de um fator de tempo polinomial. Isto é, se  $L_1 \leq_p L_2$ , então  $L_1$  não é mais que um fator polinomial mais difícil que  $L_2$ , e esse é o motivo pelo qual a notação “menor que ou igual a” para redução é mnemônica. Podemos agora definir o conjunto de linguagens NP-completas, que são os problemas mais difíceis em NP.

Uma linguagem  $L \subseteq \{0, 1\}^*$  é **NP-completa** se

1.  $L \in NP$ , e
2.  $L' \leq_p L$  para todo  $L' \in NP$ .

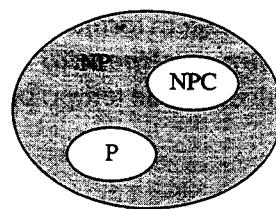


FIGURA 34.6 O modo como a maioria dos cientistas da computação teórica vê os relacionamentos entre P, NP e NPC. Tanto P quanto NPC estão inteiramente contidas dentro de NP, e  $P \cap NPC = \emptyset$

Se uma linguagem  $L$  satisfaz à propriedade 2, mas não necessariamente à propriedade 1, dizemos que  $L$  é **NP-difícil**. Também definimos NPC como a classe de linguagens NP-completas.

Como mostra o teorema a seguir, o caráter NP-completo é o ponto crucial de se determinar se P é de fato igual a NP.

#### **Teorema 34.4**

Se qualquer problema NP-completo pode ser resolvido em tempo polinomial, então  $P = NP$ . Se qualquer problema em NP não pode ser resolvido em tempo polinomial, então nenhum problema NP-completo pode ser resolvido em tempo polinomial.

**Prova** Suponha que  $L \in P$  e também que  $L \in NPC$ . Para qualquer  $L' \in NP$ , temos  $L' \leq_p L$  pela propriedade 2 da definição do caráter NP-completo. Desse modo, pelo Lema 34.3, também temos que  $L' \in P$ , o que prova o primeiro enunciado do teorema. ■

Para provar o segundo enunciado, observe que ele é o contrapositivo do primeiro enunciado. ■

É por essa razão que a pesquisa sobre a questão  $P \neq NP$  se concentra em torno dos problemas NP-completos. A maioria dos cientistas da computação teórica acredita que  $P \neq NP$ , o que conduz aos relacionamentos entre  $P$ ,  $NP$  e  $NPC$  mostrados na Figura 34.6. Porém, por tudo que sabemos, alguém poderia apresentar um algoritmo de tempo polinomial para um problema NP-completo, e desse modo provar que  $P = NP$ . Não obstante, como ainda não foi descoberto nenhum algoritmo de tempo polinomial para qualquer problema NP-completo, uma prova de que um problema é NP-completo fornece uma excelente evidência para a impossibilidade de seu tratamento.

## Satisfabilidade de circuitos

Definimos a noção de um problema NP-completo mas, até este ponto, não provamos realmente que qualquer problema é NP-completo. Depois de provarmos que pelo menos um problema é NP-completo, poderemos usar a redutibilidade de tempo polinomial como uma ferramenta para provar o caráter NP-completo de outros problemas. Desse modo, agora concentraremos nossa atenção na demonstração da existência de um problema NP-completo: o problema da satisfabilidade de circuitos.

Infelizmente, a prova formal de que o problema da satisfabilidade de circuitos é NP-completo exige detalhes técnicos que estão além do escopo deste texto. Em vez disso, descreveremos de modo informal uma prova que depende de uma compreensão básica de circuitos combinacionais booleanos.

Os circuitos combinacionais booleanos são construídos a partir de elementos combinacionais booleanos que são interconectados por fios. Um **elemento combinacional booleano** é qualquer elemento de circuito que tem um número constante de entradas e saídas booleanas e que executa uma função bem definida. Os valores booleanos são obtidos a partir do conjunto  $\{0, 1\}$ , onde 0 representa FALSE e 1 representa TRUE.

Os elementos combinacionais booleanos que utilizamos no problema de satisfabilidade de circuitos calculam uma função booleana simples e são chamados **portas lógicas**. A Figura 34.7 mostra as três portas lógicas básicas que usamos no problema de satisfabilidade de circuitos: a **porta NOT** (ou *inversora*), a **porta AND** e a **porta OR**. A porta NOT toma uma única **entrada** binária  $x$ , cujo valor é 0 ou 1, e produz uma **saída** binária  $z$  cujo valor é oposto ao valor da entrada. Cada uma das outras duas portas toma duas entradas binárias  $x$  e  $y$  e produz uma única saída binária  $z$ .

A operação de cada porta, e de qualquer elemento combinacional booleano, pode ser descrita por uma **tabela verdade**, mostrada sob cada porta na Figura 34.7. Uma tabela verdade fornece as saídas do elemento combinacional para cada configuração possível das entradas. Por exemplo, a tabela verdade para a porta OR informa que, quando as entradas são  $x = 0$  e  $y = 1$ , o valor da saída é  $z = 1$ . Usamos os símbolos  $\neg$  para denotar a função NOT,  $\wedge$  para denotar a função AND e  $\vee$  para denotar a função OR. Desse modo, por exemplo,  $0 \vee = 1$ .

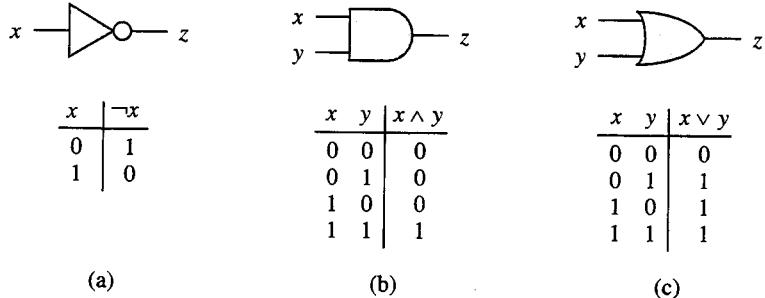


FIGURA 34.7 Três portas lógicas básicas, com entradas e saídas binárias. Sob cada porta está a tabela verdade que descreve a operação da porta. (a) A porta NOT. (b) A porta AND. (c) A porta OR

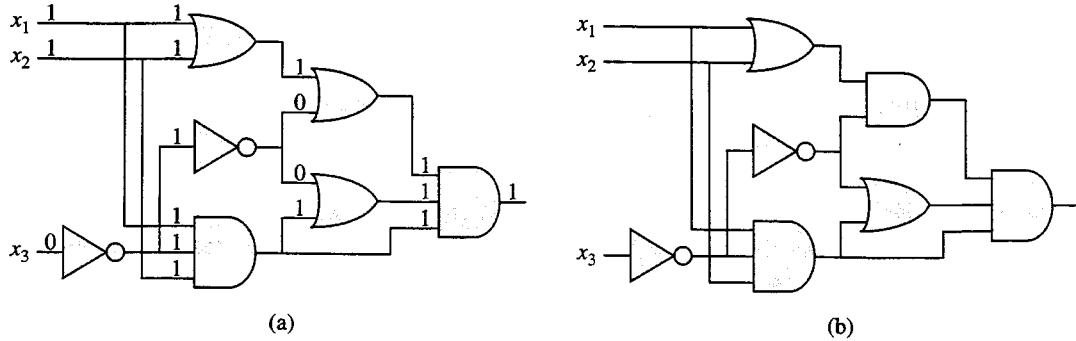
Podemos generalizar as portas AND e OR para tomar mais de duas entradas. A saída de uma porta AND é 1 se todas as suas entradas são 1, e sua saída é 0 em caso contrário. A saída de uma porta OR é 1 se qualquer de suas entradas é 1, e sua saída é 0 em caso contrário.

Um *circuito combinacional booleano* consiste em um ou mais elementos combinacionais booleanos interconectados por fios. Um fio pode conectar a saída de um elemento à entrada de outro, fornecendo assim o valor de saída do primeiro elemento como um valor de entrada do segundo. A Figura 34.8 mostra dois circuitos combinacionais booleanos semelhantes; eles diferem somente em uma porta. A parte (a) da figura também mostra os valores nos fios individuais, dada a entrada  $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ . Embora um único fio não possa ter mais de uma saída de elemento combinacional conectada a ele, é possível que esse fio alimente várias entradas de elementos. O número de entradas de elementos alimentadas por um fio é chamado **saída em leque** do fio. Se nenhuma saída de elemento está conectada a um fio, o fio é uma **entrada de circuito**, aceitando valores de entrada de uma fonte externa. Se nenhuma entrada de elemento está conectada a um fio, o fio é uma **saída de circuito**, fornecendo os resultados da computação do circuito para o mundo exterior. (Um fio interno também pode ter uma saída em leque para uma saída de circuito.) Com a finalidade de definir o problema de satisfabilidade de circuitos, limitamos o número de saídas de circuitos a 1, embora no projeto de hardware real, um circuito combinacional booleano possa ter várias saídas.

Os circuitos combinacionais booleanos não contêm nenhum ciclo. Em outras palavras, suponha que criamos um grafo orientado  $G = (V, E)$  com um vértice para cada elemento combinacional e com  $k$  arestas orientadas para cada fio cuja saída em leque é  $k$ ; existe uma aresta orientada  $(u, v)$  se um fio conecta a saída do elemento  $u$  a uma entrada de elemento  $v$ . Então,  $G$  deve ser acíclico.

Uma **atribuição verdade** para um circuito combinacional booleano é um conjunto de valores de entrada booleanos. Dizemos que um circuito combinacional booleano de uma saída é **capaz de satisfação** se ele tem uma **atribuição satisfatória**: uma atribuição verdade que faz a saída do circuito ser igual a 1. Por exemplo, o circuito da Figura 34.8(a) tem a atribuição satisfatória  $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ , e assim é capaz de satisfação. Como o Exercício 34.3-1 lhe pede para mostrar, nenhuma atribuição de valores para  $x_1, x_2$  e  $x_3$  faz o circuito da Figura 34.8(b) produzir uma saída 1; ele sempre produz 0, e assim é incapaz de satisfação.

O **problema da satisfabilidade de circuitos** é: “Dado um circuito combinacional booleano composto de portas AND, OR e NOT, ele é capaz de satisfação?” Porém, a fim de propor essa pergunta de modo formal, devemos concordar sobre uma codificação padrão para circuitos. O **tamanho** de um circuito combinacional booleano é o número de elementos combinacionais somado ao número de fios no circuito. É possível criar uma codificação semelhante à de grafos que mapeie qualquer circuito  $C$  dado em uma cadeia binária  $\langle C \rangle$  cujo comprimento seja polinomial no tamanho do próprio circuito. Como uma linguagem formal, podemos então definir



**FIGURA 34.8** Duas instâncias do problema de satisfatibilidade de circuitos. (a) A atribuição  $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$  para as entradas desse circuito faz a saída do circuito se tornar 1. O circuito é então capaz de satisfação. (b) Nenhuma atribuição para as entradas desse circuito pode fazer a saída do circuito se tornar 1. O circuito é então incapaz de satisfação.

O problema da satisfabilidade de circuitos tem grande importância na área de otimização de hardware auxiliada pelo computador. Se um subcircuito sempre produz 0, esse subcircuito pode ser substituído por um subcircuito mais simples que omite todas as portas lógicas e fornece o valor constante 0 como sua saída. Um algoritmo de tempo polinomial para esse problema teria considerável utilidade.

Dado um circuito  $C$ , podemos tentar determinar se ele é capaz de satisfação apenas verificando todas as atribuições possíveis para as entradas. Infelizmente, se existem  $k$  entradas, existem  $2^k$  atribuições possíveis. Quando o tamanho de  $C$  é polinomial em  $k$ , a verificação de cada uma demora o tempo  $\Omega(2^k)$ , que é superpolinomial no tamanho do circuito.<sup>7</sup> De fato, como se tem afirmado, há uma forte evidência de que não existe nenhum algoritmo de tempo polinomial que resolva o problema da satisfabilidade de circuitos, porque a satisfabilidade de circuitos é NP-completa. Dividimos a prova desse fato em duas partes, com base nas duas partes da definição do caráter NP-completo.

### **Lema 34.5**

O problema da satisfabilidade de circuitos pertence à classe NP.

**Prova** Forneceremos um algoritmo de tempo polinomial  $A$  de duas entradas que pode verificar CIRCUIT-SAT. Uma das entradas para  $A$  é (uma codificação padrão de) um circuito combinatorial booleano  $C$ . A outra entrada é um certificado que corresponde a uma atribuição de valores booleanos aos fios em  $C$ . (Veja no Exercício 34.3-4 um certificado menor.)

O algoritmo  $A$  é construído da maneira mostrada a seguir. Para cada porta lógica no circuito, ele verifica se o valor fornecido pelo certificado no fio de saída é corretamente calculado como uma função dos valores nos fios de entrada. Em seguida, se a saída do circuito inteiro é 1, o algoritmo fornece a saída 1, pois os valores atribuídas às entradas de  $C$  fornecem uma atribuição satisfatória. Caso contrário,  $A$  fornece a saída 0.

Sempre que um circuito capaz de satisfação  $C$  é dado como entrada para o algoritmo  $A$ , existe um certificado cujo comprimento é polinomial no tamanho de  $C$  e que faz  $A$  fornecer a saída 1. Sempre que um circuito incapaz de satisfação é dado como entrada, nenhum certificado pode fazer  $A$  acreditar que o circuito é capaz de satisfação. O algoritmo  $A$  é executado em tempo polinomial: com uma boa implementação, o tempo linear é suficiente. Desse modo, CIRCUIT-SAT pode ser verificado em tempo polinomial, e CIRCUIT-SAT  $\in$  NP.

<sup>7</sup>Por outro lado, se o tamanho do circuito  $C$  é  $\Theta(2^k)$ , então um algoritmo cujo tempo de execução é  $O(2^k)$  tem um tempo de execução que é polinomial no tamanho do circuito. Mesmo se  $P \neq NP$ , essa situação não iria contradizer o fato de que o problema é  $NP$ -completo; a existência de um algoritmo de tempo polinomial para um caso especial não implica que existe um algoritmo de tempo polinomial para todos os casos.

A segunda parte da prova de que CIRCUIT-SAT é NP-completo é mostrar que a linguagem é NP-difícil. Isto é, devemos mostrar que toda linguagem em NP é reduzível em tempo polinomial a CIRCUIT-SAT. A prova real desse fato é cheia de complexidades técnicas, e assim devemos nos limitar a um esboço da prova baseado em alguma compreensão do funcionamento interno do hardware de computadores.

Um programa de computador é armazenado na memória do computador como uma seqüência de instruções. Uma instrução típica codifica uma operação a ser executada, endereços de operandos na memória e um endereço onde o resultado deve ser armazenado. Uma posição de memória especial, chamada **contador de programa**, controla a instrução a ser executada em seguida. O contador de programa é incrementado automaticamente sempre que uma instrução é recuperada, fazendo assim o computador executar instruções seqüencialmente. Porém, a execução de uma instrução pode fazer um valor ser escrito no contador de programa, e então a execução seqüencial normal pode ser alterada, permitindo ao computador entrar em loop e executar desvios condicionais.

Em qualquer ponto durante a execução de um programa, o estado inteiro da computação é representado na memória do computador. (Fazemos a memória incluir o próprio programa, o contador de programa, o espaço de armazenamento de trabalho e quaisquer dos diversos bits de estado que um computador mantém para contabilidade.) Chamamos qualquer estado particular da memória do computador uma **configuração**. A execução de uma instrução pode ser vista como o mapeamento de uma configuração para outra. Mais importante ainda, o hardware do computador que executa esse mapeamento pode ser implementado como um circuito combinacional booleano, que denotamos por  $M$  na prova do lema a seguir.

### Lema 34.6

O problema da satisfabilidade de circuitos é NP-difícil.

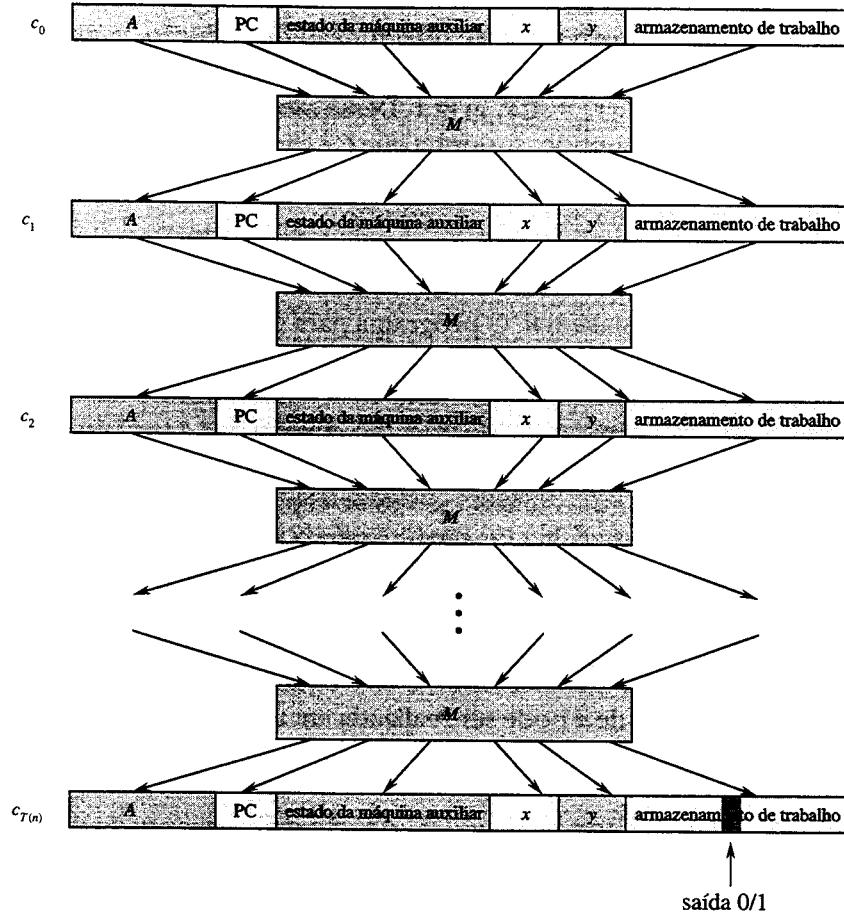
**Prova** Seja  $L$  qualquer linguagem em NP. Descreveremos um algoritmo de tempo polinomial  $F$  que calcula uma função de redução  $f$  que mapeia toda cadeia binária  $x$  para um circuito  $C = f(x)$  tal que  $x \in L$  se e somente se  $C \in \text{CIRCUIT-SAT}$ .

Tendo em vista que  $L \in \text{NP}$ , deve existir um algoritmo  $A$  que verifique  $L$  em tempo polinomial. O algoritmo  $F$  que construiremos usará o algoritmo  $A$  de duas entradas para calcular a função de redução  $f$ .

Seja  $T(n)$  o tempo de execução do pior caso do algoritmo  $A$  sobre cadeias de entrada de comprimento  $n$ , e seja  $k \geq 1$  uma constante tal que  $T(n) = O(n^k)$  e o comprimento do certificado seja  $O(n^k)$ . (O tempo de execução de  $A$  é na realidade um polinômio no tamanho total da entrada, o que inclui tanto uma cadeia de entrada quanto um certificado; contudo, como o comprimento do certificado é polinomial no comprimento  $n$  da cadeia de entrada, o tempo de execução é polinomial em  $n$ .)

A idéia básica da prova é representar a computação de  $A$  como uma seqüência de configurações. Como mostra a Figura 34.9, cada configuração pode ser dividida em partes que consistem no programa para  $A$ , o contador de programa e estado da máquina auxiliar, a entrada  $x$ , o certificado  $y$  e o espaço de armazenamento de trabalho. Começando com uma configuração inicial  $c_0$ , cada configuração  $c_i$  é mapeada em uma configuração subsequente  $c_{i+1}$  pelo circuito combinacional  $M$  que implementa o hardware do computador. A saída do algoritmo  $A$  – 0 ou 1 – é gravada em algum local designado na área de armazenamento de trabalho quando  $A$  termina sua execução e, se supomos que daí em diante  $A$  pára, o valor nunca muda. Desse modo, se o algoritmo é executado para no máximo  $T(n)$  passos, a saída aparece como um dos bits em  $c_{T(n)}$ .

O algoritmo de redução  $F$  constrói um único circuito combinacional que calcula todas as configurações produzidas por uma dada configuração inicial. A idéia é juntar  $T(n)$  cópias do circuito  $M$ . A saída do  $i$ -ésimo circuito, que produz a configuração  $c_i$ , é inserida diretamente na entrada do  $(i+1)$ -ésimo circuito. Desse modo, as configurações, em vez de terminarem em um registrador de estado, simplesmente residem como valores nos fios que conectam cópias de  $M$ .



**FIGURA 34.9** A seqüência de configurações produzidas por um algoritmo  $A$  em execução sobre uma entrada  $x$  e um certificado  $y$ . Cada configuração representa o estado do computador para um passo da computação e, além de  $A$ ,  $x$  e  $y$ , inclui o contador de programa (PC), o estado da máquina auxiliar e o espaço de armazenamento de trabalho. Exceto pelo certificado  $y$ , a configuração inicial  $c_0$  é constante. Cada configuração é mapeada para a configuração seguinte por um circuito combinacional booleano  $M$ . A saída é um bit distinto no espaço de armazenamento de trabalho

Lembre-se daquilo que o algoritmo de redução de tempo polinomial  $F$  deve fazer. Dada uma entrada  $x$ , ele tem de calcular um circuito  $C = f(x)$  que seja capaz de satisfação se e somente se existe um certificado  $y$  tal que  $A(x, y) = 1$ . Quando  $F$  obtém uma entrada  $x$ , primeiro ele calcula  $n = |x|$  e constrói um circuito combinacional  $C'$  que consiste em  $T(n)$  cópias de  $M$ . A entrada para  $C'$  é uma configuração inicial correspondente a uma computação sobre  $A(x, y)$ , e a saída é a configuração  $c_{T(n)}$ .

O circuito  $C = f(x)$  que  $F$  calcula é obtido por uma ligeira modificação de  $C'$ . Primeiro, as entradas para  $C'$  correspondentes ao programa para  $A$ , o contador de programa inicial, a entrada  $x$  e o estado inicial da memória são conectados diretamente a esses valores conhecidos. Desse modo, as únicas entradas restantes para o circuito correspondem ao certificado  $y$ . Segundo, todas as saídas do circuito são ignoradas, exceto o único bit de  $c_{T(n)}$  que corresponde à saída de  $A$ . Esse circuito  $C$  assim construído calcula  $C(y) = A(x, y)$  para qualquer entrada  $y$  de comprimento  $O(n^k)$ . O algoritmo de redução  $F$ , quando recebe uma cadeia de entrada  $x$ , calcula esse circuito  $C$  e o mostra como saída.

Resta provar duas propriedades. Primeiro, devemos mostrar que  $F$  calcula corretamente uma função de redução  $f$ . Isto é, devemos mostrar que  $C$  é capaz de satisfação se e somente se existe um certificado  $y$  tal que  $A(x, y) = 1$ . Em segundo lugar, devemos mostrar que  $F$  é executado em tempo polinomial.

Para mostrar que  $F$  calcula corretamente uma função de redução, vamos supor que exista um certificado  $y$  de comprimento  $O(n^k)$  tal que  $A(x, y) = 1$ . Então, se aplicarmos os bits de  $y$  às entradas de  $C$ , a saída de  $C$  será  $C(y) = A(x, y) = 1$ . Desse modo, se existe um certificado, então  $C$  é capaz de satisfação. Em outro sentido, suponha que  $C$  é capaz de satisfação. Conseqüentemente, existe uma entrada  $y$  para  $C$  tal que  $C(y) = 1$ , a partir da qual concluímos que  $A(x, y) = 1$ . Portanto,  $F$  calcula corretamente uma função de redução.

Para completar a prova, só precisamos mostrar que  $F$  é executado em tempo polinomial em  $n = |x|$ . A primeira observação que fazemos é que o número de bits necessários para representar uma configuração é polinomial em  $n$ . O programa para o próprio  $A$  tem tamanho constante, independente do comprimento de sua entrada  $x$ . O comprimento da entrada  $x$  é  $n$ , e o comprimento do certificado  $y$  é  $O(n^k)$ . Como o algoritmo é executado por no máximo  $O(n^k)$  passos, a quantidade de espaço de armazenamento de trabalho exigido por  $A$  também é polinomial em  $n$ . (Supomos que essa memória é contígua; o Exercício 34.3-5 lhe pede para estender o argumento à situação na qual as posições a que  $A$  tem acesso estão espalhadas por uma região de memória muito maior, e que o padrão específico de espalhamento pode diferir para cada entrada  $x$ .)

O circuito combinacional  $M$  que implementa o hardware no computador tem tamanho polinomial no comprimento de uma configuração, que é polinomial em  $O(n^k)$  e então é polinomial em  $n$ . (A maior parte desses circuitos implementa a lógica do sistema de memória.) O circuito  $C$  consiste em no máximo  $t = O(n^k)$  cópias de  $M$ , e consequentemente tem tamanho polinomial em  $n$ . A construção de  $C$  a partir de  $x$  pode ser realizada em tempo polinomial pelo algoritmo de redução  $F$ , pois cada passo da construção demora um tempo polinomial. ■

A linguagem CIRCUIT-SAT é portanto pelo menos tão difícil quanto qualquer linguagem em NP e, como pertence a NP, ela é NP-completa.

### **Teorema 34.7**

O problema da satisfatibilidade de circuitos é NP-completo.

**Prova** Imediata, a partir dos Lemas 34.5 e 34.6 e da definição do caráter NP-completo. ■

## **Exercícios**

### **34.3-1**

Confirme que o circuito da Figura 34.8(b) é incapaz de satisfação.

### **34.3-2**

Mostre que a relação  $\leq_p$  é uma relação transitiva em linguagens. Isto é, mostre que, se  $L_1 \leq_p L_2$  e  $L_2 \leq_p L_3$ , então  $L_1 \leq_p L_3$ .

### **34.3-3**

Prove que  $L \leq_p \bar{L}$  se e somente se  $\bar{L} \leq_p L$ .

### **34.3-4**

Mostre que uma atribuição satisfatória pode ser usada como um certificado em uma prova alternativa do Lema 34.5. Qual certificado favorece uma prova mais fácil?

### **34.3-5**

A prova do Lema 34.6 pressupõe que o espaço de armazenamento para o algoritmo  $A$  ocupa uma região contígua de tamanho polinomial. Em que parte da prova essa hipótese é explorada? Demonstre que essa hipótese não envolve qualquer perda de generalidade.

### **34.3-6**

Uma linguagem  $L$  é **completa** para uma classe de linguagem  $C$  com relação a reduções de tempo polinomial se  $L \in C$  e  $L' \leq_p L$  para todo  $L' \in C$ . Mostre que  $\emptyset$  e  $\{0, 1\}^*$  são as únicas linguagens em P que não são completas para P com relação a reduções de tempo polinomial.

### 34.3-7

Mostre que  $L$  é completa para NP se e somente se  $\bar{L}$  é completa para co-NP.

### 34.3-8

O algoritmo de redução  $F$  na prova do Lema 34.6 constrói o circuito  $C = f(x)$  baseado no conhecimento de  $x$ ,  $A$  e  $k$ . O professor Samuel observa que a cadeia  $x$  é uma entrada para  $F$ , mas apenas a existência de  $A$ ,  $k$  e do fator constante implícito no tempo de execução  $O(n^k)$  são conhecidos para  $F$  (pois a linguagem  $L$  pertence a NP), e não seus valores reais. Desse modo, o professor conclui que  $F$  não pode construir o circuito  $C$  e que a linguagem CIRCUIT-SAT não é necessariamente NP-difícil. Explique a falha no raciocínio do professor.

## 34.4 Provas do caráter NP-completo

O caráter NP-completo do problema da satisfabilidade de circuitos se baseia em uma prova direta de que  $L \leq_p$  CIRCUIT-SAT para toda linguagem  $L \in \text{NP}$ . Nesta seção, mostraremos como provar que as linguagens são NP-completas sem reduzir diretamente *toda* linguagem em NP à linguagem dada. Ilustraremos essa metodologia provando que vários problemas de satisfabilidade de fórmulas são NP-completos. A Seção 34.5 fornece muitos outros exemplos da metodologia.

O lema a seguir é a base de nosso método para mostrar que uma linguagem é NP-completa.

### Lema 34.8

Se  $L$  é uma linguagem tal que  $L' \leq_p L$  para alguma  $L' \in \text{NPC}$ , então  $L$  é NP-difícil. Além disso, se  $L \in \text{NP}$ , então  $L \in \text{NPC}$ .

**Prova** Tendo em vista que  $L'$  é NP-completa, para todo  $L'' \in \text{NP}$ , temos  $L'' \leq_p L'$ . Por hipótese,  $L' \leq_p L$  e, desse modo, por transitividade (Exercício 34.3-2), temos  $L'' \leq_p L$ , o que mostra que  $L$  é NP-difícil. Se  $L \in \text{NP}$ , também temos  $L \in \text{NPC}$ . ■

Em outras palavras, reduzindo a  $L$  uma linguagem NP-completa  $L'$  conhecida, reduzimos implicitamente toda linguagem em NP a  $L$ . Assim, o Lema 34.8 apresenta um método para provar que uma linguagem  $L$  é NP-completa:

1. Prove que  $L \in \text{NP}$ .
2. Selecione uma linguagem NP-completa conhecida  $L'$ .
3. Descreva um algoritmo que calcule uma função  $f$  que mapeie toda instância  $x \in \{0, 1\}^*$  de  $L'$  para uma instância  $f(x)$  de  $L$ .
4. Prove que a função  $f$  satisfaz a  $x \in L'$  se e somente se  $f(x) \in L$  para todo  $x \in \{0, 1\}^*$ .
5. Prove que o algoritmo que calcula  $f$  é executado em tempo polinomial.

(As etapas 2 a 5 mostram que  $L$  é NP-difícil.) Essa metodologia de redução a partir de uma única linguagem NP-completa conhecida é muitíssimo mais simples que o complicado processo de mostrar diretamente como efetuar reduções a partir de toda linguagem em NP. Provar que CIRCUIT-SAT  $\in \text{NPC}$  nos proporcionou uma “mão na roda”. Agora, saber que o problema da satisfabilidade de circuitos é NP-completo nos permite provar muito mais facilmente que outros problemas são NP-completos. Além disso, à medida que desenvolvemos um catálogo de problemas NP-completos conhecidos, teremos cada vez mais opções de linguagens a partir das quais poderá ser feita a redução.

## Satisfabilidade de fórmulas

Ilustramos a metodologia de redução dando uma prova do caráter NP-completo para o problema de determinar se uma fórmula booleana, não um circuito, é capaz de satisfação. Esse problema tem a honra histórica de ser o primeiro problema a ser apresentado como NP-completo.

Formulamos o problema da **satisfabilidade (de fórmulas)** em termos da linguagem SAT da maneira ilustrada a seguir. Uma instância de SAT é uma fórmula booleana  $\phi$  composta de

1.  $n$  variáveis booleanas:  $x_1, x_2, \dots, x_n$ ;
2.  $m$  conectivos booleanos: qualquer função booleana com uma ou duas entradas e uma saída, como  $\wedge$  (AND),  $\vee$  (OR),  $\neg$  (NOT),  $\rightarrow$  (implicação),  $\leftrightarrow$  (se e somente se); e
3. parênteses. (Sem perda de generalidade, supomos que não existem parênteses redundantes, isto é, existe no máximo um par de parênteses por conetivo booleano.)

É fácil codificar uma fórmula booleana  $\phi$  em um comprimento que seja polinomial em  $n + m$ . Como nos circuitos combinacionais booleanos, uma **atribuição verdade** para uma fórmula booleana  $\phi$  é um conjunto de valores para as variáveis de  $\phi$ , e uma **atribuição satisfatória** é uma atribuição verdade que faz com que ela seja avaliada como 1. Uma fórmula com uma atribuição satisfatória é uma fórmula **capaz de satisfação**. O problema da satisfabilidade pergunta se uma dada fórmula booleana é capaz de satisfação; em termos de linguagem formal,

$$\text{SAT} = \{\langle \phi \rangle : \phi \text{ é uma fórmula booleana capaz de satisfação}\}.$$

Como um exemplo, a fórmula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_A)) \wedge \neg x_2$$

tem a atribuição satisfatória  $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$ , pois

$$\begin{aligned} \phi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1) \wedge 1) \\ &= (1 \vee 0) \vee 1 \\ &= 1, \end{aligned} \tag{34.2}$$

e, desse modo, essa fórmula  $\phi$  pertence a SAT.

O algoritmo simples para determinar se uma fórmula booleana arbitrária é capaz de satisfação não é executado em tempo polinomial. Existem  $2^n$  atribuições possíveis em uma fórmula  $\phi$  com  $n$  variáveis. Se o comprimento de  $\langle \phi \rangle$  é polinomial em  $n$ , então a verificação de cada atribuição exige o tempo  $\Omega(2^n)$ , que é superpolinomial no comprimento de  $\langle \phi \rangle$ . Como mostra o teorema a seguir, é improvável que exista um algoritmo de tempo polinomial.

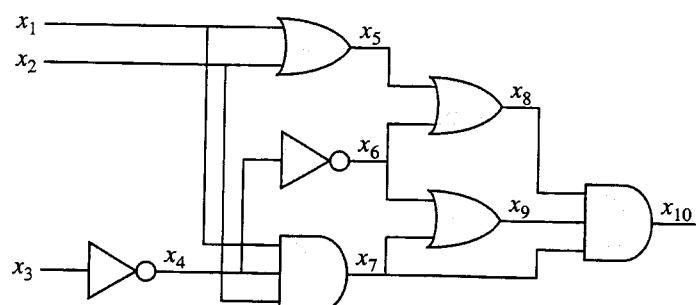


FIGURA 34.10 Redução da satisfabilidade de circuito à satisfabilidade de fórmula. A fórmula produzida pelo algoritmo de redução tem uma variável para cada fio no circuito

### **Teorema 34.9**

A satisfabilidade de fórmulas booleanas é NP-completa.

**Prova** Demonstraremos primeiro que  $SAT \in NP$ . Depois, provaremos que  $CIRCUIT-SAT$  é NP-difícil, mostrando que  $CIRCUIT-SAT \leq_p SAT$ ; pelo Lema 34.8, isso provará o teorema.

Para mostrar que  $SAT$  pertence a  $NP$ , mostraremos que um certificado que consiste em uma atribuição satisfatória para uma fórmula de entrada  $\phi$  pode ser verificado em tempo polinomial. O algoritmo de verificação simplesmente substitui cada variável na fórmula por seu valor correspondente, e então avalia a expressão, de forma muito semelhante ao que fizemos na equação (34.2) anterior. Essa tarefa é facilmente realizável em tempo polinomial. Se a expressão tem o valor 1, a fórmula é capaz de satisfação. Desse modo, a primeira condição do Lema 34.8 para o caráter NP-completo é válida.

Para provar que  $SAT$  é NP-difícil, mostramos que  $CIRCUIT-SAT \leq_p SAT$ . Em outras palavras, qualquer instância de satisfabilidade de circuito pode ser reduzida em tempo polinomial a uma instância de satisfabilidade de fórmula. A indução pode ser usada para expressar qualquer circuito combinacional booleano como uma fórmula booleana. Simplesmente observamos a porta que produz a saída do circuito e expressamos indutivamente cada uma das entradas da porta como fórmulas. A fórmula para o circuito é então obtida, escrevendo-se uma expressão que aplica a função da porta a fórmulas de suas entradas.

Infelizmente, esse método direto não constitui uma redução de tempo polinomial. Conforme o Exercício 34.4-1 lhe pede para mostrar, subfórmulas compartilhadas – que surgem a partir de portas cujos fios de saída têm saída em leque 2 ou maior – podem fazer o tamanho da fórmula gerada crescer exponencialmente. Desse modo, o algoritmo de redução deve ser um pouco mais inteligente.

A Figura 34.10 ilustra a idéia básica da redução de  $CIRCUIT-SAT$  a  $SAT$  no circuito da Figura 34.8(a). Para cada fio  $x_i$  no circuito  $C$ , a fórmula  $\phi$  tem uma variável  $x_i$ . A operação apropriada de uma porta pode agora ser expressa como uma fórmula envolvendo as variáveis de seus fios incidentes. Por exemplo, a operação da porta AND da saída é  $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$ .

A fórmula  $\phi$  produzida pelo algoritmo de redução é o AND da variável de saída do circuito com a conjunção de cláusulas descrevendo a operação de cada porta. Para o circuito da figura, a fórmula é

$$\begin{aligned}\phi = & x_{10} \wedge (x_4 \leftrightarrow \neg x_8) \\ & \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ & \wedge (x_6 \leftrightarrow \neg x_4) \\ & \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ & \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\ & \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\ & \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) .\end{aligned}$$

Dado um circuito  $C$ , produzir tal fórmula  $\phi$  em tempo polinomial é uma operação direta.

Por que o circuito  $C$  é capaz de satisfação exatamente quando a fórmula  $\phi$  é capaz de satisfação? Se  $C$  tem atribuição satisfatória, cada fio do circuito tem um valor bem definido, e a saída do circuito é 1. Então, a atribuição de valores de fios a variáveis em  $\phi$  faz cada cláusula de  $\phi$  apresentar o valor 1, e assim a conjunção de todos eles tem o valor 1. Reciprocamente, se existe uma atribuição que faz  $\phi$  apresentar o valor 1, o circuito  $C$  é capaz de satisfação por um argumento análogo. Desse modo, mostramos que  $CIRCUIT-SAT \leq_p SAT$ , o que completa a prova. ■

## Satisfabilidade de 3-CNF

Muitos problemas podem ser provados como NP-completos por redução da satisfabilidade de fórmulas. Entretanto, o algoritmo de redução deve tratar qualquer fórmula de entrada, e isso pode levar a um número enorme de casos que devem ser considerados. Assim, com freqüência é interessante efetuar a redução a partir de uma linguagem restrita de fórmulas booleanas, de forma que menos casos precisem ser considerados. É claro que não devemos restringir tanto a linguagem a ponto de ela poder ser resolvida em tempo polinomial. Uma linguagem conveniente é a satisfabilidade 3-CNF, ou 3-CNF-SAT.

Definimos a satisfabilidade 3-CNF usando os termos a seguir. Um **literal** em uma fórmula booleana é uma ocorrência de uma variável ou sua negação. Uma fórmula booleana está em **forma normal conjuntiva**, ou **CNF** (conjunctive normal form), se é expressa como um grupo AND de **cláusulas**, cada uma das quais é o OR de um ou mais literais. Uma fórmula booleana está em **3-forma normal conjuntiva**, ou **3-CNF**, se cada cláusula tem exatamente três literais distintos.

Por exemplo, a fórmula booleana

$$(x_1 \vee \neg x_1 \vee \neg x_2) (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

está em 3-CNF. A primeira de suas três cláusulas é  $(x_1 \vee \neg x_1 \vee \neg x_2)$ , que contém os três literais  $x_1$ ,  $\neg x_1$  e  $\neg x_2$ .

Em 3-CNF-SAT, somos levados a definir se uma dada fórmula booleana  $\phi$  em 3-CNF é capaz de satisfação. O teorema a seguir mostra que é improvável existir um algoritmo de tempo polinomial que possa determinar a satisfabilidade de fórmulas booleanas, mesmo quando elas são expressas nessa forma normal simples.

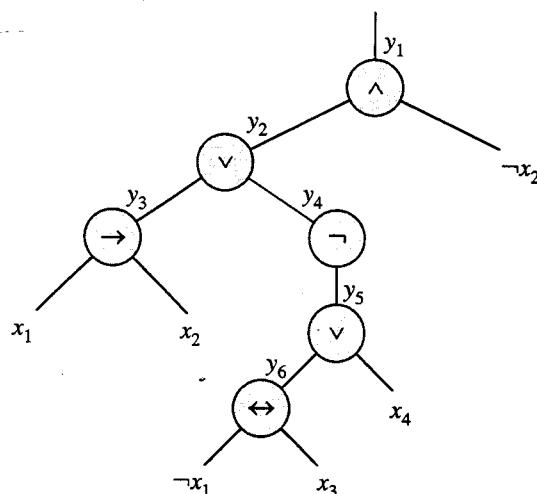


FIGURA 34.11 A árvore correspondente à fórmula  $\phi = ((x_1 \rightarrow x_2) \vee \neg ((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

### Teorema 34.10

A satisfabilidade de fórmulas booleanas em 3-forma normal conjuntiva é NP-completa.

**Prova** O argumento que usamos na prova do Teorema 34.9 para mostrar que SAT ∈ NP se aplica igualmente bem neste caso para mostrar que 3-CNF-SAT ∈ NP. Desse modo, pelo Lema 34.8, só precisamos mostrar que  $SAT \leq_p 3\text{-CNF-SAT}$ .

O algoritmo de redução pode ser dividido em três passos básicos. Cada passo transforma progressivamente a fórmula de entrada  $\phi$ , deixando-a mais próxima da 3-forma normal conjuntiva desejada.

O primeiro passo é semelhante ao que usamos para provar  $\text{CIRCUIT-SAT} \leq_p \text{SAT}$  no Teorema 34.9. Primeiro, construímos uma árvore binária “de análise” para a fórmula de entrada  $\phi$ , com literais como folhas e conectivos como nós internos. A Figura 34.11 mostra tal árvore de análise para a fórmula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg ((\neg x_1 \leftrightarrow x_3) \vee x_A)) \wedge \neg x_2 \quad (34.3)$$

Caso a fórmula de entrada contenha uma cláusula como OR de diversos literais, a associatividade pode ser usada para colocar a expressão totalmente entre parênteses, de forma que cada nó interno na árvore resultante tenha 1 ou 2 filhos. A árvore de análise binária pode agora ser vista como um circuito para calcular a função.

Imitando a redução na prova do Teorema 34.9, introduzimos uma variável  $y_i$  para a saída de cada nó interno. Em seguida, reescrevemos a fórmula original  $\phi$  como o AND da variável de raiz e uma conjunção de cláusulas descrevendo a operação de cada nó. Para a fórmula (34.3), a expressão resultante é

$$\begin{aligned} \phi = & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5)) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_7)) \\ & \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_3)) . \end{aligned}$$

| $y_1$ | $y_2$ | $x_2$ | $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ |
|-------|-------|-------|-----------------------------------------------|
| 1     | 1     | 1     | 0                                             |
| 1     | 1     | 0     | 1                                             |
| 1     | 0     | 1     | 0                                             |
| 1     | 0     | 0     | 0                                             |
| 0     | 1     | 1     | 1                                             |
| 0     | 1     | 0     | 0                                             |
| 0     | 0     | 1     | 1                                             |
| 0     | 0     | 0     | 1                                             |

FIGURA 34.12 A tabela verdade para a cláusula  $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ ,

Observe que a fórmula  $\phi'$  assim obtida é uma conjunção de cláusulas  $\phi'_i$ , cada qual com no máximo 3 literais. O único requisito adicional é que cada cláusula seja um OR de literais.

O segundo passo da redução converte cada cláusula  $\phi'_i$  para a forma normal conjuntiva. Construímos uma tabela verdade para  $\phi'_i$ , avaliando todas as atribuições possíveis para suas variáveis. Cada linha da tabela verdade consiste em uma atribuição possível das variáveis da cláusula, juntamente com o valor da cláusula sob essa atribuição. Usando as entradas da tabela verdade que são avaliadas como 0, construímos uma fórmula em **forma normal disjuntiva** (ou DNF – disjunctive normal form) – um OR de ANDs – que é equivalente a  $\neg \phi'_i$ . Em seguida, convertemos essa fórmula em uma fórmula CNF  $\phi''_i$ , usando as leis do DeMorgan (equações (B.2)) para complementar todos os literais e trocar OR por AND e AND por OR.

Em nosso exemplo, convertemos a cláusula  $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$  em CNF da maneira descrita a seguir. A tabela verdade para  $\phi'_1$  é dada na Figura 34.12. A fórmula DNF equivalente a  $\neg\phi'_1$  é

$$(y_1 \vee y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \wedge (y_1 \wedge \neg y_2 \wedge x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

Aplicando as leis de DeMorgan, obtemos a fórmula CNF

$$\begin{aligned}\phi'_1 &= (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ &\wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2),\end{aligned}$$

que é equivalente à cláusula original  $\phi'_1$ .

Cada cláusula  $\phi'_i$  da fórmula  $\phi'$  agora é convertida em uma fórmula CNF  $\phi''_i$ , e assim  $\phi'$  é equivalente à fórmula CNF  $\phi''$  que consiste na conjunção de  $\phi''_i$ . Além disso, cada cláusula de  $\phi''$  tem no máximo 3 literais.

O terceiro e último passo da redução faz uma transformação adicional da fórmula, de modo que cada cláusula tenha *exatamente* 3 literais distintos. A fórmula 3-CNF final  $\phi'''$  é construída a partir das cláusulas da fórmula CNF  $\phi''$ . Ela também usa duas variáveis auxiliares que chamaremos  $p$  e  $q$ . Para cada cláusula  $C_i$  de  $\phi''$ , incluímos as seguintes cláusulas em  $\phi'''$ :

- Se  $C_i$  tem 3 literais distintos, então simplesmente inclua  $C_i$  como uma cláusula de  $\phi'''$ .
- Se  $C_i$  tem 2 literais distintos, isto é, se  $C_i = (l_1 \vee l_2)$ , onde  $l_1$  e  $l_2$  são literais, então inclua  $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$  como cláusulas de  $\phi'''$ . Os literais  $p$  e  $\neg p$  cumprem apenas o requisito sintático de que devem existir exatamente 3 literais distintos por cláusula:  $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$  é equivalente a  $(l_1 \vee l_2)$  se  $p = 0$  ou  $p = 1$ .
- Se  $C_i$  tem apenas 1 literal  $l$  distinto, então inclua  $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$  como cláusulas de  $\phi'''$ . Observe que toda configuração de  $p$  e  $q$  faz a conjunção dessas quatro cláusulas ser avaliada como  $l$ .

Podemos ver que a fórmula 3-CNF  $\phi'''$  é capaz de satisfação se e somente se  $\phi$  é capaz de satisfação, inspecionando cada um dos três passos. Como a redução de CIRCUIT-SAT a SAT, a construção de  $\phi'$  a partir de  $\phi$  no primeiro passo preserva a satisfabilidade. O segundo passo produz uma fórmula CNF  $\phi''$  que é algebricamente equivalente a  $\phi'$ . O terceiro passo produz uma fórmula 3-CNF  $\phi'''$  que é de fato equivalente a  $\phi''$ , pois qualquer atribuição às variáveis  $p$  e  $q$  produz uma fórmula que é algebricamente equivalente a  $\phi''$ .

Também devemos mostrar que a redução pode ser calculada em tempo polinomial. A construção de  $\phi$  a partir de  $\phi$  introduz no máximo 1 variável e 1 cláusula por conectivo em  $\phi$ . A construção de  $\phi''$  a partir de  $\phi'$  pode introduzir no máximo 8 cláusulas em  $\phi''$  para cada cláusula de  $\phi'$ , pois cada cláusula de  $\phi'$  tem no máximo 3 variáveis, e a tabela verdade para cada cláusula tem no máximo  $2^3 = 8$  linhas. A construção de  $\phi'''$  a partir de  $\phi''$  introduz no máximo 4 cláusulas em  $\phi'''$  para cada cláusula de  $\phi''$ . Portanto, o tamanho da fórmula resultante  $\phi'''$  é polinomial no comprimento da fórmula original. Cada uma das construções pode ser realizada com facilidade em tempo polinomial. ■

## Exercícios

### 34.4-1

Considere a redução direta (de tempo não polinomial) na prova do Teorema 34.9. Descreva um circuito de tamanho  $n$  que, quando convertido em uma fórmula por esse método, produza uma fórmula cujo tamanho seja exponencial em  $n$ .

#### 34.4-2

Mostre a fórmula 3-CNF que resulta quando usamos o método do Teorema 34.10 sobre a fórmula (34.3).

#### 34.4-3

O professor Jaime se propõe mostrar que  $\leq_p$ , usando apenas a técnica da tabela verdade na prova do Teorema 34.10, e não os outros passos. Ou seja, o professor propõe tomar a fórmula booleana  $\phi$ , formar uma tabela verdade para suas variáveis, derivar da tabela verdade uma fórmula em 3-DNF que seja equivalente a  $\neg\phi$ , e depois efetuar a negação e aplicar as leis de DeMorgan para produzir uma fórmula 3-CNF equivalente a  $\phi$ . Mostre que essa estratégia não produz uma redução de tempo polinomial.

#### 34.4-4

Mostre que o problema de determinar se uma fórmula booleana é uma tautologia é completo para co-NP. (Sugestão: Consulte o Exercício 34.3-7.)

#### 34.4-5

Mostre que o problema de determinar a satisfabilidade de fórmulas booleanas em forma normal disjuntiva pode ser resolvido em tempo polinomial.

#### 34.4-6

Suponha que alguém lhe dê um algoritmo de tempo polinomial para decidir a satisfabilidade de fórmulas. Descreva como usar esse algoritmo para encontrar atribuições satisfatórias em tempo polinomial.

#### 34.4-7

Seja 2-CNF-SAT o conjunto de fórmulas booleanas capazes de satisfação em CNF com exatamente 2 literais por cláusula. Mostre que 2-CNF-SAT  $\in P$ . Torne seu algoritmo tão eficiente quanto possível. (Sugestão: Observe que  $x \vee y$  é equivalente a  $\neg x \rightarrow y$ . Reduza 2-CNF-SAT a um problema sobre um grafo orientado que possa ser resolvido de modo eficiente.)

### 34.5 Problemas NP-completos

Os problemas NP-completos surgem em diversos domínios: lógica booleana, grafos, aritmética, projeto de rede, conjuntos e partições, armazenamento e recuperação, seqüenciamento e agendamento, programação matemática, álgebra e teoria dos números, jogos e quebra-cabeças, autômatos e teoria da linguagem, otimização de programas, biologia, química, física e muitos outros. Nesta seção, usaremos a metodologia de redução para fornecer provas do caráter NP-completo para uma variedade de problemas extraídos da teoria dos grafos e do particionamento de conjuntos.

A Figura 34.13 representa a estrutura das provas do caráter NP-completo nesta seção e na Seção 34.4. Cada linguagem na figura é provada NP-completa por redução da linguagem que aponta para ela. Na raiz encontra-se CIRCUIT-SAT, que provamos ser NP-completa no Teorema 34.7.

#### 34.5.1 O problema do grupo exclusivo

Um **grupo exclusivo** em um grafo não orientado  $G = (V, E)$  é um subconjunto  $V' \subseteq V$  de vértices, no qual cada par está conectado por uma aresta em  $E$ . Em outras palavras, um grupo exclusivo é um subgrafo completo de  $G$ . O **tamanho** de um grupo exclusivo é o número de vértices que ele contém. O **problema do grupo exclusivo** é o problema de otimização de encontrar um grupo exclusivo de tamanho máximo em um grafo. Como um problema de decisão, simplesmente perguntamos se um grupo exclusivo de um dado tamanho  $k$  existe no grafo. A definição formal é

$$\text{CLIQUE} = \{\langle G, k \rangle : G \text{ é um grafo com um grupo exclusivo de tamanho } k\}$$

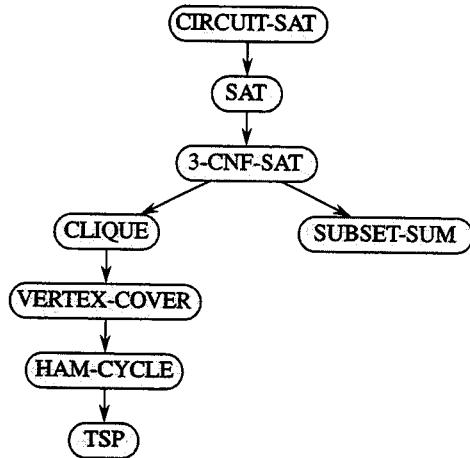


FIGURA 34.13 A estrutura de provas do caráter NP-completo nas Seções 34.4 e 34.5. Todas as provas devem correr em última análise por redução do caráter NP-completo de CIRCUIT-SAT

Um algoritmo simples para determinar se um grafo  $G = (V, E)$  com  $|V|$  vértices tem um grupo exclusivo de tamanho  $k$  é listar todos os  $k$  subconjuntos de  $V$  e conferir cada um para ver se ele forma um grupo exclusivo. O tempo de execução desse algoritmo é  $\Omega(k^2|\text{ver símbolo}|)$ , que é polinomial se  $k$  é uma constante. Porém, em geral  $k$  poderia estar próximo de  $|V|/2$ , e nesse caso, o algoritmo é executado em tempo superpolinomial. Como poderíamos suspeitar, é improvável que exista um algoritmo eficiente para o problema do grupo exclusivo.

#### **Teorema 34.11**

O problema do grupo exclusivo é NP-completo.

**Prova** Para mostrar que CLIQUE  $\in \text{NP}$ , para um dado grafo  $G = (V, E)$ , usamos o conjunto  $V' \subseteq V$  de vértices no grupo exclusivo como um certificado para  $G$ . A verificação se  $V'$  é um grupo exclusivo pode ser realizada em tempo polinomial verificando se, para todo par  $u, v \in V'$ , a aresta  $(u, v)$  pertence a  $E$ .

Em seguida, provamos que  $3\text{-CNF-SAT} \leq_p \text{CLIQUE}$ , o que mostra que o problema do grupo exclusivo é NP-difícil. É um tanto surpreendente que devamos ser capazes de provar esse resultado pois, na superfície, as fórmulas lógicas parecem ter pouca relação com grafos.

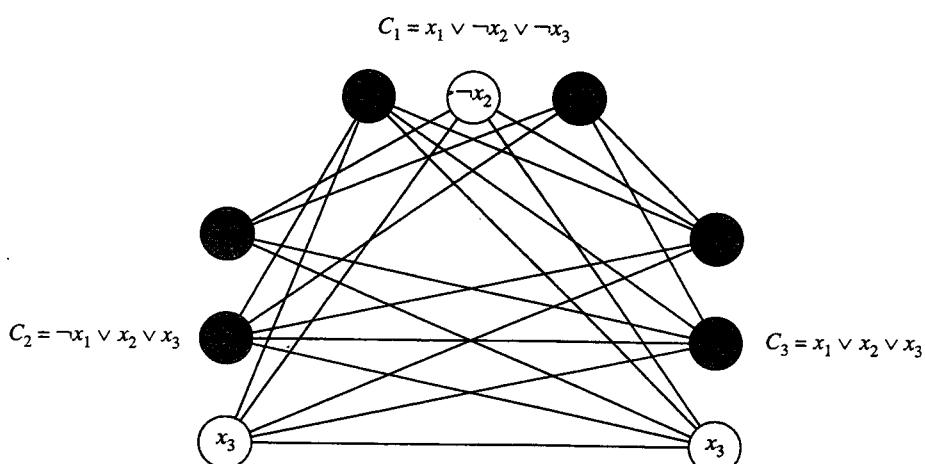


FIGURA 34.14 O grafo  $G$  derivado da fórmula  $3\text{-CNF } \phi = C_1 \wedge C_2 \wedge C_3$ , onde  $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$ ,  $C_2 = (\neg x_1 \vee x_2 \vee x_3)$  e  $C_3 = (x_1 \vee x_2 \vee x_3)$  na redução de  $3\text{-CNF-SAT}$  a  $\text{CLIQUE}$ . Uma atribuição satisfatória da fórmula é  $x_2 = 0$ ,  $x_3 = 1$ , e  $x_1$  pode ser 0 ou 1. Essa atribuição satisfaz a  $C_1$  com  $\neg x_2$  e satisfaz a  $C_2$  e  $C_3$  com  $x_3$ , correspondente ao grupo exclusivo com vértices ligeiramente sombreados

O algoritmo de redução começa com uma instância de 3-CNF-SAT. Seja  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_k$  uma fórmula booleana em 3-CNF com  $k$  cláusulas. Para  $r = 1, 2, \dots, k$ , cada cláusula  $C_r$  tem exatamente três literais distintos  $l'_1, l'_2$  e  $l'_3$ . Construiremos um grafo  $G$  tal que  $\phi$  seja capaz de satisfação se e somente se  $G$  tem um grupo exclusivo de tamanho  $k$ .

O grafo  $G = (V, E)$  é construído da maneira descrita a seguir. Para cada cláusula  $C_r = (l'_1 \vee l'_2 \vee l'_3)$  em  $\phi$ , inserimos uma tripla de vértices  $v'_1, v'_2$  e  $v'_3$  em  $V$ . Inserimos uma aresta entre dois vértices  $v'_r$  e  $v'_s$  se ambas as características a seguir se mantêm:

- $v'_r$  e  $v'_s$  estão em triplas diferentes, isto é,  $r \neq s$ , e
- seus literais correspondentes são coerentes, isto é,  $l'_r$  não é a negação de  $l'_s$ .

Esse grafo pode ser calculado com facilidade a partir de  $\phi$  em tempo polinomial. Como um exemplo dessa construção, se temos

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3),$$

então  $G$  é o grafo mostrado na Figura 34.14.

Devemos mostrar que essa transformação de  $\phi$  em  $G$  é uma redução. Primeiro, suponha que  $\phi$  tenha uma atribuição satisfatória. Então, cada cláusula  $C_r$  contém pelo menos um literal  $l'_i$  que recebe a atribuição de 1, e cada um desses literais corresponde a um vértice  $v'_i$ . Escolhendo-se um tal literal “verdadeiro” de cada cláusula, formamos um conjunto de  $V$  de  $k$  vértices. Afirmando que  $V$  é um grupo exclusivo. Para dois vértices quaisquer  $v'_r, v'_s \in V$ , onde  $r \neq s$ , ambos os literais correspondentes,  $l'_r$  e  $l'_s$ , são mapeados como 1 pela atribuição satisfatória dada, e assim os literais não podem ser complementos. Desse modo, pela construção de  $G$ , a aresta  $v'_r, v'_s$  pertence a  $E$ .

Reciprocamente, suponha que  $G$  tenha um grupo exclusivo  $V$  de tamanho  $k$ . Nenhuma aresta em  $G$  conecta vértices na mesma tripla, e assim  $V$  contém exatamente um vértice por tripla. Podemos atribuir 1 a cada literal  $l'_i$  tal que  $v'_i \in V$ , sem receio de atribuir 1 a um literal e a seu complemento, pois  $G$  não contém arestas entre literais incoerentes. Cada cláusula é satisfeita, e assim  $\phi$  é satisfeita. (Quaisquer variáveis que não correspondam a nenhum vértice no grupo exclusivo podem ser definidas de forma arbitrária.) ■

No exemplo da Figura 34.14, uma atribuição satisfatória de  $\phi$  é  $x_2 = 0$  e  $x_3 = 1$ . Um grupo exclusivo correspondente de tamanho  $k = 3$  consiste nos vértices que correspondem a  $\neg x_2$  da primeira cláusula,  $x_3$  da segunda cláusula e  $x_3$  da terceira cláusula. Como o grupo exclusivo não contém vértices correspondentes a  $x_1$  ou  $\neg x_1$ , podemos definir  $x_1$  como 0 ou 1 nessa atribuição satisfatória.

Observe que, na prova do Teorema 34.11, reduzimos uma instância arbitrária de 3-CNF-SAT a uma instância de CLIQUE com uma estrutura específica. Pode parecer que mostramos apenas que CLIQUE é NP-difícil em grafos nos quais os vértices são restritos a ocorrer em triplas e nos quais não há arestas entre vértices na mesma tripla. Realmente, mostramos que CLIQUE é NP-difícil apenas nesse caso restrito, mas essa prova basta para mostrar que CLIQUE é NP-difícil em grafos gerais. Por quê? Se tivéssemos um algoritmo de tempo polinomial que resolvesse CLIQUE em grafos gerais, ele também resolveria CLIQUE em grafos restritos.

Porém, não teria sido suficiente reduzir instâncias de 3-CNF-SAT com uma estrutura especial a instâncias gerais de CLIQUE. Por quê? Pode ter ocorrido que as instâncias de 3-CNF-SAT a partir das quais optamos por fazer a redução fossem “fáceis”, e assim não teríamos reduzido um problema NP-difícil a CLIQUE.

Observe também que a redução usou a instância de 3-CNF-SAT, mas não a solução. Teria sido um erro a redução de tempo polinomial ter se baseado no conhecimento de que a fórmula  $\phi$  é capaz de satisfação, pois não sabemos como descobrir essa informação em tempo polinomial.

### 34.5.2 O problema de cobertura de vértices

Uma **cobertura de vértices** de um grafo não orientado  $G = (V, E)$  é um subconjunto  $V' \subseteq V$  tal que  $(u, v) \in E$ , então  $u \in V'$  ou  $v \in V'$  (ou ambos). Isto é, cada vértice “cobre” suas arestas incidentes, e uma cobertura de vértices para  $G$  é um conjunto de vértices que cobre todas as arestas em  $E$ . O **tamanho** de uma cobertura de vértices é o número de vértices que ela contém. Por exemplo, o grafo na Figura 34.15(b) tem uma cobertura de vértices  $\{w, z\}$  de tamanho 2.

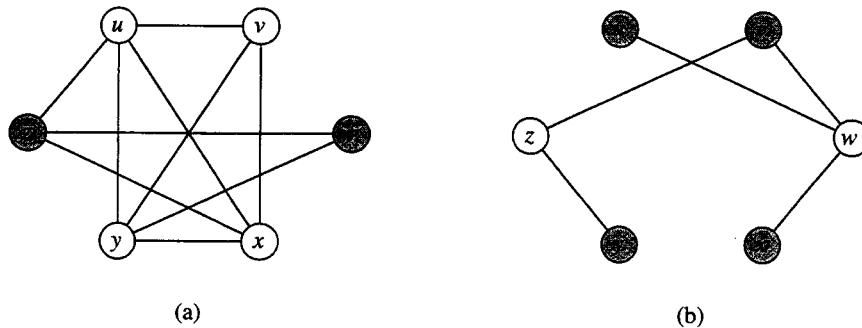


FIGURA 34.15 Redução de CLIQUE a VERTEX-COVER. (a) Um grafo não orientado  $G = (V, E)$  com grupo exclusivo  $V = \{u, v, x, y\}$ . (b) O grafo  $\bar{G}$  produzido pelo algoritmo de redução que tem cobertura de vértices  $V - V = \{w, z\}$

O **problema de cobertura de vértices** é o de encontrar uma cobertura de vértices de tamanho mínimo em um dado grafo. Enunciando novamente esse problema de otimização como um problema de decisão, desejamos determinar se um grafo tem uma cobertura de vértices de um dado tamanho  $k$ . Como uma linguagem, definimos

$$\text{VERTEX-COVER} = \{\langle G, k \rangle : \text{grafo } G \text{ tem uma cobertura de vértices de tamanho } k\}.$$

O teorema a seguir mostra que esse problema é NP-completo.

#### **Teorema 34.12**

O problema de cobertura de vértices é NP-completo.

**Prova** Primeiro mostramos que VERTEX-COVER ∈ NP. Vamos supor que temos um grafo  $G = (V, E)$  e um inteiro  $k$ . O certificado que escolhemos é a própria cobertura de vértices  $V' \subseteq V$ . O algoritmo de verificação afirma que  $|V'| = k$ , e então verifica, para cada aresta  $(u, v) \in E$ , que  $u \in V'$  ou  $v \in V'$ . Essa verificação pode ser realizada de forma direta em tempo polinomial.

Provamos que o problema de cobertura de vértices é NP-difícil mostrando que  $|V'| = k$ . Essa redução se baseia na noção de “complemento” de um grafo. Dado um grafo não orientado  $G = (V, E)$ , definimos o *complemento* de  $G$  como  $\bar{G} = (V, \bar{E})$ , onde  $\bar{E} = \{(u, v) : u, v \in V, u \neq v, \text{ e } (u, v) \notin E\}$ . Em outras palavras,  $\bar{G}$  é o grafo que contém exatamente as arestas que não estão em  $G$ . A Figura 34.15 mostra um grafo e seu complemento, e ilustra a redução de CLIQUE a VERTEX-COVER.

O algoritmo de redução toma como entrada uma instância  $\langle G, k \rangle$  do problema do grupo exclusivo. Ele calcula o complemento  $\bar{G}$ , o que é feito com facilidade em tempo polinomial. A saída do algoritmo de redução é a instância  $\langle \bar{G}, |V| - k \rangle$  do problema de cobertura de vértices. Para completar a prova, mostramos que essa transformação é de fato uma redução: o grafo  $G$  tem um grupo exclusivo de tamanho  $k$  se e somente se o grafo  $\bar{G}$  tem uma cobertura de vértices de tamanho  $|V| - k$ .

Suponha que  $G$  tenha um grupo exclusivo  $V' \subseteq V$  com  $|V'| = k$ . Afirmamos que  $V - V'$  é uma cobertura de vértices em  $\bar{G}$ . Seja  $(u, v)$  qualquer aresta em  $\bar{E}$ . Então,  $(u, v) \notin E$ , o que implica que

pelo menos um entre  $u$  e  $v$  não pertence a  $V'$ , pois todo par de vértices em  $V'$  é conectado por uma aresta de  $E$ . De modo equivalente, pelo menos um entre  $u$  e  $v$  está em  $V - V'$ , o que significa que a aresta  $(u, v)$  é coberta por  $V - V'$ . Tendo em vista que  $(u, v)$  foi escolhida arbitrariamente a partir de  $\bar{E}$ , toda aresta de  $\bar{E}$  é coberta por um vértice em  $V - V'$ . Conseqüentemente, o conjunto  $V - V'$ , que tem tamanho  $|V| - k$ , forma uma cobertura de vértices para [ver símbolo].

Reciprocamente, suponha que  $\bar{G}$  tem uma cobertura de vértices  $V' \subseteq V$ , onde  $|V'| = |V| - k$ . Então, para todo  $u, v \in V$ , se  $(u, v) \in \bar{E}$ , então  $u \in V'$  ou  $v \in V'$ , ou ambos. A contrapositiva dessa implicação é que, para todo  $u, v \in V$ , se  $u \notin V'$  e  $v \notin V'$ , então  $(u, v) \in E$ . Em outras palavras,  $V - V'$  é um grupo exclusivo, e ele tem tamanho  $|V| = |V'| - k$ . ■

Tendo em vista que VERTEX-COVER é NP-completo, não esperamos encontrar um algoritmo de tempo polinomial para localizar uma cobertura de vértices de tamanho mínimo. Porém, a Seção 35.1 apresenta um “algoritmo de aproximação” de tempo polinomial, que produz soluções “aproximadas” para o problema de cobertura de vértices. O tamanho de uma cobertura de vértices produzida pelo algoritmo é no máximo duas vezes o tamanho mínimo de uma cobertura de vértices.

Desse modo, não devemos deixar de ter esperança só porque um problema é NP-completo. Pode haver um algoritmo de aproximação de tempo polinomial que obtenha soluções aproximadas, embora a descoberta de uma solução ótima seja NP-completa. O Capítulo 35 fornece vários algoritmos de aproximação para problemas NP-completos.

### 34.5.3 O problema do ciclo hamiltoniano

Voltamos agora ao problema do ciclo hamiltoniano definido na Seção 34.2.

#### **Teorema 34.13**

O problema do ciclo hamiltoniano é NP-completo.

*Prova* Primeiro mostramos que HAM-CYCLE pertence a NP. Dado um grafo  $G = (V, E)$ , nosso certificado é a seqüência de vértices  $|V|$  que forma o ciclo hamiltoniano. O algoritmo de verificação confere se essa seqüência contém cada vértice em  $V$  exatamente uma vez e se, com o primeiro vértice repetido no final, ela forma um ciclo em  $G$ . Ou seja, ele verifica se existe uma aresta entre cada par de vértices consecutivos e entre o primeiro e o último vértice. Essa verificação pode ser executada em tempo polinomial.

Agora provamos que VERTEX-COVER  $\leq_p$  HAM-CYCLE, o que mostra que HAM-CYCLE é NP-completo. Dado um grafo não orientado  $G = (V, E)$  e um inteiro  $k$ , construímos um grafo não orientado  $G' = (V', E')$  que tem um ciclo hamiltoniano se e somente se  $G$  tem uma cobertura de vértices de tamanho  $k$ .

Nossa construção é baseada em um *dispositivo*, que é um fragmento de um grafo que impõe certas propriedades. A Figura 34.16(a) mostra o dispositivo que empregaremos. Para cada aresta  $(u, v) \in E$ , o grafo  $G'$  que construímos conterá uma cópia desse dispositivo, que denotamos por  $W_{uv}$ . Denotamos cada vértice em  $W_{uv}$  por  $[u, v, i]$  ou  $[v, u, i]$ , onde  $1 \leq i \leq 6$ , de forma que cada dispositivo  $W_{uv}$  contém 12 vértices. O dispositivo  $W_{uv}$  também contém as 14 arestas mostradas na Figura 34.16(a).

Junto com a estrutura interna do dispositivo, impomos as propriedades que queremos, limitando as conexões entre o dispositivo e o restante do grafo  $G'$  que construímos. Em particular, apenas os vértices  $[u, v, 1], [u, v, 6], [v, u, 1]$  e  $[v, u, 6]$  terão arestas incidentes do lado de fora de  $W_{uv}$ . Qualquer ciclo hamiltoniano de  $G'$  terá de percorrer as arestas de  $W_{uv}$  em um dos três modos mostrados nas Figuras 34.16(b)-(d). Se o ciclo passar pelo vértice  $[u, v, 1]$ , ele deve sair pelo vértice  $[u, v, 6]$  e visitar todos os 12 vértices do dispositivo (Figura 34.16(b)) ou os seis vértices de  $[u, v, 1]$  a  $[u, v, 6]$  (Figura 34.16(c)). Nesse último caso, o ciclo terá de entrar novamente no dispositivo para visitar os vértices  $[v, u, 1]$  a  $[v, u, 6]$ . De modo semelhante, se o ciclo entrar pelo vértice  $[v, u, 1]$ , ele deverá sair pelo vértice  $[v, u, 6]$  e visitar todos os 12 vértices do dispositivo | 795

(Figura 34.16(d)) ou os seis vértices de  $[v, u, 1]$  a  $[v, u, 6]$  (Figura 34.16(c)). Não é possível nenhum outro caminho pelo dispositivo que visite todos os 12 vértices. Em particular, é impossível construir dois caminhos de vértices disjuntos, um dos quais conecte  $[u, v, 1]$  a  $[v, u, 6]$ , e o outro conecte  $[v, u, 1]$  a  $[u, v, 6]$ , tais que a união dos dois caminhos contenha todos os vértices do dispositivo.

Os únicos vértices em  $V'$  além daqueles de dispositivos são *vértices seletores*  $s_1, s_2, \dots, s_k$ . Usamos arestas incidentes em vértices seletores de  $G'$  para selecionar os  $k$  vértices da cobertura em  $G$ .

Além das arestas em dispositivos, existem dois outros tipos de arestas em  $E'$ , que a Figura 34.17 mostra. Primeiro, para cada vértice  $u \in V$ , adicionamos arestas para unir pares de dispositivos, a fim de formar um caminho contendo todos os dispositivos que correspondem a arestas incidentes sobre  $u$  em  $G$ . Ordenamos arbitrariamente os vértices adjacentes a cada vértice  $u \in V$  como  $u^{(1)}, u^{(2)}, \dots, u^{(\text{grau}(u))}$ , onde  $(\text{grau}(u))$  é o número de vértices adjacentes a  $u$ . Criamos um caminho em  $G'$  passando por todos os dispositivos que correspondem a arestas incidentes em  $u$  adicionando a  $E'$  as arestas  $\{([u, u^{(i)}, 6], [u, u^{(i+1)}, 1]) : 1 \leq i \leq \text{grau}(u) - 1\}$ . Por exemplo, na Figura 34.17, ordenamos os vértices adjacentes a  $w$  como  $x, y, z$ , e assim o grafo  $G'$  da parte (b) da figura inclui as arestas  $([w, x, 6], [w, y, 1])$  e  $([w, y, 6], [w, z, 1])$ . Para cada vértice  $u \in V$ , essas arestas em  $G'$  completam um caminho que contém todos os dispositivos que correspondem a arestas incidentes sobre  $u$  em  $G$ .

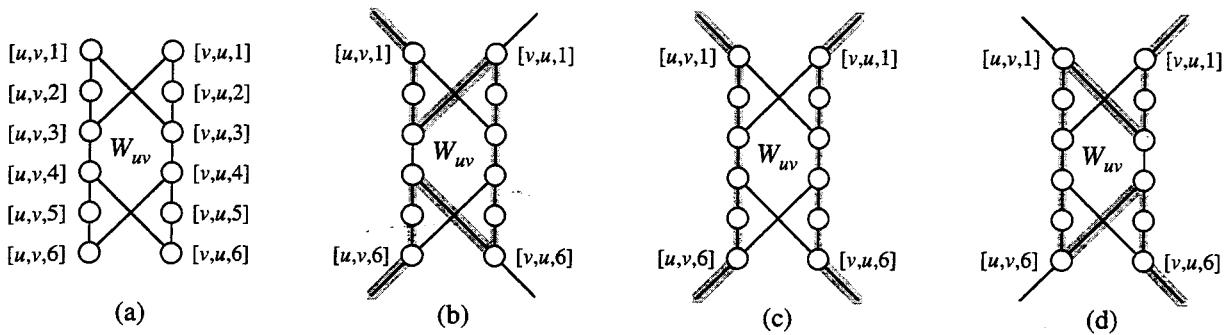


FIGURA 34.16 O dispositivo usado para reduzir o problema de cobertura de vértices ao problema de ciclo hamiltoniano. Uma aresta  $(u, v)$  do grafo  $G$  corresponde ao dispositivo  $W_{uv}$  no grafo  $G'$  criado na redução. (a) O dispositivo, com vértices individuais identificados. (b)–(d) Os caminhos sombreados são os únicos possíveis pelo dispositivo que incluem todos os vértices, supondo que as únicas conexões do dispositivo ao restante de  $G'$  são realizadas pelos vértices  $[u, v, 1], [u, v, 6], [v, u, 1]$  e  $[v, u, 6]$

A intuição por trás dessas arestas é que, se escolhermos um vértice  $u \in V$  na cobertura de vértices de  $G$ , podemos construir um caminho de  $[u, u^{(1)}, 1]$  até  $[u, u^{(\text{grau}(u))}, 6]$  em  $G'$  que “cobre” todos os dispositivos que correspondem a arestas incidentes em  $u$ . Isto é, para cada um desses dispositivos, digamos  $W_{u, u^{(i)}}$ , o caminho inclui todos os 12 vértices (se  $u$  está na cobertura de vértices mas  $u^{(i)}$  não está) ou apenas os seis vértices  $[u, u^{(i)}, 1], [u, u^{(i)}, 2], \dots, [u, u^{(i)}, 6]$  (se  $u$  e  $u^{(i)}$  estão ambos na cobertura de vértices).

O último tipo de aresta em  $E'$  une o primeiro vértice  $[u, u^{(1)}, 1]$  e o último vértice  $[u, u^{(\text{grau}(u))}, 6]$  de cada um desses caminhos a cada um dos vértices seletores. Isto é, incluímos as arestas

$$\begin{aligned} & \{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ e } 1 \leq j \leq k\} \\ & \cup \{(s_j, [u, u^{(\text{grau}(u))}, 6]) : u \in V \text{ e } 1 \leq j \leq k\} . \end{aligned}$$

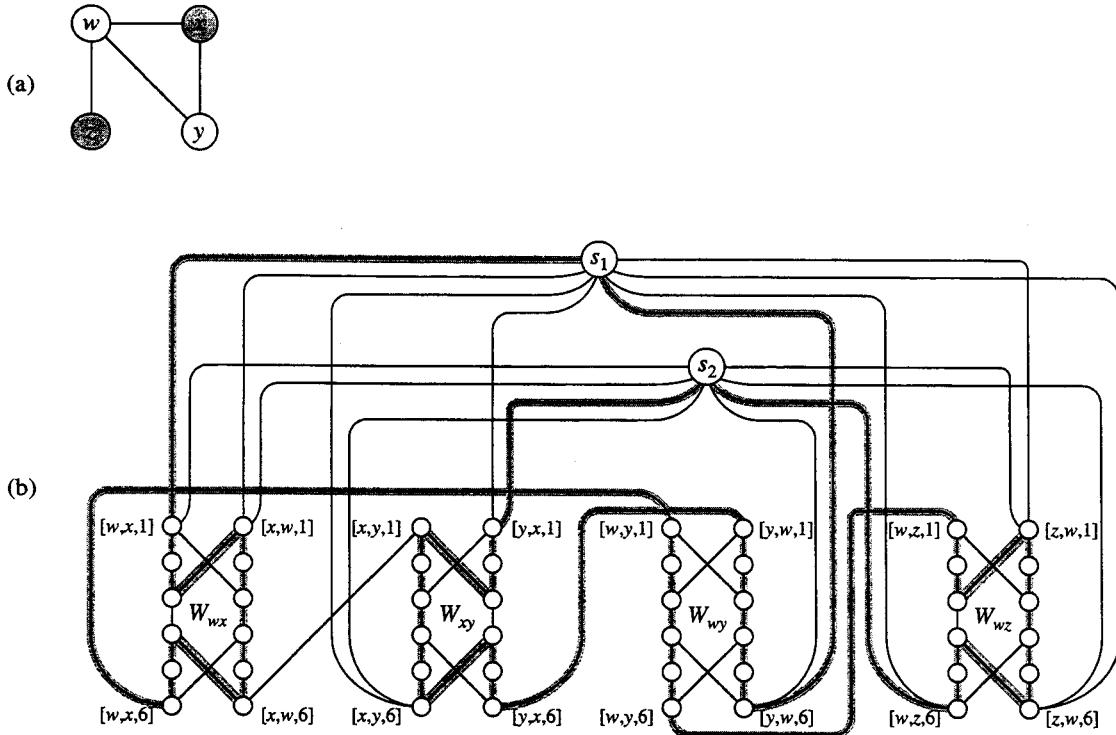


FIGURA 34.17 Redução de uma instância do problema de cobertura de vértices a uma instância do problema de ciclo hamiltoniano. (a) Um grafo não orientado  $G$  com uma cobertura de vértices de tamanho 2, consistindo nos vértices levemente sombreados  $w$  e  $y$ . (b) O grafo não orientado  $G'$  produzido pela redução, com o caminho hamiltoniano correspondendo à cobertura de vértices sombreada. A cobertura de vértices  $\{w, y\}$  corresponde às arestas  $(s_1, [w, x, 1])$  e  $(s_2, [y, x, 1])$  que aparecem no ciclo hamiltoniano

Em seguida, mostramos que o tamanho de  $G'$  é polinomial no tamanho de  $G$  e, consequentemente, podemos construir  $G'$  em tempo polinomial no tamanho de  $G$ . Os vértices de  $G'$  são os dos dispositivos, mais os vértices seletores. Cada dispositivo contém 12 vértices e existem  $k \leq |V|$  vértices seletores, o que dá um total de

$$\begin{aligned} |V'| &= 12 |E| + k \\ &\leq 12 |E| + |V| \end{aligned}$$

vértices. As arestas de  $G'$  são as dos dispositivos, aquelas que ficam entre dispositivos e aquelas que conectam vértices seletores a dispositivos. Existem 14 arestas em cada dispositivo ou  $14 |E|$  em todos os dispositivos. Para cada vértice  $u \in V$ , existem  $\text{grau}(u) - 1$  arestas entre dispositivos, de forma que, somadas sobre todos os vértices em  $V$ , existem

$$\sum_{u \in V} (\text{grau}(u) - 1) = 2 |E| - |V|$$

arestas entre dispositivos. Finalmente, há duas arestas para cada par que consiste em um vértice seletor e um vértice de  $V$ , ou  $2k |V|$  dessas arestas. O número total de arestas de  $G'$  é então

$$\begin{aligned} |E'| &= (14 |E|) + (2 |E| - |V|) + (2k |V|) \\ &= 16 |E| + (2k - 1) |V| \\ &\leq 16 |E| + (2 |V| - 1) |V|. \end{aligned}$$

Agora mostramos que a transformação do grafo  $G$  em  $G'$  é uma redução. Isto é, devemos mostrar que  $G$  tem uma cobertura de vértices de tamanho  $k$  se e somente se  $G'$  tem um ciclo hamiltoniano.

Suponha que  $G = (V, E)$  tem uma cobertura de vértices  $V^* \subseteq V$  de tamanho  $k$ . Seja  $V^* = \{u_1, u_2, \dots, u_k\}$ . Como vemos na Figura 34.17, formamos um ciclo hamiltoniano em  $G$  incluindo as arestas a seguir<sup>8</sup> para cada vértice  $u_j \in V^*$ . Incluímos as arestas  $\{([u_j, u_j^{(i)}, 6], [u_j, u_j^{(i+1)}, 1]) : 1 \leq i \leq \text{grau}(u_j)\}$ , que conectam todos os dispositivos que correspondem a arestas incidentes em  $u_j$ . Também incluímos as arestas contidas no interior desses dispositivos como mostram as Figuras 34.16(b)–(d), dependendo do fato de a aresta estar coberta por um ou dois vértices em  $V^*$ . O ciclo hamiltoniano também inclui as arestas

$$\begin{aligned} & \{(s_j, [u_j, u_j^{(i)}, 1]) : 1 \leq j \leq k\} \\ & \cup \{(s_{j+1}, [u_j, u_j^{(\text{grau}(u_j))}, 6]) : 1 \leq j \leq k-1\} \\ & \cup \{(s_1, [u_k, u_k^{(\text{grau}(u_k))}, 6])\}. \end{aligned}$$

Examinando a Figura 34.17, o leitor pode verificar que essas arestas formam um ciclo. O ciclo começa em  $s_1$ , visita todos os dispositivos que correspondem a arestas incidentes em  $u_1$ , depois visita  $s_2$ , visita todos os dispositivos que correspondem a arestas incidentes em  $u_2$  e assim por diante, até retornar a  $s_1$ . Cada dispositivo é visitado uma ou duas vezes, dependendo de um ou dois vértices de  $V^*$  cobrir(em) sua aresta correspondente. Como  $V^*$  é uma cobertura de vértices para  $G$ , cada aresta em  $E$  é incidente em algum vértice de  $V^*$ , e então o ciclo visita cada vértice em cada dispositivo de  $G'$ . Tendo em vista que o ciclo também visita todo vértice seletor, ele é hamiltoniano.

Reciprocamente, suponha que  $G' = (V', E')$  tenha um ciclo hamiltoniano  $C \subseteq E'$ . Afirmamos que o conjunto

$$V^* = \{u \in V : (s_j, [u, u^{(1)}, 1]) \in C \text{ para algum } 1 \leq j \leq k\} \quad (34.4)$$

é uma cobertura de vértices para  $G$ . Para ver por que, particione  $C$  em caminhos máximos que comecem em algum vértice seletor  $s_i$ , percorram uma aresta  $(s_i, [u, u^{(1)}, 1])$  para algum  $u \in V$  e terminem em um vértice seletor  $s_j$  sem passar por qualquer outro vértice seletor. Vamos chamar cada caminho um “caminho de cobertura”. A partir da forma como  $G'$  é construído, cada caminho de cobertura deve começar em algum  $s_i$ , seguir a aresta  $(s_i, [u, u^{(1)}, 1])$  correspondente a algum vértice  $u \in V$ , passar por todos os dispositivos que correspondem a arestas de  $E$  incidentes em  $u$ , e depois terminar em algum vértice seletor  $s_j$ . Fazemos referência a esse caminho de cobertura como  $p_u$  e, pela equação (34.4), inserimos  $u$  em  $V^*$ . Cada dispositivo visitado por  $p_u$  deve ser  $W_{uv}$  ou  $W_{vu}$  para algum  $v \in V$ . Para cada dispositivo visitado por  $p_u$ , seus vértices são visitados por um ou dois caminhos de cobertura. Se eles forem visitados por um caminho de cobertura, então a aresta  $(u, v) \in E$  é coberta em  $G$  pelo vértice  $u$ . Se dois caminhos de cobertura visitam o dispositivo, então o outro caminho de cobertura deve ser  $p_v$ , o que implica que  $v \in V^*$ , e a aresta  $(u, v) \in E$  é coberta por  $u$  e por  $v$ . Pelo fato de cada vértice em cada dispositivo ser visitado por algum caminho de cobertura, vemos que cada aresta em  $E$  é coberta por algum vértice em  $V^*$ . ■

<sup>8</sup> Tecnicamente, definimos um ciclo em termos de vértices em lugar de arestas (veja a Seção B.4). No interesse da clareza, abusamos da notação aqui e definimos o ciclo hamiltoniano em termos de arestas.

### 34.5.4 O problema do caixeiro-viajante

No **problema do caixeiro-viajante**, que está intimamente relacionado ao problema do ciclo hamiltoniano, um vendedor deve visitar  $n$  cidades. Modelando o problema como um grafo completo com  $n$  vértices, podemos dizer que o vendedor deseja fazer um *viagem* (uma *excursão*), ou um ciclo hamiltoniano, visitando cada cidade exatamente uma vez e terminando na cidade de onde partiu. Existe um custo inteiro  $c(i,j)$  para viajar da cidade  $i$  para a cidade  $j$ , e o vendedor deseja fazer a viagem cujo custo total seja mínimo, onde o custo total é a soma dos custos individuais ao longo das arestas da viagem. Por exemplo, na Figura 34.18, uma viagem de custo mínimo é  $\langle u, w, x, u \rangle$ , com custo 7. A linguagem formal para o problema de decisão correspondente é

$$\begin{aligned} \text{TSP} = \{ & \langle G, c, k \rangle : G = (V, E) \text{ é um grafo completo,} \\ & c \text{ é uma função de } V \times V \rightarrow \mathbb{Z}, \\ & k \in \mathbb{Z}, \text{ e} \\ & G \text{ tem uma viagem de caixeiro-viajante com custo no máximo igual a } k \} . \end{aligned}$$

O teorema a seguir mostra que é improvável existir um algoritmo rápido para o problema do caixeiro-viajante.

#### **Teorema 34.15**

O problema do caixeiro-viajante é NP-completo.

**Prova** Primeiro, mostraremos que TSP pertence a NP. Dada uma instância do problema, usaremos como certificado a seqüência de  $n$  vértices na viagem. O algoritmo de verificação confirma que essa seqüência contém cada vértice exatamente uma vez, totaliza os custos de arestas e verifica se a soma é no máximo  $k$ . Esse processo pode certamente ser feito em tempo polinomial.

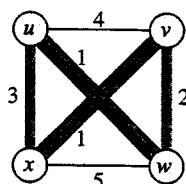


FIGURA 34.18 Uma instância do problema do caixeiro-viajante. As arestas sombreadas representam uma viagem de custo mínimo, com custo 7

Para provar que TSP é NP-difícil, mostramos que HAM-CYCLE  $\leq_p$  TSP. Seja  $G = (V, E)$  uma instância de HAM-CYCLE. Construímos uma instância de TSP da maneira descrita a seguir. Formamos o grafo completo  $G' = (V, E')$ , onde  $E' = \{(i, j) : i, j \in V\}$  e  $i \neq j\}$  definimos a função custo  $c$  como

$$c(i, j) = \begin{cases} 0 & \text{se } (i, j) \in E, \\ 1 & \text{se } (i, j) \notin E. \end{cases}$$

(Note que, como  $G$  é não orientado, ele não tem autoloops, e então  $c(v, v) = 1$  para todos os vértices  $v \in V$ .) A instância de TSP é então  $(G', c, 0)$ , que é facilmente formada em tempo polinomial.

Agora, mostraremos que o grafo  $G$  tem um ciclo hamiltoniano se e somente se o grafo  $G'$  tem um viagem de custo máximo 0. Suponha que o grafo  $G$  tem um ciclo hamiltoniano  $b$ . Cada aresta em  $b$  pertence a  $E$  e, portanto, tem custo 0 em  $G'$ . Desse modo,  $b$  é uma viagem em  $G'$  com custo 0. Reciprocamente, suponhamos que o grafo  $G'$  tem um viagem  $b'$  de custo máximo 0. Tendo em vista que os custos das arestas em  $E'$  são 0 e 1, o custo da viagem  $b'$  é exatamente 0, e cada aresta na viagem deve ter custo 0. Então,  $b'$  contém apenas arestas em  $E$ . Concluímos que  $b$  é um ciclo hamiltoniano no grafo  $G$ .

### 34.5.5 O problema da soma de subconjuntos

O próximo problema NP-completo que consideraremos é aritmético. No **problema de soma de subconjuntos**, temos um conjunto finito  $S \subset \mathbb{N}$  e um **destino**  $t \in \mathbb{N}$ . Perguntamos se existe um subconjunto  $S' \subseteq S$  cujos elementos tenham a soma  $t$ . Por exemplo, se  $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$  e  $t = 138457$ , então o subconjunto  $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$  é uma solução.

Como de hábito, definimos o problema como uma linguagem:

$$\text{SUBSET-SUM} = \{\langle S, t \rangle : \text{existe um subconjunto } S' \subseteq S \text{ tal que } t = \sum_{s \in S'} s\}.$$

Como ocorre com qualquer problema de aritmética, é importante recordar que nossa codificação padrão pressupõe que os inteiros da entrada estão codificados em binário. Com essa hipótese em mente, podemos mostrar que é improvável que o problema da soma de subconjuntos tenha um algoritmo rápido.

#### Teorema 34.15

O problema da soma de subconjuntos é NP-completo.

**Prova** Para mostrar que SUBSET-SUM está em NP, para uma instância  $\langle S, t \rangle$  do problema, seja o subconjunto  $S'$  o certificado. É possível verificar se  $t = \sum_{s \in S'} s$  por meio de um algoritmo de verificação em tempo polinomial.

Agora mostraremos que 3-CNF-SAT  $\leq_p$  SUBSET-SUM. Dada uma fórmula de 3-CNF  $\phi$  sobre as variáveis  $x_1, x_2, \dots, x_n$  com as cláusulas  $C_1, C_2, \dots, C_k$ , cada uma contendo exatamente três literais distintos, o algoritmo de redução constrói uma instância  $\langle S, t \rangle$  do problema da soma de subconjuntos, tal que  $\phi$  é capaz de satisfação se e somente se existe um subconjunto de  $S$  cuja soma é exatamente  $t$ . Sem perda de generalidade, fazemos duas suposições simplificadoras sobre a fórmula  $\phi$ . Primeiro, nenhuma cláusula contém ao mesmo tempo uma variável e sua negação, visto que tal cláusula é automaticamente satisfeita por qualquer atribuição de valores para as variáveis. Segundo, cada variável aparece em pelo menos uma cláusula pois, do contrário, não importa que valor é atribuído à variável.

A redução cria dois números no conjunto  $S$  para cada variável  $x_i$  e dois números em  $S$  para cada cláusula  $C_j$ . Criaremos números na base 10, onde cada número contém  $n + k$  dígitos e cada dígito corresponde a uma variável ou a uma cláusula. A base 10 (e outras bases, como veremos) tem a propriedade, de que necessitamos, de impedir transportes de dígitos mais baixos para dígitos mais altos.

Como mostra a Figura 34.19, construímos o conjunto  $S$  e o destino  $t$  como a seguir. Identificamos cada posição de dígito por uma variável ou por uma cláusula. Os  $k$  dígitos menos significativos são identificados pelas cláusulas e os  $n$  dígitos mais significativos são identificados por variáveis.

- O destino  $t$  tem um valor 1 em cada dígito identificado por uma variável, e um 4 em cada dígito identificado por uma cláusula.
- Para cada variável  $x_i$ , existem dois inteiros,  $v_i$  e  $v'_i$ , em  $S$ . Cada um tem o valor 1 no dígito identificado por  $x_i$ , e valores 0 nos outros dígitos de variáveis. Se o literal  $x_i$  aparece na cláusula  $C_j$ , então o dígito identificado por  $C_j$  em  $v_i$  contém um valor 1. Se o literal  $-x_i$  aparece na cláusula  $C_j$ , então o dígito identificado por  $C_j$  em  $v'_i$  contém um valor 1. Todos os outros dígitos identificados por cláusulas em  $v_i$  e  $v'_i$  são 0.

Todos os valores  $v_i$  e  $v'_i$  no conjunto  $S$  são distintos. Por quê? Para  $i \neq l$ , nenhum valor  $v_i$  ou  $v'_i$  pode igualar  $v_l$  e  $v'_l$  nos  $n$  dígitos mais significativos. Além disso, por nossas hipóteses simplificadoras anteriores, nenhum  $v_i$  e  $v'_i$  pode ser igual em todos os  $k$  dígitos menos significativos. Se  $v_i$  e  $v'_i$  fossem iguais, então  $x_i$  e  $-x_i$  teriam de aparecer exatamente no mesmo conjunto de cláusulas.

las. Contudo, supomos que nenhuma cláusula contém  $x_i$  e  $\neg x_i$  ao mesmo tempo, e que  $x_i$  ou  $\neg x_i$  aparece em alguma cláusula, e assim deve haver alguma cláusula  $C_j$  para a qual  $v_i$  e  $v'_i$  diferem.

- Para cada cláusula  $C_j$ , existem dois inteiros,  $s_j$  e  $s'_j$ , em  $S$ . Cada um tem valores 0 em todos os dígitos além do dígito identificado por  $C_j$ . Para  $s_j$ , existe um valor 1 no dígito  $C_j$ , e  $s'_j$  tem o valor 2 nesse dígito. Esses inteiros são “variáveis relaxadas”, que usamos para adicionar cada posição de dígito identificada por cláusula ao valor de destino 4.

O simples exame da Figura 34.19 demonstra que todos os valores  $s_j$  e  $s'_j$  em  $S$  são exclusivos no conjunto  $S$ .

|        | $x_1$ | $x_2$ | $x_3$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
|--------|-------|-------|-------|-------|-------|-------|-------|
| $v'_1$ | 1     | 0     | 0     | 0     | 1     | 1     | 0     |
| $v'_2$ | 0     | 1     | 0     | 1     | 1     | 1     | 0     |
| $v_3$  | 0     | 0     | 1     | 0     | 0     | 1     | 1     |
| $s_1$  | 0     | 0     | 0     | 1     | 0     | 0     | 0     |
| $s'_1$ | 0     | 0     | 0     | 2     | 0     | 0     | 0     |
| $s'_2$ | 0     | 0     | 0     | 0     | 2     | 0     | 0     |
| $s_3$  | 0     | 0     | 0     | 0     | 0     | 1     | 0     |
| $s_4$  | 0     | 0     | 0     | 0     | 0     | 0     | 1     |
| $s'_4$ | 0     | 0     | 0     | 0     | 0     | 0     | 2     |
| $t$    | 1     | 1     | 1     | 4     | 4     | 4     | 4     |

FIGURA 34.19 Redução de 3-CNF-SAT a SUBSET-SUM. A fórmula em 3-CNF é  $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ , onde  $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$ ,  $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$  e  $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$  e  $C_4 = (x_1 \vee x_2 \vee x_3)$ . Uma atribuição satisfatória de  $\phi$  é  $(x_1 = 0, x_2 = 0, x_3 = 1)$ . O conjunto  $S$  produzido pela redução consiste nos números de base 10 mostrados; de cima para baixo,  $S = \{1001001, 1000110, 1000001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$ . O destino  $t$  é 1114444. O subconjunto  $S' \subseteq S$  está ligeiramente sombreado e contém  $v'_1, v'_2$ , e  $v_3$ , que correspondem à atribuição satisfatória. Ele também contém variáveis relaxadas  $s_1, s'_1, s'_2, s_3, s_4$  e  $s'_4$  para alcançar o valor de destino 4 nos dígitos identificados por  $C_1$  a  $C_4$ .

Observe que a maior soma de dígitos em qualquer posição de dígito é 6, que ocorre nos dígitos identificados por cláusulas (três valores 1 a partir dos valores  $v_i$  e  $v'_i$ , mais os valores 1 e 2 de  $s_j$  e  $s'_j$ ). Então, interpretando esses números na base 10, não pode ocorrer nenhum transporte de dígitos mais baixos para dígitos mais altos.<sup>9</sup>

A redução pode ser executada em tempo polinomial. O conjunto  $S$  contém  $2n + 2k$  valores, cada um dos quais tem  $n + k$  dígitos, e o tempo para produzir cada dígito é polinomial em  $n + k$ . O destino  $t$  tem  $n + k$  dígitos, e a redução produz cada um deles em tempo constante.

Agora mostramos que a fórmula de 3-CNF  $\phi$  é capaz de satisfação se e somente se existe um subconjunto  $S' \subseteq S$  cuja soma é  $t$ . Primeiro, suponha que  $\phi$  tenha uma atribuição satisfatória. Para  $i = 1, 2, \dots, n$ , se  $x_i = 1$  nessa atribuição, então incluímos  $v_i$  em  $S'$ . Caso contrário, incluímos  $v'_i$ . Em outras palavras, incluímos em  $S'$  exatamente os valores  $v_i$  e  $v'_i$  que correspondem a literais com o valor 1 na atribuição satisfatória. Tendo incluído  $v_i$  ou  $v'_i$ , mas não ambos, para todo  $i$ , e tendo inserido 0 nos dígitos identificados por variáveis em todo  $s_j$  e  $s'_j$ , vemos que para cada dígito identificado por variável, a soma dos valores de  $S'$  deve ser 1, o que corresponde aos dígitos do

<sup>9</sup> De fato, qualquer base  $b$ , onde  $b \geq 7$ , iria funcionar. A instância no início desta subseção é o conjunto  $S$  e o destino  $t$  da Figura 34.19 interpretado na base 7, com  $S$  listado em seqüência ordenada.

destino  $t$ . Pelo fato de cada cláusula ser satisfeita, existe algum literal na cláusula com o valor 1. Então, cada dígito identificado por uma cláusula tem pelo menos um valor 1 em sua soma fornecido por um valor  $v_i$  ou  $v'_i$  em  $S'$ . De fato, 1, 2 ou 3 literais podem ter 1 em cada cláusula, e assim cada dígito identificado por cláusula tem a soma de 1, 2 ou 3 dos valores  $v_i$  e  $v'_i$  em  $S'$ . (Por exemplo, na Figura 34.19, os literais  $\neg x_1$ ,  $\neg x_2$  e  $x_3$  têm o valor 1 em uma atribuição satisfatória. Cada uma das cláusulas  $C_1$  e  $C_4$  contém exatamente um desses literais, e assim  $v'_1$ ,  $v'_2$  e  $v_3$  contribuem juntos com 1 para a soma nos dígitos de  $C_1$  e  $C_4$ . A cláusula  $C_2$  contém dois desses literais e  $v'_1$ ,  $v'_2$  e  $v_3$  contribuem com 2 para a soma no dígito correspondente a  $C_2$ . A cláusula  $C_3$  contém todos esses três literais,  $v'_1$ ,  $v'_2$  e  $v_3$  contribuem com 3 para a soma no dígito correspondente a  $C_3$ .) Chegamos ao destino de 4 em cada dígito identificado pela cláusula  $C_j$  incluindo em  $S'$  o subconjunto não vazio apropriado de variáveis relaxadas  $\{s_j, s'_j\}$ . (Na Figura 34.19,  $S'$  inclui  $s_1, v'_1, v'_2, s_3, s_4$  e  $v'_4$ .) Tendo em vista que equiparamos o destino em todos os dígitos da soma e não pode ocorrer nenhum transporte, a soma dos valores de  $S'$  é  $t$ .

Agora, suponha que exista um subconjunto  $S' \subseteq S$  que seja somado a  $t$ . O subconjunto  $S'$  deve incluir exatamente um dentre  $v_i$  e  $v'_i$  para cada  $i = 1, 2, \dots, n$  pois, do contrário, os dígitos identificados por variáveis não somariam 1. Se  $v_i \in S'$ , definimos  $x_i = 1$ . Caso contrário,  $v'_i \in S'$  e definimos  $x_i = 0$ . Afirmamos que toda cláusula  $C_j$ , para  $j = 1, 2, \dots, k$ , é satisfeita por essa atribuição. Para provar essa afirmação, observe que, para alcançar a soma 4 no dígito identificado por  $C_j$ , o subconjunto  $S'$  deve incluir pelo menos um valor  $v_i$  ou  $v'_i$  que tenha 1 no dígito identificado por  $C_j$ , pois as contribuições das variáveis relaxadas  $s_j$  e  $s'_j$  juntas somam no máximo 3. Se  $S'$  incluir um  $v_i$  que tenha 1 nessa posição, então o literal  $x_i$  aparecerá na cláusula  $C_j$ . Como definimos  $x_i = 1$  quando  $v_i \in S'$ , a cláusula  $C_j$  é satisfeita. Se  $S'$  incluir um  $v'_i$  que tenha 1 nessa posição, então o literal  $\neg x_i$  aparecerá em  $C_j$ . Como definimos  $x_i = 0$  quando  $v'_i \in S'$ , a cláusula  $C_j$  é novamente satisfeita. Desse modo, todas as cláusulas de  $\phi$  são satisfeitas, o que completa a prova. ■

## Exercícios

### 34.5-1

O **problema de isomorfismo de subgrafos** toma dois grafos  $G_1$  e  $G_2$  e pergunta se  $G_1$  é isomórfico para um subgrafo de  $G_2$ . Mostre que o problema de isomorfismo de subgrafos é NP-completo.

### 34.5-2

Dada uma matriz  $A$   $m \times n$  de inteiros e um vetor  $b$  de  $m$  inteiros, o **problema de programação de inteiros 0-1** pergunta se existe um vetor  $x$  de  $n$  inteiros com elementos no conjunto  $\{0, 1\}$  tal que  $Ax \leq b$ . Prove que a programação de inteiros 0-1 é NP-completa. (Sugestão: Reduza a partir de 3-CNF-SAT.)

### 34.5-3

O **problema de programação linear de inteiros** é semelhante ao problema de programação de inteiros 0-1 dado no Exercício 34.5-2, a não ser pelo fato de que os valores do vetor  $x$  podem ser quaisquer inteiros em vez de apenas 0 ou 1. Supondo que o problema de programação de inteiros 0-1 seja NP-difícil, mostre que o problema de programação linear de inteiros é NP-completo.

### 34.5-4

Mostre que o problema de soma de subconjuntos pode ser resolvido em tempo polinomial se o valor de destino  $t$  é expresso em unário.

### 34.5-5

O **problema de partição de conjuntos** toma como entrada um conjunto  $S$  de números. A questão é se os números podem ser particionados em dois conjuntos  $A$  e  $\bar{A} = S - A$  tais que  $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$ . Mostre que o problema de partição de conjuntos é NP-completo.

### 34.5-6

Mostre que o problema do caminho hamiltoniano é NP-completo.

### 34.5-7

O **problema do ciclo simples mais longo** é o problema de determinar um ciclo simples (sem vértices repetidos) de comprimento máximo em um grafo. Mostre que esse problema é NP-completo.

### 34.5-8

No problema de **satisfabilidade de metade 3-CNF**, temos uma fórmula de 3-CNF  $\phi$  com  $n$  variáveis e  $m$  cláusulas, onde  $m$  é par. Desejamos descobrir se existe uma atribuição verdade para as variáveis de  $\phi$  tais que exatamente metade das cláusulas tem o valor 0 e exatamente metade das cláusulas tem o valor 1. Prove que o problema de satisfabilidade de metade 3-CNF é NP-completo.

## Problemas

### 34-1 Conjunto independente

Um **conjunto independente** de um grafo  $G = (V, E)$  é um subconjunto  $V' \subseteq V$  de vértices, tal que cada aresta em  $E$  é incidente em no máximo um vértice em  $V'$ . O **problema do conjunto independente** é encontrar um conjunto independente de tamanho máximo em  $G$ .

- Formule um problema de decisão relacionado para o problema do conjunto independente e prove que ele é NP-completo. (*Sugestão:* Reduza a partir do problema do grupo exclusivo.)
- Vamos supor que você recebeu uma sub-rotina de “caixa-preta” para resolver o problema de decisão que definiu na parte (a). Forneça um algoritmo para encontrar um conjunto independente de tamanho máximo. O tempo de execução de seu algoritmo deve ser polinomial em  $|V|$  e  $|E|$ , onde consultas à caixa-preta são contadas como um único passo.

Embora o problema de decisão do conjunto independente seja NP-completo, certos casos especiais podem ser resolvidos em tempo polinomial.

- Forneça um algoritmo eficiente para resolver o problema do conjunto independente quando cada vértice em  $G$  tem grau 2. Analise o tempo de execução e prove que seu algoritmo funciona corretamente.
- Forneça um algoritmo eficiente para resolver o problema do conjunto independente quando  $G$  é bipartido. Analise o tempo de execução e prove que seu algoritmo funciona corretamente. (*Sugestão:* Use os resultados da Seção 26.3.)

### 34-2 Bonnie e Clyde

Bonnie e Clyde acabaram de roubar um banco. Eles têm uma bolsa de dinheiro e querem reparti-lo. Para cada dos cenários a seguir, forneça um algoritmo de tempo polinomial ou prove que o problema é NP-completo. A entrada em cada caso é uma lista dos  $n$  itens na bolsa, junto com o valor de cada um.

- Existem  $n$  moedas, mas apenas 2 valores diferentes: algumas moedas valem  $x$  dólares e algumas valem  $y$  dólares. Eles desejam dividir o dinheiro em partes exatamente iguais.
- Existem  $n$  moedas, com um número arbitrário de valores diferentes, mas cada valor é uma potência inteira não negativa de 2, isto é, os valores possíveis são 1 dólar, 2 dólares, 4 dólares etc. Eles desejam dividir o dinheiro em partes exatamente iguais.
- Existem  $n$  cheques, que são, por uma coincidência incrível, nominais a “Bonnie ou Clyde”. Eles desejam dividir os cheques de forma que cada um receba exatamente a mesma quantia.

- d. Existem  $n$  cheques como na parte (c), mas dessa vez eles estão dispostos a aceitar uma divisão em que a diferença não seja maior que 100 dólares.

### 34-3 Coloração de grafos

Uma  **$k$ -coloração** de um grafo não orientado  $G = (V, E)$  é uma função  $c : V \rightarrow \{1, 2, \dots, k\}$  tal que  $c(u) \neq c(v)$  para toda aresta  $(u, v) \in E$ . Em outras palavras, os números  $1, 2, \dots, k$  representam as  $k$  cores, e vértices adjacentes devem ter cores distintas. O **problema de coloração de grafos** é determinar o número mínimo de cores necessárias para colorir um dado grafo.

- Forneça um algoritmo eficiente para determinar 2-colorações de um grafo, se ele existir.
- Formule o problema de coloração de grafos como um problema de decisão. Mostre que seu problema de decisão pode ser resolvido em tempo polinomial se e somente se o problema de coloração de grafos pode ser resolvidos em tempo polinomial.
- Seja a linguagem 3-COR o conjunto de grafos que podem ser 3-coloridos (ou seja, coloridos com 3 cores). Mostre que, se 3-COR é NP-completo, então seu problema de decisão da parte (b) é NP-completo.

Para provar que 3-COR é NP-completo, usamos uma redução de 3-CNF-SAT. Dada uma fórmula  $\phi$ , de  $m$  cláusulas sobre  $n$  variáveis  $x_1, x_2, \dots, x_n$ , construímos um grafo  $G = (V, E)$  da maneira descrita a seguir. O conjunto  $V$  consiste em um vértice para cada variável, um vértice para a negação de cada variável, 5 vértices para cada cláusula e 3 vértices especiais: TRUE, FALSE e RED. As arestas do grafo são de dois tipos: arestas “literais” que são independentes das cláusulas, e arestas de “cláusulas”, que dependem das cláusulas. As arestas literais formam um triângulo nos vértices especiais e também formam um triângulo em  $x_i, \neg x_i$  e RED para  $i = 1, 2, \dots, n$ .

- Demonstre que, em qualquer 3-coloração  $c$  de um grafo contendo as arestas literais, exatamente uma entre uma variável e sua negação é colorida com a cor  $c(\text{TRUE})$  e a outra tem a cor  $c(\text{FALSE})$ . Demonstre que, para qualquer atribuição verdade de  $\phi$ , existe uma 3-coloração do grafo contendo apenas as arestas literais.

O dispositivo mostrado na Figura 34.20 é usado para impor a condição correspondente a uma cláusula  $(x \vee y \vee z)$ . Cada cláusula exige uma cópia única dos 5 vértices que estão fortemente sombreados na figura; eles se conectam da maneira mostrada aos literais da cláusula e ao vértice especial TRUE.

- Demonstre que, se cada  $x, y$  e  $z$  é colorido de  $c(\text{TRUE})$  ou  $c(\text{FALSE})$ , então o dispositivo pode ter 3 cores se e somente se pelo menos um  $x, y$  ou  $z$  é colorido de  $c(\text{TRUE})$ .
- Complete a prova de que 3-COR é NP-completo.

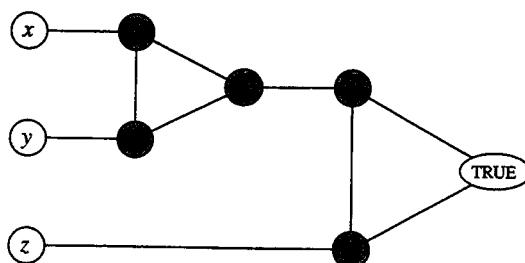


FIGURA 34.20 O dispositivo correspondente a uma cláusula  $(x \vee y \vee z)$ , usado no Problema 34-3

### 34-4 Programação com lucros e prazos finais

Suponha que você tenha uma máquina e um conjunto de  $n$  tarefas  $a_1, a_2, \dots, a_n$ . Cada tarefa  $a_j$  tem um tempo de processamento  $t_j$ , um lucro  $p_j$  e um prazo final  $d_j$ . A máquina só pode processar uma tarefa de cada vez, e a tarefa  $a_j$  deve ser executada ininterruptamente por  $t_j$  unidades de

tempo consecutivas. Se completar a tarefa  $a_j$  em seu prazo final  $d_j$ , você recebe um lucro  $p_j$  mas, se a completar após seu prazo final, você não recebe nenhum lucro. Como um problema de otimização, você tem os tempos de processamento, lucros e prazos finais para um conjunto de  $n$  tarefas, e deseja encontrar uma programação que complete todas as tarefas e retorne o maior lucro possível.

- a.** Enuncie este problema como um problema de decisão.
- b.** Mostre que o problema de decisão é NP-completo.
- c.** Forneça um algoritmo de tempo polinomial para o problema de decisão, supondo que todos os tempos de processamento são inteiros de 1 a  $n$ . (*Sugestão:* Use programação dinâmica.)
- d.** Forneça um algoritmo de tempo polinomial para o problema de otimização, supondo que todos os tempos de processamento são inteiros de 1 a  $n$ .

## Notas do capítulo

O livro de Garey e Johnson [110] é um maravilhoso guia do caráter NP-completo, discutindo a teoria em profundidade e fornecendo um catálogo de muitos problemas que eram reconhecidos como NP-completos em 1979. A prova do Teorema 34.13 foi adaptada de seu livro, e a lista de domínios de problemas NP-completos no início da Seção 34.5 foi extraída de seu sumário. Johnson escreveu uma série de 23 colunas no *Journal of Algorithms* entre 1981 e 1992 relatando novos desenvolvimentos no estudo do caráter NP-completo. Hopcroft, Motwani e Ullman [153], Lewis e Papadimitriou [204], Papadimitriou [236] e Sipser [279] têm bons tratamentos do caráter NP-completo no contexto da teoria da complexidade. Aho, Hopcroft e Ullman [5] também cobrem o caráter NP-completo e fornecem diversas reduções, inclusive uma redução para o problema de cobertura de vértice a partir do problema de ciclo hamiltoniano.

A classe P foi introduzida em 1964 por Cobham [64] e, de forma independente, em 1965 por Edmonds [84], que também apresentou a classe NP e conjecturou que  $P \neq NP$ . A noção do caráter NP-completo foi proposta em 1971 por Cook [67], que forneceu as primeiras provas do caráter NP-completo para satisfabilidade de fórmulas e satisfabilidade 3-CNF. Levin [203] descobriu de forma independente a noção, dando uma prova do caráter NP-completo para um problema de colocação de ladrilhos. Karp [173] introduziu a metodologia de reduções em 1972 e demonstrou a rica variedade de problemas NP-completos. O artigo de Karp incluía as provas originais do caráter NP-completo dos problemas do grupo exclusivo, da cobertura de vértices e do ciclo hamiltoniano. Em uma conversa durante uma reunião para comemorar os sessenta anos de Karp em 1995, Papadimitriou observou que “mais ou menos 6.000 artigos são publicados todo ano com a expressão ‘NP-completo’ em seu título, sumário ou lista de palavras-chave. Isso é mais que cada um dos termos ‘compilador’, ‘banco de dados’, ‘especialista’, ‘rede neural’ ou ‘sistema operacional’”.

O recente trabalho sobre teoria da complexidade esclarece a complexidade de soluções aproximadas de computação. Esse trabalho fornece uma nova definição de NP, usando “provas verificáveis de forma probabilística”. Essa nova definição implica que, para problemas como o do grupo exclusivo, de cobertura de vértices, do caixeiro viajante com a desigualdade de triângulos e muitos outros, o cálculo de boas soluções aproximadas é NP-difícil e, consequentemente, não é mais fácil que o cálculo de soluções ótimas. Uma introdução a esse assunto pode ser encontrada na tese de Arora [19], em um capítulo de Arora e Lund em [149], um artigo de pesquisa de Arora [20], em um livro editado por Mayr, Promel e Steger [214] e em um artigo de pesquisa de Johnson [167].

# Algoritmos de aproximação

Muitos problemas de significado prático são NP-completos, mas são importantes demais para serem abandonados apenas porque é impraticável obter uma solução ótima. Se um problema é NP-completo, temos pouca probabilidade de encontrar um algoritmo de tempo polinomial para resolvê-lo exatamente mas, mesmo assim, pode haver esperança. Existem pelo menos três abordagens para contornar o NP-completo. Primeiro, se as entradas reais são pequenas, um algoritmo com tempo de execução exponencial pode ser perfeitamente satisfatório. Em segundo lugar, podemos ser capazes de isolar casos especiais importantes que possam ser resolvidos em tempo polinomial. Em terceiro lugar, ainda é possível encontrar soluções *aproximadas* em tempo polinomial (seja no pior caso ou no caso médio). Na prática, uma aproximação com frequência é suficientemente boa. Um algoritmo que retorna soluções aproximadas é chamado *algoritmo de aproximação*. Este capítulo apresenta algoritmos de aproximação de tempo polinomial para vários problemas NP-completos.

### Razões de desempenho para algoritmos de aproximação

Suponha que estamos trabalhando em um problema de otimização em que cada solução potencial tem um custo positivo, e que desejamos encontrar uma solução aproximada. Dependendo do problema, uma solução ótima pode ser definida como uma solução com máximo custo possível ou com custo mínimo possível; isto é, o problema pode ser um problema de maximização ou um problema de minimização.

Dizemos que um algoritmo de aproximação para o problema tem uma *relação de aproximação*  $\rho(n)$  se, para qualquer entrada de tamanho  $n$ , o custo  $C$  da solução produzida pelo algoritmo de aproximação está dentro de um fator  $\rho(n)$  do custo  $C^*$  de uma solução ótima:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n) . \quad (35.1)$$

Também chamamos um algoritmo que alcança uma relação de aproximação  $\rho(n)$  um algoritmo de aproximação  $\rho(n)$ . As definições de relação de aproximação e algoritmo de aproximação  $\rho(n)$  se aplicam tanto a problemas de minimização quanto de maximização. Para um problema de maximização,  $0 < C \leq C^*$ , e a relação  $C^*/C$  fornece o fator pelo qual o custo de uma solução ótima é maior que o custo da solução aproximada. De forma semelhante, para um problema de mi-

nimização,  $0 < C^* \leq C$ , e a relação  $C/C^*$  fornece o fator pelo qual o custo da solução aproximada é maior que o custo de uma solução ótima. Como todas as soluções são consideradas de custo positivo, essas relações são sempre bem definidas. A relação de aproximação de um algoritmo de aproximação nunca é menor que 1, pois  $C/C^* < 1$  implica  $C^*/C > 1$ . Então, um algoritmo de <sup>1</sup> produz uma relação ótima, e um algoritmo de aproximação com uma relação de aproximação grande pode retornar uma solução muito pior que a solução ótima.

Para muitos problemas, foram desenvolvidos algoritmos de aproximação de tempo polinomial com pequenas relações de aproximação constantes enquanto, para outros problemas, os melhores algoritmos de aproximação de tempo polinomial conhecidos têm relações de aproximação que crescem como funções do tamanho da entrada  $n$ . Um exemplo de tal problema é o problema de cobertura de conjuntos apresentado na Seção 35.3.

Alguns problemas NP-completos permitem algoritmos de aproximação de tempo polinomial que podem alcançar relações de aproximação cada vez menores, usando cada vez mais tempo de computação. Isto é, existe um compromisso entre tempo de computação e qualidade da aproximação. Um exemplo é o problema da soma de subconjuntos estudado na Seção 35.5. Essa situação é bastante importante para merecer um nome próprio.

Um *esquema de aproximação* para um problema de otimização é um algoritmo de aproximação que toma como entrada não apenas uma instância do problema, mas também um valor  $\varepsilon$  tal que, para qualquer  $\varepsilon$  fixo, o esquema é um algoritmo de aproximação  $(1 + \varepsilon)$ . Dizemos que um esquema de aproximação é um *esquema de aproximação de tempo polinomial* se, para qualquer  $\varepsilon > 0$  fixo, o esquema é executado em tempo polinomial no tamanho  $n$  de sua instância de entrada.

O tempo de execução de um esquema de aproximação em tempo polinomial não deve aumentar muito rapidamente à medida que  $\varepsilon$  diminui. Por exemplo, o tempo de execução de um esquema de aproximação de tempo polinomial poderia ser  $O(n^{2/\varepsilon})$ . No caso ideal, se  $\varepsilon$  diminui por um fator constante, o tempo de execução para se alcançar a aproximação desejada não deve aumentar por mais que um fator constante. Em outras palavras, gostaríamos que o tempo de execução fosse polinomial em  $1/\varepsilon$ , bem como em  $n$ .

Dizemos que um esquema de aproximação é um *esquema de aproximação de tempo completamente polinomial* se ele é um esquema de aproximação e seu tempo de execução é polinomial tanto em  $1/\varepsilon$  quanto no tamanho  $n$  da instância de entrada. Por exemplo, o esquema poderia ter um tempo de execução  $O((1/\varepsilon)^2 n^3)$ . Com tal esquema, qualquer redução de fator constante em  $\varepsilon$  pode ser obtida com um aumento correspondente de fator constante no tempo de execução.

## Esboço do capítulo

As quatro primeiras seções deste capítulo apresentam alguns exemplos de algoritmos de aproximação de tempo polinomial para problemas NP-completos, e a quinta seção apresenta um esquema de aproximação de tempo completamente polinomial. A Seção 35.1 começa com um estudo do problema de cobertura de vértices, um problema de minimização NP-completo que tem um algoritmo de aproximação com uma relação de aproximação igual a 2. A Seção 35.2 apresenta um algoritmo de aproximação com relação de aproximação 2 para o caso do problema do caixeiro-viajante, no qual a função de custo satisfaz à desigualdade de triângulos. Ela também mostra que, sem a desigualdade de triângulos, para qualquer constante  $\rho \geq 1$ , um algoritmo de aproximação  $\rho$  não pode existir, a menos que  $P = NP$ . Na Seção 35.3, mostramos como um método guloso pode ser usado como um algoritmo de aproximação efetivo para o problema de cobertura de conjuntos, obtendo uma coberta cujo custo é na pior das hipóteses um fator logarítmico

---

<sup>1</sup> Quando a relação de aproximação for independente de  $n$ , usaremos os termos relação de aproximação de  $\rho$  e algoritmo de aproximação  $\rho$ , sem indicar nenhuma dependência de  $n$ .

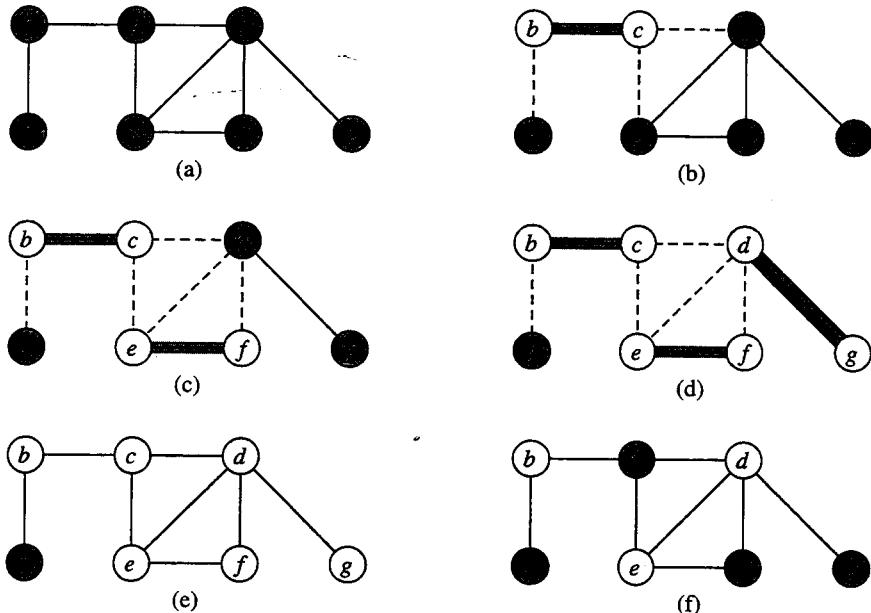
maior que o custo ótimo. A Seção 35.4 apresenta mais dois algoritmos de aproximação. Primeiro, estudaremos a versão de otimização da satisfatibilidade 3-CNF e daremos um algoritmo aleatório simples que produz uma solução com uma relação de aproximação esperada de 8/7. Em seguida, examinaremos uma variante ponderada do problema de cobertura de vértices e mostraremos como usar a programação linear para desenvolver um algoritmo de aproximação 2. Finalmente, a Seção 35.5 apresenta um esquema de aproximação de tempo completamente polinomial para o problema da soma de subconjuntos.

### 35.1 O problema de cobertura de vértices

O problema de cobertura de vértices foi definido e provado como NP-completo na Seção 34.5.2. Uma **cobertura de vértices** de um grafo não orientado  $G = (V, E)$  é um subconjunto  $V' \subseteq V$  tal que, se  $(u, v)$  é uma aresta de  $G$ , então  $u \in V'$  ou  $v \in V'$  (ou ambos). O tamanho de uma cobertura de vértices é o número de vértices que ela contém.

O **problema de cobertura de vértices** é encontrar uma cobertura de vértices de tamanho mínimo em um dado grafo não orientado. Chamamos tal cobertura de vértices uma **cobertura de vértices ótima**. Esse problema é a versão de otimização de um problema de decisão NP-completo.

Embora possa ser difícil encontrar uma cobertura de vértices ótima em um grafo  $G$ , não é muito difícil encontrar uma cobertura de vértices que seja aproximada. O algoritmo de aproximação a seguir toma como entrada um grafo não orientado  $G$  e retorna uma cobertura de vértices cujo tamanho tem a garantia de não ser maior que duas vezes o tamanho de uma cobertura de vértices ótima.



**FIGURA 35.1** A operação de APPROX-VERTEX-COVER. (a) O grafo de entrada  $G$ , que tem 7 vértices e 8 arestas. (b) A aresta  $(b, c)$ , mostrada sombreada, é a primeira aresta escolhida por APPROX-VERTEX-COVER. Os vértices  $b$  e  $c$ , mostrados levemente sombreados, são adicionados ao conjunto  $C$  que contém a cobertura de vértices que está sendo criada. As arestas  $(a, b)$ ,  $(c, e)$  e  $(c, d)$ , mostradas tracejadas, são removidas porque agora estão cobertas por algum vértice em  $C$ . (c) A aresta  $(e, f)$  é escolhida; os vértices  $e$  e  $f$  são adicionados a  $C$ . (d) A aresta  $(d, g)$  é escolhida; os vértices  $d$  e  $g$  são adicionados a  $C$ . (e) O conjunto  $C$ , que é a cobertura de vértices produzida por APPROX-VERTEX-COVER, contém os seis vértices  $b, c, d, e, f, g$ . (f) A cobertura de vértices ótima para esse problema contém apenas três vértices:  $b, d$  e  $e$ .

### APPROX-VERTEX-COVER( $G$ )

```

1  $C \leftarrow \emptyset$ 
2  $E' \leftarrow E[G]$ 
3 while  $E' \neq \emptyset$ 
4   do seja  $(u, v)$  uma aresta arbitrária de  $E'$ 
5      $C \leftarrow C \cup \{u, v\}$ 
6     remover de  $E'$  toda aresta incidente sobre  $u$  ou  $v$ 
7 return  $C$ 

```

A Figura 35.1 ilustra a operação de APPROX-VERTEX-COVER. A variável  $C$  contém a cobertura de vértices que está sendo construída. A linha 1 inicializa  $C$  como o conjunto vazio. A linha 2 define  $E'$  como uma cópia do conjunto de arestas  $E[G]$  do grafo. O loop nas linhas 3 a 6 escolhe repetidamente uma aresta  $(u, v)$  de  $E'$ , adiciona suas extremidades  $u$  e  $v$  a  $C$  e elimina todas as arestas em  $E'$  que são cobertas por  $u$  ou  $v$ . O tempo de execução desse algoritmo é  $O(V + E)$ , usando listas de adjacências para representar  $E'$ .

#### **Teorema 35.1**

APPROX-VERTEX-COVER é um algoritmo de aproximação 2 de tempo polinomial.

**Prova** Já mostramos que APPROX-VERTEX-COVER é executado em tempo polinomial.

O conjunto  $C$  de vértices que é retornado por APPROX-VERTEX-COVER é uma cobertura de vértices, pois o algoritmo entra em loop até toda aresta em  $E[G]$  ter sido coberta por algum vértice em  $C$ .

Para ver que APPROX-VERTEX-COVER retorna uma cobertura de vértices que é no máximo duas vezes o tamanho de uma cobertura ótima, seja  $A$  o conjunto de arestas que foram escolhidas na linha 4 de APPROX-VERTEX-COVER. Para cobrir as arestas em  $A$  qualquer cobertura de vértices – em particular uma cobertura ótima  $C^*$  – deve incluir pelo menos uma extremidade de cada aresta em  $A$ . Duas arestas em  $A$  não compartilham uma extremidade, pois uma vez que uma aresta é escolhida na linha 4, todas as outras arestas que são incidentes em suas extremidades são eliminadas de  $E'$  na linha 6. Desse modo, não há duas arestas em  $A$  cobertas pelo mesmo vértice de  $C^*$ , e temos o limite inferior

$$|C^*| \geq |A| \quad (35.2)$$

sobre o tamanho de uma cobertura de vértices ótima. Cada execução da linha 4 escolhe uma aresta para a qual nenhuma de suas extremidades já está em  $C$ , produzindo um limite superior (de fato, um limite superior exato) sobre o tamanho da cobertura de vértices retornada:

$$|C| = 2 |A| . \quad (35.3)$$

Combinando as equações (35.2) e (35.3), obtemos

$$|C| = 2 |A|$$

$$\leq 2 |C^*| ,$$

provando assim o teorema. ■

Vamos refletir sobre essa prova. A princípio, poderíamos imaginar como é possível provar que o tamanho da cobertura de vértices retornada por APPROX-VERTEX-COVER é no máximo duas vezes o tamanho de uma cobertura de vértices ótima, pois não sabemos qual é o tamanho da cobertura de vértices ótima. A resposta é que utilizamos um limite inferior sobre o tamanho da cobertura de vértices ótima. Como o Exercício 35.1-2 lhe pede para mostrar, o conjunto  $A$  de arestas que eram escolhidas na linha 4 de APPROX-VERTEX-COVER é realmente um emparelha-

mento máximo no grafo  $G$ . (Um **emparelhamento máximo** é um emparelhamento que não é um subconjunto próprio de qualquer outro emparelhamento.) O tamanho de um emparelhamento máximo é, como demonstramos na prova do Teorema 35.1, um limite inferior sobre o tamanho de uma cobertura de vértices ótima. O algoritmo retorna uma cobertura de vértices cujo tamanho é no máximo duas vezes o tamanho do emparelhamento máximo  $A$ . Relacionando o tamanho da solução retornada ao limite inferior, obtemos nossa relação de aproximação. Também utilizaremos essa metodologia em seções posteriores.

## Exercícios

### 35.1-1

Forneça um exemplo de um grafo para o qual APPROX-VERTEX-COVER sempre produz um solução menos que ótima.

### 35.1-2

Seja  $A$  o conjunto de arestas escolhidas na linha 4 de APPROX-VERTEX-COVER. Prove que o conjunto  $A$  é um emparelhamento máximo no grafo  $G$ .

### 35.1-3 \*

O professor Nilton propõe a seguinte heurística para resolver o problema de cobertura de vértices. Selecione repetidamente um vértice de grau mais alto, e remova todas as suas arestas incidentes. Forneça um exemplo para mostrar que a heurística do professor não tem uma relação de aproximação igual a 2. (*Sugestão:* Experimente um grafo bipartido com vértices de grau uniforme à esquerda e vértices de grau variado à direita.)

### 35.1-4

Forneça um algoritmo guloso eficiente que encontre uma cobertura de vértices ótima para uma árvore em tempo linear.

### 35.1-5

Da prova do Teorema 34.12, sabemos que o problema de cobertura de vértices e o problema do grupo exclusivo NP-completo são complementares, no sentido de que uma cobertura de vértices ótima é o complemento de um grupo exclusivo de tamanho máximo no grafo complemento. Esse relacionamento implica que existe um algoritmo de aproximação com relação de aproximação constante para o problema do grupo exclusivo? Justifique sua resposta.

## 35.2 O problema do caixeiro-viajante

No problema do caixeiro-viajante introduzido na Seção 34.5.4, temos um grafo não orientado completo  $G = (V, E)$  que tem um custo inteiro não negativo  $c(u, v)$  associado com cada aresta  $(u, v) \in E$ , e devemos encontrar um ciclo hamiltoniano (uma viagem) de  $G$  com custo mínimo. Como uma extensão de nossa notação, seja  $c(A)$  o custo total das arestas no subconjunto  $A \subseteq E$ :

$$c(A) = \sum_{(u,v) \in A} c(u, v).$$

Em muitas situações práticas, é sempre mais econômico ir diretamente de um lugar  $u$  para um lugar  $w$ ; ir por meio de qualquer parada intermediária  $v$  não pode ser menos dispendioso. Em outros termos, cortar uma parada intermediária nunca aumenta o custo. Formalizamos essa noção dizendo que a função de custo  $c$  satisfaz à **desigualdade de triângulos** se, para todos os vértices  $u, v, w \in V$ ,

A desigualdade de triângulos é uma desigualdade natural e, em muitas aplicações é satisfeita de forma automática. Por exemplo, se os vértices do grafo são pontos no plano e o custo de viajar entre dois vértices é a distância euclidiana comum entre eles, então a desigualdade de triângulos é satisfeita. (Existem muitas funções de custo além da distância euclidiana que satisfazem à desigualdade de triângulos.)

Como mostra o Exercício 35.2-2, o problema do caixeiro-viajante é NP-completo, mesmo se exigirmos que a função de custo satisfaça à desigualdade de triângulos. Desse modo, é improvável que possamos encontrar um algoritmo de tempo polinomial para resolver esse problema com exatidão. Assim, procuramos em vez disso bons algoritmos de aproximação.

Na Seção 35.2.1, examinaremos um algoritmo de aproximação 2 para o problema do caixeiro-viajante com a desigualdade de triângulos. Na Seção 35.2.2, mostraremos que, sem a desigualdade de triângulos, um algoritmo de aproximação de tempo polinomial com uma relação de aproximação constante não existe, a menos que  $P = NP$ .

### 35.2.1 O problema do caixeiro-viajante com desigualdade de triângulos

Aplicando a metodologia da seção anterior, calcularemos primeiro uma estrutura – uma árvore espalhada mínima – cujo peso é um limite inferior sobre o comprimento de uma viagem ótima do caixeiro-viajante. Em seguida, usaremos a árvore espalhada mínima para criar uma viagem cujo custo não é maior que duas vezes o custo do peso da árvore de amplitude mínima, desde que a função de custo satisfaça à desigualdade de triângulos. O algoritmo a seguir implementa essa abordagem, chamando o algoritmo de árvore de amplitude mínima MST-PRIM da Seção 23.2 como uma sub-rotina.

**APPROX-TSP-TOUR( $G, c$ )**

- 1 selecione um vértice  $r \in V[G]$  para ser um vértice “raiz”
- 2 calcule uma árvore de amplitude mínima  $T$  para  $G$  a partir da raiz  $r$   
usando  $MST-PRIM(G, c, r)$
- 3 seja  $L$  a lista de vértices visitados em um percurso de árvore de pré-ordem de  $T$
- 4 **return** o ciclo hamiltoniano  $H$  que visita os vértices na ordem  $L$

Lembramos da Seção 12.1 que um percurso de árvore de pré-ordem visita recursivamente todo vértice na árvore, listando um vértice quando ele é encontrado pela primeira vez, antes de qualquer de seus filhos ser visitado.

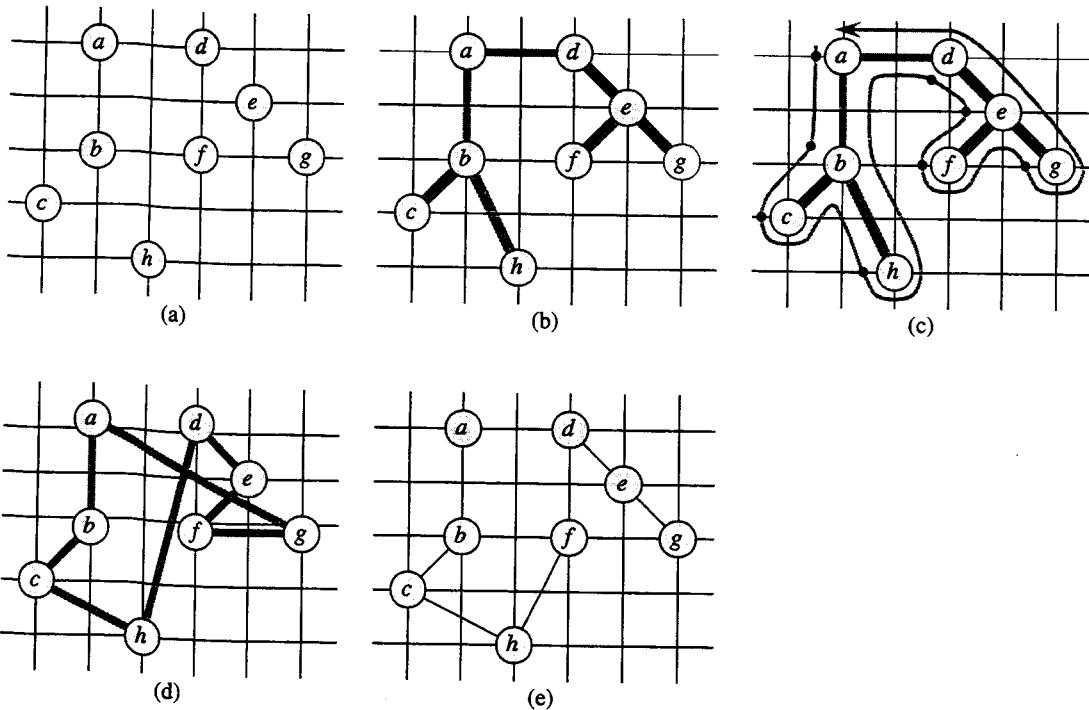
A Figura 35.2 ilustra a operação de APPROX-TSP-TOUR. A parte (a) da figura mostra o conjunto de vértices dado, e a parte (b) mostra a árvore espalhada mínima  $T$  que cresceu a partir do vértice de raiz  $a$  por MST-PRIM. A parte (c) mostra como os vértices são visitados por um percurso de pré-ordem de  $T$ , e a parte (d) exibe a viagem correspondente, que é a viagem retornada por APPROX-TSP-TOUR. A parte (e) exibe uma viagem ótima, o qual é cerca de 23% mais curto.

Pelo Exercício 23.2-2, até mesmo com uma implementação simples de MST-PRIM, o tempo de execução de APPROX-TSP-TOUR é  $\Theta(V^2)$ . Agora, mostramos que, se a função de custo para uma instância do problema do caixeiro-viajante satisfaz à desigualdade de triângulos, então APPROX-TSP-TOUR retorna uma viagem cujo custo não é maior que duas vezes o custo de uma viagem ótima.

#### **Teorema 35.2**

APPROX-TSP-TOUR é um algoritmo de aproximação 2 de tempo polinomial para o problema do caixeiro-viajante com a desigualdade de triângulos.

**Prova** Já mostramos que APPROX-TSP-TOUR é executado em tempo polinomial.



**FIGURA 35.2** A operação de APPROX-TSP-TOUR. (a) O conjunto de pontos dado, que residem nos vértices de uma grade inteira. Por exemplo,  $f$  está uma unidade à direita e duas unidades acima de  $b$ . A distância euclidiana comum é usada como a função de custo entre dois pontos. (b) Uma árvore de amplitude mínima  $T$  desses pontos, conforme é calculada por MST-PRIM. O vértice  $a$  é o vértice raiz. Os vértices são identificados de tal modo que eles são acrescentados à árvore principal por MST-PRIM em ordem alfabética. (c) Um percurso de  $T$ , começando em  $a$ . Um percurso completo da árvore visita os vértices na ordem  $a, b, c, b, b, a, d, e, f, e, g, e, d, a$ . Um percurso de pré-ordem de  $T$  lista um vértice apenas quando ele é encontrado pela primeira vez, conforme indica o ponto ao lado de cada vértice, produzindo a ordenação  $a, b, c, b, d, e, f, g$ . (d) Uma viagem dos vértices obtidos pela visita aos vértices na ordem dada pelo percurso de pré-ordem. Essa é a viagem  $H$  retornada por APPROX-TSP-TOUR. Seu custo total é aproximadamente 19,074. (e) Uma viagem ótima  $H^*$  para o conjunto de vértices dado. Seu custo total é aproximadamente 14,715

Seja  $H^*$  uma viagem ótima para o conjunto de vértices dado. Tendo em vista que obtemos uma árvore de amplitude eliminando qualquer aresta de uma viagem, o peso da árvore de amplitude mínima  $T$  é um limite inferior sobre o custo de uma viagem ótima, ou seja,

$$c(T) \leq c(H^*) . \quad (35.4)$$

Um **percurso completo** de  $T$  lista os vértices quando eles são visitados pela primeira vez, e também sempre que se retorna a eles depois de uma visita a uma subárvore. Vamos chamar esse percurso  $W$ . O percurso completo de nosso exemplo fornece a ordem

$$a, b, c, b, b, a, d, e, f, e, g, e, d, a .$$

Como o percurso completo passa por toda aresta de  $T$  exatamente duas vezes, temos (estendendo nossa definição do custo  $c$  da maneira natural para tratar conjuntos múltiplos de arestas)

$$c(W) = 2c(T) . \quad (35.5)$$

As equações (35.4) e (35.5) implicam que

$$c(W) = 2c(H^*) , \quad (35.6)$$

812 | e assim o custo de  $W$  está dentro de um fator 2 do custo de uma viagem ótima.

Infelizmente,  $W$  em geral não é uma viagem, pois visita alguns vértices mais de uma vez. Porem, pela desigualdade de triângulos, podemos eliminar uma visita a qualquer vértice a partir de  $W$ , e o custo não aumenta. (Se um vértice  $v$  é eliminado de  $W$  entre visitas a  $u$  e  $w$ , a ordenação resultante especifica a ida diretamente de  $u$  para  $w$ .) Pela aplicação repetida dessa operação, podemos remover de  $W$  tudo, menos a primeira visita a cada vértice. Em nosso exemplo, isso deixa a ordenação

$a, b, c, b, d, e, f, g$ .

Essa ordenação é a mesma que foi obtida por um percurso de pré-ordem da árvore  $T$ . Seja  $H$  o ciclo correspondente a esse percurso de pré-ordem. Ele é um ciclo hamiltoniano, pois todo vértice é visitado exatamente uma vez, e de fato é o ciclo calculado por APPROX-TSP-TOUR. Tendo em vista que  $H$  é obtido pela eliminação de vértices do percurso completo  $W$ , temos

$$c(H) \leq c(W). \quad (35.7)$$

A combinação das desigualdades (35.6) e (35.7) mostra que  $c(H) \leq 2c(H^*)$ , o que completa a prova. ■

Apesar da bela relação de aproximação fornecida pelo Teorema 35.2, APPROX-TSP-TOUR normalmente é não a melhor escolha prática para esse problema. Existem outros algoritmos de aproximação que em geral funcionam muito melhor na prática (consulte as referências no final deste capítulo).

### 35.2.2 O problema geral do caixeiro-viajante

Se eliminarmos a hipótese de que a função de custo  $c$  satisfaz à desigualdade de triângulos, boas viagens aproximadas não poderão ser encontradas em tempo polinomial, a menos que  $P = NP$ .

#### Teorema 35.3

Se  $P \neq NP$ , então para qualquer constante  $\rho \geq 1$ , não existe nenhum algoritmo de aproximação de tempo polinomial com relação de aproximação  $\rho$  para o problema geral do caixeiro-viajante.

**Prova** A prova é por contradição. Suponha que, ao contrário, para algum número  $\rho \geq 1$  existe um algoritmo de aproximação de tempo polinomial  $A$  com relação de aproximação  $\rho$ . Sem perda de generalidade, supomos que  $\rho$  é um inteiro, arredondando-o se necessário. Então, mostraremos como usar  $A$  para resolver instâncias do problema do ciclo hamiltoniano (definido na Seção 34.2) em tempo polinomial. Como o problema do ciclo hamiltoniano é NP-completo, pelo Teorema 34.13, resolvê-lo em tempo polinomial implica que  $P = NP$ , pelo Teorema 34.4.

Seja  $G = (V, E)$  uma instância do problema do ciclo hamiltoniano. Desejamos determinar de modo eficiente se  $G$  contém um ciclo hamiltoniano fazendo uso do algoritmo de aproximação hipotético  $A$ . Transformamos  $G$  em uma instância do problema do caixeiro-viajante da maneira ilustrada a seguir. Seja  $G' = (V, E')$  o grafo completo em  $V$ ; isto é,

$$E' = \{(u, v) : u, v \in V \text{ e } u \neq v\}.$$

Atribuímos um custo inteiro a cada aresta em  $E'$  como a seguir:

$$c(u, v) = \begin{cases} 1 & \text{se } (u, v) \in E, \\ \rho|V|+1 & \text{em caso contrário.} \end{cases}$$

As representações de  $G'$  e  $c$  podem ser criadas a partir de uma representação de  $G$  em tempo polinomial em  $|V|$  e  $|E|$ .

Agora, considere o problema do caixeiro-viajante  $(G', c)$ . Se o grafo original  $G$  tem um ciclo hamiltoniano  $H$ , então a função de custo  $c$  atribui a cada aresta de  $H$  um custo 1, e assim  $(G', c)$  contém uma viagem de custo  $|V|$ . Por outro lado, se  $G$  não contém um ciclo hamiltoniano, então qualquer viagem de  $G'$  deve usar alguma aresta que não está em  $E$ . Contudo, qualquer viagem que utilize uma aresta não contida em  $E$  tem um custo de pelo menos

$$\begin{aligned} (\rho|V|+1) + (|V|-1) &= \rho|V|+|V| \\ &> \rho|V|. \end{aligned}$$

Tendo em vista que as arestas que não estão em  $G$  são tão dispendiosas, existe uma lacuna de pelo menos  $|V|$  entre o custo de uma viagem que é um ciclo hamiltoniano em  $G$  (custo  $|V|$ ) e o custo de qualquer outra viagem (custo no mínimo igual a  $\rho|V| + |V|$ ).

O que acontecerá se aplicarmos o algoritmo de aproximação  $A$  ao problema do caixeiro-viajante  $(G', c)$ ? Como  $A$  tem a garantia de retornar uma viagem de custo não maior que  $\rho$  vezes o custo de uma viagem ótima, se  $G$  contém um ciclo hamiltoniano, então  $A$  deve retorná-lo. Se  $G$  não tem nenhum ciclo hamiltoniano, então  $A$  retorna uma viagem de custo maior que  $\rho|V|$ . Assim, podemos usar  $A$  para resolver o problema do ciclo hamiltoniano em tempo polinomial.

A prova do Teorema 35.3 é um exemplo de uma técnica geral para provar que um problema não pode ser bem aproximado. Suponha que, dado um problema NP-difícil  $X$ , podemos produzir um problema de minimização  $Y$  tal que instâncias “sim” de  $X$  correspondam a instâncias de  $Y$  com valor no máximo  $k$  (para algum  $k$ ), mas que instâncias “não” de  $X$  correspondam a instâncias de  $Y$  com valor maior que  $\rho k$ . Então mostramos que, a menos que  $P = NP$ , não existe nenhum algoritmo de aproximação  $\rho$  para problema  $Y$ .

## Exercícios

### 35.2-1

Suponha que um grafo não orientado completo  $G = (V, E)$  com pelo menos 3 vértices tenha uma função de custo  $c$  que satisfaça à desigualdade de triângulos. Prove que  $c(u, v) \geq 0$  para todo  $u, v \in V$ .

### 35.2-2

Mostre como podemos transformar em tempo polinomial uma instância do problema do caixeiro-viajante em outra instância cuja função de custo satisfaça à desigualdade de triângulos. As duas instâncias devem ter o mesmo conjunto de viagens ótimas. Explique por que tal transformação de tempo polinomial não contradiz o Teorema 35.3, supondo que  $P \neq NP$ .

### 35.2-3

Considere a seguinte **heurística de pontos mais próximos** para construir uma viagem aproximada do caixeiro-viajante. Comece com um ciclo trivial consistindo em um único vértice escolhido arbitrariamente. Em cada passo, identifique o vértice  $u$  que não está no ciclo, mas cuja distância até qualquer vértice no ciclo é mínima. Suponha que o vértice no ciclo mais próximo de  $u$  seja o vértice  $v$ . Estenda o ciclo para incluir  $u$ , inserindo  $u$  logo depois de  $v$ . Repita até todos os vértices estarem no ciclo. Prove que essa heurística retorna uma viagem cujo custo total não é maior que duas vezes o custo de uma viagem ótima.

### 35.2-4

O **problema do caixeiro-viajante com gargalo** é o problema de encontrar o ciclo hamiltoniano tal que o custo da aresta de maior custo no ciclo seja minimizado. Supondo que a função de custo satisfaça à desigualdade de triângulos, mostre que existe um algoritmo de aproximação de tempo polinomial com relação de aproximação igual a 3 para esse problema. (Sugestão. Mostre recursivamente que podemos visitar todos os nós em uma árvore de amplitude de gargalo,

como vimos no Problema 23-3, exatamente uma vez, tomando um percurso completo da árvore e ignorando os nós, mas sem ignorar mais de 2 nós intermediários consecutivos. Mostre que a aresta de maior custo em uma árvore de amplitude de gargalo tem um custo que é no máximo o custo da aresta de maior custo em um ciclo hamiltoniano de gargalo.)

### 35.2-5

Suponha que os vértices para uma instância do problema do caixeiro-viajante sejam pontos no plano, e que o custo  $c(u, v)$  seja a distância euclidiana entre os pontos  $u$  e  $v$ . Mostre que uma viagem ótima nunca cruza a si própria.

## 35.3 O problema de cobertura de conjuntos

O problema de cobertura de conjuntos é um problema de otimização que modela muitos problemas de seleção de recursos. Seu problema de decisão correspondente generaliza o problema de cobertura de vértices NP-completo e, portanto, também é NP-difícil. Porém, o algoritmo de aproximação desenvolvido para tratar o problema de cobertura de vértices não se aplica aqui, e assim precisamos tentar outras abordagens. Examinaremos uma heurística gulosa simples com uma relação de aproximação logarítmica. Ou seja, à medida que o tamanho da instância se torna maior, o tamanho da solução aproximada pode crescer, em relação ao tamanho de uma solução ótima. Contudo, como a função logaritmo cresce de forma bastante lenta, esse algoritmo de aproximação pode mesmo assim fornecer resultados úteis.

Uma instância  $(X, \mathcal{F})$  do **problema de coberta de conjuntos** consiste em um conjunto finito  $X$  e uma família  $\mathcal{F}$  de subconjuntos de  $X$ , tal que todo elemento de  $X$  pertence a pelo menos um subconjunto em  $\mathcal{F}$ :

$$X = \bigcup_{S \in \mathcal{F}} S$$

Dizemos que um subconjunto  $S \in \mathcal{F}$  **cobre** seus elementos. O problema é encontrar um subconjunto de tamanho mínimo  $\mathcal{C} \subseteq \mathcal{F}$  cujos membros cubram todo o  $X$ :

$$X = \bigcup_{S \in \mathcal{C}} S. \quad (35.8)$$

Dizemos que qualquer  $\mathcal{C}$  que satisfaz à equação (35.8) **cobre**  $X$ . A Figura 35.3 ilustra o problema de cobertura de conjuntos. O tamanho de  $\mathcal{C}$  é definido como o número de conjuntos que ele contém, em vez do número de elementos individuais nesses conjuntos. Na Figura 35.3, a cobertura do conjunto mínimo tem tamanho 3.

O problema de cobertura de conjuntos é uma abstração de muitos problemas combinatórios que surgem comumente. Como um exemplo simples, suponha que  $X$  represente um conjunto de habilidades necessárias para resolver um problema e que temos um dado conjunto de pessoas disponíveis para trabalhar no problema. Desejamos formar uma comissão, contendo o menor número de pessoas possível tal que, para toda habilidade requerida em  $X$ , exista um membro da comissão que tenha essa habilidade. Na versão de decisão do problema de cobertura de conjuntos, perguntamos se existe ou não uma coberta com tamanho máximo  $k$ , onde  $k$  é um parâmetro adicional especificado na instância do problema. A versão de decisão do problema é NP-completa, como o Exercício 35.3-2 lhe pede para mostrar.

### Um algoritmo guloso de aproximação

O método guloso funciona escolhendo, em cada fase, o conjunto  $S$  que cobre o maior número de elementos restantes que estão descobertos.

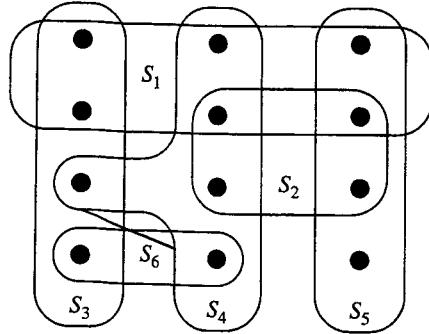


FIGURA 35.3 Uma instância  $(X, \mathcal{F})$  do problema de cobertura de conjuntos, onde  $X$  consiste nos 12 pontos pretos e  $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$ . Uma cobertura de conjuntos de tamanho mínimo é  $\mathcal{C} = \{S_3, S_4, S_5\}$ . O algoritmo guloso produz uma cobertura de tamanho 4, selecionando os conjuntos  $S_1, S_4, S_5$  e  $S_3$  nessa ordem.

#### GREEDY-SET-COVER( $X, \mathcal{F}$ )

```

1  $U \leftarrow X$ 
2  $\mathcal{C} \leftarrow \emptyset$ 
3 while  $U \neq \emptyset$ 
4   do selecione um  $S \in \mathcal{F}$  que maximize  $|S \cap U|$ 
5    $U \leftarrow U - S$ 
6    $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$ 
7 return  $\mathcal{C}$ 
```

No exemplo da Figura 35.3, GREEDY-SET-COVER adiciona a  $\mathcal{C}$  os conjuntos  $S_1, S_4, S_5$  e  $S_3$ , nessa ordem.

O algoritmo funciona da maneira descrita a seguir. O conjunto  $U$  contém, em cada fase, o conjunto de elementos descobertos restantes. O conjunto  $\mathcal{C}$  contém a cobertura que está sendo construída. A linha 4 é o passo de tomada de decisão gulosa. Um subconjunto  $S$  é escolhido para cobrir tantos elementos descobertos quanto possível (com as ligações rompidas arbitrariamente). Depois que  $S$  é selecionado, seus elementos são removidos de  $U$ , e  $S$  é colocado em  $\mathcal{C}$ . Quando o algoritmo termina, o conjunto  $\mathcal{C}$  contém uma subfamília de  $\mathcal{F}$  que cobre  $X$ .

O algoritmo GREEDY-SET-COVER pode ser implementado com facilidade para execução em tempo polinomial em  $|X|$  e  $|\mathcal{F}|$ . Como o número de iterações do loop nas linhas 3 a 6 é limitado acima por  $(|X|, |\mathcal{F}|)$ , e o corpo do loop pode ser implementado para execução no tempo  $O(|S| |\mathcal{F}|)$ , existe uma implementação que é executada no tempo  $O(|S| |\mathcal{F}|) \min(|X|, |\mathcal{F}|)$ . O Exercício 35.3-3 lhe pede um algoritmo de tempo linear.

#### Análise

Agora, mostraremos que o algoritmo guloso retorna uma cobertura de conjuntos que não é muito maior que uma cobertura de conjuntos ótima. Por conveniência, neste capítulo denotaremos o  $d$ -ésimo número harmônico  $H_d = \sum_{i=1}^d 1/i$  (consulte a Seção A.1) por  $H(d)$ . Como uma condição limite, definimos  $H(0) = 0$ .

#### *Teorema 35.4*

GREEDY-SET-COVER é um algoritmo de aproximação  $\rho(n)$  de tempo polinomial, onde

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}).$$

**Prova** Já mostramos que GREEDY-SET-COVER é executado em tempo polinomial.

Para mostrar que GREEDY-SET-COVER é um algoritmo de aproximação  $\rho(n)$  atribuímos um custo 1 a cada conjunto selecionado pelo algoritmo, distribuímos esse custo sobre os elementos

cobertos pela primeira vez e depois usamos esses custos para derivar o relacionamento desejado entre o tamanho de uma cobertura de conjuntos ótima  $\mathcal{C}^*$  e o tamanho da cobertura de conjuntos  $\mathcal{C}$  retornada pelo algoritmo. Seja  $S_i$  o  $i$ -ésimo subconjunto selecionado por GREEDY-SET-COVER; o algoritmo incorre em um custo 1 quando acrescenta  $S_i$  a  $\mathcal{C}$ . Disseminamos esse custo selecionando  $S_i$  de modo uniforme entre os elementos cobertos pela primeira vez por  $S_i$ . Seja  $c_x$  o custo alocado ao elemento  $x$ , para cada  $x \in X$ . Cada elemento recebe a atribuição de um custo apenas uma vez, ao ser coberto pela primeira vez. Se  $x$  é coberto pela primeira vez por  $S_i$ , então

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|}.$$

Em cada etapa do algoritmo é atribuída uma unidade de custo, e assim

$$|\mathcal{C}| = \sum_{x \in X} c_x.$$

O custo atribuído à cobertura ótima é

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x$$

e, tendo em vista que cada  $x \in X$  está em pelo menos um conjunto  $S \in \mathcal{C}^*$ , temos

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x.$$

Combinando as duas desigualdades anteriores, temos que

$$|\mathcal{C}| = \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x. \quad (35.9)$$

O restante da prova se baseia na desigualdade fundamental a seguir, que provaremos em breve. Para qualquer conjunto  $S$  pertencente à família  $\mathcal{F}$ ,

$$\sum_{x \in S} c_x \leq H(|S|). \quad (35.10)$$

Das desigualdades (35.9) e (35.10), decorre que

$$\begin{aligned} |\mathcal{C}| &= \sum_{x \in \mathcal{C}^*} H(|S|), \\ &\leq |\mathcal{C}| H(\max\{|S| : S \in \mathcal{F}\}) \end{aligned}$$

provando assim o teorema.

Resta provar apenas a desigualdade (35.10). Considere qualquer conjunto  $S \in \mathcal{F}$  e  $i = 1, 2, \dots, |\mathcal{C}|$ , e seja

$$u_i = |S - (S_1 \cup S_2 \cup \dots \cup S_i)|$$

o número de elementos em  $S$  que permanecem descobertos depois de  $S_1, S_2, \dots, S_i$  terem sido selecionados pelo algoritmo. Definimos  $u_0 = |S|$  como o número de elementos de  $S$ , os quais estão todos inicialmente descobertos. Seja  $k$  o índice mínimo tal que  $u_k = 0$ , de modo que cada elemento em  $S$  seja coberto por pelo menos um dos conjuntos  $S_1, S_2, \dots, S_k$ . Então,  $u_{i-1} \geq u_i$ , e  $u_{i-1} - u_i$  elementos de  $S$  são cobertos pela primeira vez por  $S_i$ , para  $i = 1, 2, \dots, k$ . Desse modo,

$$\sum_{x \in S} c_x = \sum_{i=1}^k (\mathbf{u}_{i-1} - \mathbf{u}_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})|}.$$

Observe que

$$|S_i - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup \dots \cup S_{i-1})| \\ = u_{i-1},$$

porque a escolha gulosa de  $S_i$  garante que  $S$  não poderá cobrir mais elementos novos que  $S_i$  (do contrário,  $S$  teria sido escolhido em lugar de  $S_i$ ). Conseqüentemente, obtemos

$$\sum_{x \in S} c_x \leq \sum_{i=1}^k (\mathbf{u}_{i-1} - \mathbf{u}_i) \cdot \frac{1}{u_{i-1}}.$$

Agora, limitamos essa quantidade como a seguir:

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^k (\mathbf{u}_{i-1} - \mathbf{u}_i) \cdot \frac{1}{u_{i-1}}. \\ &= \sum_{i=1}^k \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\ &\leq \sum_{i=1}^k \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \quad (\text{porque } j \leq u_{i-1}) \\ &= \sum_{i=1}^k \left( \sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\ &= \sum_{i=1}^k (H(u_{i-1}) - H(u_i)) \\ &= H(u_0) - H(u_k) \quad (\text{porque a soma se encaixa}) \\ &= H(u_0) - H(0) \\ &= H(u_0) \quad (\text{porque } H(0) = 0) \\ &= H(|S|), \end{aligned}$$

O que completa a prova da desigualdade (35.10). ■

### **Corolário 35.5**

GREEDY-SET-COVER é um algoritmo de aproximação  $(\ln |X| + 1)$  de tempo polinomial.

**Prova** Use a desigualdade (A.14) e o Teorema 35.4. ■

Em algumas aplicações,  $\max\{|S| : S \in \mathcal{F}\}$  é uma constante pequena, e assim a solução retornada por GREEDY-SET-COVER é no máximo uma constante pequena vezes maior que ótima. Uma aplicação desse tipo ocorre quando essa heurística é usada para obter uma cobertura de vértices aproximada para um grafo cujos vértices têm grau no máximo 3. Nesse caso, a solução encontrada por GREEDY-SET-COVER não é maior que  $H(3) = 11/6$  vezes tão grande quanto uma solução ótima, uma garantia de desempenho ligeiramente melhor que a de APPROX-VERTEX-COVER.

## Exercícios

### 35.3-1

Considere cada uma das seguintes palavras um conjunto de letras: {arid, dash, drain, heard, lost, nose, shun, slate, snare, thread}. Mostre qual cobertura de conjuntos GREEDY-SET-COVER produz quando as ligações são rompidas em favor da palavra que aparece primeiro no dicionário.

### 35.3-2

Mostre que a versão de decisão do problema de cobertura de conjuntos é NP-completa, por redução a partir do problema de cobertura de vértices.

### 35.3-3

Mostre como implementar GREEDY-SET-COVER de tal maneira que ele seja executado no tempo  $O(\sum_{S \in \mathcal{F}} |S|)$ .

### 35.3-4

Mostre que a forma mais fraca do Teorema 35.4 a seguir é trivialmente verdadeira:

$$|\mathcal{C}| \leq |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\}$$

### 35.3-5

GREEDY-SET-COVER pode retornar várias soluções diferentes, dependendo de como rompemos as ligações na linha 4. Forneça um procedimento BAD-SET-COVER-INSTANCE( $n$ ) que retorne uma instância de  $n$  elementos do problema de cobertura de conjuntos para a qual, dependendo de como as ligações são rompidas na linha 4, GREEDY-SET-COVER pode retornar um número de soluções diferentes exponencial em  $n$ .

## 35.4 Aleatoriedade e programação linear

Nesta seção, estudaremos duas técnicas úteis no projeto de algoritmos de aproximação: a aleatoriedade e a programação linear. Apresentaremos um algoritmo aleatório simples para uma versão de otimização de satisfatibilidade 3 CNF, e em seguida usaremos a programação linear para ajudar a projetar um algoritmo de aproximação para uma versão ponderada do problema de cobertura de vértices. Esta seção só arranha a superfície dessas duas técnicas eficientes. As notas do capítulo apresentam referências para estudo adicional dessas áreas.

### Um algoritmo de aproximação aleatório para satisfatibilidade de MAX-3-CNF

Da mesma forma que existem algoritmos aleatórios que calculam soluções exatas, existem algoritmos aleatórios que calculam soluções aproximadas. Dizemos que um algoritmo aleatório para um problema tem uma *relação de aproximação*  $\rho(n)$  se, para qualquer entrada de tamanho  $n$ , o custo esperado  $C$  da solução produzida pelo algoritmo aleatório está dentro de um fator  $\rho(n)$  do custo  $C^*$  de uma solução ótima:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n). \quad (35.11)$$

Também chamamos um algoritmo aleatório que alcança uma relação de aproximação  $\rho(n)$  de *algoritmo aleatório de aproximação*  $\rho(n)$ . Em outras palavras, um algoritmo de aproximação aleatório é semelhante a um algoritmo de aproximação determinístico, exceto pelo fato de que a relação de aproximação se refere a um valor esperado.

Uma instância específica de satisfabilidade 3-CNF, definida na Seção 34.4, pode ou não ser satisfeita. Para ser satisfeita, deve haver uma atribuição das variáveis, de forma que toda cláusula seja avaliada como 1. Se uma instância não é satisfeita, talvez seja interessante calcular o quanto ela está “próxima” de ser satisfeita, ou seja, pode ser que se deseje encontrar uma atribuição das variáveis que satisfaça tantas cláusulas quanto possível. Chamamos o problema de maximização resultante **satisfabilidade MAX-3-CNF**. A entrada para a satisfabilidade MAX-3-CNF é a mesma da satisfabilidade 3-CNF, e o objetivo é retornar uma atribuição das variáveis que maximize o número de cláusulas avaliadas como 1. Agora, mostramos que a configuração aleatória de cada variável como 1 com probabilidade 1/2 e como 0 com probabilidade 1/2 é um algoritmo aleatório de aproximação 8/7. De acordo com a definição de satisfabilidade 3-CNF da Seção 34.4, exigimos que cada cláusula consista em exatamente três literais distintos. Além disso, supomos que nenhuma cláusula contém uma variável e sua negação. (O Exercício 35.4-1 lhe pede para remover essa última hipótese.)

### **Teorema 35.6**

Dada uma instância de satisfabilidade MAX-3-CNF com  $n$  variáveis  $x_1, x_2, \dots, x_n$  e  $m$  cláusulas, o algoritmo aleatório que define independentemente cada variável como 1 com probabilidade 1/2 e como 0 com probabilidade 1/2 é um algoritmo aleatório de aproximação 8/7.

**Prova** Suponha que definimos independentemente cada variável como 1 com probabilidade 1/2 e como 0 com probabilidade 1/2. Para  $i = 1, 2, \dots, n$ , definimos a variável indicadora aleatória

$$Y_i = I\{\text{a cláusula } i \text{ é satisfeita}\},$$

de forma que  $Y_i = 1$  desde que pelo menos um dos literais na  $i$ -ésima cláusula tenha sido definido como 1. Tendo em vista que nenhum literal aparece mais de uma vez na mesma cláusula, e como supomos que nenhuma variável e sua negação aparecem na mesma cláusula, as configurações dos três literais em cada cláusula são independentes. Uma cláusula não é satisfeita somente se todos os seus três literais são definidos como 0, e assim  $\Pr\{\text{cláusula } i \text{ não é satisfeita}\} = (1/2)^3 = 1/8$ . Desse modo,  $\Pr\{\text{cláusula } i \text{ é satisfeita}\} = 1 - 1/8 = 7/8$ . Então, pelo Lema 5.1,  $E[Y_i] = 7/8$ . Seja  $Y$  o número global de cláusulas satisfeitas, de forma que  $Y = Y_1 + Y_2 + \dots + Y_m$ . Então, temos

$$\begin{aligned} E[Y] &= E\left[\sum_{i=1}^m Y_i\right] \\ &= \sum_{i=1}^m E[Y_i] \quad (\text{por linearidade de expectativa}) \\ &= \sum_{i=1}^m 7/8 \\ &= 7m/8. \end{aligned}$$

É claro que  $m$  é um limite superior sobre o número de cláusulas satisfeitas e, consequentemente, a relação de aproximação é no máximo  $m/(7m/8) = 8/7$ . ■

### **Aproximação de cobertura de vértices ponderada com o uso de programação linear**

No problema de cobertura de vértices de peso mínimo, temos um grafo não orientado  $G = (V, E)$  em que cada vértice  $v \in V$  tem um peso positivo associado  $w(v)$ . Para qualquer cobertura de vértices  $V' \subseteq V$ , definimos o peso da cobertura de vértices  $w(V') = \sum_{v \in V'} w(v)$ . A meta é encontrar uma cobertura de vértices de peso mínimo.

Não podemos aplicar o algoritmo usado para cobertura de vértices não ponderado, nem podemos usar uma solução aleatória; ambos os métodos podem fornecer soluções que estão longe de serem ótimas. Porém, calcularemos um limite inferior sobre o peso da cobertura de vértices de peso mínimo, usando um programa linear. Em seguida, “arredondaremos” essa solução e a empregaremos para obter uma cobertura de vértices.

Suponha que associamos uma variável  $x(v)$  a cada vértice  $v \in V$ , e vamos exigir que  $x(v) \in \{0, 1\}$  para cada  $v \in V$ . Interpretamos  $x(v) = 1$  como o fato de  $v$  estar na cobertura de vértices, e interpretamos  $x(v) = 0$  como o fato de  $v$  não estar na cobertura de vértices. Então, podemos escrever a restrição de que, para qualquer aresta  $(u, v)$ , pelo menos um dentre  $u$  e  $v$  deve estar na cobertura de vértices como  $x(u) + x(v) \geq 1$ . Essa visão ocasiona o seguinte **programa de inteiros 0-1** para encontrar uma cobertura de vértices de peso mínimo:

$$\text{minimizar} \quad \sum_{v \in V} w(v)x(v) \quad (35.12)$$

sujeito a

$$x(u) + x(v) \geq 1 \quad \text{para cada } (u, v) \in E \quad (35.13)$$

$$x(v) \in \{0, 1\} \quad \text{para cada } (u, v) \in E. \quad (35.14)$$

Pelo Exercício 34.5-2, sabemos que encontrar valores de  $x(v)$  que satisfaçam a (35.13) e (35.14) é NP-difícil, e então essa formulação não é imediatamente útil. Suponha, porém, que removemos a restrição de que  $x(v) \in \{0, 1\}$  e a substituímos por  $0 \leq x(v) \leq 1$ . Assim, obtemos o seguinte programa linear, conhecido como um relaxamento de programação linear:

$$\text{minimizar} \quad \sum_{v \in V} w(v)x(v) \quad (35.15)$$

sujeito a

$$x(u) + x(v) \geq 1 \quad \text{para cada } v \in V \quad (35.16)$$

$$x(v) \leq 1 \quad \text{para cada } v \in V \quad (35.17)$$

$$x(v) \geq 0 \quad \text{para cada } v \in V. \quad (35.18)$$

Qualquer solução possível para o programa de inteiros 0-1 nas linhas (35.12) a (35.14) também é uma solução possível para o programa linear nas linhas (35.15) a (35.18). Então, uma solução ótima para o programa linear é um limite inferior sobre a solução ótima para o programa de inteiros 0-1, e consequentemente um limite inferior sobre uma solução ótima para o problema de cobertura de vértices de peso mínimo.

O procedimento a seguir usa a solução para o programa linear anterior com a finalidade de construir uma solução aproximada para o problema de cobertura de vértices de peso mínimo:

**APPROX-MIN-WEIGHT-VC( $G, w$ )**

- 1  $C \leftarrow \emptyset$
- 2 calcular  $\bar{x}$ , uma solução ótima para o programa linear nas linhas (35.15) a (35.18)
- 3 **for** cada  $v \in V$
- 4   **do if**  $\bar{x}(v) \geq 1/2$
- 5       **then**  $C \leftarrow C \cup \{v\}$
- 6 **return**  $C$

O procedimento APPROX-MIN-WEIGHT-VC funciona da maneira ilustrada a seguir. A linha 1 inicializa a cobertura de vértices como vazia. A linha 2 formula o programa linear nas linhas (35.15) a (35.18) e, em seguida, resolve esse programa linear. Uma solução ótima dá a cada vértice  $v$  um valor associado  $\bar{x}(v)$ , onde  $0 \leq \bar{x}(v) \leq 1$ . Usamos esse valor para guiar a escolha de quais vértices adicionar à cobertura de vértices  $C$  nas linhas 3 a 5. Se  $\bar{x}(v) \geq 1/2$ , adicionamos  $v$  a  $C$ ; caso contrário, não o fazemos. Na realidade, estamos “arredondando” cada variável fracionária na solução do programa linear para 0 ou 1, a fim de obter uma solução para o programa de inteiros 0-1 nas linhas (35.12) a (35.14). Finalmente, a linha 6 retorna a cobertura de vértices  $C$ .

### **Teorema 35.7**

O algoritmo APPROX-MIN-WEIGHT-VC é um algoritmo de aproximação 2 de tempo polinomial para o problema de cobertura de vértices de peso mínimo.

**Prova** Como existe um algoritmo de tempo polinomial para resolver o programa linear na linha 2, e como o loop **for** das linhas 3 a 5 é executado em tempo polinomial, APPROX-MIN-WEIGHT-VC é um algoritmo de tempo polinomial.

Agora mostramos que APPROX-MIN-WEIGHT-VC é um algoritmo de aproximação 2. Seja  $C^*$  uma solução ótima para o problema de cobertura de vértices de peso mínimo, e seja  $z^*$  o valor de uma solução ótima para o programa linear nas linhas (35.15) a (35.18). Tendo em vista que uma cobertura de vértices ótima é uma solução possível para o programa linear,  $z^*$  deve ser um limite inferior sobre  $w(C^*)$ , isto é,

$$z^* \leq w(C^*) . \quad (35.19)$$

Em seguida, afirmamos que, arredondando os valores fracionários das variáveis  $\bar{x}(v)$ , produzimos um conjunto  $C$  que é uma cobertura de vértices e satisfaz a  $w(C) \leq 2z^*$ . Para ver que  $C$  é uma cobertura de vértices, considere qualquer aresta  $(u, v) \in E$ . Pela restrição (35.16), sabemos que  $x(u) + x(v) \geq 1$ , o que implica que pelo menos um dentre  $\bar{x}(u)$  e  $\bar{x}(v)$  é no mínimo 1/2. Então, pelo menos um dentre  $u$  e  $v$  será incluído na cobertura de vértices, e então toda aresta será coberta.

Agora consideramos o peso da cobertura. Temos

$$\begin{aligned} z^* &= \sum_{v \in V} w(v) \bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\ &= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\ &= \frac{1}{2} \sum_{v \in C} w(v) \\ &= \frac{1}{2} w(C) . \end{aligned} \quad (35.20)$$

A combinação das desigualdades (35.19) e (35.20) nos dá

$$w(C) \leq 2z^* \leq 2w(C^*) ,$$

## Exercícios

### 35.4-1

Mostre que, mesmo se permitirmos a uma cláusula conter uma variável e sua negação, a configuração aleatória de cada variável como 1 com probabilidade 1/2 e como 0 com probabilidade 1/2 ainda será um algoritmo aleatório de aproximação 8/7.

### 35.4-2

O **problema de satisfabilidade MAX-CNF** é semelhante ao problema de satisfabilidade MAX-3-CNF, a não ser pelo fato de que ele não restringe cada cláusula a ter exatamente 3 literais. Forneça um algoritmo aleatório de aproximação 2 para o problema de satisfabilidade MAX-CNF.

### 35.4-3

No problema MAX-CUT, temos um grafo não orientado não ponderado  $G = (V, E)$ . Definimos um corte  $(S, V - S)$  como no Capítulo 23, e o **peso** de um corte como o número de arestas que cruzam o corte. A meta é encontrar um corte de peso máximo. Suponha que, para cada vértice  $v$ , inserimos  $v$  em  $S$  de forma aleatória e independente com probabilidade 1/2 e em  $V - S$  com probabilidade 1/2. Mostre que esse algoritmo é um algoritmo aleatório de aproximação 2.

### 35.4-4

Mostre que as restrições da linha (35.17) são redundantes no sentido de que, se as removermos do programa linear das linhas (35.15) a (35.18), qualquer solução ótima para o programa linear resultante deve satisfazer  $x(v) \leq 1$  para cada  $v \in V$ .

## 35.5 O problema de soma de subconjuntos

Uma instância do problema de soma de subconjuntos é um par  $(S, t)$ , onde  $S$  é um conjunto  $\{x_1, x_2, \dots, x_n\}$  de inteiros positivos e  $t$  é um inteiro positivo. Esse problema de decisão pergunta se existe um subconjunto de  $S$  que se adicione exatamente ao valor de destino  $t$ . Esse problema é NP-completo (consulte a Seção 34.5.5).

O problema de otimização associado a esse problema de decisão surge em aplicações práticas. No problema de otimização, desejamos encontrar um subconjunto de  $\{x_1, x_2, \dots, x_n\}$  cuja soma seja tão grande quanto possível, mas não maior que  $t$ . Por exemplo, podemos ter um caminhão que não pode transportar mais de  $t$  quilogramas, e ter  $n$  caixas diferentes para transportar, das quais a  $i$ -ésima caixa pesa  $x_i$  quilogramas. Desejamos encher o caminhão com a carga mais pesada possível, sem exceder o limite de peso dado.

Nesta seção, apresentaremos um algoritmo de tempo exponencial para esse problema de otimização, e depois mostraremos como modificar o algoritmo de forma que ele se torne um esquema de aproximação de tempo completamente polinomial. (Lembre-se de que um esquema de aproximação de tempo completamente polinomial tem um tempo de execução que é polinomial em  $1/\epsilon$ , bem como no tamanho da entrada.)

### Um algoritmo de tempo exponencial

Suponha que calculamos, para cada subconjunto  $S'$  de  $S$ , a soma dos elementos em  $S'$ , e então selecionamos, entre os subconjuntos cuja soma não excede  $t$ , aquele cuja soma fosse a mais próxima de  $t$ . É claro que esse algoritmo retornaria a solução ótima, mas ele poderia demorar um tempo exponencial. Para implementar esse algoritmo, poderíamos utilizar um procedimento iterativo que, na iteração  $i'$ , calculasse as somas de todos os subconjuntos de  $\{x_1, x_2, \dots, x_i\}$ , usando como ponto de partida as somas de todos os subconjuntos de  $\{x_1, x_2, \dots, x_{i-1}\}$ . Ao fazer isso, perceberíamos que, uma vez que um determinado subconjunto  $S'$  tivesse uma soma que excedesse  $t$ , não haveria razão para mantê-lo, pois nenhum superconjunto de  $S'$  poderia ser a solução ótima. Damos agora uma implementação dessa estratégia.

O procedimento EXACT-SUBSET-SUM toma um conjunto de entrada  $S = \{x_1, x_2, \dots, x_n\}$  e um valor de destino  $t$ . Veremos seu pseudocódigo em breve. Esse procedimento calcula iterativamente  $L_i$ , a lista de somas de todos os subconjuntos de  $\{x_1, \dots, x_i\}$  que não excedem  $t$ , e depois retorna o valor máximo em  $L_n$ .

Se  $L$  é uma lista de inteiros positivos e  $x$  é outro inteiro positivo, então seja  $L + x$  a lista de inteiros derivados de  $L$  aumentando-se cada elemento de  $L$  por  $x$ . Por exemplo, se  $L = \langle 1, 2, 3, 5, 9 \rangle$ , então  $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$ . Também usamos essa notação para conjuntos, de forma que

$$S + x + \{s + x : s \in S\}.$$

Usamos um procedimento auxiliar MERGE-LISTS( $L, L'$ ) que retorna a lista ordenada que é o resultado da intercalação de suas duas listas ordenadas de entrada  $L$  e  $L'$ , com a remoção de valores duplicados. Como o procedimento MERGE que usamos na ordenação por intercalação (Seção 2.3.1), MERGE-LISTS é executado no tempo  $O(|L| + |L'|)$ . (Omitimos o pseudocódigo para MERGE-LISTS.)

**EXACT-SUBSET-SUM( $S, t$ )**

```

1  $n \leftarrow |S|$ 
2  $L_0 \leftarrow \langle 0 \rangle$ 
3 for  $i \leftarrow 1$  to  $n$ 
4   do  $L_i \leftarrow \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$ 
5   remove de  $L_i$  todo elemento maior que  $t$ 
6 return o maior elemento em  $L_n$ 
```

Para ver como EXACT-SUBSET-SUM funciona, seja  $P_i$  o conjunto de todos os valores que podem ser obtidos pela seleção de um subconjunto (possivelmente vazio) de  $\{x_1, x_2, \dots, x_i\}$  e somando-se seus elementos. Por exemplo, se  $S = \{1, 4, 5\}$ , então

$$\begin{aligned} P_1 &= \{0, 1\}, \\ P_2 &= \{0, 1, 4, 5\}, \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\}. \end{aligned}$$

Dada a identidade

$$P_i = P_{i-1} \cup (P_{i-1} + x_i), \quad (35.21)$$

podemos provar por indução sobre  $i$  (ver Exercício 35.5-1) que a lista  $L_i$  é uma lista ordenada contendo cada elemento de  $P_i$ , cujo valor não é maior que  $t$ . Como o comprimento de  $L_i$  pode ser até  $2^i$ , EXACT-SUBSET-SUM é um algoritmo de tempo exponencial em geral, embora seja um algoritmo de tempo polinomial nos casos especiais em que  $t$  é polinomial em  $|S|$  ou todos os números em  $S$  são limitados por um polinômio em  $|S|$ .

## Um esquema de aproximação de tempo completamente polinomial

Podemos derivar um esquema de aproximação de tempo completamente polinomial para o problema da soma de subconjuntos, “desbastando” cada lista  $L_i$  depois que ela é criada. A idéia é que se dois valores em  $L$  estão perto um do outro, então não existe nenhuma razão para mantê-los de forma explícita, com a finalidade de encontrar uma solução aproximada. Mais precisamente, usamos um parâmetro de desbaste  $\delta$  tal que  $0 < \delta < 1$ . **Desbastar** uma lista  $L$  por  $\delta$  significa remover tantos elementos de  $L$  quanto possível, de tal forma que, se  $L'$  é o resultado do desbaste de  $L$ , para cada elemento  $y$  que foi removido de  $L$ , ainda exista em  $L'$  um elemento  $z$  que se aproxime de  $y$ , ou seja,

$$\frac{y}{1+\delta} \leq z \leq y . \quad (35.22)$$

Podemos imaginar tal valor  $z$  como um “representante” de  $y$  na nova lista  $L'$ . Cada  $y$  é representado por um  $z$  que satisfaz à desigualdade (35.22). Por exemplo, se  $\delta = 0,1$  e

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle ,$$

então podemos desbastar  $L$  para obter

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle ,$$

onde o valor 11 eliminado é representado por 10, os valores 21 e 22 eliminados são representados por 20, e o valor 24 eliminado é representado por 23. Como todo elemento da versão desbastada da lista também é um elemento da versão original da lista, desbastar uma lista pode reduzir de forma drástica o número de elementos na lista, ao mesmo tempo que mantém um valor representativo próximo (e ligeiramente menor) na lista para cada elemento eliminado.

O procedimento a seguir desbasta uma lista de entrada  $L = \langle y_1, y_2, \dots, y_m \rangle$  no tempo  $\Theta(m)$ , dados  $L$  e  $\delta$  e supondo-se que  $L$  esteja ordenada em seqüência monotonicamente crescente. A saída do procedimento é uma lista desbastada e ordenada.

```
TRIM( $L, \delta$ )
1  $m \leftarrow |L|$ 
2  $L' \leftarrow \langle y_1 \rangle$ 
3  $\text{último} \leftarrow y_1$ 
4 for  $i \leftarrow 2$  to  $m$ 
5   do if  $y_i > \text{último} \cdot (1 + \delta)$        $\triangleright y_i \geq \text{último}$  porque  $L$  está ordenada
6     then anexar  $y_i$  ao final de  $L'$ 
7      $\text{último} \leftarrow y_i$ 
8 return  $L'$ 
```

Os elementos de  $L$  são examinados em ordem crescente, e um número é inserido na lista retornada  $L'$  somente se ele é o primeiro elemento de  $L$  ou se não pode ser representado pelo número mais recente inserido em  $L'$ .

Dado o procedimento TRIM, podemos construir nosso esquema de aproximação da maneira descrita a seguir. Esse procedimento toma como entrada um conjunto  $S = \{x_1, x_2, \dots, x_n\}$  de  $n$  inteiros (em ordem arbitrária), um inteiro de destino  $t$ , e um “parâmetro de aproximação”  $\varepsilon$ , onde

$$0 < \varepsilon < 1 . \quad (35.23)$$

Ele retorna um valor  $z$  que está dentro de um fator  $1 + \varepsilon$  da solução ótima.

```
APPROX-SUBSET-SUM( $S, t, \varepsilon$ )
1  $n \leftarrow |S|$ 
2  $L_0 \leftarrow \langle 0 \rangle$ 
3 for  $i \leftarrow 1$  to  $n$ 
4   do  $L_i \leftarrow \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$ 
5    $L_i \leftarrow \text{TRIM}(L_i, \varepsilon/2n)$ 
6   remover de  $L_i$  todo elemento maior que  $t$ 
7   seja  $z^*$  o maior valor em  $L_n$ 
8 return  $z^*$ 
```

A linha 2 inicializa a lista  $L_0$  como a lista que contém apenas o elemento 0. O loop **for** das linhas 3 a 6 tem o efeito de calcular  $L_i$  como uma lista ordenada contendo uma versão adequadamente desbastada do conjunto  $P_i$ , com todos os elementos maiores que  $t$  removidos. Como  $L_i$  é criado a partir de  $L_{i-1}$ , devemos assegurar que o desbaste repetido não introduzirá imprecisão excessiva. Em breve, veremos que APPROX-SUBSET-SUM retorna uma aproximação correta, se ela existir.

Como exemplo, suponha que temos a instância

$$S = \langle 104, 102, 201, 101 \rangle$$

com  $t = 308$  e  $\varepsilon = 0,40$ . O parâmetro de desbaste  $\delta$  é  $\varepsilon/8 = 0.05$ . APPROX-SUBSET-SUM calcula os valores a seguir nas linhas indicadas:

linha 2:  $L_0 = \langle 0 \rangle$ ,

linha 4:  $L_1 = \langle 0, 104 \rangle$ ,

linha 5:  $L_1 = \langle 0, 104 \rangle$ ,

linha 6:  $L_1 = \langle 0, 104 \rangle$ ,

linha 4:  $L_2 = \langle 0, 102, 104, 206 \rangle$

linha 5:  $L_2 = \langle 0, 102, 206 \rangle$

linha 6:  $L_2 = \langle 0, 102, 206 \rangle$

linha 4:  $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$

linha 5:  $L_3 = \langle 0, 102, 201, 303, 407 \rangle$

linha 6:  $L_3 = \langle 0, 102, 201, 303 \rangle$

linha 4:  $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$

linha 5:  $L_4 = \langle 0, 101, 201, 302, 404 \rangle$

linha 6:  $L_4 = \langle 0, 101, 201, 302 \rangle$ .

O algoritmo retorna  $z^* = 302$  como sua resposta, que está dentro de  $\varepsilon = 40\%$  da resposta ótima  $307 = 104 + 102 + 101$ ; de fato, ela está dentro 2%.

### **Teorema 35.8**

APPROX-SUBSET-SUM é um esquema de aproximação de tempo completamente polinomial para o problema da soma de subconjuntos.

**Prova** As operações de desbaste  $L_i$  na linha 5 e remoção de todo elemento  $L_i$  maior que  $t$  mantêm a propriedade de que todo elemento de  $L_i$  também é um elemento de  $P_i$ . Então, o valor  $z$  retornado na linha 8 é de fato a soma de algum subconjunto de  $S$ . Seja  $y^* \in P_n$  uma solução ótima para o problema da soma de subconjuntos. Então, pela linha 6, sabemos que  $z^* \leq y^*$ . Pela desigualdade (35.1), precisamos mostrar que  $y^*/z^* \leq 1 + \varepsilon$ . Também devemos mostrar que o algoritmo é executado em tempo polinomial, tanto em  $1/\varepsilon$  quanto no tamanho da entrada.

Por indução sobre  $i$ , é possível mostrar que, para todo elemento  $y$  em  $P_i$ , que é no máximo  $t$ , existe um  $z \in L_i$  tal que

$$\frac{y}{(1 + \varepsilon/2n)^i} \leq z \leq y \tag{35.24}$$

(veja o Exercício 35.5-2). A desigualdade (35.24) deve ser válida para  $y^* \in P_n$ , e então existe um  $z$

$$\frac{y^*}{(1+\varepsilon/2n)^n} \leq z \leq y^*,$$

e, portanto,

$$\frac{y^*}{z} \leq \left(1 + \frac{\varepsilon}{2n}\right)^n. \quad (35.25)$$

Tendo em vista que existe um  $z \in L_n$  que satisfaz à desigualdade (35.25), a desigualdade deve ser válida para  $z^*$ , que é o maior valor em  $L_n$ ; isto é,

$$\frac{y^*}{z^*} \leq \left(1 + \frac{\varepsilon}{2n}\right)^n. \quad (35.26)$$

Resta mostrar que  $y^*/z^* \leq 1 + \varepsilon$ . Fazemos isso mostrando que  $(1 + \varepsilon/2n)^n \leq 1 + \varepsilon$ . Pela equação (3.13), temos  $\lim_{n \rightarrow \infty} (1 + \varepsilon/2n)^n \leq e^{\varepsilon/2}$ . Tendo em vista que é possível demonstrar que

$$\frac{d}{dn} \left(1 + \frac{\varepsilon}{2n}\right)^n > 0, \quad (35.27)$$

a função  $(1 + \varepsilon/2n)^n$  cresce com  $n$  à medida que seu limite se aproxima de  $e^{\varepsilon/2}$ , e temos

$$\begin{aligned} \left(1 + \frac{\varepsilon}{2n}\right)^n &\leq e^{\varepsilon/2} \\ &\leq 1 + \varepsilon/2 + (\varepsilon/2)^2 && \text{(pela desigualdade (3.12))} \\ &\leq 1 + \varepsilon && \text{(pela desigualdade (35.23))}. \end{aligned} \quad (35.28)$$

A combinação das desigualdades (35.26) e (35.28) completa a análise da relação de aproximação.

Para mostrar que APPROX-SUBSET-SUM é um esquema de aproximação de tempo completamente polinomial, derivamos um limite sobre o comprimento de  $L_i$ . Depois do desbaste, elementos consecutivos  $z$  e  $z'$  de  $L_i$  devem ter o relacionamento  $z'/z > 1 + \varepsilon/2n$ . Isto é, eles devem diferir por um fator de pelo menos  $1 + \varepsilon/2n$ . Cada lista, então, contém o valor 0, possivelmente o valor 1, e até  $\lfloor \log_{1 + \varepsilon/2n} t \rfloor$  valores adicionais. O número de elementos em cada lista  $L_i$  é no máximo

$$\begin{aligned} \log_{1 + \varepsilon/2n} t + 2 &= \frac{\ln t}{\ln(1 + \varepsilon/2n)} + 2 \\ &\leq \frac{2n(1 + \varepsilon/2n) \ln t}{\varepsilon} + 2 && \text{(pela desigualdade (3.16))} \\ &\leq \frac{4n \ln t}{\varepsilon} + 2 && \text{(pela desigualdade (35.23))}. \end{aligned}$$

Esse limite é polinomial no tamanho da entrada – que é o número de bits  $\lg t$  necessários para representar  $t$ , mais o número de bits necessários para representar o conjunto  $S$ , que é por sua vez polinomial em  $n$  – e em  $1/\varepsilon$ . Tendo em vista que o tempo de execução de APPROX-SUBSET-SUM é polinomial no comprimento de  $L_i$ , APPROX-SUBSET-SUM é um esquema de aproximação de tempo completamente polinomial.

## Exercícios

### 35.5-1

Prove a equação (35.21). Em seguida mostre que, depois de se executar a linha 5 de EXACT-SUBSET-SUM,  $L_i$  é uma lista ordenada contendo todo elemento de  $P_i$  cujo valor não é maior que  $t$ .

### 35.5-2

Prove a desigualdade (35.24).

### 35.5-3

Prove a desigualdade (35.27).

### 35.5-4

Como você modificaria o esquema de aproximação apresentado nesta seção, a fim de encontrar uma boa aproximação para o menor valor não menor que  $t$  que seja uma soma de algum subconjunto da lista de entrada dada?

## Problemas

### 35-1 Pacote de caixas

Suponha que temos um conjunto de  $n$  objetos, onde o tamanho  $s_i$  do  $i$ -ésimo objeto satisfaz a  $0 < s_i < 1$ . Desejamos empacotar todos os objetos no número mínimo de caixas de tamanho unitário. Cada caixa pode conter qualquer subconjunto dos objetos cujo tamanho total não excede 1.

- Prove que o problema de determinar o número mínimo de caixas exigidas é NP-difícil. (*Sugestão:* Reduza a partir do problema da soma de subconjuntos.)

A heurística de *primeiro a caber* toma cada objeto por sua vez e o insere na primeira caixa que possa acomodá-lo. Seja  $S = \sum_{i=1}^n s_i$ .

- Demonstre que o número ótimo de caixas necessárias é pelo menos  $\lceil S \rceil$ .
- Demonstre que a heurística de primeiro a caber deixa no máximo uma caixa cheia até menos da metade.
- Prove que o número de caixas usadas pela heurística de primeiro a caber nunca é maior que  $\lceil 2S \rceil$ .
- Prove uma relação de aproximação 2 para a heurística de primeiro a caber.
- Forneça uma implementação eficiente da heurística primeiro a caber, e analise seu tempo de execução.

### 35-2 Aproximação do tamanho de um grupo exclusivo máximo

Seja  $G = (V, E)$  um grafo não orientado. Para qualquer  $k \geq 1$  definimos  $G^{(k)}$  o grafo não orientado  $(V^{(k)}, E^{(k)})$ , onde  $V^{(k)}$  é o conjunto de todas as  $k$ -tuplas ordenadas de vértices de  $V$ , e  $E^{(k)}$  é definido de modo que  $(v_1, v_2, \dots, v_k)$  seja adjacente a  $(w_1, w_2, \dots, w_k)$  se e somente se, para algum  $i$ , o vértice  $v_i$  é adjacente a  $w_i$  em  $G$ , ou então  $v_i = w_i$ .

- Prove que o tamanho do grupo exclusivo máximo em  $G^{(k)}$  é igual à  $k$ -ésima potência do tamanho do grupo exclusivo máximo em  $G$ .
- Demonstre que, se existe um algoritmo de aproximação que tenha uma relação de aproximação constante para encontrar um grupo exclusivo de tamanho máximo, então existe um esquema de aproximação de tempo completamente polinomial para o problema.

### 35-3 Problema de cobertura de conjuntos ponderada

Suponha que o problema de cobertura de conjuntos seja generalizado de forma que cada conjunto  $S_i$  na família  $\mathcal{F}$  tenha um peso associado  $w_i$ , e o peso de uma cobertura  $\mathcal{C}$  seja  $\sum_{S_i \in \mathcal{C}} w_i$ . De-

sejamos determinar uma cobertura de peso mínimo. (A Seção 35.3 trata o caso em que  $w_i = 1$  para todo  $i$ .)

Mostre que a heurística gulosa de cobertura de conjuntos pode ser generalizada de maneira natural para fornecer uma solução aproximada para qualquer instância do problema de cobertura de conjuntos ponderada. Mostre que sua heurística tem uma relação de aproximação  $H(d)$ , onde  $d$  é o tamanho máximo de qualquer conjunto  $S_i$ .

### 35-4 Emparelhamento de máximo

Vimos que, para um grafo não orientado  $G$ , um emparelhamento é um conjunto de arestas tal que não há duas arestas no conjunto incidentes no mesmo vértice. Na Seção 26.3, vimos como encontrar um emparelhamento de máximo em um grafo bipartido. Neste problema, examinaremos emparelhamentos em grafos não orientados em geral (isto é, não se exige que os grafos sejam bipartidos).

- Um **emparelhamento máximo** é um emparelhamento que não é um subconjunto próprio de qualquer outro emparelhamento. Mostre que um emparelhamento máximo não precisa ser um emparelhamento de máximo, exibindo um grafo não orientado  $G$  e uma emparelhamento máximo  $M$  em  $G$  que não seja um emparelhamento de máximo. (Existe tal grafo com apenas quatro vértices.)
- Considere um grafo não orientado  $G = (V, E)$ . Forneça um algoritmo guloso de tempo  $O(E)$  para encontrar um emparelhamento máximo em  $G$ .

Neste problema, concentraremos nossa atenção em um algoritmo de aproximação de tempo polinomial para emparelhamento de máximo. Enquanto o algoritmo mais rápido conhecido para emparelhamento de máximo demora um tempo superlinear (mas polinomial), o algoritmo de aproximação aqui apresentado funcionará em tempo linear. Você mostrará que o algoritmo guloso de tempo linear para emparelhamento máximo da parte (b) é um algoritmo de aproximação 2 para emparelhamento de máximo.

- Mostre que o tamanho de um emparelhamento de máximo em  $G$  é um limite inferior sobre o tamanho de qualquer cobertura de vértice para  $G$ .
- Considere um emparelhamento máximo  $M$  em  $G = (V, E)$ . Seja

$$T = \{v \in V : \text{alguma aresta em } M \text{ é incidente em } v\}.$$

O que você pode dizer sobre o subgrafo de  $G$  induzido pelos vértices de  $G$  que não estão em  $T$ ?

- Conclua da parte (d) que  $2|M|$  é o tamanho de uma cobertura de vértices para  $G$ .
- Usando as partes (c) e (e), prove que o algoritmo guloso da parte (b) é um algoritmo de aproximação 2 para emparelhamento de máximo.

### 35-5 Programação de máquinas paralelas

No **problema de programação de máquinas paralelas**, temos  $n$  trabalhos,  $J_1, J_2, \dots, J_n$ , onde cada trabalho  $J_k$  tem um tempo de processamento não negativo associado  $p_k$ . Também temos  $m$  máquinas idênticas,  $M_1, M_2, \dots, M_m$ . Uma **programação** específica, para cada trabalho  $J_k$ , a máquina em que ele é executado e o período de tempo durante o qual é executado. Cada trabalho  $J_k$  deve ser executado em alguma máquina  $M_i$  durante  $p_k$  unidades de tempo consecutivas e, durante esse período de tempo, nenhum outro trabalho pode ser executado em  $M_i$ . Seja  $C_k$  o **tempo de conclusão** do trabalho  $J_k$ , isto é, o tempo em que o trabalho  $J_k$  completa o processamento. Dada uma programação, definimos  $C_{\max} = \max_{1 \leq k \leq n} C_k$  como a **duração** da programação. A meta é encontrar uma programação cuja duração seja mínima.

Por exemplo, suponha que temos duas máquinas  $M_1$  e  $M_2$  e que temos quatro trabalhos  $J_1, J_2, J_3, J_4$ , com  $p_1 = 2, p_2 = 12, p_3 = 4$  e  $p_4 = 5$ . Então uma programação possível executa, na máquina

$M_1$ , o trabalho  $J_1$  seguido pelo trabalho  $J_2$  e, na máquina  $M_2$ , ela executa o trabalho  $J_4$  seguido pelo trabalho  $J_3$ . Para essa programação,  $C_1 = 2$ ,  $C_2 = 14$ ,  $C_3 = 9$ ,  $C_4 = 5$  e  $C_{\max} = 14$ . Uma programação ótima executa  $J_2$  na máquina  $M_1$ , e executa os trabalhos  $J_1, J_3$  e  $J_4$  na máquina  $M_2$ . Para essa programação,  $C_1 = 2$ ,  $C_2 = 12$ ,  $C_3 = 6$ ,  $C_4 = 11$  e  $C_{\max} = 12$ .

Dado um problema de programação de máquinas paralelas, seja  $C^*_{\max}$  a duração de uma programação ótima.

- a. Mostre que a duração ótima é pelo menos tão grande quanto o maior tempo de processamento, isto é,

$$C^*_{\max} \geq \max_{1 \leq k \leq n} p_k .$$

- b. Mostre que a duração ótima é pelo menos tão grande quanto a carga média da máquina, isto é,

$$C^*_{\max} \geq \frac{1}{m} \sum_{1 \leq k \leq n} p_k .$$

Suponha que usamos o algoritmo guloso a seguir para programação de máquinas paralelas: sempre que uma máquina estiver ociosa, programar qualquer trabalho que ainda não tenha sido programado.

- c. Escreva pseudocódigo para implementar esse algoritmo guloso. Qual é o tempo de execução de seu algoritmo?
- d. Para a programação retornada pelo algoritmo guloso, mostre que

$$C_{\max} \geq \frac{1}{m} \sum_{1 \leq k \leq n} p_k + \max_{1 \leq k \leq n} p_k .$$

Conclua que esse algoritmo é um algoritmo de aproximação 2 de tempo polinomial.

## Notas do capítulo

Embora métodos que não calculam necessariamente soluções exatas sejam conhecidos há milhares de anos (por exemplo, métodos para aproximar o valor de  $\pi$ , a noção de um algoritmo de aproximação é muito mais recente. Hochbaum credita a Garey, Graham e Ullman [109], e a Johnson [166] a formalização do conceito de um algoritmo de aproximação de tempo polinomial. O primeiro algoritmo desse tipo é freqüentemente creditado a Graham [129]. e é o assunto do Problema 35-5.

Desde esse primeiro trabalho, têm sido projetados milhares de algoritmos de aproximação para uma ampla faixa de problemas, e existe muita literatura sobre esse campo. Textos recentes de Ausiello *et al.* [25], Hochbaum [149] e Vazirani [305] lidam exclusivamente com algoritmos de aproximação, como também as pesquisas de Shmoys [277] e de Klein e Young [181]. Vários outros textos, como os de Garey e Johnson [110] e de Papadimitriou e Steiglitz [237] também têm uma cobertura significativa de algoritmos de aproximação. Lawler, Lenstra, Rinnooy Kan e Shmoys [197] apresentam um tratamento extensivo de algoritmos de aproximação para o problema do caixeiro-viajante.

Papadimitriou e Steiglitz atribuem o algoritmo APPROX-VERTEX-COVER a F. Gavril e M. Yannakakis. O problema de cobertura de vértices foi estudado extensivamente (Hochbaum [149] lista 16 algoritmos de aproximação diferentes para esse problema), mas todas as relações de aproximação são pelo menos  $2 - o(1)$ .

O algoritmo APPROX-TSP-TOUR aparece em um artigo de Rosenkrantz, Stearns e Lewis [261]. Christofides otimizou esse algoritmo e forneceu um algoritmo de aproximação  $3/2$  para o problema do caixeiro-viajante com a desigualdade de triângulos. Arora [21] e Mitchell [223]

mostram que, se os pontos estão no plano euclidiano, existe um esquema de aproximação de tempo polinomial. O Teorema (35.3) se deve a Sahni e Gonzalez [264].

A análise da heurística gulosa para o problema de cobertura de conjuntos foi modelada com base na prova publicada por Chvátal [61] de um resultado mais geral; esse resultado básico com a forma apresentada aqui se deve a Johnson [166] e Lovász [206].

O algoritmo APPROX-SUBSET-SUM e sua análise foram modelados livremente com base em algoritmos de aproximação relacionados para os problemas da mochila e de soma de subconjuntos por Ibarra e Kim [164].

O algoritmo aleatório para satisfabilidade MAX-3-CNF está implícito no trabalho de Johnson [166]. O algoritmo de cobertura de vértices ponderada é de Hochbaum [148]. A Seção 35.4 apenas toca no poder da aleatoriedade e da programação linear no projeto de algoritmos de aproximação. Uma combinação dessas duas idéias gera uma técnica chamada “arredondamento aleatório”, na qual um problema é formulado primeiro como um programa linear de inteiros. O relaxamento de programação linear é então resolvido, e as variáveis na solução são interpretadas como probabilidades. Essas probabilidades são então usadas para ajudar a orientar a solução do problema original. Essa técnica foi usada inicialmente por Raghavan e Thompson [255], e teve muitos usos subseqüentes. (Consulte Motwani, Naor e Raghavan [227] para ver uma pesquisa na área.) Várias outras idéias notáveis no campo de algoritmos de aproximação incluem o método dual primitivo (consulte [116] para ver uma pesquisa), a localização de cortes esparsos para uso em algoritmos de dividir e conquistar [199] e o uso de programação semidefinida [115].

Como mencionamos nas notas do capítulo correspondentes ao Capítulo 34, resultados recentes em demonstrações que podem ser verificadas de modo probabilístico levaram a limites inferiores sobre a possibilidade de aproximação de muitos problemas, inclusive vários deste capítulo. Além das referências existentes, o capítulo de Arora e Lund [22] contém uma boa descrição do relacionamento entre demonstrações que podem ser verificadas por meio probabilísticos e a dificuldade de aproximação de diversos problemas.



---

## *Parte VIII*

# *Apêndice: Fundamentos de matemática*

### **Introdução**

A análise de algoritmos freqüentemente exige a utilização de um conjunto de ferramentas matemáticas. Algumas dessas ferramentas são tão simples quanto a álgebra do ensino de segundo grau, mas outras talvez sejam novas para você. Este apêndice é um compêndio de vários outros conceitos e métodos que empregamos para analisar algoritmos. Conforme observamos na introdução à Parte I, é possível que você tenha visto grande parte do material deste apêndice antes de ler este livro (embora as convenções específicas de notação que utilizamos possam diferir ocasionalmente do que você encontrou em outros livros). Conseqüentemente, você deve tratar o conteúdo deste apêndice como material de referência. No entanto, como no restante do livro, incluímos exercícios e problemas para que você possa melhorar seus conhecimentos nessas áreas específicas.

O Apêndice A oferece métodos para avaliação e delimitação de somatórios, os quais surgirão com freqüência na análise de algoritmos. Muitas das fórmulas desse capítulo poderão ser encontradas em qualquer texto de cálculo, mas você achará conveniente ter esses métodos compilados em um único lugar.

O Apêndice B contém definições e notações básicas para conjuntos, relações, funções, grafos e árvores. Esse capítulo também apresenta algumas propriedades básicas desses objetos matemáticos.

O Apêndice C começa com princípios elementares de contagem: permutações, combinações e assuntos semelhantes. O restante do capítulo contém definições e propriedades de probabilidade básica. A maior parte dos algoritmos deste livro não exige nenhum conhecimento de probabilidade para sua análise e, desse modo, você poderá omitir facilmente as últimas seções do capítulo em uma primeira leitura, até mesmo sem folheá-las. Mais tarde, quando encontrar uma análise probabilística e desejar compreendê-la melhor, você encontrará no Apêndice C um guia bem organizado para fins de referência.



---

## *Apêndice A*

### *Somatórios*

Quando um algoritmo contém uma construção de controle iterativo (ou repetitivo) como um loop **while** ou **for**, seu tempo de execução pode ser expresso como a soma dos tempos gastos em cada execução do corpo do loop. Por exemplo, descobrimos na Seção 2.2 que a  $j$ -ésima iteração da ordenação por inserção demorou um tempo proporcional a  $j$  no pior caso. Somando o tempo gasto em cada iteração, obtivemos o somatório (ou a série)

$$\sum_{j=2}^n j .$$

A avaliação desse somatório produziu um limite de  $\Theta(n^2)$  no tempo de execução do pior caso do algoritmo. Esse exemplo indica a importância geral de se entender como manipular e limitar somatórios.

A Seção A.1 lista diversas fórmulas básicas que envolvem somatórios. A Seção A.2 oferece técnicas úteis para limitar somatórios. As fórmulas da Seção A.1 são dadas sem demonstração, embora as provas de algumas delas sejam apresentadas na Seção A.2 para ilustrar os métodos dessa seção. A maioria das outras provas pode ser encontrada em qualquer texto de cálculo.

#### **A.1 Fórmulas e propriedades de somatórios**

Dada uma seqüência de números  $a_1, a_2, \dots$ , a soma finita  $a_1 + a_2 + \dots + a_n$ , onde  $n$  é um inteiro não negativo, pode ser escrita como

$$\sum_{k=1}^n a_k .$$

Se  $n = 0$ , o valor do somatório é definido como 0. O valor de uma série finita é sempre bem definido, e seus termos podem ser somados em qualquer ordem.

Dada uma seqüência de números  $a_1, a_2, \dots$ , a soma infinita  $a_1 + a_2 + \dots$  pode ser escrita como

$$\sum_{k=1}^{\infty} a_k ,$$

que é interpretada com o significado

$$\lim_{n \rightarrow \infty} \sum_{k=1}^n a_k .$$

Se o limite não existe, a série ***diverge***; caso contrário, ela ***converge***. Os termos de uma série convergente nem sempre podem ser adicionados em qualquer ordem. Contudo, podemos reorganizar os termos de uma **série absolutamente convergente**, ou seja, uma série  $\sum_{k=1}^{\infty} a_k$  para a qual a série  $\sum_{k=1}^{\infty} |a_k|$  também converge.

## Linearidade

Para qualquer número real  $c$  e quaisquer seqüências finitas  $a_1, a_2, \dots, a_n$  e  $b_1, b_2, \dots, b_n$ ,

$$\sum_{k=1}^n (ca_k + b_k) = c \sum_{k=1}^n a_k + \sum_{k=1}^n b_k .$$

A propriedade de linearidade também é obedecida por séries convergentes infinitas.

A propriedade de linearidade pode ser explorada para manipular somatórios que incorporam notação assintótica. Por exemplo,

$$\sum_{k=1}^n \Theta(f(k)) = \Theta\left(\sum_{k=1}^n f(k)\right).$$

Nessa equação, a notação  $\Theta$  no lado esquerdo se aplica à variável  $k$  mas, no lado direito, ela se aplica a  $n$ . Tais manipulações também podem ser aplicadas a séries convergentes infinitas.

## Série aritmética

O somatório

$$\sum_{k=1}^n k = 1 + 2 + \dots + n ,$$

é uma **série aritmética** e tem o valor

$$\sum_{k=1}^n k = \frac{1}{2} n(n+1) \tag{A.1}$$

$$= \Theta(n^2) . \tag{A.2}$$

## Somas de quadrados e cubos

Temos os seguintes somatórios de quadrados e cubos:

$$\sum_{k=0}^n k^2 = \frac{n(n+1)(2n+1)}{6} , \tag{A.3}$$

$$\sum_{k=0}^n k^3 = \frac{n^2(n+1)^2}{4} . \tag{A.4}$$

## Série geométrica

Para o real  $x \neq 1$ , o somatório

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n$$

é uma **série geométrica** ou **exponencial** e tem o valor

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}. \quad (\text{A.5})$$

Quando o somatório é infinito e  $|x| < 1$ , temos a série geométrica infinitamente decrescente

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}. \quad (\text{A.6})$$

## Série harmônica

Para inteiros positivos  $n$ , o  $n$ -ésimo **número harmônico** é

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} \\ &= \sum_{k=1}^n \frac{1}{k} \\ &= \ln n + O(1). \end{aligned} \quad (\text{A.7})$$

(Provaremos esse limite na Seção A.2.)

## Integração e diferenciação de séries

Fórmulas adicionais podem ser obtidas por integração ou diferenciação das fórmulas anteriores. Por exemplo, diferenciando-se ambos os lados da série geométrica infinita (A.6) e multiplicando por  $x$ , obtemos

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad (\text{A.8})$$

para  $|x| < 1$ .

## Como inserir séries

Para qualquer seqüência  $a_1, a_2, \dots, a_n$ ,

$$\sum_{k=1}^{\infty} (a_k - a_{k-1}) = a_n - a_0, \quad (\text{A.9})$$

desde que cada um dos termos  $a_1, a_2, \dots, a_{n-1}$  seja adicionado exatamente uma vez e subtraído exatamente uma vez. Dizemos que a soma se **insere**. De modo semelhante,

$$\sum_{k=1}^{n-1} (a_k - a_{k-1}) = a_0 - a_n.$$

Como exemplo de uma soma por inserção, considere a série

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

Tendo em vista que podemos reescrever cada termo como

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1},$$

obtemos

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left( \frac{1}{k} - \frac{1}{k+1} \right).$$

## Produtos

O produto finito  $a_1 a_2 \dots a_n$  pode ser escrito como

$$\prod_{k=1}^n a_k.$$

Se  $n = 0$ , o valor do produto é definido como 1. Podemos converter uma fórmula com um produto em uma fórmula com um somatório, utilizando a identidade

$$\lg\left(\prod_{k=1}^n a_k\right) = \sum_{k=1}^n \lg a_k.$$

## Exercícios

### A.1-1

Encontre uma fórmula simples para  $\sum_{k=1}^n (2k - 1)$ .

### A.1-2 \*

Mostre que  $\sum_{k=1}^n 1/(2k - 1) = \ln(\sqrt{n}) + O(1)$ , manipulando a série harmônica.

### A.1-3

Mostre que  $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$  para  $0 < |x| < 1$ .

### A.1-4 \*

Mostre que  $\sum_{k=1}^{\infty} (k-1)/2^k = 0$ .

### A.1-5 \*

Avalie a soma  $\sum_{k=1}^{\infty} (2k+1)x^{2k}$ .

### A.1-6

Prove que  $\sum_{k=1}^n O(f_k(n)) = O(\sum_{k=1}^n f_k(n))$  usando a propriedade de linearidade de somatórios.

### A.1-7

Avalie o produto  $\prod_{k=1}^n 2 \cdot 4^k$ .

### A.1-8 \*

Avalie o produto  $\prod_{k=2}^n (1 - 1/k^2)$ .

## A.2 Como limitar somatórios

Existem muitas técnicas disponíveis para limitar os somatórios que descrevem os tempos de execução de algoritmos. Aqui estão alguns dos métodos mais freqüentemente empregados.

## Indução matemática

O caminho mais comum para se definir o valor de uma série é usar a indução matemática. Como exemplo, vamos demonstrar que a série aritmética  $\sum_{k=1}^n k$  tem o valor  $\frac{1}{2}n(n + 1)$ . Isso pode ser verificado facilmente para  $n = 1$ ; assim, criamos a hipótese indutiva de que ela é válida para  $n$  e demonstramos que ela vale para  $n + 1$ . Temos

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^n k + (n + 1). \\ &= \frac{1}{2}n(n + 1) + (n + 1) \\ &= \frac{1}{2}(n + 1)(n + 2).\end{aligned}$$

Não é necessário adivinhar o valor exato de um somatório para usar a indução matemática. A indução também pode ser usada para mostrar um limite. Como exemplo, vamos demonstrar que a série geométrica  $\sum_{k=0}^n 3^k$  é  $O(3^n)$ . Mais especificamente, vamos provar que  $\sum_{k=0}^n 3^k \leq c3^n$  para alguma constante  $c$ . No caso da condição inicial  $n = 0$ , temos  $\sum_{k=0}^0 3^k = 1 \leq c$  enquanto  $c \geq 1$ . Supondo que o limite se mantenha válido para  $n$ , vamos provar que ele é válido para  $n + 1$ . Temos

$$\begin{aligned}\sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^n 3^k + 3^{n+1} \\ &\leq c3^n + c3^{n+1} \\ &= \left(\frac{1}{3} + \frac{1}{c}\right)c3^{n+1} \\ &\leq c3^{n+1}\end{aligned}$$

desde que  $(1/3 + 1/c) \leq 1$  ou, de modo equivalente,  $c \geq 3/2$ . Portanto,  $\sum_{k=0}^n 3^k = O(3^n)$ , como desejávamos demonstrar.

Temos de ser cuidadosos quando usarmos a notação assintótica para provar limites por indução. Considere a seguinte prova falaciosa de que  $\sum_{k=1}^n k = O(n)$ . Certamente,  $\sum_{k=1}^n k = O(1)$ . Partindo da hipótese de que o limite é válido para  $n$ , podemos agora demonstrá-lo para  $n + 1$ :

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^n k + (n + 1) \\ &= O(n) + (n + 1) \quad <= \text{errado!!} \\ &= O(n + 1).\end{aligned}$$

O erro no argumento é que a “constante” oculta pelo “O maiúsculo” cresce com  $n$  e, portanto, não é constante. Não mostramos que a mesma constante funciona para *todo*  $n$ .

## Limitando os termos

Às vezes, um bom limite superior em uma série pode ser obtido limitando-se cada termo da série, e com freqüência é suficiente utilizar o maior termo para limitar os outros. Por exemplo, um limite superior rápido sobre a série aritmética (A.1) é

$$\begin{aligned}\sum_{k=1}^{n+1} k &\leq \sum_{k=1}^n n \\&= n^2.\end{aligned}$$

Em geral, para uma série  $\sum_{k=1}^n a_k$ , se considerarmos  $a_{\max} = \max_{1 \leq k \leq n} a_k$ , então

$$\sum_{k=1}^n a_k \leq n a_{\max}.$$

A técnica de limitar cada termo em uma série pelo maior termo é um método fraco quando a série pode de fato ser limitada por uma série geométrica. Dada a série  $\sum_{k=0}^n a_k$ , suponha que  $a_{k+1}/a_k \leq r$  para todo  $k \geq 0$ , onde  $0 < r < 1$  é uma constante. A soma pode ser limitada por uma série geométrica decrescente infinita, pois  $a_k \leq a_0 r^k$  e, desse modo,

$$\begin{aligned}\sum_{k=0}^n a_k &\leq \sum_{k=0}^{\infty} a_0 r^k \\&= a_0 \sum_{k=0}^{\infty} r^k \\&= a_0 \frac{1}{1-r}.\end{aligned}$$

Podemos aplicar esse método para limitar o somatório  $\sum_{k=1}^{\infty} (k/3^k)$ . Para iniciar o somatório em  $k = 0$ , nós o reescrevemos como  $\sum_{k=0}^{\infty} ((k+1)/3^{k+1})$ . O primeiro termo ( $a_0$ ) é  $1/3$ , e a razão ( $r$ ) entre os termos sucessivos é

$$\begin{aligned}\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} &= \frac{1}{3} \cdot \frac{k+2}{k+1} \\&\leq \frac{2}{3}\end{aligned}$$

para todo  $k \geq 1$ . Desse modo, temos

$$\begin{aligned}\sum_{k=1}^{\infty} \frac{k}{3^k} &= \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}} \\&\leq \frac{1}{3} \cdot \frac{1}{1-2/3} \\&= 1.\end{aligned}$$

Um erro comum na aplicação desse método é mostrar que a razão entre termos sucessivos é menor que 1, e então admitir como hipótese que o somatório é limitado por uma série geométrica. Um exemplo é a série harmônica infinita, que diverge desde

$$\begin{aligned}\sum_{k=1}^{\infty} \frac{1}{k} &= \lim_{n \rightarrow \infty} \sum_{k=1}^n \frac{1}{k} \\&= \lim_{n \rightarrow \infty} \Theta(\lg n)\end{aligned}$$

A razão entre o  $(k + 1)$ -ésimo e o  $k$ -ésimo termos nessa série é  $k/(k + 1) < 1$ , mas a série não é limitada por uma série geométrica decrescente. Para limitar uma série por uma série geométrica, deve-se mostrar que existe um  $r < 1$  que é uma *constante*, tal que a razão entre todos os pares de termos sucessivos nunca excede  $r$ . Na série harmônica, não existe tal  $r$  porque a razão se torna arbitrariamente próxima de 1.

## Divisão de somatórios

Uma das maneiras de obter limites em um somatório difícil é expressar a série como a soma de duas ou mais séries, particionando-se o intervalo do índice e, em seguida, limitando-se cada uma das séries resultantes. Por exemplo, suponha a tentativa de encontrar um limite inferior da série aritmética  $\sum_{k=1}^n k$ , a qual já mostramos que tem um limite superior  $n^2$ . Poderíamos tentar limitar cada termo no somatório pelo menor termo mas, como esse termo é 1, obtemos um limite inferior  $n$  para o somatório – bem longe do nosso limite superior  $n^2$ .

Podemos obter um limite inferior melhor dividindo primeiro o somatório. Por conveniência, suponha que  $n$  seja par. Temos

$$\sum_{k=1}^n k = \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^n k$$

$$\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^n (n/2)$$

$$= (n/2)^2$$

$$= \Omega(n^2),$$

que é um limite assintoticamente restrito, considerando-se que  $\sum_{k=1}^n k = O(n^2)$ .

Para um somatório que surge da análise de um algoritmo, podemos freqüentemente dividir o somatório e ignorar um número constante dos termos iniciais. Em geral, essa técnica se aplica quando cada termo  $a_k$  em um somatório  $\sum_{k=0}^n a_k$  é independente de  $n$ . Então, para qualquer constante  $k_0 > 0$ , podemos escrever

$$\sum_{k=0}^n a_k = \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^n a_k$$

$$= \Theta(1) + \sum_{k=k_0}^n a_k,$$

pois os termos iniciais do somatório são todos constantes e existe um número constante deles. Podemos então usar outros métodos para limitar  $\sum_{k=k_0}^n a_k$ . Por exemplo, encontrar um limite superior assintótico sobre

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k},$$

observamos que a razão entre termos sucessivos é

$$\begin{aligned} \frac{(k+1)^2/2^{k+1}}{k^2/2^k} &= \frac{(k+2)^2}{2k^2} \\ &\leq \frac{8}{9} \end{aligned}$$

se  $k \geq 3$ . Portanto, o somatório também pode ser dividido em

$$\begin{aligned}
\sum_{k=0}^{\infty} \frac{k^2}{2^k} &= \sum_{k=0}^2 \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \\
&\leq \sum_{k=0}^2 \frac{k^2}{2^k} + \frac{8}{9} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k \\
&= O(1),
\end{aligned}$$

pois o segundo somatório é uma série geométrica decrescente.

A técnica de dividir somatórios pode ser usada para definir limites assintóticos em situações muito mais difíceis. Por exemplo, podemos obter um limite  $O(\lg n)$  na série harmônica (A.7):

$$H_n = \sum_{k=1}^n \frac{1}{k}.$$

A idéia é dividir o intervalo de 1 a  $n$  em  $\lfloor \lg n \rfloor$  fragmentos e estabelecer um limite superior como contribuição de cada fragmento em 1. Cada fragmento consiste nos termos que começam em  $1/2^i$  e que vão até  $1/2^{i+1}$ , sem incluí-lo, fornecendo

$$\begin{aligned}
\sum_{k=1}^n \frac{1}{k} &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \\
&\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} \\
&\leq \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \\
&\leq \lg n + 1.
\end{aligned} \tag{A.10}$$

## Aproximação por integrais

Quando um somatório pode ser expresso como  $\sum_{k=m}^n f(k)$ , onde  $f(k)$  é uma função monotonicamente crescente, é possível fazer a aproximação do somatório por integrais:

$$\int_{m-1}^n f(x)dx \leq \sum_{k=m}^n f(k) \leq \int_m^{n+1} f(x)dx. \tag{A.11}$$

A justificativa para essa aproximação é mostrada na Figura A.1. O somatório é representado como a área dos retângulos na figura, e a integral é a região sombreada sob a curva. Quando  $f(k)$  é uma função monotonicamente decrescente, podemos usar um método semelhante para fornecer os limites

$$\int_{m-1}^{n+1} f(x)dx \leq \sum_{k=m}^n f(k) \leq \int_{m-1}^n f(x)dx. \tag{A.12}$$

A aproximação integral (A.12) fornece uma estimativa restrita para o  $n$ -ésimo número harmônico. No caso de um limite inferior, obtemos

$$\begin{aligned}
\sum_{k=1}^n \frac{1}{k} &\geq \int_1^{n+1} \frac{dx}{x} \\
&= \ln(n+1).
\end{aligned} \tag{A.13}$$

Para o limite superior, derivamos a desigualdade

$$\sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{dx}{x}$$

$$= \ln n ,$$

que produz o limite

$$\sum_{k=1}^n \frac{1}{k} \leq \ln n + 1 . \quad (\text{A.14})$$

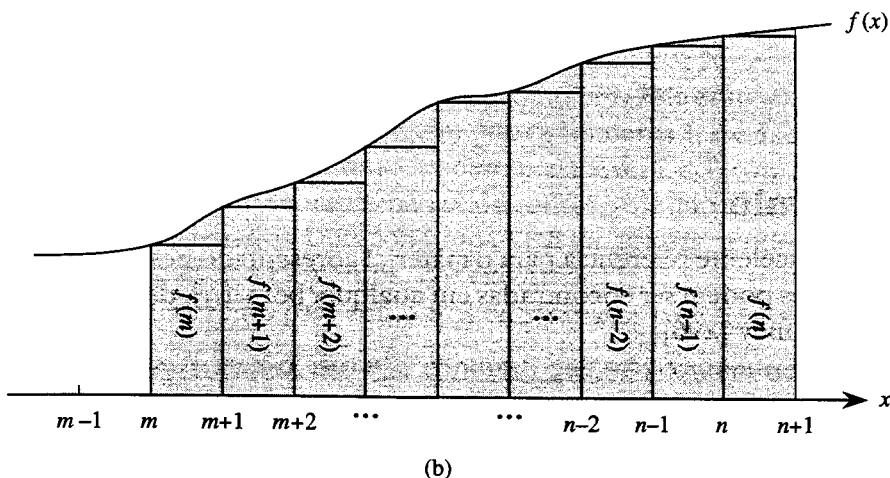
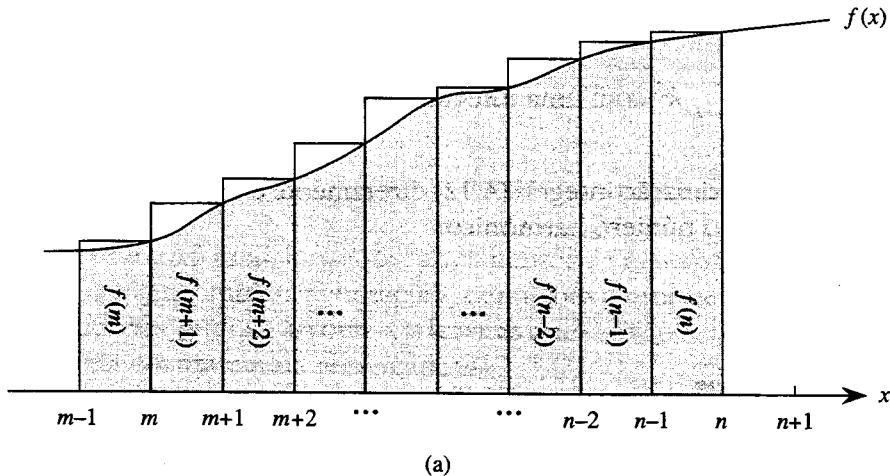


FIGURA A.1 Aproximação de  $\sum_{k=m}^n f(x)$  por integrais. A área de cada retângulo é mostrada dentro do retângulo, e a área total dos retângulos representa o valor do somatório. A integral é representada pela área sombreada sob a curva. Comparando as áreas em (a), obtemos  $\int_{m-1}^n f(x) dx \leq \sum_{k=m}^n f(k)$ , e depois, deslocando os retângulos uma unidade para a direita, obtemos  $\sum_{k=m}^n f(k) \leq \int_m^{n+1} f(x) dx$  em (b)

## Exercícios

### A.2-1

Mostre que  $\sum_{k=1}^n 1/k^2$  é limitada acima por uma constante.

### A.2-2

Encontre um limite superior assintótico sobre o somatório

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil$$

### A.2-3

Mostre que o  $n$ -ésimo número harmônico é  $\Omega(\lg n)$ , dividindo o somatório.

### A.2-4

Faça a aproximação de  $\sum_{k=1}^n k^3$  com uma integral.

### A.2-5

Por que não usamos a aproximação integral (A.12) diretamente em  $\sum_{k=1}^n 1/k$  para obter um limite superior sobre o  $n$ -ésimo número harmônico?

## Problemas

### A-1 Limitando somatórios

Forneça limites assintoticamente restritos sobre os somatórios a seguir. Suponha que  $r \geq 0$  e  $s \geq 0$  sejam constantes.

- $\sum_{k=1}^n k^r$ .
- $\sum_{k=1}^n \lg^s k$ .
- $\sum_{k=1}^n k^r \lg^s k$ .

## Notas do capítulo

Knuth [182] é uma excelente referência para o material apresentado neste capítulo. As propriedades básicas de séries podem ser encontradas em qualquer bom livro de cálculo, como Apostol [18] ou Thomas e Finney [296].

---

## *Apêndice B*

# *Conjuntos e outros temas*

Muitos capítulos deste livro mencionam os fundamentos de matemática discreta. Este capítulo reexamina de forma mais completa as notações, definições e propriedades elementares de conjuntos, relações, funções, grafos e árvores. Os leitores que já estão bem versados nesses assuntos só precisam dar uma olhada rápida neste capítulo.

### **B.1 Conjuntos**

Um conjunto é uma coleção de objetos distintos, que são chamados *elementos* ou *membros* do conjunto. Se um objeto  $x$  é um elemento de (ou pertence a) um conjunto  $S$ , escrevemos  $x \in S$  (lê-se “ $x$  é um membro de  $S$ ”, “ $x$  é elemento de  $S$ ”, “ $x$  pertence a  $S$ ” ou, de modo mais abreviado, “ $x$  está em  $S$ ”). Se  $x$  não é um elemento de  $S$ , escrevemos que  $x \notin S$ . Podemos descrever um conjunto relacionando explicitamente seus elementos como uma lista entre chaves. Por exemplo, é possível definir um conjunto  $S$  contendo exatamente os números 1, 2 e 3, escrevendo-se  $S = \{1, 2, 3\}$ . Como 2 é um elemento do conjunto  $S$ , podemos escrever  $2 \in S$ ; como 4 não é um elemento do conjunto, temos  $4 \notin S$ . Um conjunto não pode conter o mesmo objeto mais de uma vez,<sup>1</sup> e seus elementos não são ordenados. Dois conjuntos  $A$  e  $B$  são *iguais*, sendo representados por  $A = B$ , se eles contêm os mesmos elementos. Por exemplo,  $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$ .

Adotamos notações especiais para conjuntos encontrados com freqüência.

- $\emptyset$  denota o *conjunto vazio*, isto é, o conjunto que não contém nenhum elemento.
- $\mathbb{Z}$  denota o conjunto de *números inteiros*, isto é, o conjunto  $\{\dots, -2, -1, 0, 1, 2, \dots\}$ .
- $\mathbb{R}$  denota o conjunto de *números reais*.
- $\mathbb{N}$  denota o conjunto de *números naturais*, isto é, o conjunto  $\{0, 1, 2, \dots\}$ .<sup>2</sup>

Se todos os elementos de um conjunto  $A$  estão contidos em um conjunto  $B$ , ou seja, se  $x \in A$  implica  $x \in B$ , escrevemos  $A \subseteq B$  e dizemos que  $A$  é um *subconjunto* de  $B$ . Um conjunto  $A$  é um *subconjunto próprio* de  $B$ , representado por  $A \subset B$ , se  $A \subseteq B$  mas  $A \neq B$ . (Alguns autores usam o símbolo “ $\subsetneq$ ” para denotar a relação de subconjunto comum, em lugar da relação de subconjunto

---

<sup>1</sup>Uma variação de um conjunto, que pode conter o mesmo objeto mais de uma vez, é chamada um *multiconjunto*.

<sup>2</sup>Alguns autores iniciam os números naturais com 1 em vez de 0. A tendência moderna parece ser a de iniciar esse conjunto com 0.

próprio.) Para qualquer conjunto  $A$ , temos  $A \subseteq A$ . No caso de dois conjuntos  $A$  e  $B$ , temos  $A = B$  se e somente se  $A \subseteq B$  e  $B \subseteq A$ . Para três conjuntos  $A$ ,  $B$  e  $C$  quaisquer, se  $A \subseteq B$  e  $B \subseteq C$ , então  $A \subseteq C$ . Para qualquer conjunto  $A$ , temos  $\emptyset \subseteq A$ .

Algumas vezes, definimos conjuntos em termos de outros conjuntos. Dado um conjunto  $A$ , podemos definir um conjunto  $B \subseteq A$  declarando uma propriedade que distingue os elementos de  $B$ . Por exemplo, podemos definir o conjunto de números inteiros pares por  $\{x : x \in \mathbb{Z} \text{ e } x/2 \text{ é um inteiro}\}$ . Nessa notação, o sinal de dois-pontos significa “tal que”. (Alguns autores usam uma barra vertical em lugar do sinal de dois-pontos.)

Dados dois conjuntos  $A$  e  $B$ , também podemos definir novos conjuntos aplicando **operações de conjuntos**:

- A **interseção** de conjuntos  $A$  e  $B$  é o conjunto

$$A \cap B = \{x : x \in A \text{ e } x \in B\}.$$

- A **união** de conjuntos  $A$  e  $B$  é o conjunto

$$A \cup B = \{x : x \in A \text{ ou } x \in B\}.$$

- A **diferença** entre dois conjuntos  $A$  e  $B$  é o conjunto

$$A - B = \{x : x \in A \text{ e } x \notin B\}.$$

As operações de conjuntos obedecem às leis enunciadas a seguir.

**Leis de conjuntos vazios:**

$$A \cap \emptyset = \emptyset,$$

$$A \cup \emptyset = A.$$

**Leis de idempotência:**

$$A \cap A = A,$$

$$A \cup A = A.$$

**Leis comutativas:**

$$A \cap B = B \cap A,$$

$$A \cup B = B \cup A.$$

**Leis associativas:**

$$A \cap (B \cap C) = (A \cap B) \cap C,$$

$$A \cup (B \cup C) = (A \cup B) \cup C.$$

**Leis distributivas:**

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C), \tag{B.1}$$

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$$

### Leis de absorção:

$$A \cap (A \cup B) = A ,$$

$$A \cup (A \cap B) = A .$$

### Leis de DeMorgan:

$$A - (B \cap C) = (A - B) \cup (A - C) ,$$

$$A - (B \cup C) = (A - B) \cap (A - C) . \quad (\text{B.2})$$

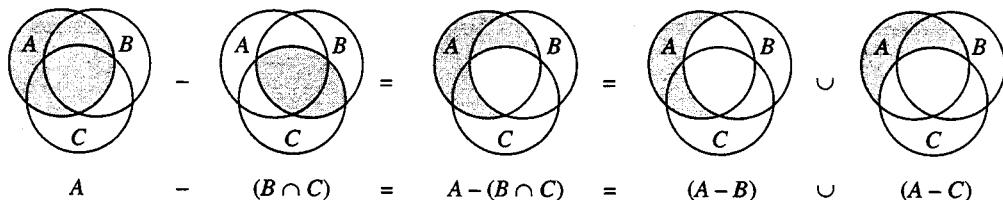


FIGURA B.1 Um diagrama de Venn que ilustra as primeiras leis de DeMorgan (B.2). Cada um dos conjuntos A, B e C é representado como um círculo

A primeira das leis de DeMorgan está ilustrada na Figura B.1 com o uso de um *diagrama de Venn*, uma imagem gráfica na qual os conjuntos são representados como regiões do plano.

Muitas vezes, todos os conjuntos sob consideração são subconjuntos de algum conjunto maior  $U$  chamado *universo*. Por exemplo, se estivermos considerando vários conjuntos formados apenas por inteiros, o conjunto  $\mathbb{Z}$  de inteiros será um universo apropriado. Dado um universo  $U$ , definimos o *complemento* de um conjunto  $A$  como  $\bar{A} = U - A$ . Para qualquer conjunto  $A \subseteq U$ , temos as seguintes leis:

$$\bar{\bar{A}} = A ,$$

$$A \cap \bar{A} = \emptyset ,$$

$$A \cup \bar{A} = U .$$

As leis de DeMorgan (B.2) podem ser reescritas com complementos. Para dois conjuntos quaisquer  $A, B \subseteq U$ , temos

$$\overline{B \cap C} = \bar{B} \cup \bar{C} ,$$

$$\overline{B \cup C} = \bar{B} \cap \bar{C}$$

Dois conjuntos  $A$  e  $B$  são *disjuntos* se não têm nenhum elemento em comum, ou seja, se  $A \cap B = \emptyset$ . Uma coleção  $\mathcal{J} = \{S_i\}$  de conjuntos não vazios forma uma *partição* de um conjunto  $S$  se:

- os conjuntos são *disjuntos aos pares*, isto é,  $S_i, S_j \in \mathcal{J}$  e  $i \neq j$  implicam  $S_i \cap S_j = \emptyset$ , e
- sua união é  $S$ , isto é,

$$S = \bigcup_{S_i \in \mathcal{J}} S_i .$$

Em outras palavras,  $\mathcal{J}$  forma uma partição de  $S$  se cada elemento de  $S$  aparece em exatamente um  $S_i \in \mathcal{J}$ .

O número de elementos em um conjunto é chamado **cardinalidade** (ou *tamanho*) do conjunto, denotada por  $|S|$ . Dois conjuntos têm a mesma cardinalidade se seus elementos podem ser colocados em uma correspondência de um para um. A cardinalidade do conjunto vazio é  $|\emptyset| = 0$ . Se a cardinalidade de um conjunto é um número natural, dizemos que o conjunto é **finito**; caso contrário, ele é **infinito**. Um conjunto infinito que pode ser colocado em uma correspondência de um para um com os números naturais  $\mathbb{N}$  é **infinito contável**; caso contrário, ele é **não contável**. Os inteiros  $\mathbb{Z}$  são contáveis, mas os reais  $\mathbb{R}$  são não contáveis.

Para dois conjuntos finitos  $A$  e  $B$ , temos a identidade

$$|A \cup B| = |A| + |B| - |A \cap B| , \quad (\text{B.3})$$

da qual podemos concluir que

$$|A \cup B| \leq |A| + |B| .$$

Se  $A$  e  $B$  são disjuntos, então  $|A \cap B| = 0$  e, portanto,  $|A \cup B| = |A| + |B|$ . Se  $A \subseteq B$ , então  $|A| \leq |B|$ .

Um conjunto finito formado por  $n$  elementos às vezes é chamado um **conjunto de  $n$  elementos**. Um conjunto de um elemento é chamado **unitário**. Um subconjunto de  $k$  elementos de um conjunto às vezes é chamado um **subconjunto de  $k$  elementos**.

O conjunto de todos os subconjuntos de um conjunto  $S$ , inclusive o conjunto vazio e o próprio conjunto  $S$ , é denotado por  $2^S$  e é chamado **conjunto potência** de  $S$ . Por exemplo,  $2^{\{a, b\}} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ . O conjunto potência de um conjunto finito  $S$  tem cardinalidade  $2^{|S|}$ .

Algumas vezes, utilizamos estruturas semelhantes a conjuntos, nas quais os elementos estão ordenados. Um **par ordenado** de dois elementos  $a$  e  $b$  é denotado por  $(a, b)$  e pode ser definido formalmente como o conjunto  $(a, b) = \{a, \{a, b\}\}$ . Desse modo, o par ordenado  $(a, b)$  não é igual ao par ordenado  $(b, a)$ .

O **produto cartesiano** de dois conjuntos  $A$  e  $B$ , denotado por  $A \times B$ , é o conjunto de todos os pares ordenados tais que o primeiro elemento do par é um elemento de  $A$  e o segundo é um elemento de  $B$ . De modo mais formal,

$$A \times B = \{(a, b) : a \in A \text{ e } b \in B\} .$$

Por exemplo,  $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$ . Quando  $A$  e  $B$  são conjuntos finitos, a cardinalidade de seu produto cartesiano é

$$|A \times B| = |A| \cdot |B| . \quad (\text{B.4})$$

O produto cartesiano de  $n$  conjuntos  $A_1, A_2, \dots, A_n$  é o conjunto de  **$n$ -tuplas**

$$A_1 \times A_2 \times \dots \times A_n = (a_1, a_2, \dots, a_n) : a_i \in A_i, i = 1, 2, \dots, n ,$$

cuja cardinalidade é

$$|A_1 \times A_2 \times \dots \times A_n| = |A_1| \cdot |A_2| \cdots |A_n|$$

se todos conjuntos são finitos. Denotamos um produto cartesiano de  $n$  termos sobre um único conjunto  $A$  pelo conjunto

$$A^n = A \times A \times \dots \times A ,$$

cuja cardinalidade é  $|A^n| = |A|^n$  se  $A$  é finito. Uma  $n$ -tupla também pode ser visualizada como uma seqüência finita de comprimento  $n$  (ver Seção B.3).

## Exercícios

### B.1-1

Trace diagramas de Venn que ilustrem a primeira das leis distributivas (B.1).

### B.1-2

Prove a generalização das leis de DeMorgan para qualquer coleção finita de conjuntos:

$$\overline{A_1 \cap A_2 \cap \dots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \dots \cup \overline{A_n},$$

$$\overline{A_1 \cup A_2 \cup \dots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \dots \cap \overline{A_n}.$$

### B.1-3 \*

Prove a generalização da equação (B.3), que é chamada **princípio de inclusão e exclusão**:

$$|A_1 \cup A_2 \cup \dots \cup A_n| =$$

$$|A_1| + |A_2| + \dots + |A_n|$$

$$- |A_1 \cap A_2| + \dots + |A_1 \cap A_3| - \dots \quad (\text{todos os pares})$$

$$+ |A_1 \cap A_2 \cap A_3| + \dots \quad (\text{todas as triplas})$$

⋮

$$+ (-1)^{n-1} |A_1 \cap A_2 \cap A_3| .$$

### B.1-4

Mostre que o conjunto de números naturais ímpares é contável.

### B.1-5

Mostre que, para qualquer conjunto finito  $S$ , o conjunto potência  $2^S$  tem  $2^{|S|}$  (ou seja, existem  $2^{|S|}$  subconjuntos distintos de  $S$ ).

### B.1-6

Dê uma definição indutiva para uma  $n$ -tupla, estendendo a definição da teoria dos conjuntos para um par ordenado.

## B.2 Relações

Uma **relação binária**  $R$  sobre dois conjuntos  $A$  e  $B$  é um subconjunto do produto cartesiano  $A \times B$ . Se  $(a, b) \in R$ , algumas vezes escrevemos  $a R b$ . Quando dizemos que  $R$  é uma relação binária sobre um conjunto  $A$ , queremos dizer que  $R$  é um subconjunto de  $A \times A$ . Por exemplo, a relação “menor que” sobre os números naturais é o conjunto  $\{(a, b) : a, b \in \mathbb{N} \text{ e } a < b\}$ . Uma relação  $n$ -ária sobre conjuntos  $A_1, A_2, \dots, A_n$  é um subconjunto de  $A_1 \times A_2 \times \dots \times A_n$ .

Uma relação binária  $R \subseteq A \times A$  é **reflexiva** se

$$a R a$$

para todo  $a \in A$ . Por exemplo, “=” e “ $\leq$ ” são relações reflexivas em  $\mathbb{N}$ , mas “ $<$ ” não é. A relação  $R$  é **simétrica** se

$$a R b \text{ implica } b R a$$

para todo  $a, b \in A$ . Por exemplo, “=” é simétrica, mas “ $<$ ” e “ $\leq$ ” não são. A relação  $R$  é **transitiva** se

$$a R b \text{ e } b R c \text{ implicam } a R c$$

para todo  $a, b, c \in A$ . Por exemplo, as relações “ $<$ ”, “ $\leq$ ” e “ $=$ ” são transitivas, mas a relação  $R = \{(a, b) : a, b \in N \text{ e } a = b - 1\}$  não é, pois  $3 R 4$  e  $4 R 5$  não implicam  $3 R 5$ .

Uma relação que é reflexiva, simétrica e transitiva é uma **relação de equivalência**. Por exemplo, “ $=$ ” é uma relação de equivalência sobre os números naturais, mas “ $<$ ” não é. Se  $R$  é uma relação de equivalência em um conjunto  $A$ , então, para  $a \in A$ , a **classe de equivalência** de  $a$  é o conjunto  $[a] = \{b \in A : a R b\}$ , ou seja, o conjunto de todos os elementos equivalentes a  $a$ . Por exemplo, se definimos  $R = \{(a, b) : a, b \in N \text{ e } a + b \text{ é um número par}\}$ , então  $R$  é uma relação de equivalência, pois  $a + a$  é par (reflexiva),  $a + b$  é par implica  $b + a$  é par (simétrica) e  $a + b$  é par e  $b + c$  é par implicam  $a + c$  é par (transitiva). A classe de equivalência de 4 é  $[4] = \{0, 2, 4, 6, \dots\}$ , e a classe de equivalência de 3 é  $[3] = \{1, 3, 5, 7, \dots\}$ . Um teorema básico de classes de equivalência é dado a seguir.

### **Teorema B.1 (Uma relação de equivalência é o mesmo que uma partição)**

As classes de equivalência de qualquer relação de equivalência  $R$  sobre um conjunto  $A$  formam uma partição de  $A$ , e qualquer partição de  $A$  determina uma relação de equivalência sobre  $A$  para a qual os conjuntos na partição são as classes de equivalência.

**Prova** Como primeira parte da prova, devemos mostrar que as classes de equivalência de  $R$  são conjuntos não vazios e disjuntos aos pares cuja união é  $A$ . Como  $R$  é reflexiva,  $a \in [a]$ , e assim as classes de equivalência são não vazias; além disso, tendo em vista que todo elemento  $a \in A$  pertence à classe de equivalência  $[a]$ , a união das classes de equivalência é  $A$ . Resta mostrar que as classes de equivalência são conjuntos disjuntos aos pares, isto é, se duas classes de equivalência  $[a]$  e  $[b]$  têm um elemento  $c$  em comum, então elas são de fato o mesmo conjunto. Agora,  $a R c$  e  $b R c$  que, por simetria e transitividade, implicam  $a R b$ . Portanto, para qualquer elemento arbitrário  $x \in [a]$ , temos que  $x R a$  implica  $x R b$  e, desse modo,  $[a] \subseteq [b]$ . De modo semelhante,  $[b] \subseteq [a]$  e, assim sendo,  $[a] = [b]$ .

Como segunda parte da prova, seja  $\mathcal{A} = \{A_i\}$  uma partição de  $A$ , e defina  $R = \{(a, b) : \text{existe } i \text{ tal que } a \in A_i \text{ e } b \in A_i\}$ . Afirmamos que  $R$  é uma relação de equivalência em  $A$ . A reflexividade se mantém, pois  $a \in A_i$  implica  $a R a$ . A simetria se mantém porque, se  $a R b$ , então  $a$  e  $b$  estão no mesmo conjunto  $A_i$  e, consequentemente,  $b R a$ . Se  $a R b$  e  $b R c$ , então todos os três elementos estão no mesmo conjunto e, desse modo,  $a R c$ , e a transitividade é mantida. Para verificar que os conjuntos na partição são as classes de equivalência de  $R$ , observe que, se  $a \in A_i$ , então  $x \in [a]$  implica  $x \in A_i$ , e  $x \in A_i$  implica  $x \in [a]$ . ■

Uma relação binária  $R$  sobre um conjunto  $A$  é **anti-simétrica** se  
 $a R b$  e  $b R a$  implicam  $a = b$ .

Por exemplo, a relação “ $<$ ” sobre os números naturais é anti-simétrica, pois  $a \leq b$  e  $b \leq a$  implicam  $a = b$ . Uma relação que é reflexiva, anti-simétrica e transitiva é uma **ordem parcial**, e chamamos um conjunto no qual uma ordem parcial é definida de **conjunto parcialmente ordenado**. Por exemplo, a relação “é descendente de” é uma ordem parcial no conjunto de todas as pessoas (se visualizarmos indivíduos como sendo seus próprios descendentes).

Em um conjunto parcialmente ordenado  $A$ , não pode haver nenhum elemento “máximo”  $a$  tal que  $b R a$  para todo  $b \in A$ . Em vez disso, pode haver diversos elementos **máximos**  $a$  tais que, para nenhum  $b \in A$ , onde  $b \neq a$ , ocorre que  $a R b$ . Por exemplo, em uma coleção de caixas de diferentes tamanhos podem existir várias caixas máximas que não se ajustam dentro de qualquer outra caixa, ainda que não exista nenhuma caixa máxima única dentro da qual se ajustará qualquer outra caixa.<sup>3</sup>

---

<sup>3</sup> Para sermos exatos, com a finalidade de fazer a relação “caber em” ser uma ordem parcial, precisamos visualizar uma caixa como cabendo dentro dela própria.

Uma ordem parcial  $R$  sobre um conjunto  $A$  é uma **ordem total** ou **ordem linear** se, para todo  $a, b \in A$ , temos  $a R b$  ou  $b R a$ , ou seja, se for possível relacionar entre si todos os pares de elementos de  $A$  em  $R$ . Por exemplo, a relação “ $\leq$ ” é uma ordem total sobre os números naturais, mas a relação “é descendente de” não é uma ordem total sobre o conjunto de todas as pessoas, pois existem indivíduos que não são descendentes uns dos outros.

## Exercícios

### B.2-1

Prove que a relação de subconjunto “ $\subseteq$ ” em todos os subconjuntos de  $\mathbb{Z}$  é uma ordem parcial mas não uma ordem total.

### B.2-2

Mostre que, para qualquer inteiro positivo  $n$ , a relação “equivalente de módulo  $n$ ” é uma relação de equivalência sobre os inteiros. (Dizemos que  $a \equiv b \pmod{n}$  se existe um inteiro  $q$  tal que  $a - b = qn$ .) Em que classes de equivalência essa relação particiona os inteiros?

### B.2-3

Dê exemplos de relações que sejam

- a.* reflexivas e simétricas, mas não transitivas,
- b.* reflexivas e transitivas, mas não simétricas,
- c.* simétricas e transitivas, mas não reflexivas.

### B.2-4

Seja  $S$  um conjunto finito, e seja  $R$  uma relação de equivalência sobre  $S \times S$ . Mostre que, se uma adição  $R$  é anti-simétrica, então as classes de equivalência de  $S$  com respeito a  $R$  são unitárias.

### B.2-5

O professor Narciso afirma que, se uma relação  $R$  é simétrica e transitiva, então ela também é reflexiva. Ele oferece a seguinte prova. Por simetria,  $a R b$  implica  $b R a$ . Por essa razão, a transitividade implica  $a R a$ . O professor está certo?

## B.3 Funções

Dados dois conjuntos  $A$  e  $B$ , uma **função**  $f$  é uma relação binária em  $A \times B$  tal que, para  $a \in A$ , existe exatamente um  $b \in B$  tal que  $(a, b) \in f$ . O conjunto  $A$  é chamado **domínio** de  $f$  e o conjunto  $B$  é chamado **contradomínio** de  $f$ . Algumas vezes, escrevemos  $f: A \rightarrow B$ ; e, se  $(a, b) \in f$ , escrevemos  $b = f(a)$ , pois  $b$  é determinado de forma unívoca pela escolha de  $a$ .

Intuitivamente, a função  $f$  atribui um elemento de  $B$  a cada elemento de  $A$ . Nenhum elemento de  $A$  recebe a atribuição de dois elementos diferentes de  $B$ , mas o mesmo elemento de  $B$  pode receber a atribuição de dois elementos diferentes de  $A$ . Por exemplo, a relação binária

$$f = \{(a, b) : a, b \in \mathbb{N} \text{ e } b = a \bmod 2\}$$

é uma função  $f: \mathbb{N} \rightarrow \{0, 1\}$  porque, para cada número natural  $a$ , existe exatamente um valor  $b$  em  $\{0, 1\}$  tal que  $b = a \bmod 2$ . Para esse exemplo,  $0 = f(0)$ ,  $1 = f(1)$ ,  $0 = f(2)$  etc. Em contraste, a relação binária

$$g = \{(a, b) : a, b \in \mathbb{N} \text{ e } a + b \text{ é par}\}$$

não é uma função, pois  $(1, 3)$  e  $(1, 5)$  estão ambos em  $g$  e, desse modo, para a opção  $a = 1$ , não existe exatamente um  $b$  tal que  $(a, b) \in g$ .

Dada uma função  $f: A \rightarrow B$ , se  $b = f(a)$ , dizemos que  $a$  é o **argumento** de  $f$  e que  $b$  é o **valor** de  $f$  em  $a$ . Podemos definir uma função declarando seu valor para cada elemento de seu domínio. Por exemplo, poderíamos definir  $f(n) = 2n$  para  $n \in \mathbb{N}$ , o que significa  $f = \{(n, 2n) : n \in \mathbb{N}\}$ . Duas funções  $f$  e  $g$  são **iguais** se elas têm o mesmo domínio e contradomínio e se, para todo  $a$  no domínio,  $f(a) = g(a)$ .

Uma **seqüência finita** de comprimento  $n$  é uma função  $f$  cujo domínio é o conjunto de  $n$  inteiros  $\{0, 1, \dots, n - 1\}$ . Com freqüência, denotamos uma seqüência finita listando seus valores:  $\langle f(0), f(1), \dots, f(n - 1) \rangle$ . Uma **seqüência infinita** é uma função cujo domínio é o conjunto  $\mathbb{N}$  dos números naturais. Por exemplo, a seqüência de Fibonacci definida pela recorrência (3.21), é a seqüência infinita  $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \dots \rangle$ .

Quando o domínio de uma função  $f$  é um produto cartesiano, freqüentemente omitimos os parênteses extras que envolvem o argumento de  $f$ . Por exemplo, se tivéssemos uma função  $f: A_1 \times A_2 \times \dots \times A_n \rightarrow B$ , escreveríamos  $b = f(a_1, a_2, \dots, a_n)$  em vez de  $b = f((a_1, a_2, \dots, a_n))$ . Também chamamos cada  $a_i$ , um **argumento** para a função  $f$ , embora tecnicamente o (único) argumento para  $f$  seja a  $n$ -tupla  $(a_1, a_2, \dots, a_n)$ .

Se  $f: A \rightarrow B$  é uma função e  $b = f(a)$ , então dizemos às vezes que  $b$  é a **imagem** de  $a$  sob  $f$ . A imagem de um conjunto  $A' \subseteq A$  sob  $f$  é definida por

$$f(A') = \{b \in B : b = f(a) \text{ para algum } a \in A'\}.$$

O **intervalo** de  $f$  é a imagem do seu domínio, ou seja,  $f(A)$ . Por exemplo, o intervalo da função  $f: \mathbb{N} \rightarrow \mathbb{N}$  definida por  $f(n) = 2n$  é  $f(\mathbb{N}) = \{m : m = 2n \text{ para algum } n \in \mathbb{N}\}$ .

Uma função é uma **sobrejeção** se seu intervalo é seu contradomínio. Por exemplo, a função  $f(n) = \lfloor n/2 \rfloor$  é uma função sobrejetora de  $\mathbb{N}$  para  $\mathbb{N}$ , pois todo elemento em  $\mathbb{N}$  aparece como o valor de  $f$  para algum argumento. Em contraste, a função  $f(n) = 2n$  não é uma função sobrejetora de  $\mathbb{N}$  para  $\mathbb{N}$ , porque nenhum argumento de  $f$  pode produzir 3 como um valor. Porém, a função  $f(n) = 2n$  é uma função sobrejetora dos números naturais para os números pares. Uma sobrejeção  $f: A \rightarrow B$  é descrita algumas vezes como o mapeamento de  $A$  sobre  $B$ . Quando dizemos que  $f$  está sobre, queremos dizer que ela é sobrejetora.

Uma função  $f: A \rightarrow B$  é uma **injeção** se argumentos distintos de  $f$  produzem valores distintos, isto é, se  $a \neq a'$  implica  $f(a) \neq f(a')$ . Por exemplo, a função  $f(n) = 2n$  é um função injetora de  $\mathbb{N}$  para  $\mathbb{N}$ , pois cada número par  $b$  é a imagem sob  $f$  de no máximo um elemento do domínio, ou seja,  $b/2$ . A função  $f(n) = \lfloor n/2 \rfloor$  não é injetora, pois o valor 1 é produzido por dois argumentos: 2 e 3. Algumas vezes, uma injeção é chamada uma função de **um para um**.

Uma função  $f: A \rightarrow B$  é um **bijeção** se ela é injetora e sobrejetora. Por exemplo, a função  $f(n) = (-1)^n \lceil n/2 \rceil$  é uma bijeção de  $\mathbb{N}$  para  $\mathbb{Z}$ :

$$0 \rightarrow 0,$$

$$1 \rightarrow -1,$$

$$2 \rightarrow 1,$$

$$3 \rightarrow -2,$$

$$4 \rightarrow 2,$$

⋮

A função é injetora, porque nenhum elemento de  $\mathbb{Z}$  é a imagem de mais de um elemento de  $\mathbb{N}$ . Ela é sobrejetora, pois todo elemento de  $\mathbb{Z}$  aparece como imagem de algum elemento de  $\mathbb{N}$ . Então, a função é bijetora. Às vezes, uma bijeção é chamada uma **correspondência de um para um**, porque emparelha elementos do domínio e do contradomínio. Uma bijeção de um conjunto  $A$  para ele mesmo é chamada às vezes **permutação**.

Quando uma função  $f$  é bijetora, sua *inversa*  $f^{-1}$  é definida como

$$f^{-1}(b) = a \text{ se e somente se } f(a) = b.$$

Por exemplo, a inversa da função  $f(n) = (-1)^n \lceil n/2 \rceil$  é

$$f^{-1}(m) = \begin{cases} 2m & \text{se } m \geq 0, \\ -2m & \text{se } m < 0. \end{cases}$$

## Exercícios

### B.3-1

Sejam  $A$  e  $B$  conjuntos finitos, e seja  $f: A \rightarrow B$  uma função. Mostre que

- se  $f$  é injetora, então  $|A| \leq |B|$ ;
- se  $f$  é sobrejetora, então  $|A| \geq |B|$ .

### B.3-2

A função  $f(x) = x + 1$  é bijetora quando o domínio e o contradomínio são  $\mathbb{N}$ ? Ela é bijetora quando o domínio e o contradomínio são  $\mathbb{Z}$ ?

### B.3-3

Dê uma definição natural para a inversa de uma relação binária tal que, se uma relação é de fato uma função bijetora, sua inversa relacional é sua inversa funcional.

### B.3-4 \*

Dê uma bijeção de  $\mathbb{Z}$  para  $\mathbb{Z} \times \mathbb{Z}$ .

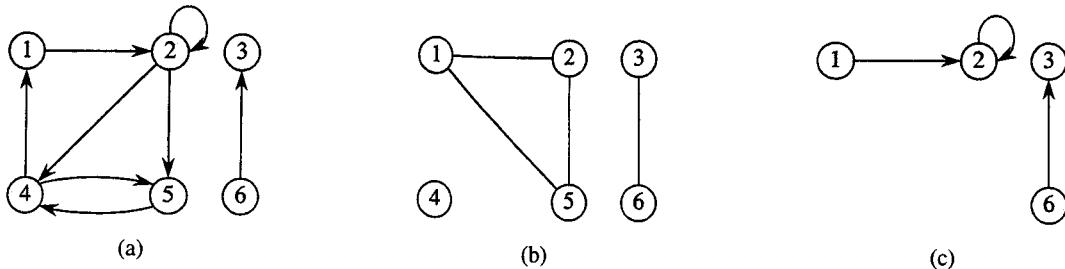
## B.4 Grafos

Esta seção apresenta dois tipos de grafos: orientado e não orientado. O leitor deve saber que certas definições na literatura diferem das que são dadas aqui mas, em sua maioria, as diferenças são ligeiras. A Seção 22.1 mostra como os grafos podem ser representados na memória do computador.

Um *grafo orientado*  $G$  é um par  $(V, E)$ , onde  $V$  é um conjunto finito e  $E$  é uma relação binária em  $V$ . O conjunto  $V$  é chamado *conjunto de vértices* de  $G$ , e seus elementos são chamados *vértices*. O conjunto  $E$  é chamado *conjunto de arestas* de  $G$ , e seus elementos são chamados *arestas*. A Figura B.2(a) é uma representação pictórica de um grafo orientado sobre o conjunto de vértices  $\{1, 2, 3, 4, 5, 6\}$ . Os vértices são representados por círculos na figura, e as arestas são representadas por setas. Observe que são possíveis *autoloops* – arestas de um vértice para ele próprio.

Em um *grafo não orientado*  $G = (V, E)$ , o conjunto de arestas  $E$  consiste em pares de vértices *não ordenados*, em lugar de pares ordenados. Isto é, uma aresta é um conjunto  $\{u, v\}$ , onde  $u, v \in V$  e  $u \neq v$ . Por convenção, usamos a notação  $(u, v)$  para uma aresta, em lugar da notação de conjuntos  $\{u, v\}$ , e  $(u, v)$  e  $(v, u)$  são consideradas a mesma aresta. Em um grafo não orientado, autoloops são proibidos, e assim toda aresta consiste em exatamente dois vértices distintos. A Figura B.2(b) é uma representação pictórica de um grafo não orientado no conjunto de vértices  $\{1, 2, 3, 4, 5, 6\}$ .

Muitas definições para grafos orientados e não orientados são idênticas, embora certos termos tenham significados ligeiramente diferentes nos dois contextos. Se  $(u, v)$  é uma aresta em um grafo orientado  $G = (V, E)$ , dizemos que  $(u, v)$  é *incidente do ou sai do* vértice  $u$  e é *incidente no ou entra no* vértice  $v$ . Por exemplo, as arestas que deixam o vértice 2 na Figura B.2(a) são  $(2, 2)$ ,  $(2, 4)$  e  $(2, 5)$ . As arestas que entram no vértice 2 são  $(1, 2)$  e  $(2, 2)$ . Se  $(u, v)$  é uma aresta em um grafo não orientado  $G = (V, E)$ , dizemos que  $(u, v)$  é *incidente nos* vértices  $u$  e  $v$ . Na Figura B.2(b), as arestas incidentes no vértice 2 são  $(1, 2)$  e  $(2, 5)$ .



**FIGURA B.2** Grafos orientados e não orientados. (a) Um grafo orientado  $G = (V, E)$ , onde  $V = \{1, 2, 3, 4, 5, 6\}$  e  $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$ . A aresta  $(2, 2)$  é um autoloop. (b) Um grafo não orientado  $G = (V, E)$ , onde  $V = \{1, 2, 3, 4, 5, 6\}$  e  $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$ . O vértice 4 é isolado. (c) O subgrafo do grafo da parte (a) induzido pelo conjunto de vértices  $\{1, 2, 3, 6\}$

Se  $(u, v)$  é uma aresta em um grafo  $G = (V, E)$ , dizemos que o vértice  $v$  é **adjacente** ao vértice  $u$ . Quando o grafo é não orientado, a relação de adjacência é simétrica. Quando o grafo é orientado, a relação de adjacência não é necessariamente simétrica. Se  $v$  é adjacente a  $u$  em um grafo orientado, às vezes escrevemos  $u \rightarrow v$ . Nas partes (a) e (b) da Figura B.2, o vértice 2 é adjacente ao vértice 1, pois a aresta  $(1, 2)$  pertence a ambos os grafos. O vértice 1 *não* é adjacente ao vértice 2 na Figura B.2(a), pois a aresta  $(2, 1)$  não pertence ao grafo.

O **grau** de um vértice em um grafo não orientado é o número de arestas incidentes nele. Por exemplo, o vértice 2 na Figura B.2(b) tem grau 2. Em um grafo orientado, o **grau de saída** de um vértice é o número de arestas que saem dele, e o **grau de entrada** de um vértice é o número de arestas que entram nele. O **grau** de um vértice em um grafo orientado é seu grau de entrada somado a seu grau de saída. O vértice 2 na Figura B.2(a) tem grau de entrada 2, grau de saída 3 e grau 5.

Um **caminho** de comprimento  $k$  de um vértice  $u$  até um vértice  $u'$  em um grafo  $G = (V, E)$  é uma seqüência  $\langle v_0, v_1, v_2, \dots, v_k \rangle$  de vértices tais que  $u = v_0$ ,  $u' = v_k$  e  $(v_{i-1}, v_i)$  para  $i = 1, 2, \dots, k$ . O comprimento do caminho é o número de arestas no caminho. O caminho **contém** os vértices  $v_0, v_1, \dots, v_k$  e as arestas  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ . (Sempre existe um caminho de comprimento 0 de  $u$  até  $u$ .) Se existe um caminho  $p$  de  $u$  até  $u'$ , dizemos que  $u'$  é **acessível** a partir de  $u$  via  $p$ , o que escrevemos algumas vezes como  $u \xrightarrow{p} u'$ , se  $G$  é orientado. Um caminho é **simples** se todos os vértices no caminho são distintos. Na Figura B.2(a), o caminho  $\langle 1, 2, 5, 4 \rangle$  é um caminho simples de comprimento 3. O caminho  $\langle 2, 5, 4, 5 \rangle$  não é simples.

Um **subcaminho** do caminho  $p = \langle v_0, v_1, \dots, v_k \rangle$  é uma subseqüência contígua de seus vértices. Isto é, para qualquer  $0 \leq i \leq j \leq k$ , a subseqüência de vértices  $\langle v_i, v_{i+1}, \dots, v_j \rangle$  é um subcaminho de  $p$ .

Em um grafo orientado, um caminho  $\langle v_0, v_1, \dots, v_k \rangle$  forma um **ciclo** se  $v_0 = v_k$  e o caminho contém pelo menos uma aresta. O ciclo é **simples** se, além disso,  $v_1, v_2, \dots, v_k$  são distintos. Um autoloop é um ciclo de comprimento 1. Dois caminhos  $\langle v_0, v_1, v_2, \dots, v_{k-1}, v_0 \rangle$  e  $\langle v'_0, v'_1, v'_2, \dots, v'_{k-1}, v'_0 \rangle$  formam o mesmo ciclo se existe um inteiro  $j$  tal que  $v'_i = v_{(i-j) \bmod k}$  para  $i = 0, 1, \dots, k-1$ . Na Figura B.2(a), o caminho  $\langle 1, 2, 4, 1 \rangle$  forma o mesmo ciclo que os caminhos  $\langle 2, 4, 1, 2 \rangle$  e  $\langle 4, 1, 2, 4 \rangle$ . Esse ciclo é simples, mas o ciclo  $\langle 1, 2, 4, 5, 4, 1 \rangle$  não é. O ciclo  $\langle 2, 2 \rangle$  formado pela aresta  $(2, 2)$  é um autoloop. Um grafo orientado sem autoloops é **simples**. Em um grafo não orientado, um caminho  $\langle v_0, v_1, \dots, v_k \rangle$  forma um **ciclo (simples)** se  $k \geq 3$ ,  $v_0 = v_k$  e  $v_1, v_2, \dots, v_k$  são distintos. Por exemplo, na Figura B.2(b), o caminho  $\langle 1, 2, 5, 1 \rangle$  é um ciclo. Um grafo sem ciclos é **acíclico**.

Um grafo não orientado é **conectado** se todo par de vértices está conectado por um caminho. Os **componentes conexos** de um grafo são as classes de equivalência de vértices sob a relação “é acessível a partir de”. O grafo da Figura B.2(b) tem três componentes conexos:  $\{1, 2, 5\}$ ,  $\{3, 6\}$  e  $\{4\}$ . Todo vértice em  $\{1, 2, 5\}$  é acessível a partir de cada um dos outros vértices em  $\{1, 2, 5\}$ . Um grafo não orientado é conectado se tem exatamente um componente conectado, isto é, se todo vértice é acessível a partir de todos os outros vértices.

Um grafo orientado é **fortemente conectado** se cada um de dois vértices quaisquer é acessível a partir do outro. Os **componentes fortemente conectados** de um grafo orientado são as classes de equivalência de vértices sob a relação “são mutuamente acessíveis”. Um grafo orientado é fortemente conectado se ele só tem um componente fortemente conectado. O grafo da Figura B.2(a) tem três componentes fortemente conectados:  $\{1, 2, 4, 5\}$ ,  $\{3\}$  e  $\{6\}$ . Todos os pares de vértices em  $\{1, 2, 4, 5\}$  são mutuamente acessíveis. Os vértices  $\{3, 6\}$  não formam um componente fortemente conectado, pois o vértice 6 não pode ser alcançado a partir do vértice 3.

Dois grafos  $G = (V, E)$  e  $G' = (V', E')$  são **isomórficos** se existe uma bijeção  $f: V \rightarrow V'$  tal que  $(u, v) \in E$  se e somente se  $(f(u), f(v)) \in E'$ . Em outras palavras, podemos identificar novamente os vértices de  $G$  como vértices de  $G'$ , mantendo as arestas correspondentes em  $G$  e  $G'$ . A Figura B.3(a) mostra um par de grafos isomórficos  $G$  e  $G'$  com os conjuntos de vértices respectivos  $V = \{1, 2, 3, 4, 5, 6\}$  e  $V' = \{u, v, w, x, y, z\}$ . O mapeamento de  $V$  para  $V'$  dado por  $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$  é a função bijetora exigida. Os grafos da Figura B.3(b) não são isomórficos. Embora ambos os grafos tenham 5 vértices e 7 arestas, o grafo superior tem um vértice de grau 4, e o grafo inferior não.

Dizemos que um grafo  $G' = (V', E')$  é um **subgrafo** de  $G = (V, E)$  se  $V' \subseteq V$  e  $E' \subseteq E$ . Dado um conjunto  $V' \subseteq V$ , o subgrafo de  $G$  **induzido** por  $V'$  é o grafo  $G' = (V', E')$ , onde

$$E' = \{(u, v) \in E : u, v \in V'\}.$$

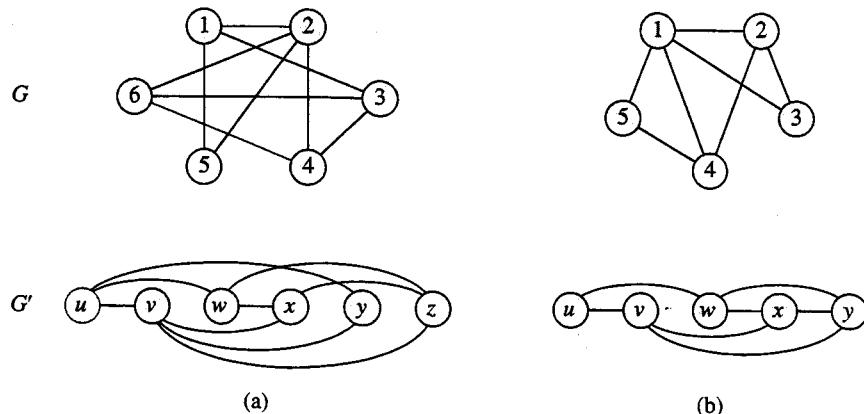


FIGURA B.3 (a) Um par de grafos isomórficos. Os vértices do grafo superior são mapeados para os vértices do grafo inferior por  $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$ . (b) Dois grafos que não são isomórficos, pois o grafo superior tem um vértice de grau 4 e o grafo inferior não tem.

O subgrafo induzido pelo conjunto de vértices  $\{1, 2, 3, 6\}$  na Figura B.2(a) aparece na Figura B.2(c) e tem o conjunto de arestas  $\{(1, 2), (2, 2), (6, 3)\}$ .

Dado um grafo não orientado  $G = (V, E)$ , a **versão orientada** de  $G$  é o grafo orientado  $G' = (V, E')$ , onde  $(u, v) \in E'$  se e somente se  $(u, v) \in E$ . Ou seja, cada aresta não orientada  $(u, v)$  em  $G$  é substituída na versão orientada pelas duas arestas orientadas  $(u, v)$  e  $(v, u)$ . Dado um grafo orientado  $G = (V, E)$ , a **versão não orientada** de  $G$  é o grafo não orientado  $G' = (V, E')$ , onde  $(u, v) \in E'$  se e somente se  $u \neq v$  e  $(u, v) \in E$ . Isto é, a versão não orientada contém as arestas de  $G$  “com suas orientações removidas” e com autoloops eliminados. (Como  $(u, v)$  e  $(v, u)$  são a mesma aresta em um grafo não orientado, a versão não orientada de um grafo orientado a contém somente uma vez, mesmo que o grafo orientado contenha ambas as arestas  $(u, v)$  e  $(v, u)$ .) Em um grafo orientado  $G = (V, E)$ , um **vizinho** de um vértice  $u$  é qualquer vértice que seja adjacente a  $u$  na versão não orientada de  $G$ . Ou seja,  $v$  é um vizinho de  $u$  se  $(u, v) \in E$  ou  $(v, u) \in E$ . Em um grafo não orientado,  $u$  e  $v$  são vizinhos se são adjacentes.

Vários tipos de grafos recebem nomes especiais. Um **grafo completo** é um grafo não orientado no qual todo par de vértices é adjacente. Um **grafo bipartido** é um grafo não orientado  $G$

$= (V, E)$  em que  $V$  pode ser particionado em dois conjuntos  $V_1$  e  $V_2$  tais que  $(u, v) \in E$  implica ou  $u \in V_1$  e  $v \in V_2$  ou  $u \in V_2$  e  $v \in V_1$ . Ou seja, todas as arestas ficam entre os dois conjuntos  $V_1$  e  $V_2$ . Um grafo acíclico não orientado é uma **floresta**, e um grafo conectado acíclico não orientado é uma **árvore (livre)** (ver Seção B.5). Com freqüência, usaremos as primeiras letras de “grafo acíclico orientado”, e chamaremos tal grafo um **gao**.

Há duas variantes de grafos que você poderá encontrar ocasionalmente. Um **multigrafo** é semelhante a um grafo não orientado, mas pode ter várias arestas entre vértices e também auto-loops. Um **hipergrafo** é semelhante a um grafo não orientado, mas cada **hiperaresta**, em lugar de conectar dois vértices, conecta um subconjunto arbitrário de vértices. Muitos algoritmos escritos para grafos orientados e não orientados comuns podem ser adaptados para funcionar nessas estruturas semelhantes a grafos.

A **contração** de um grafo não orientado  $G = (V, E)$  por uma aresta  $e = (u, v)$  é um grafo  $G' = (V', E')$ , onde  $V' = V - \{u, v\} \cup \{x\}$  e  $x$  é um novo vértice. O conjunto de arestas  $E'$  é formado a partir de  $E$  pela eliminação da aresta  $(u, v)$  e, para cada vértice  $w$  incidente em  $u$  ou  $v$ , eliminando-se o par dentre  $(u, w)$  e  $(v, w)$  que esteja em  $E$  e adicionando-se a nova aresta  $(x, w)$ .

## Exercícios

### B.4-1

Os participantes de uma festa na faculdade apertam as mãos para se cumprimentar, e cada professor memoriza quantas vezes tem de apertar as mãos de outras pessoas. No final da festa, o chefe de departamento efetua a soma do número de vezes que cada professor teve de apertar as mãos de outros. Mostre que o resultado é par, provando o **lema do cumprimento**: se  $G = (V, E)$  é um grafo não orientado, então

$$\sum_{v \in V} \text{grau}(v) = 2|E| .$$

### B.4-2

Mostre que, se um grafo orientado ou não orientado contém um caminho entre dois vértices  $u$  e  $v$ , então ele contém um caminho simples entre  $u$  e  $v$ . Mostre que, se um grafo orientado contém um ciclo, então ele contém um ciclo simples.

### B.4-3

Mostre que qualquer grafo conectado não orientado  $G = (V, E)$  satisfaz a  $|E| \geq |V| - 1$ .

### B.4-4

Verifique se, em um grafo não orientado, a relação “é acessível a partir de” é uma relação de equivalência sobre os vértices do grafo. Qual das três propriedades de uma relação de equivalência é válida em geral para a relação “é acessível a partir de” sobre os vértices de um grafo orientado?

### B.4-5

Qual é a versão não orientada do grafo orientado da Figura B.2(a)? Qual é a versão orientada do grafo não orientado da Figura B.2(b)?

### B.4-6 \*

Mostre que um hipergrafo pode ser representado por um grafo bipartido, se permitirmos que a incidência no hipergrafo corresponda à adjacência no grafo bipartido. (*Sugestão:* Faça um conjunto de vértices no grafo bipartido corresponder a vértices do hipergrafo e faça o outro conjunto de vértices do grafo bipartido corresponder a hiperarestas.)

## B.5 Árvores

Como ocorre no caso dos grafos, existem muitas noções de árvores relacionadas entre si, embora ligeiramente diferentes. Esta seção apresenta definições e propriedades matemáticas de vários tipos de árvores. As Seções 10.4 e 22.1 descrevem como as árvores podem ser representadas na memória de um computador.

### B.5.1 Árvores livres

Como definimos na Seção B.4, uma *árvore livre* é um grafo acíclico não orientado conectado. Com freqüência, omitimos o adjetivo “livre” quando dizemos que um grafo é uma árvore. Se um grafo não orientado é acíclico mas possivelmente desconectado, ele é uma *floresta*. Muitos algoritmos que funcionam para árvores também funcionam para florestas. A Figura B.4(a) mostra uma árvore livre e a Figura B.4(b) mostra uma floresta. A floresta da Figura B.4(b) não é uma árvore porque não é conectada. O grafo da Figura B.4(c) não é nem uma árvore nem uma floresta, porque contém um ciclo.

O teorema a seguir engloba muitos fatos importantes sobre árvores livres.

#### **Teorema B.2 (Propriedades de árvores livres)**

Seja  $G = (V, E)$  um grafo não orientado. As declarações a seguir são equivalentes.

1.  $G$  é uma árvore livre.
2. Dois vértices quaisquer em  $G$  estão conectados por um caminho simples único.
3.  $G$  é conectado mas, se qualquer aresta for removida de  $E$ , o grafo resultante será desconectado.
4.  $G$  é conectado, e  $|E| = |V| - 1$ .
5.  $G$  é acíclico, e  $|E| = |V| - 1$ .
6.  $G$  é acíclico mas, se qualquer aresta for adicionada a  $E$ , o grafo resultante conterá um ciclo.

**Prova** (1)  $\Rightarrow$  (2): Tendo em vista que uma árvore é conectada, dois vértices quaisquer em  $G$  são conectados por pelo menos um caminho simples. Sejam  $u$  e  $v$  vértices que estão conectados por dois caminhos simples distintos  $p_1$  e  $p_2$ , como mostra a Figura B.5. Seja  $w$  o vértice no qual os caminhos divergem pela primeira vez; isto é,  $w$  é o primeiro vértice em  $p_1$  e  $p_2$  cujo sucessor em  $p_1$  é  $x$  e cujo sucessor em  $p_2$  é  $y$ , onde  $x \neq y$ . Seja  $z$  o primeiro vértice no qual os caminhos voltam a convergir; ou seja,  $z$  é o primeiro vértice seguinte a  $w$  em  $p_1$  que também está em  $p_2$ . Seja  $p'$  o subcaminho de  $p_1$  a partir de  $w$  através de  $x$  até  $z$ , e seja  $p''$  o subcaminho de  $p_2$  a partir de  $w$  através de  $y$  até  $z$ . Os caminhos  $p'$  e  $p''$  não compartilham nenhum vértice, exceto seus pontos extremos. Portanto, o caminho obtido pela concatenação de  $p'$  com o inverso de  $p''$  é um ciclo. Isso contradiz nossa hipótese de que  $G$  é uma árvore. Desse modo, se  $G$  é uma árvore, pode haver no máximo um caminho simples entre dois vértices.

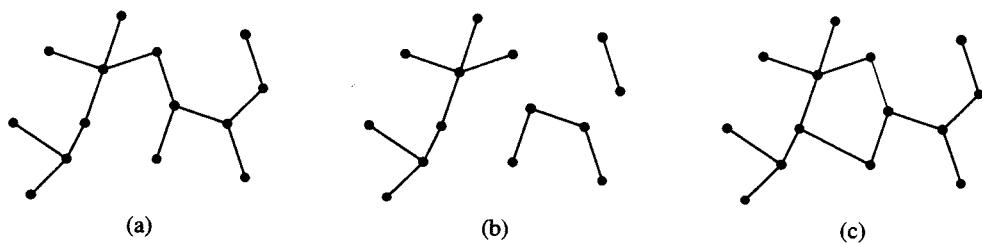
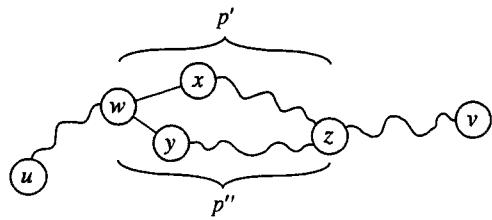


FIGURA B.4 (a) Uma árvore livre. (b) Uma floresta. (c) Um grafo que contém um ciclo e que, por essa razão, não é uma árvore nem uma floresta

(2)  $\Rightarrow$  (3): Se dois vértices quaisquer em  $G$  estão conectados por um caminho simples único, então  $G$  é conectado. Seja  $(u, v)$  qualquer aresta em  $E$ . Essa aresta é um caminho de  $u$  até  $v$  e, portanto, deve ser o caminho único de  $u$  até  $v$ . Se removermos  $(u, v)$  de  $G$ , não haverá nenhum caminho de  $u$  até  $v$  e, consequentemente, sua remoção desconecta  $G$ .



**FIGURA B.5** Uma etapa na prova do Teorema B.2: se (1)  $G$  é uma árvore livre, então (2) dois vértices quaisquer em  $G$  estão conectados por um caminho simples único. Suponha para fins de contradição que os vértices  $u$  e  $v$  estejam conectados por dois caminhos simples distintos  $p_1$  e  $p_2$ . Esses caminhos divergem primeiro no vértice  $w$ , e depois voltam a convergir primeiro no vértice  $z$ . O caminho  $p'$  concatenado com o inverso do caminho  $p''$  forma um ciclo, o que produz a contradição

(3)  $\Rightarrow$  (4): Por hipótese, o grafo  $G$  é conectado e, pelo Exercício B.4-3, temos  $|E| \geq |V| - 1$ . Provaremos  $|E| \leq |V| - 1$  por indução. Um grafo conectado com  $n = 1$  ou  $n = 2$  vértices tem  $n - 1$  arestas. Suponha que  $G$  tenha  $n \geq 3$  vértices e que todos os grafos que satisfazem a (3) com menos de  $n$  vértices também satisfazem a  $|E| \leq |V| - 1$ . A remoção de uma aresta arbitrária de  $G$  separa o grafo em  $k \geq 2$  componentes conexos (na realidade,  $k = 2$ ). Cada componente satisfaz a (3) ou, do contrário,  $G$  não satisfaria a (3). Assim, por indução, o número de arestas em todos os componentes combinados é no máximo  $|V| - k \geq |V| - 2$ . A adição da aresta removida produz  $|E| \leq |V| - 1$ .

(4)  $\Rightarrow$  (5): Suponha que  $G$  seja conectado e que  $|E| = |V| - 1$ . Devemos mostrar que  $G$  é acíclico. Vamos supor que  $G$  tenha um ciclo contendo  $k$  vértices  $v_1, v_2, \dots, v_k$  e, sem perda de generalidade, suponha que esse ciclo seja simples. Seja  $G_k = (V_k, E_k)$  o subgrafo de  $G$  que consiste no ciclo. Observe que  $|V_k| = |E_k| = k$ . Se  $k < |V|$ , deve existir um vértice  $v_{k+1} \in V - V_k$  que seja adjacente a algum vértice  $v_i \in V_k$ , pois  $G$  é conectado. Defina  $G_{k+1} = (V_{k+1}, E_{k+1})$  como o subgrafo de  $G$  com  $V_{k+1} = V_k \cup \{v_{k+1}\}$  e  $E_{k+1} = E_k \cup \{v_i, v_{k+1}\}$ . Observe que  $|V_{k+1}| = |E_{k+1}|$ . Se  $k+1 < n$ , podemos continuar, definindo  $G_{k+2}$  da mesma maneira, e assim por diante até obtermos  $G_n = (V_n, E_n)$ , onde  $n = |V|$ ,  $V_n = V$  e  $|E_n| = |V_n| = |V|$ . Como  $G_n$  é um subgrafo de  $G$ , temos  $E_n \subseteq E$  e, portanto,  $|E| \geq |V|$ , o que contradiz a hipótese de que  $|E| = |V| - 1$ . Desse modo,  $G$  é acíclico.

(5)  $\Rightarrow$  (6): Suponha que  $G$  seja acíclico e que  $|E| = |V| - 1$ . Seja  $k$  o número de componentes conectados de  $G$ . Cada componente conexo é uma árvore livre por definição e, como (1) implica (5), a soma de todas as arestas em todos os componentes conexos de  $G$  é  $|V| - k$ . Conseqüentemente, devemos ter  $k = 1$ , e  $G$  é de fato uma árvore. Como (1) implica (2), dois vértices quaisquer em  $G$  estão conectados por um caminho simples único. Portanto, a adição de qualquer aresta a  $G$  cria um ciclo.

(6)  $\Rightarrow$  (1): Suponha que  $G$  seja acíclico mas que, se qualquer aresta for adicionada a  $E$ , seja criado um ciclo. Devemos mostrar que  $G$  é conectado. Sejam  $u$  e  $v$  vértices arbitrários em  $G$ . Se  $u$  e  $v$  ainda não forem adjacentes, a adição da aresta  $(u, v)$  criará um ciclo no qual todas as arestas com exceção de  $(u, v)$  pertencerão a  $G$ . Desse modo, existe um caminho de  $u$  até  $v$  e, como  $u$  e  $v$  foram escolhidos arbitrariamente,  $G$  é conectado. ■

## B.5.2 Árvores enraizadas e ordenadas

Uma **árvore enraizada** é uma árvore livre na qual um dos vértices se distingue dos outros. O vértice distinto é chamado **raiz** da árvore. Com freqüência, faremos referência a um vértice de uma árvore enraizada como um **nó**<sup>4</sup> da árvore. A Figura B.6(a) mostra uma árvore enraizada em um conjunto de 12 nós com raiz 7.

<sup>4</sup>O termo “nó” (ou “nodo”) é usado com freqüência na literatura sobre a teoria dos grafos como sinônimo de “vértice”. Reservaremos o termo “nó” para indicar um vértice de uma árvore enraizada.

Considere um nó  $x$  em uma árvore enraizada  $T$  com raiz  $r$ . Qualquer nó  $y$  no caminho único de  $r$  a  $x$  é chamado **ancestral** de  $x$ . Se  $y$  é um ancestral de  $x$ , então  $x$  é um **descendente** de  $y$ . (Todo nó é ao mesmo tempo um ancestral e um descendente de si próprio.) Se  $y$  é um ancestral de  $x$  e  $x \neq y$ , então  $y$  é um **ancestral próprio** de  $x$ , e  $x$  é um **descendente próprio** de  $y$ . A **subárvore enraizada em  $x$**  é a árvore induzida pelos descendentes de  $x$ , com raiz em  $x$ . Por exemplo, a subárvore enraizada no nó 8 na Figura B.6(a) contém os nós 8, 6, 5 e 9.

Se a última aresta no caminho da raiz  $r$  de uma árvore  $T$  até um nó  $x$  é  $(y, x)$ , então  $y$  é o **pai** de  $x$ , e  $x$  é um **filho** de  $y$ . A raiz é o único nó em  $T$  que não tem pai. Se dois nós têm o mesmo pai, eles são **irmãos**. Um nó sem filhos é um **nó externo** ou **folha**. Um nó que não é uma folha é um **nó interno**.

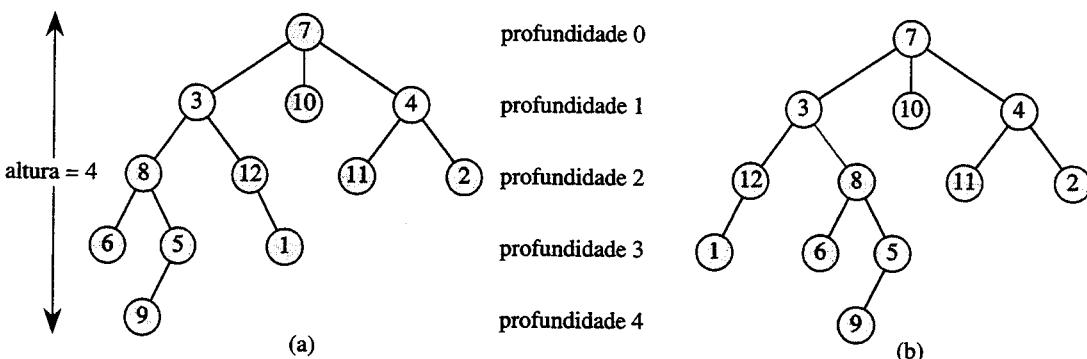


FIGURA B.6 Árvores enraizadas e ordenadas. (a) Uma árvore enraizada com altura 4. A árvore é desenhada de um modo padrão: a raiz (nó 7) está na parte superior, seus filhos (os nós com profundidade 1) estão abaixo dela, os filhos dos filhos (nós com profundidade 2) estão abaixo deles e assim por diante. Se a árvore é ordenada, a ordem relativa da esquerda para a direita dos filhos de um nó é importante; caso contrário, ela não é importante. (b) Outra árvore enraizada. Sendo uma árvore enraizada, ela é idêntica à árvore em (a) mas, como árvore ordenada é diferente, pois os filhos do nó 3 aparecem em uma ordem distinta

O número de filhos de um nó  $x$  em uma árvore enraizada  $T$  é chamado **grau** de  $x$ .<sup>5</sup> O comprimento do caminho desde a raiz  $r$  até um nó  $x$  é a **profundidade** de  $x$  em  $T$ . A **altura** de um nó em uma árvore é o número de arestas no caminho descendente simples mais longo desde o nó até uma folha, e a altura de uma árvore é a altura de sua raiz. A altura de uma árvore também é igual à maior profundidade de qualquer nó na árvore.

Uma **árvore ordenada** é uma árvore enraizada na qual os filhos de cada nó estão ordenados. Isto é, se um nó tem  $k$  filhos, então existe um primeiro filho, um segundo filho, ... e um  $k$ -ésimo filho. As duas árvores da Figura B.6 são diferentes quando consideradas árvores ordenadas, mas são idênticas quando são consideradas apenas árvores enraizadas.

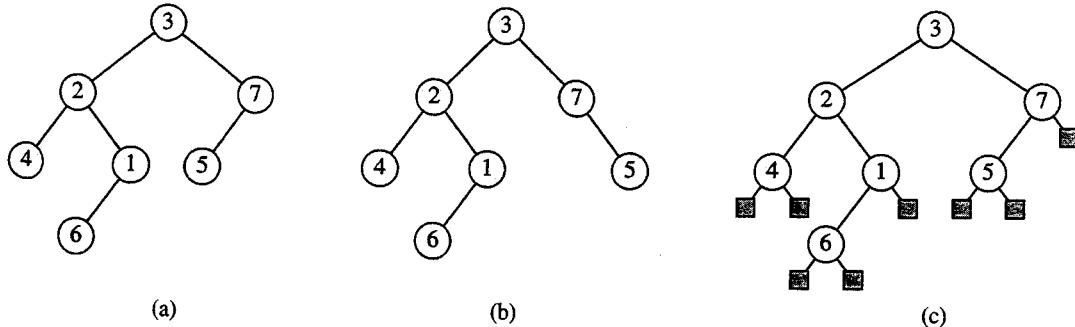
### B.5.3 Árvores binárias e árvores posicionais

As árvores binárias são definidas de modo recursivo. Uma **árvore binária**  $T$  é uma estrutura definida em um conjunto finito de nós que

- não contém nenhum nó, ou
- é formada por três conjuntos disjuntos de nós: um nó **raiz**, uma árvore binária chamada sua **subárvore da esquerda**, e uma árvore binária chamada sua **subárvore da direita**.

<sup>5</sup> Observe que o grau de um nó depende de  $T$  ser considerada uma árvore enraizada ou uma árvore livre. O grau de um vértice em uma árvore livre é, como em qualquer grafo não orientado, o número de vértices adjacentes. Porém, em uma árvore enraizada, o grau é o número de filhos – o pai de um nó não conta para definir seu grau.

A árvore binária que não contém nenhum nó é chamada **árvore vazia** ou **árvore nula**, algumas vezes denotada por NIL. Se a subárvore da esquerda é não vazia, sua raiz é chamada **filho da esquerda** da raiz da árvore inteira. Da mesma forma, a raiz de uma subárvore da direita não nula é o **filho da direita** da raiz da árvore inteira. Se uma subárvore é a árvore nula NIL, dizemos que o filho está **ausente** ou **faltando**. A Figura B.7(a) mostra uma árvore binária.

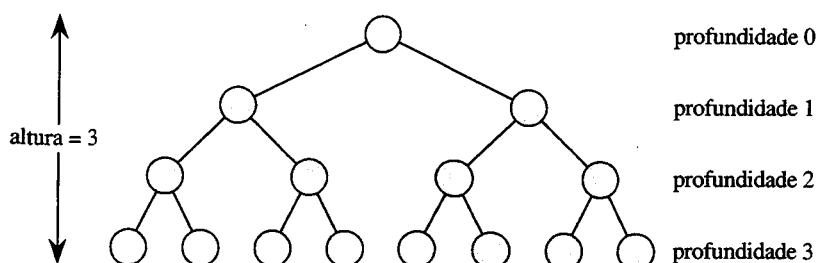


**FIGURA B.7** Árvores binárias. (a) Uma árvore binária desenhada de um modo padrão. O filho da esquerda de um nó é desenhado abaixo do nó e à esquerda. O filho da direita é desenhado abaixo e à direita do nó. (b) Uma árvore binária diferente da que está em (a). Em (a), o filho da esquerda do nó 7 é 5, e o filho da direita está ausente. Em (b), o filho da esquerda do nó 7 está ausente e o filho da direita é 5. Como árvores ordenadas, essas árvores são idênticas mas, como árvores binárias, elas são distintas. (c) A árvore binária em (a) representada pelos nós internos de uma árvore binária cheia: uma árvore ordenada na qual cada nó interno tem grau 2. As folhas na árvore são mostradas como quadrados

Uma árvore binária não é simplesmente uma árvore ordenada na qual cada nó tem grau máximo 2. Por exemplo, em uma árvore binária, se um nó tem apenas um filho, a posição do filho – seja ele o **filho da esquerda** ou o **filho da direita** – é importante. Em uma árvore ordenada, não há distinção de um único filho como da esquerda ou da direita. A Figura B.7(b) mostra uma árvore binária que difere da árvore da Figura B.7(a) por causa da posição de um único nó. Contudo, considerando-se as árvores ordenadas, as duas árvores são idênticas.

As informações de posicionamento em uma árvore binária podem ser representadas pelos nós internos de uma árvore ordenada, como mostra a Figura B.7(c). A idéia é substituir cada filho que falta na árvore binária por um nó que não tem nenhum filho. Esses nós folhas estão desenhados como quadrados na figura. A árvore resultante é uma **árvore binária cheia**: cada nó é ou uma folha ou tem grau exatamente igual a 2. Não existe nenhum nó de grau 1. Em consequência disso, a ordem dos filhos de um nó preserva as informações de posição.

As informações de posicionamento que distinguem árvores binárias de árvores ordenadas podem ser estendidas a árvores com mais de 2 filhos por nó. Em uma **árvore posicional**, os filhos de um nó são identificados com inteiros positivos distintos. O  $i$ -ésimo filho de um nó é **ausente** se nenhum filho é identificado com o inteiro  $i$ . Uma árvore  **$k$ -ária** é uma árvore posicional na qual, para todo nó, todos os filhos com rótulos maiores que  $k$  estão faltando. Desse modo, uma árvore binária é uma árvore  $k$ -ária com  $k = 2$ .



**FIGURA B.8** Uma árvore binária completa de altura 3 com 8 folhas e 7 nós internos

Uma árvore ***k*-ária completa** é uma árvore *k*-ária na qual todas as folhas têm a mesma profundidade e todos os nós internos têm grau *k*. A Figura B.8 mostra uma árvore binária completa de altura 3. Quantas folhas tem uma árvore *k*-ária completa de altura *b*? A raiz tem *k* filhos à profundidade 1, cada um dos quais tem *k* filhos à profundidade 2 e assim por diante. Portanto, o número de folhas à profundidade *b* é  $k^b$ . Conseqüentemente, a altura de uma árvore *k*-ária completa com *n* folhas é  $\log_k n$ . O número de nós internos de uma árvore *k*-ária completa de altura *b* é

$$1 + k + k^2 + \dots + k^{b-1} = \sum_{i=0}^{b-1} k^i \\ = \frac{k^b - 1}{k - 1}$$

pela equação (A.5). Assim, uma árvore binária completa tem  $2^b - 1$  nós internos.

## Exercícios

### B.5-1

Desenhe todas as árvores livres compostas pelos 3 vértices *A*, *B* e *C*. Trace todas as árvores enraizadas com nós *A*, *B* e *C*, que têm *A* como raiz. Desenhe todas as árvores ordenadas com nós *A*, *B* e *C* que têm *A* como raiz. Trace todas as árvores binárias com nós *A*, *B* e *C* que têm *A* como raiz.

### B.5-2

Seja  $G = (V, E)$  um grafo acíclico orientado no qual existe um vértice  $v_0 \in V$  tal que existe um caminho único de  $v_0$  até todo vértice  $v \in V$ . Prove que a versão não orientada de  $G$  forma uma árvore.

### B.5-3

Mostre por indução que o número de nós de grau 2 em qualquer árvore binária não vazia é uma unidade menor que o número de folhas.

### B.5-4

Mostre por indução que uma árvore binária não vazia com *n* nós tem altura pelo menos igual a  $\lfloor \lg n \rfloor$ .

### B.5-5 \*

O **comprimento do caminho interno** de uma árvore binária cheia é a soma, considerada sobre todos os nós internos da árvore, da profundidade de cada nó. Da mesma forma, o **comprimento do caminho externo** é a soma, considerada sobre todas as folhas da árvore, da profundidade de cada folha. Seja uma árvore binária cheia com *n* nós internos, comprimento de caminho interno *i* e comprimento de caminho externo *e*. Prove que  $e = i + 2n$ .

### B.5-6 \*

Vamos associar um “peso”  $w(x) = 2^d$  a cada folha *x* de profundidade *d* em uma árvore binária *T*. Prove que  $\sum_x w(x) \leq 1$ , onde a soma é considerada sobre todas as folhas *x* de *T*. (Isso é conhecido como **desigualdade de Kraft**.)

### B.5-7 \*

Mostre que toda árvore binária com *L* folhas contém uma subárvore que tem entre  $L/3$  e  $2L/3$  folhas, inclusive.

## Problemas

### B-1 Coloração de grafos

Uma ***k*-coloração** de um grafo não orientado  $G = (V, E)$  é uma função  $c : V \rightarrow \{0, 1, \dots, k-1\}$  tal que  $c(u) \neq c(v)$  para toda aresta  $(u, v) \in E$ . Em outras palavras, os números  $0, 1, \dots, k-1$  representam as *k* cores, e vértices adjacentes devem ter cores diferentes.

- a. Mostre que qualquer árvore pode ter 2 cores.
- b. Mostre que os itens seguintes são equivalentes:
  - 1.  $G$  é bipartido.
  - 2.  $G$  pode ter duas cores.
  - 3.  $G$  não tem nenhum ciclo de comprimento ímpar.
- c. Seja  $d$  o grau máximo de qualquer vértice em um grafo  $G$ . Prove que  $G$  pode ser colorido com  $d + 1$  cores.
- d. Mostre que, se  $G$  tem  $O(|V|)$  arestas, então  $G$  pode ser colorido com  $O(\sqrt{|V|})$  cores.

### B-2 Grafos amistosos

Reformule cada uma das instruções a seguir como um teorema sobre grafos não orientados, e depois prove-o. Suponha que esse caráter amistoso seja simétrico, mas não reflexivo.

- a. Em qualquer grupo de  $n \geq 2$  pessoas, existem duas pessoas com o mesmo número de amigos no grupo.
- b. Todo grupo de seis pessoas contém três amigos mútuos ou três estranhos mútuos.
- c. Qualquer grupo de pessoas pode ser particionado em dois subgrupos tais que pelo menos metade dos amigos de cada pessoa pertence ao subgrupo do qual essa pessoa *não* é um membro.
- d. Se toda pessoa em um grupo é amiga de pelo menos metade das pessoas no grupo, então o grupo pode se sentar em torno de uma mesa de tal modo que toda pessoa fique sentada entre dois amigos.

### B-3 Bisseção de árvores

Muitos algoritmos de dividir e conquistar que operam sobre grafos exigem que o grafo seja dividido em dois subgrafos de tamanho praticamente igual, que são induzidos por uma partição dos vértices. Este problema investiga a bisseção de árvores formadas pela remoção de um pequeno número de arestas. Exigimos que, sempre que dois vértices terminarem na mesma subárvore depois que as arestas forem removidas, eles estejam obrigatoriamente na mesma partição.

- a. Mostre que, removendo uma única aresta, podemos partitionar os vértices de qualquer árvore binária de  $n$  vértices em dois conjuntos  $A$  e  $B$  tais que  $|A| \leq 3n/4$  e  $|B| \leq 3n/4$ .
- b. Mostre que a constante  $3/4$  na parte (a) é ótima no pior caso, dando um exemplo de uma árvore simples cuja partição balanceada de modo mais uniforme após a remoção de uma única aresta tem  $|A| = 3n/4$ .
- c. Mostre que, removendo no máximo  $O(\lg n)$  arestas, podemos partitionar os vértices de qualquer árvore binária de  $n$  vértices em dois conjuntos  $A$  e  $B$  tais que  $|A| = \lfloor n/2 \rfloor$  e  $|B| = \lceil n/2 \rceil$ .

## Notas do capítulo

G. Boole foi o pioneiro no desenvolvimento da lógica simbólica e introduziu muitas das notações básicas de conjuntos em um livro publicado em 1854. A moderna teoria dos conjuntos foi criada por G. Cantor durante o período de 1874 a 1895. Cantor focalizou principalmente os conjuntos de cardinalidade infinita. O termo “função” é atribuído a G. W. Leibnitz, que o usou para se referir a várias espécies de fórmulas matemáticas. Sua definição limitada foi generalizada várias vezes. A teoria dos grafos teve origem em 1736, quando L. Euler provou que era impossível cruzar cada uma das sete pontes da cidade de Königsberg exatamente uma vez e retornar ao ponto de partida.

Um compêndio útil com muitas definições e resultados da teoria dos grafos é o livro de Harary [138].

## *Apêndice C*

# *Contagem e probabilidade*

Este capítulo examina a análise combinatória elementar e a teoria das probabilidades. Se tem um bom conhecimento dessas áreas, talvez você deseje dar uma olhada rápida no início do capítulo e se concentrar nas últimas seções. A maior parte dos capítulos não requer nenhum conhecimento de probabilidade, mas para alguns capítulos esse conhecimento é essencial.

A Seção C.1 examina resultados elementares de teoria da contagem, incluindo fórmulas padrão para contagem de permutações e combinações. Os axiomas de probabilidade e os fatos básicos relativos a distribuições de probabilidade são apresentados na Seção C.2. As variáveis aleatórias são introduzidas na Seção C.3, juntamente com as propriedades de expectativa e variância. A Seção C.4 investiga as distribuições geométricas e binomiais que surgem do estudo de experiências de Bernoulli. O estudo da distribuição binomial continua na Seção C.5, uma discussão avançada das “extremidades” da distribuição.

### **C.1 Contagem**

A teoria da contagem tenta responder à pergunta “Quantos?” sem realmente enumerar quantos. Por exemplo, podemos perguntar: “Quantos números diferentes de  $n$  bits existem?” ou “Quantas ordenações de  $n$  elementos distintos existem?” Nesta seção, examinaremos os elementos da teoria da contagem. Tendo em vista que uma parte do material pressupõe uma compreensão básica de conjuntos, é aconselhável que o leitor comece pela revisão do material da Seção B.1.

#### **Regras de soma e produto**

Um conjunto de itens que desejamos contar pode às vezes ser expresso como uma união de conjuntos disjuntos ou como um produto cartesiano de conjuntos.

A **regra da soma** afirma que o número de maneiras de escolher um elemento de um entre dois conjuntos *disjuntos* é a soma das cardinalidades dos conjuntos. Ou seja, se  $A$  e  $B$  são dois conjuntos finitos sem nenhum elemento em comum, então  $|A \cup B| = |A| + |B|$ , que decorre da equação (B.3). Por exemplo, cada posição na placa de um automóvel é uma letra ou um dígito. O número de possibilidades para cada posição é portanto  $26 + 10 = 36$ , pois existem 26 escolhas se for uma letra e 10 escolhas se for um dígito.

A **regra do produto** afirma que o número de maneiras de escolher um par ordenado é o número de maneiras de escolher o primeiro elemento vezes o número de maneiras de escolher o segundo elemento. Isto é, se  $A$  e  $B$  são dois conjuntos finitos, então  $|A \times B| = |A| \cdot |B|$ , que é simplesmente a equação (B.4). Por exemplo, se uma sorveteria oferece 28 sabores de sorvete e 4 coberturas, o número de sundaes possíveis com uma porção de sorvete e uma cobertura é  $28 \cdot 4 = 112$ .

## Cadeias

Uma **cadeia** sobre um conjunto finito  $S$  é uma seqüência de elementos de  $S$ . Por exemplo, existem 8 cadeias binárias de comprimento 3:

000, 001, 010, 011, 100, 101, 110, 111 .

Algumas vezes, chamamos um cadeia de comprimento  $k$  de **cadeia de  $k$  elementos**. Uma **subcadeia**  $s'$  de um cadeia  $s$  é uma seqüência ordenada de elementos consecutivos de  $s$ . Uma **subcadeia de  $k$  elementos** de uma cadeia é uma subcadeia de comprimento  $k$ . Por exemplo, 010 é uma subcadeia de 3 elementos de 01101001 (a subcadeia de 3 elementos que começa na posição 4), mas 111 não é uma subcadeia de 01101001.

Uma cadeia de  $k$  elementos sobre um conjunto  $S$  pode ser vista como um elemento do produto cartesiano  $S^k$  de tuplas de  $k$  elementos; desse modo, existem  $|S|^k$  cadeias de comprimento  $k$ . Por exemplo, o número de cadeias binárias de  $k$  elementos é  $2^k$ . Intuitivamente, para construir uma cadeia de  $k$  elementos sobre um conjunto de  $n$  elementos, temos  $n$  modos para escolher o primeiro elemento; para cada uma dessas opções, temos  $n$  modos de escolher o segundo elemento e assim por diante  $k$  vezes. Essa construção conduz ao produto de  $k$  termos  $n \cdot n \cdots n = n^k$  como o número de cadeias de  $k$  elementos.

## Permutações

Uma **permutação** de um conjunto finito  $S$  é uma seqüência ordenada de todos os elementos de  $S$ , com cada elemento aparecendo exatamente uma vez. Por exemplo, se  $S = \{a, b, c\}$ , existem 6 permutações de  $S$ :

abc, acb, bac, bca, cab, cba .

Existem  $n!$  permutações de um conjunto de  $n$  elementos, pois o primeiro elemento da seqüência pode ser escolhido de  $n$  modos, o segundo de  $n - 1$  modos, o terceiro de  $n - 2$  modos e assim por diante.

Uma **permutação de  $k$  elementos** de  $S$  é uma seqüência ordenada de  $k$  elementos de  $S$ , sem qualquer elemento aparecendo mais de uma vez na seqüência. (Desse modo, uma permutação comum é apenas uma permutação de  $n$  elementos de um conjunto de  $n$  elementos.) As doze permutações de 2 elementos do conjunto  $\{a, b, c, d\}$  são

ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc .

O número de permutações de  $k$  elementos de um conjunto de  $n$  elementos é

$$n(n - 1)(n - 2) \dots (n - k + 1) = \frac{n!}{(n - k)!}, \quad (\text{C.1})$$

pois há  $n$  modos de escolher o primeiro elemento,  $n - 1$  modos de escolher o segundo elemento e assim por diante até  $k$  elementos serem selecionados, sendo o último uma seleção de  $n - k + 1$  elementos.

## Combinações

Uma **combinação de  $k$  elementos** de um conjunto  $S$  de  $n$  elementos é simplesmente um subconjunto de  $k$  elementos de  $S$ . Existem seis combinações de 2 elementos do conjunto de 4 elementos  $\{a, b, c, d\}$ :

ab, ac, ad, bc, bd, cd .

(Aqui, usamos o recurso prático de denotar o conjunto de 2 elementos  $\{a, b\}$  por  $ab$  e assim por diante.) Podemos construir uma combinação de  $k$  elementos de um conjunto de  $n$  elementos escolhendo  $k$  elementos distintos (diferentes) do conjunto de  $n$  elementos.

O número de combinações de  $k$  elementos de um conjunto de  $n$  elementos pode ser expresso em termos do número de permutações de  $k$  elementos de um conjunto de  $n$  elementos. Para toda combinação de  $k$  elementos, existem exatamente  $k!$  permutações de seus elementos, cada uma das quais é uma permutação de  $k$  elementos distinta do conjunto de  $n$  elementos. Desse modo, o número de combinações de  $k$  elementos de um conjunto de  $n$  elementos é o número de permutações de  $k$  elementos dividido por  $k!$ ; da equação (C.1), essa quantidade é

$$\frac{n!}{k!(n-k)!} \quad (\text{C.2})$$

Para  $k = 0$ , essa fórmula nos informa que o número de maneiras de escolher 0 elementos de um conjunto de  $n$  elementos é 1 (e não 0), pois  $0! = 1$ .

## Coeficientes binomiais

Usamos a notação  $\binom{n}{k}$  (lê-se “escolher  $k$  entre  $n$  elementos”) para denotar o número de combinações de  $k$  elementos de um conjunto de  $n$  elementos. Da equação (C.2), temos

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Essa fórmula é simétrica em  $k$  e em  $n - k$ :

$$\binom{n}{k} = \binom{n!}{n-k}. \quad (\text{C.3})$$

Esses números também são conhecidos como *coeficientes binomiais*, pelo fato de aparecerem na *expansão binomial*:

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}. \quad (\text{C.4})$$

Um caso especial da expansão binomial ocorre quando  $x = y = 1$ :

$$2^n = \sum_{k=0}^n \binom{n}{k}.$$

Essa fórmula corresponde à contagem das  $2^n$  cadeias binárias de  $n$  elementos pelo número de elementos iguais a 1 que elas contêm: existem  $\binom{n}{k}$  cadeias binárias de  $n$  elementos contendo exatamente  $k$  elementos iguais a 1, pois existem  $\binom{n}{k}$  maneiras de escolher  $k$  das  $n$  posições nas quais são colocados os elementos de valor 1.

Existem muitas identidades envolvendo coeficientes binomiais. Os exercícios no final desta seção lhe darão a oportunidade de provar algumas delas.

## Limites binomiais

Algumas vezes, precisamos limitar o tamanho de um coeficiente binomial. Para  $1 \leq k \leq n$ , temos o limite inferior

$$\begin{aligned} \binom{n}{k} &= \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots1} \\ &= \binom{n}{k} \left( \frac{n-1}{k-1} \right) \dots \left( \frac{n-k+1}{1} \right) \\ &\geq \left( \frac{n}{k} \right)^k. \end{aligned}$$

Tirando proveito da desigualdade  $K! \geq (k/e)^k$  derivada da aproximação de Stirling (3.17), obtemos os limites superiores

$$\begin{aligned} \binom{n}{k} &= \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots1} \\ &\leq \frac{n^k}{k!} \\ &\leq \left(\frac{en}{k}\right)^k. \end{aligned} \quad (\text{C.5})$$

Para todo  $0 \leq k \leq n$ , podemos usar a indução (ver Exercício C.1-12) para provar o limite

$$\binom{n}{k} \leq \frac{n^n}{k^k (n-k)^{n-k}}, \quad (\text{C.6})$$

onde, por conveniência, supomos que  $0^0 = 1$ . Para  $k = \lambda n$ , onde  $0 \leq \lambda \leq n$ , esse limite pode ser reescrito como

$$\begin{aligned} \binom{n}{\lambda k} &\leq \frac{n^n}{(\lambda n)^{\lambda n} ((1-\lambda)n)^{(1-\lambda)n}} \\ &= \left( \left( \frac{1}{\lambda} \right)^\lambda \left( \frac{1}{1-\lambda} \right)^{1-\lambda} \right)^n \\ &= 2^{nH(\lambda)}, \end{aligned}$$

onde

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda) \lg (1-\lambda) \quad (\text{C.7})$$

é a **função entropia (binária)** e onde, por conveniência, supomos que  $0 \lg 0 = 0$ , de modo que  $H(0) = H(1) = 0$ .

## Exercícios

### C.1-1

Quantas subcadeias de  $k$  elementos tem uma cadeia de  $n$  elementos? (Considere subcadeias idênticas de  $k$  elementos em posições diferentes como subcadeias diferentes.) Quantas subcadeias uma cadeia de  $n$  elementos tem no total?

### C.1-2

Uma **função booleana** de entrada  $n$  e saída  $m$  é uma função de  $\{\text{TRUE}, \text{FALSE}\}^n$  para  $\{\text{TRUE}, \text{FALSE}\}^m$ . Quantas funções booleanas de entrada  $n$  e saída 1 existem? Quantas funções booleanas de entrada  $n$  e saída  $m$  existem?

### C.1-3

De quantos modos  $n$  professores podem se sentar em torno de uma mesa de conferência circular? Considere duas posições de assentos iguais se uma puder ser girada para formar a outra.

**C.1-4**

De quantos modos três números distintos podem ser escolhidos no conjunto  $\{1, 2, \dots, 100\}$  de forma que sua soma seja par?

**C.1-5**

Prove a identidade

$$\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1} \quad (\text{C.8})$$

para  $0 < k \leq n$ .

**C.1-6**

Prove a identidade

$$\binom{n}{k} = \frac{n}{n-k} \binom{n-1}{k}$$

para  $0 \leq k < n$ .

**C.1-7**

Para escolher  $k$  objetos de  $n$ , você pode tornar um dos objetos distinto e considerar se o objeto distinto foi escolhido. Use essa abordagem para provar que

$$\binom{n}{k} = \binom{n-1}{k} \binom{n-1}{k-1}.$$

**C.1-8**

Usando o resultado do Exercício C.1-7, faça uma tabela para  $n = 0, 1, \dots, 6$  e  $0 \leq k \leq n$  dos coeficientes binomiais  $\binom{n}{k}$  com  $\binom{0}{0}$  na parte superior,  $\binom{1}{0}$  e  $\binom{1}{1}$  na linha seguinte e assim por diante. Essa tabela de coeficientes binomiais é chamada *triângulo de Pascal*.

**C.1-9**

Prove que

$$\sum_{i=1}^n i = \binom{n+1}{2}.$$

**C.1-10**

Mostre que para qualquer  $n \geq 0$  e  $0 \leq k \leq n$ , o valor máximo de  $\binom{n}{k}$  é alcançado quando  $k = \lfloor n/2 \rfloor$  ou  $k = \lceil n/2 \rceil$ .

**C.1-11 \***

Demonstre que, para qualquer  $n \geq 0, j \geq 0, k \geq 0$  e  $j + k \leq n$ ,

$$\binom{n}{j+k} \leq \binom{n}{j} \binom{n-j}{k}. \quad (\text{C.9})$$

Forneça uma prova algébrica e um argumento baseado em um método para escolher  $j + k$  itens de  $n$  itens. Dê um exemplo no qual a igualdade não seja válida.

### C.1-12 \*

Use a indução sob  $k \leq n/2$  para provar a desigualdade (C.6), e use a equação (C.3) para estendê-la a todo  $k \leq n$ .

### C.1-13 \*

Use a aproximação de Stirling para provar que

$$\binom{2n}{k} = \frac{2^{2n}}{\sqrt{\pi n}} (1 + O(1/n)). \quad (\text{C.10})$$

### C.1-14 \*

Pela diferenciação da função de entropia  $H(\lambda)$ , mostre que ela alcança seu valor máximo em  $\lambda = 1/2$ . O que é  $H(1/2)$ ?

### C.1-15 \*

Mostre que, para qualquer inteiro  $n \geq 0$ ,

$$\sum_{k=0}^n \binom{n}{k} k = n2^{n-1}. \quad (\text{C.11})$$

## C.2 Probabilidade

A probabilidade é uma ferramenta essencial para o projeto e a análise de algoritmos probabilísticos e aleatórios. Esta seção examina a teoria básica da probabilidade.

Definimos probabilidade em termos de um *espaço amostral*  $S$ ; esse espaço é um conjunto cujos elementos são chamados *eventos elementares*. Cada evento elementar pode ser visto como um resultado possível de um experimento. No caso do experimento de lançar duas moedas distintas, podemos considerar o espaço amostral consistindo no conjunto de todas as cadeias possíveis de 2 elementos sobre  $\{H, T\}$ :

$$S = \{HH, HT, TH, TT\}.$$

Um *evento* é um subconjunto<sup>1</sup> do espaço amostral  $S$ . Por exemplo, no experimento de lançar duas moedas, o evento de obter uma cara e uma coroa é  $\{HT, TH\}$ . O evento  $S$  é chamado *evento certo*, e o evento  $\emptyset$  é chamado *evento nulo*. Dizemos que dois eventos  $A$  e  $B$  são **mutuamente exclusivos** se  $A \cap B = \emptyset$ . Algumas vezes tratamos um evento elementar  $s \in S$  como o evento  $\{s\}$ . Por definição, todos os eventos elementares são mutuamente exclusivos.

### Axiomas de probabilidade

Uma *distribuição de probabilidades*  $\Pr\{\cdot\}$  em um espaço amostral  $S$  é um mapeamento de eventos de  $S$  para números reais, tal que os *axiomas de probabilidade* a seguir são satisfeitos:

1.  $\Pr\{A\} \geq 0$  para qualquer evento  $A$ .
2.  $\Pr\{S\} = 1$ .

---

<sup>1</sup> No caso de uma distribuição de probabilidades geral, é possível haver alguns subconjuntos do espaço amostral  $S$  que não são considerados eventos. Essa situação costuma surgir quando o espaço amostral é incontavelmente infinito. O principal requisito é que o conjunto de eventos de um espaço amostral seja fechado sob as operações de obter o complemento de um evento, formar a união de um número finito ou contável de eventos e tomar a interseção de um número finito ou contável de eventos. A maioria das distribuições de probabilidade que veremos são sobre espaços de amostragem finitos ou contáveis, e em geral consideraremos eventos todos os subconjuntos de um espaço amostral.

Uma exceção notável é a distribuição uniforme contínua de probabilidades, que será apresentada em breve.

3.  $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$  para dois eventos mutuamente exclusivos quaisquer  $A$  e  $B$ . De modo mais geral, para qualquer seqüência de eventos (finita ou contavelmente infinita)  $A_1, A_2, \dots$  que sejam mutuamente exclusivos dois a dois,

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\}.$$

Chamamos  $\Pr\{A\}$  a **probabilidade** do evento  $A$ . Observamos nesse caso que o axioma 2 é um requisito de normalização: na realidade, não existe nada de fundamental sobre a escolha de 1 como a probabilidade do evento certo, exceto o fato de ser natural e conveniente.

Diversos resultados decorrem imediatamente desses axiomas e da teoria básica dos conjuntos (ver Seção B.1). O evento nulo  $\emptyset$  tem a probabilidade  $\Pr\{\emptyset\} = 0$ . Se  $A \subseteq B$ , então  $\Pr\{A\} \leq \Pr\{B\}$ . Usando  $\bar{A}$  para denotar o evento  $S - A$  (o **complemento** de  $A$ ), temos  $\Pr\{\bar{A}\} = 1 - \Pr\{A\}$ . Para dois eventos quaisquer  $A$  e  $B$ ,

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} \quad (\text{C.12})$$

$$\leq \Pr\{A\} + \Pr\{B\}. \quad (\text{C.13})$$

Em nosso exemplo de lançamento de moedas, suponha que cada um dos quatro eventos elementares tenha a probabilidade  $1/4$ . Então, a probabilidade de obter pelo menos uma cara é

$$\begin{aligned} \Pr\{\text{HH, HT, TH}\} &= \Pr\{\text{HH}\} + \Pr\{\text{HT}\} + \Pr\{\text{TH}\} \\ &= 3/4. \end{aligned}$$

Uma alternativa, tendo em vista a probabilidade de se obter estritamente menos de uma cara é  $\Pr\{\text{TT}\} = 1/4$ , a probabilidade de se obter pelo menos uma cara é  $1 - 1/4 = 3/4$ .

## Distribuições de probabilidades discretas

Uma distribuição de probabilidades é **discreta** se é definida sobre um espaço amostral finito ou contavelmente infinito. Seja  $S$  o espaço amostral. Então, para qualquer evento  $A$ ,

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\},$$

pois eventos elementares, especificamente aqueles em  $A$ , são mutuamente exclusivos. Se  $S$  é finito e todo evento elementar  $s \in S$  tem probabilidade

$$\Pr\{s\} = 1/|S|,$$

então temos a **distribuição de probabilidades uniforme** em  $S$ . Em tal caso, o experimento é freqüentemente descrito como “extrair um elemento de  $S$  ao acaso”.

Como exemplo, considere o processo de lançar uma **moeda comum**, uma moeda para a qual a probabilidade de obter uma cara é igual à probabilidade de obter uma coroa, ou seja,  $1/2$ . Se lançarmos a moeda  $n$  vezes, teremos a distribuição de probabilidades uniforme definida no espaço amostral  $S = \{H, T\}^n$ , um conjunto de tamanho  $2^n$ . Cada evento elementar em  $S$  pode ser representado como uma cadeia de comprimento  $n$  sobre  $\{H, T\}$ , e cada um ocorre com a probabilidade  $1/2^n$ . O evento

$$A = \{\text{ocorrem exatamente } k \text{ caras e exatamente } n - k \text{ coroas}\}$$

é um subconjunto de  $S$  de tamanho  $|A| = \binom{n}{k}$ , pois existem  $\binom{n}{k}$  cadeias de comprimento  $n$  sobre  $\{H, T\}$  que contêm exatamente  $k$  caras ( $H$ ). A probabilidade do evento  $A$  é portanto  $\Pr\{A\} = \binom{n}{k}/2^n$ .

## Distribuição uniforme contínua de probabilidades

A distribuição uniforme contínua de probabilidades é um exemplo de uma distribuição de probabilidades na qual nem todos os subconjuntos do espaço amostral são considerados eventos. A distribuição uniforme contínua de probabilidades é definida sobre um intervalo fechado  $[a, b]$  dos reais, onde  $a < b$ . Intuitivamente, queremos que cada ponto no intervalo  $[a, b]$  seja “igualmente provável”. Porém, existe um número incontável de pontos; assim, se dermos a todos os pontos a mesma probabilidade finita positiva, não poderemos satisfazer simultaneamente aos axiomas 2 e 3. Por essa razão, gostaríamos de associar uma probabilidade apenas a *alguns* dos subconjuntos de  $S$ , de tal modo que os axiomas fossem satisfeitos para esses eventos.

Para qualquer intervalo fechado  $[c, d]$ , onde  $a \leq c \leq d \leq b$ , a *distribuição uniforme contínua de probabilidades* define a probabilidade do evento  $[c, d]$  como

$$\Pr\{[c, d]\} = \frac{d - c}{b - a}.$$

Observe que, para qualquer ponto  $x = [x, x]$  a probabilidade de  $x$  é 0. Se removermos os pontos extremos de um intervalo  $[c, d]$ , obteremos o intervalo aberto  $(c, d)$ . Tendo em vista que  $[c, d] = [c, c] \cup (c, d) \cup [d, d]$ , o axioma 3 nos dá  $\Pr\{[c, d]\} = \Pr\{(c, d)\}$ . Em geral, o conjunto de eventos para a distribuição uniforme contínua de probabilidades é qualquer subconjunto do espaço amostral  $[a, b]$  que possa ser obtido por uma união finita ou contável de intervalos abertos e fechados.

## Probabilidade condicional e independência

Às vezes, temos algum conhecimento parcial antecipado sobre o resultado de um experimento. Por exemplo, suponha que um amigo tenha lançado duas moedas comuns e tenha dito a você que pelo menos uma das moedas mostrou uma cara. Qual é a probabilidade de ambas as moedas serem caras? A informação dada elimina a possibilidade de duas coroas. Os três eventos elementares restantes são igualmente prováveis; assim, deduzimos que cada um ocorre com a probabilidade  $1/3$ . Tendo em vista que apenas um desses eventos elementares mostra duas caras, a resposta à nossa pergunta é  $1/3$ .

A probabilidade condicional formaliza a noção de se ter um conhecimento parcial antecipado do resultado de um experimento. A *probabilidade condicional* de um evento  $A$  dada a ocorrência de outro evento  $B$  é definida como

$$\Pr\{A | B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \quad (\text{C.14})$$

sempre que  $\Pr\{B\} \neq 0$ . (Lê-se “ $\Pr\{A | B\}$ ” como “a probabilidade de  $A$  dado  $B$ ”.) Intuitivamente, sabendo-se que o evento  $B$  ocorre, a probabilidade de que o evento  $A$  também ocorra é  $A \cap B$ . Ou seja,  $A \cap B$  é o conjunto de resultados em que ocorrem  $A$  e  $B$ . Tendo em vista que o resultado é um dos eventos elementares em  $B$ , normalizamos as probabilidades de todos os eventos elementares em  $B$  dividindo-as por  $\Pr\{B\}$ , de tal forma que sua soma seja 1. Por conseguinte, a probabilidade condicional de  $A$  dado  $B$  é a razão entre a probabilidade do evento  $A \cap B$  e a probabilidade do evento  $B$ . No exemplo anterior,  $A$  é o evento em que ambas as moedas são caras, e  $B$  é o evento em que pelo menos uma moeda é cara. Desse modo,  $\Pr\{A | B\} = (1/4)/(3/4) = 1/3$ .

Dois eventos são *independentes* se

$$\Pr\{A \cap B\} = \Pr\{A\}\Pr\{B\}, \quad (\text{C.15})$$

que é equivalente, se  $\Pr\{B\} \neq 0$ , à condição

Por exemplo, suponha que duas moedas comuns sejam lançadas e que os resultados sejam independentes. Então, a probabilidade de duas caras é  $(1/2)(1/2) = 1/4$ . Agora, suponha que um evento seja que a primeira moeda mostre o lado cara e o outro evento seja que as moedas caiam mostrando lados diferentes. Cada um desses eventos ocorre com probabilidade  $1/2$ , e a probabilidade de que ambos os eventos ocorram é  $1/4$ ; assim, de acordo com a definição de independência, os eventos são independentes – ainda que se possa imaginar que ambos os eventos dependem da primeira moeda. Finalmente, suponha que as moedas sejam soldadas de tal forma que ambas mostrem cara ou ambas mostrem coroa, e que as duas possibilidades sejam igualmente prováveis. Então, a probabilidade de que cada moeda mostre cara é  $1/2$ , mas a probabilidade de que ambas mostrem cara é  $1/2 \neq (1/2)(1/2)$ . Em consequência disso, o evento em que uma das moedas mostra cara e o evento em que a outra mostra cara não são independentes.

Em uma coleção de eventos  $A_1, A_2, \dots, A_n$ , os eventos são ditos **independentes dois a dois** se

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\}$$

para todo  $1 \leq i < j \leq n$ . Dizemos que os eventos da coleção são (*mutuamente*) **independentes** se todo subconjunto de  $k$  elementos [ver símbolo] da coleção, onde  $2 \leq k \leq n$  e  $1 \leq i_1 < i_2 < \dots < i_k \leq n$ , satisfaz a

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \dots \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\} \dots \Pr\{A_{i_k}\}.$$

Por exemplo, suponha que sejam lançadas duas moedas comuns. Seja  $A_1$  o evento em que a primeira moeda é cara, seja  $A_2$  o evento em que a segunda moeda é cara e seja  $A_3$  o evento em que as duas moedas mostram faces diferentes. Temos

$$\Pr\{A_1\} = 1/2,$$

$$\Pr\{A_2\} = 1/2,$$

$$\Pr\{A_3\} = 1/2,$$

$$\Pr\{A_1 \cap A_2\} = 1/4,$$

$$\Pr\{A_1 \cap A_3\} = 1/4,$$

$$\Pr\{A_2 \cap A_3\} = 1/4,$$

$$\Pr\{A_1 \cap A_2 \cap A_3\} = 0.$$

Considerando-se que, para  $1 \leq i < j \leq 3$ , temos  $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$ , os eventos  $A_1, A_2$  e  $A_3$  são independentes dois a dois. Contudo, os eventos não são mutuamente independentes, porque  $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$  e  $\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\} = 1/8 \neq 0$ .

## Teorema de Bayes

Da definição de probabilidade condicional (C.14) e da lei comutativa  $A \cap B = B \cap A$ , segue-se que, para dois eventos  $A$  e  $B$ , cada um com probabilidade diferente não nula,

$$\Pr\{A \cap B\} = \Pr\{B\}\Pr\{A | B\} \tag{C.16}$$

$$= \Pr\{A\}\Pr\{B | A\}.$$

Resolvendo para  $\Pr\{A | B\}$ , obtemos

$$\Pr\{A | B\} = \frac{\Pr\{A\} \Pr\{B | A\}}{\Pr\{B\}}, \quad (\text{C.17})$$

que se conhece como **teorema de Bayes**. O denominador  $\Pr\{B\}$  é uma constante de normalização que podemos expressar novamente como a seguir. Considerando-se que  $B = (B \cap A) \cup (B \cap \bar{A})$  e que  $B \cap A$  e  $B \cap \bar{A}$  são eventos mutuamente exclusivos,

$$\begin{aligned}\Pr\{B\} &= \Pr\{B \cap A\} + \Pr\{B \cap \bar{A}\} \\ &= \Pr\{A\} \Pr\{B | A\} + \Pr\{\bar{A}\} \Pr\{B | \bar{A}\}.\end{aligned}$$

Substituindo na equação (C.17), obtemos uma forma equivalente do teorema de Bayes:

$$\Pr\{A | B\} = \frac{\Pr\{A\} \Pr\{B | A\}}{\Pr\{A\} \Pr\{B | A\} + \Pr\{\bar{A}\} \Pr\{B | \bar{A}\}}.$$

O teorema de Bayes pode simplificar o cálculo de probabilidades condicionais. Por exemplo, suponha que temos uma moeda comum e uma moeda viciada que sempre mostra a face cara. Vamos executar um experimento que consiste em três eventos independentes: uma das duas moedas é escolhida ao acaso, a moeda é lançada uma vez, e então é novamente lançada. Suponha que a moeda escolhida mostre a face cara ambas as vezes. Qual é a probabilidade de que ela seja a moeda viciada?

Resolvemos esse problema usando o teorema de Bayes. Seja  $A$  o evento em que a moeda viciada é escolhida, e seja  $B$  o evento em que a moeda mostra a face cara ambas as vezes. Desejamos determinar  $\Pr\{A | B\}$ . Temos  $\Pr\{A\} = 1/2$ ,  $\Pr\{B | A\} = 1$ ,  $\Pr\{\bar{A}\} = 1/2$  e  $\Pr\{B | \bar{A}\} = 1/4$ ; consequentemente,

$$\begin{aligned}\Pr\{A | B\} &= \frac{(1/2) \cdot 1}{(1/2) \cdot 1 + (1/2) \cdot (1/4)} \\ &= 4/5.\end{aligned}$$

## Exercícios

### C.2-1

Prove a **desigualdade de Boole**: para qualquer seqüência finita ou contavelmente infinita de eventos  $A_1, A_2, \dots$ ,

$$\Pr\{A_1 \cup A_2 \cup \dots\} \leq \Pr\{A_1\} + \Pr\{A_2\} + \dots. \quad (\text{C.18})$$

### C.2-2

O professor Rosencrantz lança uma moeda comum uma vez. O professor Guildenstern lança uma moeda comum duas vezes. Qual é a probabilidade de que o professor Rosencrantz obtenha mais caras que o professor Guildenstern?

### C.2-3

Um baralho de 10 cartas, cada uma representando um número distinto de 1 até 10, é embaralhado de modo que as cartas se misturem completamente. Três cartas são removidas do baralho, uma de cada vez. Qual é a probabilidade de que as três cartas sejam selecionadas em seqüência ordenada (crescente)?

### C.2-4 \*

Descreva um procedimento que utilize como entrada dois inteiros  $a$  e  $b$  tais que  $0 < a < b$  e, usando lançamentos de moedas comuns, produza como saída resultados cara com probabilidade  $a/b$  e coroa com probabilidade  $(b-a)/b$ . Forneça um limite sobre o número esperado de lançamentos de moedas, que deve ser  $O(1)$ . (Sugestão: Represente  $a/b$  em binário.)

### C.2-5

Prove que

$$\Pr\{A \mid B\} + \Pr\{\bar{A} \mid B\} = 1.$$

### C.2-6

Prove que, para qualquer coleção de eventos  $A_1, A_2, \dots, A_n$ ,

$$\Pr\{A_1 \cap A_2 \cap \dots \cap A_n\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \dots$$

$$\Pr\{A_n \mid A_1 \cap A_2 \cap \dots \cap A_{n-1}\}.$$

### C.2-7 \*

Mostre como elaborar um conjunto de  $n$  eventos que sejam independentes dois a dois, mas tais que, nenhum subconjunto de  $k > 2$  elementos desses eventos seja mutuamente independente.

### C.2-8 \*

Dois eventos  $A$  e  $B$  são **condicionalmente independentes**, dado  $C$ , se

$$\Pr\{A \cap B \mid C\} = \Pr\{A \mid C\} \cdot \Pr\{B \mid C\}.$$

Dê um exemplo simples mas não trivial de dois eventos que não sejam independentes, mas que sejam condicionalmente independentes dado um terceiro evento.

### C.2-9 \*

Você é concorrente em um programa de competição em que um prêmio está escondido atrás de uma entre três cortinas. Você ganhará o prêmio se selecionar a cortina correta. Depois de escolher uma cortina mas antes da cortina ser erguida, o apresentador ergue uma das outras cortinas, revelando um cenário vazio, e pergunta se você gostaria de trocar sua seleção atual pela cortina restante. De que modo suas chances mudarão se você trocar a seleção?

### C.2-10 \*

Um carcereiro escolheu um prisioneiro ao acaso entre três condenados à morte, a fim de libertá-lo. Os outros dois serão executados. O guarda sabe qual deles será libertado, mas é proibido de dar a qualquer prisioneiro informações relativas a seu status. Vamos chamar os prisioneiros  $X, Y$  e  $Z$ . O prisioneiro  $X$  pergunta reservadamente ao guarda qual dos outros prisioneiros,  $Y$  ou  $Z$ , será executado, argumentando que, tendo em vista que já sabe que pelo menos um deles deve morrer, o guarda não estará revelando quaisquer informações sobre seu próprio status. O guarda diz a  $X$  que  $Y$  deverá ser executado. O prisioneiro  $X$  se sente mais feliz agora, pois chegou à conclusão de que ele ou o prisioneiro  $Z$  será libertado, o que significa que sua probabilidade de ficar livre é agora de  $1/2$ . Ele está certo, ou sua chance de viver ainda é de  $1/3$ ? Explique.

## C.3 Variáveis aleatórias discretas

Uma **variável aleatória (discreta)**  $X$  é uma função de um espaço amostral finito ou contavelmente infinito  $S$  para os números reais. Ela associa um número real a cada resultado possível de um experimento, o que nos permite trabalhar com a distribuição de probabilidades induzida no

conjunto de números resultante. As variáveis aleatórias também podem ser definidas para espaços amostrais incontavelmente infinitos, mas elas levantam questões técnicas, as quais não há necessidade de abordar para nossos propósitos. Daqui em diante, partiremos do princípio de que variáveis aleatórias são discretas.

Para uma variável aleatória  $X$  e um número real  $x$ , definimos o evento  $X = x$  como  $\{s \in S : X(s) = x\}$ ; portanto,

$$\Pr\{X = x\} = \sum_{\{s \in S : X(s) = x\}} \Pr\{s\}.$$

A função

$$f(x) = \Pr\{X = x\}$$

é a **função de densidade de probabilidades** da variável aleatória  $X$ . Dos axiomas de probabilidade, concluímos que  $\Pr\{X = x\} \geq 0$  e  $\sum_x \Pr\{X = x\} = 1$ .

Como exemplo, considere o experimento de rolar um par de dados comuns de 6 faces. Existem 36 eventos elementares possíveis no espaço amostral. Supomos que a distribuição de probabilidades é uniforme, de forma que cada evento elementar  $s \in S$  é igualmente provável:  $\Pr\{s\} = 1/36$ . Defina a variável aleatória  $X$  como o *máximo* dos dois valores mostrados nas faces superiores dos dados. Temos  $\Pr\{X = 3\} = 5/36$ , pois  $X$  atribui o valor 3 a 5 dos 36 eventos elementares possíveis, isto é,  $(1, 3)$ ,  $(2, 3)$ ,  $(3, 3)$ ,  $(3, 2)$  e  $(3, 1)$ .

É comum a definição de diversas variáveis aleatórias no mesmo espaço amostral. Se  $X$  e  $Y$  são variáveis aleatórias, a função

$$f(x, y) = \Pr\{X = x \text{ e } Y = y\}$$

é a **função de densidade de probabilidades conjunta** de  $X$  e  $Y$ . Para um valor fixo  $y$ ,

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ e } Y = y\},$$

e, de modo semelhante, para um valor fixo  $x$ ,

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ e } Y = y\}.$$

Usando a definição (C.14) de probabilidade condicional, temos

$$\Pr\{X = x \text{ e } Y = y\} = \frac{\Pr\{X = x \text{ e } Y = y\}}{\Pr\{Y = y\}}.$$

Definimos duas variáveis aleatórias  $X$  e  $Y$  como **independentes** se, para todo  $x$  e  $y$ , os eventos  $X = x$  e  $Y = y$  são independentes ou, de modo equivalente, se para todo  $x$  e  $y$ , temos  $\Pr\{X = x \text{ e } Y = y\} = \Pr\{X = x\}\Pr\{Y = y\}$ .

Dado um conjunto de variáveis aleatórias definidas sobre o mesmo espaço amostral, é possível definir novas variáveis aleatórias como somas, produtos ou outras funções das variáveis originais.

## Valor esperado de uma variável aleatória

O resumo mais simples e mais útil da distribuição de uma variável aleatória é a “média” dos valores que ela assume. O **valor esperado** (ou, como sinônimo, **expectativa** ou **média**) de uma variável aleatória discreta  $X$  é

$$E[X] = \sum_x x \Pr\{X = x\} , \quad (\text{C.19})$$

que é bem definido se a soma é finita ou absolutamente convergente. Às vezes, a expectativa de  $X$  é denotada por  $\mu_x$  ou, quando a variável aleatória é aparente a partir do contexto, simplesmente por  $\mu$ .

Considere um jogo em que você lança duas moedas comuns. Você ganha R\$ 3,00 para cada cara, mas perde R\$ 2,00 para cada coroa. O valor esperado da variável aleatória  $X$  que representa seus ganhos é

$$\begin{aligned} E[X] &= 6 \cdot \Pr\{2 \text{ H's}\} + 1 \cdot \Pr\{1 \text{ H}, 1 \text{ T}\} - 4 \cdot \Pr\{2 \text{ T's}\} \\ &= 6(1/4) + 1(1/2) - 4(1/4) \\ &= 1 . \end{aligned}$$

A expectativa da soma de duas variáveis aleatórias é a soma de suas expectativas, ou seja,

$$E[X + Y] = E[X] + E[Y] , \quad (\text{C.20})$$

sempre que  $E[X]$  e  $E[Y]$  são definidos. Denominamos essa propriedade **linearidade de expectativa**, e ela é válida até mesmo se  $X$  e  $Y$  não são independentes. Ela também se estende a somatórios de expectativas finitos e absolutamente convergentes. A linearidade de expectativa é a propriedade fundamental que nos permite executar análises probabilísticas utilizando variáveis indicadoras aleatórias (consulte a Seção 5.2).

Se  $X$  é qualquer variável aleatória, qualquer função  $g(x)$  define uma nova variável aleatória  $g(X)$ . Se a expectativa de  $g(X)$  é definida, então

$$E[g(X)] = \sum_x g(x) \Pr\{X = x\} .$$

Fazendo  $g(x) = \alpha x$  temos, para qualquer constante  $\alpha$ ,

$$E[\alpha X] = \alpha E[X] . \quad (\text{C.21})$$

Conseqüentemente, expectativas são lineares: para duas variáveis aleatórias quaisquer  $X$  e  $Y$  e uma constante qualquer  $\alpha$ ,

$$E[\alpha X + Y] = \alpha E[X] + E[Y] . \quad (\text{C.22})$$

Quando duas variáveis aleatórias  $X$  e  $Y$  são independentes e cada uma tem uma expectativa definida,

$$\begin{aligned} E[XY] &= \sum_x \sum_y xy \Pr\{X = x \text{ e } Y = y\} \\ &= \sum_x \sum_y xy \Pr\{X = x\} \Pr\{Y = y\} \\ &= \left( \sum_x x \Pr\{X = x\} \right) \left( \sum_y y \Pr\{Y = y\} \right) \\ &= E[X] E[Y] \end{aligned}$$

Em geral, quando  $n$  variáveis aleatórias  $X_1, X_2, \dots, X_n$  são mutuamente independentes,

$$E[X_1, X_2, \dots, X_n] = E[X_1] E[X_2] \dots E[X_n]. \quad (\text{C.23})$$

Quando uma variável aleatória  $X$  assume valores pertencentes ao conjunto dos números naturais  $\mathbf{N} = \{0, 1, 2, \dots\}$ , existe uma fórmula elegante para representar sua expectativa:

$$\begin{aligned} E[X] &= \sum_{i=0}^{\infty} i \Pr\{X = i\} \\ &= \sum_{i=0}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i + 1\}) \\ &= \sum_{i=0}^{\infty} \Pr\{X \geq i\}, \end{aligned} \quad (\text{C.24})$$

pois cada termo  $\Pr\{X \geq i\}$  é adicionado em  $i$  vezes e subtraído em  $i - 1$  vezes (exceto  $\Pr\{X \geq 0\}$ , que é adicionado em 0 vezes e não é subtraído de modo algum).

Quando aplicamos uma função convexa  $f(x)$  a uma variável aleatória  $X$ , a *desigualdade de Jensen* nos dá

$$E[f(X)] \geq f(E[X]), \quad (\text{C.25})$$

Desde que as expectativas existam e sejam finitas. (Uma função  $f(x)$  é *convexa* se para todo  $x$  e  $y$ , e para todo  $0 \leq \lambda \leq 1$ , temos  $f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$ .)

## Variância e desvio padrão

O valor esperado de uma variável aleatória não nos informa como os valores da variável estão “espalhados”. Por exemplo, se temos variáveis aleatórias  $X$  e  $Y$  para as quais  $\Pr\{X = 1/4\} = \Pr\{X = 3/4\} = 1/2$  e  $\Pr\{Y = 0\} = \Pr\{Y = 1\} = 1/2$ , então tanto  $E\{X\}$  quanto  $E\{Y\}$  são 1/2, ainda que os valores reais assumidos por  $Y$  estejam mais distantes da média que os valores reais assumidos por  $X$ .

A noção de variância expressa matematicamente o quanto os valores de uma variável aleatória devem provavelmente estar afastados da média. A *variância* de uma variável aleatória  $X$  com expectativa  $E[X]$  é

$$\begin{aligned} \text{Var}[X] &= E[(X - E[X])^2] \\ &= E[X^2 - 2XE[X] + E^2[X]] \\ &= E[X^2] - 2E[XE[X]] + E^2[X] \\ &= E[X^2] - 2E^2[X] + E^2[X] \\ &= E[X^2] - E^2[X]. \end{aligned} \quad (\text{C.26})$$

A justificativa para as igualdades  $E[X^2 - E[X]^2]$  e  $E^2[X] - E^2[X]$  é que  $E[XE[X]] = E^2[X]$  não é uma variável aleatória, mas simplesmente um número real, o que significa que a equação (C.21) se aplica (com  $a = E[X]$ ). A equação (C.26) pode ser reescrita, a fim de se obter uma expressão para a expectativa do quadrado de uma variável aleatória:

$$E[X^2] = \text{Var}[X] + E^2[X] \quad (\text{C.27})$$

A variância de uma variável aleatória  $X$  e a variância de  $\alpha X$  estão relacionadas entre si (veja o Exercício C.3-10):

$$\text{Var}[\alpha X] = \alpha^2 \text{Var}[X].$$

Quando  $X$  e  $Y$  são variáveis aleatórias independentes,

$$\text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y].$$

Em geral, se  $n$  variáveis aleatórias  $X_1, X_2, \dots, X_n$  são independentes duas a duas, então

$$\text{Var}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \text{Var}[X_i]. \quad (\text{C.28})$$

O **desvio padrão** de uma variável aleatória  $X$  é a raiz quadrada positiva da variância de  $X$ . O desvio padrão de uma variável aleatória  $X$  é denotado às vezes por  $\sigma_X$ , ou simplesmente  $\sigma$ , quando a variável aleatória  $X$  é entendida a partir do contexto. Com essa notação, a variância de  $X$  é denotada por  $\sigma^2$ .

## Exercícios

### C.3-1

Dois dados comuns de 6 faces são rolados. Qual é a expectativa da soma dos dois valores exibidos nas faces superiores dos dados? Qual é a expectativa do máximo dos dois valores exibidos?

### C.3-2

Um arranjo  $A[1..n]$  contém  $n$  números distintos que estão ordenados aleatoriamente, com cada permutação dos  $n$  números sendo igualmente provável. Qual é a expectativa do índice do elemento máximo no arranjo? Qual é a expectativa do índice do elemento mínimo no arranjo?

### C.3-3

Um certo jogo consiste em três dados colocados em um recipiente. Um jogador pode apostar R\$ 1,00 em qualquer dos números de 1 a 6. Os dados são rolados, e o pagamento é feito como descrevemos a seguir. Se o número escolhido pelo jogador não aparecer em nenhum dos dados, ele perde seu dinheiro. Caso contrário, se seu número aparecer em exatamente  $k$  dos três dados, para  $k = 1, 2, 3$ , o jogador mantém seu dinheiro e ganha  $k$  vezes o valor apostado. Qual é o ganho esperado quando se aposta nesse jogo uma vez?

### C.3-4

Demonstre que, se  $X$  e  $Y$  são variáveis aleatórias não negativas, então

$$E[\max(X, Y)] \leq E[X] + E[Y].$$

### C.3-5 \*

Sejam  $X$  e  $Y$  variáveis aleatórias independentes. Prove que  $f(X)$  e  $g(Y)$  são independentes para qualquer escolha de funções  $f$  e  $g$ .

### C.3-6 \*

Seja  $X$  uma variável aleatória não negativa, e suponha que  $E[X]$  esteja bem definida. Prove a **desigualdade de Markov**:

$$\Pr\{X \geq t\} \leq E[X]/t \quad (\text{C.29})$$

para todo  $t > 0$ .

### C.3-7 \*

Seja  $S$  um espaço amostral, e sejam  $X$  e  $X'$  variáveis aleatórias tais que  $X(s) \geq X'(s)$  para todo  $s \in S$ . Prove que, para qualquer constante real  $t$ ,

$$\Pr\{X \geq t\} \geq \Pr[X' \geq t]$$

### C.3-8

O que é maior: a expectativa do quadrado de uma variável aleatória ou o quadrado de sua expectativa?

### C.3-9

Mostre que, para qualquer variável aleatória  $X$  que admita apenas os valores 0 e 1, temos  $\text{Var}[X] = E[X]E[1 - X]$ .

### C.3-10

Prove que  $\text{Var}[\alpha X] = \alpha^2 \text{Var}[X]$ , a partir da definição (C.26) de variância.

## C.4 As distribuições geométrica e binomial

Uma moeda lançada é uma instância de uma *experiência de Bernoulli*, que é definida como um experimento com apenas dois resultados possíveis: *sucesso*, que ocorre com a probabilidade  $p$ , e *insucesso*, que ocorre com a probabilidade  $q = 1 - p$ . Quando falamos coletivamente de *experiências de Bernoulli*, queremos dizer que as experiências (ou tentativas) são mutuamente independentes e, a menos que seja estabelecido especificamente o contrário, cada uma delas tem a mesma probabilidade  $p$  de sucesso. Duas distribuições importantes surgem das experiências de Bernoulli: a distribuição geométrica e a distribuição binomial.

### A distribuição geométrica

Suponha que temos uma sequência de experiências de Bernoulli, cada uma com a probabilidade de sucesso  $p$  e com uma probabilidade de insucesso  $q = 1 - p$ . Quantas experiências ocorrem antes de se obter um sucesso? Seja a variável aleatória  $X$  o número de experiências necessárias para se obter um sucesso. Então,  $X$  tem valores no intervalo  $\{1, 2, \dots\}$  e, para  $k \geq 1$ ,

$$\Pr\{X = k\} = q^{k-1} p, \quad (\text{C.30})$$

pois temos  $k - 1$  insucessos antes do único sucesso. Uma distribuição de probabilidades que satisfa z à equação (C.30) é dita uma *distribuição geométrica*. A Figura C.1 ilustra tal distribuição.

Supondo-se  $p < 1$ , a expectativa de uma distribuição geométrica pode ser calculada com o uso da identidade (A.8):

$$\begin{aligned} E[X] &= \sum_{k=1}^{\infty} k q^{k-1} p \\ &= \frac{p}{q} \sum_{k=0}^{\infty} k q^k \\ &= \frac{p}{q} \cdot \frac{q}{(1-q)^2} \\ &= 1/p. \end{aligned} \quad (\text{C.31})$$

Desse modo, em média, ocorrem  $1/p$  tentativas antes de se obter um sucesso, um resultado intuitivo. A variância, que pode ser calculada de modo semelhante, mas usando-se o Exercício A.1-3, é

$$878 \mid \text{Var}[X] = q/p^2. \quad (\text{C.32})$$

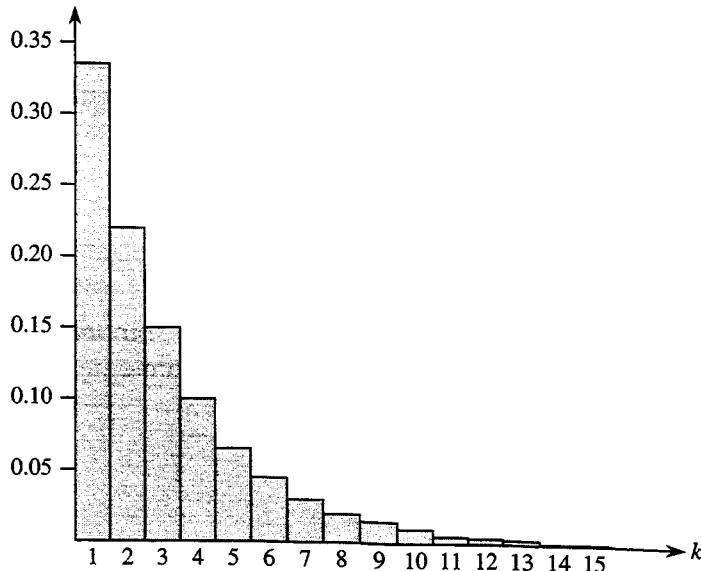


FIGURA C.1 Uma distribuição geométrica com probabilidade de sucesso  $p = 1/3$  e uma probabilidade de insucesso  $q = 1 - p$ . A expectativa da distribuição é  $1/p = 3$

Como exemplo, suponha que dois dados sejam rolados repetidamente até se obter o resultado sete ou onze. Dos 36 resultados possíveis, 6 produzem um sete e 2 produzem um onze. Desse modo, a probabilidade de sucesso é  $p = 8/36 = 2/9$ , e devemos rolar os dados  $1/p = 9/2 = 4,5$  vezes em média para obter um valor sete ou onze.

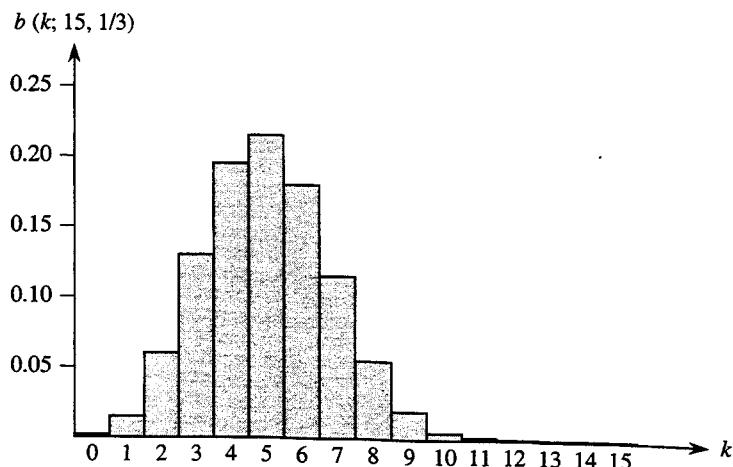


FIGURA C.2 A distribuição binomial de  $b(k; 15, 1/3)$  resultante de  $n = 15$  experiências de Bernoulli, cada uma com probabilidade de sucesso  $p = 1/3$ . A expectativa da distribuição é  $np = 5$

### A distribuição binomial

Quantos sucessos ocorrem durante  $n$  experiências de Bernoulli, onde um sucesso ocorre com probabilidade  $p$  e um insucesso com probabilidade  $q = 1 - p$ ? Defina a variável aleatória  $X$  como o número de sucessos em  $n$  experiências. Então,  $X$  tem valores no intervalo  $\{0, 1, \dots, n\}$  e, para  $k = 0, \dots, n$ ,

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}, \quad (\text{C.33})$$

pois existem  $\binom{n}{k}$  modos de escolher quais  $k$  das  $n$  experiências são sucessos, e a probabilidade de ocorrer cada uma é  $p^k q^{n-k}$ . Uma distribuição de probabilidades que satisfaz à equação (C.33) é chamada **distribuição binomial**. Por conveniência, definimos a família de distribuições binomiais usando a notação

$$b(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k}. \quad (\text{C.34})$$

A Figura C.2 ilustra uma distribuição binomial. O nome “binomial” vem do fato de que (C.33) é o  $k$ -ésimo termo da expansão de  $(p + q)^n$ . Conseqüentemente, tendo em vista que  $p + q = 1$ ,

$$\sum_{k=0}^n b(k; n, p) = 1, \quad (\text{C.35})$$

como é exigido pelo axioma 2 dos axiomas de probabilidade.

Podemos calcular a expectativa de uma variável aleatória que tem uma distribuição binomial a partir das equações (C.8) e (C.35). Seja  $X$  uma variável aleatória que segue a distribuição binomial  $b(k; n, p)$ , e seja  $q = 1 - p$ . Pela definição de expectativa, temos

$$\begin{aligned} E[X] &= \sum_{k=0}^n k \Pr\{X = k\} \\ &= \sum_{k=0}^n k b(k; n, p) \\ &= \sum_{k=1}^n k \binom{n}{k} p^k q^{n-k} \\ &= np \sum_{k=1}^n \binom{n-1}{k-1} p^{k-1} q^{n-k} \\ &= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \\ &= np \sum_{k=0}^{n-1} b(k; n-1, p) \\ &= np. \end{aligned} \quad (\text{C.36})$$

Usando a linearidade de expectativa, podemos obter o mesmo resultado com uma quantidade de cálculos algébricos substancialmente menor. Seja  $X_i$  a variável aleatória que descreve o número de sucessos na  $i$ -ésima experiência. Então,  $E[X_i] = p \cdot 1 + q \cdot 0 = p$  e, por linearidade de expectativa (equação (C.20)), o número esperado de sucessos para  $n$  experiências (tentativas) é

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^n X_i\right] \\ &= \sum_{i=1}^n E[X_i] \\ &= \sum_{i=1}^n p \\ &= np. \end{aligned} \quad (\text{C.37})$$

A mesma abordagem pode ser usada para calcular a variância da distribuição. Empregando a equação (C.26), temos  $\text{Var}[X_i] = E[X_i^2] - E^2[X_i]$ . Tendo em vista que  $X_i$  assume apenas os valores 0 e 1, temos  $E[X_i^2] = E[X_i] = p$  e, consequentemente,

$$\text{Var}[X_i] = p - p^2 = pq . \quad (\text{C.38})$$

Para calcular a variância de  $X$ , aproveitamo-nos da independência das  $n$  experiências; desse modo, pela equação (C.28),

$$\begin{aligned} \text{Var}[X] &= \text{Var}\left[\sum_{i=1}^n X_i\right] \\ &= \sum_{i=1}^n \text{Var}[X_i] \\ &= \sum_{i=1}^n pq \\ &= npq . \end{aligned} \quad (\text{C.39})$$

Como podemos ver na Figura C.2, a distribuição binomial  $b(k; n, p)$  aumenta à medida que  $k$  varia de 0 a  $n$  até alcançar a média  $np$ , e então diminui. É possível provar que a distribuição sempre se comporta dessa maneira, examinando-se a razão entre termos consecutivos:

$$\begin{aligned} \frac{b(k; n, p)}{b(k-1; n, p)} &= \frac{\binom{n}{k} p^k q^{n-k}}{\binom{n}{k-1} p^{k-1} q^{n-k+1}} \\ &= \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \\ &= \frac{(n-k+1)p}{kq} \\ &= 1 + \frac{(n+1)p - k}{kq} . \end{aligned} \quad (\text{C.40})$$

Essa razão é maior que 1 exatamente quando  $(n+1)p - k$  é positiva. Conseqüentemente,  $b(k; n, p) > b(k-1; n, p)$  para  $k < (n+1)p$  (a distribuição aumenta), e  $b(k; n, p) < b(k-1; n, p)$  para  $k > (n+1)p$  (a distribuição diminui). Se  $k = (n+1)p$  é um inteiro, então  $b(k; n, p) = b(k-1; n, p)$ , e assim a distribuição tem dois máximos: em  $k = (n+1)p$  e em  $k-1 = (n+1)p-1 = np - q$ . Caso contrário, ela atinge um máximo no inteiro único  $k$  que reside no intervalo  $np - q < k < (n+1)p$ .

O lema a seguir fornece um limite superior sobre a distribuição binomial.

### Lema C.1

Seja  $n \geq 0$ , seja  $0 < p < 1$ , seja  $q = 1 - p$  e seja  $0 \leq k \leq n$ . Então,

$$b(k; n, p) \leq \left(\frac{np}{k}\right)^k \left(\frac{np}{n-k}\right)^{n-k} .$$

**Prova** Usando a equação (C.6), temos

$$\begin{aligned} b(k; n, p) &= \binom{n}{k} p^k q^{n-k} \\ &\leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \\ &\leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}. \end{aligned}$$

■

## Exercícios

### C.4-1

Verifique o axioma 2 dos axiomas de probabilidade para a distribuição geométrica.

### C.4-2

Quantas vezes em média devemos lançar 6 moedas comuns antes de obtermos 3 caras e 3 coroas?

### C.4-3

Mostre que  $b(k; n, p) = b(n - k; n, q)$ , onde  $q = 1 - p$ .

### C.4-4

Mostre que o valor máximo da distribuição binomial  $b(k; n, p)$  é aproximadamente  $1/\sqrt{2\pi npq}$ , onde  $q = 1 - p$ .

### C.4-5 \*

Mostre que a probabilidade de nenhum sucesso em  $n$  experiências de Bernoulli, cada uma com a probabilidade  $p = 1/n$ , é aproximadamente  $1/e$ . Mostre que a probabilidade de exatamente um sucesso também é aproximadamente  $1/e$ .

### C.4-6 \*

O professor Rosencrantz lança uma moeda comum  $n$  vezes, e o professor Guildenstern faz o mesmo. Mostre que a probabilidade de eles conseguirem o mesmo número de caras é  $\binom{2n}{n} / 4^n$ . (Sugestão: Para o professor Rosencrantz, considere uma cara um sucesso; para o professor Guildenstern, considere uma coroa um sucesso.) Use seu argumento para verificar a identidade

$$\sum_{k=0}^n \binom{n}{k}^2 = \binom{2n}{n}.$$

### C.4-7 \*

Mostre que, para  $0 \leq k \leq n$ ,

$$b(k; n, 1/2) \leq 2^{nH(k/n) - n},$$

onde  $H(x)$  é a função de entropia (C.7).

### C.4-8 \*

Considere  $n$  experiências de Bernoulli onde, para  $i = 1, 2, \dots, n$ , a  $i$ -ésima experiência tem uma probabilidade de sucesso  $p_i$ , e seja  $X$  a variável aleatória que denota o número total de sucessos. Seja  $p \geq p_i$  para todo  $i = 1, 2, \dots, n$ . Prove que, para  $1 \leq k \leq n$ ,

$$882 \quad \Pr\{X < k\} \leq \sum_{i=0}^{k-1} b(i; n, p).$$

#### C.4-9 \*

Seja  $X$  a variável aleatória correspondente ao número total de sucessos em um conjunto  $A$  de  $n$  experiências de Bernoulli, onde a  $i$ -ésima experiência tem uma probabilidade de sucesso  $p_i$ , e seja  $X'$  a variável aleatória que corresponde ao número total de sucessos em um segundo conjunto  $A'$  de  $n$  experiências de Bernoulli, onde a  $i$ -ésima experiência tem uma probabilidade de sucesso  $p'_i \geq p_i$ . Prove que, para  $0 \leq k \leq n$ ,

$$\Pr\{X' \geq k\} \geq \Pr\{X \geq k\}.$$

(*Sugestão:* Mostre como obter as experiências de Bernoulli em  $A'$  por meio de um experimento envolvendo as experiências de  $A$ , e use o resultado do Exercício C.3-7.)

## ★ C.5 As extremidades da distribuição binomial

A probabilidade de haver no mínimo, ou no máximo,  $k$  sucessos em  $n$  experiências de Bernoulli, cada uma com probabilidade de sucesso  $p$ , é freqüentemente de maior interesse que a probabilidade de haver exatamente  $k$  sucessos. Nesta seção, investigaremos as **extremidades** da distribuição binomial: as duas regiões da distribuição  $b(k; n, p)$  que estão longe do  $np$  médio. Demonstraremos vários limites importantes em uma (soma de todos os termos em uma) extremidade.

Primeiro, forneceremos um limite sobre a extremidade direita da distribuição  $b(k; n, p)$ . Limites sobre a extremidade esquerda podem ser determinados, invertendo-se os papéis de sucessos e insucessos.

### **Teorema C.2**

Considere uma seqüência de  $n$  experiências de Bernoulli, onde o sucesso ocorre com probabilidade  $p$ . Seja  $X$  a variável aleatória que denota o número total de sucessos. Então, para  $0 \leq k \leq n$ , a probabilidade de pelo menos  $k$  sucessos é

$$\begin{aligned}\Pr\{X \geq k\} &\leq \sum_{i=k}^n b(i; n, p) \\ &\leq \binom{n}{p} p^k.\end{aligned}$$

**Prova** Para  $S \subseteq \{1, 2, \dots, n\}$ , seja  $A_S$  o evento em que a  $i$ -ésima experiência é um sucesso para todo  $i \in S$ . É claro que  $\Pr\{A_S\} = p^k$  se  $|S| = k$ . Temos

$$\begin{aligned}\Pr\{X \geq k\} &= \Pr\{\text{existe } S \subseteq \{1, 2, \dots, n\} : |S| = k \text{ e } A_S\} \\ &= \Pr\left\{\bigcup_{S \subseteq \{1, 2, \dots, n\} : |S|=k} A_S\right\} \\ &\leq \sum_{S \subseteq \{1, 2, \dots, n\} : |S|=k} \Pr\{A_S\} \\ &= \binom{n}{p} p^k,\end{aligned}$$

onde a desigualdade decorre da desigualdade de Boole (C.18). ■

O corolário a seguir declara novamente o teorema para a extremidade esquerda da distribuição binomial. Em geral, deixaremos a cargo do leitor adaptar as prova de uma extremidade à outra.

### Corolário C.3

Considere uma seqüência de  $n$  experiências de Bernoulli, onde o sucesso ocorre com a probabilidade  $p$ . Se  $X$  é a variável aleatória que denota o número total de sucessos, então para  $0 \leq k \leq n$ , a probabilidade de no máximo  $k$  sucessos é

$$\begin{aligned}\Pr\{X \leq k\} &= \sum_{i=0}^k b(i; n, p) \\ &\leq \binom{n}{n-k} (1-p)^{n-k} \\ &= \binom{n}{k} (1-p)^{n-k}\end{aligned}$$

■

Nosso próximo limite se concentra na extremidade esquerda da distribuição binomial. Seu corolário mostra que, longe da média, a extremidade esquerda diminui exponencialmente.

### Teorema C.4

Considere uma seqüência de  $n$  experiências de Bernoulli, onde o sucesso ocorre com probabilidade  $p$  e o insucesso com probabilidade  $q = 1-p$ . Seja  $X$  a variável aleatória que denota o número total de sucessos. Então, para  $0 < k < np$ , a probabilidade de ocorrerem menos de  $k$  sucessos é

$$\begin{aligned}\Pr\{X < k\} &= \sum_{i=0}^{k-1} b(i; n, p) \\ &< \frac{kq}{np - k} b(k; n, p).\end{aligned}$$

**Prova** Limitamos a série  $\sum_{i=0}^{k-1} b(i; n, p)$  por uma série geométrica, utilizando a técnica da Seção A.2. Para  $i = 1, 2, \dots, k$  obtemos, da equação (C.40)

$$\begin{aligned}\frac{b(i-1; n, p)}{b(i; n, p)} &= \frac{iq}{(n-i+1)p} \\ &< \frac{iq}{(n-i)p} \\ &\leq \frac{kq}{(n-k)p}\end{aligned}$$

Se temos

$$\begin{aligned}x &= \frac{kq}{(n-k)p} \\ &< \frac{kq}{(n-np)p} \\ &= \frac{kq}{nqp} \\ &= \frac{k}{np}\end{aligned}$$

segue-se que

$$b(i-1; n, p) < xb(i; n, p)$$

para  $0 < i \leq k$ . Por iteração, aplicando essa desigualdade  $k - i$  vezes, obtemos

$$b(i; n, p) < x^{k-i} b(k; n, p)$$

para  $0 \leq i < k$  e, consequentemente,

$$\begin{aligned} \sum_{i=0}^{k-1} b(i; n, p) &< \sum_{i=0}^{k-1} x^{k-1-i} b(k-i; n, p) \\ &< b(k; n, p) \sum_{i=1}^{\infty} x^i \\ &= \frac{x}{1-x} b(k; n, p) \\ &= \frac{kq}{np-k} b(k; n, p). \end{aligned}$$

■

### **Corolário C.5**

Considere uma sequência de  $n$  experiências de Bernoulli, onde o sucesso ocorre com probabilidade  $p$  e o insucesso com probabilidade  $q = 1 - p$ . Então, para  $0 < k \leq np/2$ , a probabilidade de menos de  $k$  sucessos é menor que metade da probabilidade de menos de  $k + 1$  sucessos.

**Prova** Como  $k \leq np/2$ , temos

$$\begin{aligned} \frac{kq}{np-k} &\leq \frac{(np/2)q}{np - (np/2)} \\ &= \frac{(np/2)q}{np/2} \\ &\leq 1, \end{aligned}$$

pois  $q \leq 1$ . Sendo  $X$  a variável aleatória que denota o número de sucessos, o Teorema C.4 implica que a probabilidade de menos de  $k$  sucessos é

$$\Pr\{X < k\} = \sum_{i=0}^{k-1} b(i; n, p) < b(k; n, p)$$

Portanto, temos

$$\begin{aligned} \frac{\Pr\{X < k\}}{\Pr\{X < k+1\}} &= \frac{\sum_{i=0}^{k-1} b(i; n, p)}{\sum_{i=0}^k b(i; n, p)} \\ &= \frac{\sum_{i=0}^{k-1} b(i; n, p)}{\sum_{i=0}^{k-1} b(i; n, p) + b(k; n, p)} \\ &< 1/2 \end{aligned}$$

$$\text{pois } \sum_{i=0}^{k-1} b(i; n, p) < b(k; n, p)$$

■ | 885

Os limites sobre a extremidade direita podem ser determinados de modo semelhante. Suas provas ficam para o Exercício C.5-2.

### Corolário C.6

Considere uma seqüência de  $n$  experiências de Bernoulli, onde o sucesso ocorre com probabilidade  $p$ . Seja  $X$  a variável aleatória que descreve o número total de sucessos. Então, para  $np < k < n$ , a probabilidade de mais de  $k$  sucessos é

$$\begin{aligned}\Pr\{X > k\} &= \sum_{i=k+1}^n b(i; n, p) \\ &< \frac{(n-k)p}{k-np} b(k; n, p).\end{aligned}$$

### Corolário C.7

Considere uma seqüência de  $n$  experiências de Bernoulli, onde o sucesso ocorre com probabilidade  $p$  e o insucesso com probabilidade  $q = 1 - p$ . Então, para  $(np + n)/2 < k < n$ , a probabilidade de mais de  $k$  sucessos é menor que metade da probabilidade de mais de  $k + 1$  sucessos. ■

O próximo teorema considera  $n$  experiências de Bernoulli cada uma com uma probabilidade  $p_i$  de sucesso, para  $i = 1, 2, \dots, n$ . Como mostra o corolário subsequente, podemos usar o teorema para fornecer um limite sobre a extremidade direita da distribuição binomial, definindo  $p_i = p$  para cada experiência.

### Teorema C.8

Considere uma seqüência de  $n$  experiências de Bernoulli, onde na  $i$ -ésima experiência, para  $i = 1, 2, \dots, n$ , o sucesso ocorre com probabilidade  $p_i$  e o insucesso ocorre com probabilidade  $q_i = 1 - p_i$ . Seja  $X$  a variável aleatória que descreve o número total de sucessos, e seja  $\mu = E[X]$ . Então para  $r > \mu$ ,

$$\Pr\{X - \mu \geq r\} \leq \left(\frac{\mu e}{r}\right)^r.$$

**Prova** Tendo em vista que, para qualquer  $\alpha > 0$ , a função  $e^{\alpha x}$  é estritamente crescente em  $x$ ,

$$\Pr\{X - \mu \geq r\} = \Pr\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\}, \quad (\text{C.41})$$

onde  $\alpha$  será determinado mais tarde. Usando a desigualdade de Markov (C.29), obtemos

$$\Pr\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\} \leq E[e^{\alpha(X-\mu)}] e^{-\alpha r}. \quad (\text{C.42})$$

A parte mais importante da prova consiste em limitar  $E[e^{\alpha(X-\mu)}]$  e substituir  $\alpha$  por um valor adequado na desigualdade (C.42). Primeiro, avaliamos  $E[e^{\alpha(X-\mu)}]$ . Usando a notação da Seção 5.2, seja  $X_i = I\{\text{a } i\text{-ésima experiência de Bernoulli é um sucesso}\}$  para  $i = 1, 2, \dots, n$ ; isto é,  $X_i$  é a variável aleatória que é igual a 1 se a  $i$ -ésima experiência de Bernoulli é um sucesso, e 0 se ela é um insucesso. Desse modo,

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

e, por linearidade de expectativa,

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

o que implica

$$X = \mu = \sum_{i=1}^n (X_i - p_i).$$

Para avaliar  $E[e^{\alpha(X-\mu)}]$ , substituímos  $X - \mu$ , obtendo

$$\begin{aligned} E[e^{\alpha(X-\mu)}] &= E[e^{\alpha \sum_{i=1}^n (X_i - p_i)}] \\ &= E\left[\prod_{i=1}^n e^{\alpha(X_i - p_i)}\right] \\ &= \prod_{i=1}^n E[e^{\alpha(X_i - p_i)}], \end{aligned}$$

que decorre de (C.23), pois a independência mútua das variáveis aleatórias  $X_i$  implica a independência mútua das variáveis aleatórias  $e^{\alpha(X_i - p_i)}$  (veja o Exercício C.3-5). Pela definição de expectativa,

$$\begin{aligned} E[e^{\alpha(X_i - p_i)}] &= e^{\alpha(1-p_i)} p_i + e^{\alpha(0-p_i)} q_i \\ &= p_i e^{\alpha q_i} + q_i e^{\alpha p_i} \\ &\leq \exp\left(\sum_{i=1}^n p_i e^\alpha\right) \\ &\leq \exp(p_i e^\alpha), \end{aligned} \tag{C.43}$$

onde  $\exp(x)$  denota a função exponencial:  $\exp(x) = e^x$ . (A desigualdade (C.43) se segue das desigualdades  $\alpha > 0$ ,  $q_i \leq 1$ ,  $e^{\alpha q_i} \leq e^\alpha$ , e  $e^{-\alpha q_i} \leq 1$ , e a última linha decorre da desigualdade (3.11).) Conseqüentemente,

$$\begin{aligned} E[e^{\alpha(X-\mu)}] &= \left[\prod_{i=1}^n E[e^{\alpha(X_i - p_i)}]\right] \\ &\leq \prod_{i=1}^n \exp[p_i e^\alpha] \\ &= \exp\left(\sum_{i=1}^n p_i e^\alpha\right) \\ &= \exp(\mu e^\alpha), \end{aligned} \tag{C.44}$$

pois  $\mu = \sum_{i=1}^n p_i$ . Então, da equação (C.41) e das desigualdades (C.42) e (C.44), segue-se que

$$\Pr\{X - \mu \geq r\} \leq \exp(\mu e^\alpha - \alpha r). \tag{C.45}$$

Escolhendo  $\alpha = \ln(r/\mu)$ , (veja o Exercício C.5-7), obtemos

$$\Pr\{X - \mu \geq r\} \leq \exp(\mu e^{\ln(r/\mu)\mu} - r \ln(r/\mu))$$

$$= \exp(r - r \ln(r/\mu))$$

$$\begin{aligned} &= \frac{e^r}{(r/\mu)^r} \\ &= \left(\frac{\mu e}{r}\right)^r. \end{aligned}$$

Quando aplicado a experiências de Bernoulli em que cada experiência tem a mesma probabilidade de sucesso, o Teorema C.8 produz o corolário a seguir, que limita a extremidade direita de uma distribuição binomial.

### **Corolário C.9**

Considere uma sequência de  $n$  experiências de Bernoulli onde cada sucesso em uma experiência ocorre com probabilidade  $p$  e um insucesso ocorre com probabilidade  $q = 1 - p$ . Então, para  $r > np$ ,

$$\begin{aligned}\Pr\{X - np \geq r\} &= \sum_{k=[np+r]}^n b(k; n, p) \\ &\leq \left(\frac{npe}{r}\right)^r.\end{aligned}$$

**Prova** Pela equação (C.36), temos  $\mu = E[X] = np$ . ■

### **Exercícios**

**C.5-1** \*

O que é menos provável: não obter nenhuma cara quando se lança uma moeda comum  $n$  vezes, ou obter menos de  $n$  caras quando se lança a moeda  $4n$  vezes?

**C.5-2** \*

Prove os Corolários C.6 e C.7.

**C.5-3** \*

Mostre que

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k; n, a/(a+1))$$

para todo  $a > 0$  e todo  $k$  tal que  $0 < k < n$ .

**C.5-4** \*

Prove que, se  $0 < k < np$ , onde  $0 < p < 1$  e  $q = 1 - p$ , então

$$\sum_{i=0}^{k-1} p^i q^{n-1-i} < \frac{kq}{np-k} \left(\frac{np}{k}\right)^i \left(\frac{nq}{n-k}\right)^{n-k}.$$

**C.5-5** \*

Mostre que as condições do Teorema C.8 implicam que

$$\Pr\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r.$$

De modo semelhante, mostre que as condições do Corolário C.9 implicam que

$$\Pr\{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r.$$

### C.5-6 \*

Considere uma seqüência de  $n$  experiências de Bernoulli onde, na  $i$ -ésima experiência, para  $i = 1, 2, \dots, n$ , o sucesso ocorre com probabilidade  $p_i$  e o insucesso ocorre com probabilidade  $q_i = 1 - p_i$ . Seja  $X$  a variável aleatória que descreve o número total de sucessos, e seja  $\mu = E[X]$ . Mostre que, para  $r \geq 0$ ,

$$\Pr\{X - \mu \geq r\} \leq e^{-r^2/2n}.$$

(*Sugestão:* Prove que  $p_i e^{\alpha q_i} + q_i e^{\alpha p_i} \leq e^{\alpha^2/2}$ . Depois, siga o esboço da prova do Teorema C.8, usando essa desigualdade em lugar da desigualdade (C.43).)

### C.5-7 \*

Mostre que a escolha de  $\alpha = \ln(r/\mu)$  minimiza o lado direito da desigualdade (C.45).

## Problemas

### C-1 Bolas e caixas

Neste problema, investigamos o efeito de várias hipóteses sobre o número de modos de colocar  $n$  bolas em  $b$  caixas distintas.

- a. Suponha que as  $n$  bolas sejam distintas, e que sua ordem dentro de uma caixa não tenha importância. Demonstre que o número de modos de inserir as bolas nas caixas é  $b^n$ .
- b. Suponha que as bolas sejam distintas e que as bolas em cada caixa estejam ordenadas. Demonstre que o número de modos de inserir as bolas nas caixas é  $(b+n-1)!/(b-1)!$ . (*Sugestão:* Considere o número de modos de arranjar  $n$  bolas distintas e  $b-1$  bastões indistintos em uma linha.)
- c. Suponha que as bolas sejam idênticas e, consequentemente, sua ordem dentro de uma caixa não tenha importância. Demonstre que o número de modos de inserir as bolas nas caixas é  $\binom{b+n-1}{n}$ . (*Sugestão:* Dos arranjos dados na parte (b), quantos se repetem se as bolas se tornam idênticas?)
- d. Suponha que as bolas sejam idênticas e que nenhuma caixa possa conter mais de uma bola. Demonstre que o número de modos de inserir as bolas é  $\binom{b}{n}$ .
- e. Suponha que as bolas sejam idênticas e que nenhuma caixa possa ficar vazia. Demonstre que o número de modos de inserir as bolas é  $\binom{n-1}{b-1}$ .

## Notas do capítulo

Os primeiros métodos gerais para resolver problemas de probabilidade foram discutidos em uma famosa correspondência entre B. Pascal e P. de Fermat, que começou em 1654, e em um livro de C. Huygens de 1657. A teoria da probabilidade rigorosa começou com o trabalho de J. Bernoulli em 1713 e de A. De Moivre em 1730. Desenvolvimentos adicionais da teoria foram propostos por P. S. de Laplace, S.-D. Poisson e C. F. Gauss.

Somas de variáveis aleatórias foram estudadas originalmente por P. L. Chebyshev e A. A. Markov. Os axiomas da teoria da probabilidade foram desenvolvidos por A. N. Kolmogorov em 1933. Limites sobre as extremidades de distribuições foram fornecidos por Chernoff [59] e Hoeffding [150]. Um trabalho produtivo em estruturas combinatórias aleatórias foi realizado por P. Erdős.

Knuth [182] e Liu [205] são boas referências para o estudo de análise combinatória elementar e contagem. Livros-texto padrão como os de Billingsley [42], Chung [60], Drake [80], Feller [88] e Rozanov [263] oferecem amplas introduções à probabilidade.

## Bibliografia

1. Milton Abramowitz e Irene A. Stegun, editores. *Handbook of Mathematical Functions*. Dover, 1965.
2. G. M. Adel'son-Vel'skii e E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259-1263, 1962.
3. Leonard M. Adleman, Carl Pomerance e Robert S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117:173-206, 1983.
4. Alok Aggarwal e Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116-1127, 1988.
5. Alfred V. Aho, John F. Hopcroft e Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
6. Alfred V. Aho, John E. Hopcroft e Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
7. Ravindra K. Ahuja, Thomas L. Magnanti e James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
8. Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin e Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37:213-223, 1990.
9. Ravindra K. Ahuja e James B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748-759, 1989.
10. Ravindra K. Ahuja, James B. Orlin e Robert E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939-954, 1989.
11. Miklos Ajtai, János Komlós e Endre Szemerédi. Sorting in  $c \log n$  parallel steps. *Combinatorica*, 3:1-19, 1983.
12. Miklos Ajtai, Nimrod Megiddo e Orli Waarts. Improved algorithms and analysis for secretary problems and generalizations. Em *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pp. 473-482, 1995.
13. Mohamad Akra e Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195-210, 1998.
14. Noga Alon. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters*, 35:201-204, 1990.
15. Arne Andersson. Balanced search trees made simple. Em *Proceedings of the Third Workshop on Algorithms and Data Structures*, número 709 em Lecture Notes in Computer Science, pp. 60-71. Springer-Verlag, 1993.
16. Arne Andersson. Faster deterministic sorting and searching in linear space. Em *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pp. 135-141, 1996.
17. Arne Andersson, Torben Hagerup, Stefan Nilsson e Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57:74-93, 1998.
18. Tom M. Apostol. *Calculus*, volume 1. Blaisdell Publishing Company, segunda edição, 1967.
19. Sanjeev Arora. *Probabilistic checking of proofs and the hardness of approximation problems*. Tese de doutorado, University of California, Berkeley, 1994.
20. Sanjeev Arora. The approximability of NP-hard problems. Em *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pp. 337-348, 1998.
21. Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753-782, 1998.
22. Sanjeev Arora e Carsten Lund. Hardness of approximations. Em Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pp. 399-446. PWS Publishing Company, 1997.
23. Javed A. Aslam. A simple bound on the expected height of a randomly built binary search tree. Relatório técnico TR2001-387, Dartmouth College Department of Computer Science, 2001.
24. Mikhail J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
25. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela e M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, 1999.
26. Sara Baase e Alan Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, terceira edição, 2000.
27. Eric Bach. Comunicação particular, 1989.
28. Eric Bach. Number-theoretic algorithms. Em *Annual Review of Computer Science*, volume 4, pp. 119-172. Annual Reviews, Inc., 1990.
29. Eric Bach e Jeffery Shallit. *Algorithmic Number Theory – Volume I: Efficient Algorithms*. The MIT Press, 1996.
30. David H. Bailey, King Lee e Horst D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357-371, 1990.
31. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290-306, 1972.
32. R. Bayer e E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173-189, 1972.
33. Pierre Beauchemin, Gilles Brassard, Claude Crépeau, Claude Goutier e Carl Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology*, 1:53-64, 1988.
34. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

35. Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87-90, 1958.
36. Michael Ben-Or. Lower bounds for algebraic computation trees. Em *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pp. 80-86, 1983.
37. Michael A. Bender, Erik D. Demaine e Martin Farach-Colton. Cache-oblivious B-trees. Em *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pp. 399-409, 2000.
38. Samuel W. Bent e John W. John. Finding the median requires  $2n$  comparisons. Em *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pp. 213- 216, 1985.
39. Jon L. Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982.
40. Jon L. Bentley. *Programming Pearls*. Addison-Wesley, 1986.
41. Jon L. Bentley, Dorothea Haken e James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36-44, 1980.
42. Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, segunda edição, 1986.
43. Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest e Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448-461, 1973.
44. Béla Bollobás. *Random Graphs*. Academic Press, 1985.
45. J. A. Bondy e U. S. R. Murty. *Graph Theory with Applications*. American Elsevier, 1976.
46. Gilles Brassard e Paul Bratley. *Algorithmics: Theory and Practice*. Prentice-Hall, 1988.
47. Gilles Brassard e Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.
48. Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20(2):176-184, 1980.
49. Mark R. Brown. *The Analysis of a Practical and Nearly Optimal Priority Queue*. Tese de PhD, Computer Science Department, Stanford University, 1977. Relatório técnico STAN-CS-77-600.
50. Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298-319, 1978.
51. J. P. Buhler, H. W. Lenstra, Jr. e Carl Pomerance. Factoring integers with the number field sieve. Em A. K. Lenstra e H. W. Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pp. 50-94. Springer-Verlag, 1993.
52. J. Lawrence Carter e Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2): 143-154, 1979.
53. Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6): 1028-1047, 2000.
54. Joseph Cheriyan e Torben Hagerup. A randomized maximum-flow algorithm. *SIAM Journal on Computing*, 24(2):203-226, 1995.
55. Joseph Cheriyan e S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18(6):1057-1086, 1989.
56. Boris V. Cherkassky e Andrew V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390-410, 1997.
57. Boris V. Cherkassky, Andrew V. Goldberg e Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2): 129-174, 1996.
58. Boris V. Cherkassky, Andrew V. Goldberg e Craig Silverstein. Buckets, heaps, lists and monotone priority queues. *SIAM Journal on Computing*, 28(4): 1326-1346, 1999.
59. H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493-507, 1952.
60. Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes*. Springer-Verlag, 1974.
61. V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233-235, 1979.
62. V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.
63. V. Chvátal, D. A. Klarnet e D. E. Knuth. Selected combinatorial research problems. Relatório técnico STAN-CS-72-292, Computer Science Department, Stanford University, 1972.
64. Alan Cobham. The intrinsic computational difficulty of functions. Em *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pp. 24-30. North-Holland, 1964.
65. H. Cohen e H. W. Lenstra, Jr. Primality testing and Jacobi sums. *Mathematics of Computation*, 42(165):297-330, 1984.
66. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2): 121-137, 1979.
67. Stephen Cook. The complexity of theorem proving procedures. Em *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151-158, 1971.
68. James W. Cooley e John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297-301, 1965.
69. Don Coppersmith. Modifications to the number field sieve. *Journal of Cryptology*, 6:169-180, 1993.
70. Don Coppersmith e Shmuel Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9(3):251-280, 1990.
71. Thomas H. Cormen. *Virtual Memory for Data-Parallel Computing*. Tese de doutorado, Department of Electrical Engineering and Computer Science, MIT, 1992.
72. Eric V. Denardo e Bennett L. Fox. Shortest-route methods: 1. Reaching, pruning, and buckets. *Operations Research*, 27(1):161-186, 1979.
73. Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert e Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738-761, 1994.
74. Whitfield Diffie e Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644-654, 1976.
75. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269-271, 1959.
76. E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11(5):1277-1280, 1970.

77. Brandon Dixon, Monika Rauch e Robert E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184-1192, 1992.
78. John D. Dixon. Factorization and primality tests. *The American Mathematical Monthly*, 91(6):333-352, 1984.
79. Dorit Dor e Uri Zwick. Selecting the median. Em *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, pp. 28-37, 1995.
80. Alvin W. Drake. *Fundamentals of Applied Probability Theory*. McGraw-Hill, 1967.
81. James R. Driscoll, Harold N. Gabow, Ruth Shrairman e Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343-1354, 1988.
82. James R. Driscoll, Neil Sarnak, Daniel D. Sleator e Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86-124, 1989.
83. Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.
84. Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449-467, 1965.
85. Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126-136, 1971.
86. Jack Edmonds e Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248-264, 1972.
87. Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.
88. William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, terceira edição, 1968.
89. Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.
90. Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.
91. Robert W. Floyd. Permuting information in idealized two-level storage. Em Raymond E. Miller e James W. Thatcher, editores, *Complexity of Computer Computations*, pp. 105-109. Plenum Press, 1972.
92. Robert W. Floyd e Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165-172, 1975.
93. Lester R. Ford, Jr. e D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
94. Lester R. Ford, Jr. e Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66:387-389, 1959.
95. Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83-89, 1976.
96. Michael L. Fredman, János Komlós e Endre Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *Journal of the ACM*, 31(3):538-544, 1984.
97. Michael L. Fredman e Michael E. Saks. The cell probe complexity of dynamic data structures. Em *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 1989.
98. Michael L. Fredman e Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596-615, 1987.
99. Michael L. Fredman e Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424-436, 1993.
100. Michael L. Fredman e Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533-551, 1994.
101. Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74:107-114, 2000.
102. Harold N. Gabow, Z. Galil, T. Spencer e Robert E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109-122, 1986.
103. Harold N. Gabow e Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209-221, 1985.
104. Harold N. Gabow e Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5): 1013-1036, 1989.
105. Zvi Galil e Oded Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2): 103-139, 1997.
106. Zvi Galil e Oded Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2):243-254, 1997.
107. Zvi Galil e Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280-294, 1983.
108. Igal Galperin e Ronald L. Rivest. Scapegoat trees. Em *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pp. 165-174, 1993.
109. Michael R. Garey, R. L. Graham e J. D. Ullman. Worst-case analysis of memory allocation algorithms. Em *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, pp. 143-150, 1972.
110. Michael R. Garey e David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
111. Saul Gass. *Linear Programming: Methods and Applications*. International Thomson Publishing, quarta edição, 1975.
112. Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180-187, 1972.
113. Alan George e Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.
114. E. N. Gilbert e E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933-967, 1959.
115. Michel X. Goemans e David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6): 1115-1145, 1995.

116. Michel X. Goemans e David P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. Em Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pp. 144-191. PWS Publishing Company, 1997.
117. Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. Tese de doutorado, Department of Electrical Engineering and Computer Science, MIT, 1987.
118. Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494-504, 1995.
119. Andrew V. Goldberg, Éva Tardos e Robert E. Tarjan. Network flow algorithms. Em Bernhard Korte, László Lovász, Hans Jürgen Prömel e Alexander Schrijver, editores, *Paths, Flows, and VLSI-Layout*, pp. 101-164. Springer-Verlag, 1990.
120. Andrew V. Goldberg e Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45:783-797, 1998.
121. Andrew V. Goldberg e Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921-940, 1988.
122. D. Goldfarb e M. J. Todd. Linear programming. Em G. L. Nemhauser, A. H. G. Rinnooy-Kan e M. J. Todd, editores, *Handbook in Operations Research and Management Science, Vol. 1, Optimization*, pp. 73-170. Elsevier Science Publishers, 1989.
123. Shafi Goldwasser e Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270-299, 1984.
124. Shafi Goldwasser, Silvio Micali e Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281-308, 1988.
125. Gene H. Golub e Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, terceira edição, 1996.
126. G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.
127. Rafael C. Gonzalez e Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.
128. Michael T. Goodrich e Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.
129. Ronald L. Graham. Bounds for certain multiprocessor anomalies. *Bell Systems Technical Journal*, 45:1563-1581, 1966.
130. Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132-133, 1972.
131. Ronald L. Graham e Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43-57, 1985.
132. Ronald L. Graham, Donald E. Knuth e Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, segunda edição, 1994.
133. David Gries. *The Science of Programming*. Springer-Verlag, 1981.
134. M. Grötschel, László Lovász e Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1988.
135. Leo J. Guibas e Robert Sedgewick. A dichromatic framework for balanced trees. Em *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pp. 8-21. IEEE Computer Society, 1978.
136. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
137. Yijie Han. Improved fast integer sorting in linear space. Em *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pp. 793-796, 2001.
138. Frank Harary. *Graph Theory*. Addison-Wesley, 1969.
139. Gregory C. Harfst e Edward M. Reingold. A potential-based amortized analysis of the union-find data structure. *SIGACT News*, 31(3):86-95, 2000.
140. J. Hartmanis e R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285-306, 1965.
141. Michael T. Heideman, Don H. Johnson e C. Sidney Burrus. Gauss and the history of the Fast Fourier Transform. *IEEE ASSP Magazine*, pp. 14-21, 1984.
142. Monika R. Henzinger e Valerie King. Fully dynamic biconnectivity and transitive closure. Em *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pp. 664-672, 1995.
143. Monika R. Henzinger e Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502-516, 1999.
144. Monika R. Henzinger, Satish Rao e Harold N. Gabow. Computing vertex connectivity: New bounds from old techniques. *Journal of Algorithms*, 34(2):222-250, 2000.
145. Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software*, 16(4):352-368, 1990.
146. C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321-322, 1961.
147. C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10-15, 1962.
148. Dorit S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Math*, 6:243-254, 1983.
149. Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.
150. W. Hoeffding. On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics*, 27:713-721, 1956.
151. Micha Hofri. *Probabilistic Analysis of Algorithms*. Springer-Verlag, 1987.
152. John E. Hopcroft e Richard M. Karp. An  $n^{3/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225-231, 1973.

153. John E. Hopcroft, Rajeev Motwani e Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, segunda edição, 2001.
154. John E. Hopcroft e Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372-378, 1973.
155. John E. Hopcroft e Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294-303, 1973.
156. John E. Hopcroft e Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
157. Ellis Horowitz e Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.
158. Ellis Horowitz, Sartaj Sahni e Sanguthevar Rajasekaran. *Computer Algorithms*. Computer Science Press, 1998.
159. T. C. Hu e M. T. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2):362-373, 1982.
160. T. C. Hu e M. T. Shing. Computation of matrix chain products. Part II. *SIAM Journal on Computing*, 13(2):228-251, 1984.
161. T. C. Hu e A. C. Tucker. Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics*, 21(4):514-532, 1971.
162. David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098-1101, 1952.
163. Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao e Thomas Turnbull. Implementation of Strassen's algorithm for matrix multiplication. Em *SC96 Technical Papers*, 1996.
164. Oscar H. Ibarra e Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463-468, 1975.
165. R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2:18-21, 1973.
166. David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256-278, 1974.
167. David S. Johnson. The NP-completeness column: An ongoing guide – the tale of the second prover. *Journal of Algorithms*, 13(3):502-524, 1992.
168. Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1-13, 1977.
169. David R. Karger, Philip N. Klein e Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321-328, 1995.
170. David R. Karger, Daphne Koller e Steven J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199-1217, 1993.
171. Howard Karloff. *Linear Programming*. Birkhäuser, 1991.
172. N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373-395, 1984.
173. Richard M. Karp. Reducibility among combinatorial problems. Em Raymond E. Miller e James W. Thatcher, editores, *Complexity of Computer Computations*, pp. 85-103. Plenum Press, 1972.
174. Richard M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34:165-201, 1991.
175. Richard M. Karp e Michael O. Rabin. Efficient randomized pattern-matching algorithms. Relatório técnico TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.
176. A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434-437, 1974.
177. Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263-270, 1997.
178. Valerie King, Satish Rao e Robert E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17:447-474, 1994.
179. Jeffrey H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis*. Addison-Wesley, 1990.
180. D. G. Kirkpatrick e R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(2):287-299, 1986.
181. Philip N. Klein e Neal E. Young. Approximation algorithms for NP-hard optimization problems. Em *CRC Handbook on Algorithms*, pp. 34-1-34-19. CRC Press, 1999.
182. Donald E. Knuth. *Fundamental Algorithms*, volume 1 de *The Art of Computer Programming*. Addison-Wesley, 1968. Segunda edição, 1973.
183. Donald E. Knuth. *Seminumerical Algorithms*, volume 2 de *The Art of Computer Programming*. Addison-Wesley, 1969. Segunda edição, 1981.
184. Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14-25, 1971.
185. Donald E. Knuth. *Sorting and Searching*, volume 3 de *The Art of Computer Programming*. Addison-Wesley, 1973.
186. Donald E. Knuth. Big omicron and big omega and big theta. *ACM SIGACT News*, 8(2):18-23, 1976.
187. Donald E. Knuth, James H. Morris, Jr. e Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323-350, 1977.
188. J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57-65, 1985.
189. Bernhard Korte e László Lovász. Mathematical structures underlying greedy algorithms. Em F. Gecseg, editor, *Fundamentals of Computation Theory*, número 117 em Lecture Notes in Computer Science, pp. 205-209. Springer-Verlag, 1981.
190. Bernhard Korte e László Lovász. Structural properties of greedoids. *Combinatorica*, 3:359-374, 1983.
191. Bernhard Korte e László Lovász. Greedoids – a structural framework for the greedy algorithm. Em W. Pulleybank, editor, *Progress in Combinatorial Optimization*, pp. 221-24. Academic Press, 1984.
192. Bernhard Korte e László Lovász. Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):229-238, 1984.
193. Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.

194. David W. Krumme, George Cybenko e K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111-139, 1992.
195. J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48-50, 1956.
196. Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.
197. Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan e D. B. Shmoys, editores. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.
198. C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346-365, 1961.
199. Tom Leighton e Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. Em *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pp. 422-431, 1988.
200. Debra A. Lewiner e Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261-296, 1987.
201. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse e J. M. Pollard. The number field sieve. Em A. K. Lenstra e H. W. Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volume 1554 de *Lecture Notes in Mathematics*, pp. 11-42. Springer-Verlag, 1993.
202. H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649-673, 1987.
203. L. A. Levin. Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265-266, 1973. Em russo.
204. Harry R. Lewis e Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, segunda edição, 1998.
205. C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.
206. László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383-390, 1975.
207. László Lovász e M. D. Plummer. *Matching Theory*, volume 121 de *Annals of Discrete Mathematics*. North Holland, 1986.
208. Bruce M. Maggs e Serge A. Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291-293, 1988.
209. Michael Main. *Data Structures and Other Objects Using Java*. Addison-Wesley, 1999.
210. Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.
211. Conrado Martínez e Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288-323, 1998.
212. William J. Masek e Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1): 18-31, 1980.
213. H. A. Maurer, Th. Ottmann e H.-W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1): 11-14, 1976.
214. Ernst W. Mayr, Hans Jürgen Prömel e Angelika Steger, editores. *Lectures on Proof Verification and Approximation Algorithms*. Número 1367 em Lecture Notes in Computer Science. Springer-Verlag, 1998.
215. C. C. McGeoch. All pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426-441, 1995.
216. M. D. McIlroy. A killer adversary for quicksort. *Software – Practice and Experience*, 29(4):341-344, 1999.
217. Kurt Mehlhorn. *Sorting and Searching*, volume 1 de *Data Structures and Algorithms*. Springer-Verlag, 1984.
218. Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 de *Data Structures and Algorithms*. Springer-Verlag, 1984.
219. Kurt Mehlhorn. *Multidimensional Searching and Computational Geometry*, volume 3 de *Data Structures and Algorithms*. Springer-Verlag, 1984.
220. Alfred J. Menezes, Paul C. van Oorschot e Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
221. Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300-317, 1976.
222. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
223. Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST. and related problems. *SIAM Journal on Computing*, 28(4): 1298-1309, 1999.
224. Louis Monier. *Algorithmes de Factorisation D'Enriens*. Tese de doutorado, L'Université Paris-Sud, 1980.
225. Louis Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1):97-108, 1980.
226. Edward F. Moore. The shortest path through a maze. Em *Proceedings of the International Symposium on the Theory of Switching*, pp. 285-292. Harvard University Press, 1959.
227. Rajeev Motwani, Joseph (Seffi) Naor e Prabhakar Raghavan. Randomized approximation algorithms in combinatorial optimization. Em Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pp. 447-481. PWS Publishing Company, 1997.
228. Rajeev Motwani e Prabhakar Raghavan. *Randomized Algorithms*. Cambridge Unveristy Press, 1995.
229. J. I. Munro e V. Raman. Fast stable in-place sorting with  $O(n)$  data moves. *Algorithmica*, 16(2):151-160, 1996.
230. J. Nievergelt e E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33-43, 1973.
231. Ivan Niven e Herbert S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, quarta edição, 1980.
232. Alan V. Oppenheim e Ronald W. Schafer, with John R. Buck. *Discrete-Time Signal Processing*. Prentice-Hall, segunda edição, 1998.
233. Alan V. Oppenheim e Alan S. Willsky, com S. Hamid Nawab. *Signals and Systems*. Prentice-Hall, segunda edição, 1997.
234. James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(1): 109-129, 1997.

235. Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1993.
236. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
237. Christos H. Papadimitriou e Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.
238. Michael S. Paterson, 1974. Conferência não publicada, Ile de Berder, França.
239. Michael S. Paterson. Progress in selection. Em *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory*, pp. 368-379, 1996.
240. Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000.
241. Steven Phillips e Jeffery Westbrook. Online load balancing and network flow. Em *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pp. 402-411, 1993.
242. J. M. Pollard. A Monte Carlo method for factorization. *BIT*, 15:331-334, 1975.
243. J. M. Pollard. Factoring with cubic integers. Em A. K. Lenstra e H. W. Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pp. 4-10. Springer, 1993.
244. Carl Pomerance. On the distribution of pseudoprimes. *Mathematics of Computation*, 37(156):587-593, 1981.
245. Carl Pomerance, editor. *Proceedings of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography*. American Mathematical Society, 1990.
246. William K. Pratt. *Digital Image Processing*. John Wiley & Sons, segunda edição, 1991.
247. Franco P. Preparata e Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.
248. William H. Press, Brian P. Flannery, Saul A. Teukolsky e William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1986.
249. William H. Press, Brian P. Flannery, Saul A. Teukolsky e William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.
250. R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389-1401, 1957.
251. William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668-676, 1990.
252. Paul W. Purdom, Jr. e Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, and Winston, 1985.
253. Michael O. Rabin. Probabilistic algorithms. Em J. F Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pp. 21-39. Academic Press, 1976.
254. Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128-138, 1980.
255. P. Raghavan e C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365-374, 1987.
256. Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81-87, 1997.
257. Edward M. Reingold, Jurg Nievergelt e Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.
258. Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. Progress in Mathematics. Birkhäuser, 1985.
259. Ronald L. Rivest, Adi Shamir e Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, 1978. Consulte também U.S. Patent 4,405,829.
260. Herbert Robbins. A remark on Stirling's formula. *American Mathematical Monthly*, 62(1):26-29, 1955.
261. D. J. Rosenkrantz, R. E. Steams e P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6:563-581, 1977.
262. Salvador Roura. An improved master theorem for divide-and-conquer recurrences. Em *Proceedings of Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, pp. 449-459, 1997.
263. Y. A. Rozanov. *Probability Theory: A Concise Course*. Dover, 1969.
264. S. Sahni e T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23:555-565, 1976.
265. A. Schönhage, M. Paterson e N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13(2): 184-199, 1976.
266. Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.
267. Alexander Schrijver. Paths and flows – a historical survey. *CWI Quarterly*, 6:169-183, 1993.
268. Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847-857, 1978.
269. Robert Sedgewick. *Algorithms*. Addison-Wesley, segunda edição, 1988.
270. Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995.
271. Raimund Seidel e C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464-497, 1996.
272. João Setubal e João Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.
273. Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall, segunda edição, 2001.
274. Jeffrey Shallit. Origins of the analysis of the Euclidean algorithm. *Historia Mathematica*, 21(4):401-419, 1994.
275. Michael I. Shamos e Dan Hoey. Geometric intersection problems. Em *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pp. 208-215, 1976.
276. M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7:67-72, 1981.
277. David B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. Em William Cook, László Lovász e Paul Seymour, editores, *Combinatorial Optimization*, volume 20 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1995.
278. Avi Shoshan e Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. Em *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pp. 605-614, 1999.

279. Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
280. Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.
281. Daniel D. Sleator e Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362-391, 1983.
282. Daniel D. Sleator e Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652-686, 1985.
283. Joel Spencer. *Ten Lectures on the Probabilistic Method*. Regional Conference Series on Applied Mathematics (No. 52). SIAM, 1987.
284. Daniel A. Spielman e Shang-Hua Teng. The simplex algorithm usually takes a polynomial number of steps. Em *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, 2001.
285. Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.
286. Gilbert Strang. *Linear Algebra and Its Applications*. Harcourt Brace Jovanovich, terceira edição, 1988.
287. Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354- 356, 1969.
288. T. G. Szymanski. A special case of the maximal common subsequence problem. Relatório técnico TR-170, Computer Science Laboratory, Princeton University, 1975.
289. Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146-160, 1972.
290. Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215-225, 1975.
291. Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2): 110-127, 1979.
292. Robert E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.
293. Robert F. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306-318, 1985.
294. Robert E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999.
295. Robert E. Tarjan e Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245-281, 1984.
296. George B. Thomas, Jr. e Ross L. Finney. *Calculus and Analytic Geometry*. Addison-Wesley, sétima edição, 1988.
297. Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. Em *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pp. 550-555, 1998.
298. Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362-394, 1999.
299. Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86-109, 2000.
300. Richard Tolimieri, Myoung An e Chao Lu. *Mathematics of Multidimensional Fourier Transform Algorithms*. Springer-Verlag, segunda edição, 1997.
301. P. van Emde Boas. Preserving order in a forest in less than logarithmic time. Em *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pp. 75-84. IEEE Computer Society, 1975.
302. Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier Science Publishers e The MIT Press, 1990.
303. Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.
304. Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, 1996.
305. Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.
306. Rakesh M. Verma. General techniques for analyzing recursive algorithms with applications. *SIAM Journal on Computing*, 26(2):568-581, 1997.
307. Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309-315, 1978.
308. Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1): 11-12, 1962.
309. Michael S. Waterman. *Introduction to Computational Biology, Maps, Sequences and Genomes*. Chapman & Hall, 1995.
310. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, 1994.
311. Mark Allen Weiss. *Algorithms, Data Structures and Problem Solving with C++*. Addison-Wesley, 1996.
312. Mark Allen Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.
313. Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, 1999.
314. Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509-533, 1935.
315. Herbert S. Wilf. *Algorithms and Complexity*. Prentice-Hall, 1986.
316. J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7:347-348, 1964.
317. S. Winograd. On the algebraic complexity of functions. Em *Actes du Congrès International des Mathématiciens*, volume 3, pp. 283-288, 1970.
318. Andrew C.-C. Yao. A lower bound to finding convex hulls. *Journal of the ACM*, 28(4):780-787, 1981.
319. Yinyu Ye. *Interior Point Algorithms: Theory and Analysis*. John Wiley & Sons, 1997.
320. Daniel Zwillinger, editor. *CRC Standard Mathematical Tables and Formulae*. CRC Press, 30<sup>a</sup> edição, 1996.

# Índice

Este índice usa as convenções a seguir. Os números estão em ordem alfabética como são lidos; por exemplo, “2-3-4, árvore” é indexado como se fosse “dois-três-quatro, árvore”. Quando uma entrada se referir a um local que não seja o texto principal, o número da página será seguido por uma identificação: ex. para exercício, pr. para problema, fig. para figura e n. para nota de rodapé. Um número de página identificado dessa maneira indica com freqüência a primeira página de um exercício, um problema, uma figura ou uma nota de rodapé; essa não é necessariamente a página na qual a referência aparece de fato. Por exemplo, se “linear, pesquisa” fosse definida na página 6 em um exercício que começasse na página 5, a entrada de índice correspondente à expressão “pesquisa linear” seria “5 ex.”.

- $\alpha(n)$ , 408
- $\varepsilon$ -denso, grafo, 507 pr.
- $\varepsilon$ -universal, função hash, 191 ex.
- $\circ$ , notação, 37-38
- $O$ , notação, 44 fig., 35
- $O'$ , notação, 48 pr.
- $\tilde{O}$ , notação, 48 pr.
- $\omega$ , notação, 38
- $\Omega$ , notação, 34 fig., 35-36
- $\rho(n)$ -aproximação, algoritmo, 1022
- $\Theta$ , notação, 32-34, 34 fig.
- $\Theta$ , notação, 48 pr.
- { } (conjunto), 845
- $\in$  (pertence a conjunto), 845
- $\notin$  (não pertence a conjunto), 845
- $\emptyset$  (conjunto vazio), 845
- $\subseteq$  (subconjunto), 845
- $\subset$  (subconjunto próprio), 845
- : (tal que), 846
- $\cap$  (interseção de conjuntos), 846
- $\cup$  (união de conjuntos), 846
- $-$  (diferença de conjuntos), 846
- || (cardinalidade de conjunto), 848
- (comprimento de uma cadeia), 718
- (valor de fluxo), 510
- $\times$  (produto cartesiano), 848
- (produto cruzado), 934
- $\langle \rangle$  (codificação padrão), 770
- (sequência), 852
- ! (fatorial), 43
- $\lceil$  (teto), 40
- $\lfloor$  (piso), 40
- $\Sigma$  (somatório), 835
- $\prod$  (produto), 838
- $\rightarrow$  (relação de adjacência), 853
- (relação de acessibilidade), 854
- $\wedge$  (AND), 500, 779
- $\neg$  (NOT), 779
- $\vee$  (OR), 500, 779
- $\oplus$  (operador de grupo), 682
- $\otimes$  (operador de convolução), 653-654
- \* (operador de fechamento), 771
- | (relação divide), 673
- $\times$  (relação não divide), 673
- $\equiv$  (equivalente, módulo  $n$ ), 71, 1077 ex.
- $\not\equiv$  (não equivalente, módulo  $n$ ), 71
- [ $\alpha$ ]<sub>n</sub> (classe de equivalência, módulo  $n$ ), 674
- $+_n$  (adição, módulo  $n$ ), 683
- $\cdot_n$  (multiplicação, módulo  $n$ ), 683
- $(\frac{a}{b})$  (símbolo de Legendre), 714 pr.
- $\epsilon$  (cadeia vazia), 716, 771
- (relação prefixo), 718
- (relação sufixo), 718
- $>_x$  (relação acima), 744
- $\triangleright$  (símbolo de comentário), 14
- $\leq_p$  (relação de redutibilidade de tempo polinomial), 776-777, 784 ex.
- $Z_n^*$  (elementos de grupo multiplicativo, módulo  $n$ ), 684
- $Z_n^{\pm}$  (elementos diferentes de zero de, 703-704
- AA, árvore, 241
- abeliano, grupo, 682
- aberto, intervalo, 249
- ABOVE, 745
- absolutamente convergente, série, 836
- absorção, leis para conjuntos, 847
- abstrato, problema, 768
- aceitação
  - por um algoritmo, 771
  - por uma automação finita, 725
- aceitação, estado, 725
- aceitável, par de inteiros, 707-708
- acessibilidade em um grafo, 854
- acesso, espúrio, 722
- acesso aleatório, máquina, 16-17
- versus* redes de comparação, 555
- acíclico, grafo, 855
- relação a matrizes, 322 pr.
- acima, relação, 744
- Ackermann, função, 715
- add, instrução, 16-17
- adiantada, tarefa, 319
- adição
  - de inteiros binários, 16 ex.
  - de matrizes, 574
  - de polinômios, 651
  - módulo  $n$  (+<sub>n</sub>), 683
- adjacentes, vértices, 854
- admissível, rede, 538-540
- agregado, fluxo, 625
- agregados, análise, 325-328
  - para algoritmo de Dijkstra, 473
  - para busca de Graham, 754
  - para caminhos mais curtos em um grafo, 468
- para contadores binários, 326-327
- para estruturas de dados de conjuntos disjuntos, 402, 403 ex., 406 ex.
- para heaps de Fibonacci, 393 ex.
- para operações de pilhas, 325-326
- para pesquisa primeiro na extensão, 424-425
- para pesquisa primeiro na profundidade, 431
- para tabelas dinâmicas, 334-335
- agrupamento, 193-194
- Akra-Bazzi, método para resolver uma recorrência, 72
- AKS, rede de ordenação, 570
- aleatória, amostragem, 124
- aleatória, permutação, 81-83
  - uniforme, 74-75, 81
- aleatória, variável, 873-878
- indicador, *ver* indicador, variável aleatória
- aleatório, algoritmo, 75, 79-84
- arredondamento aleatório, 830-831
- e análise probabilística, 79-81
- e entradas médias, 20
- hash universal, 188-191
- ordenação de comparação, 143-144 pr.
- para inserção em uma árvore de pesquisa binária com chaves iguais, 216 pr.
- para o problema da contratação, 80
- para particionamento, 124, 128 ex., 129 pr., 130-131 pr.
- para permutação de um arranjo, 81-83
- para pesquisa em uma lista compacta, 177 pr.
- para satisfabilidade de MAX-3-CNF, 820
- para seleção, 149-152
- pior caso, desempenho do, 124 ex.
- Pollard, heurística rho, 710-713, 713 ex.
- quicksort, 124, 128 ex., 129 pr., 130-131 pr.
- teste de caráter primo de Miller-Rabin, 704-709
- aleatório, arredondamento, 830-831
- alfabeto, 725, 770
- algoritmo, 3
  - como uma tecnologia, 8-9
  - correção de, 4
  - origem da palavra, 31
  - tempo de execução de, 17

Alice, 697  
**ALLOCATE-NODE**, 355  
**ALLOCATE-OBJECT**, 172  
 alocação de objetos, 171-172  
 alta, extremidade de um intervalo, 249  
 alterando  
     uma chave em um heap de Fibonacci, 396-397 pr.  
     variáveis no método de substituição, 53  
 altura  
     de um heap, 104-105, 104-105 ex.  
     de um heap  $d$ -ário, 115 pr.  
     de um nó em um heap, 104-105, 109 ex.  
     de um nó em uma árvore, 859  
     de uma árvore, 859  
     de uma árvore B, 353-354  
     de uma árvore binomial, 367  
     de uma árvore de decisão, 134-135  
     de uma árvore vermelho-preto, 222  
         do preto, 222  
         exponencial, 213  
 altura, função em algoritmos de push-relabel, 531  
 altura balanceada, árvore, 237 pr.  
 amortizada, análise, 324-344  
     análise agregada, 325-328  
     método de contabilidade de, 328-329  
     método potencial de, 329-333  
     para algoritmo de Dijkstra, 473  
     para algoritmo de Knuth-Morris-Pratt, 733  
     para algoritmo genérico de push-relabel, 536-537  
     para árvores de peso balanceado, 341 pr.  
     para busca de Graham, 754  
     para busca em largura, 424-425  
     para busca em profundidade, 431  
     para caminhos mais curtos em um grafo, 468  
     para criação de dinâmica de pesquisa binária, 341 pr.  
     para estruturas de dados de conjuntos disjuntos, 402-403, 403 ex., 406 ex., 408-412, 413 ex.  
     para heaps de Fibonacci, 383-386, 389-390, 393, 399 ex.  
     para permutação de reversão de bits, 340 pr.  
     para pilhas em armazenamento secundário, 363 pr.  
     para reestruturação de árvores vermelho-preto, 342-343 pr.  
     para tabelas dinâmicas, 333-340  
 amortizado, custo  
     em análise agregada, 325  
     no método de contabilidade, 328  
     no método potencial, 330  
 amostragem, 124  
 amostral, espaço, 868  
 ampliando caminho, 517-518, 549 pr.  
 ampliando estruturas de dados, 242-255  
 análise, árvore, 788  
 análise de algoritmos, 16-21  
     ver também amortizada, análise; probabilística, análise  
 ancestral, 859  
     menos comum, 415 pr.  
**AND**, função ( $\wedge$ ), 501, 779  
**AND**, porta, 779  
 ângulo polar, 743 ex.  
 aninhando caixas, 486 pr.  
 aniversário, paradoxo, 85-87  
 anti-simetria, 1076  
**ANY-SEGMENTS-INTERSECT**, 746  
 aos pares, conjuntos disjuntos, 848  
 aos pares, independência, 870-871  
 aos pares, primos relativos, 676  
**APPROX-MIN-WEIGHT-VC**, 821  
**APPROX-SUBSET-SUM**, 825  
**APPROX-TSP-TOUR**, 811  
**APPROX-VERTEX-COVER**, 809  
 aproximação  
     por mínimos quadrados, 603-606  
     por somatório de integrais, 842  
 aproximação, algoritmo, 806-831  
     aleatória, 819  
     para cobertura de vértices de peso mínimo, 820-822  
     para empacotamento de caixas, 828 pr.  
     para emparelhamento máximo, 829 pr.  
     para o problema da cobertura de conjuntos ponderada, 828 pr.  
     para o problema da cobertura de vértices, 807-810  
     para o problema da programação de máquinas paralelas, 829 pr.  
     para o problema da satisfabilidade de MAX-CNF, 823 ex.  
     para o problema de cobertura de conjunto, 815-819  
     para o problema do caixeiro-viajante, 810-815  
     para o problema do clique máximo, 828 pr.  
     para o problema do somatório de subconjuntos, 823-827  
     para o problema MAX-CUT, 823 ex.  
     para satisfabilidade de MAX-3-CNF, 819-820  
 aproximação, esquema, 807  
 aproximação, razão, 806  
 aproximação de 1, algoritmo, 807  
 arbitragem, 486 pr.  
 arco, ver aresta  
 aresta, 853  
     admissível, 538  
     árvore, 427, 429, 433-434  
     capacidade de, 510  
     crítica, 524  
     cruzada, 433-434  
     de peso negativo, 461  
     dianteira, 433-434  
     inadmissível, 538  
     luz, 447  
     ordenação em busca em largura, 442-443 pr.  
     ordenação em busca em profundidade, 434  
     peso de, 420-421  
     ponte, 443 pr.  
     residual, 516  
     saturada, 531  
     segura, 446  
     traseira, 433-434  
 arestas, conectividade, 525 ex.  
 arestas, conjunto, 853  
 argumento de uma função, 852  
 aritmética, modular, 41, 683-687  
 aritmética, série, 836  
 aritmética com infinidades, 464-465  
 aritméticas, instruções, 16-17  
 armazenamento, gerenciamento, 103, 171-172, 173 ex., 185 ex.  
 armazenar, instrução, 16-17  
 arranjo, 14  
     Monge, 71 pr.  
 arredondamento, 822  
     aleatório, 830-831  
 articulação, ponto, 443 pr.  
 árvore, 857-861  
     altura de, 859  
     análise, 788  
     árvores AA, 242  
     árvores B, 349-364  
     AVL, 237-238 pr.  
     binária, ver binária, árvore  
     binomial, 366-368, 383  
     bissecção de, 862 pr.  
     bode expiatório, 241  
     caminhos mais curtos, 462, 482-484  
     de altura balanceada, 237-238 pr.  
     de pesquisa binária ótima, 285-291, 295  
     decisão, 134-135  
     diâmetro de, 428 ex.  
     dinâmica, 346  
     2-3, 241, 364  
     2-3-4, 353, 363 pr.  
     espalhada mínima, ver espalhada mínima, árvore  
     espalhada, ver espalhada mínima, árvore; espalhada, árvore  
     enraizada, 174-176, 858-859  
     fusão, 146, 346-347  
     heap, 103-116  
     intervalo, 249-254  
      $k$ -vizinhos, 241  
     livre, 856, 857-858  
     oblíqua, 241, 346  
     ordem estatística, 242-247  
     percurso, ver árvore, percurso  
     percurso completo de, 812  
     peso balanceado, árvores, 241  
     pesquisa, ver pesquisa, árvore  
     primeiro na extensão, 423, 427-428  
     primeiro na profundidade, 429  
     recursão, 27-28, 54-58  
     treap, 237-238 pr.  
     vermelho-preto, ver vermelho-preto, árvore  
     árvore, aresta, 427, 429, 433-434  
     árvore, percurso, 205, 209 ex., 244-245, 812-813  
     árvore de pesquisa binária, propriedade, 205  
         versus propriedade de heap mínimo, 207 ex.  
 assinatura, 698-699  
 assintótica, eficiência, 32  
 assintótica, notação, 32-40, 47-48 pr.  
     e algoritmos de grafos, 417-418  
     e linearidade de somatórios, 836  
 assintoticamente maior, 39  
 assintoticamente menor, 39  
 assintoticamente não negativo, 33  
 assintoticamente positivo, 33  
 assintoticamente restrito, limite, 33  
 assintótico inferior, limite, 35  
 assintótico superior, limite, 35  
 associativa, operação, 382  
 associativas, leis para conjuntos, 847

atrasada, tarefa, 319  
 atribuição  
     múltipla, 14  
     satisfatória, 780, 785-786  
     verdade, 780, 785-786  
 atributo de um objeto, 15  
 aumentando uma chave em um heap  
     máximo, 112-113  
 áurea, razão, 45, 69 pr.  
 ausente, filho, 860  
 autenticação, 203 pr.  
 autoloop, 853  
 automação  
     de emparelhamento de cadeias,  
         726-730  
     finita, 725  
 auxiliar, função hash, 193  
 auxiliar, programa linear, 643  
 avaliação de um polinômio, 30 pr., 653,  
     657 ex.  
     e suas derivadas, 669 pr.  
     em vários pontos, 670 pr.  
 AVL, árvore, 237 pr.  
 AVL-INSERT, 237 pr.  
 axiomas, para probabilidade, 868

**B**, árvore, 350-364  
     altura de, 353-354  
     árvores 2-3-4, 353  
     chave mínima de, 360 ex.  
     criando, 355  
     dividindo um nó em, 356-358  
     eliminação de, 360-363  
     grau mínimo de, 353  
     inserção em, 356-360  
     nó total em, 353  
     pesquisando, 355  
     propriedades de, 352-355  
 $B^*$ , árvore, 353  
 $B^+$ , árvore, 352  
 BAD-SET-COVER-INSTANCE, 819 ex.  
 BALANCE, 237 pr.  
 balanceada, árvore de pesquisa  
     árvores AA, 241  
     árvores AVL, 237 pr.  
     árvores B, 350-364  
     árvores de bode expiatório, 241  
     árvores de  $k$  vizinhos, 241  
     árvores de peso balanceado, 241, 341  
         pr.  
     árvores 2-3, 341, 364  
     árvores 2-3-4, 353, 363 pr.  
     árvores oblíquas, 241, 346  
     árvores vermelho-preto, 220-241  
         treaps, 237 pr.  
 balde, 140  
 base, 280-281  
 base, função, 603  
 base  $a$ , pseudoprímo, 703-704  
 básica, solução, 628  
 básica, solução viável, 628  
 básica, variável, 620  
 básico, caso, 52  
 Batcher, rede de intercalação ímpar-par, 568 pr.  
 Bayes, teorema, 871-872  
 BELLMAN-FORD, 465  
 Bellman-Ford, algoritmo, 465-468  
     e funções objetivas, 479-480 ex.  
     no algoritmo de Johnson, 505-506  
     otimização de Yen para, 485 pr.

para caminhos mais curtos de todos os pares, 490, 505-506  
 para resolver sistemas de restrições de diferença, 478  
**BELOW**, 745  
 Bernoulli, experiência, 878  
     e bolas e caixas, 88-89  
     e sequências, 89-92  
 BFS, 423  
 BIASED-RANDOM, 75 ex.  
 biconectado, componente, 442 pr.  
 bijetora, função, 852  
 binária, árvore, 859  
     completa, 860  
     número de tipos diferentes, 218 pr.  
     representação de, 174  
         ver também pesquisa binária, árvore binária, pesquisa, 28 ex.  
         com inserção rápida, 341 pr.  
         em árvores B de pesquisa, 360-361  
             ex.  
         em ordenação por inserção, 28 ex.  
 binária, relação, 849  
 binário, algoritmo mdc, 714 pr.  
 binário, contador  
     analisado pelo método de contabilidade, 329  
     analisado pelo método potencial, 331-332  
     analisado por análise agregada, 327  
     de inversão de bits, 340 pr.  
     e heaps binomiais, 378 ex.  
 binário, heap, ver heap  
 binários, código de caracteres, 308  
 binomial, árvore, 366-368  
     não ordenada, 383  
 binomial, coeficiente, 865-866  
 binomial, distribuição, 879-882  
     e bolas e caixas, 88  
     finais de, 883-889  
     valor máximo de, 882 ex.  
 binomial, expansão, 864-865  
 binomial, heap, 365-380  
     chave mínima de, 370  
     criando, 370  
     diminuindo uma chave em, 377  
     e contador binário e adição binária, 378 ex.  
     eliminação de, 377  
     em algoritmo de árvore espalhada mínima, 379-380 pr.  
     extraíndo a chave mínima de, 376  
     inserção em, 375  
     propriedades de, 368  
     tempos de execução de operações sobre, 366 fig.  
     unificando, 371-376

**BINOMIAL-HEAP-DECREASE-KEY**, 377  
**BINOMIAL-HEAP-DELETE**, 377  
**BINOMIAL-HEAP-EXTRACT-MIN**, 375  
**BINOMIAL-HEAP-INSERT**, 375  
**BINOMIAL-HEAP-MERGE**, 372  
**BINOMIAL-HEAP-MINIMUM**, 371  
**BINOMIAL-HEAP-UNION**, 371  
**BINOMIAL-LINK**, 371  
 bipartido, emparelhamento, 396-397,  
     525-529  
     Hopcroft-Karp, algoritmo para,  
         549-550 pr.  
 bipartido, grafo, 856  
      $d$ -regular, 529 ex.

e hipergrafos, 856 ex.  
 fluxo em rede correspondente de, 526  
 bisbilhotando, 343  
 biscoito a ser mantido fora do cesto, 511  
 bissecção de uma árvore, 862 pr.  
 bit, operação, 673  
     no algoritmo de Euclides, 714 pr.  
 bitônica, rede de ordenação, 561-564  
 bitônica, seqüência, 561  
     e caminhos mais curtos, 488 pr.  
 bitônico, percurso, 291 pr.  
**BITONIC-SORTER**, 563  
**BIT-REVERSE-COPY**, 667  
**BIT-REVERSED-INCREMENT**, 340 pr.  
 bits, vetor, 180 ex.  
 Bob, 697  
 bode expiatório, árvore, 241  
 bolas e caixas, 88-89, 889 pr.  
 bom sujeito, 428 ex.  
 Boole, desigualdade, 793 ex.  
 booleana, fórmula, 764, 776 ex., 785-786, 791 ex.  
 booleana, função, 866 ex.  
 booleana, multiplicação de matrizes e fecho transitivo, 600 ex.  
 booleano, circuito combinacional, 780  
 booleano, conectivo, 785-786  
 booleano, elemento combinacional, 779  
 borboleta, operação, 664-665  
 vka, algoritmo, 458  
 braço, 350  
 branco, vértice, 422, 429  
**B-TREE-CREATE**, 355  
**B-TREE-DELETE**, 360  
**B-TREE-INSERT**, 357-358  
**B-TREE-INSERT-NONFULL**, 359  
**B-TREE-SEARCH**, 355, 360-361 ex.  
**B-TREE-SPLIT-CHILD**, 356-357  
**BUBBLESORT**, 29 pr.  
 bucket sort, 140-143  
**BUCKET-SORT**, 140  
**BUILD-MAX-HEAP**, 107  
**BUILD-MAX-HEAP'**, 115 pr.  
**BUILD-MIN-HEAP**, 109

cabeça, em uma unidade de disco, 350  
 cache, 16-17  
 cache, sem memória, 364  
 cadeia, 717, 864  
 cadeia de matrizes, multiplicação, 266-272  
 cadeia de uma envoltória convexa, 754-755  
 cadeias, emparelhamento, 717-737  
     algoritmo de Knuth-Morris-Pratt, 730-736  
     algoritmo de Rabin-Karp, 721-725  
     algoritmo simples, 720-721  
     baseada em fatores de repetição, 736 pr.  
     com caracteres de intervalos, 720  
         ex., 730 ex.  
     por autômatos finitos, 725-730  
 caixa, aninhamento, 486 pr.  
 caixas, empacotamento, 828 pr.  
 camadas  
     convexas, 760 pr.  
     máximas, 760 pr.

caminho, 854  
     ampliando, 517-518, 549-550 pr.  
     crítico, 470  
     hamiltoniano, 776 ex.  
     localização, 404  
     mais curto, *ver* mais curtos, caminhos  
     mais longo, 274, 763  
     peso de, 459  
 caminho, cobertura, 546 pr.  
 caminho, compactação, 404  
 caminho branco, teorema, 433  
 caminho externo, comprimento, 861 ex.  
 caminho hamiltoniano, problema, 802  
     ex.  
 caminho total, comprimento, 217-218  
     pr.  
 caminhos mais curtos, 459-508  
     algoritmo de Bellman-Ford, 465-468  
     algoritmo de Dijkstra, 470-475  
     algoritmo de escalonamento de  
         Gabow, 486 pr.  
     algoritmo de Floyd-Warshall,  
         497-500  
     algoritmo de Johnson, 503-506  
     árvore de, 462, 482-484  
     com arestas de peso negativo, 461  
     com caminhos bitônicos, 488 pr.  
     como um programa linear, 623-624  
     de destino único, 460  
     de par único, 273-274, 460  
     de única origem, 460-489  
     e busca em largura, 424-427, 460  
     e caminhos mais longos, 763  
     e ciclos de peso negativo, 460-461  
     e relaxação, 463-465  
     e restrições de diferença, 475-480  
     em grafos g-densos, 507 pr.  
     em um grafo acíclico orientado,  
         468-470  
     em um grafo não ponderado,  
         273-274, 425  
     em um grafo ponderado, 459  
     estimativa de, 463  
     limite superior, propriedade de, 464,  
         480-481  
     nenhum caminho, propriedade de,  
         465, 481  
     por elevação ao quadrado repetida,  
         494-496  
     por multiplicação de matrizes,  
         492-497  
     propriedade de convergência de,  
         464, 481  
     relaxação de caminho, propriedade  
         de, 465, 481-482  
     subestrutura ótima de, 460-461  
     subgrafo de predecessor,  
         propriedade de, 465, 484  
     todos os pares, 460, 490-508  
     triângulos, desigualdade de, 464,  
         480-481  
     variantes de problemas, 460  
 campo de um objeto, 15  
 campo numérico, peneira, 716  
 cancelamento, lema, 659  
 cancelamento de fluxo, 511  
 capacidade  
     de um corte, 518  
     de uma aresta, 510  
     residual, 515, 517-518  
 capacidade, restrição, 510  
 caracteres, código, 308  
 caráter primo, testes, 702-709, 715  
     caráter pseudoprímo, testes, 704  
     teste de Miller-Rabin, 704-709  
 cardinalidade de um conjunto ( $| \cdot |$ ), 848  
 carga, fator  
     de uma tabela dinâmica, 333-334  
     de uma tabela hash, 183  
 carga, instrução, 16-177  
 carimbo de tempo, 429, 485 ex.  
 Carmichael, número, 704, 709 ex.  
 cartesiana, soma, 657 ex.  
 cartesiano, produto ( $\times$ ), 848  
 CASCADING-CUT, 392  
 cascata, corte, 392  
 caso médio, tempo de execução, 20  
 catalães, números, 218-219 pr., 267  
 certificado  
     em um sistema de criptografia, 701  
     para algoritmos de verificação, 774  
 certo, evento, 868  
 CHAINED-HASH-DELETE, 183  
 CHAINED-HASH-INSERT, 182  
 CHAINED-HASH-SEARCH, 182  
 chamada  
     de uma sub-rotina, 16, 17 n.  
     por valor, 15  
 chapelaria, problema, 79 ex.  
 chave, 11, 99, 111-112, 159  
     estática, 198  
     mediana, de um nó de árvore B, 356  
     pública, 697, 699  
     secreta, 697, 699  
 chave pública, sistema de criptografia,  
     697  
 chinês, teorema do resto, 691-693  
 cíclica, rotação, 736 ex.  
 cílico, grupo, 694  
 ciclo de um grafo, 854-855  
     de peso negativo, *ver* peso negativo,  
         ciclo  
     e caminhos mais curtos, 462  
     hamiltoniano, 764, 773  
         peso médio mínimo, 487 pr.  
 ciclo hamiltoniano, problema, 773,  
     795-799  
 cilindro, 350  
 cinza, vértice, 422, 429  
 circuito  
     combinacional booleano, 780  
     para transformação rápida de  
         Fourier, 667  
 circuito, satisfabilidade, 779-784  
 CIRCUIT-SAT, 781  
 circular, lista duplamente ligada com  
     uma sentinela, 168  
 circular, lista ligada, 166  
     *ver* também ligada, lista  
 classe  
     complexidade, 771-772  
     equivalência, 849  
 classificação de arestas  
     em busca em largura, 442 pr.  
     em busca em profundidade, 433-434,  
         435 ex.  
 cláusula, 788  
 clique, 791  
 CLIQUE, 791  
 clique, problema, 791-794  
     algoritmo de aproximação para, 828  
         pr.  
 CNF (forma normal conjuntiva), 764,  
     768  
 CNF, satisfabilidade, 823 ex.  
 cobertura  
     caminho, 547 pr.  
     por um subconjunto, 815  
     vértice, 794, 807-808, 820-823  
 cobertura de conjunto, problema,  
     815-818  
     ponderado, 828 pr.  
 cobertura de vértices, problema  
     algoritmo de aproximação para,  
         807-810  
     caráter NP-completo de, 794-795  
 codificação de instâncias de problemas,  
     768-770  
 código, 308  
     Huffman, 308-314  
 co-domínio, 851  
 coeficiente  
     binomial, 865  
     de um polinômio, 41, 651  
     em forma relaxada, 621  
 coeficiente, representação, 653  
     e multiplicação rápida, 655-657  
 co-fator, 577  
 coincidente, vértice, 525-526  
 colecionador de cupons, problema, 89  
 colinearidade, 739  
 colisão, 181  
     resolução por encadeamento,  
         182-185  
     resolução por endereçamento  
         aberto, 192-198  
 colocação entre parênteses de um  
     produto de cadeia de matrizes, 266  
 coloração de grafo, problema, 804 pr.  
 colorindo, 804 pr., 861 pr.  
 coluna, ordem, 576  
 coluna, vetor, 572  
 com inversão de bits, contador binário,  
     340 pr.  
 combinação, 1864-865  
 combinacional, circuito, 780  
 combinacional, elemento, 779  
 comentário em pseudocódigo ( $>$ ), 14  
 compacta, lista, 177 pr.  
 COMPACTIFY-LIST, 173 ex.  
 COMPACT-LIST-SEARCH, 177 pr.  
 COMPACT-LIST-SEARCH', 178 pr.  
 comparação, ordenação, 133  
     aleatória, 144 pr.  
     e árvores de pesquisa binária, 206 ex.  
     e heaps intercaláveis, 390-391 ex.  
     e seleção, 154  
 comparação, rede, 555-559  
 comparador, 556  
 comparáveis, segmentos de linhas, 744  
 compatíveis, atividades, 297  
 compatíveis, matrizes, 266-267, 574  
 complementar, relaxação 648 pr.  
 complemento  
     de um conjunto, 647  
     de um evento, 869  
     de um grafo, 794-795  
     de uma linguagem, 771  
     de Schur, 591, 602  
 complemento de Schur, lema, 602  
 completa, árvore  $k$ -ária, 861  
     *ver* também heap  
 completeza de uma linguagem, 784 ex.  
 completo, grafo, 856  
 complexa, raiz de unidade, 658  
     interpolação de, 663

complexidade, classe, 771-772  
     co-NP, 775  
     NP, 764, 775  
     NPC, 764, 778  
     P, 764, 768  
 complexidade, medida, 771-772  
 componente  
     biconectado, 442 pr.  
     conectado, 855  
     fortemente conectado, 855  
 componente, grafo, 439  
 comprimento  
     de um caminho, 854  
     de uma cadeia, 718, 864  
     de uma espinha, 237-238 pr.  
     de uma seqüência, 852  
 comprimento de caminho, de uma árvore, 218 pr., 861 ex.  
 comprimento fixo, código, 308  
 comprimento variável, código, 308  
 computacional, geometria, 738-762  
 computacional, problema, 3-4  
 COMPUTE-PREFIX-FUNCTION, 733  
 COMPUTE-TRANSITION-FUNCTION, 730  
 comum, divisor, 674-675  
     máximo, *ver* máximo divisor comum  
 comum, moeda, 869  
 comum, subexpressão, 664-665  
 comum, subseqüência, 281  
     mais longa, 281-285, 295  
 comum múltiplo, 681 ex.  
 comutativas, leis para conjuntos, 846  
 comutatividade sob um operador, 682  
 concatenação  
     de cadeias, 718  
     de linguagens, 771  
 conclusão, tempo, 321 pr., 829 pr.  
 concreto, problema, 768  
 condicional, independência, 873 ex.  
 condicional, probabilidade, 871  
 conectado, grafo, 855  
 conectivo, 785-786  
 conexo, componente, 1855  
     identificado com o uso de estruturas de dados de conjuntos disjuntos, 399  
     identificado com o uso de busca em profundidade, 436 ex.  
 configuração, 782  
 conjugada, transposição, 600 ex.  
 conjunctive normal form (CNF), *ver* forma normal conjuntiva  
 conjunta, função de densidade de probabilidades, 874  
 conjuntiva, forma normal, *ver* forma normal conjuntiva  
 conjunto ({ }), 845-849  
     convexo, 514 ex.  
     independente, 803 pr.  
 conjunto vazio, leis, 846  
 conjuntos disjuntos, estrutura de dados de, 398-416  
     análise de, 408-412, 413 ex.  
     caso especial de tempo linear de, 416  
     determinação em profundidade, 413 pr.  
     em ancestrais mínimos comuns off-line, 415 pr.  
     em mínimo off-line, 413 pr.  
     em programação de tarefas, 322 pr.  
     implementação de floresta de conjuntos disjuntos, 403-406  
         implementação de lista ligada de, 400-403  
         no logaritmo de Kruskal, 451  
 conjuntos disjuntos, floresta de, 403-406  
     análise de, 408-412, 413 ex.  
     propriedades de ordem de, 408, 413 ex.  
     *ver também* conjuntos disjuntos, estrutura de dados de  
 CONNECTED-COMPONENTS, 399-400  
 co-NP, 775  
 conservação do fluxo, 510  
 consistência de literais, 792  
 consolidando uma lista raiz de um heap de Fibonacci, 386-387  
 CONSOLIDATE, 387  
 construída aleatoriamente, árvore de pesquisa binária, 213-216, 217-218 pr.  
 consulta, 160  
 contabilidade, método, 328-329  
     para contadores binários, 329  
     para operações de pilha, 328-329, 329 ex.  
     para tabelas dinâmicas, 335  
 contador, *ver* binário, contador  
 contagem, 863-868  
     probabilística, 95 pr.  
 contagem, ordenação, 135-137  
     em radix sort, 172  
 contavelmente infinito, conjunto, 848  
 contém em um caminho, 854  
 contínua, distribuição uniforme de probabilidades, 870  
 contração  
     de um grafo não orientado por uma aresta, 856  
     de um matróide, 317  
     de uma tabela dinâmica, 336-338  
 contratação, problema, 73-74  
     análise probabilística de, 78-79  
     on-line, 93-95  
 controle, instruções, 16-17  
 convergência, propriedade, 465, 481-482  
 convergente, série, 836  
 convexa, combinação de pontos, 739  
 convexa, envoltória, 749-756, 761 pr.  
 convexa, função, 876  
 convexas, camadas, 760 pr.  
 convexo, conjunto, 514 ex.  
 convexo, polígono, 742 ex.  
 convolução ( $\otimes$ ), 653  
 convolução, teorema, 663  
 cópia, instrução, 16-17  
 cor de um nó de árvore vermelho-preto, 220  
 corda, 247 ex.  
 correção de um algoritmo, 4  
 corte  
     de um fluxo em rede, 517-520  
     de um grafo não orientado, 447  
     em cascata, 391  
     peso de, 823 ex.  
 COUNTING-SORT, 135-136  
 co-vertical, 745  
 crédito, 328  
 criptografia, sistema, 697-702  
 crítica, aresta, 524  
 crítico, caminho, 470  
 cruzada, aresta, 433-434  
 cruzado, produto ( $\times$ ), 739  
 cruzando um corte, 446-447  
     cúbica, curva, 607 pr.  
     curto-circuito, operador, 15  
     curva, 607 pr.  
     curva elíptica, método de fatoração, 716  
     curvas, ajuste, 603-606  
     custo mínimo, árvore espalhada, *ver* mínima, árvore espalhada  
     custo mínimo, fluxo, 779-780  
     custo mínimo, fluxo de várias mercadorias, 781 ex.  
 CUT, 391  
     dados, estrutura, 6, 159-255, 345-416  
     ampliação de, 242-255  
     árvore 2-3, 241, 364  
     árvore 2-3-4, 353, 363 pr.  
     árvore AA, 241  
     árvore AVL, 237-238 pr.  
     árvore B, 350-364  
     árvore de bode expiatório, 241  
     árvore de fusão, 146, 346  
     árvore de intervalos, 249-254  
     árvore de  $k$  vizinhos, 241  
     árvore de ordem estatística, 242-247  
     árvore de peso balanceado, 241  
     árvore de pesquisa binária, 204-219  
     árvore de pesquisa exponencial, 146, 347  
     árvore dinâmicas, 346  
     árvore enraizada, 174-176  
     árvore oblíquas, 241, 346  
     árvore vermelho-preto, 220-241  
     bits, vetores, 180 ex.  
     conjuntos dinâmicos, 159-160  
     deques, 166 ex.  
     dicionários, 159  
     filas, 163-166  
     filas de prioridades, 111-114  
     heaps, 103-116  
     heaps 2-3-4, 379 pr.  
     heaps binomiais, 365-380  
     heaps de Fibonacci, 381-397  
     heaps relaxados, 397  
     listas de saltos, 241  
     listas ligadas, 166-170  
     no armazenamento secundário, 349-351  
     para conjuntos disjuntos, 398-416  
     para grafos dinâmicos, 347  
     persistente, 236 pr., 346  
     pilhas, 163-164  
     potencial de, 330  
     raiz de árvores, 217 pr.  
     tabelas de endereço direto, 180  
     tabelas hash, 181-185  
     treaps, 238 pr.  
     van Emde Boas, 346  
 DAG-SHORTEST-PATHS, 468  
 d-ário, heap, 115 pr.  
     em algoritmos de caminhos mais curtos, 507 pr.  
 decisão, árvore, 134-135  
     princípio zero-um para, 561 ex.  
 decisão, problema, 765, 767-768  
     e problemas de otimização, 765  
 decisão por um algoritmo, 771  
 DECREASE-KEY, 111-112, 365  
 DECREMENT, 327 ex.  
 definida como positiva, matriz, 377  
 degeneração, 636

deixando um vértice, 853  
 deixando variável, 629  
**DELETE**, 160, 365  
**DELETE-LARGER-HALF**, 333 ex.  
 demanda, paginação, 16-17  
**DeMorgan**, leis, 847  
 densidade de números primos, 702-703  
 densidade de probabilidades, função, 874  
 denso, grafo, 419  
     g-denso, 507 pr.  
 dependência  
     e indicadores de variáveis aleatórias,  
         77  
     linear, 576  
     ver também independência  
 deque, 166 ex.  
**DEQUEUE**, 165  
 derivada de série, 837  
 desalocação de objetos, 171-172  
 descarga de um vértice de  
     transbordamento, 539-540  
 descendente, 858  
 descoberto, vértice, 422, 429  
 descriptor, 112, 366, 382  
 desigualdade, restrição, 616-617  
     e restrições de igualdade, 618  
 desigualdade linear, 612  
 desigualdade linear, problema de  
     viabilidade, 648-649 pr.  
 deslocamento, instrução, 16-17  
 deslocamento em emparelhamento de  
     cadeias, 717  
 desmembrando  
     árvores 2-3-4, 363 pr.  
     nós de árvore B, 356-357  
 destino, 1013  
 destino, vértice, 460  
 destino único, caminhos mais curtos,  
     460  
 devio, instruções, 16-17  
 devio condicional, instrução, 16-17  
 devio padrão, 877  
 det, ver determinante  
 determinação da profundidade,  
     problema, 519 pr.  
 determinante, 577  
     e multiplicação de matrizes, 600 ex.  
 determinística, 79-80  
**DETERMINISTIC-SEARCH**, 95 pr.  
**DFS**, 429-430  
**DFS-VISIT**, 430  
**DFT** (Discrete Fourier Transform), 660  
 diagonal, matriz, 572  
     decomposição de LUP de, 596 ex.  
 diâmetro de uma árvore, 428 ex.  
 dianteira, aresta, 433-434  
 dicionário, 159  
 diferença, equação, ver recorrência  
 diferença, restrições, 475-480  
 diferença de conjuntos (-), 846  
     simétrica, 549-550 pr.  
 diferenciação de séries, 837  
 digital, assinatura, 698  
 digrafo, ver orientado, grafo  
**DIJKSTRA**, 470  
 Dijkstra, algoritmo, 470-475  
     com pesos de arestas inteiros,  
         474-475 ex.  
     implementado com um heap de  
         Fibonacci, 473-474  
     implementado com um heap  
         mínimo, 473-474  
     no algoritmo de Johnson, 504-506  
     para caminhos mais curtos de todos  
         os pares, 490, 504-506  
     semelhança com a busca em largura,  
         473, 474, 474 ex.  
     semelhança com o algoritmo de Prim,  
         452, 473-474  
 diminuindo uma chave  
     em heaps binomiais, 376-377  
     em heaps de Fibonacci, 390-393  
     em heaps 2-3-4, 379 pr.  
 dinâmica, árvore, 346  
 dinâmica, tabela, 333-340  
     analisada por análise agregada,  
         334-335  
     analisada por método de  
         contabilidade, 335  
     analisada por método potencial,  
         335-336, 338-339  
     fator de carga de, 333-334  
 dinâmico, conjunto, 159-161  
     ver também estrutura de dados  
 dinâmico, grafo, 399 n.  
     algoritmo de árvore espalhada  
         mínima, 455 ex.  
     estruturas de dados para, 347  
     fecho transitivo de, 507 pr.  
**DIRECT-ADDRESS-DELETE**, 180  
**DIRECT-ADDRESS-INSERT**, 180  
**DIRECT-ADDRESS-SEARCH**, 180  
**DIRECTION**, 741  
 direita, conversão, 225 ex.  
 direita, espinha, 237-238 pr.  
 direita, filho da, 860  
 direita, rotação, 223  
 direita, subárvore da, 860  
 direta, substituição, 588  
 direto, endereçamento, 180-181  
 direto, raio horizontal, 743-744 ex.  
**DISCHARGE**, 539-540  
 disco, 350  
 disco, 749 ex.  
     ver também secundário,  
         armazenamento  
 discreta, variável aleatória, 873-878  
 discreto, logaritmo, 694  
 disjuntiva, forma normal, 789  
 disjuntos, conjuntos, 847  
**DISK-READ**, 351  
**DISK-WRITE**, 351  
 dispositivo, 795  
 distância  
     de um caminho mais curto, 424  
     edição, 291 pr.  
     euclíadiana, 756-757  
      $L_m$ , 760 ex.  
     Manhattan, 156 pr., 760 ex.  
 distribuição  
     binomial, 879-882  
     de entradas, 74-75, 79-80  
     de envoltória esparsa, 762 pr.  
     de números primos, 703  
     geométrica, 872  
     probabilidades, 868-869  
 distribuição de probabilidades, função,  
     143 ex.  
 distributivas, leis para conjuntos, 847  
 divergente, série, 836  
 divide, instrução, 16-17  
 divide, relação ( $\sqsubset$ ), 673  
 dividir e conquistar, método, 21-25  
     análise de, 25  
     e árvores de recursão, 54  
     para algoritmo de Strassen, 579-585  
     para conversão de binário em  
         decimal, 677 ex.  
     para Fast Fourier Transform,  
         661-663  
     para inversão de matrizes, 598-600  
     para localizar a envoltória convexa,  
         749-750  
     para localizar o par de pontos mais  
         próximos, 757-759  
     para multiplicação, 668 pr.  
     para ordenação por intercalação,  
         21-28  
     para pesquisa binária, 28 ex.  
     para quicksort, 117-132  
     para seleção, 149-155  
     relação com a programação  
         dinâmica, 259  
     resolvendo recorrências para, 50-72  
 divisão, método, 186-187  
 divisão em duas partes iguais, lema, 659  
 divisão teorema, 674  
 divisor, 673-674  
     comum, 674  
     ver também máximo divisor comum  
 DNA, 281, 291-292 pr.  
**DNF** (disjunctive normal form), 789  
     ver também disjuntiva, forma  
         normal  
 2-3, árvore, 241, 364  
 2-3-4, árvore, 353  
     e árvores vermelho-preto, 354-355  
         ex.  
     desmembrando, 363 pr.  
     junção, 363 pr.  
 2-3-4, heap, 379 pr.  
**2-CNF**, satisfatibilidade, 791 ex.  
     e satisfatibilidade de 3-CNF, 764  
**2-CNF-SAT**, 791-792 ex.  
 domina, relação, 760 pr.  
 domínio, 851  
 d-regular, grafo, 529 ex.  
 dualidade, 638-642  
     fraca, 638  
 duas passagens, método, 405  
 duplamente ligada, lista, 166  
     ver também ligada, lista  
 duplo, hash, 194-195, 197 ex.  
 duplo linear, programa, 658  
 e (and), em pseudocódigo, 15  
 e, 42  
 E[ ] (valor esperado), 875  
 edição, distância, 292 pr.  
 Edmonds-Karp, algoritmo, 521-525  
 eixo, 350  
 elementar, evento, 868  
 elemento de um conjunto ( $\in$ ), 845  
 elemento máximo de um conjunto  
     parcialmente ordenado, 850  
 elevação ao quadrado, repetida  
     para caminhos mais curtos de todos  
         os pares, 494-496  
     para elevar um número a uma  
         potência, 696  
 eliminação  
     de árvores B, 360-363  
     de árvores de intervalos, 251  
     de árvores de ordem estatística, 246  
     de árvores de pesquisa binária,  
         211-212

- de árvores vermelho-preto, 231-236  
 de filas, 164  
 de heaps, 114 ex.  
 de heaps 2-3-4, 379 pr.  
 de heaps binomiais, 376-377  
 de heaps de Fibonacci, 393, 396 pr.  
 de listas ligadas, 168  
 de pilhas, 163  
 de status de linhas de varredura, 745  
 de tabelas dinâmicas, 338  
 de tabelas hash de endereço aberto, 192-193  
     de tabelas hash encadeadas, 183  
 else, em pseudocódigo, 14  
 em largura, busca, 422-428  
     e caminhos mais curtos, 425-426, 460  
     semelhança com o algoritmo de Dijkstra, 473, 474 ex.  
 em grau, 854  
 em ordem, percurso de árvore, 205, 209-210 ex., 244-245  
 emparelhamento  
     bipartido ponderado, 396-397  
     de cadeias, 717-737  
     de máximo, 829 pr.  
     e fluxo máximo, 525-529  
         máximo, 809, 829 pr.  
 emparelhamento de cadeias, autômato, 726-730, 730 ex.  
 em profundidade, busca, 429-436  
     na localização de componentes fortemente conexos, 438-441  
     na localização de pontos de articulação, pontes e componentes biconectados, 443 pr.  
     na ordenação topológica, 436-438  
 empurrão, operação (em algoritmos de push-relabel), 531  
 empurrão sobre uma pilha de tempo de execução, 130-131 pr.  
 encadeamento  
     de árvores binomiais, 366-367  
     de raízes de heap de Fibonacci, 386  
 encadeando, 182-184, 202 pr.  
 endereçamento, aberto, *ver* aberto, endereçamento  
 endereço aberto, tabela hash, 192-198  
     duplo, hash, 194-195, 197-198 ex.  
     linear, sondagem, 193  
     quadrática, prova, 193-194, 202 pr.  
 endereço direto, tabela, 179-181  
 enraizada, árvore, 858  
     representação de, 174-176  
 entrada  
     distribuição de, 74, 79-80  
     para um algoritmo, 3  
     para um circuito combinacional, 780  
     para uma porta lógica, 779  
     tamanho da, 17  
 entrada, alfabeto, 725  
 entrada, fio, 556  
 entrada, sequência, 556  
 entropia, função, 866  
 entropia binária, função, 866  
 envoltória convexa, 749-756, 762 pr.  
 envoltória esparsa, distribuição, 762 pr.  
 equação  
     e notação assintótica, 36-37  
     normal, 605  
     recorrência, *ver* recorrência
- equivalência, classe, 849  
     módulo  $n$  ( $[a]_n$ ), 674  
 equivalência, modular, 41, 851 ex.  
 equivalência, relação, 849-850  
     e equivalência modular ( ), 851 ex.  
 equivalentes, programas lineares, 617-618  
 escalar, múltiplo, 574  
 escalar, produto de fluxo, 514 ex.  
 escalonamento  
     em caminhos mais curtos de uma única origem, 486 pr.  
     em fluxo máximo, 548 pr.  
 escape, problema, 547 pr.  
 escolha  $\overset{k}{\sim}$ , 864-865  
 espalhada, árvore, 315, 445  
     gargalo, 457 pr.  
     verificação de, 458  
     *ver também* mínima, árvore espalhada  
 esparso, grafo, 419  
 espelhamento, 660 n.  
 esperado, tempo de execução, 20  
 esperado, valor, 875-876  
     de uma distribuição binomial, 880  
     de uma distribuição geométrica, 878  
     de um indicador de variável aleatória, 76  
 espinha, 237-238 pr.  
 espúrio, acesso, 722  
 esquerda, espinha, 237-238 pr.  
 esquerda, filho, 860  
 esquerda, rotação, 223  
 esquerda, subárvore, 860  
 essencial, termo, 582  
 essencialidade, teorema, 528  
 estabilidade  
     numérica, 571, 587, 608-609  
     de algoritmos de ordenação, 137, 140 ex.  
 estado de um autômato finito, 725  
 estado final, função, 726  
 estático, conjunto de chaves, 198  
 estático, grafo, 399 n.  
 estrelado, polígono, 756 ex.  
 estritamente crescente, 40  
 estritamente decrescente, 40  
 estrutura de blocos em pseudocódigo, 14  
 estrutura de parênteses de busca em profundidade, 431  
 EUCLID, 679  
 Euclides, algoritmo, 679-682, 714 pr.  
 euclidiana, distância, 756  
 euclidiana, norma, 575  
 euclidiano bitônico, problema do caixeiro-viajante, 291-292 pr.  
 Euler, função phi, 685  
 Euler, teorema, 694, 709 ex.  
 Euler, viagem, 763  
     e ciclos hamiltonianos, 763  
 evento, 868  
 evento, ponto, 745  
 evidência do caráter composto de um número, 704-705  
 EXACT-SUBSET-SUM, 824  
 excesso, fluxo, 529-530  
 exclusão e inclusão, 849 ex.  
 exclusiva, fatoração de inteiros, 676  
 execução, tempo, 17  
     caso médio, 20  
     melhor caso, 20 ex., 36
- de uma rede de comparação, 57  
 esperado, 20  
 de um algoritmo de grafo, 417-418  
 ordem de crescimento, 20  
 taxa de crescimento, 20  
 pior caso, 20, 36  
 executar uma sub-rotina, 17 n.  
 expansão de uma tabela dinâmica, 334  
 expectativa, *ver* esperado, valor  
 experiência, Bernoulli, 878  
 experiência, divisão, 703  
 explorado, vértice, 731  
 exponenciação  
     modular, 696  
 exponenciação, instrução, 16-17  
 exponencial, altura, 213  
 exponencial, árvore de pesquisa, 146, 347  
 exponencial, função, 41-42  
 exponencial, série, 837  
 EXTENDED-EUCLID, 681  
 EXTEND-SHORTEST-PATHS, 493  
 extensão de um conjunto, 315  
 exterior de um polígono, 743 ex.  
 externo, nó, 859  
 externo, produto, 575  
 extração  
     de uma pilha em tempo de execução, 130-131 pr.  
     operação de pilha, 164  
 EXTRACT-MAX, 112  
 EXTRACT-MIN, 112, 365  
 extraíndo a chave máxima  
     de heaps d-ários, 115 pr.  
     de heaps máximos, 112  
 extraíndo a chave mínima  
     de heaps binomiais, 375-376  
     de heaps de Fibonacci, 385-390  
     de heaps 2-3-4, 379 pr.  
     de quadros de Young, 115 pr.  
 extremidade baixa de um intervalo, 249
- falha em uma experiência de Bernoulli, 878  
 fan-out, 780  
 Farkas, lema, 649 pr.  
 FASTER-ALL-PAIRS-SHORTEST-PATHS, 496, 496 ex.  
 FASTEST-WAY, 264  
 fator, 674  
     giro, 662  
 fator de desvio, em árvores B, 651-652  
 fator de repetição de uma cadeia, 737 pr.  
 fatoração, 896-901, 716  
     única, 676  
 fatorial, função (!), 43-44  
 fechado, intervalo, 249  
 fechado, semi-anel, 508  
 fechamento  
     de uma linguagem, 771  
     operador (\*), 771  
     propriedade de grupo, 682  
     transitivo, *ver* transitivo, fecho  
 Fermat, teorema, 694  
 FFT (Fast Fourier Transform), *ver* transformação rápida de Fourier (FFT)  
 FIB-HEAP-CHANGE-KEY, 397 pr.  
 FIB-HEAP-DECREASE-KEY, 390-391  
 FIB-HEAP-DELETE, 393  
 FIB-HEAP-EXTRACT-MIN, 386

**FIB-HEAP-INSERT**, 384  
**FIB-HEAP-LINK**, 387  
**FIB-HEAP-PRUNE**, 397 pr.  
**FIB-HEAP-UNION**, 385-386  
**Fibonacci**, heap, 381-397
 

- alterando uma chave em, 397 pr.
- chave mínima de, 385
- criando, 384
- diminuindo uma chave em, 391-393
- eliminação de, 393, 396 pr.
- extraíndo a chave mínima de, 386-390
- função potencial para, 383
- grau máximo de, 383, 394-396
- inserção em, 384-385
- no algoritmo de Dijkstra, 473-474
- no algoritmo de Johnson, 506
- no algoritmo de Prim, 454
- podando, 397 pr.
- tempos de execução de operações sobre, 366 fig.
- unificando, 385-386

**Fibonacci**, números, 45, 69 pr., 394-395
 

- computação de, 714 pr.

**FIFO** ("first in, first out", primeiro a entrar, primeiro a sair), 163
 

- ver também* fila

**fila**, 163-165
 

- em algoritmos de push-relabel, 546 ex.
- em busca em largura, 423
- implementação de lista ligada de, 169 ex.
- implementada por pilhas, 166 ex.
- prioridades. *ver* prioridades, fila

**filha**, 858-859, 860
 

- filho da esquerda, representação de irmão da direita, 173-174, 176 ex.

**final**

- de uma distribuição binomial, 883-889
- de uma fila, 164-165
- de uma lista ligada, 166

**final**, recursão, 130-131 pr., 301
 **FIND-DEPTH**, 413 pr.
 **FIND-SET**, 399
 

- implementação de, com floresta de conjuntos disjuntos, 405, 416
- implementação de, com lista ligada, 400

**finita**, sequência, 852
 **FINITE-AUTOMATON-MATCHER**, 727-728
 **finito**, autômato, 725
 

- para emparelhamento de cadeias, 726-730

**finito**, conjunto, 847-848
 **finito**, grupo, 682
 **fio**, 556, 780
 **floresta**, 856, 857
 

- de conjuntos disjuntos, 403-406
- primeiro na profundidade, 429

**FLOYD-WARSHALL**, 498
 **FLOYD-WARSHALL'**, 502 ex.
 **Floyd-Warshall**, algoritmo, 497-500, 501 ex.
 **fluxo**, 510-514
 

- agregado, 625
- bloqueio, 550-551
- de valor inteiro, 527
- excesso, 529
- líquido total, 510-511
- positivo total, 510-511

 soma, 514 ex.
 

- valor de, 510

**fluxo**, conservação, 51
 **fluxo**, rede, 510-515
 

- com capacidades negativas, 549 pr
- correspondendo a um grafo bipartido, 526
- corte de, 518-520
- fluxo de bloqueio, 550
- fluxo máximo, corte mínimo, teorema, 520
- folha, 859
- for
  - e loops invariantes, 13 n.
  - em pseudocódigo, 14

**FORD-FULKERSON**, 520-521
 **Ford-Fulkerson**, método, 515-526
 **FORD-FULKERSON-METHOD**, 515
 forma canônica para programação de tarefas, 319
 forma normal conjuntiva, 764, 788
 forma normal disjuntiva, *ver* disjuntiva, forma normal
 formal, série de potências, 69 pr.
 fortemente conectado, grafo, 855
 fortemente conexo, componente, 85
 decomposição em, 438-441
 fraca, dualidade, 638
 **FREE-OBJECT**, 172
 freqüência, domínio, 651
 função, 1077-1080
 

- convexa, 876
- de Ackermann, 415
- de base, 603
- linear, 19, 612
- objetivo, *ver* objetivo, função
- prefixo, 728-732
- quadrática, 19
- sufixo, 726
- transição, 725

 função sufixo, desigualdade, 728
 função sufixo, lema de recursão, 728
 funcional, iteração, 44
 fundo de uma pilha, 163
 fusão, árvore, 146
 **Gabow**, algoritmo de escalonamento para caminhos mais curtos de única origem, 486 pr.
 **grafo**, *ver* orientado, grafo acíclico
 **gargalo**, árvore espalhada, 457 pr.
 **gargalo**, problema do caixeiro-viajante, 1033 ex.
 gaussiana, eliminação, 590
 **GENERIC-MST**, 446
 **GENERIC-PUSH-RELABEL**, 533
 geométrica, distribuição, 878
 e bolas e caixas, 88
 geométrica, série, 837
 gerador
 

- de um subgrupo, 686

 geral, peneira de campo numérico, 716
 gerando função, 69 pr.
 giro, fator, 662
 global, variável, 14
 **Goldberg**, algoritmo, *ver* push-relabel, algoritmo
 grade, 547 pr.
 gráfico, matróide, 314, 458
 **grafo**, 853-856
 

- algoritmos para, 417-551
- Busca em largura de, 442-428

**Busca em profundidade de**, 429-436
 caminho mais curto em, 425
 complemento de, 794-795
 componente, 439-440
 denso, 419
 dinâmico, 499 n.
 **g-denso**, 507 pr.
 esparsa, 419
 estático, 399 n.
 hamiltoniano, 773
 intervalo, 303 ex.
 matriz de incidência de, 322 pr., 422 ex.
 não hamiltoniano, 773
 ponderado, 420-421
 por lista de adjacência, representação de, 420
 por matriz de adjacência, representação de, 420-421
 unicamente conectado, 436 ex.
 viagem de, 798
 *ver também* orientado, grafo
 acíclico; orientado, grafo; fluxo, rede; não orientado, grafo; árvore
 **GRAFT**, 413 pr.
 **Graham**, varredura, 750-754
 **GRAHAM-SCAN**, 750
 **GRAPH-ISOMORPHISM**, 775 ex.
 **grau**

- de um nó, 859
- de um polinômio, 41
- de um vértice, 854
- de uma raiz de árvore binomial, 366-367
- máximo, de um heap de Fibonacci, 383, 390 ex., 394-396
- mínimo, de uma árvore B, 353

**grau**, limite, 651
 **greedoid**, 322-323
 **GREEDY**, 316-317
 **GREEDY-ACTIVITY-SELECTOR**, 302-303
 **GREEDY-SET-COVER**, 816
 **grupo**, 682-687
 

- cíclico, 694

**grupo aditivo**, módulo  $n$ , 6383
 **grupo multiplicativo**, módulo  $n$ , 684
 **gulosa**, propriedade de escolha, 304-305
 

- de códigos de Huffman, 310-312
- de um matróide ponderado, 317

**guloso**, algoritmo, 296-323
 

- algoritmo de Dijkstra, 470-475
- algoritmo de Kruskal, 470-452
- algoritmo de Prim, 452-454
- comparação com programação dinâmica, 273-274, 280-281 ex., 299-300, 304, 305-306
- e matróides, 314-318
- elementos de, 303-307
- em um matróide ponderado, 316-318
- para árvore espalhada mínima, 445
- para código de Huffman, 308-314
- para o problema da cobertura de conjunto ponderado, 828 pr.
- para o problema da cobertura de conjuntos, 815-818
- para problema da mochila fracionária, 305-306
- para programação de tarefas, 319-321, 321 pr., 322 pr.

para seleção de atividade, 297-303  
 para troca de moeda, 321 pr.  
 propriedade de escolha gulosa em, 304-305  
 subestrutura ótima em, 305  
**HALF-CLEANER**, 562  
**Hall**, teorema, 529-530 ex.  
**HAM-CYCLE**, 774  
 hamiltoniano, caminho, 776 ex.  
 hamiltoniano, ciclo, 764, 773-774  
 hamiltoniano, grafo, 773-774  
**HAM-PATH**, 776 ex.  
 handshake, lema, 856 ex.  
 harmônica, série, 837  
 harmônico, número, 837  
 hash, 179-203  
     duplo, 194-195, 197 ex.  
     encadeamento, 182-184, 202 pr.  
     endereçamento aberto, 192-198  
     k-universal, 203 pr.  
     perfeito, 198-201  
     universal, 188-191  
 hash, função, 181, 185-192  
     auxiliar, 193  
     divisão, método, 186-187  
     g-universal, 191 ex.  
     multiplicação, método, 187  
     resistente a colisões, 701  
     universal, 188-191  
 hash, tabela, 181-185  
     dinâmica, 339 ex.  
     secundária, 198  
     *ver também* hash  
 hash, valor, 181  
**HASH-DELETE**, 197 ex.  
**HASH-INSERT**, 192, 197 ex.  
**HASH-SEARCH**, 193, 197 ex.  
 heap, 103-116  
     altura de, 104-105  
     analizado pelo método potencial, 333  
         ex.  
     aumentando uma chave em, 112-113  
     binomial, *ver* binomial, heap  
     chave máxima de, 112  
     como armazenamento do lixo  
         coletado, 103  
     como uma fila de prioridades, 111-114  
     construindo, 107-109, 115 pr.  
     d-ário, 115 pr., 507 pr.  
     2-3-4, 379 pr.  
     *e treaps*, 238 pr.  
     extraíndo a chave máxima de, 112  
     Fibonacci, *ver* heap de Fibonacci  
     heap máximo, 104  
     heap mínimo, 104-105  
     inserção em, 113  
     intercalável, *ver* intercalável, heap  
     no algoritmo de Dijkstra, 473-474  
     no algoritmo de Huffman, 310  
     no algoritmo de Johnson, 506  
     no algoritmo de Prim, 454  
     para implementar um heap  
         intercalável, 365  
     relaxado, 397  
     tempo de execução de operações  
         sobre, 366 fig.  
 heap, propriedade, 104  
     manutenção de, 105-107  
     *versus* árvore de pesquisa binária,  
         propriedade, 207 ex.  
 heap máximo, propriedade, 104  
     manutenção de, 105-107  
 heap mínimo, ordenado, 368  
 heap mínimo, propriedade, 104, 368  
     manutenção de, 107 ex.  
     *versus* propriedade de árvore de  
         pesquisa binária, 207 ex.  
**HEAP-DECREASE-KEY**, 114 ex.  
**HEAP-DELETE**, 114 ex.  
**HEAP-EXTRACT-MAX**, 112  
**HEAP-EXTRACT-MIN**, 113 ex.  
**HEAP-INCREASE-KEY**, 113  
**HEAP-MAXIMUM**, 112  
**HEAP-MAXIMUM**, 113 ex.  
 heapsort, 103-116  
**HEAPSORT**, 110  
 hereditários, família de subconjuntos, 314  
 hermitiana, matriz, 600 ex.  
 hiperaresta, 856  
 hipergrafo, 856  
     *e grafos bipartidos*, 856 ex.  
**HIRE-ASSISTANT**, 73-74  
**HOARE-PARTITION**, 129 pr.  
**HOPCROFT-KARP**, 549-550 pr.  
 Hopcroft-Karp, algoritmo de  
     emparelhamento bipartido, 549-550  
     pr.  
 hora de término, em seleção de  
     atividade, 297  
 hora de término, na busca em  
     profundidade, 429  
     *e componentes fortemente conexos*,  
         440  
 horizontal, raio, 506 ex.  
 Horner, regra, 30 pr., 653  
     no algoritmo de Rabin-Karp, 721  
**HUFFMAN**, 310  
 Huffman, código, 308-314  
 idempotência, leis para conjuntos, 846  
 identidade, 682  
 identidade, matriz, 572-573  
 if, em pseudocódigo, 14  
 igualdade  
     *de conjuntos*, 845  
     *de funções*, 852  
     *linear*, 612  
 igualdade, restrição, 479 ex., 616-617  
     *e restrições de desigualdade*, 618  
     *rígida*, 627  
     *violação de*, 627  
 imagem, 852  
 ímpar-par, rede de intercalação, 568 pr.  
 ímpar-par, rede de ordenação, 568 pr.  
 inadmissível, aresta, 538  
 incidência, 853  
 incidência, matriz  
     *de um grafo não orientado*, 322 pr.  
     *de um grafo orientado*, 322 pr., 422  
         ex.  
         *e restrições de diferença*, 477  
 inclinada, simetria, 510  
 inclusão e exclusão, 848 ex.  
 incondicional, instrução de desvio, 16-17  
**INCREASE-KEY**, 111-112  
**INCREMENT**, 326  
 incremental, método de projeto, 20  
     *para localizar a envoltória convexa*,  
         749-750  
 independência  
     *de eventos*, 870, 873 ex.  
 de subproblemas em programação  
     dinâmica, 275-276  
 de variáveis aleatórias, 874  
 independente, conjunto, 803 pr.  
     *de tarefas*, 320  
 independente, família de subconjuntos, 314  
 indicador, variável aleatória, 76-79  
     *em análise da altura esperada de*  
         uma árvore de pesquisa binária  
         construída aleatoriamente,  
             213-215  
     *em análise de inserção em um treap*,  
         237 pr.  
     *em análise de seleção aleatória*,  
         150-152, 152 ex.  
     *em análise de seqüências*, 91-92  
     *em análise do paradoxo do*  
         aniversário, 87-88  
     *na análise da bucket sort*, 141-143  
     *na análise de hash*, 184  
     *na análise de hash universal*,  
         188-189  
     *na análise de quicksort*, 127-128,  
         129 pr.  
     *na análise do problema de*  
         contratação, 78-79  
     *na limitação do final direito da*  
         distribuição binomial, 887  
     *no algoritmo de aproximação para*  
         satisfabilidade de MAX-3-CNF,  
         820  
 índice de um elemento de „”, 694  
 induzido, subgrafo, 855  
 inferior, matriz triangular, 573  
 inferior, mediana, 147  
 inferiores, limites  
     *e funções potenciais*, 343  
 para cobertura de vértices de peso  
     mínimo, 822  
 para envoltória convexa, 756 ex.  
 para intercalação, 145 pr.  
 para localização da mediana,  
     156-157  
 para ordenação, 133-136  
 para ordenação média, 145 pr.  
 para tamanho de cobertura ótima de  
     vértices, 809  
 para tamanho de uma rede de  
     intercalação, 566 ex.  
 infinitade, aritmética com, 464-465  
 infinita, seqüência, 852  
 infinita, soma, 835  
 infinito, conjunto, 847-848  
 inicial, estado, 725  
 inicial, submatriz, 601  
 início  
     *de uma fila*, 164-165  
     *de uma lista ligada*, 166  
 início, hora, 297  
**INITIALIZE-PREFLOW**, 532  
**INITIALIZE-SIMPLEX**, 631, 644  
**INITIALIZE-SINGLE-SOURCE**, 463  
 injetora, função, 852  
**INORDER-TREE-WALK**, 205-206  
 inserção  
     *em árvores B*, 356-359  
     *em árvores de intervalos*, 251  
     *em árvores de ordem estatística*, 246  
     *em árvores de pesquisa binária*,  
         210-311  
 em árvores vermelho-preto, 226-230

- em filas, 164  
 em heaps, 113  
 em heaps 2-3-4, 379 pr.  
 em heaps binomiais, 375  
 em heaps  $d$ -ários, 115 pr.  
 em heaps de Fibonacci, 384-385  
 em listas ligadas, 167-168  
 em pilhas, 163  
 em quadros de Young, 115 pr.  
 em status de linhas de varredura, 745  
 em tabelas de endereço direto, 180  
 em tabelas dinâmicas, 334-335  
 em tabelas hash de endereço aberto, 192-193  
 em tabelas hash encadeadas, 1873  
 inserção, ordenação, 8, 11-14, 18-19  
 árvore de decisão para, 134 fig.  
 comparação com a ordenação por intercalação, 9 ex.  
 comparação com quicksort, 123 ex.  
 em bucket sort, 140-143  
 em ordenação por intercalação, 28 pr.  
 em quicksort, 128 ex.  
 usando pesquisa binária, 28 ex.  
 inserindo séries, 837-838  
 inserindo soma, 838  
 INSERT, 111, 160, 333 ex., 365  
 INSERTION-SORT, 13, 18  
 instância  
     de um problema, 3  
     de um problema abstrato, 765, 768  
 instruções do modelo de RAM, 16  
 integração de séries, 837  
 integral para aproximação de somatórios, 842  
 inteiro, tipo de dados, 16-17  
 inteiros ( $Z$ ), 845  
 intercalação  
     de dois arranjos ordenados, 21  
     de  $k$  listas ordenadas, 114 ex.  
     limites inferiores para, 145 pr.  
     usando uma rede de comparação, 564-565  
 intercalação, ordenação por, 8, 21-38  
     comparação com ordenação por inserção, 9 ex.  
     rede de ordenação, implementação de, 566-568  
     uso de ordenação por inserção em, 28 pr.  
 intercalação, rede, 564-565  
     ímpar-par, 568 pr.  
 intercalável, heap, 345, 365  
     e ordenações por comparação, 391 ex.  
     heaps 2-3-4, 379 pr.  
     lista ligada, implementação de, 176 pr.  
     relaxados, heaps, 397  
     tempos de execução de operações sobre, 366 fig.  
     ver também binomial, heap; Fibonacci, heap  
 intercalável, heap máximo, 176 n., 345 n., 365 n.  
 interceptação, 740  
 interior de um polígono, 743 ex.  
 intermediário, vértice, 497  
 interno, comprimento de caminho, 861 ex.  
 interno, nó, 859  
 interno, produto, 575  
 interpolação por uma curva cúbica, 607 pr.  
 interseção  
     de conjuntos ( $\cap$ ), 846  
     de cordas, 247 ex.  
     de linguagens, 770  
     determinando, para dois segmentos de linhas, 740-742  
     determinando, para um conjunto de segmentos de linhas, 743-749  
 INTERVAL-DELETE, 250  
 INTERVAL-INSERT, 250  
 intervalo, 852  
     intervalo, 249  
         ordenação nebulosa de, 131 pr.  
 intervalo, árvore, 249-254  
 intervalo, caráter, 720 ex., 730 ex.  
 intervalo, heurística, 546 ex.  
 intervalo, tricotomia, 250  
 intervalos, problema de coloração grafo de, 303 ex.  
 INTERVAL-SEARCH, 250, 251-252  
 INTERVAL-SEARCH-EXACTLY, 254 ex.  
 intratabilidade, 763  
 introduzindo um vértice, 853  
 introduzindo variável, 629  
 inválido, deslocamento, 717  
 inversa, substituição, 588  
 inversão de bits, permutação, 340 pr., 666  
 inversão em uma seqüência, 30 pr., 79-80 ex.  
 inverso  
     de uma função bijetora, 852  
     de uma matriz, 575, 578 ex.  
     de uma matriz a partir de uma decomposição de LUP, 597-598  
     em um grupo, 683  
     multiplicativo, módulo  $n$ , 690  
 inverso multiplicativo, módulo  $n$ , 690  
 inversor, 779  
 invertível, matriz, 575  
 inviável, programa linear, 617  
 inviável, solução, 617  
 irmão, 859  
 isolado, vértice, 854  
 isomórficos, grafos, 855  
 isomorfismo de subgrafo, problema, 805 ex.  
 iteração de função de prefixo, lema, 734  
 ITERATIVE-FFT, 666  
 ITERATIVE-TREE-SEARCH, 207-208  
 Jarvis, marcha, 754-755  
 Jensen, desigualdade, 876  
 JOHNSON, 505-506  
 Johnson, algoritmo, 503-506  
 Josephus, permutação, 255 pr.  
 junção  
     de árvores 2-3-4, 363 pr.  
     de árvores vermelho-preto, 237 pr.  
 $k$ -ária, árvore, 860  
 Karmarkar, algoritmo, 616, 650  
 Karp, algoritmo do ciclo de peso médio mínimo, 487 pr.  
 $k$ -cadeia, 864  
 $k$ -CNF, 764  
 $k$ -coloração, 804 pr., 861 pr.  
 $k$ -combinação, 864  
 $k$ -conjuntiva, forma normal, 764  
 $k$ -ésima, potência, 677 ex.  
 Kleene, estrela (\*), 771  
 KMP, algoritmo, 731-736  
 KMP-MATCHER, 733  
 Knuth-Morris-Pratt, algoritmo, 731-736  
 $k$ -ordenado, 145 pr.  
 $k$ -permutação, 864  
 Kraft, desigualdade, 861 ex.  
 Kruskal, algoritmo, 450-452  
 $k$ -subcadeia, 864  
 $k$ -subconjunto, 847-848  
 $k$ -universal, hash, 203 pr.  
 $k$ -vizinhos, árvore, 239-240  
 lado de um polígono, 742 ex.  
 Lagrange, fórmula, 654  
 Lagrange, teorema, 686  
 Lamé, teorema, 680  
 LCA, 415 pr.  
 LCS, ver subsequência comum mais longa  
 LCS-LENGTH, 283  
 LEFT, 104  
 LEFT-ROTATE, 224, 253 ex.  
 Legendre, símbolo  $(\frac{a}{p})$ , 714 pr.  
 leve, aresta, 447  
 lexicográfica, ordenação, 217 pr.  
 lexicograficamente menor que, 217 pr.  
 lg (logaritmo binário), 42  
 lg lg (composição de logaritmos), 42  
 lg\* (função logaritmo repetido), 44-45  
 lg <sup>$k$</sup>  (exponenciação de logaritmos), 42  
 liberação, hora, 321 pr.  
 liberação de objetos, 171-172  
 LIFO ("last in, first out", último a entrar, primeiro a sair), 163  
     ver também pilha  
 ligada, lista, 166-170  
     compacta, 173 ex., 177 pr.  
     eliminação de, 167-168  
     inserção em, 167  
     lista de vizinhos, 539-540  
     para implementar conjuntos disjuntos, 400-403  
     pesquisando, 167, 191 ex.  
 limitar um somatório, 838-844  
 limite  
     assintoticamente restrito, 33  
     assintótico inferior, 35  
     assintótico superior, 34  
     em coeficientes binomiais, 865-806  
     em distribuições binomiais, 881  
     nas extremidades de uma distribuição binomial, 883-889  
     polilogarítmico, 43  
 limite, condição, 51-52  
 limite de um polígono, 742 ex.  
 limite superior, propriedade, 465, 481  
 limpa, seqüência, 562  
 linear, dependência, 576  
 linear, desigualdade, 612  
 linear, função, 19, 612  
 linear, igualdade, 612  
 linear, independência, 576  
 linear, ordem, 851  
 linear, pesquisa, 16 ex.  
 linear, programação, 610-650  
     algoritmo de Karmarkar para, 616, 649  
     algoritmo simplex para, 626-638  
     algoritmos para, 615-616  
     aplicações de, 615

dualidade em, 638-643  
 e caminho mais curto de par único, 623  
 e caminhos mais curtos de única origem, 475-480  
 e fluxo de custo mínimo, 624-625  
 e fluxo de várias mercadorias, 625-626  
 e fluxo de várias mercadorias de custo mínimo, 626 ex.  
 e fluxo máximo, 623  
 encontrando uma solução inicial para, 643-647  
 forma padrão para, 616-619  
 forma relaxada para, 619-621  
 métodos de pontos interiores para, 615, 650  
 teorema fundamental da, 647  
*ver também* inteiros, problema de programação linear; 0-1, problema de programação de inteiros  
 linear, restrição, 612  
 linear, sondagem, 193  
 lineares, equações  
     resolvendo, modulares, 688-690  
     resolvendo sistemas de, 585-597  
     resolvendo sistemas tridiagonais de, 607 pr.  
 linearidade de expectativa, 875  
 linearidade de somatórios, 836  
 linguagem, 770  
     completeza de, 787 ex.  
     provando o caráter NP-completo de, 785-786  
     verificação de, 774  
 linha, segmento, 739  
     descobrindo a mudança de direção de, 740  
     descobrindo se dois se interceptam, 740-742  
     descobrindo se quaisquer se interceptam, 743-749  
 linha, vetor, 572  
 linha de montagem, programação, 260-265  
 linha de varredura, status, 745-746  
 linhas, ordem, 576  
 LINK, 405  
 líquido, fluxo através de um corte, 518  
 lista, *ver* ligada, lista  
 lista de adjacência, representação, 420  
 lista filha em um heap de Fibonacci, 382  
 LIST-DELETE, 167-168  
 LIST-DELETE', 167-168  
 LIST-INSERT, 167-168  
 LIST-INSERT', 168-169  
 LIST-SEARCH, 167  
 LIST-SEARCH', 168-169  
 literal, 788  
 livre, árvore, 856, 857-858  
 livre, lista, 172  
 lixo, coleta, 103, 171  
 $L_m$ -distância, 760 ex.  
 ln (logaritmo natural), 42  
 local, variável, 14  
 localização, caminho, 404  
 localização da agência postal, problema, 155-156 pr.  
 logaritmo, função (log), 42-43  
     discreta, 694  
     repetida ( $\lg^*$ ), 44-45

logaritmo discreto, teorema, 694  
 lógica, porta, 779  
 LONGEST-PATH, 772 ex.  
 LONGEST-PATH-LENGTH, 772 ex.  
 LOOKUP-CHAIN, 279  
 loop, invariante, 13  
     e loops for, 12 n.  
     e término, 13  
     inicialização de, 13  
     manutenção de, 13  
     origem de, 31  
     para algoritmo de Prim, 454  
     para algoritmo simplex, 632  
     para aumentar uma chave em um heap, 114 ex.  
 para autômatos de emparelhamento de cadeias, 727-728, 729  
 para busca em largura, 424-425  
 para consolidar a lista de raízes na extração do nó mínimo de um heap de Fibonacci, 387  
 para construção de um heap, 108-109  
 para determinar a ordem de um elemento em uma árvore de ordem estatística, 244  
 para exponenciação modular, 696-697  
 para heapsort, 110 ex.  
 para inserção em árvore vermelho-preto, 227-228  
 para intercalação, 23  
 para o algoritmo de Dijkstra, 472  
 para o algoritmo de Rabin-Karp, 724  
 para o algoritmo de relabel-to-front, 543  
 para o algoritmo genérico de árvore espalhada mínima, 446  
 para o algoritmo genérico de push-relabel, 534  
 para ordenação por inserção, 13-14  
 para particionamento, 118  
 para permutação aleatória de um arranjo, 82-83  
 para pesquisa em uma árvore de intervalos, 252  
 para regra de Horner, 30 pr.  
 para unificação de heaps binomiais, 378 ex.  
 loops, construções em pseudocódigo, 14  
 LU, decomposição, 590-593  
 LU-DECOMPOSITION, 592  
 LUP, decomposição, 587  
     computação de, 592-596  
     de uma matriz de permutação, 595 ex.  
     de uma matriz diagonal, 595 ex.  
     e multiplicação de matrizes, 600 ex.  
     em inversão de matrizes, 597-598  
     uso de, 587-590  
 LUP-DECOMPOSITION, 595  
 LUP-SOLVE, 589  
 mais longa, subsequência comum, 281-285, 295  
 mais longo, caminho simples, 763  
     em um grafo não-ponderado, 274  
 mais longo, problema do ciclo simples, 802 ex.  
 MAKE-BINOMIAL-HEAP, 370  
 MAKE-HEAP, 365  
 MAKE-SET, 398  
     floresta de conjuntos disjuntos, implementação de, 405

lista ligada, implementação de, 400-401  
 MAKE-TREE, 414 pr.  
 Manhattan, distância, 156 pr., 760 ex.  
 mão única, função hash, 701  
 marcado, nó, 383, 391  
 Markov, desigualdade, 877-878 ex.  
 matric, matróide, 314  
 MATRIX-CHAIN-MULTIPLY, 271  
 MATRIX-CHAIN-ORDER, 270  
 MATRIX-MULTIPLY, 266, 494  
 matriz, 571-579  
     adjacência, 421  
     hermitiana, 600 ex.  
     incidência, 322 pr., 422 ex.  
     ordenando as entradas de, 568 ex.  
     predecessora, 491  
     pseudo-inversa de, 605  
     simétrica definida como positiva, 601-603  
     Toeplitz, 669 pr.  
     transposição conjugada de, 600 ex.  
     transposição de, 421, 572  
     *ver também* matrizes, inversão; matrizes, multiplicação  
 matriz de adjacência, representação, 421  
 matrizes, inversão, 597-600  
 matrizes, multiplicação  
     booleana, 600 ex.  
     e cálculo do determinante, 600 ex.  
     e decomposição de LUP, 600 ex.  
     e inversão de matrizes, 598-599  
 Pan, método, 585 ex.  
 para caminhos mais curtos de todos os pares, 491-497  
 Strassen, algoritmo, 579-586  
 matróide, 314-318, 322 pr., 458  
 mau sujeito, 428 ex.  
 MAX-3-CNF, satisfabilidade, 820  
 MAX-CNF, satisfabilidade, 823 ex.  
 MAX-CUT, problema, 823 ex.  
 MAX-FLOW-BY-SCALING, 548 pr.  
 MAX-HEAPIFY, 105-106  
 MAX-HEAP-INSERT, 113  
     construindo um heap com, 114 pr.  
 máximas, camadas, 760 pr.  
 maximização, programa linear, 612  
     e programas lineares de minimização, 617  
 máximo, 147  
     de uma distribuição binomial, 882 ex.  
     em árvores de ordem estatística, 248 ex.  
     em árvores de pesquisa binária, 208  
     em árvores vermelho-preto, 222-223  
     em heaps, 112  
     localizando, 148  
     máximo, fluxo, 509-551  
     algoritmo de Edmonds-Karp para, 523-525  
     algoritmo de relabel-to-front, 538-546  
     algoritmos de push-relabel para, 529-546  
     atualizando, 548 pr.  
     com capacidades negativas, 548-549 pr.  
     como um programa linear, 623  
     e emparelhamento bipartido máximo, 526-529

escalonamento, algoritmo, 548 pr.  
 Ford-Fulkerson, método, 515-526  
 máximo, emparelhamento de, 829 pr.  
 máximo, emparelhamento, 809, 829 pr.  
 máximo, emparelhamento bipartido,  
     526-529, 537 ex.  
     algoritmo de Hopcroft-Karp para, 549  
         pr.  
 máximo, grau em um heap de Fibonacci,  
     384, 390 ex., 394-396  
 máximo, heap, 104  
     aumentando uma chave em, 112-113  
     chave máxima de, 112  
     como uma fila de prioridade máxima,  
         111-114  
     construindo, 107-109  
     eliminação de, 115 ex.  
     em heapsort, 109-112  
     extraíndo a chave máxima de, 112  
     inserção em, 113  
     intercalável, *ver* intercalável, heap  
         máximo  
     máximo, ponto, 760 pr.  
 máximo divisor comum (mdc), 675  
     calculado por algoritmo binário de  
         mdc, 713 pr.  
     calculado por algoritmo de Euclides,  
         677-682  
     com mais de dois argumentos, 681  
         ex.  
     teorema de recursão para, 678  
 MAXIMUM, 111, 160  
 MAYBE-MST-A, 457 pr.  
 MAYBE-MST-B, 457 pr.  
 MAYBE-MST-C, 458 pr.  
 mdc, 674-675, 677 ex.  
     *ver também* máximo divisor comum  
 mediana, 147-157  
     de listas ordenadas, 155 ex.  
     ponderada, 155 pr.  
 mediana, chave de um nó de árvore B,  
     356  
 mediana de 3, método, 130-131 pr.  
 médio, peso de um ciclo, 487 pr.  
 médio, *ver* esperado, valor  
 meio aberto, intervalo, 249  
 melhor caso, tempo de execução, 20 ex.,  
     36  
 membro de um conjunto ( $\in$ ), 845  
 MEMOIZED-MATRIX-CHAIN, 279  
 memória, 349  
 memória, hierarquia, 16-17  
 memorização, 278-280  
 menor ancestral comum, 415 pr.  
 menor de uma matriz, 577  
 mercadoria, 625  
 MERGE, 22  
 MERGE-LISTS, 824  
 MERGER, 565  
 MERGE-SORT, 24-25  
     árvore de recursão para, 280 ex.  
 mestre, método para resolver uma  
     recorrência, 59-61  
 mestre, teorema, 59  
     prova de, 61-68  
 metade 3-CNF, satisfabilidade, 803 ex.  
 MILLER-RABIN, 706  
 Miller-Rabin, teste de caráter primo,  
     704-709  
 MIN-GAP, 254 ex.  
 MIN-HEAPIFY, 107 ex.  
 MIN-HEAP-INSERT, 114 ex.  
 mínima, árvore espalhada, 445-458  
     algoritmo genérico, 446-450  
     algoritmo de Boruvka para, 458  
     construída com o uso de heaps  
         binomiais, 379 pr.  
     em algoritmo de aproximação para  
         problema do caixeteiro-viajante,  
         811  
     em grafos dinâmicos, 455 ex.  
     Kruskal, algoritmo, 450-452  
     Prim, algoritmo, 452-454  
     relação com matróides, 314, 315-316  
     segundo melhor, 455 pr.  
 mínima, chave em heaps 2-3-4, 379 pr.  
 mínima, cobertura de caminho, 546 pr.  
 minimização, programa linear, 612  
     e programas lineares de  
         maximização, 617  
 mínimo, 147  
     em árvores B, 359 ex.  
     em árvores de ordem estatística, 248  
         ex.  
     em árvores de pesquisa binária, 208  
     em árvores vermelho-preto, 222-223  
     em heaps binomiais, 370-371  
     em heaps de Fibonacci, 385  
     localizando, 148  
     off-line, 413 pr.  
 mínimo, ciclo de peso médio, 487 pr.  
 mínimo, corte, 518  
 mínimo, grau de uma árvore B, 353  
 mínimo, heap, 104  
     analisado por método potencial, 333  
     como uma fila de prioridade mínima,  
         114 ex.  
     construindo, 107-109  
     intercalável, *ver* intercalável, heap  
         mínimo  
     no algoritmo de Dijkstra, 473-474  
     no algoritmo de Huffman, 310  
     no algoritmo de Johnson, 506  
     no algoritmo de Prim, 454  
 mínimo, nó de um heap de Fibonacci,  
     383  
 mínimo múltiplo comum, 661 ex.  
 mínimos quadrados, aproximação,  
     603-606  
 MINIMUM, 111-112, 147-148, 160  
 mmc (mínimo múltiplo comum), 682 ex.  
 mochila, problema  
     fracionária, 269, 307 ex.  
         0-1, 305-306, 307 ex.  
 mochila fracionária, problema, 305-306,  
     307 ex.  
 mod, 40, 674  
 modular, aritmética, 41, 682-687  
 modular, exponenciação, 696  
 modulares, equações lineares, 688-690  
 MODULAR-EXPONENTIATION, 696  
 MODULAR-LINEAR-EQUATION-SOLVER,  
     689-690  
 módulo, 40, 674  
 moeda, troca, 321 pr.  
 Monge, arranjo, 71 pr.  
 monotônica, seqüência, 116  
 monotonicamente crescente, 40  
 monotonicamente decrescente, 40  
 movimentação de dados, instruções,  
     16-17  
 MST, 380 pr.  
 MST-KRUSKAL, 450  
 MST-PRIM, 453-454  
 MST-REDUCE, 456-457 pr.  
 multidimensional, transformação rápida  
     de Fourier, 669 pr.  
 multigrafo, 856  
     convertendo no grafo não orientado  
         equivalente, 421 ex.  
 múltipla, atribuição, 14  
 múltiplas, origens e sorvedores, 512  
 multiplicação  
     de matrizes, 574, 578-579 ex.  
     de números complexos, 585 ex.  
     de polinômios, 652  
     de uma cadeia de matrizes, *ver*  
         cadeia de matrizes, multiplicação  
     dividir e conquistar, método para,  
         669 pr.  
     módulo  $n$  ( $\mod n$ ), 683  
 multiplicação, instrução, 16-17  
 multiplicação, método, 187-188  
 múltiplo, 574, 673  
     de um elemento, módulo  $n$ ,  
         688-690  
     mínimo comum, 682 ex.  
 múltiplo, número, 673  
     testemunha para, 704  
 MULTIPOP, 325  
 MULTIPUSH, 327 ex.  
 mutuamente exclusivos, eventos, 868  
 mutuamente independentes, eventos,  
     870  
 N (conjunto de números naturais), 845  
 n elementos, conjunto, 848  
 NAIVE-STRING-MATCHER, 717  
 não básica, variável, 620  
 não correspondente, vértice, 525  
 não determinístico, tempo de  
     polinômio, 774 n.  
     *ver também* NP  
 não divide, relação, 673  
 não enumerável, conjunto, 848  
 não hamiltoniano, grafo, 773  
 não instância, 769 n.  
 não invertível, matriz, 575  
 não limitado, programa linear, 616-617  
 não negatividade, restrição, 616, 617  
 não ordenada, árvore binomial, 383  
 não ordenada, lista ligada, 166  
     *ver também* ligada, lista  
 não orientada, versão de um grafo  
     orientado, 855  
 não orientado, grafo, 853  
     biconectado, componente de, 442 pr.  
     calculando uma árvore espalhada  
         mínima em, 445-458  
     clique em, 791  
     cobertura de vértices de, 794,  
         807-808  
     coloração de, 804 pr., 861 pr.  
     conjunto independente de, 803 pr.  
     convertendo um multigrafo em, 421  
         ex.  
     d-regular, 529 ex.  
     emparelhamento de, 525-526  
     grade, 546 pr.  
     hamiltoniano, 773  
     não hamiltoniano, 773  
     ponte de, 442 pr.  
     ponto de articulação de, 442 pr.  
     *ver também* grafo  
 não ponderados, caminhos mais curtos,  
     273-274

não ponderados, caminhos simples mais longos, 274  
 não saturante, empurrão, 532, 536  
 não singular, matriz, 575  
 não sobreposto, padrão de cadeia, 730 ex.  
 não trivial, potência, 676-677 ex.  
 não trivial, raiz quadrada de 1, módulo  $n$ , 695  
 naturais, números ( $N$ ), 845  
 natural, curva cúbica, 607 pr.  
 nebulosa, ordenação, 131 pr.  
 negativa de uma matriz, 574  
 nenhum caminho, propriedade, 464-465, 481-482  
 NEXT-TO-TOP, 750  
 NIL, 15  
 nível de uma função, 407  
 nó, 858  
     ver também vértice  
 nó, de uma curva, 607 pr.  
 norma de um vetor, 575  
 normal, equação, 605  
 NOT, função ( $\neg$ ), 779  
 NOT, porta, 779  
 novamente identificado, vértice, 532  
 NP (classe de complexidade), 764, 775, 776 ex.  
 NPC (classe de complexidade), 765, 768  
 caráter NP-completo, 763-785  
     de determinar se uma fórmula booleana é uma tautologia, 790 ex.  
     de programação com lucros e prazos finais, 804 pr.  
     do problema da cobertura de conjuntos, 818 ex.  
     do problema da cobertura de vértices, 794-795  
     do problema da partição de conjuntos, 802 ex.  
     do problema da programação de inteiros de 0 a 1, 802 ex.  
     do problema da satisfabilidade de 3-CNF, 788-790  
     do problema da satisfabilidade de circuito, 779-784  
     do problema da satisfabilidade de fórmula, 785-787  
     do problema da satisfabilidade de metade 3-CNF, 803 ex.  
     do problema da soma de subconjuntos, 799-802  
     do problema de coloração de grafos, 804 pr.  
     do problema de programação linear de inteiros, 802 ex.  
     do problema do caixeiro-viajante, 799  
     do problema do caminho hamiltoniano, 802 ex.  
     do problema do ciclo hamiltoniano, 795-798  
     do problema do ciclo simples mais longo, 802 ex.  
     do problema do clique, 791-794  
     do problema do conjunto independente, 803 pr.  
     do problema do isomorfismo de subgrafos, 802 ex.  
     prova, de uma linguagem, 785  
 NP-completo, 764, 778  
 NP-difícil, 778  
 $n$ -tupla, 848-849  
 núcleo de um polígono, 756 ex.  
 nula, árvore, 860  
 nulo, evento, 668  
 nulo, vetor, 576  
 numérica, estabilidade, 571, 587, 608-609  
 números aleatórios, gerador, 75  
 números complexos, multiplicação de, 585 ex.  
 números primos, teorema, 703  
 números pseudo-aleatórios, gerador, 75  
 O, notação, 34 fig., 34-35  
 o, notação, 37-38  
 O', notação, 48 pr.  
 Ó, notação, 48 pr.  
 O maiúsculo, notação, 34 fig., 34-35  
 o minúsculo, notação, 37-38  
 objetivo, função, 475, 479-480 ex., 613, 616  
 objetivo, valor, 614, 617  
     ótimo, 617  
 objeto, 15  
      alocação e liberação de, 171-172  
     implementação de arranjo de, 170-173  
     passagem como parâmetro, 15  
 oblíqua, árvore, 346  
 ocorrência de um padrão, 717  
 ocorrência em ciclo, de algoritmo simplex, 636  
 off-line, problema  
     ancestrais comuns mínimos, 415 pr.  
     mínimo, 413 pr.  
 OFF-LINE-MINIMUM, 414 pr.  
 Ômega, notação, 34 fig., 36  
 ômega maiúsculo, notação, 34 fig., 36  
 ômega minúsculo, notação, 38  
 on-line, problema da contratação, 93-95  
 on-line, problema da envoltória convexa, 756 ex.  
 ON-LINE-MAXIMUM, 93  
 ON SEGMENT, 741  
 OPTIMAL-BST, 289  
 OR, função ( $\vee$ ), 779  
 OR, porta, 779  
 ordem  
     colunas, 576  
     de um nó em uma floresta de conjuntos disjuntos, 404, 408, 413 ex.  
     de um número em um conjunto ordenado, 242  
     de uma matriz, 576, 578-579 ex.  
     em árvores de ordem estatística, 244-245, 246 ex.  
     linhas, 576  
     total, 576  
 ordem  
     de um grupo, 686  
     linear, 851  
     parcial, 850  
     total, 851  
 ordem, estatísticas, 147-157  
     dinâmicas, 242-247  
 ordem de crescimento, 20  
 ordem dinâmica, estatística, 242-247  
 ordem estatística, árvore, 242-247  
     consultando, 249 ex.

ordenação, rede, 557  
     AKS, 570  
     baseada na ordenação por inserção, 558 ex.  
     baseada na ordenação por intercalação, 566-568  
     bitônica, 561-564  
     ímpar-par, 568 pr.  
     profundidade, 567 ex.  
     tamanho, 558 ex.  
 ordenada, árvore, 859  
 ordenada, lista ligada, 166  
     ver também ligada, lista  
 ordenado, par, 848  
 ordenando, 11-14, 21-28, 99-146  
     bubblesort, 29 pr.  
     bucket sort, 140-143  
     de pontos por ângulo polar, 743 ex.  
     de uma matriz, 568 ex.  
     em tempo linear, 135-143, 143 pr.  
     heapsort, 103-116  
     itens de comprimento variável, 144 pr.  
     lexicográfica, 217 pr.  
     limite inferior do caso médio para ordenação, 144 pr.  
     limites inferiores para, 133-136  
     nebulosa, 131 pr.  
     no local, 12, 100  
     ordenação de seleção, 20 ex.  
     ordenação de Shell, 31  
     ordenação por comparação, 133  
     ordenação por contagem, 136-137  
     ordenação por inserção, 8, 11-14  
     ordenação por intercalação, 8, 21-28  
     problema de, 3, 11, 99  
     quicksort, 117-132  
     radix sort, 137-139  
     rede para, ver ordenação, rede topológica, ver topológica,  
         ordenação  
     usando redes, ver ordenação, rede usando uma árvore de pesquisa binária, 212 ex.  
 orientada, versão de um grafo não orientado, 855  
 orientado, grafo, 853  
     caminho mais curto em, 459  
     caminhos mais curtos de todos os pares em, 490-508  
     caminhos mais curtos de única origem em, 459-489  
     cobertura de caminho de, 546 pr.  
     conexos isoladamente, 436 ex.  
     e caminhos mais longos, 763  
     fecho transitivo de, 500  
     hamiltoniano, ciclo de, 764  
     PERT, diagrama, 470, 470 ex.  
     quadrado de, 421 ex.  
     restrição, grafo, 477  
     semiconectado, 442 ex.  
     transposição de, 421 ex.  
     viagem de Euler de, 443 pr., 763  
     ver também circuito; orientado, grafo acíclico; grafo; rede  
 orientado, grafo acíclico (grafo), 856  
     algoritmo para caminhos mais curtos de única origem, 468-470  
     e arestas traseiras, 436  
     e grafos componentes, 439  
     e problema do caminho hamiltoniano, 776 ex.  
     ordenação topológica de, 436

orientado, segmento, 739  
 origem, 422, 460, 510, 512  
 origem, 739  
 ortonormal, 608-609  
 OS-KEY-RANK, 246 ex.  
 OS-RANK, 244  
 OS-SELECT, 244  
 ótima, árvore de pesquisa binária, 285-290, 295  
 ótima, cobertura de vértices, 807-808  
 ótima, solução, 617  
 ótima, subestrutura  
     de árvores de pesquisa binária, 287-288  
     de caminhos mais curtos, 460-461, 492, 497  
     de caminhos mais curtos não ponderados, 274  
     de códigos de Huffman, 312-313  
     de matróides ponderados, 317  
     de multiplicação de cadeias de matrizes, 267-268  
     de programação de linha de montagem, 261-262  
     de seqüencial de atividade, 371-373  
     de subseqüências comuns mais longas, 282  
     do problema da mochila 0-1, 305-306  
     do problema da mochila fracionária, 305-306  
     em algoritmos gulosos, 305  
     em programação dinâmica, 272-276  
 otimização, problema, 259, 764-765, 768  
     algoritmos de aproximação para, 806-831  
     e problemas de decisão, 765  
 ótimo, subconjunto de um matróide, 315-316  
 ótimo, valor de objetivo, 617  
 ou (or), em pseudocódigo, 15  
  
 P (classe de complexidade), 764, 768, 771-772, 773 ex.  
 pacote, embalagem, 754  
 padrão, emparelhamento, *ver* cadeias, emparelhamento  
 padrão, forma, 612-613, 616-619  
 padrão em emparelhamento de cadeias, 717  
     não sobrepostas, 730 ex.  
 página em um disco, 350-351 pr.  
 paginando, 16-17  
 pai, 858-859  
     em uma árvore de primeiro na extensão, 423  
 Pan, método para multiplicação de matrizes, 585 ex.  
 par, ordenado, 847-848  
 par mais distante, problema, 749-750  
 par mais próximo, localizando, 756-760  
 par único, caminho mais curto, 273-274  
     como um programa linear, 622-623  
 parâmetro, 15  
     custos da passagem, 68 pr.  
 parcial, ordem, 850  
 PARENT, 104  
 parênteses, teorema, 431  
 partição de conjunto, problema, 802 ex.  
 partição de um conjunto, 847-848, 850

particionamento, algoritmo, 118-120  
     aleatório, 124  
     em torno da mediana de 3 elementos, 128 ex.  
 PARTITION, 118  
 Pascal, triângulo, 867 ex.  
 PATH, 765, 771  
 percurso de uma árvore, *ver* árvore, percurso  
 perdido, filho, 860  
 perfeito, emparelhamento, 529 ex.  
 perfeito, hash, 198-201  
 permutação, 852  
     aleatória, 81-83  
     aleatória uniforme, 74-75, 81  
     de um conjunto, 864  
     inversão de bits, 340 pr., 666  
     Josephus, 255 pr.  
     no local, 82  
 permutação, matriz, 574, 578 ex.  
     decomposição de LUP de, 596 ex.  
 permutação, rede, 569 pr.  
 PERMUTE-BY-CYCLIC, 84 ex.  
 PERMUTE-BY-SORTING, 81  
 PERMUTE-WITH-ALL, 84 ex.  
 PERMUTE-WITHOUT-IDENTITY, 84 ex.  
 persistente, estrutura de dados, 236 pr., 346  
 PERSISTENT-TREE-INSERT, 236 pr.  
 PERT, diagrama, 470, 470 ex.  
 peso  
     de um caminho, 459  
     de um corte, 823 ex.  
     de uma aresta, 421  
     médio, 488 pr.  
 peso, função  
     em um matróide ponderado, 315  
     para um grafo, 421  
 peso balanceado, árvore, 241, 342 pr.  
 peso mínimo, árvore espalhada, *ver*  
     mínima, árvore espalhada  
 peso mínimo, cobertura de vértices, 820-823  
 peso negativo, arestas, 461  
 peso negativo, ciclo  
     e caminhos mais curtos, 461  
     e relaxação, 485 ex.  
     e restrições de diferença, 477  
 pesquisa, árvore, *ver* pesquisa  
     balanceada, árvore; pesquisa binária, árvore; B, árvore; exponencial, árvore de pesquisa; intervalos, árvore; ótima, árvore de pesquisa binária; ordem estatística, árvore; vermelho-preto, árvore; oblíqua, árvore; 2-3, árvore; 2-3-4, árvore  
 pesquisa binária, árvore, 204-219  
     árvore AA, 241  
     árvore AVL, 237-238 pr.  
     árvore de bode expiatório, 241  
     árvore de  $k$  vizinhos, 241  
     árvore de peso balanceado, 241  
     árvore oblíquas, 241  
     chave máxima de, 208  
     chave mínima de, 208  
     com chaves iguais, 216 pr.  
     construída aleatoriamente, 213-216, 217-218 pr.  
     consultando, 207-210  
     e treaps, 237-238 pr.  
     eliminação de, 211-212  
     inscrição em, 210  
     ótima, 285-291, 295  
     para ordenação, 212 ex.  
     pesquisando, 207-208  
     predecessor em, 208-209  
     sucessor em, 208-209  
     *ver também* vermelho-preto, árvore pesquisando  
     em árvores B, 354-355  
     em árvores de intervalos, 251-253  
     em árvores de pesquisa binária, 207-208  
     em árvores vermelho-preto, 222-223  
     em listas compactas, 177 pr.  
     em listas ligadas, 167  
     em tabelas de endereço direto, 180  
     em tabelas hash de endereço aberto, 192-193  
     em tabelas hash encadeadas, 193  
     em um arranjo não ordenado, 95 pr.  
     para um intervalo exato, 254 ex.  
     pesquisa binária, 28 ex.  
     pesquisa linear, 15 ex.  
     problema de, 15 ex.  
 phi, função, 685  
 pilha, 163-164  
     em busca de Graham, 750  
     implementação de lista ligada de, 169 ex.  
     implementada por filas, 166 ex.  
     no armazenamento secundário, 363 pr.  
     operações analisadas pelo método de contabilidade, 328-329  
     operações analisadas pelo método potencial, 331  
     operações analisadas por análise agregada, 325-327  
     para execução de procedimento, 130-131 pr.  
 pior caso, tempo de execução, 20, 36  
 PISANO-DELETE, 396 pr.  
 piso, função, ( $\lfloor \cdot \rfloor$ ), 40  
     em teorema mestre, 65-68  
 piso, instrução, 16-17  
 pivô  
     em decomposição LU, 592  
     em programação linear, 629-631, 637 ex.  
     em quicksort, 118  
 PIVOT, 630  
 podando um heap de Fibonacci, 396-397 pr.  
 podando uma lista, 824  
 podar e pesquisar, método, 749-750  
 polígono, 743 ex.  
     estrelado, 756 ex.  
     núcleo de, 756 ex.  
 polilogaritmicamente limitado, 43  
 polinomialmente limitado, 41  
 polinomialmente relacionado, 769  
 polinômio, 41, 651  
     adição de, 651  
     assintótico, comportamento de, 46 pr.  
     avaliação de, 30 pr., 653, 657 ex., 669-670 pr.  
 interpolação por, 653, 658 ex.  
 multiplicação de, 652, 655-657, 669 pr.  
 representação de coeficientes de, 653

representação de valor de ponto de, 653-654  
 Pollard, heurística rho, 710-713, 713 ex.  
**POLLARD-RHO**, 710  
 ponderada, cobertura de vértices, 820-822  
 ponderada, heurística de união, 402  
 ponderada, mediana, 155 pr.  
 ponderado, matróide, 315-318  
 ponderado, emparelhamento bipartido, 397  
 ponderado, problema de cobertura de conjuntos, 828 pr.  
 ponte, 442 pr.  
 ponteiro, 15  
 ponto de evento, programação, 745  
 ponto extremo  
     de um intervalo, 249  
     de um segmento de linha, 739  
 ponto flutuante, tipo de dados, 16-17  
 ponto interior, método, 615  
 ponto mais próximo, heurística, 815 ex.  
**POP**, 164  
 porta, 779  
 posição, 179-180  
 posicional, árvore, 861  
 positiva simétrica, matriz definida, 601-603  
 positivo, fluxo, 510  
 pós-ordem, percurso de árvore, 205  
 potência  
     de um elemento, módulo  $n$ , 693-698  
      $k$ -ésima, 677 ex.  
 potencial, função, 330  
     para limites inferiores, 343  
 potencial, método, 330-333  
     para algoritmo de Knuth-Morris-Pratt, 733-734  
     para algoritmo genérico de push-relabel, 536-537  
     para contadores binários, 331-332  
     para estruturas de dados de conjuntos disjuntos, 408-412  
     para heaps de Fibonacci, 384-386, 389-390, 393  
     para heaps mínimos, 333 ex.  
     para operações de pilhas, 330-331  
     para reestruturação de árvores vermelho-preto, 342 pr.  
     para tabelas dinâmicas, 335-336, 338-340  
 potencial de uma estrutura de dados, 330  
 potências, conjunto, 848  
 potências, série, 69 pr.  
**Pr{ } (distribuição de probabilidades)**, 868  
 prazo final, 319  
 predecessor  
     em árvores B, 359-360 ex.  
     em árvores de caminhos mais curtos, 462  
     em árvores de ordem estatística, 248 ex.  
     em árvores de pesquisa binária, 208-209  
     em árvores vermelho-preto, 222  
         em listas ligadas, 166  
**PREDECESSOR**, 160  
 predecessor, subgrafo  
     em busca em largura, 426  
     em busca em profundidade, 429  
 em caminhos mais curtos de todos os pares, 491  
 em caminhos mais curtos de única origem, 462  
 predecessora, matriz, 491  
 preempção, 321 pr.  
 prefixo  
     de uma cadeia ( $\square$ ), 718  
     de uma sequência, 281  
 prefixo, código, 308  
 prefixo, função, 730-732  
 pré-ordem, percurso de árvore, 205  
 pré-ordenação, 759  
 presente, embalagem, 754-755  
 preto, altura, 222  
 preto, vértice, 422, 429  
**Prim**, algoritmo, 452-454  
     com pesos de arestas inteiros, 455 ex.  
     com uma matriz de adjacência, 454 ex.  
     em algoritmo de aproximação para o problema do caixeiro-viajante, 811  
     implementado com um heap de Fibonacci, 454  
     implementado com um heap mínimo, 454  
     para grafos esparsos, 456 pr.  
     semelhança com o algoritmo de Dijkstra, 452, 473-474  
**primal**, programa linear, 638  
 primeiro a entrar, primeiro a sair, 163  
     *ver também* fila  
 primeiro ajuste, heurística, 828 pr.  
 primeiro na extensão, árvore, 423, 427  
 primeiro na profundidade, árvore, 429  
 primeiro na profundidade, floresta, 429  
 primeiro-adiantada, forma, 319  
 primo, número, 674  
     densidade de, 702-703  
 primos, função de distribuição, 703  
 principal, agrupamento, 193  
 principal, memória, 349  
 principal, raiz da unidade, 658-659  
 princípio de inclusão e exclusão, 849 ex.  
**PRINT-ALL-PAIRS-SHORTEST-PATH**, 491  
**PRINT-INTERSECTING-SEGMENTS**, 748 ex.  
**PRINT-LCS**, 284  
**PRINT-OPTIMAL-PARENTS**, 271, 271 ex.  
**PRINT-STATIONS**, 265  
 prioridade máxima, fila, 111-112  
 prioridade mínima, fila, 111-112, 114 ex.  
     na construção de códigos de Huffman, 310  
     no algoritmo de Dijkstra, 473  
     no algoritmo de Prim, 454  
 prioridades, fila, 111-114  
     com extrações monotônicas, 116  
     fila de prioridade máxima, 111-112  
     fila de prioridade mínima, 111-112, 114 ex.  
     implementação de heap de, 111-114  
     na construção de códigos de Huffman, 310  
     no algoritmo de Dijkstra, 473  
     no algoritmo de Prim, 454  
     *ver também* pesquisa binária, árvore; binomial, heap; Fibonacci, heap  
 probabilidade, 868-873  
 probabilidades, distribuição, 868  
 probabilidades, distribuição discreta, 869  
 probabilística, análise, 74-75, 85-95  
     de algoritmo de aproximação para satisfabilidade de MAX-3-CNF, 820  
 de algoritmo de Rabin-Karp, 724  
 de altura de uma árvore de pesquisa binária construída aleatoriamente, 213-216  
 de bolas e caixas, 88-89  
 de bucket sort, 140-143, 143 ex.  
 de colisões, 185 ex., 201 ex.  
 de comparação de arquivos, 724 ex.  
 de contagem probabilística, 95 pr.  
 de envoltória convexa em uma distribuição de envoltória esparsa, 964 pr.  
 de hash com encadeamento, 183-184  
 de hash de endereço aberto, 195-197, 197 ex.  
 de hash perfeito, 199-201  
 de hash universal, 188-190  
 de heurística rho de Pollard, 711-713  
 de inserção em uma árvore de pesquisa binária com chaves iguais, 216 pr.  
 de limite de prova mais longo para hash, 201 pr.  
 de limite de tamanho de posição para encadeamento, 202 pr.  
 de limite inferior do caso médio para ordenação, 145-146 pr.  
 de paradoxo do aniversário, 85-87  
 de particionamento, 123 ex., 128 ex., 129 pr., 131 pr.  
 de pesquisa em uma lista compacta, 177 pr.  
 de pontos de ordenação por distância a partir da origem, 143 ex.  
 de profundidade de nó médio em uma árvore de pesquisa binária construída aleatoriamente, 217 pr.  
 de quicksort, 126-128, 129 pr., 131 pr., 216 ex.  
 de seleção aleatória, 149-152  
 de sequências, 89-92  
 do problema da contratação, 78-79  
 do teste de caráter primo de Miller-Rabin, 707-709  
 e algoritmos aleatórios, 79-81  
 e entradas médias, 19-20  
 probabilística, contagem, 95 pr.  
 problema  
     abstrato, 767-768  
     computacional, 3-4  
     concreto, 768  
     decisão, 765, 767-768  
     intratável, 763  
     otimização, 259, 764, 767-768  
     solução para, 4, 768  
     tratável, 763  
 problema de satisfabilidade de fórmula, 786-787  
 problema do caixeiro-viajante  
     algoritmo de aproximação para, 810-815  
     caráter NP-completo de, 799  
     com desigualdade de triângulos, 811-813

- euclidiano bitônico, 291 pr.  
 gargalo, 814-815 ex.  
 sem desigualdade de triângulos, 813-814  
 procedimento, 4, 11-12  
 produto  
     cartesiano, 848  
     cruzado, 738-739  
     de matrizes, 574-575, 578-579 ex.  
     de polinômios, 652  
     externo, 575  
     interno, 575  
     regra de, 864  
 profissional, lutador, 428 ex.  
 profundidade  
     de árvore de recursão de quicksort, 123 ex.  
     de SORTER, 567 ex.  
     de um nó em uma árvore enraizada, 859  
     de uma pilha, 131 pr.  
     de uma rede de comparação, 557-558  
     de uma rede de ordenação, 558 ex.  
     média, de um nó em uma árvore de pesquisa binária construída aleatoriamente, 217 pr.  
 programa, contador, 781-782  
 programação, 829 pr.  
     ponto de evento, 745  
 programação (agendamento), 295 pr., 321 pr., 804 pr., 829 pr.  
 programação, *ver* dinâmica,  
     programação; linear, programação  
 programação de máquina paralela,  
     problema, 829 pr.  
 programação dinâmica, método, 259-295  
     comparação com algoritmos gulosos, 273-274, 280 ex., 299-300, 304, 305-307  
     e memorização, 278-280  
     elementos de, 272-280  
     para algoritmo de Floyd-Warshall, 497-500  
     para algoritmo de Viterbi, 294 pr.  
     para árvores de pesquisa binária ótimas, 285-291  
     para caminhos mais curtos de todos os pares, 492-500  
     para distância de edição, 291 pr.  
     para fecho transitivo, 500-502  
     para impressão esmerada, 291 pr.  
     para multiplicação de cadeias de matrizes, 266-272  
     para problema de mochila de 0-1, 307 ex.  
     para problema euclidiano bitônico do caixeiro-viajante, 260 pr.  
     para programação, 295 pr.  
     para programação de linha de montagem, 260-265  
     para seleção de atividade, 303 ex.  
     para subseqüência comum mais longa, 281-285  
     reconstruindo uma solução ótima em, 278  
     sobrepondo subproblemas em, 276-278  
     subestrutura ótima em, 272-276  
     programação linear, relaxação, 821  
     programação linear inteira, problema, 616, 649 pr., 802 ex.  
     pronto, vértice, 429
- próprio, ancestral, 858  
 próprio, descendente, 858  
 próprio, subconjunto ( $\subset$ ), 846  
 próprio, subgrupo, 686  
 prova, 192, 201 pr.  
 prova, seqüência, 192  
 pseudocódigo, 11, 14-15  
 pseudo-inverso, 605  
 PSEUDOPRIME, 704  
 pseudoprímo, 704  
 pública, chave, 697, 699-700  
 PUSH  
     pilha, operação, 194  
     push-relabel, operação, 531  
 push-relabel, algoritmo, 529-546  
     algoritmo genérico, 532-538  
     algoritmo relabel-to-front, 538-546  
     com uma fila de vértices de sobrecarga, 546 ex.  
     descarregando um vértice de sobrecarga de altura máxima, 546 ex.  
     heurística de intervalos para, 546 ex.  
     operações básicas em, 531-532  
     para encontrar um emparelhamento bipartido máximo, 537 ex.  
 push-relabel, algoritmo genérico, 532-538
- quadrada, matriz, 572  
 quadrado de um grafo orientado, 421 ex.  
 quadrática, função, 19  
 quadrática, sondagem, 194, 202 pr.  
 quadrático, resíduo, 714 pr.  
 quantil, 154 ex.  
 quicksort, 117-132  
     análise de, 120-124, 125-128  
     análise do pior caso de, 125  
     boa implementação do pior caso de, 154 ex.  
     caso médio, análise de, 125-128  
     com método de mediana de 1, 130 pr.  
     descrição de, 117-120  
     em comparação com a ordenação por inserção, 123 ex.  
     em comparação com a radix sort, 139  
     profundidade de pilha de, 130 pr.  
     uso de ordenação por inserção em, 128 ex.  
     versão aleatória de, 124, 129 pr.  
     versão de extremidade recursiva de, 130 pr.
- QUICKSORT, 118  
 QUICKSORT', 130 pr.  
 quociente, 673-674  
 R (conjunto de números reais), 845  
 Rabin-Karp, algoritmo, 721-725  
 RABIN-KARP-MATCHER, 723-724  
 radix sort, 137-139  
     comparação com quicksort, 139  
 RADIX-SORT, 139  
 raio, 743 ex.  
 raiz  
     de uma árvore, 858  
     de unidade, 658  
 raiz, árvore, 217 pr.  
 raiz quadrada, módulo primo, 714-715 pr.  
 raízes, lista  
     de um heap binomial, 368  
     de um heap de Fibonacci, 383
- RAM, *ver* acesso aleatório, máquina  
 RANDOM, 75, 75 ex.  
 RANDOMIZED-HIRE-ASSISTANT, 80  
 RANDOMIZED-PARTITION, 124  
 RANDOMIZED-QUICKSORT, 124, 216 ex.  
     relação com árvores de pesquisa binária construídas aleatoriamente, 217 pr.  
 RANDOMIZED-SELECT, 186  
 RANDOMIZE-IN-PLACE, 82-83  
 RANDOM-SEARCH, 95-96 pr.  
 RB-DELETE, 231  
 RB-DELETE-FIXUP, 232  
 RB ENUMERATE, 249 ex.  
 RB-INSERT, 225  
 RB-INSERT-FIXUP, 226  
 RB-JOIN, 237 pr.  
 reais, números (R), 845  
 reconstruindo uma solução ótima em programação dinâmica, 278  
 recorrência, 25, 50-72  
     solução pelo método de Akra-Bazzi, 72  
     solução pelo método de árvore de recursão, 54-58  
     solução pelo método de substituição, 51-54  
     solução pelo método mestre, 59-61  
 recorrência, equação, *ver* recorrência  
 recortando, em um heap de Fibonacci, 391-392  
 recuo em pseudocódigo, 14  
 recursão, 21  
 recursão, árvore, 27-28, 54-58  
     e o método de substituição, 57-58  
     em prova de teorema mestre, 61-63  
     para ordenação por intercalação, 280 ex.  
 RECURSIVE-ACTIVITY-SELECTOR, 301  
 RECURSIVE-FFT, 662  
 RECURSIVE-MATRIX-CHAIN, 277  
 rede  
     admissível, 538-540  
     bitônica, ordenação, 561-564  
     comparação, 555-559  
     fluxo, *ver* fluxo, rede  
     ímpar-par, intercalação, 568 pr.  
     ímpar-par, ordenação, 568 pr.  
     ordenação, 565-570  
     para intercalação, 564-566  
     permutação, 569 pr.  
     residual, 515-517  
     transposição, 568 pr.  
     redefinindo o peso  
         em caminhos mais curtos de todos os pares, 503  
         em caminhos mais curtos de única origem, 486 pr.  
     redução, algoritmo, 766, 777  
     redução, função, 777  
     redutibilidade, 777-779  
     referência a um conjunto de arestas, 447  
     reflexiva, relação, 849  
     reflexividade de notação assintótica, 39  
     região, viável, 612  
     registro (dados), 99  
     regra da soma, 863  
     regra do produto, 863-864

rejeição  
     por um algoritmo, 771  
     por uma automação finita, 726  
**RELABEL**, 532  
**relabel**, operação (em algoritmos push-relabel), 532, 535  
**RELABEL-TO-FRONT**, 543  
**relabel-to-front**, algoritmo, 538-546  
**relação**, 849-851  
**relativamente**, primo, 676  
**RELAX**, 464  
**relaxação**,  
     de uma aresta, 463-465  
     programação linear, 821  
**relaxação**, 619  
**relaxação de caminho**, propriedade, 465, 482  
**relaxada**, árvore vermelho-preto, 223 ex.  
**relaxada**, forma, 612-613, 619-621  
     unicidade de, 635  
**relaxada**, variável, 619  
**relaxado**, heap, 396-397  
**repeat**, em pseudocódigo, 14  
**repetida**, elevação ao quadrado  
     para caminhos mais curtos de todos os pares, 494-496  
     para elevar um número a uma potência, 696  
**repetida**, função, 48 pr.  
**repetida**, função logaritmo, 44-45  
**REPETITION-MATCHER**, 737 pr.  
**representação de um conjunto**, 398  
**RESET**, 330 ex.  
**residual**, aresta, 516  
**residual**, capacidade, 515, 517  
**residual**, rede, 515-517  
**resíduo**, 40, 673-674, 714 pr.  
**resto**, 40, 673-674  
**resto**, instrução, 16-17  
**restrição**, 616  
     desigualdade, 616-617, 618  
     diferença, 476  
     igualdade, 479 ex., 617, 618  
     linear, 612  
     não negatividade, 616, 618  
     rígida, 627  
         violação de, 627  
**restrição**, grafo, 477-478  
**retângulo**, 254 ex.  
**retorno**, instrução, 16-17  
**rho**, heurística, 710-713, 713 ex.  
**RIGHT**, 104  
**RIGHT-ROTATE**, 224  
**rígida**, restrição, 627  
**rotação**  
     cíclica, 736 ex.  
     em uma árvore vermelho-preto, 223-225  
**rotacional**, varredura, 749, 750-754  
**RSA**, sistema de criptografia de chave pública, 697-702  
  
**saída**  
     de um algoritmo, 3  
     de um circuito combinacional, 780  
     de uma porta lógica, 779  
**saída**, fio, 556  
**saída**, grau, 854  
**saída**, seqüência, 556  
**saltos**, lista, 241  
**SAME-COMPONENT**, 400  
**SAT**, 786

satélite, dados, 99, 159  
**satisfabilidade**, 780, 785-788, 819-820, 823 ex.  
**satisfação**, atribuição, 780, 785  
**satisfatória**, fórmula, 764, 785  
**saturação**, empurrão, 531, 536  
**saturada**, aresta, 531  
**Schur**, complemento, 591, 602  
**SCRAMBLE-SEARCH**, 96 pr.  
**SEARCH**, 160  
**secreta**, chave, 697, 699-700  
**secundária**, tabela hash, 198  
**secundário**, agrupamento, 194  
**secundário**, armazenamento  
     árvore de pesquisa para, 349-364  
     pilhas em, 363 pr.  
**segmento**, *ver* orientado, segmento;  
     linha, segmento  
**SEGMENTS-INTERSECT**, 741  
**segunda melhor**, árvore espalhada  
     mínima, 456 pr.  
**segura**, aresta, 446  
**seleção**  
     de atividades, *ver* seleção de  
         atividades, problema  
     e ordenações de comparação, 154  
     em árvores de ordem estatística,  
         243-244  
     em tempo linear esperado, 149-152  
     problema de, 147  
     tempo linear no pior caso, 152-155  
**seleção**, ordenação, 20 ex.  
**seleção de atividade**, problema, 297-303  
**SELECT**, 152-153  
**seletor**, vértice, 796  
**semiconectado**, grafo, 442 ex.  
**sentinela**, 22, 167-169, 222  
**seqüência** ( $\langle \rangle$ )  
     bitônica, 488 pr., 561  
     entrada, 556  
     finita, 852  
     infinita, 852  
     inversão em, 30 pr., 79-80 ex.  
     limpa, 562  
     prova, 192  
     saída, 556  
     seqüências, 89-92  
     série, 69 pr., 836-837  
     Shell, ordenação, 31  
**SHORTEST-PATH**, 765  
**símbolos**, tabela, 179, 186, 188  
**simetria** de notação de  $\Theta$ , 39  
**simetria** de transposição de notação assintótica, 39  
**simétrica**, diferença, 549-550 pr.  
**simétrica**, matriz, 573-574, 578 ex.  
**simétrica**, relação, 849-850  
**simples**, algoritmo para emparelhamento  
     de cadeias, 719-721  
**simples**, caminho, 854  
     mais longo, 274, 763  
**simples**, ciclo, 854  
**simples**, grafo, 855  
**simples**, hash uniforme, 183  
**simples**, polígono, 743 ex.  
**simplex**, 614  
**SIMPLEX**, 632  
**simplex**, algoritmo, 614, 626-638, 650  
**singular**, decomposição de valores, 608-609  
**singular**, matriz, 575

sistemas de equações lineares, 586-597, 607 pr.  
**sistemas de restrições de diferença**, 475-480  
**SLOW-ALL-PAIRS-SHORTEST-PATHS**, 494  
**sobre**, 852  
**sobrecarga**  
     de uma fila, 164  
     de uma pilha, 164  
**sobrecarregando** vértice, 530  
**sobrejeção**, 852  
**solução**  
     básica, 628  
     inviável, 616-617  
     ótima, 616-617  
     para um problema abstrato, 768  
     para um problema computacional, 4  
     para um problema concreto, 468  
     para um sistema de equações  
         lineares, 586  
         viável, 475, 613, 617  
**soma**  
     cartesiana, 658 ex.  
     de matrizes, 573-574  
     de polinômios, 651  
     fluxo, 514 ex.  
     infinita, 835  
     inserindo, 837-838  
     regra da, 863  
**soma** de subconjuntos, problema  
     algoritmo de aproximação para, 823-827  
     caráter NP-completo de, 800-802  
     com destino unário, 1802 ex.  
**somatório**, 835-844  
     em notação assintótica, 37, 836  
     fórmulas e propriedades de, 835-838  
     implícito, 512  
     limites, 838-844  
     linearidade de, 836  
**somatório**, lema, 659  
**somatório** implícito, notação, 512  
**somatórios**, desmembrando, 841-842  
**sombra** de um ponto, 756 ex.  
**sondagem**, *ver* linear, sondagem;  
     quadrática sondagem  
**SORTER**, 566, 567 ex.  
**sorvedor**, 421 ex., 510, 512  
**STACK-EMPTY**, 164  
**Stirling**, aproximação, 44  
**STOOGES-SORT**, 130 pr.  
**Strassen**, algoritmo, 579-585  
**STRONGLY-CONNECTED-COMPONENTS**, 439  
**subárvore**, 858-859  
     mantendo os tamanhos de, em  
         árvores de ordem estatística,  
         245-247  
**subcadeia**, 864  
**subcaminho**, 854  
**subcarga**  
     de uma fila, 164  
     de uma pilha, 164  
**subconjunto**  
     hereditários, família de, 314  
     independentes, família de, 314  
**subconjunto** ( $\subseteq$ ), 846  
**subconjunto** máximo em um matróide, 315

- subdeterminado, sistema de equações lineares, 587  
 subgrafo, 855  
     predecessor, *ver* predecessor, subgrafo  
 subgrafo de predecessor, propriedade, 464-465, 484  
 subgrupo, 686-687  
 sub-rotina  
     chamando, 15, 16, 17 n.  
     executando, 17 n.  
 subseqüência, 280  
 SUBSET-SUM, 799  
 substituição, método, 51-54  
     e árvores de recursão, 57-58  
 subtração de matrizes, 574  
 subtrair, instrução, 16  
 SUCCESSOR, 160  
 sucesso em uma experiência de Bernoulli, 878  
 sucessor  
     em árvores de ordem estatística, 248 ex.  
     em árvores de pesquisa binária, 208-209  
     em árvores vermelho-preto, 222-223  
     em listas ligadas, 166  
     localizando o  $i$ -ésimo, de um nó em uma árvore de ordem estatística, 246 ex.  
 sufixo ( $\sqsupseteq$ ), 718  
 sufixo, função, 726  
 superdeterminado, sistema de equações lineares, 587  
 superior, matriz triangular, 573  
 superior, mediana, 147  
 superorigem, 512  
 superpolinomial, tempo, 763  
 superpostos, intervalos, 249  
     localizando todos, 254 ex.  
     ponto de superposição máxima, 255 pr.  
 superpostos, lema de sufixos, 718-719  
 superpostos, retângulos, 254 ex.  
 superpostos, subproblemas, 276-278  
 supersorvedor, 512  
 suspensão, problema, 763  
 SVD, 609
- TABLE-DELETE, 338  
 TABLE-INSERT, 334  
 tamanho  
     da entrada de um algoritmo, 17, 672-673, 768-770  
     de um circuito combinacional booleano, 780-781  
     de um clique, 791  
     de um conjunto, 847-848  
     de uma árvore binomial, 366-367  
     de uma cobertura de vértices, 793, 807-808  
     de uma rede de comparação, 557  
     de uma rede de ordenação, 558 ex.  
     de uma subárvore em um heap de Fibonacci, 395  
 tarefa, 319  
 tarefas, programação, 319-321, 323 pr.  
 Tarjan, algoritmo de ancestrais menos comuns off-line de, 415 pr.  
 tautologia, 776 ex., 790-791 ex.  
 taxa de crescimento, 20  
 Taylor, séries, 218 pr.
- tempo, domínio, 651  
 tempo, *ver* execução, tempo  
 tempo de descoberta em busca em profundidade, 429-430  
 tempo polinomial, aceitação, 771  
 tempo polinomial, algoritmo, 673  
 tempo polinomial, computabilidade, 769  
 tempo polinomial, decisão, 771  
 tempo polinomial, esquema de aproximação, 807  
 tempo polinomial, redutibilidade ( $\leq_p$ ), 776-777, 784 ex.  
 tempo polinomial, resolvabilidade, 768  
 tempo polinomial, verificação, 773-776  
 tempo unitário, tarefa, 319  
 teorema fundamental de programação linear, 647  
 testes  
     de caráter primo, 702-709, 715  
     de caráter pseudoprímo, 704  
 teto, função ( $\lceil \cdot \rceil$ ), 40  
     em teorema mestre, 65-68  
 teto, instrução, 16-17  
 texto cifrado, 698-699  
 texto em emparelhamento de cadeias, 717  
 then, em pseudocódigo, 14  
 todos os pares, caminhos mais curtos, 460, 490-508  
     em grafos  $\varepsilon$ -densos, 507 pr.  
     algoritmo de Floyd-Warshall, 497-500  
     algoritmo de Johnson, 503-506  
     por multiplicação de matrizes, 492-497  
     por elevação ao quadrado repetida, 494-496  
 Toeplitz, matriz, 669 pr.  
 TOP, 750  
 topo de uma pilha, 163  
 topológica, ordenação, 436-438  
     no cálculo de caminhos mais curtos de única origem em um grafo, 468  
 TOPOLOGICAL-SORT, 436  
 total, árvore binária, 860, 861 ex.  
     relação com o código ótimo, 308-309  
 total, fluxo líquido, 511  
 total, fluxo positivo, 510  
 total, nó, 353  
 total, ordem, 851  
 total, ordem, 576  
 total, percurso de uma árvore, 812-813  
 totalmente entre parênteses, 266  
 totalmente polinomial, esquema de aproximação de tempo, 807  
     para o problema da soma de subconjuntos, 823-827  
     para o problema do clique máximo, 828 pr.  
 transformação discreta de Fourier, 660  
 transformação rápida de Fourier (FFT)  
     círculo para, 667  
     implementação iterativa de, 665-667  
     implementação recursiva de, 661-663  
     multidimensional, 669 pr.  
     usando aritmética modular, 670 pr.  
 transição, função, 725-726, 729-730  
 transitiva, relação, 849  
 TRANSITIVE-CLOSURE, 501  
 transitividade de notação assintótica, 30
- transitivo, fecho, 500-502  
     e multiplicação de matrizes booleanas, 600 ex.  
     de grafos dinâmicos, 507 pr.  
 transposição  
     conjugada, 600 ex.  
     de um grafo orientado, 421 ex.  
     de uma matriz, 420-421, 572  
 transposição, rede, 568 pr.  
 traseira, aresta, 433-434, 437  
 tratabilidade, 763  
 travessia de uma árvore, *ver* árvore, percurso  
 treap, 238 pr.  
 TREAP-INSERT, 238 pr.  
 TREE-DELETE, 211, 231-232  
 TREE-INSERT, 210, 225  
 TREE-MAXIMUM, 208  
 TREE-MINIMUM, 208  
 TREE-PREDECESSOR, 209  
 TREE-SEARCH, 207  
 TREE-SUCCESSOR, 208-209  
 3-CNF, 788  
 3-CNF, satisfabilidade, 788-1791  
     algoritmo de aproximação para, 819-820  
     e satisfabilidade de 2-CNF, 764  
 3-CNF-SAT, 788  
 3-COLOR, 804 pr.  
 3-forma normal conjuntiva, 788  
 triangular, matriz, 573-574, 578 ex.  
 triângulos, desigualdade, 811  
     e arestas de peso negativo, 814 ex.  
     para caminhos mais curtos, 464-465, 480-481  
 tricotomia, intervalo, 249  
 tricotomia, propriedade de números reais, 39  
 triadiagonais, sistemas lineares, 607 pr.  
 triagonal, matriz, 573  
 trie, *ver* raiz, árvore  
 trilha, 350  
 TRIM, 825  
 trinado, transformação, 664 ex.  
 trivial, divisor, 673-674  
 troca, propriedade, 314  
 TSP, 799  
 tupla, 848
- último a entrar, primeiro a sair, 163  
     *ver* também pilha  
 um para um, emparelhamento, 852  
 um para um, função, 852  
 uma passagem, método, 416  
 unário, 769  
 união  
     de conjuntos ( $\cup$ ), 846  
     de conjuntos dinâmicos, *ver* unificando  
     de linguagens, 770-771  
 união por ordem, 404  
 única origem, caminhos mais curtos, 459-489  
     algoritmo de escalonamento de Gabow, 486 pr.  
     Bellman-Ford, algoritmo, 465-468  
     com caminhos bitônicos, 488 pr.  
     Dijkstra, algoritmo, 471-475  
     e caminhos mais longos, 763  
     e restrições de diferença, 475-480

- em grafos g-densos, 507 pr.  
 em um grafo acíclico orientado, 468-471  
 unicamente conectado, grafo, 436 ex.  
 unicamente ligada, lista, 166  
*ver também* ligada, lista  
 unidade (1), 673-674  
 unificando  
     heaps, 365  
     heaps 2-3-4, 379  
     heaps binomiais, 371-375  
     heaps de Fibonacci, 385-386  
     listas ligadas, 169  
 uniforme, distribuição de probabilidades, 869  
 uniforme, hash, 193  
 uniforme, permutação aleatória, 74-75, 81  
 UNION, 365, 399  
     implementação de floresta de conjuntos disjuntos de, 405  
     implementação de lista ligada de, 401-403, 403 ex.  
 unitária, matriz triangular inferior, 573  
 unitária, matriz triangular superior, 573  
 unitário, 848  
 unitário, vetor, 572  
 universal, sorvedor, 421 ex.  
 universal, hash, 188-191  
 universo, 847  
 válido, deslocamento, 717  
 valor  
     de um fluxo, 510  
     de uma função, 852  
     objetivo, 614, 617  
 valor de ponto, representação, 653-654  
 valor inteiro, fluxo, 527  
 van Emde Boas, estrutura de dados, 116, 346  
 Vandermonde, matriz, 579 ex.  
 Var[ ] (variância), 877  
 variância, 876  
     de uma distribuição binomial, 880-881  
     de uma distribuição geométrica, 878  
 várias mercadorias, fluxo, 625-623  
     custo mínimo, 626 ex.
- variável  
     aleatória, 873-877  
     básica, 620  
     deixando, 629  
     em pseudocódigo, 14  
     entrando, 629  
     não básica, 620  
     relaxada, 619  
*ver também* indicador, variável  
     aleatória  
 vários conjuntos, 845 n.  
 varredura, 743-749, 760 pr.  
 varredura, linha, 743  
 vazia, árvore, 1860  
 vazia, cadeia ( $\epsilon$ ), 718, 770  
 vazia, linguagem, 770-771  
 vazia, pilha, 164  
 vazio, conjunto, 845  
 Venn, diagrama, 847  
 verdade, atribuição, 780, 786  
 verdade, tabela, 779  
 verificação, 773-776  
     de árvores espalhadas, 458  
 verificação, algoritmo, 774  
 vermelho-preto, árvore, 220-240  
     altura de, 221  
     ampliação de, 248-249  
     chave máxima de, 222  
     chave mínima de, 222  
     comparação com árvores B, 354  
     e árvores 2-3-4, 354-355 ex.  
     eliminação de, 231-236  
     junção, 237 pr.  
     para determinar se quaisquer  
         segmentos de linhas se  
         interceptam, 745-746  
     para enumeração de chaves em um  
         intervalo, 249 ex.  
     pesquisando em, 222  
     predecessor em, 222  
     propriedades de, 220-223  
     reestruturando, 342-343 pr.  
     relaxada, 223 ex.  
     rotação em, 223-225  
     sucessor em, 222  
*ver também* intervalo, árvore; ordem  
     estatística, árvore  
 VERTEX-COVER, 794  
 vértice
- de um polígono, 742 ex.  
 em um grafo, 853  
 intermediário, 497  
 isolado, 854  
 ponto de articulação, 443 pr.  
 seletor, 796  
 vértices, cobertura, 794, 807-808, 820-823  
 vértices, conjunto, 853  
 vetor, 572, 575-576  
     convolução de, 653  
     produto cruzado de, 739  
     ortonormal, 609  
     no plano, 739  
 viabilidade, problema, 475, 648 pr.  
 viagem  
     bitônico, 291 pr.  
     de Euler, 443 pr., 763  
     de um grafo, 798  
 viável, programa linear, 616  
 viável, região, 613  
 viável, solução, 475, 613, 616  
 violação de uma restrição de igualdade, 627  
 virtual, memória, 16-17  
 vizinhança, 529 ex.  
 vizinho, 855-856  
 vizinhos, lista, 540  
 VLSI (very-large-scale integration – integração em escala muito grande), 70 n.  
 while, em pseudocódigo, 14  
 WITNESS, 705  
 Yen, aperfeiçoamento para o algoritmo de Bellman-Ford, 485  
 Young, quadro, 115  
 Z (conjunto de inteiros), 845  
 0-1, problema da mochila, 305-306, 307  
 0-1, problema de programação de inteiros, 802, 821  
 zero, matriz, 572  
 zero de um polinômio, módulo primo, 690  
 zero-um, princípio, 559-561, 564-565  
 $Z_n$  (classes de equivalência, módulo  $n$ ), 674