

Algoritmos

Sanjoy Dasgupta

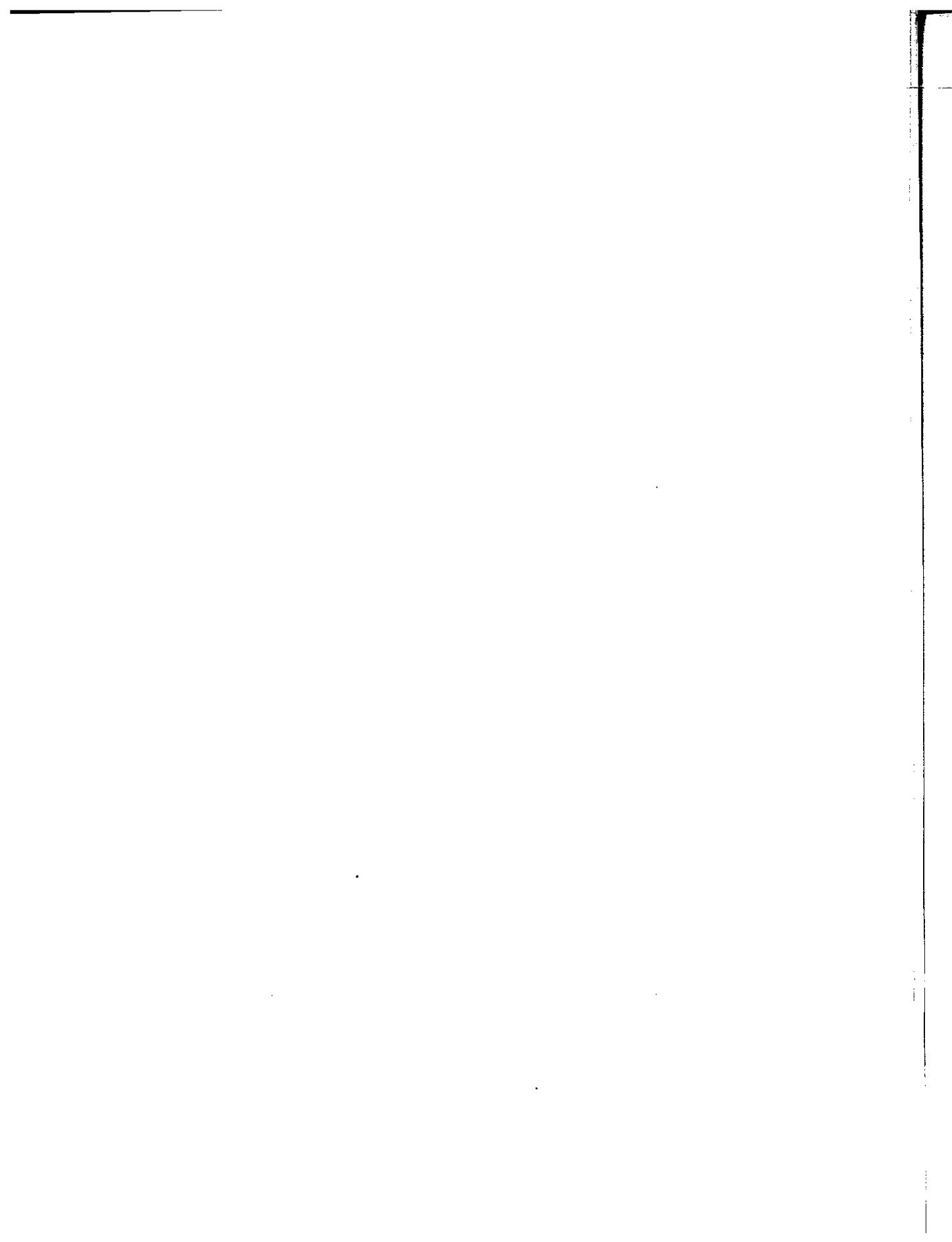
Universidade da Califórnia, San Diego

Christos Papadimitriou

Universidade da Califórnia em Berkeley

Umesh Vazirani

Universidade da Califórnia em Berkeley



Algoritmos

Sanjoy Dasgupta

Universidade da Califórnia, San Diego

Christos Papadimitriou

Universidade da Califórnia em Berkeley

Umesh Vazirani

Universidade da Califórnia em Berkeley

Tradutor Técnico

Prof. Dr. Guilherme Albuquerque Pinto

Departamento de Ciência da Computação

UNB - Universidade de Brasília



Bangcoc Bogotá Beijing Caracas Cidade do México
Cingapura Lisboa Londres Madri Milão Montreal Nova Delhi
Santiago São Paulo Seul Sydney Taipé Toronto

Algoritmos

ISBN: 978-85-7726-032-4

A reprodução total ou parcial deste volume por quaisquer formas ou meios, sem o consentimento, por escrito, da editora é ilegal e configura apropriação indevida dos direitos intelectuais e patrimoniais dos autores.

© 2009 by McGraw-Hill Interamericana do Brasil Ltda.

Todos os direitos reservados.

Av. Brigadeiro Faria Lima , 201, 17º andar

São Paulo – SP – CEP 05426-100

© 2009 by McGraw-Hill Interamericana Editores, S.A. de C.V.

Todos os direitos reservados.

Prol. Paseo de la Reforma 1015 Torre A Piso 17, Col. Desarrollo Santa Fé, Delegación Alvaro Obregón
México 01376, D.F., México

Tradução da edição em inglês de *Algorithms*

Publicada pela McGraw-Hill/Irwin, uma unidade de negócios da McGraw-Hill Companies. Inc, 1221 Avenue of the Americas,
New York, NY 10020.

© 2008 by The McGraw-Hill Companies, Inc.

ISBN: 978-0-07-352340-8

Coordenadora Editorial: *Guacira Simonelli*

Editora de Desenvolvimento: *Gisélia Costa*

Produção Editorial: *Nilcéia Esposito*

Supervisora de Pré-imprensa: *Natália Toshiyuki*

Preparação de Texto: *Mônica de Aguiar*

Diagramação: *ERJ Composição Editorial*

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Dasgupta, Sanjoy

Algoritmos / Sanjoy Dasgupta, Christos

Papadimitriou, Umesh Vazirani ; [tradução técnica Guilherme A. Pinto]. — São Paulo : McGraw-Hill, 2009.

Título original: *Algorithms*.

Bibliografia.

ISBN 978-85-7726-032-4

1. Algoritmos 2. Dados - Estruturas (Ciência da computação) I. Papadimitriou, Christos.
II. Vazirani, Umesh. III. Título.

08-04904

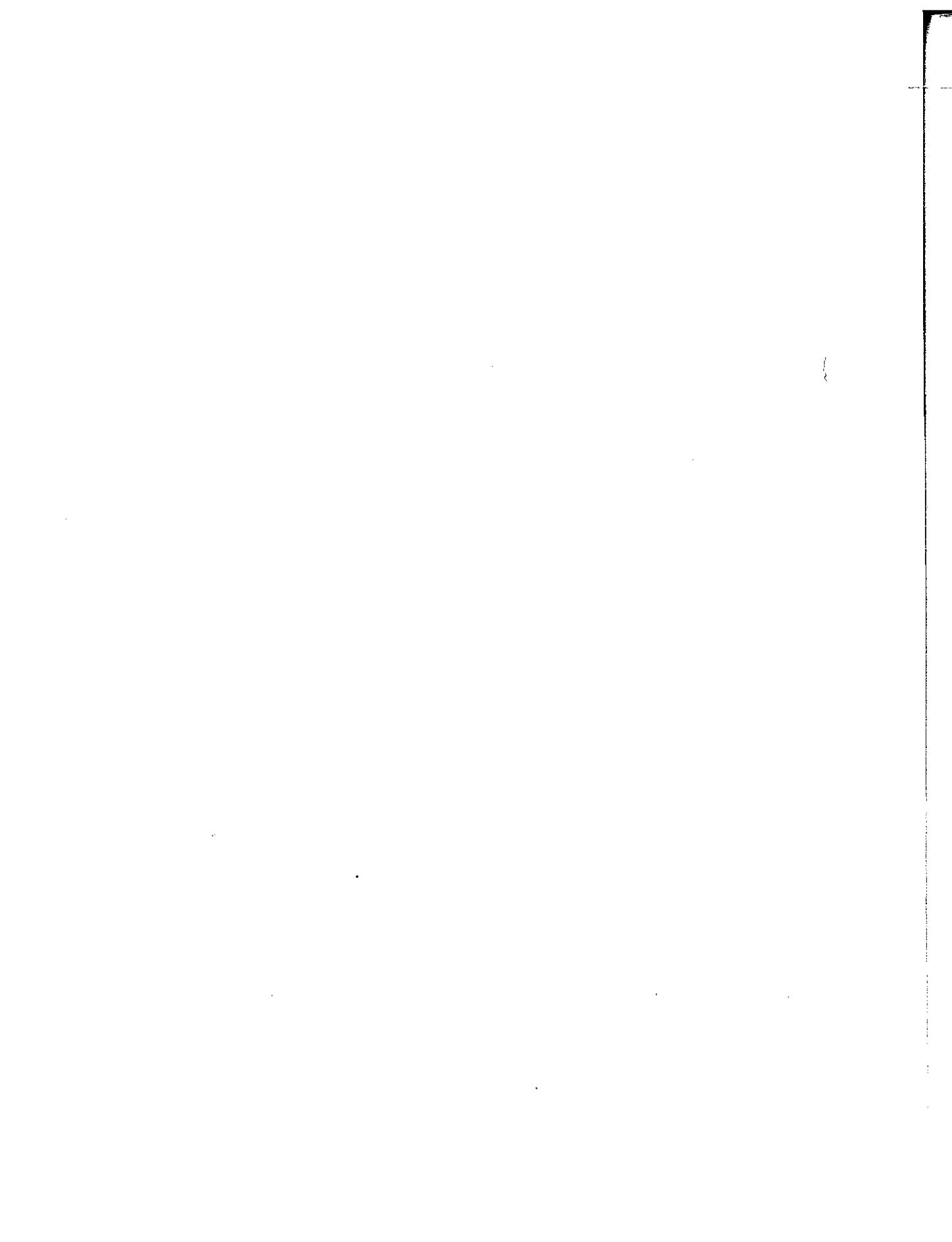
CDD-005.1

Índices para catálogo sistemático:

1. Algoritmos : Computadores : Programação :
Processamento de dados 005.1

A McGraw-Hill tem forte compromisso com a qualidade e procura manter laços estreitos com seus leitores. Nossa principal objetivo é oferecer obras de qualidade a preços justos, e um dos caminhos para atingir essa meta é ouvir o que os leitores têm a dizer. Portanto, se você tem dúvidas, críticas ou sugestões, entre em contato conosco — preferencialmente por correio eletrônico (mh_brasil@mcgraw-hill.com) — e nos ajude a aprimorar nosso trabalho. Teremos prazer em conversar com você. Em Portugal use o endereço servico_clientes@mcgraw-hill.com.

**A nossos estudantes e professores,
e a nossos pais.**

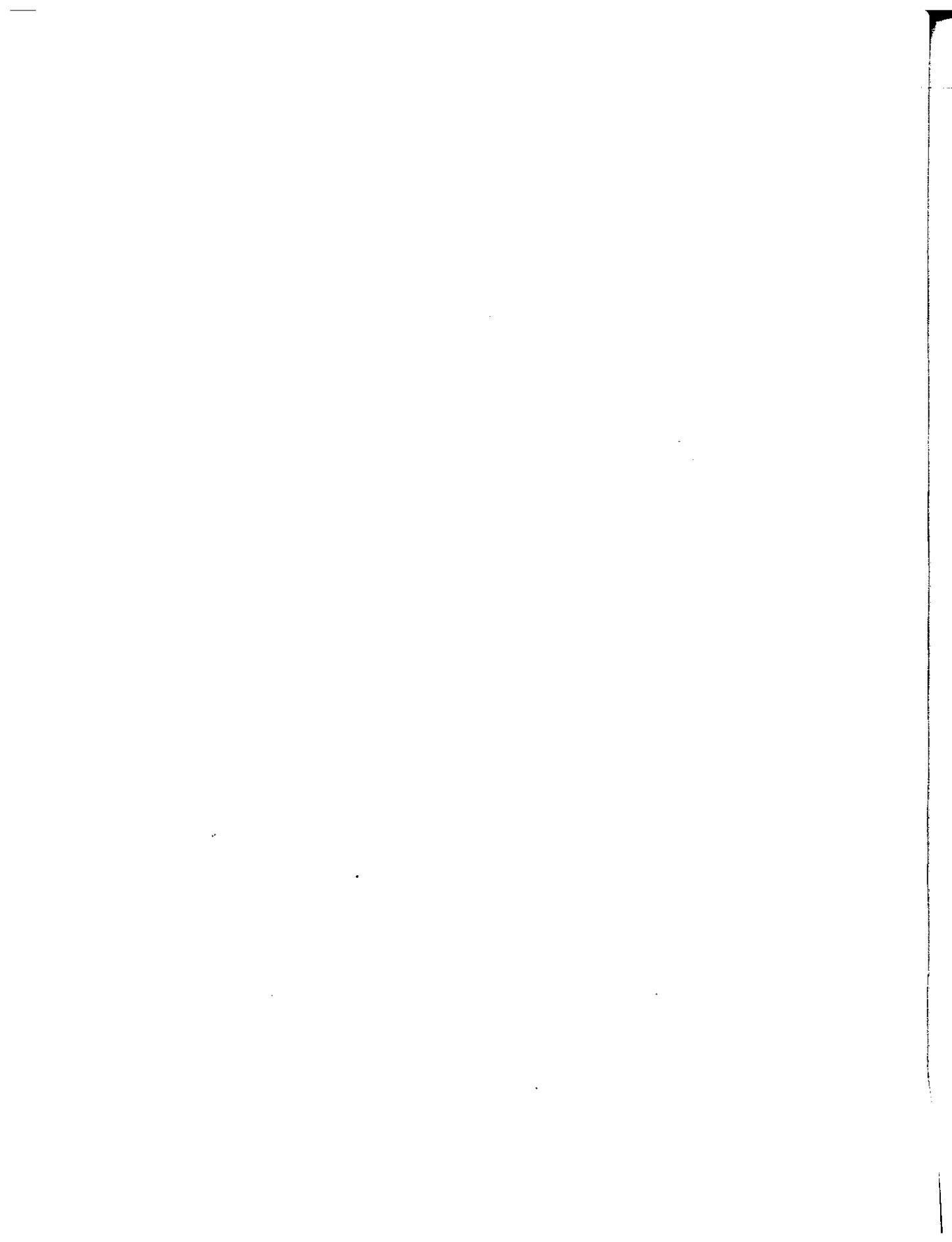


Sumário

<i>Prefácio</i>	xiii
0 Prólogo	1
0.1 Livros e algoritmos	1
0.2 Fibonacci entra em cena	2
0.3 A notação O	6
Exercícios	8
1 Algoritmos sobre números	11
1.1 Aritmética básica	11
1.2 Aritmética modular	16
1.3 Teste de primalidade	23
1.4 Criptografia	30
1.5 Espalhamento universal	35
Exercícios	38
Algoritmos randomizados: um capítulo virtual	29
2 Algoritmos de divisão-e-conquista	45
2.1 Multiplicação	45
2.2 Relações de recorrência	49
2.3 Mergesort	50
2.4 Medianas	53
2.5 Multiplicação de matrizes	56
2.6 A transformada rápida de Fourier	57
Exercícios	70
3 Decomposição de grafos	80
3.1 Por que grafos?	80
3.2 Busca em profundidade em grafos não-direcionados	83
3.3 Busca em profundidade em grafos direcionados	87
3.4 Componentes fortemente conexas	91
Exercícios	95
4 Caminhos em grafos	104
4.1 Distâncias	104
4.2 Busca em largura	105
4.3 Comprimentos nas arestas	107
4.4 O algoritmo de Dijkstra	108

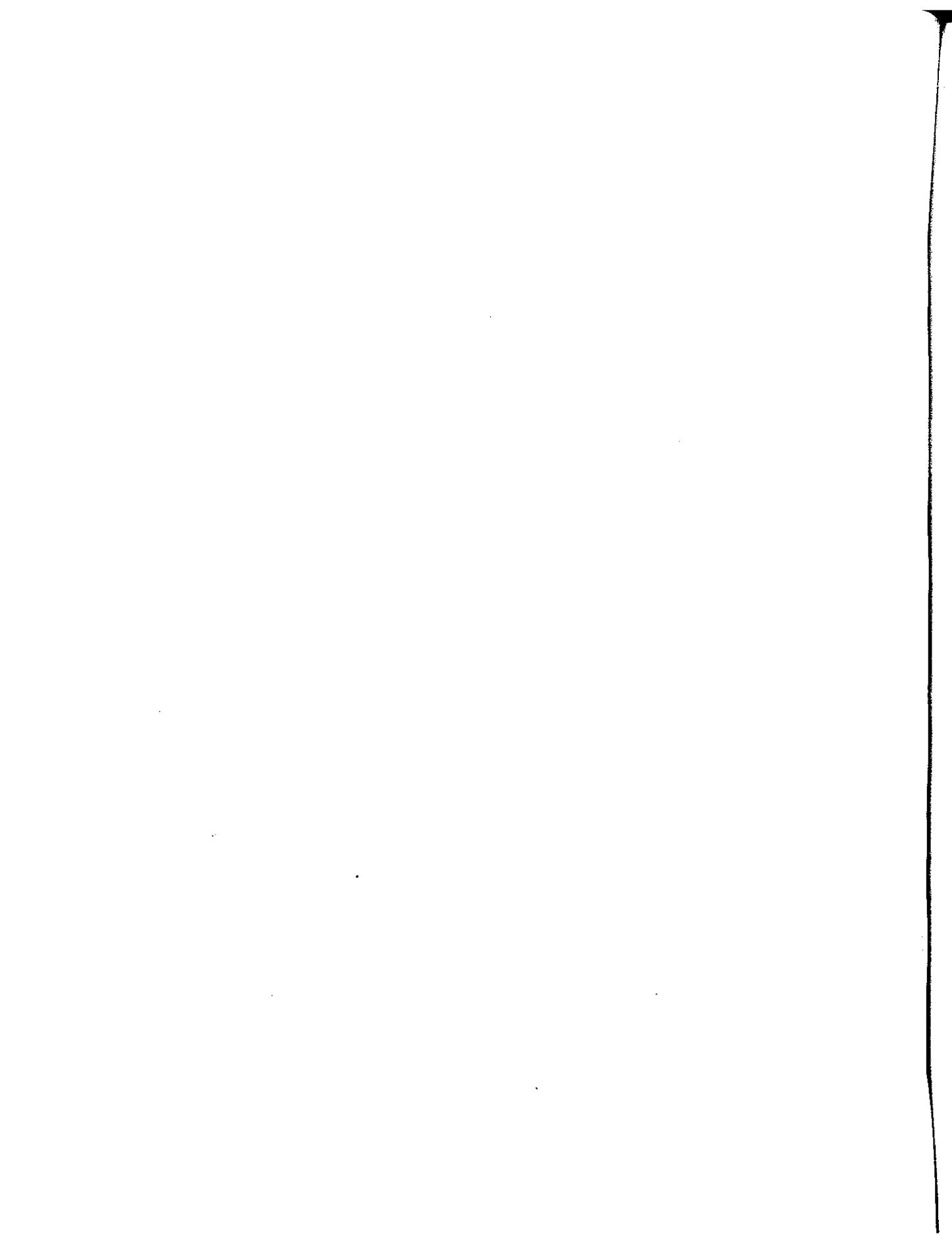
4.5	Implementações de fila de prioridades	113
4.6	Caminhos mínimos na presença de arestas negativas	115
4.7	Caminhos mínimos em dags	119
	Exercícios	120
5	Algoritmos gulosos	127
5.1	Árvores espalhadas mínimas	127
5.2	Codificação de Huffman	138
5.3	Fórmulas Horn	144
5.4	Cobertura de vértices	145
	Exercícios	148
6	Programação dinâmica	156
6.1	Caminhos mínimos em dags, revisitados	156
6.2	Subseqüência crescente mais longa	157
6.3	Distância de edição	159
6.4	Mochila	164
6.5	Multiplicação de cadeias de matrizes	168
6.6	Caminhos mínimos	171
6.7	Conjuntos independentes em árvores	175
	Exercícios	177
7	Programação linear e reduções	188
7.1	Uma introdução à programação linear	188
7.2	Fluxo em redes	198
7.3	Emparelhamento bipartido	205
7.4	Dualidade	206
7.5	Jogos de soma-zero	209
7.6	O algoritmo simplex	213
7.7	Pós-escrito: avaliação de circuito	221
	Exercícios	222
8	Problemas NP-completos	232
8.1	Problemas de busca	232
8.2	Problemas NP-completos	243
8.3	As reduções	247
	Exercícios	264
9	Lidando com NP-completude	271
9.1	Busca exaustiva inteligente	272
9.2	Algoritmos de aproximação	276
9.3	Heurísticas de busca local	285
	Exercícios	293
10	Algoritmos quânticos	297
10.1	Qubits, superposição e medida	297
10.2	O plano	301

10.3 A transformada de Fourier quântica	303
10.4 Periodicidade	305
10.5 Circuitos quânticos	307
10.6 Fatoração como periodicidade	310
10.7 O algoritmo quântico para fatoração	311
Exercícios	314
Notas históricas e leitura adicional	317
Índice remissivo	319



Quadros

Bases e logaritmos	12
Complemento de dois	17
O seu número de CPF é primo?	24
Ei, isso foi teoria de grupos!	27
Números de Carmichael	28
Algoritmos randomizados: um capítulo virtual	29
Busca binária	50
Uma cota inferior $n \log n$ para ordenação	52
O comando <code>sort</code> do Unix	56
Por que multiplicar polinômios?	59
A lenta disseminação de um algoritmo rápido	70
Qual é o tamanho do seu grafo?	82
Varrendo rapidamente	94
Qual heap é melhor?	114
Árvores	129
Um algoritmo randomizado para o corte mínimo	140
Entropia	143
Recursão? Não, obrigado	160
Programação?	161
Subproblemas comuns	165
De ratos e homens	166
Memorização	169
Sobre tempo e memória	175
Um truque mágico chamado dualidade	192
Reduções	196
Notação matriz-vetor	198
Visualizando dualidade	209
Eliminação gaussiana	219
Programação linear em tempo polinomial	220
A história de Sissa e Moore	233
Por que P e NP?	244
As duas maneiras de usar reduções	246
Problemas insolúveis	263
Emaranhamento	300
A transformada de Fourier de um vetor periódico	306
Preparando uma superposição periódica	312
Implicações para ciência da computação e física quântica	314



Prefácio

Este livro evoluiu nestes últimos dez anos, com base em um conjunto de notas de aulas desenvolvidas durante a aplicação do curso de Algoritmos para graduação em Berkeley e U.C. San Diego.

Nossa maneira de ministrar este curso também evoluiu tremendamente ao longo desses anos, em parte para considerar a experiência dos nossos alunos (habilidade formal subdesenvolvida, fora da área de programação), em parte para refletir o amadurecimento da área em geral, da forma como nós o percebemos. As notas se cristalizaram, aos poucos, numa narrativa, e nós estruturamos o curso progressivamente para enfatizar a “linha da história” implícita na progressão do material. Como resultado, os tópicos foram cuidadosamente selecionados e agrupados. Nenhuma tentativa foi feita com o intuito de dar uma cobertura enciclopédica e isto nos deixou livres para incluir tópicos tradicionalmente pouco enfatizados ou omitidos em muitos livros sobre Algoritmos.

Jogando com o que nossos estudantes são fortes (que é comum na maioria dos graduandos em Ciência da Computação), em vez de elaborar provas formais, destilamos em cada caso a idéia matemática clara que faz o algoritmo funcionar. Em outras palavras, enfatizamos o rigor em oposição ao formalismo, e descobrimos que os estudantes eram muito mais receptivos ao rigor matemático. É essa progressão de idéias claras que ajuda a tecer a história.

Uma vez que você pense sobre Algoritmos dessa maneira, faz sentido iniciar pelo princípio histórico de tudo, no qual os personagens são familiares e os contrastes dramáticos: números, primalidade e fatoração. Este é o assunto da Parte I do livro, que também inclui o sistema de criptografia RSA, algoritmos de divisão-e-conquista para multiplicação inteira, ordenação e busca da mediana, bem como a transformada rápida de Fourier.

Há ainda, outras três partes: A parte II, é a seção mais tradicional do livro, e concentra-se em estruturas de dados e grafos – o contraste aqui é entre a estrutura intricada dos problemas subjacentes e os curtos e claros pseudocódigos que os solucionam. Professores que desejem lecionar um curso mais tradicional podem simplesmente começar pela Parte II, que é autocontida, (uma vez lido o prólogo), e, então, cobrir a Parte I conforme as necessidades. Nas Partes I e II introduzimos certas técnicas (tais como gulosa e divisão-e-conquista) que funcionam para tipos especiais de problemas. A Parte III lida com os “pesos-pesados” do negócio, técnicas que são poderosas e gerais: programação dinâmica (uma abordagem inovadora ajuda a esclarecer esta tradicional barreira para os estudantes) e programação linear (um tratamento limpo e intuitivo do algoritmo simplex, dualidade e reduções ao problema básico). A última parte, a Parte IV, aborda maneiras de lidar com problemas difíceis: NP-completude, várias heurísticas, bem como algoritmos quânticos, talvez o mais avançado e moderno tópico. Como por vezes acontece, terminamos a história exatamente onde começamos, com o algoritmo quântico de Shor para fatoração.

O livro inclui ainda três linhas secundárias, na forma de três séries de “quadros” separados, fortalecendo a narrativa (e considerando variações nas necessidades e interesses dos estudantes), mas mantendo o fluxo intacto: peças que provêem contexto histórico; descrições de como os algoritmos explicados são usados na prática (com ênfase em aplicações na Internet); e excursões para os matematicamente sofisticados.

Muitos de nossos colegas deram contribuições cruciais para este livro. Somos agradecidos pelo *feedback* de Dimitris Achlioptas, Dorit Aharonov, Mike Clancy, Jim Demmel, Monika Henzinger, Mike Jordan, Milena Mihail, Gene Myers, Dana Randall, Satish Rao, Tim Roughgarden, Jonathan Shewchuk, Martha Sideri, Alistair Sinclair e David Wagner, que testaram versões anteriores. Satish Rao, Leonard Schulman e Vijay Vazirani que deram forma à exposição de várias seções importantes. Gene Myers, Satish Rao, Luca Trevisan, Vijay Vazirani e Lofti Zadeh que proveram exercícios. E, finalmente, há os estudantes de U.C. Berkeley e, mais tarde, U.C. San Diego, que inspiraram este projeto e que o viram por suas muitas encarnações.

Capítulo 0

Prólogo

Olhe ao seu redor. Computadores e redes estão em todo lugar, possibilitando uma intrincada teia de complexas atividades humanas: educação, comércio, diversão, pesquisa, manufatura, gerenciamento de saúde, comunicação humana, até mesmo guerra. Dos dois principais pilares tecnológicos dessa proliferação incrível, um é óbvio: o impressionante passo com o qual os avanços em microeletrônica e projeto de circuitos integrados têm nos trazido hardware cada vez mais rápido.

Este livro conta a história do outro empreendimento intelectual que crucialmente abastece a revolução da computação: *algoritmos eficientes*. É uma história fascinante.

Aproximem-se e ouçam com atenção.

0.1 Livros e algoritmos

Duas idéias mudaram o mundo. Em 1448, na cidade alemã de Mainz, um ourives chamado Johann Gutenberg descobriu uma maneira de imprimir livros juntando peças metálicas móveis. A alfabetização se espalhou, a Idade Média terminou, o intelecto humano foi liberado, ciência e tecnologia triunfaram, a Revolução Industrial aconteceu. Muitos historiadores dizem que devemos tudo isso à tipografia. Imagine um mundo no qual somente uma elite pudesse ler essas linhas! Mas outros insistem em que o desenvolvimento-chave não foi a tipografia, mas os *algoritmos*.



Johann Gutenberg
1398–1468

© Corbis

Hoje estamos tão acostumados a escrever números em decimal, que é fácil esquecer que Gutenberg escreveria o número 1448 como MCDXLVIII. Como você soma dois numerais romanos? O que é MCDXLVIII + DCCCXII? (E apenas tente pensar em multiplicá-los.) Mesmo um homem inteligente como Gutenberg provavelmente sabia apenas somar e subtrair pequenos números usando seus dedos; para qualquer coisa mais complicada ele tinha de consultar um especialista em ábaco.

O sistema decimal, inventado na Índia por volta de 600 d.C., foi uma revolução no raciocínio quantitativo: usando apenas dez símbolos, mesmo números muito grandes podiam ser escritos de maneira compacta, e aritmética podia ser feita eficientemente sobre eles seguindo passos elementares. Entretanto, essas idéias levaram um longo tempo para serem difundidas, impedidas por barreiras tradicionais de linguagem, distância e ignorância. O meio mais influente de transmissão acabou sendo um livro-texto, escrito em árabe no século IX por um homem que vivia em Bagdá. Al Khwarizmi estabeleceu os métodos básicos para adicionar, multiplicar e dividir números — até mesmo extrair a raiz quadrada e calcular os dígitos de π . Esses procedimentos eram precisos, não ambíguos, mecânicos, eficientes, corretos — em suma, eram *algoritmos*, um termo cunhado para homenagear o sábio homem, depois que o sistema decimal foi finalmente adotado na Europa, após muitos séculos.

Desde então, o sistema decimal posicional e seus algoritmos numéricos desempenharam um papel enorme na civilização ocidental. Eles possibilitaram a ciência e a tecnologia; aceleraram a indústria e o comércio. E quando, muito depois, o computador foi projetado, ele incorporou explicitamente o sistema posicional nos seus *bits*, palavras e unidade aritmética. Em todo lugar, cientistas se ocuparam em desenvolver algoritmos mais e mais complexos para todo tipo de problemas e inventar novas aplicações — por fim, mudando o mundo.

0.2 Fibonacci entra em cena

O trabalho de Al Khwarizmi não poderia ter chegado ao Ocidente não fosse pelos esforços de um homem: o matemático italiano do século XV Leonardo Fibonacci, que enxergou o potencial do sistema posicional e trabalhou duro para desenvolvê-lo mais e divulgá-lo.

Hoje Fibonacci é mais amplamente conhecido por sua famosa seqüência de números

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

cada um é a soma dos dois antecessores imediatos. Mais formalmente, os números de Fibonacci F_n são gerados pela regra simples

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n > 1 \\ 1 & \text{se } n = 1 \\ 0 & \text{se } n = 0. \end{cases}$$

Nenhuma outra seqüência de números foi estudada tão extensivamente, ou aplicada a mais áreas: biologia, demografia, arte, arquitetura, música, para citar apenas algumas. E, com as potências de 2, ela é a seqüência favorita da ciência da computação.

De fato, os números de Fibonacci crescem *quase* tão rapidamente quanto as potências de 2: por exemplo, F_{30} é mais que um milhão e F_{100} já tem 21 dígitos de comprimento! Em geral, $F_n \approx 2^{0.694n}$ (veja o Exercício 0.3).

Mas qual é o valor preciso de F_{100} ou de F_{200} ? O próprio Fibonacci certamente gostaria de saber tais coisas. Para respondermos, precisamos de um algoritmo para computar o n -ésimo número de Fibonacci.



Leonardo de Pisa (Fibonacci)
1170-1250

© Corbis

Um algoritmo exponencial

Uma idéia é implementar cegamente a definição recursiva de F_n . Aqui está o algoritmo resultante, na notação de “pseudocódigo” usada em todo este livro:

```
função fib1(n)
  se n = 0 : retorna 0
  se n = 1 : retorna 1
  retorna fib1(n - 1) + fib1(n - 2)
```

Quando temos um algoritmo, existem três perguntas que sempre fazemos sobre ele:

1. Ele é correto?
2. Quanto tempo ele toma, em função de n ?
3. E, será que podemos fazer melhor?

A primeira questão não vale a pena considerar aqui, pois esse algoritmo é precisamente a definição de Fibonacci para F_n . Mas a segunda requer uma resposta. Seja $T(n)$ o número de *passos de computação* necessários para computar $\text{fib1}(n)$, o que podemos dizer sobre essa função? Para os valores iniciais, se n é menor que 2, o procedimento pára quase imediatamente, após apenas um par de passos. Portanto,

$$T(n) \leq 2 \text{ para } n \leq 1.$$

Para valores maiores de n , há duas chamadas recursivas de `fib1`, tomando tempo $T(n - 1)$ e $T(n - 2)$, respectivamente, mais três passos de computação (checagem do valor de n e a adição final). Portanto,

$$T(n) = T(n - 1) + T(n - 2) + 3 \text{ para } n > 1.$$

Compare isso à relação de recorrência de F_n : vemos imediatamente que $T(n) \geq F_n$.

Isto é notícia muito ruim: o tempo de execução do algoritmo cresce tão rapidamente quanto os próprios números de Fibonacci! $T(n)$ é exponencial em n , o que implica que o algoritmo é impraticavelmente lento, exceto para valores muito pequenos de n .

Sejamos um pouco mais concretos sobre quão ruim é o tempo exponencial. Para computar F_{200} , o algoritmo `fib1` executa $T(200) \geq F_{200} \geq 2^{138}$ passos elementares de computação. Quanto tempo na verdade isso toma depende, claro, do computador usado. Hoje, o computador mais rápido do mundo é o NEC Earth Simulator, que executa 40 trilhões de passos por segundo. Mesmo nessa máquina, `fib1(200)` tomaria pelo menos 2^{92} segundos. Isso significa que, se começarmos a execução agora, ela ainda estaria rodando muito depois de o Sol se transformar numa estrela gigante vermelha.

Mas a tecnologia está melhorando rapidamente — a velocidade dos computadores tem dobrado mais ou menos a cada 18 meses, um fenômeno às vezes chamado de *lei de Moore*. Com esse crescimento extraordinário, talvez `fib1` rode muito mais rápido nas máquinas do próximo ano. Vejamos — o tempo de execução de `fib1(n)` é proporcional a $2^{0,694n} \approx (1,6)^n$, portanto computar F_{n+1} toma 1,6 vez mais tempo do que F_n . E, segundo a lei de Moore, os computadores ficam mais ou menos 1,6 vez mais rápidos a cada ano. Portanto, se pudermos computar F_{100} em tempo razoável na tecnologia deste ano, no próximo ano conseguiremos computar F_{101} . E no ano seguinte, F_{102} . E assim por diante: apenas mais um número de Fibonacci a cada ano! Tal é a praga do tempo exponencial.

Em suma, nosso algoritmo recursivo ingênuo é correto, mas irremediavelmente inefficiente. Será que podemos fazer melhor?

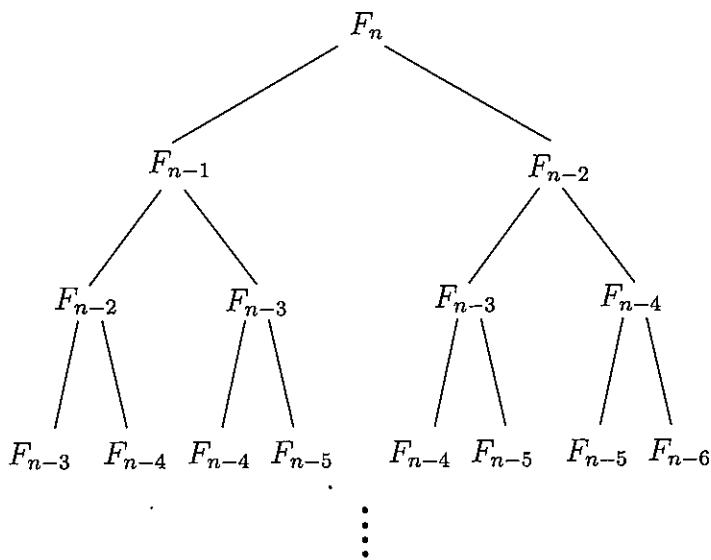
Um algoritmo polinomial

Tentemos entender por que `fib1` é tão lento. A Figura 0.1 mostra a cascata de chamadas recursivas disparada por uma única chamada a `fib1(n)`. Note que muitas computações são repetidas!

Um esquema mais sensato salvaria os resultados intermediários — os valores F_0, F_1, \dots, F_{n-1} — tão logo eles se tornem conhecidos.

```
função fib2(n)
  se n = 0: retorna 0
  crie um vetor f[0...n]
  f[0] = 0, f[1] = 1
  para i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
  retorna f[n]
```

Figura 0.1 A proliferação de chamadas recursivas em `fib1`.



Como no caso de `fib1`, a correção desse algoritmo é auto-evidente porque ele usa diretamente a definição de F_n . Quanto tempo ele toma? O loop interno consiste em um único passo de computação executado $n - 1$ vezes. Portanto, o número de passos usados por `fib2` é *linear* em n . De exponencial, baixamos para *polinomial*, um enorme avanço em tempo de execução. Agora é perfeitamente razoável computar F_{200} ou mesmo $F_{200.000}$ ¹.

Como veremos em todo o livro, o algoritmo certo faz toda a diferença.

Uma análise mais cuidadosa

Nesta discussão até agora, temos contado o número de *passos básicos de computação* executados por algoritmo e pensado neles como se tomassem uma quantidade constante de tempo. Isso é uma simplificação muito útil. Além disso, o conjunto de instruções do processador tem uma variedade de primitivas básicas — desvio, armazenamento na memória, comparação de números, aritmética simples e assim por diante — e, em vez de distinguir entre essas operações elementares, é muito mais conveniente juntá-las em uma única categoria.

Mas ao revermos nosso tratamento para os algoritmos de Fibonacci, fomos liberais demais com o que consideramos um passo básico. É razoável tratar adição como um único passo de computação se números pequenos estão sendo adicionados, digamos, números de 32 bits. Mas o n -ésimo número de Fibonacci tem comprimento de cerca de

¹para melhor avaliar a importância dessa dicotomia entre algoritmos exponenciais e polinomiais, o leitor pode se adiantar e ler a *história de Sissa e Moore*, no Capítulo 8.

$0,694n$, e isso pode exceder bastante 32, quando n cresce. Operações aritméticas em números arbitrariamente grandes não podem ser realizadas em um passo único, de tempo constante. Precisamos verificar nossas estimativas anteriores de tempo de execução e torná-las mais honestas.

Veremos no Capítulo 1 que a adição de dois números de n -bits toma tempo proporcional mais ou menos a n ; isso não é muito difícil de entender se você pensa no procedimento para adição da escola, que opera um dígito a cada vez. Portanto `fib1`, que realiza cerca de F_n adições, na verdade usa um número de *passos básicos* proporcional mais ou menos a nF_n . Da mesma maneira, o número de passos tomados por `fib2` é proporcional a n^2 , ainda polinomial em n e, portanto, exponencialmente superior a `fib1`. Essa correção na análise de tempo de execução não diminui nosso avanço.

Mas, será que podemos fazer melhor do que fib2? De fato podemos: veja o Exercício 0.4.

0.3 A notação O

Acabamos de ver como a negligência na análise de tempos de execução pode levar a um nível inaceitável de imprecisão no resultado. Mas o perigo oposto também está presente: é possível ser preciso *demais*. Uma análise perspicaz é sempre baseada nas simplificações certas.

Expressar o tempo de execução em termos de *passos básicos de computação* já é uma simplificação. Inclusive, o tempo gasto por um tal passo depende crucialmente do processador em questão e mesmo de detalhes, tais como a estratégia de *cache* (que resulta em tempos sutilmente diferentes de uma execução para próxima). Levar em conta as minúcias da arquitetura é um pesadelo, uma tarefa tão complexa que leva a um resultado não-generalista de um computador para outro. Faz mais sentido, portanto, buscar uma caracterização simples, independente da máquina, para a eficiência de um algoritmo. Para esse fim, sempre expressaremos o tempo de execução contando o número de passos básicos de computação como uma função do tamanho da entrada.

Essa simplificação leva a uma outra. Em vez de reportar que um algoritmo toma, digamos, $5n^3 + 4n + 3$ passos em uma entrada de tamanho n , é muito mais fácil deixar de fora termos de menor ordem, tais como $4n$ e 3 (que se tornam insignificantes à medida que n cresce), e mesmo o detalhe do coeficiente 5 do termo de maior ordem (os computadores serão 5 vezes mais rápidos de qualquer forma em poucos anos), e apenas dizer que o algoritmo toma tempo $O(n^3)$ (pronunciado “ozão de n^3 ”).

É chegada a hora de definir essa notação precisamente. Assim, pense em $f(n)$ e $g(n)$ como os tempos de execução de dois algoritmos sobre entradas de tamanho n .

Sejam $f(n)$ e $g(n)$ duas funções de inteiros positivos em reais positivos. Dizemos que $f = O(g)$ (que significa que “ f não cresce mais rápido do que g ”) se existe uma constante $c > 0$ tal que $f(n) \leq c \cdot g(n)$.

Dizer $f = O(g)$ está em leve analogia a dizer “ $f \leq g$ ”. Difere da noção usual de \leq por causa da constante c , de forma que, por exemplo, $10n = O(n)$. Tal constante também nos permite desconsiderar o que acontece para pequenos valores de n . Por exemplo,

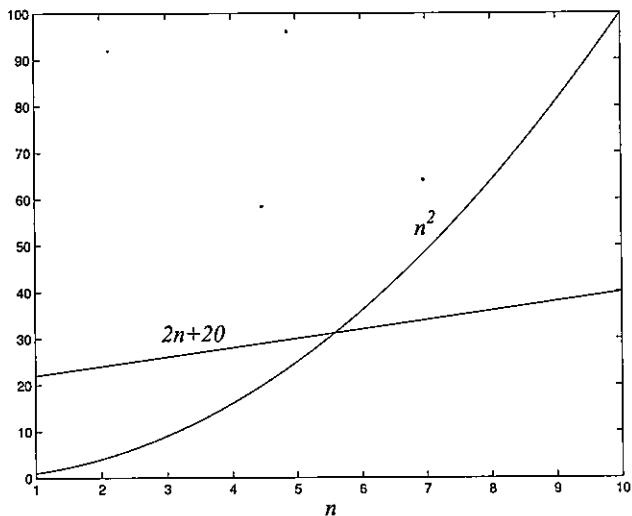
suponha que estejamos escolhendo entre dois algoritmos para determinada tarefa computacional. Um toma $f_1(n) = n^2$ passos, enquanto o outro toma $f_2(n) = 2n + 20$ passos (Figura 0.2). Qual é melhor? Bem, isso depende do valor de n . Para $n \leq 5$, n^2 é menor; depois disso, $2n + 20$ é claramente o vencedor. Nesse caso, f_2 escala muito melhor à medida que n cresce e, portanto, é superior.

Essa superioridade é capturada pela notação O : $f_2 = O(f_1)$, porque

$$\frac{f_2(n)}{f_1(n)} = \frac{2n + 20}{n^2} \leq 22$$

para todo n ; por outro lado, $f_1 \neq O(f_2)$, pois a razão $f_1(n)/f_2(n) = n^2/(2n + 20)$ pode tornar-se arbitrariamente grande e, portanto, nenhuma constante c fará a definição funcionar.

Figura 0.2 Qual tempo de execução é melhor?



Agora outro algoritmo aparece, um que usa $f_3(n) = n + 1$ passos. Isso é melhor do que f_2 ? Certamente, mas apenas por um fator constante. A discrepância entre $2n + 20$ e $n + 1$ é mínima comparada ao enorme vão entre n^2 e $2n + 20$. Para concentrarmo-nos na visão geral, tratamos funções como equivalentes se elas diferem apenas por constantes multiplicativas.

Retornando à definição de O , vemos que $f_2 = O(f_3)$:

$$\frac{f_2(n)}{f_3(n)} = \frac{2n + 20}{n + 1} \leq 20,$$

e, claro, $f_3 = O(f_2)$, desta vez com $c = 1$.

Assim como $O(\cdot)$ é análogo a \leq , podemos definir análogos de \geq e $=$ como se segue:

$$f = \Omega(g) \text{ significa } g = O(f)$$

$$f = \Theta(g) \text{ significa } f = O(g) \text{ e } f = \Omega(g).$$

No exemplo anterior, $f_2 = \Theta(f_3)$ e $f_1 = \Omega(f_3)$.

A notação O nos permite ter uma idéia do todo. Quando confrontados com uma função complicada como $3n^2 + 4n + 5$, simplesmente a substituímos por $O(f(n))$, onde $f(n)$ é tão simples quanto possível. Neste exemplo particular, usamos $O(n^2)$, porque a porção quadrática da soma domina o restante. Aqui estão algumas regras criteriosas que, omitindo termos dominados, ajudam a simplificar funções:

1. Constantes multiplicativas podem ser omitidas: $14n^2$ se torna n^2 .
2. n^a domina n^b se $a > b$: por exemplo, n^2 domina n .
3. Qualquer exponencial domina qualquer polinomial: 3^n domina n^5 (domina até mesmo 2^n).
4. Da mesma maneira, qualquer polinomial domina qualquer logaritmo: n domina $(\log n)^3$. Isso também significa, por exemplo, que n^2 domina $n \log n$.

Não interprete mal esse cavalheirismo com constantes. Programadores e projetistas de algoritmos estão *muito* interessados em constantes e virariam noites para fazer um algoritmo rodar duas vezes mais rápido. Mas entender algoritmos no nível deste livro seria impossível sem a simplicidade fornecida pela notação O .

Exercícios

- 0.1. Em cada uma das seguintes situações, indique se $f = O(g)$, ou $f = \Omega(g)$ ou ambos (caso em que $f = \Theta(g)$).

$f(n)$	$g(n)$
(a) $n - 100$	$n - 200$
(b) $n^{1/2}$	$n^{2/3}$
(c) $100n + \log n$	$n + (\log n)^2$
(d) $n \log n$	$10n \log 10n$
(e) $\log 2n$	$\log 3n$
(f) $10 \log n$	$\log(n^2)$
(g) $n^{1.01}$	$n \log^2 n$
(h) $n^2/\log n$	$n(\log n)^2$
(i) $n^{0.1}$	$(\log n)^{10}$
(j) $(\log n)^{\log n}$	$n/\log n$
(k) \sqrt{n}	$(\log n)^3$
(l) $n^{1/2}$	$5^{\log_2 n}$
(m) $n2^n$	3^n

(n)	2^n	2^{n+1}
(o)	$n!$	2^n
(p)	$(\log n)^{\log n}$	$2^{(\log_2 n)^2}$
(q)	$\sum_{i=1}^n i^k$	n^{k+1}

0.2. Mostre que, se c é um número real positivo, então $g(n) = 1 + c + c^2 + \dots + c^n$ é:

- (a) $\Theta(1)$ se $c < 1$.
- (b) $\Theta(n)$ se $c = 1$.
- (c) $\Theta(c^n)$ se $c > 1$.

Moral da história: em termos de Θ , a soma de uma série geométrica é simplesmente o primeiro termo se a série é estritamente decrescente, o último termo se a série é estritamente crescente, ou o número de termos se a série é invariável.

0.3. O números de Fibonacci F_0, F_1, F_2, \dots , são definidos pela regra

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

Neste problema vamos confirmar que a seqüência cresce exponencialmente e obter algumas cotas sobre o seu crescimento.

- (a) Use indução para provar que $F_n \geq 2^{0.5n}$ para $n \geq 6$.
- (b) Encontre uma constante $c < 1$ tal que $F_n \leq 2^{cn}$ para todo $n \geq 0$. Mostre que sua resposta está correta.
- (c) Qual é a maior constante c que você pode encontrar para a qual $F_n = \Omega(2^{cn})$?

0.4. Existe uma maneira mais rápida de computar o n -ésimo número de Fibonacci do que com `fib2` (página 4)? Uma idéia envolve *matrizes*.

Começamos por escrever as equações $F_1 = F_1$ e $F_2 = F_0 + F_1$ em notação de matriz:

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

De modo similar,

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

e, em geral

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Então, para computar F_n , é suficiente elevar a matriz 2×2 , X , a n -ésima potência.

- (a) Mostre que duas matrizes 2×2 podem ser multiplicadas usando 4 adições e 8 multiplicações.

Mas quantas multiplicações de matrizes são necessárias para computar X^n ?

- (b) Mostre que $O(\log n)$ multiplicações de matrizes são suficientes para computar X^n . (*Dica:* pense na computação de X^8 .)

Portanto o número de operações aritméticas realizadas por nosso algoritmo baseado em matrizes, chamemos de `fib3`, é apenas $O(\log n)$, em comparação a $O(n)$ para `fib2`. Quebramos outra barreira exponencial!

O problema é que nosso novo algoritmo envolve multiplicação, não apenas adição; e multiplicações de números grandes são mais lentas que adições. Já vimos que, quando a complexidade das operações aritméticas é levada em conta, o tempo de execução de `fib2` torna-se $O(n^2)$.

- (c) Mostre que todos os resultados intermediários de `fib3` possuem $O(n)$ bits de comprimento.
- (d) Seja $M(n)$ o tempo de execução de um algoritmo para multiplicar números de n bits e assuma que $M(n) = O(n^2)$ (o método da escola para multiplicação, recordado no Capítulo 1, alcança isto). Prove que o tempo de execução de `fib3` é $O(M(n) \log n)$.
- (e) Você pode provar que o tempo de execução de `fib3` é $O(M(n))$? Assuma $M(n) = \Theta(n^a)$ para algum $1 \leq a \leq 2$. (*Dica:* O tamanho dos números multiplicados dobra com cada elevação ao quadrado.)

Concluindo, para que `fib3` seja mais rápido do que `fib2` depende de podermos multiplicar inteiros de n bits mais rápido que $O(n^2)$. Você acha que isso é possível? (A resposta está no Capítulo 2.)

Por fim, existe uma fórmula para os números de Fibonacci:

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n.$$

Assim, pareceria que precisamos apenas elevar um par de números à n -ésima potência para computar F_n . O problema é que esses números são irracionais, e computá-los com precisão suficiente é não trivial. De fato, nosso método de matrizes, `fib3`, pode ser visto como uma maneira indireta de elevar esses números irracionais à n -ésima potência. Se você se lembra da álgebra linear, deve entender o porquê. (*Dica:* Quais são os autovalores da matriz X ?)

Capítulo 1

Algoritmos sobre números

Um dos principais temas deste capítulo é o intenso contraste entre dois problemas antigos que a princípio parecem muito similares:

FATORAÇÃO: Dado um número N , expresse-o como um produto de seus fatores primos.

PRIMALIDADE: Dado um número N , determine se ele é primo.

Fatoração é difícil. Apesar de séculos de esforços de alguns dos mais inteligentes matemáticos e cientistas da computação, os métodos mais rápidos para fatorar um número N tomam tempo exponencial no número de bits de N .

Por sua vez, logo veremos que *podemos* eficientemente testar se N é primo! Além disso (isso fica ainda mais interessante), a estranha disparidade entre esses dois problemas intimamente relacionados, um muito difícil e o outro muito fácil, está no coração da tecnologia que possibilita a comunicação segura no ambiente global de informação de hoje.

Seguindo a percepção dessas idéias, precisamos desenvolver algoritmos para uma variedade de tarefas computacionais envolvendo números. Comecemos com aritmética básica, um ponto de partida especialmente apropriado, porque, como sabemos, a palavra *algoritmo* originalmente se aplicava somente a métodos para estes problemas.

1.1 Aritmética básica

1.1.1 Adição

Éramos tão jovens quando aprendemos a técnica padrão para adição que dificilmente teríamos pensado em perguntar *por que* ela funciona. Mas vamos voltar agora e examinar melhor.

Veja uma propriedade básica de números decimais:

A soma de quaisquer três números de um dígito tem comprimento de no máximo dois dígitos.

Checagem rápida: a soma é no máximo $9 + 9 + 9 = 27$, dois dígitos de comprimento. De fato, essa regra funciona não apenas em decimal, mas em *qualquer* base $b \geq 2$ (Exercício 1.1). Em binário, por exemplo, a soma máxima de três números de um dígito é 3, que é um número de dois bits.

Bases e logaritmos

Naturalmente, não há nada especial com o número 10 — apenas acontece que temos 10 dedos e, assim, 10 era um ponto óbvio para parar e levar a contagem para o próximo nível. Os maias desenvolveram um sistema posicional similar baseado no número 20 (não havia sapatos, percebe?). E, claro, os computadores hoje representam números em binário.

Quantos dígitos são necessários para representar o número $N \geq 0$ em base b ? Vejamos — com k dígitos em base b , podemos expressar números até $b^k - 1$; por exemplo, em decimal, três dígitos nos levam até $999 = 10^3 - 1$. Resolvendo para k , encontramos que $\lceil \log_b(N + 1) \rceil$ é dígitos (cerca de $\log_b N$ dígitos, mais 1 ou menos 1) são necessários para escrever N em base b .

Quanto muda o tamanho de um número quando mudamos de base? Relembre a regra para conversão de logaritmos da base a para base b : $\log_b N = (\log_a N) / (\log_a b)$. Assim, o tamanho do inteiro N na base a é o mesmo que na base b vezes um fator constante $\log_a b$. Na notação O , portanto, a base é irrelevante e escrevemos o tamanho simplesmente como $O(\log N)$. Quando não especificarmos uma base, que será quase sempre o caso, queremos dizer $\log_2 N$.

A propósito, essa função $\log N$ aparece repetidamente na nossa discussão, em várias formas. Veja um exemplo:

1. $\log N$ é, claro, a potência a qual você precisa elevar 2 para obter N .
2. No sentido inverso, ela pode ser vista como o número de vezes que você precisa dividir N ao meio para chegar em 1. (Mais precisamente: $\lfloor \log N \rfloor$.) Isso é útil quando um número é dividido ao meio em cada iteração de um algoritmo, como em vários exemplos mais adiante neste capítulo.
3. É o número de bits na representação binária de N . (Mais precisamente: $\lceil \log(N + 1) \rceil$.)
4. É também a profundidade de uma árvore binária completa com N nós. (Mais precisamente: $\lfloor \log N \rfloor$.)
5. É até a soma $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N}$, a menos de um fator constante (Exercício 1.5).

Essa regra simples nos dá uma maneira de adicionar dois números em qualquer base: alinhe pela direita e, então, realize uma única passada da direita para a esquerda na qual a soma é computada dígito por dígito, mantendo o excesso como *carry*. Como sabemos que cada soma individual é um número de 2 dígitos, o excesso é sempre *apenas um dígito* e, assim, a cada passo, três números de um dígito são somados. Aqui está um exemplo mostrando a adição 53 + 35 em binário.

Excesso: 1	1	1	1				
	1	1	0	1	0	1	(53)
	1	0	0	0	1	1	(35)
1	0	1	1	0	0	0	(88)

Normalmente escreveríamos o algoritmo em pseudocódigo, mas, neste caso, ele é tão familiar que não o repetiremos. Em vez disso, partiremos direto para analisar sua eficiência.

Dados dois números binários x e y , quanto tempo o nosso algoritmo gasta para adicioná-los? Esse é o tipo de questão que faremos persistentemente ao longo do livro. Queremos a resposta expressa como uma função do *tamanho da entrada*: o número de bits de x e y , o número de teclas que precisamos pressionar para digitá-los.

Suponha que x e y possuam, respectivamente, n bits de comprimento; neste capítulo usaremos a letra n para o tamanho dos números. Então a soma de x e y possui no máximo $n + 1$ bits e cada bit dessa soma é computado em tempo constante. O tempo de execução total para o algoritmo de adição é, portanto, da forma $c_0 + c_1n$, onde c_0 e c_1 são constantes; em outras palavras, ele é *linear*. Em vez de nos preocuparmos com os valores precisos de c_0 e c_1 , concentraremos-nos no geral e denotaremos o tempo de execução por $O(n)$.

Agora que temos um algoritmo cujo tempo de execução conhecemos, nosso pensamento é levado à questão de existir alguma coisa ainda melhor.

Será que existe um algoritmo mais rápido? (Essa é outra questão persistente.) Para adição, a resposta é fácil: para adicionarmos dois números de n bits, temos que, no mínimo, ler os números e escrever a resposta, e só isso já requer n operações. Portanto o algoritmo para adição é ótimo, à altura de constantes multiplicativas!

Alguns leitores podem estar confusos neste momento: por que $O(n)$ operações? Adição binária não é algo que os computadores hoje realizam em apenas uma operação? Há duas respostas. Primeiro, é certamente verdade que em uma única instrução podemos adicionar inteiros cujo tamanho em bits cabe no *tamanho da palavra* dos computadores de hoje — 32 bits talvez. Como ficará evidente mais adiante neste capítulo, entretanto é freqüentemente útil e necessário manipular números muito maiores do que isso, talvez vários milhares de bits em comprimento. Adição e multiplicação de tais números grandes em computadores reais realizam-se da mesma maneira que nas operações bit a bit. Segundo, quando queremos entender algoritmos, faz sentido estudar os algoritmos mais básicos codificados no hardware dos computadores atuais. Desse modo, focaremos na *complexidade de bit* dos algoritmos, o número de operações elementares sobre bits individuais — porque essa abordagem reflete a quantidade de hardware, transistores e fios, necessária para implementar o algoritmo.

1.1.2 Multiplicação e divisão

Vamos à multiplicação! O algoritmo da escola para multiplicar dois números x e y consiste em criar uma série de somas intermediárias, cada uma representando o produto de x por um único dígito de y . Esses valores são apropriadamente deslocados para a esquerda e, então, somados. Suponha, por exemplo, que queiramos multiplicar 13×11 , ou em notação binária, $x = 1101$ e $y = 1011$. A multiplicação procederia assim.

$$\begin{array}{r}
 & 1 & 1 & 0 & 1 \\
 \times & 1 & 0 & 1 & 1 \\
 \hline
 & 1 & 1 & 0 & 1 & (1101 vezes 1) \\
 & 1 & 1 & 0 & 1 & (1101 vezes 1, deslocado uma vez) \\
 0 & 0 & 0 & 0 & (1101 vezes 0, deslocado duas vezes) \\
 + & 1 & 1 & 0 & 1 & (1101 vezes 1, deslocado três vezes) \\
 \hline
 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & (binário 143)
 \end{array}$$

Em binário isso é particularmente fácil, pois cada linha intermediária é ou zero ou o próprio x , deslocado para a esquerda um número apropriado de vezes. Também note que deslocar para a esquerda é justamente uma maneira rápida de multiplicar pela base, que nesse caso é 2. (Da mesma forma, o efeito de deslocar para a direita é dividir pela base, arredondando para baixo se necessário.)

A correção desse procedimento de multiplicação é o assunto do Exercício 1.6; vamos seguir e descobrir quanto tempo ele toma. Se x e y têm, ambos, n bits, então há n linhas intermediárias, com comprimentos de até $2n$ bits (levando o deslocamento em conta). O tempo total tomado para adicionar essas linhas, operando dois números por vez, é

$$\underbrace{O(n) + O(n) + \cdots + O(n)}_{n - 1 \text{ vezes}},$$

que é $O(n^2)$, *quadrático* no tamanho das entradas: ainda polinomial, mas muito mais lento do que adição (como sempre suspeitamos desde a escola).

Porém Al Khwarizmi conhecia outra maneira de multiplicar, um método usado hoje em alguns países europeus. Para multiplicar dois números decimais x e y , escreva-os lado a lado, como no exemplo a seguir. Então repita o seguinte: divida o primeiro número por 2, arredondando para baixo o resultado (ou seja, descartando o ,5 se o número for ímpar) e dobre o segundo número. Continue até que o primeiro número torne-se 1. Depois risque todas as linhas em que o primeiro número é par e some o que restar na segunda coluna.

$$\begin{array}{r}
 11 & 13 \\
 5 & 26 \\
 2 & 52 & \text{(riscada)} \\
 1 & 104 \\
 \hline
 143 & \text{(resposta)}
 \end{array}$$

Mas se agora compararmos os dois algoritmos, multiplicação binária e multiplicação por divisão sucessiva do multiplicador por 2, notamos que eles estão fazendo a mesma coisa! Os três números adicionados no segundo algoritmo são precisamente os múltiplos de 13 por potências de 2 que foram adicionados no método binário. Apenas dessa vez 11 não é dado explicitamente em binário e, portanto, tivemos de extrair sua representação binária examinando a paridade dos números obtidos dele por sucessivas divisões por 2. O segundo algoritmo de Al Khwarizmi é uma fascinante mistura de decimal e binário!

O mesmo algoritmo pode, portanto, ser reformulado de diferentes maneiras. Em nome da variedade, adotamos uma terceira formulação, o algoritmo recursivo da Figura 1.1, que diretamente implementa a regra

$$x \cdot y = \begin{cases} 2(x \cdot \lfloor y/2 \rfloor) & \text{se } y \text{ é par} \\ x + 2(x \cdot \lfloor y/2 \rfloor) & \text{se } y \text{ é ímpar.} \end{cases}$$

Esse algoritmo é correto? A regra recursiva anterior é transparentemente correta; portanto

Figura 1.1 Multiplicação à la Français.

função mutiplica(x,y)

Entrada: Dois inteiros de n bits x e y , onde $y \geq 0$

Saída: O produto deles

```

se  $y = 0$  : retorna 0
 $z = \text{mutiplica}(x, \lfloor y/2 \rfloor)$ 
se  $y$  é par:
    retorna  $2z$ 
senão:
    retorna  $x + 2z$ 

```

checar a correção do algoritmo é meramente uma questão de verificar se ele espelha a regra e se trata o caso-base ($y = 0$) apropriadamente.

Quanto tempo toma o algoritmo? Ele tem de terminar depois de n chamadas recursivas, porque a cada chamada y é dividido por 2 — seu número de bits diminui uma unidade. E cada chamada recursiva requer estas operações: uma divisão por 2 (deslocamento para a direita); um teste de paridade (checar o último bit); uma multiplicação por 2 (deslocamento para a esquerda); e possivelmente uma adição, um total de $O(n)$ operações de bit. O tempo total gasto é, portanto, $O(n^2)$, o mesmo de antes.

Será que podemos fazer melhor? Intuitivamente, parece que multiplicação requer adição de cerca de n múltiplos de uma das entradas e sabemos que cada adição é linear; assim pareceria que n^2 operações de bits são inevitáveis. Surpreendentemente, veremos no Capítulo 2 que *podemos* fazer muito melhor!

Divisão é a próxima operação. Dividir um inteiro x por outro inteiro $y \neq 0$ significa achar o quociente q e o resto r , onde $x = yq + r$ e $r < y$. Mostramos a versão recursiva da divisão na Figura 1.2; como a multiplicação, ela toma tempo quadrático. A análise deste algoritmo é o assunto do Exercício 1.8.

Figura 1.2 Divisão.

função divide(x,y)

Entrada: Dois inteiros de n bits x e y , onde $y \geq 1$

Saída: O quociente e o resto de x dividido por y

```

se  $x = 0$ : retorna  $(q, r) = (0, 0)$ 
 $(q, r) = \text{divide}(\lfloor x/2 \rfloor, y)$ 
 $q = 2 \cdot q$ ,  $r = 2 \cdot r$ 
se  $x$  é ímpar:  $r = r + 1$ 
se  $r \geq y$ :  $r = r - y$ ,  $q = q + 1$ 
retorna  $(q, r)$ 

```

1.2 Aritmética modular

Com sucessivas adições e multiplicações, números podem ficar complicadamente grandes. Portanto é uma sorte que a hora volte a zero quando alcança 24 e que o mês volte a ser janeiro depois de um período de 12 meses. De maneira similar, para as operações aritméticas embutidas nos processadores dos computadores, números são restritos a um certo tamanho, digamos 32 bits, o que é considerado generoso o suficiente para a maioria dos propósitos.

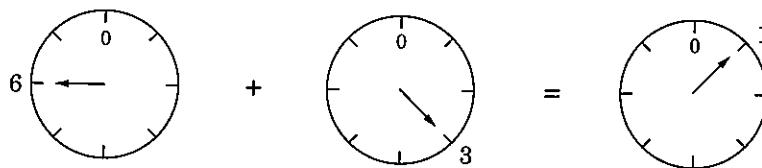
Para as aplicações na direção das quais estamos trabalhando — teste de primalidade e criptografia —, é necessário lidar com números significativamente maiores do que 32 bits, mas que ainda estão num intervalo restrito.

Aritmética modular é um sistema para lidar com inteiros de um intervalo restrito. Definimos x módulo N como o resto da divisão de x por N ; quer dizer, se $x = qN + r$ com $0 \leq r < N$, então x módulo N é igual a r . Isto proporciona uma noção realçada de equivalência entre números: x e y são *congruentes módulo N* se diferem por um múltiplo de N , ou em símbolos,

$$x \equiv y \pmod{N} \iff N \text{ divide } (x - y).$$

Por exemplo, $253 \equiv 13$ porque $253 - 13$ é múltiplo de 60; de modo mais familiar, 253 minutos são 4 horas e 13 minutos. Esses números também podem ser negativos, como em $59 \equiv -1$: quando são 59 minutos após a hora, também é 1 minuto antes da próxima hora.

Figura 1.3 Adição módulo 8.



Uma maneira de pensar sobre aritmética modular é que ela limita os números a um intervalo predefinido $\{0, 1, \dots, N - 1\}$ e, sempre que você tenta sair desse intervalo por uma extremidade, ela circula para a outra extremidade — como os ponteiros de um relógio (Figura 1.3).

Outra interpretação é que aritmética modular lida com todos os inteiros, porém os divide em N classes de equivalência, cada uma da forma $\{i + kN : k \in \mathbb{Z}\}$ para algum i entre 0 e $N - 1$. Por exemplo, há três classes de equivalência módulo 3:

$$\begin{array}{ccccccccc} \dots & -9 & -6 & -3 & 0 & 3 & 6 & 9 & \dots \\ \dots & -8 & -5 & -2 & 1 & 4 & 7 & 10 & \dots \\ \dots & -7 & -4 & -1 & 2 & 5 & 8 & 11 & \dots \end{array}$$

Qualquer membro de uma classe de equivalência é substituível por qualquer outro da classe; quando vistos módulo 3, os números 5 e 11 não têm qualquer diferença. Sobre tais substituições, adição e multiplicação permanecem bem-definidas:

Complemento de dois

Aritmética modular é muito bem ilustrada em *complemento de dois*, o formato mais comum para representar inteiros com sinal. Ele usa n bits para representar números no intervalo $[-2^{n-1}, 2^{n-1} - 1]$ e em geral é descrito assim:

- Inteiros positivos, no intervalo 0 a $2^{n-1} - 1$, são guardados em binário normal e tem o bit 0 como o mais significativo.
- Inteiros negativos $-x$, com $1 \leq x \leq 2^{n-1}$, são guardados primeiro construindo x em binário, depois negando todos os bits, e finalmente somando 1. O bit mais significativo neste caso é 1.

(E a descrição usual de adição e multiplicação neste formato é ainda mais difícil de entender!)

Veja uma maneira muito mais simples de pensar sobre isso: qualquer número no intervalo -2^{n-1} a $2^{n-1} - 1$ é guardado módulo 2^n . Números negativos $-x$, portanto, acabam representados como $2^n - x$. Operações aritméticas como adição e subtração podem ser realizadas diretamente nesse formato, ignorando qualquer bit de excesso que apareça.

Regra da substituição Se $x \equiv x' \pmod{N}$ e $y \equiv y' \pmod{N}$, então:

$$x + y \equiv x' + y' \pmod{N} \text{ e } xy \equiv x'y' \pmod{N}.$$

(Veja o Exercício 1.9.) Por exemplo, suponha que você assista a uma temporada inteira da sua série de TV preferida de uma só vez, começando à meia-noite. Há 25 episódios, cada um durando 3 horas. A que horas do dia você termina de assistir? Resposta: a hora de término é $(25 \times 3) \bmod 24$, que (como $25 \equiv 1 \pmod{24}$) é $1 \times 3 = 3 \bmod 24$, ou três horas da manhã.

Não é difícil checar que, em aritmética modular, as propriedades usuais de associação, comutação e distribuição da adição e multiplicação continuam valendo, por exemplo:

$x + (y + z) \equiv (x + y) + z \pmod{N}$	Associação
$xy \equiv yx \pmod{N}$	Comutação
$x(y+z) \equiv xy+yz \pmod{N}$	Distribuição

Tomadas com a regra da substituição, isso implica que, ao realizar uma seqüência de operações aritméticas, é possível reduzir resultados intermediários a seus restos módulo N em qualquer estágio. Tais simplificações podem ser uma enorme ajuda em cálculos grandes. Testemunha disso é, por exemplo:

$$2^{345} \equiv (2^5)^{69} \equiv 32^{69} \equiv 1^{69} \equiv 1 \pmod{31}.$$

1.2.1 Adição e multiplicação modular

Para *adicionar*mos dois números x e y módulo N , começamos com a adição regular. Como x e y estão ambos no intervalo 0 a $N - 1$, sua soma está entre 0 e $2(N - 1)$. Se

a soma excede $N - 1$, subtraímos N meramente para trazê-la de volta para o intervalo requerido. A computação total, portanto, consiste em uma adição e, possivelmente, uma subtração, de números que nunca excedem $2N$. O tempo de execução é linear no tamanho desses números, em outras palavras $O(n)$, onde $n = \lceil \log N \rceil$ é o tamanho de N ; como lembrete, nossa convenção é usar a letra n para denotar o tamanho da entrada.

Para *multiplicarmos* dois números x e y módulo N , novamente começamos com a multiplicação regular e, então, reduzimos a resposta módulo N . O produto pode ser tão grande quanto $(N - 1)^2$, mas isso ainda tem no máximo $2n$ bits de comprimento, pois $\log(N - 1)^2 = 2\log(N - 1) \leq 2n$. Para reduzir a resposta módulo N , computamos o resto dividindo-o por N , usando nosso algoritmo de tempo quadrático para divisão. Multiplicação, desse modo, permanece uma operação quadrática.

Divisão não é tão simples assim. Em aritmética ordinária há apenas um caso complicado — divisão por zero. Em aritmética modular há potencialmente outros casos semelhantes, que iremos caracterizar ao final desta seção. Sempre que divisão é possível, entretanto ela pode ser resolvida em tempo cúbico, $O(n^3)$.

Para completarmos a coleção de primitivas de aritmética modular que precisamos para criptografia, veremos agora *exponenciação modular* e depois o *máximo divisor comum*, que é a chave para divisão. Para ambas as tarefas, os procedimentos mais óbvios tomam tempo exponencial, mas com alguma engenhosidade podemos encontrar soluções de tempo polinomial. A escolha cuidadosa do algoritmo faz toda a diferença.

1.2.2 Exponenciação modular

No sistema de criptografia que visamos, é necessário computar $x^y \bmod N$ para valores de x , y e N que têm comprimentos de várias centenas de bits. Isso pode ser feito rapidamente?

O resultado é algum número módulo N e, portanto, ele próprio tem comprimento e algumas centenas de bits. Entretanto, o valor bruto de x^y pode ser muito, muito maior do que isso. Mesmo quando x e y possuem apenas 20 bits, x^y é pelo menos $(2^{19})^{(2^{19})} = 2^{(19)(524288)}$, que tem cerca de 10 milhões de bits! Imagine o que acontece se y for um número de 500 bits!

Para nos certificarmos de que os números com os quais lidamos nunca cresçam demais, temos de realizar todas as computações intermediárias módulo N . Assim, aqui está uma idéia: calcule x^y multiplicando sucessivamente por x módulo N . A seqüência resultante de produtos intermediários,

$$x \bmod N \rightarrow x^2 \bmod N \rightarrow x^3 \bmod N \rightarrow \cdots \rightarrow x^y \bmod N,$$

consiste em números menores do que N e, portanto, as multiplicações individuais não tomam muito tempo. Mas há um problema: se y possui 500 bits, precisamos realizar $y - 1 \approx 2^{500}$ multiplicações! Esse algoritmo é claramente exponencial no tamanho de y .

Por sorte, *podemos* fazer melhor: começando com x e elevando ao quadrado módulo N , temos

$$x \bmod N \rightarrow x^2 \bmod N \rightarrow x^4 \bmod N \rightarrow x^8 \bmod N \rightarrow \cdots \rightarrow x^{2^{\lceil \log y \rceil}} \bmod N.$$

Cada multiplicação toma apenas tempo $O(\log^2 N)$ e, neste caso, há apenas $\log y$ multiplicações. Para determinarmos $x^y \bmod N$, simplesmente multiplicamos um subconjunto apropriado dessas potências, aquelas correspondendo aos 1 da representação binária de y . Por exemplo,

$$x^{25} = x^{11001_2} = x^{10000_2} \cdot x^{1000_2} \cdot x^{1_2} = x^{16} \cdot x^8 \cdot x^1.$$

Um algoritmo de tempo polinomial está finalmente ao alcance!

Figura 1.4 Exponenciação modular.

função modexp (x, y, N)

Entrada: Dois inteiros de n bits x e N , um expoente inteiro y

Saída: $x^y \bmod N$

```

se  $y = 0$ : retorna 1
z = modexp( $x, \lfloor y/2 \rfloor, N$ )
se  $y$  é par:
    retorna  $z^2 \bmod N$ 
senão:
    retorna  $x \cdot z^2 \bmod N$ 

```

Podemos formular essa idéia de uma maneira particularmente simples: o algoritmo recursivo da Figura 1.4, que funciona executando, módulo N , a regra auto-evidente

$$x^y = \begin{cases} (x^{\lfloor y/2 \rfloor})^2 & \text{se } y \text{ é par} \\ x \cdot (x^{\lfloor y/2 \rfloor})^2 & \text{se } y \text{ é ímpar.} \end{cases}$$

Isso se compara fortemente com nosso algoritmo recursivo para multiplicação (Figura 1.1). Por exemplo, aquele algoritmo computaria o produto $x \cdot 25$ por uma decomposição análoga àquela que acabamos de ver: $x \cdot 25 = x \cdot 16 + x \cdot 8 + x \cdot 1$. E enquanto para multiplicação os termos $x \cdot 2^i$ vêm de sucessivas *multiplicações por 2*, para exponenciação os termos correspondentes x^{2^i} são gerados por sucessivas elevações ao quadrado.

Seja n o tamanho em bits de x , y e N (o tamanho do maior dos três). Assim como na multiplicação, o algoritmo vai parar após no máximo n chamadas recursivas, e durante cada chamada ele multiplica números de n bits (realizar computação módulo N nos salva aqui), com um tempo de execução total de $O(n^3)$.

1.2.3 Algoritmo de Euclides para o máximo divisor comum

Nosso próximo algoritmo foi descoberto há mais de dois mil anos pelo matemático Euclides, na Grécia antiga. Dados dois inteiros a e b , ele encontra o maior inteiro que divide ambos, conhecido como o *máximo divisor comum* (mdc).

A abordagem mais óbvia é primeiro fatorar a e b e, depois, multiplicar seus fatores comuns. Por exemplo, $1035 = 3^2 \cdot 5 \cdot 23$ e $759 = 3 \cdot 11 \cdot 23$, portanto seu mdc

é $3 \cdot 23 = 69$. Contudo, não temos um algoritmo eficiente para fatoração. Existe alguma outra maneira de computar máximos divisores comuns?

O algoritmo de Euclides usa a seguinte fórmula simples.



Euclides de Alexandria
325-265 a.C.

© Corbis

Regra de Euclides Se x e y são inteiros positivos com $x \geq y$, então $\text{mdc}(x, y) = \text{mdc}(x \bmod y, y)$.

Prova. É suficiente mostrar a seguinte regra ligeiramente mais simples $\text{mdc}(x, y) = \text{mdc}(x - y, y)$, da qual a regra de Euclides pode ser derivada subtraindo-se y de x sucessivamente.

Vamos lá. Qualquer inteiro que divida tanto x quanto y tem de dividir também $x - y$, portanto $\text{mdc}(x, y) \leq \text{mdc}(x - y, y)$. Da mesma forma, qualquer inteiro que divida tanto $x - y$ quanto y tem de dividir também tanto x quanto y , portanto $\text{mdc}(x, y) \geq \text{mdc}(x - y, y)$. ■

Figura 1.5 Algoritmo de Euclides para encontrar o máximo divisor comum de dois números.

função Euclides (a, b)

Entrada: Dois inteiros a e b , com $a \geq b \geq 0$

Saída: $\text{mdc}(a, b)$

```
se  $b = 0$ : retorna  $a$ 
retorna Euclides ( $b, a \bmod b$ )
```

A regra de Euclides nos permite escrever um elegante algoritmo recursivo (Figura 1.5), e sua correção segue imediatamente a regra. Para encontrarmos seu tempo de execução, precisamos entender quão rapidamente os argumentos (a, b) decrescem com cada sucessiva chamada recursiva. Em uma rodada, os argumentos (a, b) tornam-se $(b, a \bmod b)$: sua ordem é trocada, e o maior deles, a , é reduzido a $\bmod b$. Essa redução é substancial.

Lema Se $a \geq b$, então, $a \bmod b < a/2$.

Prova. Observe que ou $b \leq a/2$ ou $b > a/2$. Esses dois casos são apresentados na seguinte figura. Se $b \leq a/2$, então, temos $a \bmod b < b \leq a/2$; e se $b > a/2$, então, $a \bmod b = a - b < a/2$. ■



Isso significa que depois de quaisquer duas rodadas consecutivas, ambos os argumentos, a e b , caíram pela metade pelo menos — o tamanho de cada um decresce de pelo menos 1 bit. Se eles são inicialmente inteiros de n bits, então o caso-base será alcançado em $2n$ chamadas recursivas. Como cada chamada envolve uma divisão de tempo quadrático, o tempo total é $O(n^3)$.

1.2.4 Uma extensão do algoritmo de Euclides

Uma pequena extensão do algoritmo de Euclides é a chave para divisão no mundo modular.

Para motivar isso, suponha que alguém afirmre que d é o máximo divisor comum de a e b : como podemos checar? Não é suficiente verificar que d divide tanto a quanto b , porque isso apenas mostra que d é um fator comum, não necessariamente o maior. Veja um teste que pode ser usado se d caracteriza-se por uma forma particular.

Lema Se d divide tanto a quanto b e $d = ax + by$ para inteiros x e y , então, necessariamente $d = \text{mdc}(a, b)$.

Prova. Por um lado, pelas duas primeiras condições, d é um divisor comum de a e b , portanto, não pode exceder o máximo divisor comum; ou seja, $d \leq \text{mdc}(a, b)$. Por outro lado, como o $\text{mdc}(a, b)$ é um divisor comum de a e b , ele tem de dividir também $ax + by = d$, o que implica que $\text{mdc}(a, b) \leq d$. Juntando isso, $d = \text{mdc}(a, b)$. ■

Assim, se pudermos fornecer dois números x e y tal que $d = ax + by$, então, teremos certeza de que $d = \text{mdc}(a, b)$. Por exemplo, sabemos que $\text{mdc}(13, 4) = 1$ porque $13 \cdot 1 + 4 \cdot (-3) = 1$. Mas quando podemos achar esses números? Sob que circunstâncias o $\text{mdc}(a, b)$ pode ser expresso nessa forma verificável? Ele *sempre* pode. E o que é ainda melhor, os coeficientes x e y podem ser encontrados com uma pequena extensão do algoritmo de Euclides; veja a Figura 1.6.

Figura 1.6 Uma extensão simples do algoritmo de Euclides.

função Euclides-estendido (a, b)

Entrada: Dois inteiros a e b com $a \geq b \geq 0$

Saída: Inteiros x, y, d tais que $d = \text{mdc}(a, b)$ e $ax + by = d$

se $b = 0$: retorna $(1, 0, a)$

$(x', y', d) := \text{Euclides-estendido} (b, a \bmod b)$

retorna $(y', x' - \lfloor a/b \rfloor y', d)$

Lema Para quaisquer inteiros positivos a e b , o algoritmo estendido de Euclides retorna inteiros x , y e d tal que $\text{mdc}(a, b) = d = ax + by$.

Prova. A primeira coisa a confirmar é que se você ignora os x e y , o algoritmo estendido é exatamente igual ao original. Portanto, pelo menos computamos $d = \text{mdc}(a, b)$.

Para o restante, a natureza recursiva do algoritmo sugere uma prova por indução. A recursão termina quando $b = 0$, portanto é conveniente fazer a indução no valor de b .

O caso-base $b = 0$ é fácil e suficiente para uma checagem direta. Agora tome qualquer valor maior de b . O algoritmo encontra $\text{mdc}(a, b)$ chamando $\text{mdc}(b, a \bmod b)$. Como $a \bmod b < b$, podemos aplicar a hipótese de indução a essa chamada recursiva e concluir que o x' e o y' que ela retorna são corretos:

$$\text{mdc}(b, a \bmod b) = bx' + (a \bmod b)y'.$$

Escrevendo $(a \bmod b)$ como $(a - [a/b]b)$, encontramos

$$\begin{aligned} d = \text{mdc}(a, b) &= \text{mdc}(b, a \bmod b) = bx' + (a \bmod b)y' \\ &= bx' + (a - [a/b]b)y' = ay' + b(x' - [a/b]y'). \end{aligned}$$

Desse modo, $d = ax + by$ com $x = y'$ e $y = x' - [a/b]y'$, validando assim o comportamento do algoritmo sobre a entrada (a, b) . ■

Exemplo. Para computar $\text{mdc}(25, 11)$, o algoritmo de Euclides procederia da seguinte forma:

$$\begin{aligned} \underline{\underline{25}} &= 2 \cdot \underline{11} + 3 \\ \underline{11} &= 3 \cdot \underline{3} + 2 \\ \underline{3} &= 1 \cdot \underline{2} + 1 \\ \underline{2} &= 2 \cdot \underline{1} + 0 \end{aligned}$$

(em cada estágio, a computação do mdc foi reduzida aos números sublinhados). Assim, $\text{mdc}(25, 11) = \text{mdc}(11, 3) = \text{mdc}(3, 2) = \text{mdc}(2, 1) = \text{mdc}(1, 0) = 1$.

Para encontrarmos x e y tal que $25x + 11y = 1$, começamos por expressar 1 em termos do último par $(1, 0)$. Depois vamos para trás, expressando 1 em termos de $(2, 1)$, $(3, 2)$, $(11, 3)$ e finalmente $(25, 11)$. O primeiro passo é:

$$1 = \underline{1} - \underline{0}.$$

Para reescrevermos isso em termos de $(2, 1)$, usamos a substituição $0 = 2 - 2 \cdot 1$ da última linha do cálculo do mdc obtendo:

$$1 = \underline{1} - (2 - 2 \cdot \underline{1}) = -1 \cdot \underline{2} + 3 \cdot \underline{1}.$$

A penúltima linha do cálculo do mdc no diz que $1 = 3 - 1 \cdot 2$. Substituindo:

$$1 = -1 \cdot \underline{2} + 3(\underline{3} - 1 \cdot \underline{2}) = 3 \cdot \underline{3} - 4 \cdot \underline{2}.$$

Continuando da mesma maneira com as substituições $2 = 11 - 3 \cdot 3$ e $3 = 25 - 2 \cdot 11$ obtemos:

$$1 = 3 \cdot \underline{3} - 4(\underline{11} - 3 \cdot \underline{3}) = -4 \cdot \underline{11} + 15 \cdot \underline{3} = -4 \cdot \underline{11} + 15(25 - 2 \cdot \underline{11}) = 15 \cdot \underline{25} - 34 \cdot \underline{11}.$$

Está feito: $15 \cdot 25 - 34 \cdot 11 = 1$, portanto $x = 15$ e $y = -34$.

1.2.5 Divisão modular

Na aritmética real, todo número $a \neq 0$ tem um inverso, $1/a$, e dividir por a é o mesmo que multiplicar pelo inverso. Na aritmética modular, podemos fazer uma definição similar.

Dizemos que x é o inverso multiplicativo de a , módulo N , se $ax \equiv 1 \pmod{N}$.

Pode haver no máximo um tal x módulo N (Exercício 1.23) e o denotaremos por a^{-1} . Entretanto, este inverso nem sempre existe! Por exemplo, 2 não é inversível módulo 6: quer dizer, $2x \not\equiv 1 \pmod{6}$ para qualquer possível escolha de x . Nesse caso, a e N são ambos pares e assim $a \pmod{N}$ é sempre par, porque $a \pmod{N} = a - kN$ para algum k . Mais em geral, podemos estar certos de que $\text{mdc}(a, N)$ divide $ax \pmod{N}$, porque essa última quantidade pode ser escrita na forma $ax + kN$. Assim, se $\text{mdc}(a, N) > 1$, então $ax \not\equiv 1 \pmod{N}$, não importa quem seja x e, portanto, a não pode ter um inverso multiplicativo módulo N .

De fato, essa é a única circunstância na qual a não é inversível. Quando $\text{mdc}(a, N) = 1$ (dizemos que a e N são *primos relativos*), o algoritmo estendido de Euclides nos dá inteiros x e y tal que $ax + Ny = 1$, o que significa que $ax \equiv 1 \pmod{N}$. Portanto, x é o inverso de a .

Exemplo. Continuando com nosso exemplo anterior, suponha que queiramos calcular $11^{-1} \pmod{25}$. Usando o algoritmo estendido de Euclides, descobrimos que $15 \cdot 25 - 34 \cdot 11 = 1$. Reduzindo ambos os lados módulo 25, temos $-34 \cdot 11 \equiv 1 \pmod{25}$. Portanto $-34 \equiv 16 \pmod{25}$ é o inverso de $11 \pmod{25}$.

Teorema da divisão modular *Para qualquer $a \pmod{N}$, a possui um inverso multiplicativo módulo N se e somente se ele é primo relativo de N . Quando esse inverso existe, ele pode ser encontrado em tempo $O(n^3)$ (onde, como usualmente, n denota o número de bits de N) executando o algoritmo estendido de Euclides.*

Isto resolve a questão da divisão modular: ao trabalharmos módulo N , podemos dividir por números primos relativos de N — e somente por eles. E para efetivamente realizarmos a divisão, multiplicamos pelo inverso.

1.3 Teste de primalidade

Há algum teste que nos diga se um número é primo sem na verdade precisarmos fatorar o número? Colocamos nossas esperanças em um teorema do ano 1640.

Pequeno teorema de Fermat *Se p é primo, então, para todo $1 \leq a < p$,*

$$a^{p-1} \equiv 1 \pmod{p}.$$

Prova. Seja S o conjunto dos inteiros não-nulos módulo p ; ou seja, $S = \{1, 2, \dots, p-1\}$. Esta é a observação crucial: o efeito de multiplicar esses números por a (módulo p) é simplesmente permutá-los. Por exemplo, veja uma figura do caso $a = 3$, $p = 7$:

O seu número de CPF é primo?

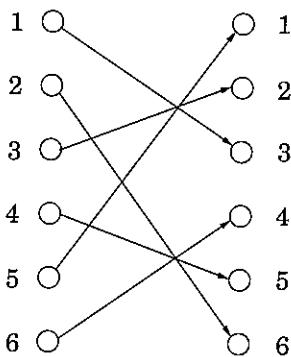
Os números 7, 17, 19, 71 e 79 são primos, mas o que dizer de 717-19-7179? Dizer se um número razoavelmente grande é primo parece algo tedioso porque há fatores candidatos demais para tentar. Entretanto, existem alguns truques para acelerar o processo. Por exemplo, você pode omitir candidatos pares depois de ter eliminado o número 2. Você pode, na verdade, eliminar todos os candidatos, exceto os que são primos.

De fato, um pouco mais de raciocínio irá convencê-lo de que pode proclamar N um primo tão logo você tenha rejeitado todos os candidatos até \sqrt{N} , porque se N pode mesmo ser fatorado como $N = K \cdot L$, então, é impossível para ambos os fatores exceder \sqrt{N} .

Parece que estamos fazendo progresso! Talvez omitindo mais e mais fatores candidatos, um teste de primalidade verdadeiramente eficiente poderia ser descoberto.

Infelizmente, não existe teste de primalidade rápido por esse caminho. A razão é que estamos tentando dizer se um número é primo *fatorando-o*. E fatoração é um problema difícil!

A criptografia moderna, bem como o balanço deste capítulo, tem a ver com a seguinte idéia importante: *fatoração é difícil e primalidade é fácil*. Não podemos fatorar grandes números, mas podemos facilmente testar a primalidade de números gigantescos! (Presumivelmente, se um número é composto, tal teste detectará isto *sem encontrar um fator*.)



Vamos elaborar esse exemplo um pouco mais. Da figura, concluímos

$$\{1, 2, \dots, 6\} = \{3 \cdot 1 \bmod 7, 3 \cdot 2 \bmod 7, \dots, 3 \cdot 6 \bmod 7\}.$$

Multiplicar todos os números em cada representação resulta em $6! \equiv 3^6 \cdot 6! \pmod{7}$, e dividir por $6!$ temos $3^6 \equiv 1 \pmod{7}$, exatamente o resultado que queríamos para o caso $a = 3$, $p = 7$.

Agora generalizemos esse argumento para outros valores de a e p , com $S = \{1, 2, \dots, p - 1\}$. Provaremos que, quando todos os elementos de S são multiplicados por a módulo p , os números resultantes são todos distintos e não-nulos. E como eles estão no intervalo $[1, p - 1]$, simplesmente têm de ser uma permutação de S .

Os números $a \cdot i \bmod p$ são distintos porque se $a \cdot i \equiv a \cdot j \pmod{p}$, então, dividirímos os dois lados por a nos dá $i \equiv j \pmod{p}$. Eles são não-nulos porque $a \cdot i \equiv 0$ de modo similar implica $i \equiv 0$. (E podemos dividir por a , porque assumimos que ele é não-nulo e, portanto, primo relativo de p .)

Temos agora duas maneiras de escrever o conjunto S :

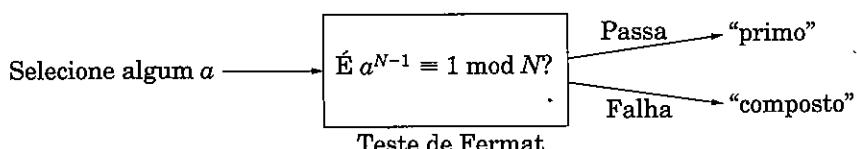
$$S = \{1, 2, \dots, p - 1\} = \{a \cdot 1 \bmod p, a \cdot 2 \bmod p, \dots, a \cdot (p - 1) \bmod p\}.$$

Podemos multiplicar seus elementos em cada uma dessas representações para obter

$$(p - 1)! \equiv a^{p-1} \cdot (p - 1)! \pmod{p}.$$

Dividirmos por $(p - 1)!$ (o que podemos fazer porque ele é primo relativo de p , já que assumimos que p é primo) nos dá o teorema. ■

Esse teorema sugere um teste “sem fatoração” para determinar se um número N é primo:



O problema é que o teorema de Fermat não é uma condição se-e-somente-se; ele não informa o que acontece quando N não é primo, portanto nesses casos o diagrama anterior é questionável. De fato, é possível para um número composto N passar no teste de Fermat (ou seja, $a^{N-1} \equiv 1 \pmod{N}$) para certas escolhas de a . Por exemplo, $341 = 11 \cdot 31$ não é primo e ainda assim $2^{340} \equiv 1 \pmod{341}$. Entretanto, podemos ter esperança de que para um composto N , a maioria dos valores de a falhará no teste. Isso é de fato verdade, em um sentido que logo tornaremos preciso e que motiva o algoritmo da Figura 1.7: em vez de fixarmos antes um valor arbitrário para a , nós o escolheremos aleatoriamente de $\{1, \dots, N - 1\}$.

Figura 1.7 Um algoritmo para testar primalidade.

função primalidade (N)

Entrada: Inteiro positivo N

Saída: sim/não

Selezione aleatoriamente um inteiro positivo $a < N$
se $a^{N-1} \equiv 1 \pmod{N}$:

 retorna sim

senão:

 retorna não

Ao analisarmos o comportamento desse algoritmo, precisamos primeiro nos livrar de um caso ruim de menor importância. Acontece que certos números compostos N extremamente raros, chamados *números de Carmichael*, passam no teste de Fermat para *todas* os primos relativos de N . Sobre tais números, nosso algoritmo falhará; mas eles são patologicamente raros e veremos mais adiante como lidar com eles (página 28), portanto vamos ignorá-los por enquanto.

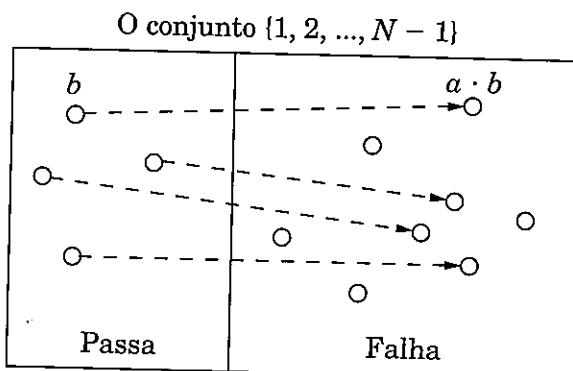
Em um mundo sem números de Carmichael, nosso algoritmo funciona bem. Qualquer número primo obviamente passará no teste de Fermat e produzirá a resposta correta. Por sua vez, qualquer número composto não-Carmichael N tem de falhar para algum valor de a ; e como mostraremos agora, isso implica imediatamente que N falha no teste de Fermat para *pelo menos metade dos possíveis valores de a* !

Lema *Se $a^{N-1} \not\equiv 1 \pmod{N}$ para algum a primo relativo de N , então o mesmo acontece para pelo menos metade das escolhas de $a < N$.*

Prova. Fixe algum valor a para o qual $a^{N-1} \not\equiv 1 \pmod{N}$. A chave é notar que qualquer elemento $b < N$ que passa no teste de Fermat com relação a N (ou seja, $b^{N-1} \equiv 1 \pmod{N}$) possui um gêmeo, $a \cdot b$, que falha no teste:

$$(a \cdot b)^{N-1} \equiv a^{N-1} \cdot b^{N-1} \equiv a^{N-1} \not\equiv 1 \pmod{N}.$$

Além disso, todos esses elementos $a \cdot b$, para a fixo, mas para escolhas diferentes de b , são distintos, pela mesma razão que $a \cdot i \not\equiv a \cdot j$ na prova do teste de Fermat: simplesmente dividia por a .



A função injetora $b \mapsto a \cdot b$ mostra que pelo menos tantos elementos falham no teste quantos passam. ■

Ao ignorarmos números de Carmichael, podemos agora afirmar

Se N é primo, então, $a^{N-1} \equiv 1 \pmod{N}$ para todo $a < N$.

Se N não é primo, então, $a^{N-1} \equiv 1 \pmod{N}$ para pelo menos metade dos valores de $a < N$.

O algoritmo da Figura 1.7, portanto, possui o seguinte comportamento probabilístico.

$$\Pr(\text{Algoritmo 1.7 retorna sim quando } N \text{ é primo}) = 1$$

$$\Pr(\text{Algoritmo 1.7 retorna sim quando } N \text{ não é primo}) \leq \frac{1}{2}$$

Ei, isso foi teoria de grupos!

Para qualquer inteiro N , o conjunto de todos os números módulo N que são primos relativos de N constitui o que os matemáticos chamam de *grupo*:

- Existe uma operação multiplicativa definida sobre esse conjunto.
- O conjunto contém um elemento neutro (a saber 1: qualquer número multiplicado por isso se mantém inalterado).
- Todos os elementos possuem um inverso bem-definido.

Esse grupo particular é chamado de *grupo multiplicativo de N* , usualmente denotado por \mathbb{Z}_N^* .

Teoria de grupos é um ramo muito bem desenvolvido da matemática. Um de seus conceitos-chave é que um grupo pode conter um *subgrupo* — um subconjunto que é ele próprio um subgrupo em si. E um importante fato sobre um subgrupo é que seu tamanho tem de dividir o tamanho do grupo inteiro.

Considere agora o conjunto $B = \{b : b^{N-1} \equiv 1 \pmod{N}\}$. Não é difícil ver que ele é um subgrupo de \mathbb{Z}_N^* (simplesmente verifique que B é fechado por multiplicação e inversão). Portanto o tamanho de B tem de dividir o de \mathbb{Z}_N^* . O que significa que se B não contém todo \mathbb{Z}_N^* , o próximo maior tamanho possível que ele pode ter é $|\mathbb{Z}_N^*|/2$.

Podemos reduzir esse erro, que ocorre em *uma só direção*, repetindo o procedimento várias vezes, selecionando aleatoriamente vários valores de a e testando todos eles (Figura 1.8).

$$\Pr(\text{Algoritmo 1.8 retorna sim quando } N \text{ não é primo}) \leq \frac{1}{2^k}$$

Essa probabilidade de erro cai exponencialmente e pode ser arbitrariamente pequena escolhendo-se um k grande o suficiente. Testar $k = 100$ valores de a faz a probabilidade de falha ser no máximo 2^{-100} , o que é mínimo: muito menos, por exemplo, que a probabilidade de um raio cósmico sabotar o computador por acaso durante a computação!

Figura 1.8 Um algoritmo para testar primalidade, com probabilidade baixa de erro.

função primalidade2 (N)

Entrada: Inteiro positivo N

Saída: sim/não

Selecione aleatoriamente inteiros positivos $a_1, a_2, \dots, a_k < N$

se $a_i^{N-1} \equiv 1 \pmod{N}$ para todo $i = 1, 2, \dots, k$:

retorna sim

senão:

retorna não

Números de Carmichael

O menor número de Carmichael é 561. Ele não é primo: $561 = 3 \cdot 11 \cdot 17$; ainda assim ele engana o teste de Fermat, porque $a^{560} \equiv 1 \pmod{561}$ para todos os valores de a primos relativos de 561. Por muito tempo se pensou que poderia haver apenas uma quantidade finita de números desse tipo; hoje sabemos que eles são infinitos, mas excessivamente raros.

Existe uma maneira de contornar números de Carmichael, usando um teste de primalidade ligeiramente mais refinado, devido a Rabin e Miller. Escreva $N - 1$ na forma $2^t u$. Como antes, escolheremos uma base aleatória a e verificaremos o valor de $a^{N-1} \pmod{N}$. Realize essa computação primeiro determinando a^u e, então, elevando sucessivamente ao quadrado, para obter a seqüência:

$$a^u \pmod{N}, a^{2u} \pmod{N}, \dots, a^{2^t u} = a^{N-1} \pmod{N}.$$

Se $a^{N-1} \not\equiv 1 \pmod{N}$, quando N é composto segundo o pequeno teorema de Fermat e, assim, terminamos. Mas se $a^{N-1} \equiv 1 \pmod{N}$, conduzimos um pequeno teste adicional: em algum momento na seqüência anterior, passamos por um 1 pela primeira vez. Se isso aconteceu depois da primeira posição (ou seja, se $a^u \pmod{N} \neq 1$), e se o valor anterior na lista não é $-1 \pmod{N}$, declaramos N composto.

No segundo caso, encontramos uma *raiz quadrada não trivial* de 1 módulo N : um número que não é $\pm 1 \pmod{N}$, mas que, quando elevado ao quadrado, é igual a 1 módulo N . Tal número somente pode existir se N é composto (Exercício 1.40). Verifica-se que se combinarmos essa checagem da raiz quadrada com nosso teste anterior de Fermat, pelo menos três quartos dos possíveis valores de a entre 1 e $N - 1$ revelarão um número composto N , mesmo se ele é um número de Carmichael.

1.3.1 Gerando primos aleatórios

Estamos agora perto de ter todas as ferramentas de que precisamos para aplicações criptográficas. A peça final do quebra-cabeça é um algoritmo rápido para selecionar primos aleatórios que têm algumas centenas de bits de comprimento. O que torna essa tarefa bem fácil é que primos são abundantes — um número aleatório de n bits tem cerca de uma chance em n de ser primo (na verdade, cerca de $1/(\ln 2^n) \approx 1.44/n$). Por exemplo, *cerca de 1 a cada 20 números de CPF são primos!*

Teorema dos números primos de Lagrange *Seja $\pi(x)$ o número de primos $\leq x$. Então $\pi(x) \approx x/(\ln x)$, ou mais precisamente,*

$$\lim_{x \rightarrow \infty} \frac{\pi(x)}{(x/\ln x)} = 1.$$

Tal abundância torna fácil gerar um primo aleatório de n bits:

- Selecione um número aleatório N de n bits.
- Execute um teste de primalidade sobre N .
- Se ele passar no teste, apresente N ; do contrário, repita o processo.

Quão rápido é esse algoritmo? Se o N selecionado aleatoriamente é realmente primo, o que acontece com probabilidade de pelo menos $1/n$, ele certamente passará no teste.

Algoritmos randomizados: um capítulo virtual

Surpreendentemente — quase de maneira paradoxal — alguns dos mais rápidos e engenhosos algoritmos que temos dependem do *acaso*: em determinados passos eles procedem de acordo com o resultado de um cara ou coroa aleatório. Esses *algoritmos randomizados* são em geral bastante simples e elegantes e permitem que seus resultados sejam incorretos *com pequena probabilidade*. Esta cota sobre a probabilidade de falha vale para todas as entradas, ela depende apenas das escolhas aleatórias feitas pelo próprio algoritmo e pode facilmente ser feita tão pequena quanto se deseje.

Em vez de dedicarmos um capítulo especial a esse tópico, intercalamos algoritmos randomizados nos capítulos e seções onde aparecem naturalmente. Além disso, nenhum conhecimento especializado de probabilidade é necessário. Você precisa apenas estar familiarizado com o conceito de probabilidade, valor esperado, o número esperado de vezes que temos de lançar uma moeda até dar cara, e a propriedade conhecida como “linearidade do valor esperado”.

Aqui estão ponteiros para os principais algoritmos randomizados neste livro: um dos mais antigos e intensos exemplos de algoritmos randomizados é o teste de primalidade probabilístico da Figura 1.8. Muito embora um teste de primalidade determinístico tenha sido recentemente descoberto, o teste randomizado é muito mais rápido e, portanto, permanece a escolha certa. Mais adiante neste capítulo, na Seção 1.5 (página 35), discutimos espalhamento, uma estrutura de dados randomizada que suporta inserções, remoções e busca. De novo, na prática, ela leva a acesso mais rápido a dados do que esquemas determinísticos, como árvores binárias de busca.

Há duas variedades de algoritmos randomizados. Algoritmos *Monte Carlo*, por um lado, sempre executam rápido, mas seus resultados apresentam uma chance pequena de estar incorretos; o teste de primalidade é um exemplo. Algoritmos *Las Vegas*, por outro lado, sempre resultam em uma resposta correta, mas garantem um tempo de execução curto com probabilidade alta. Exemplos desse tipo são os algoritmos randomizados para ordenação e busca da mediana descritos no Capítulo 2 (páginas 50 e 53, respectivamente).

O algoritmo mais rápido conhecido para o problema do corte mínimo é o randomizado Monte Carlo, descrito na página 139. Randomização tem um papel importante também em heurísticas; estas são descritas na Seção 9.3. E, por fim, o algoritmo quântico para fatoração (Seção 10.7) funciona de maneira bastante semelhante ao algoritmo randomizado, tendo seu resultado correto com probabilidade alta — exceto que ele deriva sua aleatoriedade não do lançamento de moedas, mas do princípio da superposição em mecânica quântica.

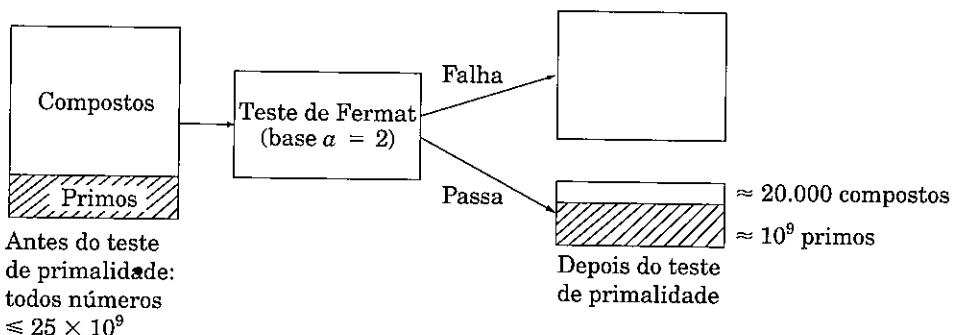
Exercícios virtuais: 1.29, 1.34, 1.46, 2.24, 2.33, 5.35, 9.8, 10.8.

Assim, a cada iteração, tal procedimento tem uma chance de pelo menos $1/n$ de parar. Portanto em média ele vai parar dentro de $O(n)$ rodadas (Exercício 1.34).

Agora, exatamente qual teste de primalidade deve ser usado? Nesta aplicação, como os números cuja primalidade estamos testando são escolhidos aleatoriamente, e não por um adversário, é suficiente realizar o teste de Fermat com base $a = 2$ (ou para ser realmente garantido, $a = 2, 3, 5$), porque para números aleatórios o teste de Fermat tem

uma probabilidade de falha muito menor do que a cota de pior caso $1/2$ que provamos antes. Números que passam neste teste têm sido jocosamente chamados de “primos de nível industrial”. O algoritmo resultante é bastante rápido, gerando primos que têm centenas de bits de comprimento em uma fração de segundo em um PC.

A questão importante que resta é: qual a probabilidade de que a saída do algoritmo seja realmente um primo? Para respondermos, temos primeiro de entender quão perspicaz é o teste de Fermat. Como exemplo concreto, suponha que realizemos o teste com base $a = 2$ para todos os números $N \leq 25 \times 10^9$. Nesse intervalo, há cerca de 10^9 primos e cerca de 20.000 compostos que passam no teste (veja a figura seguinte). Portanto a chance de gerar erradamente um composto é aproximadamente $20.000/10^9 = 2 \times 10^{-5}$. Essa chance de erro decresce rapidamente à medida que aumentamos o comprimento dos números envolvidos (até as poucas centenas de dígitos que nossas aplicações esperam).

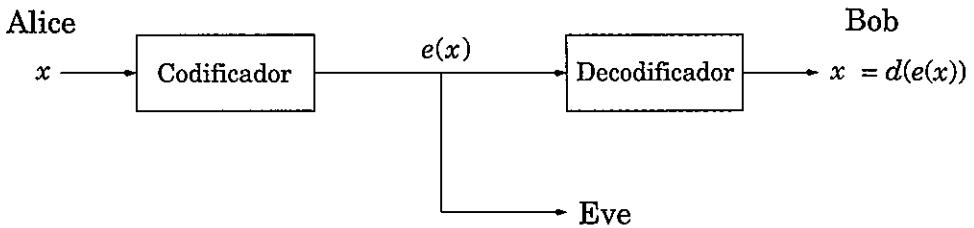


1.4 Criptografia

Nosso próximo tópico, o sistema de criptografia Rivest-Shamir-Adleman (RSA), usa todas as idéias que introduzimos neste capítulo! Ele deriva garantias muito fortes de segurança ao explorar engenhosamente o largo vão entre a computabilidade em tempo polinomial de certas tarefas em teoria de números (exponenciação modular, máximo divisor comum, teste de primalidade) e a intratabilidade de outras (fatoração).

A cena típica para a criptografia pode ser descrita com um elenco de três personagens: Alice e Bob, que desejam se comunicar em privacidade, e Eve, uma bisbilhoteira que fará de tudo para descobrir o que eles estão dizendo. Em outras palavras, digamos que Alice quer mandar uma mensagem específica x , escrita em binário (por que não), para seu amigo Bob. Ela a codifica como $e(x)$, a envia e, então, Bob aplica sua função de decodificação $d(\cdot)$ para decodificá-la: $d(e(x)) = x$. Aqui $e(\cdot)$ e $d(\cdot)$ são transformações apropriadas das mensagens.

Alice e Bob preocupam-se com a possibilidade de Eve interceptar $e(x)$: por exemplo, ela pode estar monitorando a rede. Mas idealmente a função de codificação $e(\cdot)$ é



escolhida de tal maneira que, sem conhecer $d(\cdot)$, Eve não pode fazer coisa alguma com a informação que apanhou. Em outras palavras, o fato de Eve conhecer $e(x)$ lhe diz pouco ou nada sobre o que pode ser x .

Por séculos, criptografia foi baseada no que hoje chamamos de *protocolos de chave privada*. Nesse esquema, Alice e Bob se encontram antes e, juntos, escolhem um livro secreto de códigos, com o qual codificam todas as futuras correspondências entre eles. A única esperança de Eve é coletar algumas mensagens codificadas e usá-las para pelo menos descobrir parcialmente o livro de códigos.

Esquemas de *chave pública* tal como o RSA são bem mais sutis e engenhosos: permitem a Alice mandar uma mensagem a Bob sem que eles nunca tenham se encontrado antes. A função de codificação $e(\cdot)$ de Bob está disponível publicamente, e Alice pode codificar a mensagem dela com esta função, dessa forma *travando digitalmente* a mensagem. Somente Bob conhece a chave para rapidamente abrir a trava digital: a função de decodificação $d(\cdot)$. A questão é que Alice e Bob precisam apenas fazer cálculos simples para travar e destravar a mensagem, respectivamente — operações que qualquer dispositivo computacional de bolso pode realizar. Já, para destravar a mensagem sem a chave, Eve tem de realizar operações como fatoração de números grandes, que requerem mais poder computacional do que poderia ser obtido pelos mais sofisticados computadores do mundo combinados. Essa garantia concernente possibilita o comércio seguro em rede, tal como enviar números de cartões de crédito para companhias pela Internet.

1.4.1 Esquemas de chave privada: cifra de uso único e AES

Se Alice quer enviar uma mensagem importante para Bob, seria prudente da parte dela embaralhar a mensagem com uma função de codificação,

$$e : \langle \text{mensagens} \rangle \rightarrow \langle \text{mensagens codificadas} \rangle.$$

Claro, essa função tem de ser inversível — para possibilitar a decodificação — e é, portanto, uma bijeção. Sua inversa é a função de decodificação $d(\cdot)$.

Na cifra de uso único (*one-time pad*), Alice e Bob se encontram antes e, secretamente, escolhem uma string binária r do mesmo tamanho — digamos, n bits — que a mensagem importante x enviada por Alice posteriormente. A função de codificação de Alice é um *ou-exclusivo bit a bit*, $e_r(x) = x \oplus r$: cada posição na mensagem codificada é o ou-exclusivo das correspondentes posições em x e r . Por exemplo, se $r = 01110010$ a mensagem 11110000 é embaralhada assim:

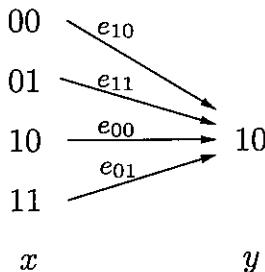
$$e_r(11110000) = 11110000 \oplus 01110010 = 10000010.$$

Essa função e_r é uma bijeção de *strings* de n bits para *strings* de n bits, como evidenciado pelo fato de ela ser sua própria inversa!

$$e_r(e_r(x)) = (x \oplus r) \oplus r = x \oplus (r \oplus r) = x \oplus \bar{0} = x,$$

onde $\bar{0}$ é a *string* só de zeros. Assim Bob pode decodificar a transmissão de Alice aplicando a mesma função de codificação uma segunda vez: $d_r(y) = y \oplus r$.

Como devem Alice e Bob escolher r para que esse esquema seja seguro? Simples: devem escolher r aleatoriamente, lançando uma moeda para cada bit, tal que a *string* resultante pode ser qualquer elemento de $\{0, 1\}^n$ com igual probabilidade. Isso garante que se Eve interceptar a mensagem codificada $y = e_r(x)$, ela não obterá nenhuma informação sobre x . Suponha, por exemplo, que Eve descubra $y = 10$, o que ela pode deduzir? Eve não conhece r , e todos os possíveis valores que pode tomar correspondem a mensagens originais x diferentes:



Portanto, dado o que Eve conhece, todas as possibilidades para x são igualmente prováveis!

A desvantagem da cifra de uso único é que tem de ser descartada depois de usada, daí o nome. Uma segunda mensagem codificada com a mesma cifra não seria segura, porque se Eve conhece $x \oplus r$ e $z \oplus r$ para duas mensagens x e z , ela poderia usar o ou-exclusivo para obter $x \oplus z$, que pode ser informação importante, por exemplo, (1) isso revela se as duas mensagens começam ou terminam iguais, e (2) se uma mensagem contém uma seqüência longa de zeros (como poderia facilmente ser o caso se a mensagem é uma imagem), desse modo a parte correspondente da outra mensagem seria exposta. Portanto, a *string* aleatória que Alice e Bob compartilham tem de ser do tamanho total de todas as mensagens que eles precisam trocar.

A cifra de uso único é um esquema criptográfico trivial cujo comportamento e propriedades teóricas são completamente claros. No outro extremo do espectro, está o *advanced encryption standard* (AES), um protocolo largamente usado que foi aprovado pelo U.S. National Institute of Standards and Technologies em 2001. O AES é novamente chave privada: Alice e Bob têm de concordar com uma *string* aleatória comum r . Mas agora a *string* tem um tamanho fixo pequeno, 128 bits para ser preciso (variantes com 192 ou 256 bits também existem), e especifica uma bijeção e_r de *strings* de 128 bits para *strings* de 128 bits. A diferença crucial é que essa função pode ser usada repetidamente, de maneira que uma mensagem longa pode ser codificada dividindo-a em segmentos de 128 bits e aplicando e_r a cada segmento.

A segurança do AES não foi estabelecida rigorosamente, mas é certo que hoje o público em geral não sabe como quebrar o código — restaurar x a partir de $e_r(x)$ —, exceto usando técnicas não muito melhores do que a abordagem força bruta de tentar todas as possibilidades para a string compartilhada r .

1.4.2 RSA

Ao contrário dos dois protocolos anteriores, o esquema RSA é um exemplo de *criptografia de chave pública*: qualquer um pode enviar uma mensagem para qualquer outra pessoa usando informação pública disponível, de maneira semelhante a endereços e números de telefone. Cada pessoa tem uma chave pública conhecida por todos e uma chave secreta conhecida somente por ela. Quando Alice envia mensagem x para Bob, ela a codifica usando a chave pública dele. Ele decodifica a mensagem usando sua chave secreta, para restaurar x . Eve pode ver tantas mensagens codificadas para Bob quantas quiser, mas não será capaz de decodificá-las, sob certas hipóteses simples.

O esquema RSA é baseado fortemente em teoria de números. Pense nas mensagens de Alice para Bob como números módulo N ; mensagens maiores do que N podem ser quebradas em pedaços menores. A função de codificação será, então, uma bijeção sobre $\{0, 1, \dots, N - 1\}$ e a função de decodificação será a sua inversa. Quais valores de N são apropriados e qual bijeção deve ser usada?

Propriedade Selecione quaisquer dois primos p e q e seja $N = pq$. Para qualquer e primo relativo de $(p - 1)(q - 1)$:

1. O mapeamento $x \mapsto x^e \bmod N$ é uma bijeção sobre $\{0, 1, \dots, N - 1\}$.
2. Além disso, o mapeamento inverso pode ser realizado facilmente: seja d o inverso de e módulo $(p - 1)(q - 1)$. Então para todo $x \in \{0, \dots, N - 1\}$,

$$(x^e)^d \equiv x \bmod N.$$

A primeira propriedade nos diz que o mapeamento $x \mapsto x^e \bmod N$ é uma maneira razoável de codificar mensagens x ; nenhuma informação é perdida. Portanto, se Bob publica (N, e) como sua *chave pública*, qualquer outra pessoa pode usar isso para enviar mensagens codificadas para ele. A segunda propriedade nos diz como a decodificação pode ser alcançada. Bob deve manter o valor d como sua *chave secreta*, com a qual pode decodificar todas as mensagens que chegam a ele apenas elevando-as à potência d , módulo N .

Exemplo. Seja $N = 55 = 5 \cdot 11$. Escolha o expoente de codificação $e = 3$, que satisfa faz a condição $\text{mdc}(e, (p - 1)(q - 1)) = \text{mdc}(3, 40) = 1$. O expoente de decodificação é, então, $d = 3^{-1} \bmod 40 = 27$. Agora para qualquer mensagem $x \bmod 55$, a codificação de x é $y = x^3 \bmod 55$, e a decodificação de y é $x = y^{27} \bmod 55$. Assim, por exemplo, se $x = 13$, então $y = 13^3 = 52 \bmod 55$ e $13 = 52^{27} \bmod 55$.

Vamos provar a assertiva anterior e examinar a segurança do esquema.

Prova. Se o mapeamento $x \mapsto x^e \pmod{N}$ é inversível, ele tem de ser uma bijeção; portanto a sentença 2 implica a sentença 1. Para provarmos a sentença 2, começamos observando que e é inversível módulo $(p-1)(q-1)$ porque é primo relativo deste número. Para entendermos que $(x^e)^d \equiv x \pmod{N}$, examinamos o expoente: como $ed \equiv 1 \pmod{(p-1)(q-1)}$, podemos escrever ed na forma de $1 + k(p-1)(q-1)$ para algum k . Agora precisamos mostrar que a diferença

$$x^{ed} - x = x^1 + k(p-1)(q-1) - x$$

sempre é 0 módulo N . A segunda forma da expressão é conveniente, pois pode ser simplificada por meio do pequeno teorema de Fermat. Ela é divisível por p (pois $x^{p-1} \equiv 1 \pmod{p}$) e igualmente por q . Como p e q são primos, esta expressão tem de ser divisível também pelo produto módulo N deles. Portanto $x^{ed} - x = x^1 + k(p-1)(q-1) - x \equiv 0 \pmod{N}$, exatamente como precisávamos. ■

Figura 1.9 RSA.

Bob escolhe suas chaves pública e secreta.

- Ele inicia selecionando dois primos aleatórios grandes (n bits) p e q .
- Sua chave pública é (N, e) onde $N = pq$ e e é um número de $2n$ bits primo relativo de $(p-1)(q-1)$. Uma escolha comum é $e = 3$ porque permite codificação rápida.
- Sua chave secreta é d , o inverso de e módulo $(p-1)(q-1)$, computado usando o algoritmo estendido de Euclides.

Alice deseja enviar uma mensagem x para Bob.

- Ela pega a chave pública dele (N, e) e envia $y = (x^e \pmod{N})$, computado por meio de um algoritmo eficiente de exponenciação modular.
- Ela decodifica a mensagem computando $y^d \pmod{N}$.

O protocolo RSA está resumido na Figura 1.9. Ele é certamente conveniente: as computações que requer de Alice e Bob são elementares. Mas quão seguro ele é contra Eve?

A segurança do RSA se baseia em uma hipótese simples:

Dados N, e , e $y = x^e \pmod{N}$, é computacionalmente intratável determinar x .

Essa hipótese é bastante plausível. Como pode Eve tentar adivinhar x ? Ela poderia experimentar todos os possíveis valores de x , checando toda vez se $x^e \equiv y \pmod{N}$, mas isso tomaria tempo exponencial. Ou ela poderia tentar fatorar N para restaurar p e q e, então, descobrir d invertendo e módulo $(p-1)(q-1)$, mas acreditamos que fatoração é difícil. Intratabilidade em geral é uma fonte de tormento; a idéia do RSA é se aproveitar disso.

1.5 Espalhamento universal

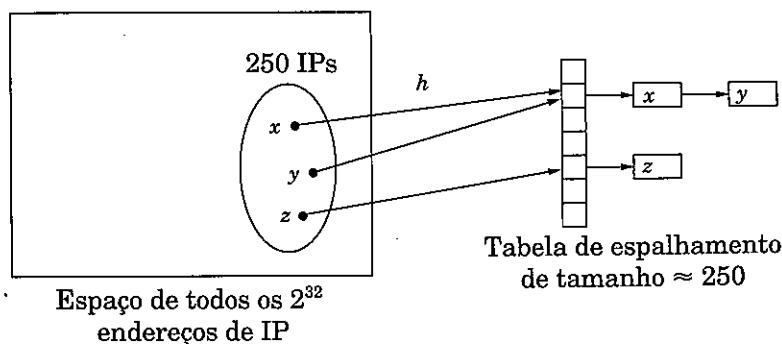
Encerramos este capítulo com uma aplicação da teoria dos números ao projeto de funções de espalhamento (*hash functions*). Espalhamento é um método bastante útil para guardar itens de dados em uma tabela, para suportar inserções, remoções e busca.

Suponha, por exemplo, que precisemos manter uma lista, que está sempre mudando, de cerca de 250 endereços de IP (protocolo de Internet), talvez os endereços dos usuários correntes de um Web service. (Lembre-se de que um endereço de IP consiste em 32 bits codificando a localização de um computador na Internet, usualmente mostrado com quatro campos de 8 bits, por exemplo 128.32.168.80.) Poderíamos obter tempos de busca rápidos se mantivéssemos os registros em um vetor indexado por endereços de IP. Mas isso seria muito desperdício de memória: o vetor teria $2^{32} \approx 4 \times 10^9$ entradas, a vasta maioria delas em branco. Ou, alternativamente, poderíamos usar uma lista ligada de apenas 250 registros. Mas, então, acessar registros seria muito lento, tomando tempo proporcional a 250, o número total de usuários. Será que há uma maneira de ter o melhor dos dois mundos: usar uma quantidade de memória proporcional ao número de usuários e, ainda assim, alcançar tempos de busca rápidos? É exatamente aqui que o espalhamento entra.

1.5.1 Tabelas de espalhamento

Apresentamos a seguir uma visão em alto nível de espalhamento. Vamos dar um “apelido” a cada um dos 2^{32} possíveis endereços de IP. Você pode pensar neste nome curto como simplesmente um número entre 1 e 250 (mais tarde ajustaremos um pouco esse intervalo). Assim, muitos endereços de IP vão necessariamente ter o mesmo apelido. Entretanto, esperamos que para a maioria dos 250 endereços de IP dos nossos específicos usuários sejam designados nomes distintos; e vamos guardar seus registros em um vetor de tamanho 250, indexado por esses nomes. O que acontece se houver mais de um registro associado ao mesmo nome? Fácil: cada entrada do vetor aponta para uma lista ligada contendo todos os registros com aquele nome. Desse modo, o armazenamento total é proporcional a 250, o número de usuários, e é independente do número total de endereços possíveis de IP. Além disso, se não forem designados endereços de IP demais a um mesmo nome, a busca será rápida, porque o tamanho médio da lista ligada de que precisamos varrer é pequeno.

Mas como designamos um nome curto a cada endereço de IP? Esse é o papel da função de espalhamento: no nosso exemplo, uma função h que mapeia endereços de IP em posições em uma tabela de aproximadamente 250 de comprimento (o número esperado



de itens de dados). O nome designado a um endereço de IP x é assim $h(x)$ e o registro para x é guardado na posição $h(x)$ da tabela. Como já descrito, cada posição da tabela é na verdade um *recipiente*, uma lista ligada que contém todos os endereços de IP correntes que mapeiam para ela. Com sorte, haverá bem poucos recipientes que contenham mais do que uns poucos endereços de IP.

1.5.2 Famílias de funções de espalhamento

Projetar funções de espalhamento é complicado. Uma função de espalhamento tem de ser em certo sentido “aleatória” (para que espalhe os itens de dados por todos os lados), mas deve ser também uma função e, portanto, “consistente” (para obtermos o mesmo resultado toda vez que a aplicamos). E a estatística dos itens de dados pode trabalhar contra nós. Em nosso exemplo, uma possível função de espalhamento poderia mapear um endereço de IP ao número de 8 bits que é o seu último segmento: $h(128.32.168.80) = 80$. Uma tabela de $n = 256$ recipientes seria então requerida. Mas, será essa uma boa função de espalhamento? Não se, por exemplo, o último segmento de um endereço de IP tende a ser um número pequeno (um ou dois dígitos); pois os recipientes de numeração menor ficariam congestionados. Tomar o primeiro segmento do endereço de IP também é um convite ao desastre — por exemplo, se a maioria dos usuários vier da Ásia.

Não há nada inherentemente errado com essas duas funções. Se nossos 250 endereços de IP fossem tomados uniformemente entre todas as $N = 2^{32}$ possibilidades, então essas funções se comportariam bem. O problema é que não temos qualquer garantia de que a distribuição dos endereços de IP seja uniforme.

Por sua vez, não há nenhuma função de espalhamento, não importa quão sofisticada, que se comporte bem em todos os conjuntos de dados. Como uma função de espalhamento mapeia 2^{32} endereços de IP a apenas 250 nomes, tem de haver uma coleção de pelo menos $2^{32}/250 \approx 2^{24} \approx 16.000.000$ endereços de IP designados ao mesmo nome (ou, em terminologia de espalhamento, “colidem”). Se muitos desses aparecerem no nosso conjunto de usuários, estaremos com problemas.

Obviamente, precisamos de alguma forma de randomização. Aqui está uma idéia: peguemos uma função de espalhamento *aleatoriamente* de alguma classe de funções. Vamos, então, mostrar que, não importa de qual conjunto de 250 endereços de IP estamos tratando, a maioria das escolhas de função de espalhamento levará a muito poucas colisões entre endereços.

Para isso, precisamos definir uma classe de funções de espalhamento da qual podemos pegar aleatoriamente uma função; e é aqui que nos voltamos para teoria de números. Tomemos para o número de recipientes não 250, mas $n = 257$ — *um número primo!* E consideremos cada endereço de IP como uma quádrupla $x = (x_1, \dots, x_4)$ de inteiros módulo n — lembre-se de que ele é de fato uma quádrupla de inteiros entre 0 e 255, de modo que não há qualquer perigo nisso. Podemos definir uma função h de endereços de IP em um número módulo n como se segue: fixe quaisquer quatro números módulo $n = 257$, digamos 87, 23, 125 e 4. Depois mapeie o endereço de IP (x_1, \dots, x_4) a $h(x_1, \dots, x_4) = (87x_1 + 23x_2 + 125x_3 + 4x_4) \bmod 257$. De fato, quaisquer quatro números módulo n definem uma função de espalhamento.

Para quaisquer quatro coeficientes $a_1, \dots, a_4 \in \{0, 1, \dots, n - 1\}$ escreva $a = (a_1, a_2, a_3, a_4)$ e defina h_a como a seguinte função de espalhamento:

$$h_a(x_1, \dots, x_4) = \sum_{i=1}^4 a_i \cdot x_i \bmod n.$$

Mostraremos que se pegarmos esses coeficientes a aleatoriamente, h_a tem muita probabilidade de ser boa no seguinte sentido.

Propriedade Considera qualquer par de endereços distintos de IP $x = (x_1, \dots, x_4)$ e $y = (y_1, \dots, y_4)$. Se os coeficientes $a = (a_1, a_2, a_3, a_4)$ são escolhidos uniforme e aleatoriamente de $\{0, 1, \dots, n - 1\}$, então

$$\Pr\{h_a(x_1, \dots, x_4) = h_a(y_1, \dots, y_4)\} = \frac{1}{n}.$$

Em outras palavras, a chance de colisão entre x e y sobre h_a é a mesma que seria se a cada fosse designado um apelido aleatório e independentemente. Essa condição garante que o tempo esperado de busca para qualquer item seja pequeno. Aqui está a razão. Se desejamos buscar x na nossa tabela, o tempo requerido é dominado pelo tamanho do seu recipiente, ou seja, o número de itens aos quais foi designado o mesmo nome que x . Mas há apenas 250 itens na tabela de espalhamento e a probabilidade de que um número qualquer tenha o mesmo nome de x é $1/n = 1/257$. Portanto, o número esperado de itens aos quais foi designado o mesmo nome que x por uma função de espalhamento h_a escolhida aleatoriamente é $250/257 \approx 1$, o que significa que o tamanho esperado do recipiente de x é menor do que 2^1 .

Demonstremos agora a propriedade anterior.

Prova Como $x = (x_1, \dots, x_4)$ e $y = (y_1, \dots, y_4)$ são distintos, estas quadrúplas têm de diferir em algum componente; sem perda de generalidade vamos assumir que $x_4 \neq y_4$. Desejamos computar a probabilidade $\Pr[h_a(x_1, \dots, x_4) = h_a(y_1, \dots, y_4)]$, isto é, a probabilidade de que $\sum_{i=1}^4 a_i \cdot x_i \equiv \sum_{i=1}^4 a_i \cdot y_i \pmod{n}$. Esta última equação pode ser reescrita como

$$\sum_{i=1}^3 a_i \cdot (x_i - y_i) \equiv a_4 \cdot (y_4 - x_4) \pmod{n}. \quad (1)$$

Suponha que escolhamos uma função h_a aleatoriamente selecionando $a = (a_1, a_2, a_3, a_4)$ de modo aleatório. Começamos por escolher, a_1, a_2 e a_3 , e, então, fazemos uma pausa e pensamos: qual a probabilidade de que o último número a_4 seja tal que a equação (1) valha? Até agora o lado esquerdo da equação (1) resulta em algum número, chamemos de c . E como n é primo e $x_4 \neq y_4$, $(y_4 - x_4)$ possui um inverso módulo n . Assim, para equação (1) valer, o último número a_4 tem de ser precisamente $c \cdot (y_4 - x_4)^{-1} \pmod{n}$, de seus n possíveis valores. A probabilidade de que isso ocorra é $1/n$ e a prova está completa. ■

Vamos retroceder e analisar o que acabamos de alcançar. Como não temos qualquer controle sobre o conjunto de itens de dados, decidimos selecionar uma função de espa-

¹Quando uma função de espalhamento h_a é escolhida aleatoriamente, seja a variável aleatória Y_i (para $i = 1, \dots, 250$) igual a 1 se o item i receber o mesmo nome que x e 0 em caso contrário. Assim, o valor esperado de Y_i é $1/n$. Agora, $Y = Y_1 + Y_2 + \dots + Y_{250}$ é o número de itens que recebem o mesmo nome que x e, por linearidade do valor esperado, o valor esperado de Y é apenas a soma dos valores esperados de Y_i até Y_{250} . É, portanto, $250/n = 250/257$.

lhamento h uniforme e aleatoriamente de uma família \mathcal{H} de funções de espalhamento. No nosso exemplo,

$$\mathcal{H} = \{h_a : a \in \{0, \dots, n-1\}^4\}.$$

Para escolhermos uniforme e aleatoriamente uma função dessa família, apenas escolhemos 4 números a_1, \dots, a_4 módulo n . (A propósito, note que as duas funções de espalhamento simples que consideramos antes, a saber, tomar o último ou o primeiro segmento de 8 bits, pertencem a essa classe. Elas são $h_{(0, 0, 0, 1)}$ e $h_{(1, 0, 0, 0)}$, respectivamente.) E insistimos em que a família tivesse a seguinte propriedade:

Para quaisquer dois itens de dados distintos x e y , exatamente $|\mathcal{H}|/n$ de todas as funções de espalhamento em \mathcal{H} mapeiam x e y ao mesmo recipiente, onde n é o número de recipientes.

Uma família de funções de espalhamento com essa propriedade é chamada *universal*. Em outras palavras, para quaisquer dois itens de dados, a probabilidade de eles colidirem é $1/n$ se a função de espalhamento é selecionada aleatoriamente de um famílio universal. Essa também é a probabilidade de colisão se mapearmos x e y uniforme e aleatoriamente — em certo sentido o padrão ouro de espalhamento. Nós, então, mostramos que essa propriedade implica que as operações em tabela de espalhamento tenham um bom desempenho em *valor esperado*.

Tal idéia, motivada como foi pela aplicação hipotética em endereços de IP, pode claro ser aplicada mais geralmente. Comece por escolher o tamanho n da tabela como um número primo que seja um pouco maior do que o número de itens esperados na tabela (existe normalmente um número primo perto de qualquer número pelo qual comecemos; na verdade, para assegurar que as operações na tabela de espalhamento tenham bom desempenho, é melhor que o tamanho da tabela de espalhamento seja cerca de o dobro do número de itens). Depois suponha que o tamanho do domínio de todos os itens de dados seja $N = n^k$, uma potência de n (se precisarmos superestimar o verdadeiro número de itens de dados, que seja). Então cada item de dados pode ser considerado uma k -tupla de inteiros módulo n e $\mathcal{H} = \{h_a : a \in \{0, \dots, n-1\}^k\}$ é a família universal de funções de espalhamento.

Exercícios

- 1.1. Mostre que, em qualquer base $b \geq 2$, a soma de quaisquer três números de um dígito tem comprimento de dois dígitos no máximo.
- 1.2. Mostre que qualquer inteiro binário tem comprimento no máximo quatro vezes maior que o correspondente inteiro decimal. Para números muito grandes, qual é a razão entre esses dois comprimentos, aproximadamente?
- 1.3. Uma árvore d -ária é uma árvore com raiz na qual cada nó tem no máximo d filhos. Mostre que qualquer árvore d -ária com n nós tem de ter uma profundidade de $\Omega(\log n / \log d)$. Você pode dar uma fórmula precisa para a profundidade mínima que ela pode ter?
- 1.4. Mostre que

$$\log(n!) = \Theta(n \log n).$$

(Dica: Para mostrar uma cota superior, compare $n!$ com n^n . Para mostrar uma cota inferior, compare com $(n/2)^{n/2}$.)

- 1.5. Ao contrário de uma série geométrica decrescente, a somatória da série harmônica $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \dots$ diverge; ou seja,

$$\sum_{i=1}^{\infty} \frac{1}{i} = \infty.$$

Acontece que, para n grande, a somatória dos primeiros n termos dessa série pode ser bem aproximada como

$$\sum_{i=1}^n \frac{1}{i} \approx \ln n + \gamma,$$

onde \ln é o logaritmo natural (\log base $e = 2,718\dots$) e γ é uma particular constante $0,57721\dots$. Mostre que

$$\sum_{i=1}^n \frac{1}{i} = \Theta(\log n).$$

(Dica: Para mostrar a cota superior, reduza cada denominador até a próxima potência de 2. Para a cota inferior, aumente cada denominador até a próxima potência de 2.)

- 1.6. Prove que o algoritmo da escola para multiplicação (página 13), quando aplicado a números binários, sempre fornece a resposta correta.
- 1.7. Quanto tempo o algoritmo recursivo para multiplicação (página 15) toma para multiplicar um número de n bits por um de m bits? Justifique sua resposta.
- 1.8. Justifique a correção do algoritmo recursivo para divisão dado na página 15 e mostre que ele toma tempo $O(n^2)$ sobre entradas de n bits.
- 1.9. Começando pela definição de $x \equiv y \pmod N$ (a saber, que N divide $x - y$), justifique a regra de substituição

$$x \equiv x' \pmod N, y \equiv y' \pmod N \Rightarrow x + y \equiv x' + y' \pmod N,$$

e, também, a correspondente regra para multiplicação.

- 1.10. Mostre que se $a \equiv b \pmod N$ e se M divide N , então $a \equiv b \pmod M$.
- 1.11. É $4^{1536} - 9^{4824}$ divisível por 35?
- 1.12. Quanto é $2^{2^{2006}} \pmod 3$?
- 1.13. A diferença entre $5^{30.000}$ e $6^{123.456}$ é um múltiplo de 31?
- 1.14. Suponha que você queira computar o n -ésimo número de Fibonacci F_n , use um módulo para inteiro p . Você pode encontrar uma maneira mais eficiente de fazer isso? (Dica: Reveja o Exercício 0.4.)
- 1.15. Determine condições necessárias e suficientes sobre x e c tal que o seguinte valha: para quaisquer a, b , se $ax \equiv bx \pmod c$, então $a \equiv b \pmod c$.
- 1.16. O algoritmo para computar $a^b \pmod c$ por sucessivas elevações ao quadrado não necessariamente leva ao número mínimo de multiplicações. Dê um exemplo de $b > 10$ para o qual a exponenciação pode ser realizada, usando menos multiplicações, por algum outro método.

- 1.17. Considere o problema de computar x^y para dados inteiros x e y : queremos a resposta completa, não abrir um módulo de outro inteiro. Conhecemos dois algoritmos para fazer isso: o algoritmo iterativo que realiza $y - 1$ multiplicações por x ; e o algoritmo recursivo baseado na expansão binária de y . Compare os tempos de execução desses dois algoritmos, considerando que o tempo de multiplicar um número de n bits por outro de m bits seja $O(mn)$.
- 1.18. Compute o mdc(210, 588) de duas maneiras diferentes: encontrando a fatoração de cada número e usando o algoritmo de Euclides.
- 1.19. Os números de Fibonacci F_0, F_1, \dots são dados pela recorrência $F_{n+1} = F_n + F_{n-1}$, $F_0 = 0$, $F_1 = 1$. Mostre que para qualquer $n \geq 1$, $\text{mdc}(F_{n+1}, F_n) = 1$.
- 1.20. Encontre o inverso de: $20 \bmod 79$, $3 \bmod 62$, $21 \bmod 91$, $5 \bmod 23$.
- 1.21. Quantos inteiros módulo 11^3 possuem inversos? (Note que: $11^3 = 1331$.)
- 1.22. Prove ou demonstre o contrário: Se a tem um inverso módulo b , então, b possui um inverso módulo a .
- 1.23. Mostre que se a possui um inverso multiplicativo módulo N , então, este inverso é único (módulo N).
- 1.24. Se p é primo, quantos elementos de $\{0, 1, \dots, p^n - 1\}$ têm um inverso módulo p^n ?
- 1.25. Calcule $2^{125} \bmod 127$ usando qualquer método que você quiser. (Dica: 127 é primo.)
- 1.26. Qual é o dígito menos significativo de 17^{1717} ? (Dica: Para primos distintos p, q e qualquer a primo relativo de pq , demonstramos a fórmula $a^{(p-1)(q-1)} \equiv 1 \pmod{pq}$ na Seção 1.4.2.)
- 1.27. Considere um conjunto de chaves RSA com $p = 17$, $q = 23$, $N = 391$ e $e = 3$ (como na Figura 1.9). Qual valor de d deve ser usado como chave secreta? Qual é a codificação da mensagem $M = 41$?
- 1.28. Em um sistema de criptografia RSA, $p = 7$ e $q = 11$ (como na Figura 1.9). Encontre expoentes apropriados d e e .
- 1.29. Denote por $[m]$ o conjunto $\{0, 1, \dots, m - 1\}$. Para cada uma das seguintes famílias de funções de espalhamento, diga se ela é ou não universal e determine quantos bits aleatórios são necessários para escolher uma função da família.
- $H = \{h_{a_1, a_2} : a_1, a_2 \in [m]\}$, onde m é um primo fixo e
- $$h_{a_1, a_2}(x_1, x_2) = a_1 x_1 + a_2 x_2 \bmod m.$$
- Note que cada uma dessas funções possui uma assinatura: $h_{a_1, a_2} : [m]^2 \rightarrow [m]$, isto é, elas mapeiam um par de inteiros em $[m]$ a um inteiro apenas em $[m]$.
- H como antes, exceto que agora $m = 2^k$ é alguma potência de 2 fixa.
 - H é o conjunto de todas as funções $f : [m] \rightarrow [m - 1]$.
- 1.30. O algoritmo da escola para multiplicação de dois números binários de n bits x e y consiste em adicionar n cópias de x , cada uma apropriadamente deslocada para a esquerda. Cada cópia, quando deslocada, tem no máximo $2n$ bits.

Neste problema, vamos examinar um esquema para adicionar n números binários, cada um com m bits, usando um circuito ou uma arquitetura paralela. O principal parâmetro de interesse nesta questão é, portanto, a profundidade do circuito ou o caminho mais longo entre a entrada e a saída do circuito. Isso determina o tempo total tomado para computar a função.

Para adicionarmos dois números binários de m bits ingenuamente, precisamos esperar pelo bit de excesso da posição $i - 1$ antes de descobrir o i -ésimo bit da resposta. Isso leva a um circuito de profundidade $O(m)$. Entretanto, circuitos de *carry lookahead* (veja em [wikipedia.com](https://en.wikipedia.org/wiki/Carry_lookahead), se você quiser saber mais sobre isso) podem adicionar em profundidade $O(\log m)$.

- (a) Supondo que você tenha circuitos de *carry lookahead* para adição, mostre como adicionar n números de m bits usando um circuito de profundidade $O((\log n)(\log m))$.
- (b) Ao adicionarmos três números binários de m bits $x + y + z$, há um truque que podemos usar para paralelizar o processo. Em vez de realizarmos a adição completamente, podemos reexpressar o resultado como uma soma de apenas dois números binários $r + s$, tal que os i -ésimos bits de r e s podem ser computados independentemente dos outros bits. Mostre como isso pode ser feito. (*Dica:* Um dos números representa os bits de excesso.)
- (c) Mostre como usar o truque da parte anterior para projetar um circuito de profundidade $O(\log n)$ para multiplicar dois números de n bits.

- 1.31. Considere o problema de computar $N! = 1 \cdot 2 \cdot 3 \cdots N$.
 - (a) Se N é um número de n bits, quantos bits de comprimento tem $N!$, aproximadamente (na forma $\Theta(\cdot)$)?
 - (b) Forneça um algoritmo para computar $N!$ e analise seu tempo de execução.
- 1.32. Um inteiro positivo N é uma *potência* se ele é da forma q^k , onde q e k são inteiros positivos e $k > 1$.
 - (a) Forneça um algoritmo eficiente que tome como entrada um inteiro N e determine se ele é um quadrado, isto é, se ele pode ser escrito como q^2 para algum inteiro positivo q . Qual é o tempo de execução do seu algoritmo?
 - (b) Mostre que se $N = q^k$ (com N, q e k todos inteiros positivos), então, ou $k \leq \log N$ ou $N = 1$.
 - (c) Forneça um algoritmo eficiente para determinar se um inteiro positivo N é uma potência. Analise seu tempo de execução.
- 1.33. Forneça um algoritmo eficiente para computar o *mínimo múltiplo comum* de dois números de n bits x e y , ou seja, o menor número divisível tanto por x quanto por y . Qual o tempo de execução do seu algoritmo como uma função de n ?
- 1.34. Na página 29, afirmamos que, como uma fração de cerca de $1/n$ dos números de n bits são primos, na média é suficiente selecionar $O(n)$ números de n bits aleatoriamente antes de obter um primo. Vamos agora justificar isso rigorosamente.

Suponha que uma particular moeda tenha uma probabilidade p de dar cara. Quantas vezes você precisa lançá-la, na média, até que dê cara? (*Dica:* Método 1: comece mostrando que a expressão correta é $\sum_{i=1}^{\infty} i(1-p)^{i-1} p$. Método 2: se E é o número médio de lançamentos da moeda, mostre que $E = 1 + (1-p)E$.)

1.35. O teorema de Wilson diz que um número N é primo se e somente se

$$(N - 1)! \equiv -1 \pmod{N}.$$

- (a) Se p é primo, então, sabemos que todo número $1 \leq x < p$ é inversível módulo p . Quais desses números são seus próprios inversos?
- (b) Colocando inversos multiplicativos em pares, mostre que $(p - 1)! \equiv -1 \pmod{p}$ para p primo.
- (c) Mostre que se N não é primo, então, $(N - 1)! \not\equiv -1 \pmod{N}$. (Dica: Considere $d = \text{mdc}(N, (N - 1)!)$.)
- (d) Ao contrário do pequeno teorema de Fermat, o teorema de Wilson é uma condição se-e-somente-se para primalidade. Por que não podemos imediatamente basear um teste de primalidade nessa regra?

1.36. *Raízes quadradas*. Neste problema, veremos que é fácil computar raízes quadradas módulo um primo p com $p \equiv 3 \pmod{4}$.

- (a) Suponha que $p \equiv 3 \pmod{4}$. Mostre que $(p + 1)/4$ é um inteiro.
- (b) Dizemos que x é uma raiz quadrada de a módulo p se $a \equiv x^2 \pmod{p}$. Mostre que se $p \equiv 3 \pmod{4}$ e se a tem uma raiz quadrada módulo p , então, $a^{(p+1)/4}$ é uma tal raiz quadrada.

1.37. *O teorema chinês do resto*.

- (a) Faça uma tabela com três colunas. A primeira coluna é de todos os números de 0 a 14. A segunda é dos resíduos desses números módulo 3; a terceira coluna é dos resíduos módulo 5.
- (b) Prove que se p e q são primos distintos, para todo par (j, k) com $0 \leq j < p$ e $0 \leq k < q$ existe um único inteiro $0 \leq i < pq$, tal que $i \equiv j \pmod{p}$ e $i \equiv k \pmod{q}$. (Dica: Prove que não pode haver dois i diferentes neste intervalo com o mesmo (j, k) , e, então, conte.)
- (c) Nesta correspondência um-para-um entre inteiros e pares, é fácil ir de i para (j, k) . Prove que a seguinte fórmula leva-o para outra direção:

$$i = \{j \cdot q \cdot (q^{-1} \pmod{p}) + k \cdot p \cdot (p^{-1} \pmod{q})\} \pmod{pq}.$$

- (d) Você pode generalizar as partes (b) e (c) para mais de dois primos?

1.38. Para verificar se um número, digamos 562437487, é divisível por 3, você simplesmente soma os dígitos da sua representação decimal e examina se o resultado é divisível por 3. ($5 + 6 + 2 + 4 + 3 + 7 + 4 + 8 + 7 = 46$, portanto não é divisível por 3.)

Para verificar se o mesmo número é divisível por 11, você pode fazer isto: subdivida o número em pares de dígitos, da direita para a esquerda (87, 74, 43, 62, 5), adicione esses números e veja se a soma é divisível por 11 (se for muito grande, repita).

E o que dizer de 37? Para verificar se o número é divisível por 37, subdivida-o em triplas, da direita para a esquerda (487, 437, 562) adicione as triplas e veja se a soma é divisível por 37.

Isso é verdade para qualquer primo p com exceção de 2 e 5. Ou seja, para qualquer primo $p \neq 2, 5$, existe um inteiro r tal que para vermos se p divide um número decimal n , quebramos n em tuplas de r dígitos decimais (começando da direita), adicionamos essas tuplas e verificamos se a soma é divisível por p .

- (a) Qual é o menor tal r para $p = 13$? E para $p = 17$?
- (b) Mostre que r é um divisor de $p - 1$.

- 1.39. Forneça um algoritmo polinomial para computar $a^{bc} \bmod p$, dados a, b, c e um primo p .
- 1.40. Mostre que se x é uma raiz quadrada não trivial de 1 módulo N , isto é, se $x^2 \equiv 1 \pmod{N}$, mas $x \not\equiv \pm 1 \pmod{N}$, então, N tem de ser composto. (Por exemplo, $4^2 \equiv 1 \pmod{15}$, mas $4 \not\equiv \pm 1 \pmod{15}$; assim 4 é uma raiz quadrada não trivial de 1 módulo 15.)
- 1.41. *Resíduos quadráticos.* Fixe um inteiro positivo N . Dizemos que a é um resíduo quadrático módulo N se existe x tal que $a \equiv x^2 \pmod{N}$.
 - (a) Seja N um primo ímpar e a um resíduo quadrático não-nulo módulo N . Mostre que há exatamente dois valores em $\{0, 1, \dots, N - 1\}$ que satisfazem $x^2 \equiv a \pmod{N}$.
 - (b) Mostre que se N é um primo ímpar, existem exatamente $(N + 1)/2$ resíduos quadráticos em $\{0, 1, \dots, N - 1\}$.
 - (c) Dê um exemplo de inteiros positivos a e N tal que $x^2 \equiv a \pmod{N}$ tenha mais do que duas soluções em $\{0, 1, \dots, N - 1\}$.
- 1.42. Suponha que em vez de usarmos um composto $N = pq$ no sistema de criptografia RSA (Figura 1.9), simplesmente usamos um primo módulo p . Como no RSA, teríamos um expoente de codificação e e a codificação de uma mensagem $m \bmod p$ seria $m^e \bmod p$. Prove que o novo sistema criptográfico não é seguro, apresentando um algoritmo eficiente para decodificação, ou seja, um algoritmo que dados p, e e $m^e \bmod p$ como entrada, computa $m \bmod p$. Justifique a correção e analise o tempo de execução do seu algoritmo de decodificação.
- 1.43. No sistema RSA, a chave pública de Alice (N, e) está disponível para todo mundo. Suponha que sua chave privada d esteja comprometida e se torne conhecida para Eve. Mostre que se $e = 3$ (uma escolha comum), então, Eve pode fatorar N eficientemente.
- 1.44. Alice e seus três amigos são usuários do sistema de criptografia RSA. Seus amigos têm chaves públicas $(N_i, e_i = 3)$, $i = 1, 2, 3$, onde como sempre, $N_i = p_i q_i$ para primos de n bits tomados aleatoriamente p_i, q_i . Mostre que se Alice envia a mesma mensagem de n bits M (codificada com RSA) para cada um de seus amigos, então qualquer pessoa que intercepte todas as três mensagens conseguirá eficientemente restaurar M . (*Dica:* Ajuda aqui ter resolvido o problema 1.37 primeiro.)
- 1.45. *RSA e assinaturas digitais.* Lembre que, no sistema de criptografia de chave pública RSA, cada usuário tem uma chave pública $P = (N, e)$ e uma chave secreta d . Em um *esquema de assinatura digital*, existem dois algoritmos, *assina* e *verifica*. O procedimento *assina* toma uma mensagem e uma chave secreta e resulta em uma assinatura σ . O procedimento *verifica* toma a chave pública (N, e) , a assinatura σ e a mensagem M , e retorna “verdadeiro” se σ pode ter sido criada por *assina* (quando chamada com a mensagem M e a chave secreta corresponde à chave pública (N, e)), e “falso” caso contrário.

- (a) Por que iríamos querer assinaturas digitais?
- (b) Uma assinatura RSA consiste em $\text{assina}(M, d) = M^d \pmod{N}$, onde d é a chave secreta e N é parte da chave pública. Mostre que qualquer um que conheça a chave pública (N, e) pode realizar $\text{verifica}((N, e), M^d, M)$, isto é, pode checar que a assinatura foi realmente criada pela chave privada. Forneça uma implementação e prove a sua correção.
- (c) Gere o seu próprio módulo RSA $N = pq$, chave pública e e chave privada d (não é necessário usar um computador). Tome p e q tal que você tenha um módulo de 4 dígitos e trabalhe manualmente. Agora assine o seu nome usando o expoente privado deste módulo RSA. Para tanto, você precisará especificar alguns mapeamentos um-para-um de *strings* para inteiros em $[0, N - 1]$. Especifique qualquer mapeamento que você quiser. Dê o mapeamento do seu nome para números m_1, m_2, \dots, m_k , então, assine o primeiro número dando o valor $m_1^d \pmod{N}$ e finalmente mostre que $(m_1^d)^e = m_1 \pmod{N}$.
- (d) Alice quer enviar uma mensagem que parece ter sido assinada digitalmente por Bob. Ela nota que a chave pública de Bob é $(17, 391)$. A qual expoente ela deve elevar a sua mensagem?

1.46. *Assinaturas digitais, continuação.* Considere o esquema de assinatura do Exercício 1.45.

- (a) Assinaturas implicam decodificação e, por consequência, risco. Mostre que se Bob concorda em assinar qualquer coisa que lhe solicitam, Eve pode se aproveitar disso e decodificar qualquer mensagem de Alice para Bob.
- (b) Suponha que Bob seja mais cuidadoso e se recuse a assinar mensagens suspeitas se suas assinaturas se parecerem com texto. (Consideramos que uma mensagem escolhida aleatoriamente — ou seja, um número aleatório no intervalo $\{1, \dots, -1\}$ — é muito pouco provável se parecer com texto.) Descreva uma maneira segundo a qual Eve pode de qualquer modo ainda decodificar mensagens de Alice para Bob, fazendo Bob assinar mensagens cujas assinaturas pareçam aleatórias.

Capítulo 2

Algoritmos de divisão-e-conquista

A estratégia de divisão-e-conquista soluciona um problema:

1. Dividindo-o em subproblemas, os quais são instâncias menores do mesmo tipo de problema
2. Solucionando recursivamente esses subproblemas
3. Combinando apropriadamente suas respostas

O trabalho real é feito aos poucos em três lugares diferentes: no particionamento de problemas em subproblemas; na base final da recursão, quando os subproblemas são tão pequenos que podem ser resolvidos direto; e na montagem das respostas parciais. Isso tudo é reunido e coordenado pela estrutura recursiva do núcleo do algoritmo.

Como exemplo introdutório, veremos como essa estrutura leva a um novo algoritmo para multiplicação de números, um que é muito mais eficiente do que o método que aprendemos na escola!

2.1 Multiplicação

O matemático Carl Friedrich Gauss (1777-1855) uma vez notou que embora o produto de dois números complexos

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

pareça envolver quatro multiplicações de números reais, ele pode ser de fato feito com apenas três: ac , bd e $(a + b)(c + d)$, pois

$$bc + ad = (a + b)(c + d) - ac - bd.$$

Na nossa maneira de pensar com a notação O , reduzir o número de multiplicações de quatro para três parece ingenuidade desperdiçada. Mas este aperfeiçoamento modesto torna-se muito significativo *quando aplicado recursivamente*.

Deixemos os números complexos de lado e vamos ver como isto ajuda com multiplicação regular. Suponha que x e y sejam dois inteiros de n bits e considere, por conveniência, que n seja uma potência de 2 (o caso mais geral não tem quase nenhuma diferença). Como um primeiro passo para multiplicar x e y , separe cada um deles em suas metades



Carl Friedrich Gauss
1777-1855

© Corbis

esquerda e direita, que têm $n/2$ bits de comprimento:

$$x = \boxed{x_L} \quad \boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L} \quad \boxed{y_R} = 2^{n/2}y_L + y_R.$$

Por exemplo, se $x = 10110110_2$ (o subscrito 2 significa “binário”), então $x_L = 1011_2$, $x_R = 0110_2$ e $x = 1011_2 \times 2^4 + 0110_2$. O produto de x e y pode, então, ser reescrito como

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

Vamos computar xy por meio da expressão da direita. As adições tomam tempo linear, assim como as multiplicações por potências de 2 (que são meramente deslocamentos para a esquerda). As operações significativas são as quatro multiplicações de $n/2$ bits, $x_L y_L$, $x_L y_R$, $x_R y_L$, $x_R y_R$; dessas cuidamos com quatro chamadas recursivas. Assim, nosso método para multiplicar números de n bits começa fazendo quatro chamadas recursivas para multiplicar esses quatro pares de números de $n/2$ bits (quatro subproblemas de metade do tamanho) e, então, calcula a expressão anterior em tempo $O(n)$. Escrevendo $T(n)$ para o tempo de execução total para entradas de n bits, obtemos a *relação de recorrência*

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

Veremos em breve estratégias gerais para resolver tais equações. Por enquanto, esta particular relação resulta em $O(n^2)$, o mesmo tempo de execução da técnica tradicional da escola para multiplicação. Portanto temos um algoritmo radicalmente novo, mas não fizemos qualquer progresso na eficiência. Como nosso método pode ser acelerado?

Aqui é quando o truque de Gauss vem à mente. Embora a expressão para xy pareça demandar quatro multiplicações de $n/2$ bits, como antes, três apenas resolvem: $x_L y_L$, $x_R y_R$ e $(x_L + x_R)(y_L + y_R)$, pois $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$.

Figura 2.1 Um algoritmo de divisão-e-conquista para multiplicação.

função multiplica(x, y)

Entrada: Inteiros positivos x e y de n bits

Saída: O produto deles

se $n = 1$: retorna xy

$x_L, x_R = [n/2]$ bits mais à esquerda e $[n/2]$ bits mais à direita, de x
 $y_L, y_R = [n/2]$ bits mais à esquerda e $[n/2]$ bits mais à direita, de y

$P_1 = \text{multiplica}(x_L, y_L)$

$P_2 = \text{multiplica}(x_R, y_R)$

$P_3 = \text{multiplica}(x_L + x_R, y_L + y_R)$

retorna $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$

O algoritmo resultante, apresentado na Figura 2.1, tem um tempo de execução aperfeiçoado de¹

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

A questão é que agora o aperfeiçoamento por um fator constante, de 4 para 3, ocorre no próprio nível da recursão, e o efeito acumulado leva a um tempo de execução sensivelmente menor $O(n^{1.59})$.

Esse tempo de execução pode ser derivado examinando-se o padrão de chamadas recursivas do algoritmo, que forma uma estrutura de árvore, como na Figura 2.2. Vamos tentar entender a forma dessa árvore. Em cada nível sucessivo de recursão os subproblemas têm seu tamanho diminuído pela metade. No nível $(\log_2 n)^a$, o tamanho dos subproblemas cai para 1 e, portanto, a recursão termina. Assim, a altura da árvore é $\log_2 n$. O fator de ramificação é 3 — cada problema produz recursivamente três problemas menores — com o resultado de que no nível k da árvore existem 3^k subproblemas, cada um com tamanho $n/2^k$.

Para cada subproblema, uma quantidade linear de trabalho é feita para identificar subproblemas seguintes e combinar suas respostas. Portanto, o tempo total gasto no nível k na árvore é

$$3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n).$$

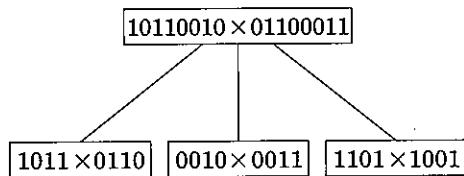
¹Na verdade, a recorrência deveria ser

$$T(n) \leq 3T(n/2 + 1) + O(n)$$

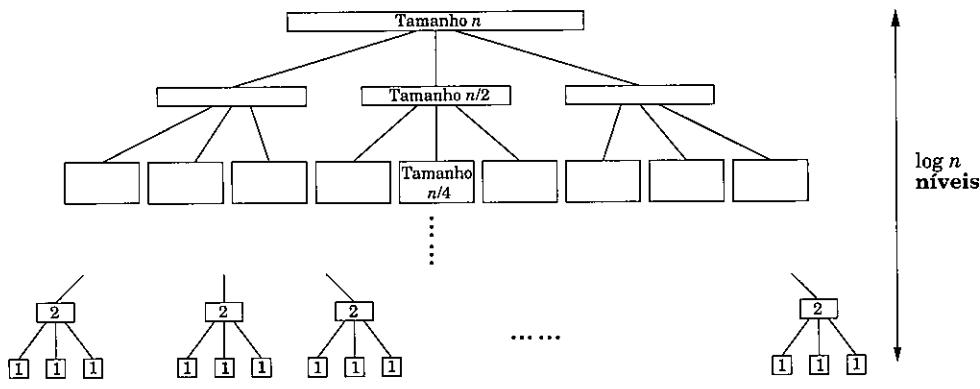
pois os números $(x_L + x_R)$ e $(y_L + y_R)$ poderiam ter $n/2 + 1$ bits. A relação que estamos usando é mais simples de manusear e pode-se ver que ela implica no mesmo tempo de execução em notação O .

Figura 2.2 Multiplicação inteira por divisão-e-conquista. (a) Cada problema é dividido em três subproblemas. (b) Os níveis de recursão.

(a)



(b)



No nível mais alto, quando $k = 0$, isto resulta em $O(n)$. No nível mais baixo, quando $k = \log_2 n$, resulta em $O(3^{\log_2 n})$, que pode ser reescrito como $O(n^{\log_3 3})$ (você entende por quê?). Entre esses dois extremos, o trabalho feito cresce geometricamente de $O(n)$ até $O(n^{\log_3 3})$, por um fator de $3/2$ por nível. A somatória de qualquer série geométrica crescente é, dentro de um fator constante, simplesmente o último termo da série: tal é a rapidez do crescimento (Exercício 0.2). Portanto o tempo de execução total é $O(n^{\log_3 3})$, que é cerca de $O(n^{1.59})$.

Na ausência do truque de Gauss, a árvore de recursão teria o mesmo tamanho, mas o fator de ramificação seria 4. Havéria $4^{\log_2 n} = n^2$ folhas, e, portanto, o tempo de execução seria pelo menos de tal ordem. Em algoritmos de divisão-e-conquista, o número de subproblemas se traduz no fator de ramificação da árvore de recursão; pequenas mudanças neste coeficiente podem causar um grande impacto no tempo de execução.

Uma nota prática: em geral não faz sentido realizar chamadas recursivas até 1 bit. Para a maioria dos processadores, multiplicação de 16 ou 32 bits é uma operação simples; portanto, no momento em que os números diminuem essa magnitude, eles devem ser operados diretamente pelo processador.

Por fim, a eterna questão: *será que podemos fazer melhor?* Acontece de fato que existem algoritmos ainda mais rápidos para multiplicação de números, baseados em um outro importante algoritmo de divisão-e-conquista: a transformada rápida de Fourier, a ser explicada na Seção 2.6.

2.2 Relações de recorrência

Algoritmos de divisão-e-conquista freqüentemente seguem um padrão genérico: eles resolvem um problema de tamanho n solucionando recursivamente, digamos, a subproblemas de tamanho n/b e, então, combinando estas respostas em tempo $O(n^d)$, para certos $a, b, d > 0$ (no algoritmo de multiplicação, $a = 3, b = 2$ e $d = 1$). Seus tempos de execução podem portanto ser capturados pela equação $T(n) = aT([n/b]) + O(n^d)$. Agora derivamos uma solução fechada para essa recorrência geral de maneira que não tenhamos mais de solucioná-la explicitamente a cada nova instância.

Teorema mestre² Se $T(n) = aT([n/b]) + O(n^d)$ para constantes $a > 0, b > 1$ e $d \geq 0$, então

$$T(n) = \begin{cases} O(n^d) & \text{se } d > \log_b a \\ O(n^d \log n) & \text{se } d = \log_b a \\ O(n^{\log_b a}) & \text{se } d < \log_b a. \end{cases}$$

Esse único teorema nos dá o tempo de execução da maioria dos procedimentos de divisão-e-conquista que provavelmente usaremos.

Prova. Para provar a afirmação, suponhamos, por conveniência, que n seja uma potência de b . Isso não influenciará a cota final de nenhuma maneira importante — além do mais, n está no máximo a um fator multiplicativo de b de alguma potência de b (Exercício 2.2) — e permitirá ignorarmos o efeito de arredondamento em $[n/b]$.

A seguir, note que o tamanho dos subproblemas decresce por um fator de b com cada nível de recursão e, portanto, alcança o caso base depois de $\log_b n$ níveis. Essa é a altura da árvore de recursão. Seu fator de ramificação é a , assim o nível k -ésimo da árvore é composto de a^k subproblemas, cada um de tamanho n/b^k (Figura 2.3). O trabalho total feito neste nível é

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k.$$

À medida que k vai de 0 (a raiz) até $\log_b n$ (as folhas), esses números formam uma série geométrica com razão a/b^d . Encontrar a somatória de uma tal série em termos de notação O é fácil (Exercício 0.2) e resulta em três casos.

1. A razão é menor do que 1.
Então a série é decrescente e sua somatória é dada apenas pelo seu primeiro termo, $O(n^d)$.
2. A razão é maior do que 1.
A série é crescente e sua somatória é dada por seu último termo, $O(n^{\log_b a})$:

$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$

3. A razão é exatamente 1.
Nesse caso todos os $O(\log n)$ termos da série são iguais a $O(n^d)$.

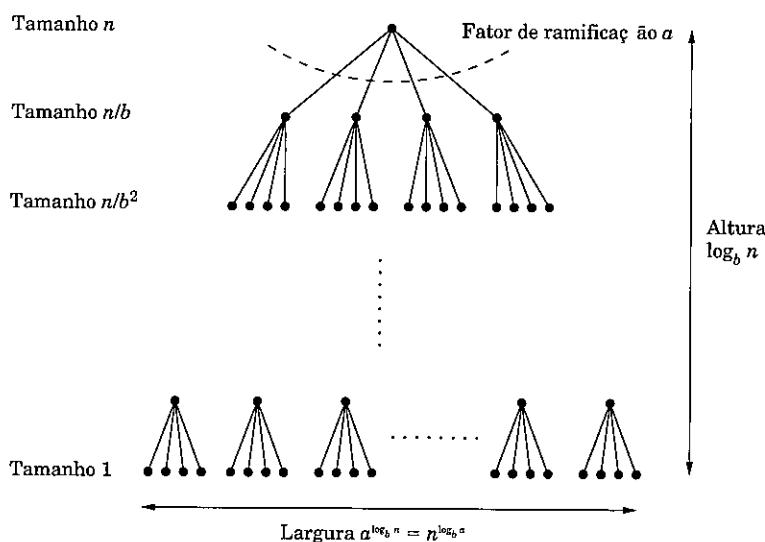
Esses casos se traduzem diretamente nas três condições da sentença do teorema. ■

²Existem resultados desse tipo ainda mais gerais, mas não precisaremos deles.

Busca binária

O mais famoso dos algoritmos de divisão-e-conquista é, claro, *busca binária*: para encontrarmos a chave k em um arquivo grande contendo chaves $z[0, 1, \dots, n - 1]$ ordenadas, primeiro comparamos k com $z[n/2]$ e, dependendo do resultado, fazemos recursão ou na primeira metade do arquivo, $z[0, \dots, n/2 - 1]$, ou na segunda metade $z[n/2, \dots, n - 1]$. A recorrência agora é $T(n) = T(\lceil n/2 \rceil) + O(1)$, que é o caso $a = 1, b = 2, d = 0$. Ligando isso ao nosso teorema mestre, obtemos a solução familiar: um tempo de execução de apenas $O(\log n)$.

Figura 2.3 Cada problema de tamanho n é dividido em a subproblemas de tamanho n/b .



2.3 Mergesort

O problema de ordenar uma lista de números sugere imediatamente uma estratégia de divisão-e-conquista: dividir a lista em duas metades, ordenar recursivamente cada metade e, então, fundir as duas sublistas ordenadas (fazer o *merge* das listas).

função mergesort($a[1 \dots n]$)

Entrada: Um vetor de números $a[1 \dots n]$

Saída: Uma versão ordenada desse vetor

se $n > 1$:

retorna merge(mergesort($a[1 \dots [n/2]]$),
 mergesort($a[[n/2] + 1 \dots n]$))

senão:

retorna a

A correção desse algoritmo é auto-evidente, desde que uma sub-rotina `merge` correta seja especificada. Dados dois vetores ordenados $x[1 \dots k]$ e $y[1 \dots l]$, como podemos eficientemente fazer o merge deles em um único vetor ordenado $z[1 \dots k+l]$? Bem, o primeiro elemento de z é ou $x[1]$ ou $y[1]$, aquele que for menor. O restante de $z[\cdot]$ pode, então, ser construída recursivamente.

```
função merge( $x[1 \dots k]$ ,  $y[1 \dots l]$ )
  se  $k = 0$ : retorna  $y[1 \dots l]$ 
  se  $l = 0$ : retorna  $x[1 \dots k]$ 
  se  $x[1] \leq y[1]$ :
    retorna  $x[1] \circ \text{merge}(x[2 \dots k], y[1 \dots l])$ 
  senão:
    retorna  $y[1] \circ \text{merge}(x[1 \dots k], y[2 \dots l])$ 
```

Aqui \circ denota concatenação. Este procedimento `merge` faz uma quantidade constante de trabalho por chamada recursiva (desde que o espaço requerido pelo vetor esteja previamente alocado), totalizando um tempo de execução de $O(k + l)$. Assim, `merge` é linear, e o tempo total tomado por `mergesort` é

$$T(n) = 2T(n/2) + O(n),$$

ou $O(n \log n)$.

Examinando novamente o algoritmo `mergesort`, vemos que todo o trabalho real é feito na fusão dos vetores, o que não começa até que a recursão desça até vetores de um elemento. Os vetores são fundidos aos pares, gerando vetores de dois elementos. Então, pares desses vetores de dois elementos são fundidos, produzindo vetores de quatro elementos e assim por diante. A Figura 2.4 mostra um exemplo.

Esse ponto de vista também sugere como `mergesort` pode tornar-se iterativo. Em qualquer dado instante, existe um conjunto de vetores “ativos” — inicialmente, os vetores de um elemento — que são fundidos aos pares para dar origem ao próximo lote de vetores ativos. Esses vetores podem ser organizados em uma fila e processados removendo-se sucessivamente dois vetores da fila, fundindo-os, e colocando o resultado no fim da fila.

No pseudocódigo seguinte, a operação primitiva `inserir` adiciona um elemento no final da fila, enquanto `remover` retira e retorna o elemento no início da fila.

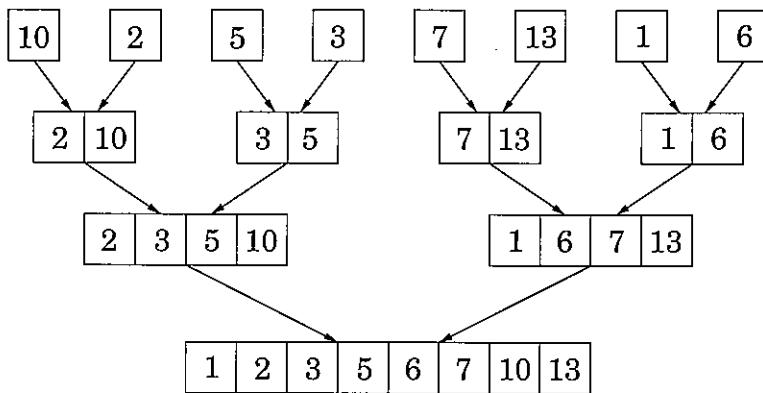
```
função mergesort-iterativo( $a[1 \dots n]$ )
  Entrada: elementos  $a_1, a_2, \dots, a_n$  a serem ordenados

   $Q = []$  (fila vazia)
  para  $i = 1$  até  $n$ :
    inserir( $Q, [a_i]$ )
  enquanto  $|Q| > 1$ :
    inserir( $Q, \text{merge}(\text{remover}(Q), \text{remover}(Q))$ )
  retorna  $\text{remover}(Q)$ 
```

Figura 2.4 A seqüência de operações de *merge* no mergesort.

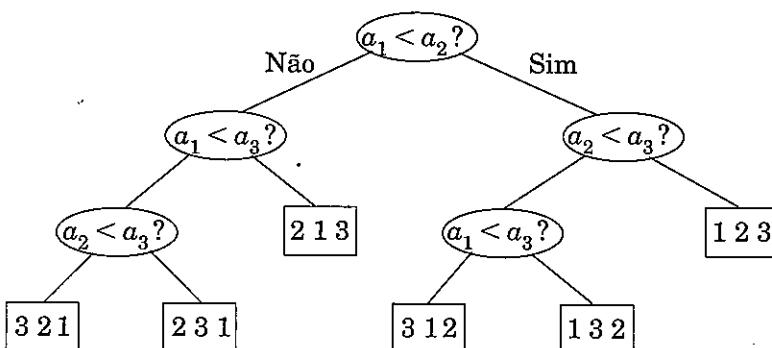
Entrada:

10	2	5	3	7	13	1	6
----	---	---	---	---	----	---	---



Uma cota inferior $n \log n$ para ordenação

Algoritmos de ordenação podem ser representados como árvores. A árvore da figura a seguir ordena um vetor de três elementos a_1, a_2, a_3 . Ela começa por comparar a_1 com a_2 e, se o primeiro for maior, compara-o com a_3 ; caso contrário, compara a_2 com a_3 e assim por diante. Em algum momento terminamos em uma folha e esta folha está rotulada com a ordem verdadeira dos três elementos como uma permutação de 1, 2, 3. Por exemplo, se $a_2 < a_1 < a_3$, obtemos a folha rotulada “2 1 3”.



A *altura* da árvore — o número de comparações no caminho mais longo da raiz até uma folha, neste caso 3 — é exatamente a complexidade de tempo de pior caso do algoritmo.

Uma cota inferior $n \log n$ para ordenação (continuação)

Essa abordagem para algoritmos de ordenação é útil porque nos permite argumentar que *mergesort* é ótimo, no sentido que $\Omega(n \log n)$ comparações são necessárias para ordenar n elementos.

Aqui está o argumento: considere qualquer árvore como essa que ordene n elementos. Cada uma de suas folhas é rotulada com uma permutação de $\{1, 2, \dots, n\}$. De fato, *toda* permutação tem de aparecer como o rótulo de alguma folha. A razão é simples: se uma particular permutação está ausente, o que acontece se rodarmos o algoritmo sobre uma entrada ordenada de acordo com esta permutação? E como existem $n!$ permutações de n elementos, conclui-se que a árvore tem de ter pelo menos $n!$ folhas.

Estamos quase no fim: essa árvore é binária e argumentamos que ela tem pelo menos $n!$ folhas. Lembre-se agora de que uma árvore binária de altura d tem no máximo 2^d folhas (prova: uma indução simples sobre d). Portanto, a altura da nossa árvore — que é a complexidade do nosso algoritmo — tem de ser pelo menos $\log(n!)$.

E é um fato bem conhecido que $\log(n!) \geq c \cdot n \log n$ para algum $c > 0$. Há várias maneiras de ver isso. A mais fácil é notar que $n! \geq (n/2)^{(n/2)}$ porque $n! = 1 \cdot 2 \cdots \cdot n$ contém pelo menos $n/2$ fatores maiores do que $n/2$; e, então, tomar o logaritmo de ambos os lados. Outra maneira é recorrer à fórmula de Stirling

$$n! \approx \sqrt{\pi \left(2n + \frac{1}{3} \right)} \cdot n^n \cdot e^{-n}.$$

De qualquer maneira, estabelecemos que qualquer árvore de comparação que ordene n elementos tem de fazer, no pior caso, $\Omega(n \log n)$ comparações e, assim, *mergesort* é ótimo!

Bem, há uma ressalva: esse argumento se aplica somente a *algoritmos que usam comparações*. É concebível que existam estratégias alternativas para ordenação, talvez por meio de manipulações numéricas sofisticadas, que trabalhem em tempo linear? A resposta é *sim*, sob certas circunstâncias excepcionais: o exemplo canônico para isso é quando os elementos a serem ordenados são inteiros pertencentes a um pequeno intervalo (Exercício 2.20).

2.4 Medianas

A *mediana* de uma lista de números é o seu quinquagésimo percentil: metade dos números é maior do que ele e metade é menor. Por exemplo, a mediana de $[45, 1, 10, 30, 25]$ é 25, pois esse elemento fica no meio quando os números são postos em ordem. Se a lista tem tamanho par, existem duas opções para o elemento do meio e, nesse caso, escolhemos, digamos, o menor dos dois.

O propósito da mediana é resumir o conjunto de números por um valor único, típico. A *média* é também bastante usada para isso, mas a mediana é em certo sentido mais típica dos dados: ela é sempre um dos valores dos dados, ao contrário da média, e é menos sensível a valores fora da curva. Por exemplo, a mediana de uma lista de cem 1 é (justamente) 1, assim como a média. Entretanto, se apenas um desses números é corrompido accidentalmente e se torna 10.000, a média vai a mais de 100, enquanto a mediana não é afetada.

Computar a mediana de n números é fácil: simplesmente ordene os números. A desvantagem é que isso toma tempo $O(n \log n)$, enquanto idealmente gostaríamos de alguma coisa linear. Temos razão para ser otimistas, porque ordenar é fazer muito mais trabalho do que realmente precisamos — ou seja, precisamos somente do elemento do meio e não nos importamos com a ordem relativa dos demais.

Ao procurar por uma solução recursiva é muitas vezes, paradoxalmente, mais fácil trabalhar com uma versão *mais geral* do problema — pela razão simples de que isso nos dá um passo de recursão mais poderoso. No nosso caso, a generalização que consideraremos é a *seleção*.

Seleção

Entrada: Uma lista de números S ; um inteiro k

Saída: O k -ésimo menor elemento de S

Por exemplo, se $k = 1$, o menor elemento de S é buscado, enquanto, se $k = \lfloor |S|/2 \rfloor$, é a mediana.

Um algoritmo de divisão-e-conquista randomizado para seleção

Aqui está uma abordagem divisão-e-conquista para seleção. Para qualquer número v , imagine dividir a lista S em três categorias: elementos menores do que v , aqueles iguais a v (pode haver duplicados) e aqueles maiores do que v . Denote as categorias por S_L , S_v e S_R , respectivamente. Por exemplo, se o vetor

$$S : \boxed{2 \ 36 \ 5 \ 21 \ 8 \ 13 \ 11 \ 20 \ 5 \ 4 \ 1}$$

é dividido para $v = 5$, os três subvetores gerados são

$$S_L : \boxed{2 \ 4 \ 1}$$

$$S_v : \boxed{5 \ 5}$$

$$S_R : \boxed{36 \ 21 \ 8 \ 13 \ 11 \ 20}.$$

A busca pode ser instantaneamente reduzida a uma dessas sublistas. Se quisermos, digamos, o oitavo menor elemento de S , sabemos que ele tem de ser o *terceiro* menor elemento de S_R , pois $|S_L| + |S_v| = 5$. Ou seja, $\text{seleção}(S, 8) = \text{seleção}(S_R, 3)$. De maneira mais geral, checando k em relação aos tamanhos dos subvetores, podemos rapidamente determinar qual deles contém o elemento desejado:

$$\text{seleção}(S, k) = \begin{cases} \text{seleção}(S_L, k) & \text{se } k \leq |S_L| \\ v & \text{se } |S_L| < k \leq |S_L| + |S_v| \\ \text{seleção}(S_R, k - |S_L| - |S_v|) & \text{se } k > |S_L| + |S_v|. \end{cases}$$

As três sublistas S_L , S_v e S_R podem ser computadas de S em tempo linear; de fato, essa computação pode até ser feita *in place*, isto é, sem alocar mais memória (Exercício 2.15). Fazemos, então, recursão na sublista apropriada. O efeito da divisão, portanto, é diminuir o número de elementos de $|S|$ para no máximo $\{|S_L, S_R|\}$.

Nosso algoritmo de divisão-e-conquista para seleção está completamente especificado agora, exceto pelo detalhe crucial de como escolher v . Ele deve ser escolhido rapidamente e deve reduzir o vetor substancialmente, sendo a situação ideal $|S_L|, |S_R| \approx \frac{1}{2}|S|$. Se pudéssemos sempre garantir essa situação, obteríamos um tempo de execução de

$$T(n) = T(n/2) + O(n),$$

que é linear, como queríamos. Mas isso requer tomar a mediana como v , o que é nosso objetivo final! Em vez disso, seguimos uma alternativa muito mais simples: *tomamos v aleatoriamente de S*.

Análise de eficiência

Naturalmente, o tempo de execução do nosso algoritmo depende das escolhas aleatórias de v . É possível que devido a uma falta de sorte persistente, continuamos tomando o maior elemento do vetor como v (ou o menor elemento) e, com isso, reduzindo o vetor em apenas um elemento a cada vez. No exemplo anterior, poderíamos tomar $v = 36$, depois $v = 21$ e assim por diante. Esse cenário de pior caso forçaria nosso algoritmo de seleção a realizar

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

operações (quando usado para computar a mediana), mas isso é bastante improvável que ocorra. Igualmente improvável é o melhor caso possível que discutimos antes, no qual cada v escolhido aleatoriamente acaba dividindo o vetor em duas metades exatas, resultando em um tempo de execução de $O(n)$. Onde, neste espectro de $O(n)$ até $\Theta(n^2)$, está o tempo de execução médio? Felizmente, está bem perto do tempo de melhor caso.

Para distinguirmos entre escolhas felizes e infelizes de v , chamaremos v de *bom* se estiver entre o 25º e o 75º percentil do vetor de onde é tomado. Preferimos essas escolhas de v porque asseguram que as sublistas S_L e S_R tenham tamanho no máximo três quartos do de S (você percebe o porquê?), para que o vetor seja muito reduzido. Felizmente, bons v são abundantes: metade dos elementos de qualquer lista tem de estar entre o 25º e o 75º percentil!

Dado que um v escolhido aleatoriamente tem 50% de chance de ser bom, quantos v precisamos tomar em média até conseguir um bom? Aqui está uma reformulação mais familiar (veja também o Exercício 1.34):

Lema *Na média, uma moeda justa precisa ser lançada duas vezes antes de dar "cara".*

Prova. Seja E o número esperado de lançamentos antes de dar cara. Certamente precisamos de pelo menos um lançamento e, se der cara, estamos satisfeitos. Se der coroa (o que ocorre com probabilidade 1/2), precisamos repetir. Assim $E = 1 + \frac{1}{2}E$, que resulta em $E = 2$. ■

Portanto, depois de duas operações de divisão em média, o vetor será reduzido para no máximo três quartos do tamanho original. Denotando por $T(n)$ o tempo de execução esperado para um vetor de tamanho n , obtemos

$$T(n) \leq T(3n/4) + O(n).$$

Isto segue ao tomarmos o valor esperado de ambos os lados da seguinte sentença:

Tempo tomado para um vetor de tamanho n

\leq (tempo tomado para um vetor de tamanho $3n/4$) + (tempo para reduzir o tamanho do vetor para $\leq 3n/4$),

e, para o lado direito, ao usarmos a propriedade familiar de que *o valor esperado da soma é a soma dos valores esperados*.

O comando sort do Unix

Comparando os algoritmos para ordenação e para a busca da mediana, notamos que, além da estrutura e filosofia comuns de divisão-e-conquista, eles são opostos exatos. Mergesort divide o vetor em dois da maneira mais conveniente (primeira metade, segunda metade), sem qualquer relação com as magnitudes dos elementos em cada metade; mas depois ele trabalha duro para juntar os subvetores ordenados. Em contraste, o algoritmo da mediana é cuidadoso com a divisão (números menores antes, depois os maiores), mas seu trabalho termina com a chamada recursiva.

Quicksort é um algoritmo para ordenação que divide o vetor exatamente da mesma maneira que o algoritmo da mediana; e, uma vez que os subvetores estejam ordenados, pelas duas chamadas recursivas, não há nada mais a fazer. Seu desempenho de pior caso é $\Theta(n^2)$, assim como o da busca da mediana. Mas pode-se provar (Exercício 2.24) que seu caso médio é $O(n \log n)$. Além disso, empiricamente ele supera em desempenho os demais algoritmos para ordenação. Isso tem feito do quicksort o favorito em muitas aplicações — por exemplo, ele é a base do código com o qual arquivos realmente enormes são ordenados.

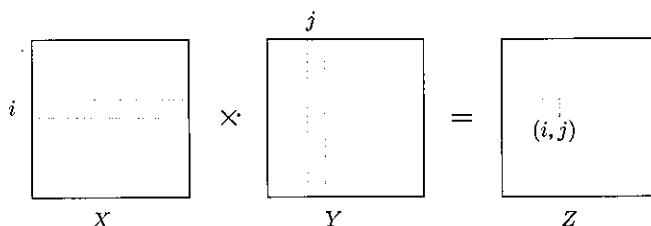
Dessa recorrência concluímos que $T(n) = O(n)$: para *qualquer* entrada, nosso algoritmo retorna a resposta correta depois de um número linear de passos, na média.

2.5 Multiplicação de matrizes

O produto de duas matrizes $n \times n$, X e Y , é uma terceira matriz $n \times n$, $Z = XY$, com a (i, j) -ésima célula igual a

$$Z_{ij} = \sum_{k=1}^n X_{ik} Y_{kj}.$$

Para tornar isso mais real, Z_{ij} é o produto escalar da i -ésima linha com a j -ésima coluna de Y :



Em geral, XY não é o mesmo de YX ; multiplicação de matrizes não é comutativa.

A fórmula anterior implica um algoritmo $O(n^3)$ para multiplicação de matrizes: há n^2 células para computar e cada uma toma tempo $O(n)$. Por muito tempo, acreditou-se que esse era o melhor tempo de execução possível e há, até mesmo, uma prova de que em certos modelos de computação nenhum algoritmo pode fazer melhor. Foi, portanto, uma razão de grande excitação o anúncio, em 1969, feito pelo matemático alemão Volker Strassen de um algoritmo significativamente mais eficiente, baseado em divisão-e-conquista.

É particularmente fácil quebrar multiplicação de matrizes em subproblemas, porque ela pode ser realizada em *blocos*. Para ver o que isso significa, particione X em quatro blocos $n/2 \times n/2$ e Y igualmente:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Seu produto pode ser expresso em termos desses blocos e é exatamente como se os blocos fossem simples elementos (Exercício 2.11).

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Agora temos uma estratégia de divisão-e-conquista: para computar o produto XY , de tamanho n , compute recursivamente oito produtos de tamanho $n/2$, AE , BG , AF , BH , CE , DG , CF , DH e faça umas poucas adições de tempo $O(n^2)$. O tempo de execução total é descrito pela relação de recorrência

$$T(n) = 8T(n/2) + O(n^2).$$

Isso resulta em um tempo nada impressionante de $O(n^3)$, o mesmo que para o algoritmo trivial. Mas a eficiência *pode* ser melhorada e, como com multiplicação inteira, a chave é uma álgebra inteligente. Acontece que XY pode ser computado com apenas sete subproblemas de tamanho $n/2 \times n/2$, pela via de uma decomposição tão engenhosa e intricada que nos perguntamos de que maneira Strassen conseguiu descobri-la!

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

onde

$$P_1 = A(F - H) \qquad P_5 = (A + D)(E + H)$$

$$P_2 = (A + B)H \qquad P_6 = (B - D)(G + H)$$

$$P_3 = (C + D)E \qquad P_7 = (A - C)(E + F)$$

$$P_4 = D(G - E)$$

O novo tempo de execução é

$$T(n) = 7T(n/2) + O(n^2),$$

que, pelo teorema mestre, resulta em $O(n^{\log_2 7}) \approx O(n^{2.81})$.

2.6 A transformada rápida de Fourier

Até agora vimos como divisão-e-conquista leva a algoritmos rápidos para multiplicar inteiros e matrizes; nosso próximo alvo são *polinômios*. O produto de dois polinômios de grau d é um polinômio de grau $2d$, por exemplo:

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4.$$

Mas em geral, se $A(x) = a_0 + a_1x + \dots + a_dx^d$ e $B(x) = b_0 + b_1x + \dots + b_dx^d$, o produto deles $C(x) = A(x) \cdot B(x) = c_0 + c_1x + \dots + c_{2d}x^{2d}$ possui coeficientes

$$c_k = a_0b_k + a_1b_{k-1} + \dots + a_kb_0 = \sum_{i=0}^k a_i b_{k-i}$$

(para $i > d$, tome a_i e b_i iguais a zero). Computar c_k por essa fórmula toma $O(k)$ passos, e encontrar todos os $2d + 1$ coeficientes iria requerer tempo $\Theta(d^2)$. Será que podemos multiplicar polinômios mais rapidamente do que isso?

A solução que iremos desenvolver, a transformada rápida de Fourier, revolucionou — até mesmo, definiu — a área de processamento de sinais (veja o próximo quadro). Em razão da sua enorme importância, e da sua grande quantidade de idéias profundas de diferentes áreas de estudo, vamos abordá-la com mais calma do que normalmente. O leitor que deseje apenas o núcleo do algoritmo pode ir diretamente para a Seção 2.6.4.

2.6.1 Uma representação alternativa para polinômios

Para alcançarmos um algoritmo rápido para multiplicação de polinômios vamos tirar inspiração de uma importante idéia sobre polinômios.

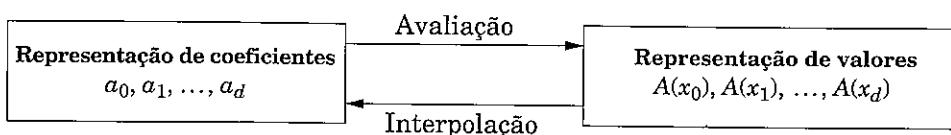
Fato Um polinômio de grau d é unicamente caracterizado por seus valores em $d + 1$ pontos distintos.

Uma instância familiar disso é que “quaisquer dois pontos determinam uma reta”. Veremos mais adiante por que a sentença mais geral é verdadeira (página 64), mas por ora esse fato nos dá uma *representação alternativa* para polinômios. Fixe quaisquer pontos distintos x_0, \dots, x_d . Podemos especificar um polinômio de grau d , $A(x) = a_0 + a_1x + \dots + a_dx^d$, de uma das seguintes maneiras:

1. Seus coeficientes a_0, a_1, \dots, a_d
2. Os valores $A(x_0), A(x_1), \dots, A(x_d)$

Dessas duas representações, a segunda é a mais atrativa para multiplicação polinomial. Como o produto $C(x)$ tem grau $2d$, ele é completamente determinado por seu valor em quaisquer $2d + 1$ pontos. E seu valor em qualquer dado ponto z é bem fácil de descobrir, simplesmente $A(z)$ vezes $B(z)$. Assim *multiplicação toma tempo linear na representação de valores*.

O problema é que esperamos que os polinômios de entrada, e também seu produto, sejam especificados por coeficientes. Portanto, precisamos primeiro traduzir de coeficientes para valores — o que é apenas uma questão de *avaliar* o polinômio nos pontos escolhidos — depois multiplicar na representação de valores, e finalmente traduzir de volta para coeficientes, um processo chamado *interpolação*.

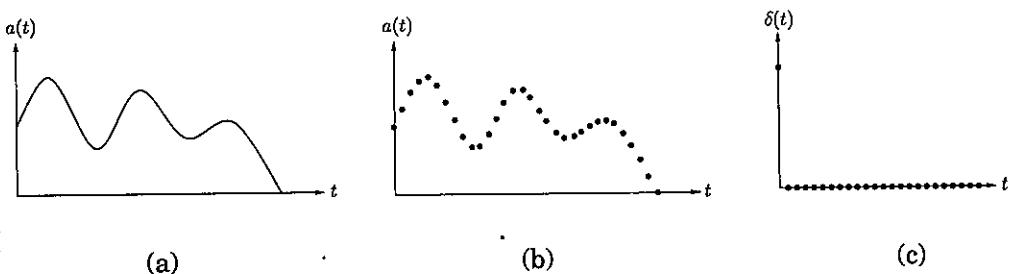


A Figura 2.5 apresenta o algoritmo resultante.

Por que multiplicar polinômios?

Uma das razões é que os algoritmos mais rápidos que temos para multiplicar inteiros se baseiam fortemente em multiplicação de polinômios; afinal, polinômios e inteiros binários são bastante similares — simplesmente substitua a variável x pela base 2 e cuide dos dígitos de excesso. Mas, talvez, principalmente porque multiplicação de polinômios é crucial para *processamento de sinais*.

Um *sinal* é qualquer quantidade que é uma função do tempo (como na Figura (a)) ou da posição. Ele pode, por exemplo, capturar uma voz humana medindo flutuações na pressão do ar próximo à boca de quem fala, ou alternativamente, o padrão de estrelas no céu à noite, medindo o brilho como uma função de ângulo.



Para podermos extrair informação de um sinal, precisamos primeiro *digitalizá-lo* por amostragem (Figura (b)) — e, então, colocá-lo por um *sistema* que irá transformá-lo de alguma maneira. A saída é chamada de *resposta* do sistema:

$$\text{sinal} \rightarrow \boxed{\text{SISTEMA}} \rightarrow \text{resposta.}$$

Uma classe importante de sistemas são aqueles que são *lineares* — a resposta à soma de dois sinais é simplesmente a soma de suas respostas individuais — e *invariantes no tempo* — deslocar o sinal de entrada um tempo t produz a mesma saída, também deslocada por t . Qualquer sistema com essas propriedades é caracterizado por sua resposta ao sinal de entrada mais simples possível: o *impulso unitário* $\delta(t)$, consistindo apenas em um impulso isolado em $t = 0$ (Figura (c)). Para tanto, primeiramente considere a variante $\delta(t - i)$, um impulso deslocado que ocorre no tempo i . Qualquer sinal $a(t)$ pode ser expresso como uma combinação linear desses sinais, fazendo $\delta(t - i)$ amostrar seu comportamento no tempo i ,

$$a(t) = \sum_{i=0}^{T-1} a(i) \delta(t - i)$$

(se o sinal consiste em T amostras). Por linearidade, a resposta do sistema à entrada $a(t)$ é determinada pelas respostas dos vários $\delta(t - i)$ que, por invariância no tempo, são por sua vez apenas cópias deslocadas da *resposta impulso* $b(t)$, a resposta a $\delta(t)$.

Em outras palavras, a saída do sistema no tempo k é

$$c(k) = \sum_{i=0}^k a(i) b(k - i),$$

exatamente a fórmula para multiplicação de polinômios!

Figura 2.5 Multiplicação de polinômios.

Entrada: Coeficientes de dois polinômios, $A(x)$ e $B(x)$, de grau d
Saída: Seu produto $C = A \cdot B$

Seleção

Selecionar alguns pontos x_0, x_1, \dots, x_{n-1} , onde $n \geq 2d + 1$

Avaliação

Computar $A(x_0), A(x_1), \dots, A(x_{n-1})$ e $B(x_0), B(x_1), \dots, B(x_{n-1})$

Multiplicação

Computar $C(x_k) = A(x_k)B(x_k)$ para todo $k = 0, \dots, n - 1$

Interpolação

Restaurar $C(x) = c_0 + c_1x + \dots + c_{2d}x^{2d}$

A equivalência entre as duas representações de polinômios torna claro que esta abordagem de alto nível está correta, mas qual é a eficiência dela? Certamente o passo de seleção e as n multiplicações não são nenhum problema, simplesmente tempo linear³. Mas (deixando interpolação de lado, sobre o que sabemos menos ainda) o que dizer da avaliação? Avaliar um polinômio de grau $d \leq n$ em um único ponto toma $O(n)$ passos (Exercício 2.29) e, portanto, para n pontos nossa cota de partida é $\Theta(n^2)$. Veremos agora que a transformada rápida de Fourier (TRF) faz isso em tempo $O(n \log n)$ apenas, para uma escolha inteligente de x_0, \dots, x_{n-1} para a qual as computações requeridas pelos pontos individualmente se sobrepõem umas às outras e podem ser compartilhadas.

2.6.2 Avaliação por divisão-e-conquista

Aqui está uma idéia sobre como selecionar os n pontos nos quais avaliar um polinômio $A(x)$ de grau $\leq n - 1$. Se os escolhermos como pares positivo-negativo, isto é,

$$\pm x_0, \pm x_1, \dots, \pm x_{n/2-1},$$

as computações requeridas para cada $A(x_i)$ e $A(-x_i)$ se sobrepõem muito, porque as potências pares de x_i coincidem com as de $-x_i$.

Para investigarmos isso, precisamos dividir $A(x)$ em potências pares e ímpares, por exemplo

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^3).$$

Note que os termos entre parênteses são polinômios em x^2 . Em geral,

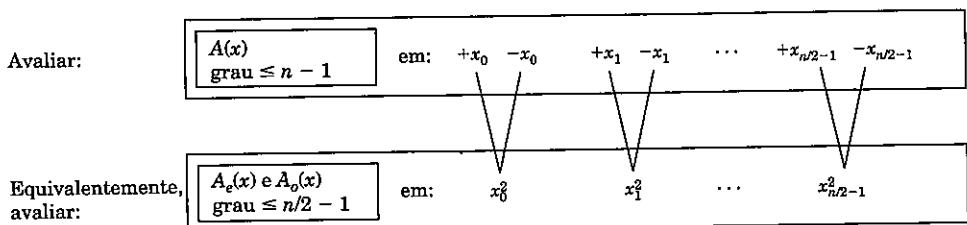
$$A(x) = A_e(x^2) + xA_o(x^2),$$

³Em um caso típico de multiplicação de polinômios, os coeficientes são números reais, além do que são pequenos o suficiente para que operações aritméticas básicas (adição e multiplicação) tomem tempo unitário. Suponhamos que esse seja o caso sem grandes perdas de generalidades; em particular, os limites de tempo que obtemos são facilmente ajustáveis a situações com números maiores.

onde $A_e(\cdot)$, com os coeficientes pares, e $A_o(\cdot)$, com os ímpares, são polinômios de grau $\leq n/2 - 1$ (considere por conveniência que n é par). Dados os *pares* de pontos $\pm x_i$, os cálculos necessários para $A(x_i)$ podem ser reciclados para calcular $A(-x_i)$:

$$\begin{aligned} A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2). \end{aligned}$$

Em outras palavras, avaliar $A(x)$ em n pontos casados $\pm x_0, \dots, \pm x_{n/2-1}$ se reduz a avaliar $A_e(x)$ e $A_o(x)$ (cada um deles tem metade do grau de $A(x)$) em apenas $n/2$ pontos, $x_0^2, \dots, x_{n/2-1}^2$.



O problema original de tamanho n desta forma é reformulado como dois subproblemas de tamanho $n/2$, seguido de um pouco de aritmética em tempo linear. Se pudéssemos fazer recursão, obteríamos um procedimento de divisão-e-conquista com tempo de execução

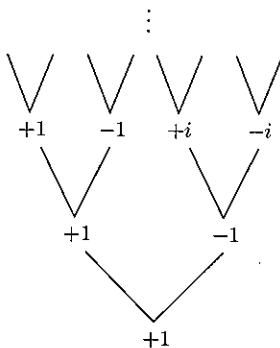
$$T(n) = 2T(n/2) + O(n),$$

o que é $O(n \log n)$, exatamente o que queremos.

Mas temos um problema: o truque positivo-negativo funciona somente no primeiro nível de recursão. Para fazermos a recursão no próximo nível, precisamos que os $n/2$ pontos de avaliação $x_0^2, x_1^2, \dots, x_{n/2-1}^2$ sejam eles próprios pares positivo-negativo. Mas como pode um número elevado ao quadrado ser negativo? A tarefa parece impossível! *A menos que, claro, usemos números complexos.*

Ótimo, mas quais números complexos? Para tanto, vamos fazer “engenharia reversa” no processo. Na base da recursão, temos um único ponto. Ele pode muito bem ser 1, caso em que o nível acima tem de consistir nas suas raízes quadradas, $\pm\sqrt{1} = \pm 1$.

O próximo nível acima tem $\pm\sqrt{+1} = \pm 1$, bem como os números *complexos* $\pm\sqrt{-1} = \pm i$, onde i é a unidade imaginária. Continuando dessa maneira, em algum momento alcançamos o conjunto inicial de n pontos. Talvez você já tenha adivinhado



o que eles são: *as raízes n-ésimas complexas da unidade*, ou seja, as n soluções complexas da equação $z^n = 1$.

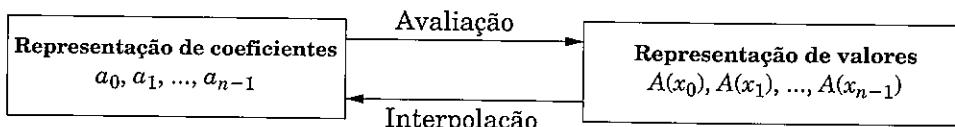
A Figura 2.6 é uma revisão ilustrada de alguns fatos básicos sobre números complexos. O terceiro painel introduz as raízes n -ésimas da unidade: os números complexos $1, \omega, \omega^2, \dots, \omega^{n-1}$, onde $\omega = e^{2\pi i/n}$. Se n é par,

1. As raízes n -ésimas são pares positivo-negativo, $\omega^{n/2+j} = -\omega^j$.
2. Elevá-las ao quadrado produz as raízes $(n/2)$ -ésimas da unidade.

Portanto, se começarmos com esses números para algum n que é uma potência de 2, nos sucessivos níveis de recursão teremos as raízes $(n/2^k)$ -ésimas da unidade, para $k = 0, 1, 2, 3, \dots$. Todos estes conjuntos de pontos são pares positivo-negativo e, portanto, nossa divisão-e-conquista, como apresentada no último painel, funciona perfeitamente. O algoritmo resultante é a transformada rápida de Fourier (Figura 2.7).

2.6.3 Interpolação

Vamos parar um pouco e analisar onde estamos. Primeiro desenvolvemos um esquema em alto nível para multiplicar polinômios (Figura 2.5), baseado na observação de que polinômios podem ser representados de duas maneiras, em termos de seus *coeficientes* ou em termos dos seus *valores* em um conjunto selecionado de pontos.



A representação de valores torna trivial a multiplicação de polinômios, mas não podemos ignorar a representação de coeficientes, pois ela é a forma na qual a entrada e a saída do nosso algoritmo como um todo são especificadas.

Então projetamos a TRF, uma maneira de mover de coeficientes para valores em tempo $O(n \log n)$ apenas, quando os pontos $\{x_i\}$ são raízes n -ésimas complexas da unidade $(1, \omega, \omega^2, \dots, \omega^{n-1})$.

$$\langle \text{valores} \rangle = \text{TRF}(\langle \text{coeficientes} \rangle, \omega).$$

Figura 2.6 As raízes complexas da unidade são ideais para nosso esquema de divisão-e-conquista.

	<p>Multipliação é fácil em coordenadas polares</p> <p>Multiplique as distâncias e some os ângulos: $(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.</p> <p>Para todo $z = (r, \theta)$,</p> <ul style="list-style-type: none"> $-z = (r, \theta + \pi)$, pois $-1 = (1, \pi)$. Se z está no <i>círculo unitário</i> (i.e., $r = 1$), então $z^n = (1, n\theta)$.
	<p>A raiz n-ésima complexa da unidade</p> <p>Soluções para a equação $z^n = 1$.</p> <p>Pela regra da multiplicação: as soluções são $z = (1, \theta)$, para θ um múltiplo de $2\pi/n$ (mostrado aqui para $n = 16$).</p> <p>Para n par:</p> <ul style="list-style-type: none"> Estes números estão em <i>pares positivo-negativo</i>: $-(1, \theta) = (1, \theta + \pi)$. Seus quadrados são as raízes $(n/2)$-ésimas da unidade, mostradas aqui com quadradinhos ao redor delas.
	<p>O passo de divisão-e-conquista</p> <p>Avaliar $A(x)$ nas raízes n-ésimas da unidade</p> <p>$(n$ é uma potência de 2)</p> <p>Divisão-e-conquista</p> <p>Avaliar $A_r(x), A_o(x)$ nas raízes $(n/2)$-ésimas</p> <p>Ainda emparelhados</p>

Figura 2.7 A transformada rápida de Fourier (formulação com polinômios).**função TRF(A, ω)**

Entrada: Representação de coeficientes de um polinômio $A(x)$ de grau $\leq n - 1$, onde n é uma potência de 2ω , uma raiz n -ésima da unidade

Saída: Representação de valores $A(\omega^0), \dots, A(\omega^{n-1})$

se $\omega = 1$: retorna $A(1)$

expressar $A(x)$ na forma $A_e(x^2) + xA_o(x^2)$

chame $\text{TRF}(A_e, \omega^2)$ para avaliar A_e nas potências pares de ω

chame $\text{TRF}(A_o, \omega^2)$ para avaliar A_o nas potências pares de ω

para $j = 0$ até $n - 1$:

compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$

retorna $A(\omega^0), \dots, A(\omega^{n-1})$

A última peça restante do quebra-cabeça é a operação inversa, interpolação. De forma impressionante, veremos que

$$\langle \text{coeficientes} \rangle = \frac{1}{n} \text{TRF}(\langle \text{valores} \rangle, \omega^{-1}).$$

Interpolação será, então, solucionada da maneira mais simples e elegante que poderíamos ter esperado — usando o mesmo algoritmo TRF, mas chamado com ω^{-1} no lugar de ω ! Isso pode parecer uma coincidência miraculosa, mas fará muito mais sentido quando reformularmos nossas operações de polinômios na linguagem da álgebra linear. Mas por ora, nosso algoritmo $O(n \log n)$ para multiplicação de polinômios (Figura 2.5) já está totalmente especificado.

Uma reformulação com matrizes

Para uma visão mais clara de interpolação, vamos explicitamente estabelecer a relação entre nossas duas representações para um polinômio $A(x)$ de grau $\leq n - 1$. Eles são ambos vetores de n números e um é uma transformação linear do outro:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

Chame a matriz do meio de M . Seu formato especializado — uma matriz de *Vandermonde* — dá a ela muitas propriedades notáveis, das quais a seguinte é particularmente relevante para nós.

Se x_0, \dots, x_{n-1} são números distintos, então M é inversível.

A existência de M^{-1} nos permite inverter a equação matricial anterior para expressar coeficientes em termos de valores. Em resumo,

Avaliação é multiplicação por M, enquanto interpolação é multiplicação por M^{-1} .

Esta reformulação de nossas operações de polinômios revela sua natureza essencial mais claramente. Entre outras coisas, ela finalmente justifica uma hipótese de que $A(x)$ é caracterizada unicamente por seus valores em quaisquer n pontos — de fato, agora temos uma fórmula explícita que nos dará os coeficientes de $A(x)$ nessa situação. Matrizes Vandermonde também têm a distinção de poder ser invertidas mais rapidamente do que matrizes mais gerais, em tempo $O(n^2)$ em vez de $O(n^3)$. Entretanto, usar isso para interpolação ainda não seria suficiente, assim nos voltamos para a escolha especial de pontos — as raízes complexas da unidade.

Interpolação resolvida

Em termos de álgebra linear, a TRF multiplica um vetor arbitrário n -dimensional — que temos chamado de *representação de coeficientes* — por uma matriz $n \times n$

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ & \vdots & & & \vdots \\ 1 & \omega^j & \omega^{2j} & \ddots & \omega^{(n-1)j} \\ & \vdots & & & \vdots \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \quad \begin{array}{l} \leftarrow \text{linha para } \omega^0 = 1 \\ \leftarrow \omega \\ \leftarrow \omega^2 \\ \vdots \\ \leftarrow \omega^j \\ \vdots \\ \leftarrow \omega^{n-1} \end{array}$$

onde ω é a raiz n -ésima da unidade e n é, uma potência de 2. Note quão facilmente podemos descrever esta matriz: sua célula (j, k) -ésima (começando a contagem das linhas e colunas de zero) é ω^{jk} .

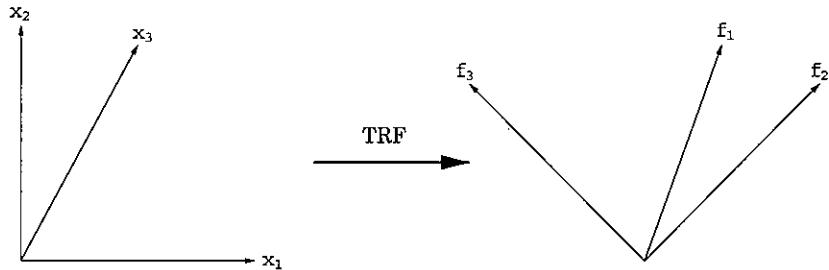
Multiplicação por $M = M_n(\omega)$ mapeia o k -ésimo eixo coordenado (o vetor só de zeros com exceção de um 1 na posição k) na k -ésima coluna de M . Agora aqui está a observação crucial, que iremos provar em breve: *as colunas de M são ortogonais (em ângulos retos) umas às outras*. Portanto podem ser imaginadas como eixos de um sistema de coordenadas alternativo, freqüentemente chamado de *base de Fourier*. O efeito de multiplicar um vetor por M é rotacioná-lo da base padrão, com o conjunto usual de eixos, para a base de Fourier, definida pelas colunas de M (Figura 2.8). A TRF é assim uma troca de base, uma *rotação rígida*. A inversa de M é a rotação oposta, de base de Fourier de volta para a base padrão. Quando escrevermos a condição de ortogonalidade precisamente, vamos poder identificar a transformação inversa com facilidade:

Fórmula de inversão $M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

Mas ω^{-1} também é uma raiz n -ésima da unidade e, portanto, interpolação — ou equivalente, multiplicação por $M_n(\omega)^{-1}$ — é ela própria simplesmente uma operação de TRF, mas com ω substituído por ω^{-1} .

Agora vejamos os detalhes. Tome ω como $e^{2\pi i/n}$ por conveniência e pense nas colunas de M como vetores em \mathbb{C}^n . Lembre-se de que o ângulo entre dois vetores $u = (u_0, \dots, u_{n-1})$ e $v = (v_0, \dots, v_{n-1})$ em \mathbb{C}^n é apenas um fator de escala vezes seu *produto interno*

Figura 2.8 A TRF toma pontos do sistema de coordenadas padrão, cujos eixos são mostrados como x_1, x_2, x_3 e os rotaciona para a base de Fourier, cujos eixos são as colunas de $M_n(\omega)$, mostradas como f_1, f_2, f_3 . Por exemplo, pontos na direção x_1 são mapeados para a direção f_1 .



produto inteiro

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \cdots + u_{n-1} v_{n-1}^*,$$

onde z^* denota o conjugado complexo⁴ de z . Essa quantidade é maximizada quando os vetores estão na mesma direção e é zero quando os vetores são ortogonais entre si.

A observação fundamental que precisamos é a seguinte.

Lema As colunas da matriz M são ortogonais entre si.

Prova. Tome o produto inteiro de quaisquer colunas j e k da matriz M ,

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \cdots + \omega^{(n-1)(j-k)}.$$

Isso é uma série geométrica com o primeiro termo 1, último termo $\omega^{(n-1)(j-k)}$ e razão ω^{j-k} . Portanto e resulta em $(1 - \omega^{n(j-k)})/(1 - \omega^{j-k})$, que é 0 — exceto quando $j = k$, caso em que todos os termos são 1 e a soma é n . ■

A propriedade de ortogonalidade pode ser resumida na equação única

$$MM^* = nI,$$

pois $(MM^*)_{ij}$ é o produto interno da i -ésima e da j -ésima colunas de M (você entende por quê?). Tal fato implica imediatamente $M^{-1} = (1/n)M^*$: temos uma fórmula de inversão! Mas será que é a mesma fórmula que propusemos antes? Vejamos — a (j,k) -ésima célula de M^* é o conjugado complexo da correspondente célula de M , em outras palavras ω^{-jk} . Após o que $M^* = M_n(\omega^{-1})$, e terminamos.

E agora podemos finalmente retroceder e visualizar a questão toda geometricamente. A tarefa que precisamos realizar, multiplicação de polinômios, é muito mais fácil na

⁴O conjugado complexo de um número complexo $z = re^{i\theta}$ é $z^* = re^{-i\theta}$. O conjugado complexo de um vetor (ou matriz) é obtido tomando os conjugados complexos de todas as células.

base de Fourier do que na base padrão. Portanto, primeiro rotacionamos os vetores para a base de Fourier (*avaliação*), depois realizamos a tarefa (*multiplicação*) e, finalmente, rotacionamos de volta (*interpolação*). Os vetores iniciais são *representações de coeficientes*, enquanto suas contrapartes rotacionadas são *representações de valores*. Comutar eficientemente entre essas representações, indo e voltando, é o domínio da TRF.

2.6.4 Observando de perto a transformada rápida de Fourier

Agora que nosso esquema eficiente para multiplicação de polinômios está totalmente realizado, vamos, então, refinar nosso conhecimento sobre a sub-rotina central que torna tudo possível, a transformada rápida de Fourier.

O algoritmo definitivo para TRF

A TRF toma como entrada um vetor $a = (a_0, \dots, a_{n-1})$ e um número complexo ω cujas potências $1, \omega, \omega^2, \dots, \omega^{n-1}$ são as raízes n -ésimas complexas da unidade. Ela multiplica o vetor a pela matriz $n \times n$, $M_n(\omega)$, que possui ω^{jk} como sua (j,k) -ésima célula (começando a contagem das linhas e colunas de zero). O potencial para usar divisão-e-conquista nesta multiplicação matriz-vetor torna-se visível quando as colunas de M são segregadas em pares e ímpares:

$$\begin{array}{ccc} \begin{matrix} k \\ | \\ \omega^{jk} \\ | \\ M_n(\omega) \end{matrix} & = & \begin{matrix} \text{Coluna } 2k & \text{Coluna } 2k+1 \\ | & | \\ \omega^{2jk} & \omega^j \cdot \omega^{2jk} \\ | & | \\ \text{Colunas pares} & \text{Colunas ímpares} \\ | & | \\ a_0 & a_1 \\ a_2 & a_3 \\ \vdots & \vdots \\ a_{n-2} & a_{n-1} \\ a_{n-1} & \end{matrix} & = & \begin{matrix} \text{Coluna } 2k & \text{Coluna } 2k+1 \\ | & | \\ \omega^{2jk} & \omega^j \cdot \omega^{2jk} \\ | & | \\ \text{Linha } j & \text{Linha } j+n/2 \\ | & | \\ a_0 & a_0 \\ a_2 & a_2 \\ \vdots & \vdots \\ a_{n-2} & a_{n-2} \\ a_1 & a_1 \\ a_3 & a_3 \\ \vdots & \vdots \\ a_{n-1} & a_{n-1} \\ a_{n-1} & \end{matrix} \end{array}$$

No segundo passo, simplificamos as células na metade inferior da matriz usando $\omega^{n/2} = -1$ e $\omega^n = 1$. Note que a submatriz $n/2 \times n/2$ do canto superior esquerdo é $M_{n/2}(\omega^2)$, assim como a do canto inferior esquerdo. E as submatrizes dos cantos superior e inferior direito são quase iguais à $M_{n/2}(\omega^2)$, mas com suas j -ésimas linhas multiplicadas inteiramente por ω^j e $-\omega^j$, respectivamente. Portanto o produto final é o vetor:

$$\begin{array}{c} \text{Linha } j \\ \left[\begin{array}{cc|cc} M_{n/2} & \begin{matrix} a_0 & a_1 \\ a_2 & a_3 \\ \vdots & \vdots \\ a_{n-2} & a_{n-1} \end{matrix} & +\omega^j & M_{n/2} \\ M_{n/2} & \begin{matrix} a_0 & a_1 \\ a_2 & a_3 \\ \vdots & \vdots \\ a_{n-2} & a_{n-1} \end{matrix} & -\omega^j & M_{n/2} \end{array} \right] \\ j+n/2 \end{array}$$

Em resumo, o produto de $M_n(\omega)$ com o vetor (a_0, \dots, a_{n-1}) , um problema de tamanho n , pode ser expresso em termos de dois subproblemas de tamanho $n/2$: o produto de $M_{n/2}(\omega^2)$ com $(a_0, a_2, \dots, a_{n-2})$ e com $(a_1, a_3, \dots, a_{n-1})$. Esta estratégia de divisão-

e-conquista leva ao algoritmo definitivo para TRF da Figura 2.9, cujo tempo de execução é $T(n) = 2T = (n/2) + O(n) = O(n \log n)$.

Figura 2.9 A transformada rápida de Fourier.

função TRF(a, ω)

Entrada: Um vetor $a = (a_0, a_1, \dots, a_{n-1})$, para n uma potência de 2

Uma raiz n -ésima primitiva da unidade, ω

Saída: $M_n(\omega)a$

se $\omega = 1$: retorna a

$(s_0, s_1, \dots, s_{n/2-1}) = \text{TRF}((a_0, a_2, \dots, a_{n-2}), \omega^2)$

$(s'_0, s'_1, \dots, s'_{n/2-1}) = \text{TRF}((a_1, a_3, \dots, a_{n-1}), \omega^2)$

para $j = 0$ até $n/2 - 1$:

$$r_j = s_j + \omega^j s'_j$$

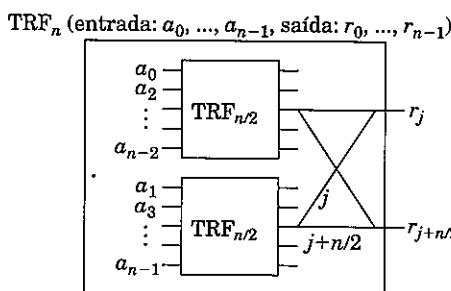
$$r_{j+n/2} = s_j - \omega^j s'_j$$

retorna $(r_0, r_1, \dots, r_{n-1})$

A transformada rápida de Fourier desdobrada

Por toda a nossa discussão, a transformada rápida de Fourier permaneceu firmemente encapsulado dentro de um formalismo de divisão-e-conquista. Para expormos totalmente sua estrutura, agora desdobramos a recursão.

O passo de divisão-e-conquista da TRF pode ser desenhado como um circuito muito simples. Apresentamos a seguir como um problema de tamanho n é reduzido a dois subproblemas de tamanho $n/2$ (por clareza, um par de saídas $(j, j + n/2)$ é destacado):



Estamos usando uma abreviação particular: as arestas são fios carregando números complexos da esquerda para a direita. Um peso de j significa “multiplique o número neste fio por ω^j ”. E quando dois fios chegam a uma junção vindos da esquerda, os números que eles carregam são adicionados. Assim, as duas saídas ilustradas estão executando os comandos

$$r_j = s_j + \omega^j s'_j$$

$$r_{j+n/2} = s_j - \omega^j s'_j$$

do algoritmo TRF (Figura 2.9), via um padrão de fios conhecido por *borboleta*: \times .

Desdobrando o circuito TRF completamente para $n = 8$ elementos, obtemos a Figura 2.10. Note o seguinte.

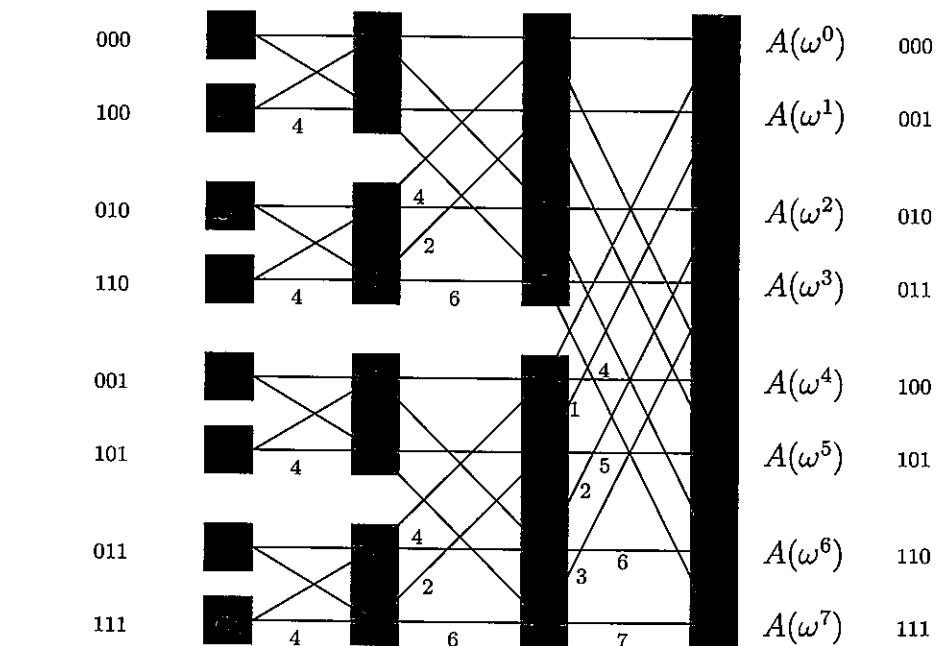
1. Para n entradas existem $\log_2 n$ níveis, cada um com n nós, totalizando $n \log n$ operações.
2. As entradas são arranjadas em uma ordem peculiar: 0, 4, 2, 6, 1, 5, 3, 7.

Por quê? Recorde que no nível mais alto da recursão, primeiro processamos os coeficientes pares da entrada e, depois, prosseguimos para os ímpares. Então, no próximo nível, os coeficientes pares do primeiro grupo (que, portanto, são múltiplos de 4 ou, equivalentemente, têm zero com seus dois bits menos significativos) são processados e assim por diante. Em outras palavras, as entradas são arranjadas em ordem crescente por *último* bit da representação binária de seus índices, resolvendo os empates examinando o(s) bit(s) mais significativo(s). A ordem resultante em binário, 000, 100, 010, 110, 001, 101, 011, 111, é a mesma que a ordem natural, 000, 001, 010, 011, 100, 101, 110, 111 *exceto que os bits estão espelhados!*

3. Existe um único caminho entre cada entrada a_j e cada saída $A(\omega^k)$.

Esse caminho é mais facilmente descrito usando as representações binárias de j e k (mostradas na Figura 2.10). Há duas arestas saindo de cada nó, uma indo para cima

Figura 2.10 O circuito da transformada rápida de Fourier.



(a aresta 0) e uma indo para baixo (a aresta 1). Para obter $A(\omega^k)$ de qualquer nó de entrada, simplesmente siga as arestas especificadas na representação binária de k , começando pelo bit mais à direita. (Você pode especificar, de maneira similar, o caminho na direção inversa?)

4. No caminho entre a_j e $A(\omega^k)$, os rótulos totalizam $jk \bmod 8$.

Como $\omega^8 = 1$, isso significa que a contribuição da entrada a_j para a saída $A(\omega^k)$ é $a_j \omega^{jk}$ e, assim, o circuito computa corretamente os valores do polinômio $A(x)$.

5. E, finalmente, note que o circuito da TRF é um candidato natural para computação paralela e implementação direta em hardware.

A lenta disseminação de um algoritmo rápido

Em 1963, durante uma reunião do conselheiro científico do presidente Kennedy, John Tukey, um matemático de Princeton, explicou para Dick Garwin, da IBM, um método rápido para computar transformadas de Fourier. Garwin escutou atentamente, porque naquele momento trabalhava em formas de detectar explosões nucleares por meio de dados sismográficos, e transformadas de Fourier eram o gargalo do seu método. Quando ele retornou à IBM, pediu a John Cooley que implementasse o algoritmo de Tukey; eles decidiram que um artigo deveria ser publicado de forma que a idéia não pudesse ser patenteada.

Tukey não tinha muito interesse em escrever um artigo sobre o assunto, portanto Cooley tomou a iniciativa. E foi assim que um dos mais famosos e mais citados artigos científicos foi publicado em 1965, em co-autoria de Cooley e Tukey. A razão pela qual Tukey estava relutante em publicar a TRF não era sigilo ou a busca de lucros com patentes. Ele apenas achou que isto era uma simples observação que provavelmente já era conhecida. Tal visão foi típica do período: naquela época (e por algum tempo mais) algoritmos eram considerados objetos matemáticos de segunda classe, desprovidos de profundidade e elegância e indignos de atenção séria.

Mas Tukey estava certo sobre uma coisa: foi descoberto mais tarde que engenheiros britânicos já haviam usado a TRF para cálculos manuais durante o final dos anos de 1930. E — para terminar este capítulo com o mesmo grande matemático com o qual ele começou — um artigo de Gauss do início dos anos de 1800 sobre (o que mais?) interpolação continha essencialmente a mesma idéia! O artigo de Gauss havia permanecido em segredo por tanto tempo porque estava protegido por uma técnica criptográfica fora de moda: assim como muitos artigos científicos de sua era, estava escrito em latim.

Exercícios

- 2.1. Use o algoritmo de multiplicação inteira por divisão-e-conquista para multiplicar os dois inteiros binários 10011011 e 10111010.
- 2.2. Mostre que, para qualquer inteiro positivo n e qualquer base b , tem de haver alguma potência de b no intervalo $[n, bn]$.

102

2.3. A Seção 2.2 descreve um método para resolver relações de recorrência baseado na análise da árvore de recursão, derivando uma fórmula para o trabalho feito a cada nível. Outro método (fortemente relacionado) é expandir a recorrência umas poucas vezes, até emergir um padrão. Por exemplo, vamos começar com a familiar $T(n) = 2T(n/2) + O(n)$. Pense em $O(n)$ como $\leq cn$ para alguma constante c , assim: $T(n) \leq 2T(n/2) + cn$. Aplicando sucessivamente essa regra, podemos limitar $T(n)$ em termos de $T(n/2)$, depois $T(n/4)$, depois $T(n/8)$ e assim por diante, em cada passo, chegando mais perto do valor de $T(\cdot)$ que conhecemos, a saber $T(1) = O(1)$.

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &\leq 2[2T(n/4) + cn/2] + cn = 4T(n/4) + 2cn \\ &\leq 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn \\ &\leq 8[2T(n/16) + cn/8] + 3cn = 16T(n/16) + 4cn \\ &\vdots \end{aligned}$$

Um padrão está emergindo ... o termo geral é

$$T(n) \leq 2^k T(n/2^k) + kcn.$$

Conectando $k = \log_2 n$ a esse termo, obtemos $T(n) \leq nT(1) + cn\log_2 n = O(n \log n)$.

- (a) Faça a mesma coisa para a recorrência $T(n) = 3T(n/2) + O(n)$. Qual é o k -ésimo termo geral neste caso? E qual valor de k deve ser conectado para se obter a resposta?
- (b) Agora tente a recorrência $T(n) = T(n - 1) + O(1)$, um caso que não é coberto pelo teorema mestre. Você pode resolver este também?

2.4. Suponha que esteja escolhendo entre os seguintes três algoritmos:

- Algoritmo *A* resolve problemas dividindo-os em cinco subproblemas de metade do tamanho, solucionando cada subproblema recursivamente e, então, combinando as soluções em tempo linear.
- Algoritmo *B* resolve problemas de tamanho n resolvendo recursivamente dois subproblemas de tamanho $n - 1$ e, então, combinando as soluções em tempo constante.
- Algoritmo *C* soluciona problemas de tamanho n dividindo-os em nove subproblemas de tamanho $n/3$, resolvendo recursivamente cada subproblema e, então, combinando as respostas em tempo $O(n^2)$.

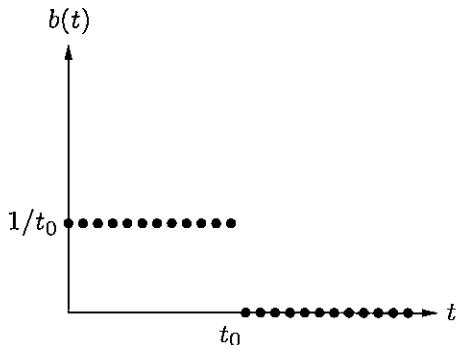
Qual o tempo de execução de cada um desses algoritmos (em notação O) e qual você escolheria?

2.5. Resolva as seguintes relações de recorrência e dê uma cota Θ para cada uma delas.

- (a) $T(n) = 2T(n/3) + 1$
- (b) $T(n) = 5T(n/4) + n$
- (c) $T(n) = 7T(n/7) + n$
- (d) $T(n) = 9T(n/3) + n^2$

- (e) $T(n) = 8T(n/2) + n^3$
- (f) $T(n) = 49T(n/25) + n^{3/2}\log n$
- (g) $T(n) = T(n - 1) + 2$
- (h) $T(n) = T(n - 1) + n^c$, onde $c \geq 1$ é uma constante
- (i) $T(n) = T(n - 1) + c^n$, onde $c > 1$ é alguma constante
- (j) $T(n) = 2T(n - 1) + 1$
- (k) $T(n) = T(\sqrt{n}) + 1$

2.6. Um sistema linear e invariante no tempo tem a seguinte resposta ao impulso:



- (a) Descreva em palavras o efeito desse sistema.
 - (b) Qual é o polinômio correspondente?
- 2.7. Qual o resultado da soma das raízes n -ésimas da unidade? Qual o seu produto se n é ímpar? Se n é par?
- 2.8. Pratique com a transformada rápida de Fourier.
- (a) Qual a TRF de $(1, 0, 0, 0)$? Qual o valor apropriado de ω neste caso? E $(1, 0, 0, 0)$ é a TRF de qual seqüência?
 - (b) Repita para $(1, 0, 1, -1)$.
- 2.9. Pratique com multiplicação de polinômios usando TRF.
- (a) Suponha que você queira multiplicar os dois polinômios $x + 1$ e $x^2 + 1$ usando TRF. Escolha uma potência de dois apropriada, encontre a TRF das duas seqüências, multiplique os resultados por componente, e compute a TRF inversa para obter o resultado final.
 - (b) Repita para o par de polinômios $1 + x + 2x^2$ e $2 + 3x$.
- 2.10. Encontre o único polinômio de grau 4 que assume valores $p(1) = 2$, $p(2) = 1$, $p(3) = 0$, $p(4) = 4$ e $p(5) = 0$. Escreva a sua resposta na representação de coeficientes.

- 2.11. Ao justificarmos nosso algoritmo para multiplicação de matrizes (Seção 2.5), afirmamos a seguinte propriedade: se X e Y são matrizes $n \times n$ e

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix},$$

onde A, B, C, D, E, F, G e H são submatrizes $n/2 \times n/2$, então o produto XY pode ser expresso em termos dessas submatrizes:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Prove essa propriedade.

- 2.12. Quantas linhas, em função de n (em forma de $\Theta(\cdot)$), o seguinte programa imprime? Escreva e resolva uma recursão. Você pode considerar que n é uma potência de 2.

```
função f(n)
    se n > 1:
        imprime_linha("ainda rodando")
        f(n/2)
        f(n/2)
```

- 2.13. Uma árvore binária é *cheia* se todos os seus vértices têm ou zero ou dois filhos. Seja B_n o número de árvores binárias cheias com n vértices.

- (a) Fazendo desenhos de todas as árvores binárias cheias com 3, 5 ou 7 vértices, determine o valor exato de B_3 , B_5 e B_7 . Por que deixamos de fora números pares de vértices, como B_4 ?
- (b) Para n geral, derive uma relação de recorrência para B_n .
- (c) Mostre por indução que B_n é $2^{\Omega(n)}$.

- 2.14. É dado um vetor de n elementos e você nota que alguns dos elementos são duplicados, ou seja, eles aparecem mais de uma vez no vetor. Mostre como remover todos os duplicados do vetor em tempo $O(n \log n)$.

- 2.15. No nosso algoritmo de busca da mediana (Seção 2.4), uma primitiva básica é a operação de *divisão*, que toma como entrada um vetor S e um valor v e, então, divide S em três conjuntos: os elementos menores do que v , os elementos iguais a v e os elementos maiores do que v . Mostre como implementar essa operação de *divisão in place*, isto é, sem alocar memória adicional.

- 2.16. É dado um vetor infinito $A[\cdot]$ no qual as primeiras n células contêm inteiros em ordem crescente e o restante das células é preenchido com ∞ . Não é dado o valor de n . Descreva um algoritmo que toma um inteiro x como entrada e encontre uma posição no vetor contendo x , se tal posição existir, em tempo $O(\log n)$. (Se você estiver incomodado com o fato de o vetor A ter tamanho infinito, considere alternativamente que ele tenha tamanho n , mas que você não conheça este tamanho e que a implementação do tipo de dados vetor na sua linguagem de programação retorne a mensagem de erro ∞ sempre que elementos $A[i]$ com $i > n$ são acessados.)

- 2.17. Dado um vetor ordenado de inteiros distintos $A[1, \dots, n]$, você quer saber se existe um índice i para o qual $A[i] = i$. Dê um algoritmo de divisão-e-conquista que execute em tempo $O(\log n)$.
- 2.18. Considere a tarefa de procurar em um vetor ordenado $A[1 \dots n]$ por um elemento x : uma tarefa normalmente realizada com busca binária em tempo $O(\log n)$. Mostre que qualquer algoritmo que acesse o vetor usando apenas comparações (ou seja, fazendo perguntas da forma “ $A[i] \leq z?$ ”) tem de tomar $\Omega(\log n)$ passos.
- 2.19. Uma operação de merge em k vias. Suponha que você tenha k vetores ordenados, cada um com n elementos e você queira combiná-los em um único vetor ordenado de kn elementos.
- Aqui está uma estratégia: Usando o procedimento `merge` da Seção 2.3, combine os primeiros dois vetores, depois combine o resultado com o terceiro, depois com o quarto e assim por diante. Qual a complexidade de tempo deste algoritmo, em termos de k e n ?
 - Dê uma solução mais eficiente para este problema, usando divisão-e-conquista.
- 2.20. Mostre que qualquer vetor de inteiros $x[1 \dots n]$ pode ser ordenado em tempo $O(n + M)$, onde

$$M = \max_i x_i - \min_i x_i.$$

Para M pequeno, isso é tempo linear: por que a cota inferior $\Omega(n \log n)$ não se aplica neste caso?

- 2.21. Média e mediana. Uma das tarefas mais básicas em estatística é resumir um conjunto de observações $\{x_1, x_2, \dots, x_n\} \subseteq \mathbb{R}$ com um único número. Duas escolhas populares para essa estatística sumária são:
- A mediana, que chamaremos de μ_1
 - A média, que chamaremos de μ_2

- (a) Mostre que a mediana é o valor de μ que minimiza a função

$$\sum_i |x_i - \mu|.$$

Você pode assumir por simplicidade que n é ímpar. (Dica: Mostre que para qualquer $\mu \neq \mu_1$, a função decresce se você movimenta μ ligeiramente para a esquerda ou para a direita.)

- (b) Mostre que a média é o valor de μ que minimiza a função

$$\sum_i (x_i - \mu)^2.$$

Uma maneira de fazer isso é usando cálculo. Outro método é provar que para qualquer $\mu \in \mathbb{R}$,

$$\sum_i (x_i - \mu)^2 = \sum_i (x_i - \mu_2)^2 + n(\mu - \mu_2)^2.$$

Note como a função para μ_2 penaliza pontos que estão distantes de μ muito mais pesadamente do que a função para μ_1 . Assim μ_2 se esforça muito mais para estar perto de *todas* as observações. Isso pode soar como uma boa coisa em certo sentido, mas é estatisticamente indesejável porque uns poucos pontos fora da curva podem jogar longe a estimativa de μ_2 . Portanto, às vezes se diz que μ_1 é uma estimativa mais robusta do que μ_2 . Pior do que ambos, entretanto, é μ_∞ , o valor de μ que minimiza a função

$$\max_i |x_i - \mu|.$$

- (c) Mostre que μ_∞ pode ser computado em tempo $O(n)$ (considerando que os números x_i são pequenos o suficiente para que operações aritméticas básicas sobre eles tomem tempo unitário).
- 2.22. São dadas duas listas ordenadas de tamanho m e n . Dê um algoritmo de tempo $O(\log m + \log n)$ para computar o k -ésimo menor elemento da união das duas listas.
- 2.23. Diz-se que um vetor $A[1 \dots n]$ possui um *elemento majoritário* se mais do que a metade de seus elementos são iguais. Dado um vetor, a tarefa é projetar um algoritmo eficiente para dizer se o vetor possui um elemento majoritário e, se afirmativo, encontrar esse elemento.
Os elementos do vetor não são necessariamente de algum domínio ordenado como os inteiros e, portanto, não podem existir comparações do tipo “ $A[i] > A[j]$ ”? (Pense nos elementos do vetor como, digamos, arquivos GIF.) Entretanto, você pode responder questões da forma: “ $A[i] = A[j]?$ ” em tempo constante.
- (a) Mostre como resolver este problema em tempo $O(n \log n)$. (Dica: Divida o vetor A em dois vetores A_1 e A_2 de metade do tamanho. Será que conhecer o elemento majoritário de A_1 e de A_2 ajuda a descobrir o elemento majoritário de A ? Se afirmativo, você pode usar a abordagem divisão-e-conquista.)
- (b) Você pode fornecer um algoritmo linear? (Dica: Aqui está uma outra abordagem de divisão-e-conquista:
- Emparelhe os elementos de A arbitrariamente, para obter $n/2$ pares
 - Analise cada par: se os dois elementos forem diferentes, descarte ambos; se forem iguais, mantenha apenas um deles
- Mostre que depois desse procedimento restam no máximo $n/2$ elementos e que eles têm um elemento majoritário se A também tem.)
- 2.24. Na página 56 há uma descrição em alto nível do algoritmo *quicksort*.
- (a) Escreva o pseudocódigo para o *quicksort*.
- (b) Mostre que seu tempo de execução de *pior caso* sobre um vetor de tamanho n é $\Theta(n^2)$.
- (c) Mostre que seu tempo de execução *esperado* satisfaz à relação de recorrência

$$T(n) \leq O(n) + \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-i)).$$

Depois, mostre que a solução para essa recorrência é $O(n \log n)$.

- 2.25. Na Seção 2.1 descrevemos um algoritmo para multiplicar dois inteiros binários de n bits x e y em tempo n^a , onde $a = \log_2 3$. Chame esse procedimento de `multiplica_rápido(x, y)`.

- (a) Queremos converter o inteiro decimal 10^n (um 1 seguido de n zeros) para binário. Aqui está o algoritmo (considere que n é uma potência de 2):

```
função pot2bin(n)
    se n = 1: retorna 10102
    senão:
        z = ???
        retorna multiplica_rápido(z, z)
```

Complete os detalhes que estão faltando. Depois forneça uma relação de recorrência para o tempo de execução do algoritmo e resolva a recorrência.

- (b) Depois, queremos converter qualquer inteiro decimal x com n bits (onde n é uma potência de 2) para binário. O algoritmo é o seguinte:

```
função dec2bin(x)
    se n = 1: retorna binário[x]
    senão:
        divida x em dois números decimais  $x_L, x_R$  com  $n/2$  dígitos cada
        retorna ???
```

Aqui, `binário[·]` é um vetor que contém a representação binária de todos os inteiros de um dígito. Ou seja, $\text{binário}[0] = 0_2$, $\text{binário}[1] = 1_2$, até $\text{binário}[9] = 1001_2$. Considere que um acesso `binário` toma tempo $O(1)$. Complete os detalhes ausentes. Mais uma vez, forneça uma recorrência para o tempo de execução do algoritmo e a resolva.

- 2.26. O professor F. Lake conta para seus alunos que é assintoticamente mais rápido elevar um número de n bits ao quadrado do que multiplicar dois inteiros de n bits. Eles devem acreditar nele?

- 2.27. O *quadrado* de uma matriz A é seu produto consigo mesma, AA .

- (a) Mostre que cinco multiplicações são suficientes para computar o quadrado de uma matriz 2×2 .
- (b) O que está errado com o seguinte algoritmo para computar o quadrado de uma matriz $n \times n$?

"Use uma abordagem de divisão-e-conquista como no algoritmo de Strassen, exceto que em vez de obter 7 subproblemas de tamanho $n/2$, agora obtém 5 subproblemas de tamanho $n/2$ graças ao item (a). Usando a mesma análise do algoritmo de Strassen, podemos concluir que o algoritmo executa em tempo $O(n^{\log_2 5})$."

- (c) De fato, elevar matrizes ao quadrado não é mais fácil do que multiplicar matrizes. Mostre que se matrizes $n \times n$ podem ser elevadas ao quadrado em tempo $O(n^c)$, então quaisquer duas matrizes $n \times n$ podem ser multiplicadas em tempo $O(n^c)$.

- 2.28. As matrizes de Hadamard H_0, H_1, H_2, \dots são definidas como se segue:

- H_0 é a matriz 1×1 [1]

- Para $k > 0$, H_k é a matriz $2^k \times 2^k$

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array} \right].$$

Mostre que se v é um vetor coluna de tamanho $n = 2^k$, então o produto matriz-vetor $H_k v$ pode ser calculado usando $O(n \log n)$ operações. Considere que todos os números envolvidos sejam pequenos o suficiente para que operações aritméticas básicas como adição e multiplicação tomem tempo unitário.

- 2.29. Suponha que queiramos avaliar o polinômio $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ no ponto x .

- (a) Mostre que a seguinte rotina simples, conhecida como *regra de Horner*, realiza o trabalho e deixa a resposta em z .

```

 $z = a_n$ 
para  $i = n - 1$  até 0:
 $z = zx + a_i$ 

```

- (b) Quantas adições e multiplicações essa rotina usa, como uma função de n ? Você pode encontrar um polinômio para o qual um método alternativo é substancialmente melhor?

- 2.30. Este problema ilustra como fazer a transformada de Fourier (TF) em aritmética modular, por exemplo, módulo 7.

- (a) Existe um número ω tal que todas as potências $\omega, \omega^2, \dots, \omega^6$ são distintas (módulo 7). Encontre esse ω e mostre que $\omega + \omega^2 + \dots + \omega^6 = 0$. (Interessantemente, para qualquer módulo primo existe um tal número.)

- (b) Usando a forma de matriz da TF, produza a transformada da seqüência $(0, 1, 1, 1, 5, 2)$ módulo 7, ou seja, multiplique esse vetor pela matriz $M_6(\omega)$, para o valor de ω que você encontrou antes. Na multiplicação de matrizes, todos os cálculos devem ser realizados módulo 7.

- (c) Escreva a matriz necessária para realizar a TF inversa. Mostre que multiplicar por essa matriz retorna a seqüência original. (Novamente, toda aritmética deve ser feita módulo 7.)

- (d) Agora mostre como multiplicar os polinômios $x^2 + x + 1$ e $x^3 + 2x - 1$ usando a TF módulo 7.

- 2.31. Na Seção 1.2.3, estudamos o algoritmo de Euclides para computar o *máximo divisor comum* (mdc) de dois inteiros positivos: o maior inteiro que divide ambos. Aqui vamos examinar um algoritmo alternativo baseado em divisão-e-conquista.

- (a) Mostre que a seguinte regra é verdadeira.

$$\text{mdc}(a, b) = \begin{cases} 2 \text{ mdc}(a/2, b/2) & \text{se } a, b \text{ são pares} \\ \text{mdc}(a, b/2) & \text{se } a \text{ é ímpar, } b \text{ é par} \\ \text{mdc}((a - b)/2, b) & \text{se } a, b \text{ são ímpares} \end{cases}$$

- (b) Forneça um algoritmo de divisão-e-conquista eficiente para o máximo divisor comum.
- (c) Como a eficiência do seu algoritmo se compara com a do algoritmo de Euclides se a e b são inteiros de n bits? (Em particular, como n pode ser grande, você não pode considerar que operações aritméticas básicas como adição tomam tempo constante.)
- 2.32. Neste problema vamos desenvolver um algoritmo de divisão-e-conquista para a seguinte tarefa geométrica.

PAR MAIS PRÓXIMO

Entrada: Um conjunto de pontos no plano, $\{p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)\}$

Saída: O par mais próximo: ou seja, o par $p_i \neq p_j$ para o qual a distância entre p_i e p_j , isto é,

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2},$$

é minimizada.

Por simplicidade, considere que n seja uma potência de dois e que todas as coordenadas x, x_i sejam distintas, bem como as coordenadas y .

A seguir uma visão geral do algoritmo:

- Encontre um valor x para o qual exatamente metade dos pontos tem $x_i < x$ e metade tem $x_i > x$. Com base nisso, divida os pontos em dois grupos, L e R .
 - Encontre recursivamente o par mais próximo em L e em R . Digamos que sejam $p_L, q_L \in L$ e $p_R, q_R \in R$, com distâncias d_L e d_R , respectivamente. Seja d a menor dessas duas distâncias.
 - Resta saber se existe um ponto em L e um ponto em R que estão a uma distância menor do que d um do outro. Para esse fim, descarte todos os pontos com $x_i < x - d$ ou $x_i > x + d$ e ordene os pontos restantes por coordenada y .
 - Agora, percorra a lista ordenada e, para cada ponto, compute a sua distância para os sete pontos subsequentes da lista. Seja p_M, q_M o par mais próximo encontrado desta maneira.
 - A resposta é um dos três pares $\{p_L, q_L\}$, $\{p_R, q_R\}$, $\{p_M, q_M\}$, aquele que for mais próximo.
- (a) Para provar a correção deste algoritmo, comece mostrando a seguinte propriedade: qualquer quadrado de tamanho $d \times d$ no plano contém no máximo quatro pontos de L .
- (b) Agora mostre que o algoritmo é correto. O único caso que necessita de consideração cuidadosa é quando o par mais próximo está dividido entre L e R .

- (c) Escreva o pseudocódigo para o algoritmo e mostre que seu tempo de execução é dado pela recorrência:

$$T(n) = 2T(n/2) + O(n \log n).$$

Mostre que a solução para essa recorrência é $O(n \log^2 n)$.

- (d) Você pode diminuir o tempo de execução para $O(n \log n)$?

- 2.33. Suponha que sejam dadas matrizes $n \times n$, A , B , C e você queira saber se $AB = C$. Você pode fazer isso em $O(n^{\log_2 7})$ passos usando o algoritmo de Strassen. Nesta questão vamos explorar um teste randomizado muito mais rápido, de tempo $O(n^2)$.

- (a) Seja v um vetor n -dimensional cujas células são aleatórias e independentemente escolhidas entre 0 e 1 (cada com probabilidade 1/2). Prove que se M é uma matriz $n \times n$ não-nula, então $\Pr[Mv = 0] \leq 1/2$.
- (b) Mostre que $\Pr[ABv = Cv] \leq 1/2$ se $AB \neq C$. Por que isso leva a um teste randomizado de tempo $O(n^2)$ para checar se $AB = C$?

- 2.34. *3SAT linear.* O problema 3SAT é definido na Seção 8.1. Resumidamente, a entrada é uma fórmula booleana — expressa como um conjunto de cláusulas — sobre algum conjunto de variáveis e o objetivo é determinar se existe uma atribuição (de valores verdadeiro/falso) a essas variáveis que faça a fórmula inteira avaliar como verdadeiro.

Considere uma instância de 3SAT com a seguinte propriedade especial de localidade. Suponha que existam n variáveis na fórmula booleana e que elas estejam numeradas 1, 2, ..., n de tal maneira que cada cláusula envolve variáveis cujos números estão em um intervalo de ± 10 um do outro. Forneça um algoritmo de tempo linear para solucionar uma tal instância de 3SAT.

Capítulo 3

Decomposição de grafos

3.1 Por que grafos?

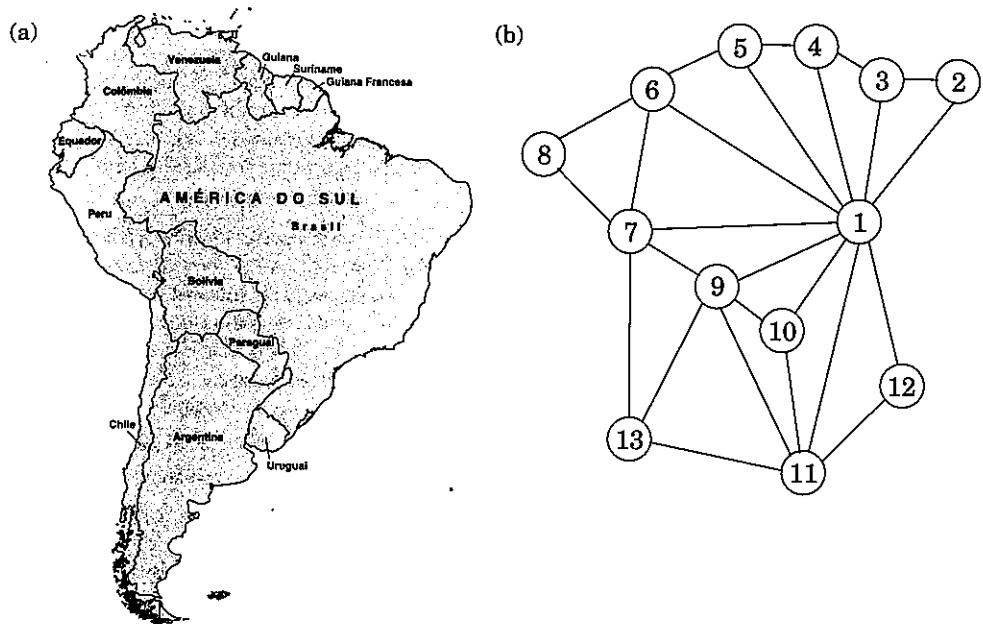
Uma ampla variedade de problemas pode ser expressa com clareza e precisão na linguagem pictórica, concisa, dos grafos. Por exemplo, considere a tarefa de colorir um mapa político. Qual o número mínimo de cores necessário, com a restrição óbvia de que países vizinhos devem ter cores diferentes? Uma das dificuldades em abordar este problema é que o mapa em si, mesmo uma versão simplificada como a da Figura 3.1(a), é normalmente sobrecarregado de informação irrelevante: fronteiras intrincadas, postos de fronteira onde três ou mais países se encontram, o mar aberto e rios serpenteando pelo mapa. Tais distrações estão ausentes do objeto matemático da Figura 3(b), um grafo com um *vértice* para cada país (1 é o Brasil, 11 a Argentina) e *arestas* entre vizinhos. Ele contém exatamente a informação necessária para coloração e nada mais. O objetivo preciso agora é atribuir uma cor a cada vértice para que nenhuma aresta tenha extremidades da mesma cor.

Coloração de grafos não é domínio exclusivo de desenhistas de mapas. Suponha que uma universidade precise agendar exames para todas as suas turmas e queira usar o menor número de horários possível. A única restrição é que dois exames não podem ser agendados concorrentemente se algum aluno irá fazer ambos. Para expressar esse problema como um grafo, use um vértice para cada exame e coloque uma aresta entre dois vértices se existe um conflito, ou seja, se existe alguém tomando ambos os exames, das duas extremidades. Idealize cada horário com sua própria cor: atribuir horários é exatamente o mesmo que colorir o grafo!

Algumas operações básicas em grafos aparecem com tal freqüência e em tal diversidade de contextos, que muito esforço foi feito para se encontrar procedimentos eficientes para elas. Este capítulo é dedicado a alguns dos mais fundamentais entre esses algoritmos — aqueles que revelam a estrutura da conectividade básica de um grafo.

Formalmente, um grafo é especificado por um conjunto de vértices (também chamados de *nós*) V e por arestas E entre alguns pares de vértices. No exemplo do mapa, $V = \{1, 2, 3, \dots, 13\}$ e E inclui, entre muitas outras arestas, $\{1, 2\}$, $\{9, 11\}$ e $\{7, 13\}$. Aqui uma aresta entre x e y especificamente significa “ x faz fronteira com y ”. Essa

Figura 3.1 (a) Um mapa e (b) seu grafo.



relação é simétrica — ela implica também que y faz fronteira com x — e a denotamos usando notação de conjuntos, $e = \{x, y\}$. Tais arestas são *não-direcionadas* e são parte de um *grafo não-direcionado*.

Às vezes, grafos representam relações que não possuem esta propriedade, caso no qual é necessário usar arestas com direções. Pode haver *arestas direcionadas e de x para y* (escrito $e = (x, y)$), ou de y para x (escrito (y, x)) ou ambas. Um exemplo, particularmente enorme, de um *grafo direcionado* é o grafo de todos os links na World Wide Web. Ele possui um vértice para cada site na Internet e uma aresta direcionada (u, v) sempre que o site u tem um link para o site v : no total, bilhões de nós e arestas! Entender as mais básicas propriedades de conectividade da Web é de grande interesse econômico e social. Embora o tamanho desse problema seja assustador, veremos em breve que muita informação valiosa sobre a estrutura de um grafo pode, felizmente, ser determinada em tempo linear apenas.

3.1.1 Como um grafo é representado?

Podemos representar um grafo com uma *matriz de adjacência*; se há $n = |V|$ vértices v_1, \dots, v_n , então, ela é uma matriz $n \times n$ cuja (i, j) -ésima célula é

$$a_{ij} = \begin{cases} 1 & \text{se existe uma aresta de } v_i \text{ para } v_j \\ 0 & \text{caso contrário.} \end{cases}$$

Para grafos não-direcionados, a matriz é simétrica, pois uma aresta $\{1, v\}$ pode ser tomada em ambas as direções.

Qual é o tamanho do seu grafo?

Qual das duas representações, matriz de adjacência ou lista de adjacência, é melhor? Bem, depende da relação entre $|V|$, o número de nós do grafo e $|E|$, o número de arestas. $|E|$ pode ser tão pequeno quanto $|V|$ (se ele se torna muito menor, então, o grafo degenera — por exemplo, possui vértices isolados), ou tão grande quanto $|V|^2$ (quando todas as possíveis arestas estão presentes). Quando $|E|$ está perto do limite superior desse intervalo, chamamos o grafo de *denso*. No outro extremo, se $|E|$ está perto de $|V|$, o grafo é *esparsa*. Como veremos neste e nos próximos dois capítulos, *o ponto exato onde $|E|$ está neste intervalo é usualmente um fator crucial na seleção do algoritmo certo*.

Ou, aliás, na seleção da representação certa do grafo. Se é o grafo da World Wide Web que desejamos guardar na memória do computador, devemos pensar duas vezes antes de usar uma matriz de adjacência: no momento da redação deste livro, os mecanismos de busca da Internet conheciam cerca de oito bilhões de vértices deste grafo e, assim, a matriz de adjacência tomaria *dúzias de milhões de terabits*. De novo, no momento em que escrevemos estas linhas, não estava claro se existia memória de computador suficiente no mundo inteiro para alcançar isto. (E esperar alguns anos até que exista memória suficiente é insensato: a Web crescerá também e provavelmente mais rápido.)

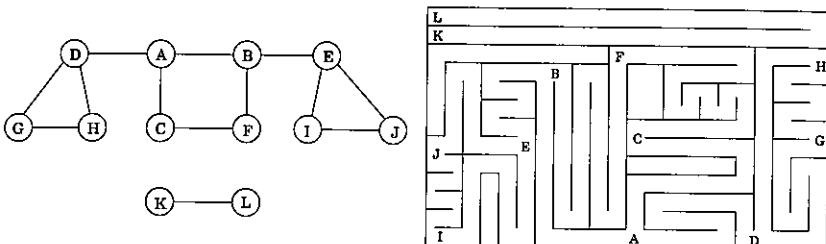
Com listas de adjacência, representar a World Wide Web torna-se factível: há apenas umas poucas dúzias de bilhões de hiperlinks na Web, e cada ocupará alguns bytes na lista de adjacência. Você pode levar um dispositivo que guarde o resultado, um terabyte ou dois, no seu bolso (vai caber em breve dentro de um brinco, mas então a Web terá crescido também).

A razão pela qual listas de adjacência são tão mais efetivas no caso da World Wide Web é que a Web é muito esparsa: uma página da Web tem em média apenas cerca de meia dúzia de hiperlinks para outras páginas, das bilhões de possibilidades.

A grande conveniência desse formato é que a presença de uma particular aresta pode ser checada em tempo constante, com apenas um acesso à memória. Por sua vez, a matriz toma espaço $O(n^2)$, o que é desperdício se o grafo não possui muitas arestas.

Uma representação alternativa, com tamanho proporcional ao número de arestas, é a *lista de adjacência*. Ela consiste em $|V|$ listas ligadas, uma por vértice. A lista ligada para o vértice u contém os nomes dos vértices para os quais sai uma aresta de u — isto é, vértices v para os quais $(u, v) \in E$. Portanto, cada aresta aparece em exatamente uma das listas ligadas se o grafo é direcionado ou duas das listas se o grafo é não-direcionado. De qualquer forma, o tamanho total da estrutura de dados é $O(|E|)$. Checar por uma particular aresta (u, v) não é mais em tempo constante, porque requer um percurso pela lista de adjacência de u . Mas é fácil iterar por todos os vizinhos de um vértice (percorrendo a lista ligada correspondente) e, como logo veremos, isto é de fato uma operação muito útil em algoritmos em grafos. Novamente, para grafos não-direcionados, esta representação possui uma simetria: v está na lista de adjacência de u se e somente se u está na lista de adjacência de v .

Figura 3.2 Explorar um grafo é bem semelhante a navegar por um labirinto.



3.2 Busca em profundidade em grafos não-direcionados

3.2.1 Explorando labirintos

Busca em profundidade é um procedimento de tempo linear surpreendentemente versátil que revela uma gama de informações sobre um grafo. A questão mais básica que ela responde é,

Quais partes do grafo são alcançáveis por meio de um dado vértice?

Para entender essa tarefa, tente se colocar na posição de um computador que acabou de receber um novo grafo, digamos na forma de uma lista de adjacência. Essa representação oferece apenas uma operação básica: encontrar os vizinhos de um vértice. Com apenas essa primitiva, o problema da alcançabilidade é bem semelhante a explorar um labirinto (Figura 3.2). Você começa a andar de um lugar fixo e sempre que chega a uma junção (vértice) existe uma variedade de caminhos (arestas) que pode seguir. Uma escolha descuidada dos caminhos pode levá-lo a andar em círculos ou a deixar de visualizar alguma parte alcançável do labirinto. Claramente, você precisa guardar alguma informação intermediária durante a exploração.

Esse desafio clássico divertiu pessoas por séculos. Todo mundo sabe que tudo o que se precisa para explorar um labirinto é um novelo de linha e um pedaço de giz. O giz previne círculos, marcando as junções que você já visitou. A linha sempre o leva de volta ao lugar inicial, possibilitando que você retorne a passagens que viu anteriormente, mas ainda não investigou.

Como podemos simular essas duas primitivas, giz e linha, em um computador? As marcas de giz são fáceis: para cada vértice, mantenha uma variável booleana indicando se ele já foi visitado. Para o novelo de linha, o análogo cibernetico correto é uma *pilha*. Além de tudo, o papel exato da linha é oferecer duas operações primitivas — *desenrolar* para alcançar uma nova junção (o equivalente na pilha é *empilhar, push*, o novo vértice) e *enrolar* para retornar à junção anterior (*desempilhar, pop*, na pilha).

Em vez de explicitamente manter uma pilha, faremos isso implicitamente por meio da recursão (implementada usando uma pilha de registros de ativação). O algoritmo resultante

Figura 3.3 Encontrando todos os nós alcançáveis partindo de um certo nó.

```
procedimento explorar( $G, v$ )
```

Entrada: $G = (V, E)$ é um grafo; $v \in V$

Saída: $\text{visitado}(u)$ é tornado verdadeiro para todos os nós u , alcançáveis partindo de v

```
visitado( $v$ ) = verdadeiro
```

```
pré-visita( $v$ )
```

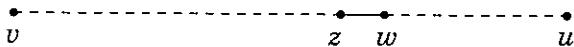
```
para cada aresta  $(v, u) \in E$ :
```

```
  se não visitado( $u$ ): explorar
```

```
pós-visita( $v$ )
```

está na Figura 3.3¹. Os procedimentos pré-visita e pós-visita são opcionais, destinados à realização de operações sobre um vértice quando ele é descoberto pela primeira vez e, também, quando é deixado pela última vez. Veremos em breve alguns usos criativos para esses procedimentos.

Antes de tudo, precisamos confirmar que **explorar** sempre funciona corretamente. Ele certamente não se aventura longe demais, porque sempre se move de nós para seus vizinhos e, portanto, nunca pode pular para uma região não alcançável partindo de v . Mas ele encontra *todos* os vértices alcançáveis partindo de v ? Bem, se há algum u que ele esqueça, escolha qualquer caminho de v até u e observe o último vértice neste caminho que o procedimento de fato visita. Chame esse nó de z e seja w o nó que vem imediatamente após nesse mesmo caminho.



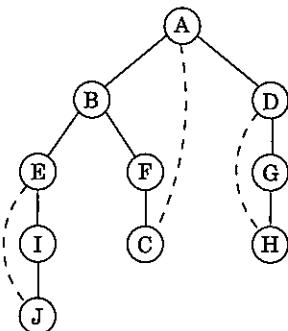
Portanto, z foi visitado, mas w não. Isso é uma contradição: quando o procedimento **explorar** estava em z , ele teria notado w e se movido para ele.

A propósito, esse padrão de raciocínio ocorre freqüentemente nos estudos de grafos e é, em essência, uma indução simplificada. Uma prova indutiva mais formal começaria por estabelecer uma hipótese, tal como “para qualquer $k \geq 0$, todos os nós a k passos partindo de v são visitados”. O caso-base, como normalmente, é trivial, pois v é certamente visitado. E o caso geral — mostrar que se todos os nós a k passos de distância são visitados, então todos os nós a $k + 1$ passos de distância também são — é precisamente o mesmo argumento que acabamos de fazer.

A Figura 3.4 mostra o resultado de executar **explorar** no nosso exemplo anterior de grafo, começando com o nó A e usando a ordem alfabética para escolher qual o próximo nó a visitar sempre que existe uma escolha. As arestas contínuas são aquelas que de fato percorremos, cada qual foi obtida por uma chamada a **explorar** e levou a descoberta

¹Assim como em muitos de nossos algoritmos em grafos, este se aplica a grafos tanto não-direcionados como direcionados. Em tais casos, adotamos a notação para arestas *direcionadas*, (x, y) . Se o grafo é não-direcionado, cada uma de suas arestas deve ser imaginada existindo nas duas direções: (x, y) e (y, x) .

Figura 3.4 O resultado de explorar(*A*) sobre o grafo da Figura 3.2.



de um novo vértice. Por exemplo, quando *B* estava sendo visitado, a aresta *B* – *E* foi notada e, como *E* ainda era desconhecido, foi percorrida por uma chamada a *explorar(E)*. As arestas contínuas formam uma árvore (um grafo conexo sem ciclos) e são, portanto, chamadas de *arestas de árvore*. As arestas pontilhadas são ignoradas porque levavam de volta a terreno conhecido, a vértices visitados anteriormente. Elas são chamadas *arestas de retorno*.

3.2.2 Busca em profundidade

O procedimento *explorar* visita somente a porção do grafo alcançável partindo de seu ponto de começo. Para examinarmos o restante do grafo, precisamos recomeçar o procedimento em outro lugar, em algum vértice que ainda não foi visitado. O algoritmo da Figura 3.5, chamado *busca em profundidade* (DFS, do inglês *depth-first search*), faz isso repetidamente até que o grafo inteiro tenha sido percorrido.

Figura 3.5. Busca em profundidade.

```

procedimento dfs(G)
para todo  $v \in V$ :
    visitado( $v$ ) = falso
para todo  $v \in V$ :
    se não visitado( $v$ ): explorar( $v$ )
  
```

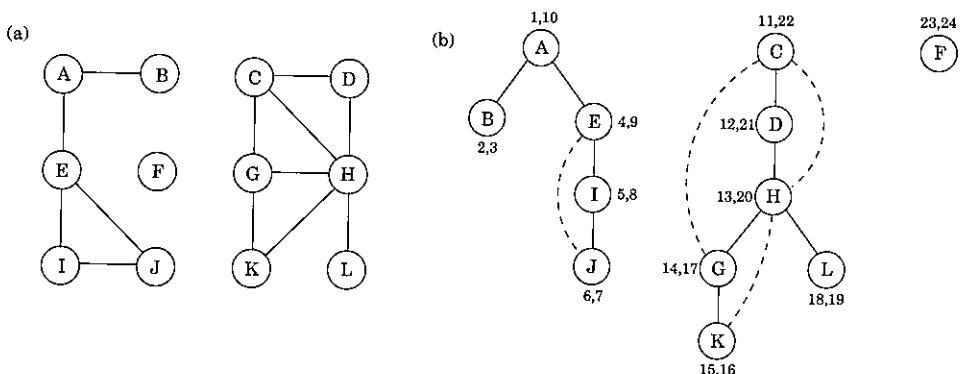
O primeiro passo da análise do tempo de execução de DFS é observar que cada vértice é explorado apenas uma vez, graças ao vetor *visitado* (as marcas de giz). Durante a exploração de um vértice, existem os seguintes passos:

1. Alguma quantidade fixa de trabalho — marcar o lugar como visitado e a pré e pós-visita.

2. Um loop no qual arestas adjacentes são examinadas, para ver se elas levam a algum lugar novo.

Esse loop toma uma quantidade de tempo diferente para cada vértice, assim vamos considerar todos os vértices juntos. O trabalho total feito no passo 1 é, então, $O(|V|)$. No passo 2, pelo curso de toda a DFS, cada aresta $\{x, y\} \in E$ é examinada exatamente *duas vezes*, uma vez durante *explorar(x)* e uma vez durante *explorar(y)*. O tempo total para o passo 2 é, portanto, $O(|E|)$ e assim a busca em profundidade tem um tempo de execução de $O(|V| + |E|)$, linear no tamanho da sua entrada. Isso é tão eficiente quanto poderíamos ter esperado, pois só ler a lista de adjacência já toma esse tanto.

Figura 3.6 (a) Um grafo de 12 nós. (b) A floresta de busca da DFS.



A Figura 3.6 mostra o resultado da busca em profundidade em um grafo de 12 nós, mas uma vez desempatando alfabeticamente (ignore os pares de números por enquanto). O loop mais externo do DFS chama *explorar* três vezes, sobre A, C e finalmente F. Como resultado, existem três árvores, cada uma enraizada em um dos três pontos de partida. Juntas constituem uma *floresta*.

3.2.3 Conectividade em grafos não-direcionados

Um grafo não-direcionado é *conexo* se existe um caminho entre qualquer par de vértices. O grafo da Figura 3.6 *não* é conexo porque, por exemplo, não há caminho entre A e K. Entretanto, ele tem três regiões conexas disjuntas, correspondentes aos seguintes conjuntos de vértices:

$$\{A, B, E, I, J\} \quad \{C, D, G, H, K, L\} \quad \{F\}.$$

Essas regiões são chamadas de *componentes conexas*: cada uma delas é um subgrafo que é internamente conexo, mas não possui arestas para os vértices restantes. Quando *explorar* é iniciado em um particular vértice, ele identifica precisamente as componentes conexas contendo aquele vértice. E cada vez que o loop externo do DFS chama *explorar*, uma nova componente conexa é extraída.

Assim, busca em profundidade pode ser trivialmente adaptada para verificar se um grafo é conexo e, mais geralmente, atribuir a cada nó v um inteiro $\text{ccnum}[v]$ identificando a componente conexa a qual ele pertence. Tudo o que é necessário é

```
procedimento pré-visita( $v$ )
   $\text{ccnum}[v] = \text{cc}$ 
```

onde cc precisa ser inicializado com zero e incrementado cada vez que o procedimento DFS chama `explorar`.

3.2.4 Pré-ordem e pós-ordem

Vimos como a busca em profundidade — umas poucas e modestas linhas de código — é capaz de revelar a estrutura de conectividade de um grafo não-direcionado em tempo apenas linear. Mas ela é muito mais versátil do que isso. Para exercitá-la mais, vamos coletar um pouco mais de informação durante o processo de exploração: para cada nó, vamos anotar o tempo de dois eventos importantes, o momento da primeira descoberta (correspondente a *pré-visita*) e aquele da partida final (*pós-visita*). A Figura 3.6 mostra esses números para nosso exemplo anterior, no qual há 24 eventos. O quinto evento é a descoberta de I . O 21º evento consiste em deixar D definitivamente.

Uma maneira de gerar vetores *pré* e *pós* com esses números é definir um simples relógio, inicialmente valendo 1, que é atualizado como segue.

```
procedimento pré-visita( $v$ )
   $\text{pré}[v] = \text{relógio}$ 
   $\text{relógio} = \text{relógio} + 1$ 
```

```
procedimento pós-visita( $v$ )
   $\text{pós}[v] = \text{relógio}$ 
   $\text{relógio} = \text{relógio} + 1$ 
```

Esses tempos irão em breve adquirir maior importância. Por enquanto, você pode ter notado da Figura 3.4 que:

Propriedade *Para quaisquer nós u e v , os dois intervalos $[\text{pré}(u), \text{pós}(u)]$ e $[\text{pré}(v), \text{pós}(v)]$ são ou disjuntos ou um está contido dentro do outro.*

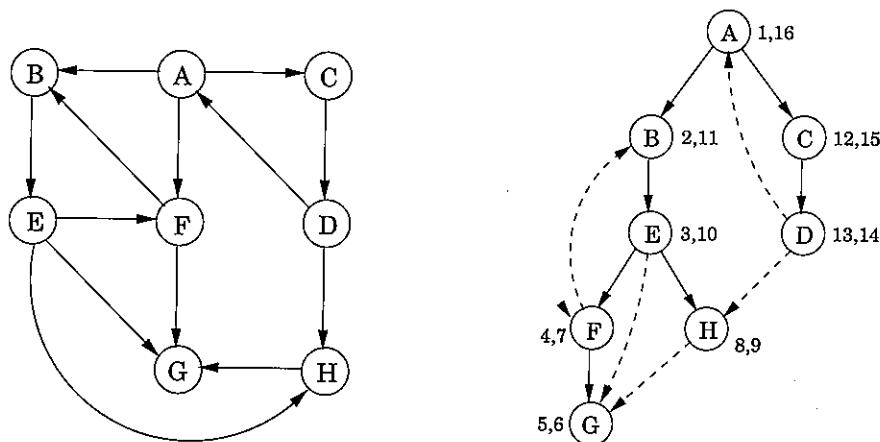
Por quê? Porque $[\text{pré}(u), \text{pós}(u)]$ é essencialmente o tempo durante o qual o vértice u esteve na pilha. O comportamento último-a-entrar, primeiro-a-sair da pilha explica o resto.

3.3 Busca em profundidade em grafos direcionados

3.3.1 Tipos de arestas

Nosso algoritmo de busca em profundidade pode ser executado do jeito que está em grafos direcionados, tomando o cuidado de percorrer arestas apenas nas suas direções prescritas: A Figura 3.7 mostra um exemplo e a árvore de busca que resulta quando os vértices são considerados em ordem lexicográfica.

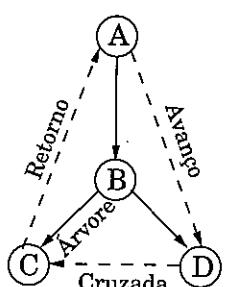
Figura 3.7 DFS sobre um grafo direcionado.



Em uma análise mais aprofundada do caso direcionado, é útil ter uma terminologia para os relacionamentos importantes entre nós da árvore. *A* é a *raiz* da árvore de busca; tudo o mais são seus *descendentes*. De forma similar, *E* tem os descendentes *F*, *G* e *H* e, de forma oposta, é um *ascendente* desses três nós. A analogia com família é estendida: *C* é o *pai* de *D*, que é seu *filho*.

Para grafos não-direcionados, distinguimos entre arestas de árvore e as demais arestas. No caso direcionado, há uma taxonomia ligeiramente mais elaborada:

Árvore DFS



Arestas de árvore são de fato parte da floresta DFS.

Arestas de avanço levam de um nó a um descendente que *não é filho* na árvore DFS.

Arestas de retorno levam a um ascendente na árvore DFS.

Arestas cruzadas não levam nem a um descendente, nem a um ascendente, elas, portanto, levam a um nó que já foi completamente explorado (isto é, já pós-visitado).

A Figura 3.7 possui duas arestas de avanço, duas arestas de retorno e duas arestas cruzadas. Você pode identificá-las?

As relações de ascendência e descendência, bem como os tipos de arestas, podem ser extraídas diretamente dos números pré e pós. Em razão da estratégia de exploração por profundidade, um vértice *u* é um ascendente de um vértice *v* exatamente naqueles

casos em que u é descoberto primeiro e v é descoberto durante $\text{explorar}(u)$. Isso é o mesmo que dizer $\text{pré}(u) < \text{pré}(v) < \text{pós}(v) < \text{pós}(u)$, que podemos representar como dois intervalos aninhados:

$$\begin{array}{cccc} [& [&] &] \\ u & v & v & u \end{array}$$

O caso de descendentes é simétrico, pois u é um descendente de v se e somente se v é um ascendente de u . E, como as categorias de arestas são baseadas inteiramente nos relacionamentos ascendente-descendente, segue que elas também podem ser extraídas dos números pré e pós. Veja um resumo das várias possibilidades para uma aresta (u, v) :

pré/pós	$\text{ordem para } (u, v)$	Tipo de aresta
$\begin{array}{cccc} [& [&] &] \\ u & v & v & u \end{array}$		Árvore/avanço
$\begin{array}{cccc} [& [&] &] \\ v & u & u & v \end{array}$		Retorno
$\begin{array}{cc} [&] \\ v & v \end{array} \quad \begin{array}{cc} [&] \\ u & u \end{array}$		Cruzada

Confirme cada uma dessas caracterizações consultando o diagrama de tipos de arestas. Você pode ver por que nenhuma outra ordem é possível?

3.3.2 Grafos direcionados acíclicos

Um *ciclo* em um grafo direcionado é um caminho circular $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_0$. A Figura 3.7 tem vários deles, por exemplo, $B \rightarrow E \rightarrow F \rightarrow B$. Um grafo sem ciclos é dito *acíclico*. E é o caso que podemos testar se um grafo é acíclico em tempo linear, com uma única busca em profundidade.

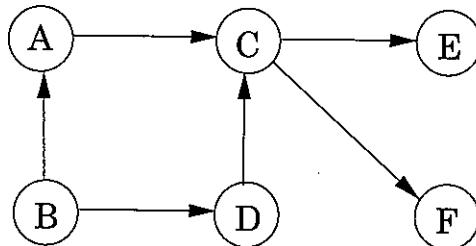
Propriedade *Um grafo direcionado possui um ciclo se e somente se a sua busca em profundidade revela uma aresta de retorno.*

Prova. Uma direção é bastante fácil: se (u, v) é uma aresta de retorno, existe um ciclo consistindo nesta aresta junto com o caminho de v até u na árvore de busca.

Para a direção oposta, se o grafo tem um ciclo $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$, observe o primeiro nó neste ciclo a ser descoberto (o nó com o menor número pré). Suponha que ele seja v_i . Todos os demais v_j no ciclo são alcançáveis partindo dele e serão, portanto, seus descendentes na árvore de busca. Em particular, a aresta $v_{i-1} \rightarrow v_i$ (ou $v_k \rightarrow v_0$ se $i = 0$) leva de um nó a seu ascendente e é, assim, por definição, uma aresta de retorno. ■

Grafos direcionados acíclicos, ou *dags* (do inglês *directed acyclic graphs*), aparecem o tempo todo. Eles são bons para modelar relações como causalidades, hierarquias e dependências temporais. Por exemplo, suponha que você precise realizar muitas tarefas, mas algumas delas não podem ser feitas antes que certas outras sejam completadas

Figura 3.8 Um grafo direcionado acíclico com uma fonte, dois sorvedouros e quatro possíveis linearizações.



(você tem de acordar antes de levantar da cama; você tem de estar fora da cama, mas não ainda vestido, para tomar um banho, e assim por diante). A questão é, então, qual é uma ordem válida para realizar as tarefas?

Tais restrições podem ser convenientemente representadas por um grafo direcionado no qual cada tarefa é um nó e existe uma aresta de u para v se u é uma precondição para v . Em outras palavras, antes de realizar uma tarefa, todas as tarefas que apontam para ela devem ser completadas. Se, por um lado, esse grafo tem um ciclo, não há esperança: nenhuma ordem pode funcionar. Se, por outro lado, o grafo é um dag, iríamos querer, se possível, *linearizá-lo* (ou *ordená-lo topologicamente*), ordenar os vértices um após o outro de tal forma que toda aresta vá de um vértice anterior para um posterior, de modo que todas as restrições de precedência estejam satisfeitas. Na Figura 3.8, por exemplo, uma ordem válida é B, A, D, C, E, F . (Você pode identificar as outras três?)

Quais tipos de dags podem ser linearizados? Simples: *todos eles*. Mais uma vez, busca em profundidade nos diz exatamente como fazer isso: simplesmente realize as tarefas em ordem *decrescente* de seus números pós. Afinal de contas, as únicas arestas (u, v) em um grafo para as quais $\text{pós}(u) < \text{pós}(v)$ são as arestas de retorno (consulte a tabela de tipos de arestas na página 88) — e vimos que um dag não pode ter arestas de retorno. Portanto:

Propriedade *Em um dag, toda aresta leva a um vértice com um número de pós menor.*

Isso nos dá um algoritmo de tempo linear para ordenar os nós de um dag. E, juntamente com nossas observações anteriores, nos diz que três propriedades que soam bem distintas — ausência de ciclo, possibilidade de linearização e ausência de arestas de retorno em uma busca em profundidade — são de fato uma só coisa.

Como um dag é linearizado por números de pós decrescentes, o vértice com o menor número de pós vem por último nesta linearização e tem de ser um *sorvedouro* — sem arestas que saem dele. De maneira simétrica, aquele com o maior número de pós é uma *fonte*, um nó sem arestas que chegam nele.

Propriedade *Todo dag tem pelo menos uma fonte e pelo menos um sorvedouro.*

A existência garantida de uma fonte sugere uma abordagem alternativa para linearização:

Encontre uma fonte, retorne-a com resposta e a remova do grafo.

Repita até que o grafo esteja vazio.

Você pode ver por que isso gera uma linearização válida para qualquer dag? O que acontece se o grafo possui ciclos? E como esse algoritmo pode ser implementado em tempo linear? (Exercício 3.14.)

3.4 Componentes fortemente conexas

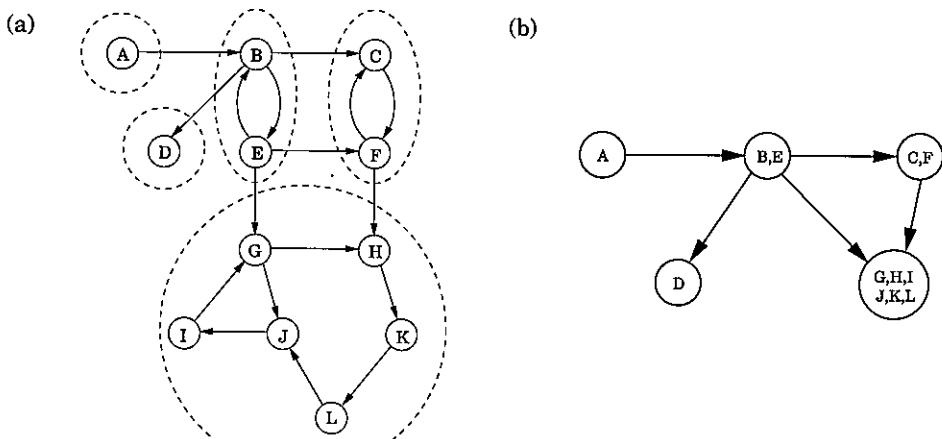
3.4.1 Definindo conectividade para grafos direcionados

Conectividade em grafos não-direcionados é bastante simples: um grafo que não é conexo pode ser decomposto de uma maneira óbvia e natural em várias componentes conexas (Figura 3.6 é um caso desse). Como vimos na Seção 3.2.3, busca em profundidades faz isso convenientemente, com cada recomeço marcando uma nova componente conexa.

Em grafos direcionados, conectividade é algo mais sutil. Em sentido primitivo, o grafo direcionado da Figura 3.9(a) é “conexo” — ele não pode ser “separado”, por assim dizer, sem remover arestas. Mas essa noção é pouco interessante ou informativa. O grafo não pode ser considerado conexo, porque, por exemplo, não há caminho de G até B ou de F até A . A maneira correta de definir conectividade para grafos direcionados é esta:

Dois nós u e v de um grafo direcionado estão conectados se existe um caminho de u até v e um caminho de v até u .

Figura 3.9 (a) Um grafo direcionado e suas componentes fortemente conexas.
(b) O metagrafo.



Essa relação particiona V em conjuntos disjuntos (Exercício 3.30) que chamamos *componentes fortemente conexas*. O grafo da Figura 3.9(a) tem cinco delas.

Agora colapse cada componente fortemente conexa em um único metanó e desenhe uma aresta de um metanó para outro se existir uma aresta (na mesma direção) entre suas respectivas componentes (Figura 3.9(b)). O *metagrafo* resultante tem de ser um dag. A razão é simples: um ciclo contendo várias componentes fortemente conexas uniria todas elas em uma única componente fortemente conexa. Reformulando,

Propriedade *Todo grafo direcionado é um dag de suas componentes fortemente conexas.*

Isso nos diz algo importante: a estrutura de conectividade de um grafo direcionado possui dois níveis. No nível superior temos um dag, que é uma estrutura bastante simples — por exemplo, ele pode ser linearizado. Se quisermos detalhes mais finos, podemos observar dentro de um dos nós desse dag e examinar a componente fortemente conexa totalmente desenvolvida.

3.4.2 Um algoritmo eficiente

A decomposição de um grafo direcionado em suas componentes fortemente conexas é muito útil e informativa. Felizmente, ela pode ser encontrada em tempo linear por meio de busca em profundidade. O algoritmo é baseado em algumas propriedades que já vimos, mas que tornaremos mais precisas agora.

Propriedade 1 *Se a sub-rotina explorar é iniciada no nó u , então ela termina precisamente quando todos os nós alcançáveis partindo de u tenham sido visitados.*

Portanto, se chamamos *explorar* em um nó que está em algum lugar de uma componente fortemente conexa sorvedoura (uma componente conexa que é um sorvedouro no metagrafo), então vamos recuperar exatamente aquela componente. A Figura 3.9 possui duas componentes fortemente conexas sorvedouras. Começando *explorar* no nó K , por exemplo, ela percorrerá completamente a maior delas e depois irá parar.

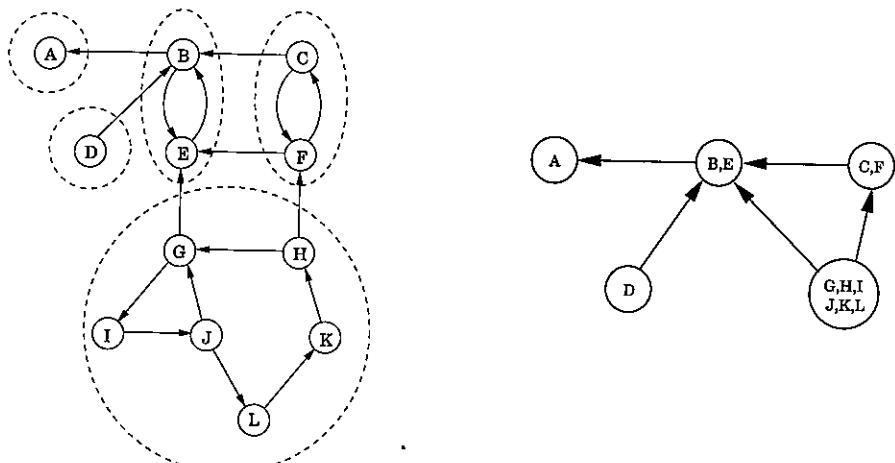
Isso sugere uma maneira de encontrar uma componente fortemente conexa, mas ainda deixa em aberto duas questões importantes: (A) como encontramos um nó que sabemos, com clareza, está em uma componente fortemente conexa sorvedoura e (B) como continuamos, uma vez que essa primeira componente tenha sido descoberta?

Vamos começar com o problema (A). Não há nenhuma maneira fácil, direta, de selecionar um nó que garantidamente esteja em uma componente fortemente conexa sorvedoura. Mas há uma maneira de selecionar um nó que esteja em uma componente fortemente conexa *fonte*.

Propriedade 2 *O nó que recebe o maior número de pós em uma busca em profundidade tem de estar em uma componente fortemente conexa fonte.*

Isso é consequência da seguinte propriedade mais geral.

Figura 3.10 O reverso do grafo da Figura 3.9.



Propriedade 3 Se C e C' são componentes fortemente conexas e existe uma aresta de um nó em C para um nó em C' , então o mais alto número de pós em C é maior do que o mais alto número de pós em C' .

Prova. Para provar a Propriedade 3, existem dois casos a considerar. Se a busca em profundidade visita a componente C antes da componente C' , então claramente C e C' serão totalmente percorridas antes que o procedimento pare (veja a Propriedade 1). Portanto, o primeiro nó visitado em C terá um número de pós maior do que qualquer nó em C' . Por sua vez, se C' é visitada primeiro, então, a busca em profundidade vai parar depois de ver toda C' , mas antes de ver qualquer coisa de C , caso em que a propriedade segue imediatamente. ■

A Propriedade 3 pode ser reformulada dizendo que as *componentes fortemente conexas podem ser linearizadas arranjando-as em ordem decrescente de seus maiores números de pós*. Isso é uma generalização do nosso algoritmo anterior para linearizar dags; em um dag, cada nó é uma componente fortemente conexa de um elemento.

A Propriedade 2 nos ajuda a encontrar um nó na componente fortemente conexa fonte de G . Entretanto, o que precisamos é de um nó na componente *sorvedoura*. Nossas possibilidades parecem ser o oposto de nossas necessidades! Mas considere o grafo *reverso* G^R , o mesmo que G , mas com todas as arestas revertidas (Figura 3.10). G^R tem exatamente as mesmas componentes fortemente conexas de G (por quê?). Assim, se fizermos uma busca em profundidade em G^R , o nó com o maior número de pós virá de uma componente fortemente conexa fonte em G^R , que é uma componente fortemente conexa sorvedoura em G . Resolvemos o problema (A)!

Prossigamos para o problema (B). Como continuamos depois que a primeira componente sorvedouro é identificada? A solução também é fornecida pela Propriedade 3. Uma vez que tenhamos encontrado a primeira componente fortemente conexa e a

Varrendo rapidamente

Tudo o que vimos supõe que o grafo nos é dado pronto, com vértices numerados de 1 a n e arestas organizadas em listas de adjacência. As realidades da World Wide Web são muito diferentes. Os nós do grafo da Web não são conhecidos previamente e eles têm de ser descobertos um por um durante o processo de busca. E, claro, recursão está fora de questão.

Ainda assim, varredura na Web é feita por algoritmos muito similares à busca em profundidade. Uma pilha explícita é mantida, contendo todos os nós que já foram descobertos (como pontos extremos de hiperlinks), mas ainda não explorados. De fato, essa “pilha” não é exatamente uma lista do tipo última-a-entrar, primeiro-a-sair. Ela dá maior prioridade não aos nós que foram inseridos mais recentemente (nem aos que foram inseridos menos recentemente, isso seria *busca em largura*, veja o Capítulo 4), mas àqueles que parecem mais “interessantes” — um critério heurístico cujo propósito é impedir que a pilha estoure e, no pior caso, deixar inexplorados apenas aqueles nós que muito improvavelmente levam a novas vastas regiões.

De fato, a varredura é tipicamente feita por muitos computadores executando *explorar* simultaneamente: cada um toma o próximo nó a ser explorado do topo da pilha, baixa o arquivo http (o tipo de arquivo da Web que aponta uns para os outros) e o percorre procurando por hiperlinks. Mas quando um novo documento http é encontrado no final de um hiperlink, nenhuma chamada recursiva é feita: em vez disso, o novo vértice é inserido na pilha central.

Mas uma questão permanece: quando vemos um “novo” documento, como sabemos se ele é de fato novo, que ainda não o vimos antes na nossa varredura? E como podemos dar a ele um *nome*, para que possa ser inserido na pilha e lembrado como “já visto”? A resposta é, *por espalhamento*.

A propósito, pesquisadores executaram o algoritmo para componentes fortemente conexas na Web e encontraram algumas estruturas muito interessantes.

removido do grafo, o nó com o maior número de pós entre aqueles restantes pertencerá a uma componente fortemente conexa sorvedoura de tudo aquilo que resto de G . Portanto, podemos continuar usando a numeração de pós da nossa busca em profundidade inicial sobre G^R para produzir sucessivamente a segunda componente fortemente conexa, a terceira componente fortemente conexa e assim por diante. O algoritmo resultante é este.

1. Execute busca em profundidade em G^R .
2. Execute o algoritmo para componentes conexas não-direcionadas (da Seção 3.2.3) sobre G e, durante a busca em profundidade, processe os vértices em ordem decrescente de seus números de pós dados pelo passo 1.

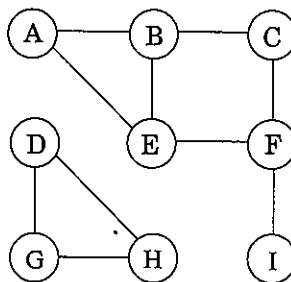
Esse algoritmo é linear, apenas a constante do termo linear é cerca de duas vezes a da busca em profundidade direta. (Questão: como se constrói uma representação por lista de adjacência de G^R em tempo linear? E como, em tempo linear, se pode ordenar os vértices de G por valores decrescentes de pós?)

Vamos executar esse algoritmo sobre o grafo da Figura 3.9. Se o passo 1 considera vértices em ordem lexicográfica, a ordem que ele prepara para o segundo passo (a saber,

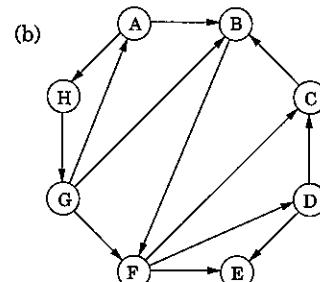
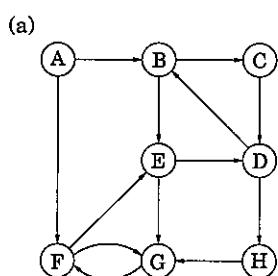
número de pós decrescentes na busca em profundidade de G^R é: $G, I, J, L, K, H, D, C, F, B, E, A$. Então o passo 2 extrai as componentes na seguinte seqüência: $\{G, H, I, J, K, L\}$, $\{D\}$, $\{C, F\}$, $\{B, E\}$, $\{A\}$.

Exercícios

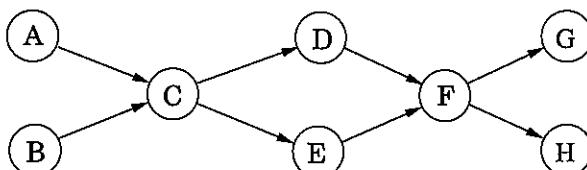
- 3.1. Realize uma busca em profundidade no seguinte grafo; sempre que existir uma escolha entre vértices, selecione aquele que vem primeiro alfabeticamente. Classifique cada aresta como aresta de árvore, ou aresta de retorno e dê os números pré e pós de cada vértice.



- 3.2. Realize uma busca em profundidade em cada um dos seguintes grafos; sempre que existir uma escolha entre vértices, selecione aquele que vem primeiro alfabeticamente. Classifique cada aresta como aresta de árvore, aresta de avanço, aresta de retorno ou aresta cruzada e dê os números pré e pós de cada vértice.

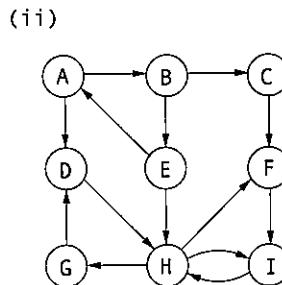
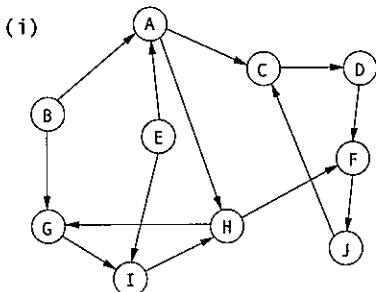


- 3.3. Execute o algoritmo de ordenação topológica baseado em DFS no seguinte grafo. Sempre que você tiver uma escolha entre vértices a explorar, selecione aquele que vem primeiro alfabeticamente.



- (a) Indique os números pré e pós dos nós.

- (b) Quais são as fontes e sorvedouros do grafo?
- (c) Qual ordem topológica é encontrada pelo algoritmo?
- (d) Quantas ordenações topológicas esse grafo possui?
- 3.4. Execute o algoritmo de componentes fortemente conexas nos seguintes grafos direcionados G . Ao fazer DFS em G^R , sempre que houver uma escolha de vértices a explorar, selecione aquele que vem primeiro alfabeticamente.



Em cada caso responda às seguintes questões.

- (a) Em que ordem são encontradas as componentes fortemente conexas (CFCs)?
- (b) Quais são CFCs fontes e quais são CFCs sorvedouras?
- (c) Desenhe o “metagrafo” (cada metanó é uma CFC de G).
- (d) Qual é o número mínimo de arestas que você tem de adicionar para tornar este grafo fortemente conexo?
- 3.5. O *reverso* de um grafo direcionado $G = (V, E)$ é outro grafo direcionado $G^R = (V, E^R)$ sobre o mesmo conjunto de vértices, mas com todas as arestas revertidas; ou seja, $E^R = \{(v, u) : (u, v) \in E\}$.

Forneça um algoritmo de tempo linear para computar o reverso de um grafo no formato de lista de adjacência.

- 3.6. Em um grafo não-direcionado, o *grau* $d(u)$ de um vértice u é o número de vizinhos que u tem, ou equivalentemente, o número de arestas incidentes nele. Em um grafo direcionado, distinguimos entre o *grau de entrada* $d_{in}(u)$, que é o número de arestas que chegam a u e o *grau de saída* $d_{out}(u)$, o número de arestas que saem de u .
- (a) Mostre que em um grafo não-direcionado, $\sum_{u \in V} d(u) = 2|E|$.
- (b) Use a parte (a) para mostrar que em um grafo não-direcionado tem de haver um número par de vértices cujo grau é ímpar.
- (c) Será que uma sentença similar vale para o número de vértices com grau de entrada ímpar em um grafo direcionado?
- 3.7. Um *grafo bipartido* é um grafo $G = (V, E)$ cujos vértices podem ser particionados em dois conjuntos ($V = V_1 \cup V_2$ e $V_1 \cap V_2 = \emptyset$) tal que não existam arestas entre vértices de um mesmo conjunto (por exemplo, se $u, v \in V_1$, então não existe aresta entre u e v).

- (a) Dê um algoritmo de tempo linear para determinar se um grafo não-direcionado é bipartido.
- (b) Há muitas outras maneiras de formular essa propriedade. Por exemplo, um grafo é bipartido se e somente se ele pode ser colorido com apenas duas cores. Prove a seguinte formulação: um grafo não-direcionado é bipartido se e somente se ele não contém nenhum ciclo de tamanho ímpar.
- (c) No máximo quantas cores são necessárias para colorir um grafo não-direcionado com exatamente *um* ciclo de tamanho ímpar?
- 3.8. *Despejando água.* Nós temos três recipientes cujos volumes são 10, 7 e 4 litros, respectivamente. Os recipientes de 7 e 4 litros começam cheios de água, mas o de 10 litros está inicialmente vazio. É permitido apenas um tipo de operação: despejar o conteúdo de um recipiente em outro, parando somente quando o recipiente de origem estiver vazio ou quando o recipiente de destino estiver cheio. Queremos saber se existe uma seqüência de despejos que deixe exatamente 2 litros no recipiente de 7 litros ou no de 4 litros.
- (a) Modele como um problema em grafos: forneça uma definição precisa do grafo envolvido e formule a questão específica sobre este grafo que precisa ser respondida.
- (b) Qual algoritmo deve ser aplicado para resolver este problema?
- 3.9. Para cada nó u em um grafo não-direcionado, seja $\text{segundograu}[u]$ a soma dos graus dos vizinhos de u . Mostre como computar o vetor inteiro de valores de $\text{segundograu}[\cdot]$ em tempo linear, dado um grafo na forma de lista de adjacência.
- 3.10. Reescreva o procedimento *explorar* (Figura 3.3) de modo não-recursivo (isto é, use explicitamente uma pilha). As chamadas para *pré-visita* e *pós-visita* devem ser posicionadas de modo que tenham o mesmo efeito que no procedimento recursivo.
- 3.11. Projete um algoritmo de tempo linear que, dado um grafo não-direcionado G e uma particular aresta e nele, determina se G possui um ciclo contendo e .
- 3.12. Prove ou dê um contra-exemplo: se $\{u, v\}$ é uma aresta em um grafo não-direcionado e , durante uma busca em profundidade pós (u) < pós (v), então v é um ascendente de u na árvore DFS.
- 3.13. *Coneectividade não-direcionada versus direcionada.*
- (a) Prove que em qualquer grafo não-direcionado conexo $G = (V, E)$ existe um vértice $v \in V$ cuja remoção deixa G desconexo. (Dica: considere a árvore de busca DFS para G .)
- (b) Dê um exemplo de um grafo direcionado fortemente conexo $G = (V, E)$ tal que, para todo $v \in V$, removendo v de G deixa um grafo direcionado que não é fortemente conexo.
- (c) Em um grafo não-direcionado com 2 componentes conexas é sempre possível tornar o grafo conexo adicionando uma única aresta. Dê um exemplo de um grafo direcionado com duas componentes fortemente conexas tal que nenhuma adição de uma aresta pode tornar o grafo fortemente conexo.

- 3.14. O capítulo sugere um algoritmo alternativo para linearização (ordenação topológica) que sucessivamente remove nós-fonte do grafo (página 90). Mostre que esse algoritmo pode ser implementado em tempo linear.
- 3.15. O departamento de polícia na cidade de Computópolis tornou todas as ruas de mão única. A prefeita afirma que ainda existe uma maneira de dirigir legalmente de qualquer interseção na cidade para qualquer outra interseção, mas a oposição não está convencida. Um programa de computador é necessário para determinar se a prefeita está correta. Entretanto, a eleição na cidade está se aproximando e há tempo suficiente apenas para executar um algoritmo *de tempo linear*.
- Formule este problema com a teoria de grafos e explique por que ele pode ser de fato solucionado em tempo linear.
 - Suponha agora que a afirmação original da prefeita seja falsa. Ela em seguida afirma algo mais fraco: se você começa dirigindo da prefeitura, navegando ruas de mão única, então não importa onde você chegue, sempre existe uma maneira de dirigir legalmente de volta à prefeitura. Formule essa propriedade mais fraca como um problema em grafos e cuidadosamente mostre como ela também pode ser checada em tempo linear.
- 3.16. Suponha que uma grade curricular de computação consista em n disciplinas, todas elas obrigatórias. O grafo de pré-requisito G tem um nó para cada disciplina e uma aresta da disciplina v para a w se e somente se v é um pré-requisito para w . Encontre um algoritmo que funcione diretamente com essa representação de grafo e compute o número mínimo de semestre necessário para completar o curso (suponha que um estudante pode fazer qualquer número de disciplinas em um semestre). O tempo de execução de seu algoritmo deve ser linear.
- 3.17. *Caminhos infinitos.* Seja $G = (V, E)$ um grafo direcionado com um “vértice inicial” designado, $s \in V$, um conjunto $V_G \subseteq V$ de vértices “bons” e um conjunto $V_B \subseteq V$ de vértices “ruins”. Uma trajetória infinita p de G é uma seqüência infinita $v_0v_1v_2\dots$ de vértices $v_i \in V$ tal que (1) $v_0 = s$ e (2) para todo $i \geq 0$, $(v_i, v_{i+1}) \in E$. Quer dizer, p é um caminho infinito em G começando no vértice s . Como o conjunto V de vértices é finito, toda trajetória infinita de G tem de visitar alguns vértices infinitas vezes.
- Se p é uma trajetória infinita, seja $Inf(p) \subseteq V$ o conjunto de vértices que ocorrem infinitas vezes em p . Mostre que $Inf(p) \subseteq V$ é um subconjunto de uma componente fortemente conexa de G .
 - Descreva um algoritmo que determina se G possui uma trajetória infinita.
 - Descreva um algoritmo que determina se G tem uma trajetória infinita que visita algum vértice bom de V_G infinitas vezes.
 - Descreva um algoritmo que determina se G tem uma trajetória infinita que visita algum vértice bom de V_G infinitas vezes, mas não visita nenhum vértice ruim de V_B infinitas vezes.
- 3.18. É dada uma árvore binária $T = (V, E)$ (em formato de lista de adjacência), junto com um nó raiz $r \in V$. Lembre-se de que u é considerado um *ascendente* de v na árvore enraizada, se o caminho de r até v em T passar por u .

Você deseja pré-processar a árvore tal que perguntas da forma “ u é um ascendente de v ?” possam ser respondidas em tempo constante. O pré-processamento deve tomar tempo linear. Como isso pode ser feito?

- 3.19. Como no problema anterior, é dada uma árvore binária $T = (V, E)$ com um nó raiz. Além disso, existe um vetor $x[\cdot]$ com um valor para cada nó em V . Defina um novo vetor $z[\cdot]$ como se segue: para cada $u \in V$,

$$z[u] = \text{o máximo dos valores } x \text{ associados aos descendentes de } u.$$

Dê um algoritmo de tempo linear que calcule o vetor z inteiro.

- 3.20. É dado uma árvore $T = (V, E)$ juntamente com um nó raiz $r \in V$. O *pai* de qualquer nó $v \neq r$, denotado $p(v)$ é definido como o nó adjacente a v no caminho de r até v . Por convenção, $p(r) = r$. Para $k > 1$, defina $p^k(v) = p^{k-1}(p(v))$ e $p^1(v) = p(v)$ (assim $p^k(v)$ é o k -ésimo ascendente de v).

Cada vértice v da árvore tem um rótulo inteiro não-negativo associado $l(v)$. Forneça um algoritmo de tempo linear para atualizar os rótulos de todos os vértices em T de acordo com a seguinte regra: $l_{\text{novo}}(v) = l(p^{l(v)}(v))$.

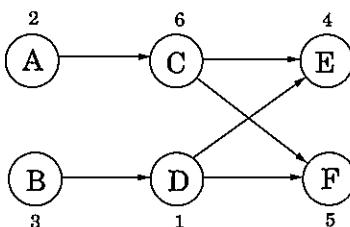
- 3.21. Forneça um algoritmo de tempo linear para encontrar um ciclo de tamanho ímpar em um grafo direcionado. (Dica: Primeiro resolva este problema com a hipótese de que o grafo é fortemente conexo.)
- 3.22. Forneça um algoritmo eficiente que receba como entrada um grafo direcionado $G = (V, E)$ e que determine se existe ou não um vértice $s \in V$ partindo do qual todos os outros vértices são alcançáveis.
- 3.23. Forneça um algoritmo eficiente que tome como entrada um grafo direcionado acíclico $G = (V, E)$ e dois vértices $s, t \in V$ e compute o número de caminhos direcionados diferentes de s até t em G .

- 3.24. Forneça um algoritmo de tempo linear para a seguinte tarefa.

Entrada: Um grafo direcionado acíclico G

Pergunta: G contém um caminho direcionado que passa por cada vértice exatamente uma vez?

- 3.25. É dado um grafo direcionado no qual cada nó $u \in V$ tem um *preço* associado p_u que é um inteiro positivo. Defina o vetor *custo* como se segue: para cada $u \in V$,
- $$\text{custo}[u] = \text{preço do nó mais barato alcançável partindo de } u \text{ (incluindo o próprio } u).$$
- Por exemplo, no grafo a seguir (com preços mostrados para cada vértice), o valor de custo dos nós A, B, C, D, E, F são 2, 1, 4, 1, 4, 5, respectivamente.



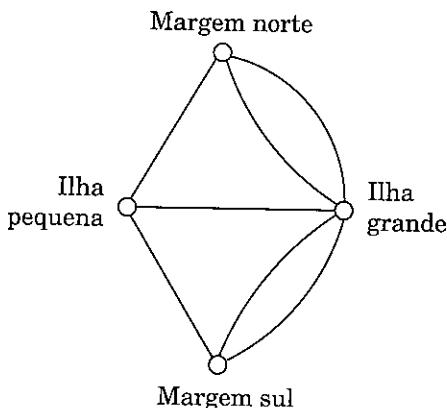
Seu objetivo é projetar um algoritmo que completa o vetor *custo* inteiro (ou seja, para todos os vértices).

- (a) Forneça um algoritmo de tempo linear que funcione para grafos direcionados *acíclicos*. (*Dica:* Manipule os vértices em uma certa *ordem*.)
- (b) Estenda-o para um algoritmo de tempo linear que funcione para todos os grafos direcionados. (*Dica:* Recorde a estrutura de dois níveis de grafos direcionados.)

3.26. Um *círculo euleriano* em um grafo não-direcionado é um ciclo que pode passar por cada vértice múltiplas vezes, mas tem de usar cada aresta exatamente uma vez.

Esse conceito simples foi usado por Euler em 1736 para solucionar o famoso problema das pontes de Konigsberg, dando início à área de teoria de grafos. A cidade de Konigsberg (hoje chamada Kaliningrado, na Rússia ocidental) é o ponto de encontro de dois rios com uma pequena ilha no meio. Existem sete pontes através dos rios e uma questão popular de passatempo daquela época era determinar se é possível realizar um *tour* no qual cada ponte é atravessada *exatamente uma vez*.

Euler formulou a informação relevante como um grafo com quatro nós (denotando áreas de terra) e sete arestas (denotando pontes), como mostrado aqui.



Note uma característica não usual deste problema: múltiplas arestas entre certos pares de nós.

- (a) Mostre que um grafo não-direcionado tem um circuito euleriano se e somente se todos os seus vértices têm grau par. Conclua que não existe um *tour* euleriano para as pontes de Konigsberg.
 - (b) Um *caminho euleriano* é aquele que usa cada aresta exatamente um vez. Você pode dar uma caracterização similar se-e-somente-se sobre quais grafos não-direcionados têm um caminho euleriano?
 - (c) Você pode apresentar uma situação análoga à parte (a) para grafos *direcionados*?
- 3.27. Dois caminhos em um grafo são chamados *aresta-disjuntos* se eles não têm nenhuma aresta em comum. Mostre que, para qualquer grafo não-direcionado, é possível formar pares com todos os vértices de grau ímpar e encontrar caminhos entre cada par tal que todos esses caminhos são aresta-disjuntos.

- 3.28. No problema 2SAT, é dado um conjunto de *cláusulas*, em que cada cláusula é uma disjunção (ou) de dois literais (um literal é uma variável booleana ou a negação de uma variável booleana). Você procura por uma maneira de atribuir um valor verdadeiro ou falso a cada uma das variáveis tal que *todas* as cláusulas sejam satisfeitas — ou seja, exista ao menos um literal verdadeiro em cada cláusula. Por exemplo, aqui está uma instância de 2SAT:

$$(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_1 \vee x_2) \wedge (\bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee x_4).$$

Essa instância possui uma atribuição que satisfaz: atribua a x_1, x_2, x_3 e x_4 , verdadeiro, falso, falso e verdadeiro, respectivamente.

- (a) Existem outras atribuições que satisfazem a fórmula 2SAT? Se afirmativo, encontre todas elas.
- (b) Forneça uma instância de 2SAT com quatro variáveis e que não pode ser satisfeita.

O propósito deste problema é levá-lo a uma forma de resolver o 2SAT eficientemente reduzindo-o ao problema de encontrar as componentes fortemente conexas de um grafo direcionado. Dada uma instância I de 2SAT com n variáveis e m cláusulas, construa um grafo direcionado $G_I = (V, E)$ como se segue.

- G_I tem $2n$ nós, um para cada variável e sua negação.
- G_I tem $2m$ arestas: para cada cláusula $(\alpha \vee \beta)$ de I (onde α, β são literais), G_I tem uma aresta da negação de α para β e uma da negação de β para α .

Note que a cláusula $(\alpha \vee \beta)$ é equivalente a qualquer uma das implicações $\bar{\alpha} \Rightarrow \beta$ ou $\beta \Rightarrow \alpha$. Nesse sentido, G_I registra todas as implicações de I .

- (c) Desenhe esta construção para a instância de 2SAT dada anteriormente e para a instância que você construiu em (b).
- (d) Mostre que se G_I tem uma componente fortemente conexa contendo ambos x e \bar{x} para alguma variável x , então I é insatisfatória.
- (e) Agora mostre o oposto de (d): a saber, que se nenhuma das componentes fortemente conexas de G_I contém tanto um literal quanto sua negação, então a instância I tem de ser satisfatória. (Dica: Atribua valores às variáveis da seguinte forma: sucessivamente selecione uma componente conexa sorvedoura de G_I . Atribua valor verdadeiro a todos os literais no sorvedouro, atribua falso a todas as suas negações e remova todos estes literais. Mostre que isso acaba por descobrir uma atribuição satisfatória.)
- (f) Conclua que existe um algoritmo de tempo linear para resolver 2SAT.

- 3.29. Seja S um conjunto finito. Uma *relação* binária sobre S é simplesmente uma coleção R de pares ordenados $(x, y) \in S \times S$. Por exemplo, S poderia ser um conjunto de pessoas e cada par $(x, y) \in R$ poderia significar “ x conhece y ”.

Uma *relação de equivalência* é uma relação binária que satisfaz três propriedades:

- Reflexiva: $(x, x) \in R$ para todo $x \in S$
- Simétrica: se $(x, y) \in R$, então, $(y, x) \in R$
- Transitiva: se $(x, y) \in R$ e $(y, z) \in R$, então, $(x, z) \in R$

Por exemplo, a relação binária “faz aniversário no mesmo dia que” é uma relação de equivalência, ao passo que “é o pai de” não é, pois ela viola todas as três propriedades.

Mostre que uma relação de equivalência partitiona o conjunto S em grupos disjuntos S_1, S_2, \dots, S_k (em outras palavras, $S = S_1 \cup S_2 \cup \dots \cup S_k$ e $S_i \cup S_j = \emptyset$ para todo $i \neq j$) tal que:

- Quaisquer dois membros de um grupo estão relacionados, isto é, $(x, y) \in R$ para qualquer $x, y \in S_i$, para qualquer i .
- Membros de grupos diferentes não estão relacionados, isto é, para todo $i \neq j$, para todo $x \in S_i$ e $y \in S_j$, temos $(x, y) \notin R$.

(Dica: Represente uma relação de equivalência com um grafo não-direcionado.)

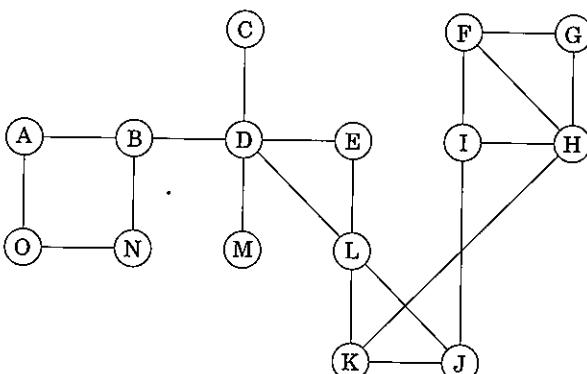
- 3.30. Na página 91, definimos a relação binária “conectados” sobre o conjunto de vértices de um grafo *direcionado*. Mostre que essa é uma relação de equivalência (veja o Exercício 3.29), e conclua que ela partitiona os vértices em componentes fortemente conexas disjuntas.
- 3.31. *Componentes biconexas.* Seja $G = (V, E)$ um grafo não-direcionado. Para quaisquer duas arestas $e, e' \in E$, diremos que $e \sim e'$ se ou $e = e'$ ou existe um ciclo (simples) contendo ambas e e e' .

- (a) Mostre que \sim é uma relação de equivalência (reveja o Exercício 3.29) sobre as arestas.

As classes de equivalência nas quais essa relação partitiona as arestas são chamadas de *componentes biconexas* de G . Uma *ponte* é uma aresta que por si própria é uma componente conexa.

Um *vértice separador* é um vértice cuja remoção desconecta o grafo.

- (b) Particione as arestas do grafo a seguir em componentes biconexas e identifique as pontes e vértices separadores.



Componentes biconexas não somente partitionam as arestas do grafo, elas também *quase* partitionam os vértices no sentido seguinte.

- (c) Associe a cada componente biconexa todos os vértices que são extremidades de suas arestas. Mostre que os vértices correspondentes a duas diferentes

componentes biconexas são ou disjuntos ou interceptam em um único vértice separador.

- (d) Colapse cada componente biconexa em um único metanó e mantenha nós individuais para cada vértice separador. (Desse modo, há arestas entre cada nó-componente e seus vértices separadores.) Mostre que o grafo resultante é uma árvore.

DFS pode ser usado para identificar as componentes biconexas, pontes e vértices separadores de um grafo em tempo linear.

- (e) Mostre que a raiz de uma árvore de DFS é um vértice separador se e somente se ele tem mais de um filho na árvore.
 (f) Mostre que um vértice v não-raiz de uma árvore DFS é um vértice separador se e somente se ele tem um filho v' cujos descendentes (incluindo ele próprio) não tenham nenhuma aresta de retorno para um ascendente próprio de v .
 (g) Para cada vértice u defina:

$$\text{baixo}(u) = \min \begin{cases} \text{pré}(u) \\ \text{pré}(w) \text{ onde } (v, w) \text{ é uma aresta de retorno para} \\ \text{algum descendente } v \text{ de } u \end{cases}$$

Mostre que o vetor inteiro de valores baixo pode ser computado em tempo linear.

- (h) Mostre como computar todos os vértices separadores, pontes e componentes biconexas de um grafo em tempo linear. (Dica: Use os valores de baixo para identificar vértices separadores e execute outra DFS com uma pilha extra de arestas para remover componentes biconexas uma por vez.)

Capítulo 4

Caminhos em grafos

4.1 Distâncias

Busca em profundidade identifica prontamente todos os vértices de um grafo que podem ser alcançados partindo de um ponto inicial designado. Ela também encontra caminhos específicos para esses vértices, resumidos na sua árvore de busca (Figura 4.1). Entretanto, os caminhos podem não ser os mais econômicos possíveis. Na figura, o vértice C é alcançável a partir de S percorrendo apenas uma aresta, enquanto a árvore de DFS mostra um caminho de tamanho 3. Este capítulo é sobre algoritmos para encontrar *caminhos mais próximos* em grafos.

Tamanho de caminhos nos permite falar quantitativamente sobre a extensão segundo a qual diferentes vértices de um grafo estão separados uns dos outros:

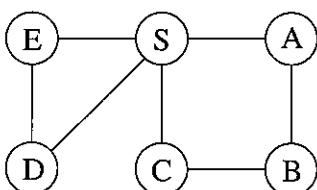
A distância entre dois nós é o tamanho do caminho mais próximo entre eles.

Para um entendimento concreto desse conceito, considere a representação física de um grafo que tenha uma bola para cada vértice e um trecho de linha para cada aresta. Se você elevar a bola do vértice s alto o suficiente, as outras bolas que se elevam junto são precisamente os vértices alcançáveis de s . E para encontrar suas distâncias de s , você precisa apenas medir quão distantes de s elas estão penduradas.

Na Figura 4.2, por exemplo, o vértice B está a uma distância 2 de S e existem dois caminhos mais próximos para ele. Quando S é elevado, as linhas ao longo de cada um desses caminhos se esticam.

Figura 4.1 (a) Um grafo simples e (b) sua árvore de busca em profundidade.

(a)



(b)

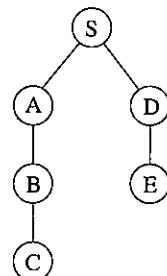
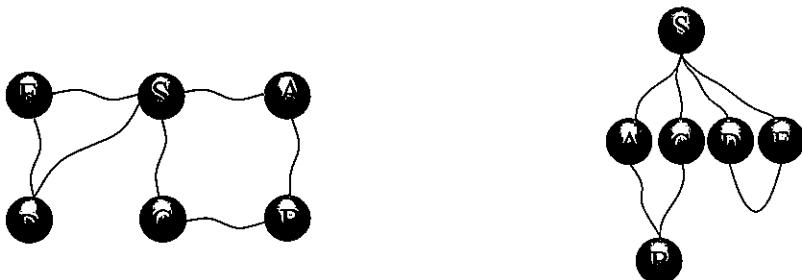


Figura 4.2 Um modelo físico de um grafo.



Por outro lado, a aresta (D, E) não tem nenhum papel em qualquer caminho mais próximo e, portanto, permanece frouxa.

4.2 Busca em largura

Na Figura 4.2, a elevação de s partitiona o grafo em camadas: o próprio s , os nós à distância 1 dele, os nós à distância 2 dele e assim por diante. Uma maneira conveniente de computar distâncias partindo de s para os outros vértices é realizar camada por camada. Uma vez que tenhamos selecionado os nós à distância $0, 1, 2, \dots, d$, os que estão à distância $d + 1$ podem ser facilmente determinados: eles são precisamente os nós ainda não vistos, adjacentes à camada à distância d . Isso sugere um algoritmo iterativo no qual duas camadas estão ativas em qualquer dado momento: alguma camada d , completamente identificada, e $d + 1$, que está sendo descoberta percorrendo-se os vizinhos da camada d .

Busca em largura (BFS, do inglês *breadth-first search*) implementa diretamente este raciocínio simples (Figura 4.3). Inicialmente a fila Q consiste somente em s , o nó à distância zero. E, para cada distância subsequente $d = 1, 2, 3, \dots$, existe um ponto no tempo no qual Q contém todos os nós à distância d e nada mais. À medida que esses nós são processados (ejetados da frente da fila), seus vizinhos ainda não vistos são injetados no fim da fila.

Vamos testar esse algoritmo no exemplo anterior (Figura 4.1) para confirmar se ele faz a coisa certa. Se S é o ponto de partida e os nós são ordenados alfabeticamente, eles são visitados na seqüência mostrada na Figura 4.4. A árvore da busca em largura, na parte direita, contém as arestas pelas quais cada nó é inicialmente descoberto. Ao contrário da árvore DFS que vimos antes, ela tem a propriedade de que todos os seus caminhos a partir de S são os menores possíveis. Ela é, portanto, uma *árvore de caminho mínimo*.

Correção e eficiência

Desenvolvemos o conceito básico da busca por largura. Para checarmos que o algoritmo funciona corretamente, precisamos nos assegurar de que ele execute fielmente esse conceito. O que esperamos, precisamente, é que

Para cada $d = 0, 1, 2, \dots$ existe um momento no qual (1) todos os nós à distância $\leq d$ partindo de s têm suas distâncias corretamente computadas; (2) todos os outros nós têm suas distâncias com valor ∞ ; e (3) a fila contém exatamente os nós à distância d .

Figura 4.3 Busca em largura.

```
procedimento bfs( $G, s$ )
```

Entrada: Grafo $G = (V, E)$, direcionado ou não-direcionado;
vértice $s \in V$

Saída: Para todos os vértices u alcançáveis partindo de s ,
à $\text{dist}(u)$ é atribuída a distância de s até u .

para todo $u \in V$:

$\text{dist}(u) = \infty$

$\text{dist}(s) = 0$

$Q = [s]$ (fila contendo apenas s)

enquanto Q não está vazia:

$u = \text{ejetar}(Q)$

para todas as arestas $(u, v) \in E$:

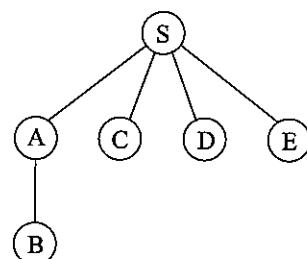
se $\text{dist}(v) = \infty$:

 injetar(Q, v)

$\text{dist}(v) = \text{dist}(u) + 1$

Figura 4.4 O resultado da busca em largura sobre o grafo da Figura 4.1.

Ordem de visitação	Conteúdo da fila depois de processar o nó
	[S]
S	[$A \ C \ D \ E$]
A	[$C \ D \ E \ B$]
C	[$D \ E \ B$]
D	[$E \ B$]
E	[B]
B	[]



Isso foi expresso com um argumento indutivo em mente. Nós já discutimos tanto o caso-base quanto o passo indutivo. Você pode completar os detalhes?

O tempo de execução total desse algoritmo é linear, $O(|V| + |E|)$, exatamente pelas mesmas razões da busca em profundidade. Cada vértice é colocado na fila exatamente uma vez, quando ele é encontrado pela primeira vez, portanto existem $2|V|$ operações na fila. O restante do trabalho é feito no loop mais interno do algoritmo. Durante o curso da execução, o loop visualiza cada aresta uma vez (em grafos direcionados) ou duas vezes (em grafos não-direcionados) e, portanto, toma tempo $O(|E|)$.

Agora que vimos tanto BFS quanto DFS, como os estilos de exploração delas se comparam? Busca em profundidade faz incursões profundas no grafo, retrocedendo somente quando não tem mais nós novos a visitar. Essa estratégia vai de encontro

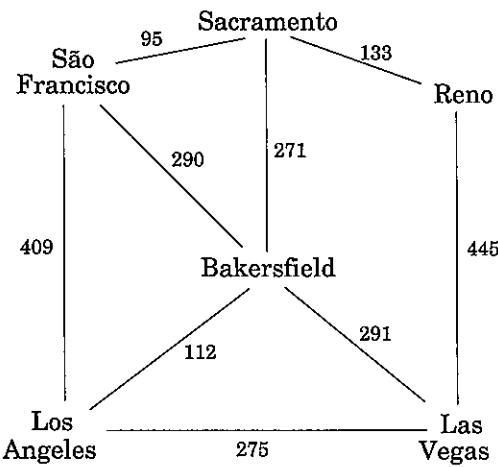
às propriedades espetaculares, sutis e extremamente úteis abordadas no Capítulo 3. Mas isso também significa que a DFS pode terminar tomando uma rota longa e complicada para um vértice que está, na verdade, muito próximo, como na Figura 4.1. Busca em largura se assegura de visitar vértices em ordem crescente de suas distâncias partindo do ponto inicial, ou seja, uma busca mais ampla e superficial, bem parecida com a propagação de ondas na água. E isso é alcançado usando quase exatamente o mesmo código da DFS — mas com uma fila no lugar da pilha.

Também note uma diferença na estilística da DFS: como somente estamos interessados em distâncias partindo de s , não recomeçamos a busca em outras componentes conexas. Nós não alcançáveis partindo de s são simplesmente ignorados.

4.3 Comprimentos nas arestas

Busca em largura trata todas as arestas com o mesmo comprimento. Isso é raramente verdade para aplicações em que caminhos mínimos devem ser encontrados. Por exemplo, suponha que você vá dirigir de São Francisco a Las Vegas e queira saber a rota mais rápida. A Figura 4.5 mostra as principais rodovias que você pode pensar em usar. Selecionar a combinação certa é um problema de caminho mínimo no qual o comprimento de cada aresta (cada trecho de rodovia) é importante. Para o restante deste capítulo, lidaremos com um cenário mais geral, marcando cada aresta $e \in E$ com um comprimento l_e . Se $e = (u, v)$, escreveremos às vezes também $l(u, v)$ ou l_{uv} .

Figura 4.5 Comprimento das arestas freqüentemente tem importância.



Esses l_e não têm de corresponder a comprimentos físicos. Eles poderiam denotar tempo (tempo de estrada entre cidades) ou dinheiro (custo de tomar um ônibus) ou qualquer outra quantidade que desejemos poupar. De fato, há casos em que precisamos usar comprimentos negativos, mas por enquanto ignoraremos essa particular complicação.

4.4 O algoritmo de Dijkstra

4.4.1 Uma adaptação da busca em largura

Busca em largura encontra os caminhos mínimos em qualquer grafo cujas arestas têm comprimento unitário. Será que podemos adaptá-la para um grafo $G = (V, E)$ mais geral cujos comprimentos l_e das arestas são *inteiros positivos*?

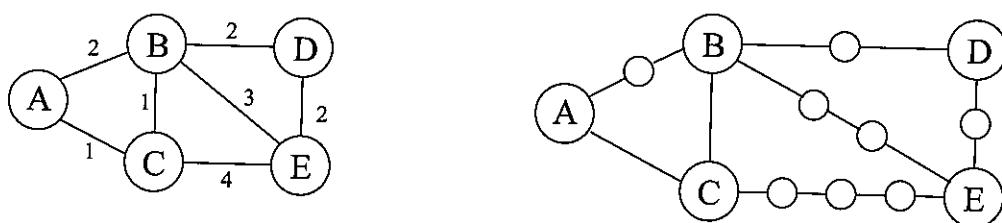
Um grafo mais conveniente

Veja um truque simples para converter G em algo que BFS pode manipular: quebre as arestas longas de G em pedaços de comprimento unitário introduzindo nós “falsos”. A Figura 4.6 mostra um exemplo dessa transformação. Para construir o novo grafo G' ,

Para qualquer aresta $e = (u, v)$ de E , a substitua por l_e arestas de comprimento 1, adicionando $l_e - 1$ nós falsos entre u e v .

O grafo G' contém todos os vértices V que nos interessam, e as distâncias entre eles são exatamente as mesmas em G . Mais importante, as arestas de G' têm todas comprimento unitário. Portanto, podemos computar distâncias em G executando uma BFS em G' .

Figura 4.6 Quebrando arestas em pedaços de comprimento unitário.



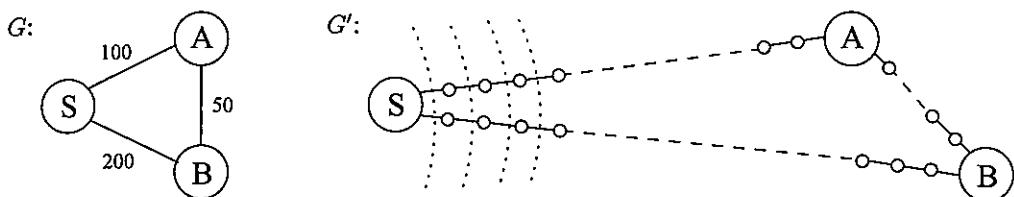
Relógios despertadores

Se eficiência não fosse uma questão, poderíamos parar por aqui. Mas se G tem arestas muito longas, o G' que ele produz é densamente povoado por nós falsos, e a BFS gasta a maior parte de seu tempo computando distâncias para esses nós que não nos interessam absolutamente.

Para melhor entender, considere os grafos G e G' da Figura 4.7 e imagine que a BFS, iniciada no nó s de G' , avança a uma unidade de distância por minuto. Pelos primeiros 99 minutos ela tediosamente progride ao longo de $S - A$ e $S - B$, um deserto sem fim de nós falsos. Será que existe uma maneira de cochilar durante essas fases entediantes e ter um alarme que nos desperte sempre que algo *interessante* está acontecendo — especificamente, sempre que um dos nós reais (do grafo original G) seja alcançado?

Fazemos isso preparando dois alarmes no início, um para o nó A , para despertar no tempo $T = 100$ e um para B , no tempo $T = 200$. São *tempos estimados de chegada*, baseados nas arestas percorridas no momento. Cochilamos e acordamos em $T = 100$ para ver que A foi descoberto. Nesse ponto, o tempo estimado de chegada para B é ajustado para $T = 150$ e trocamos seu alarme de acordo.

Figura 4.7 BFS sobre G' é em maior parte sem evento algum. As linhas pontilhadas mostram as “frentes de onda” iniciais.



Mas em geral, em qualquer dado momento, a busca em largura está avançando ao longo de certas arestas de G e existe um alarme para cada nó de extremidade para onde ela está se movendo, preparado para disparar no tempo de chegada estimado àquele nó. Alguns desses podem ser superestimativas, porque a BFS pode encontrar atalhos mais tarde, como resultado de chegadas em outros lugares. No exemplo anterior, uma rota mais rápida para B foi revelada quando da chegada a A . Entretanto, *nada de interessante pode acontecer antes de um alarme disparar*. O soar do próximo alarme tem de sinalizar a chegada da frente de onda a um nó real $u \in V$ na BFS. Naquele ponto, a BFS pode começar a avançar ao longo de algumas novas arestas que saem de u , e alarmes têm de ser preparados para suas extremidades.

O seguinte “algoritmo do relógio despertador” fielmente simula a execução da BFS sobre G' .

- Prepare um despertador para o nó s no tempo 0.
- Repita até que não haja mais alarmes:

Digamos que o próximo alarme dispare no tempo T , para o nó u . Então:

- A distância de s para u é T .
- Para cada vizinho v de u em G :
 - * Se não existe nenhum alarme ainda para v , prepare um para o tempo $T + l(u, v)$.
 - * Se o alarme de v está preparado para depois de $T + l(u, v)$, então ajuste-o para esse tempo mais cedo.

Algoritmo de Dijkstra

O algoritmo do relógio despertador computa distâncias em qualquer grafo com comprimentos de arestas inteiros positivos. Ele está quase pronto para o uso, exceto que precisamos de alguma maneira para implementar o sistema de alarmes. A estrutura de dados correta para este trabalho é uma *fila de prioridades* (em geral implementada via um *heap*), que mantém um conjunto de elementos (nós) com valores numéricos de chave associados (tempos de alarme) e suporta as seguintes operações:

Inserir. Adiciona um novo elemento ao conjunto.

Diminuir-chave. Realiza a diminuição do valor da chave de um elemento em particular.¹

¹O termo *diminuir-chave* é padrão, mas é um pouco enganoso: a fila de prioridades tipicamente não muda o valor das chaves. O que o procedimento realmente faz é notificar a fila de que um certo valor de chave sofreu diminuição.

Remover-min. Retorna o elemento com a menor chave e o remove do conjunto.

Construir-fila. Constrói uma fila de prioridades dos elementos dados com seus valores de chave. (Em muitas implementações, isto é significativamente mais rápido do que inserir os elementos um por um.)

As duas primeiras nos permitem preparar alarmes e a terceira nos diz qual alarme é o próximo a disparar. Juntando tudo isso, obtemos o algoritmo de Dijkstra (Figura 4.8).

Figura 4.8 Algoritmo de Dijkstra para caminho mais próximo.

procedimento dijkstra(G, l, s)

Entrada: Grafo $G = (V, E)$, direcionado ou não-direcionado;
comprimentos de aresta positivos $\{l_e : e \in E\}$; vértice $s \in V$

Saída: Para todos os vértices u alcançáveis a partir de s , é atribuído a $\text{dist}(u)$ a distância de s até u .

para todo $u \in V$:

$\text{dist}(u) = \infty$

$\text{prev}(u) = \text{null}$

$\text{dist}(s) = 0$

$H = \text{construir-fila}(V)$ (usando valores de dist como chave)
enquanto H é não-vazio:

$u = \text{remover-min}(H)$

 para todas as arestas $(u, v) \in E$:

 se $\text{dist}(v) > \text{dist}(u) + l(u, v)$:

$\text{dist}(v) = \text{dist}(u) + l(u, v)$

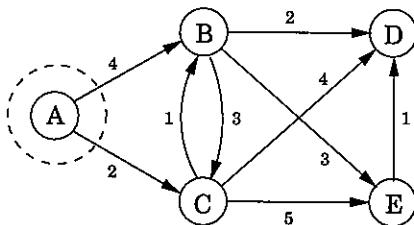
$\text{prev}(v) = u$

 diminuir-chave(H, v)

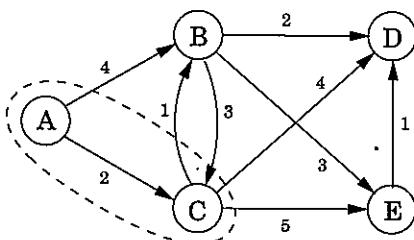
No código, $\text{dist}(u)$ refere-se ao alarme corrente para o nó u . Um valor de ∞ significa que o alarme ainda não foi preparado. Há também um vetor especial, prev , que guarda uma peça de informação crucial para cada nó u : a identidade do nó que está imediatamente antes dele no menor caminho de s até u . Seguindo os ponteiros de retorno, podemos facilmente reconstruir os caminhos mínimos, e, portanto, este vetor é um sumário compacto de todos os caminhos encontrados. Um exemplo completo da operação do algoritmo, juntamente com a árvore de caminhos mínimos final, é apresentado na Figura 4.9.

Resumindo, podemos pensar no algoritmo de Dijkstra simplesmente como BFS, exceto que ele usa uma fila de prioridades em vez de uma fila regular, para priorizar os nós para levar o comprimento das arestas em conta. Esse ponto de vista permite uma apreciação concreta de como e por que o algoritmo funciona, mas há uma derivação mais

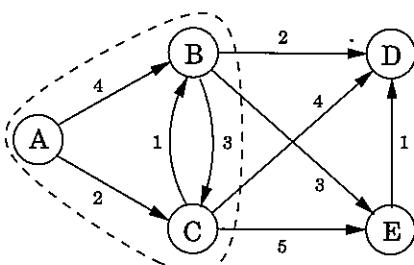
Figura 4.9 Uma execução completa do algoritmo de Dijkstra, com o nó A como ponto de partida. Os valores de $dist$ associados e a árvore final de caminhos mínimos também são apresentados.



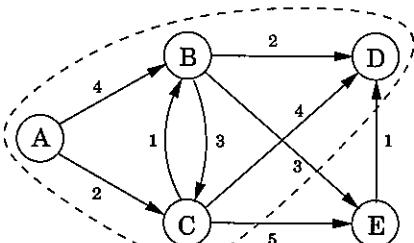
A: 0	D: ∞
B: 4	E: ∞
C: 2	



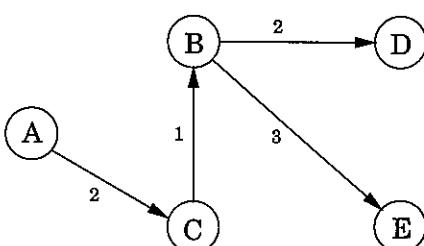
A: 0	D: 6
B: 3	E: 7
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



direta, mais abstrata, que não depende da BFS em absoluto. Comecemos do zero com essa interpretação complementar.

4.4.2 Uma derivação alternativa

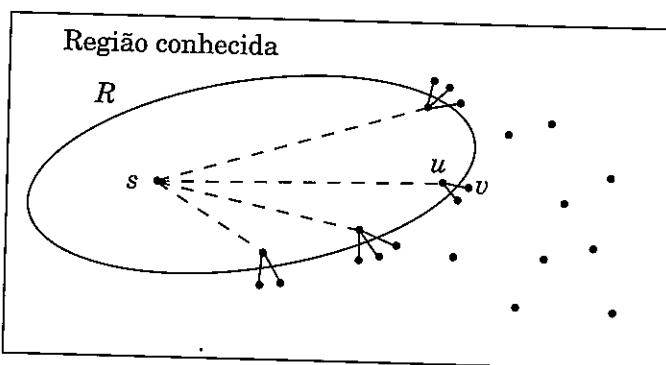
Aqui está um plano para computar caminhos mínimos pode ser definido da seguinte maneira: expandir partindo do ponto inicial s , aumentando constantemente a região do grafo para a qual as distâncias e os caminhos mínimos são conhecidos. O crescimento deve ser ordenado, primeiro incorporando os nós mais próximos e, então, partindo para aqueles mais afastados. Mais precisamente, quando a região conhecida é algum subconjunto de vértices R que inclui s , a próxima inclusão deve ser o nó *fora de R que está mais perto de s* . Vamos chamar este nó de v ; a questão é: como o identificamos?

Para responder, considere u o nó imediatamente anterior a v no caminho mais próximo de s até v :



Como supomos que todos os comprimentos de aresta são positivos, u tem de estar mais próximo de s do que v , ou seja, u está em R — caso contrário isso contradiria o *status* de v como o nó mais próximo de s fora de R . Portanto, o caminho mínimo de s até v é simplesmente um *caminho mínimo já conhecido estendido de uma única aresta*.

Figura 4.10 Extensões de uma única aresta a caminhos mínimos já conhecidos.



Mas tipicamente haverá muitas extensões de uma única aresta para os caminhos mínimos conhecidos (Figura 4.10); qual delas identifica v ? A resposta é: *o menor desses caminhos estendidos*. Porque, se um caminho ainda mais curto existisse, novamente contradiria o *status* de v como o nó fora de R mais próximo de s . Portanto, é fácil encontrar v : ele é o nó fora de R para o qual o menor valor de distância(s, u) + $l(u, v)$ é obtida, à medida que u varia em R . Em outras palavras, *tente todas as extensões de uma única aresta dos caminhos mínimos atualmente conhecidos, encontre o menor caminho estendido e proclame sua extremidade como o próximo nó de R* .

Agora temos um algoritmo para aumentar R visualizando extensões do conjunto atual de caminhos mínimos. Alguma eficiência extra vem da observação de que, em qualquer

dada iteração, as únicas *novas* extensões são aquelas que envolvem o nó mais recentemente adicionado à região R . Todas as demais extensões terão sido avaliadas previamente e não precisam ser recomputadas. No pseudocódigo que se segue, $\text{dist}(v)$ é o comprimento do caminho mínimo atual estendido por uma única aresta, levando a v ; ele é ∞ para vértices não-adjacentes à R .

```

Iniciarizar  $\text{dist}(s)$  como 0, os demais valores de  $\text{dist}(\cdot)$  como  $\infty$ 
 $R = \{\}$  (a "região conhecida")
enquanto  $R \neq V$ :
    Selecionar o nó  $v \notin R$  com o menor  $\text{dist}(\cdot)$ 
    Adicionar  $v$  à  $R$ 
    para todas as arestas  $(v,z) \in E$ :
        se  $\text{dist}(z) > \text{dist}(v) + l(v,z)$ :
             $\text{dist}(z) = \text{dist}(v) + l(v,z)$ 
```

Incorporar operações de fila de prioridades nos fornece novamente o algoritmo de Dijkstra (Figura 4.8).

Para justificarmos esse algoritmo formalmente, usaremos uma prova por indução, como em busca por largura. Veja a seguir uma hipótese de indução apropriada.

No final de cada iteração do loop "enquanto", as seguintes condições são válidas: (1) existe um valor d tal que todos os nós em R estão à distância $\leq d$ de s e todos os nós fora de R estão à distância $\geq d$, e (2) para cada nó u , o valor $\text{dist}(u)$ é o comprimento do menor caminho partindo de s até u cujos nós intermediários estão restritos a R (se não existe um tal caminho, o valor é ∞).

O caso-base é direto (com $d = 0$), e os detalhes do passo de indução podem ser completados por meio da discussão anterior.

4.4.3 Tempo de execução

No nível de abstração da Figura 4.8, o algoritmo de Dijkstra é estruturalmente idêntico à busca em largura. Entretanto, ele é mais lento porque as primitivas da lista de prioridades são computacionalmente mais exigentes do que o `enfileirar` de tempo constante e o `injetar` da BFS. Como `construir-fila` toma no máximo tanto quanto $|V|$ operações de `inserir`, obtemos um total de $|V|$ operações `remover-min` e $|V|+|E|$ operações `inserir/diminuir chave`. O tempo necessário para essas operações varia com a implementação; por exemplo, um heap binário leva a um tempo total de execução de $O((|V|+|E|)\log|V|)$.

4.5 Implementações de fila de prioridades

4.5.1 Vetor

A implementação mais simples de uma fila de prioridades é um vetor não-ordenado de valores de chave para todos os potenciais elementos (os vértices do grafo, no caso do algoritmo de Dijkstra). Inicialmente, esses valores são todos ∞ .

Qual heap é melhor?

O tempo de execução do algoritmo de Dijkstra depende fortemente da implementação usada para a fila de prioridades. Veja as escolhas típicas.

Implementação	<code>remover-min</code>	<code>inserir/diminuir-chave</code>	$ V \times \text{remover-min} + (V + E) \times \text{inserir}$
Vetor	$O(V)$	$O(1)$	$O(V ^2)$
Heap binário	$O(\log V)$	$O(\log V)$	$O((V + E)\log V)$
Heap d -ário	$O\left(\frac{d \log V }{\log d}\right)$	$O\left(\frac{\log V }{\log d}\right)$	$O\left((V \cdot d + E)\frac{\log V }{\log d}\right)$
Heap de Fibonacci	$O(\log V)$	$O(1)$ (amortizado)	$O(V \log V + E)$

Assim, por exemplo, mesmo uma implementação simples com vetor leva a uma complexidade de tempo respeitável de $O(|V|^2)$, ao passo que com um heap binário obtemos $O((|V|+|E|)\log |V|)$. Qual é preferível?

Isso depende de se o grafo é *espars*o (possui poucas arestas) ou *denso* (tem muitas delas). Para todos os grafos, $|E|$ é menos do que $|V|^2$. Se é $\Omega(|V|^2)$, então, claramente a implementação de vetor é a mais rápida. Por sua vez, o heap binário torna-se preferível tão logo $|E|$ cai abaixo de $|V|^2/\log |V|$.

O heap d -ário é uma generalização do heap binário (que corresponde a $d = 2$) e leva a um tempo de execução que é uma função de d . A escolha ótima é $d \approx |E|/|V|$; em outras palavras, para otimizarmos, precisamos fazer o grau do heap ser igual ao *grau médio* do grafo. Isso funciona bem tanto para grafos esparsos quanto para densos. Para grafos muito esparsos, nos quais $|E|=O(|V|)$, o tempo de execução é $O(|V|\log |V|)$, tão bom quanto com um heap binário. Para grafos densos, $|E|=\Omega(|V|^2)$, o tempo de execução é $O(|V|^2)$, tão bom quanto com uma lista ligada. Finalmente, para grafos com densidade intermediária $|E|=|V|^{1+\delta}$, o tempo de execução é $O(|E|)$, linear!

A última linha da tabela fornece os tempos de execução usando uma estrutura de dados sofisticada chamada *heap de Fibonacci*. Muito embora sua eficiência seja impressionante, essa estrutura de dados requer consideravelmente mais trabalho para ser implementada do que as outras, e isso parece diminuir seu apelo na prática. Falaremos pouco sobre ela, exceto mencionar uma característica curiosa de suas cotas de tempo. Sua operação de *inserir* toma tempo variável, mas a *média* é garantidamente $O(1)$ ao longo de uma execução do algoritmo. Em tais situações (uma das quais encontraremos no Capítulo 5) dizemos que o custo *amortizado* da *inserção* no heap é $O(1)$.

Uma *inserir* ou *diminuir-chave* é rápido, porque envolve apenas ajustar o valor da chave, uma operação $O(1)$. Para *remover-min*, por sua vez, é necessário percorrer a lista, em tempo linear.

4.5.2 Heap binário

Neste caso, os elementos são guardados em uma árvore binária *completa*, ou seja, uma árvore binária na qual cada nível é completado da esquerda para a direita e tem

de estar cheio antes que o próximo nível seja iniciado. Além disso, uma restrição especial de ordem é imposta: *o valor da chave de cada nó da árvore é menor ou igual aos dos seus filhos*. Em especial, portanto, a raiz sempre contém o menor elemento. Veja a Figura 4.11(a) para um exemplo.

Para inserir, coloque o novo elemento na parte inferior da árvore (na primeira posição disponível) e, depois, faça-o “ascender”. Quer dizer, se ele é menor do que seu pai, troque um pelo outro e repita (Figura 4.11(b)-(d)). O número de trocas é no máximo a altura da árvore, que é $\lfloor \log_2 n \rfloor$ quando existem n elementos. Para diminuir a chave é similar, exceto que o elemento já está na árvore, portanto fazemos ele ascender de sua posição atual.

Para remover-min, retorne o valor da raiz. Para, então, remover esse elemento do heap, pegue o último nó da árvore (o da posição mais à direita da linha inferior) e o coloque na raiz. Depois, faça-o “descender”: se ele é maior do que algum filho, troque-o com o menor dos filhos e repita (Figura 4.11(e)-(g)). De novo, isso toma tempo $O(\log n)$.

A regularidade de uma árvore binária completa facilita uma representação usando vetor. Os nós da árvore têm uma ordem natural: linha por linha, começando na raiz e movendo da esquerda para a direita em cada linha. Se há n nós, essa ordem especifica suas posições $1, 2, \dots, n$ dentro do vetor. Mover para cima ou para baixo na árvore é facilmente simulado no vetor usando o fato de que o nó de número j tem pai $\lfloor j/2 \rfloor$ e filhos $2j$ e $2j + 1$ (Exercício 4.16).

4.5.3 Heap d -ário

Um heap d -ário é idêntico a um heap binário, exceto que os nós têm d filhos em vez de apenas dois. Isso reduz a altura da árvore com n elementos para $\Theta(\log_d n) = \Theta((\log n)/(\log d))$. Inserções são, portanto, aceleradas por um fator de $\Theta(\log d)$. Operações de remover-min, entretanto, tomam um pouco mais de tempo, ou seja $O(d \log_d n)$ (você percebe o porquê?).

A representação de vetor de um heap binário é facilmente estendida para o caso d -ário. Dessa vez, o nó de número j tem pai $\lceil (j-1)/d \rceil$ e filhos $\{(j-1)d+2, \dots, \min\{n, (j-1)d+d+1\}\}$ (Exercício 4.16).

4.6 Caminhos mínimos na presença de arestas negativas

4.6.1 Arestras negativas

O algoritmo de Dijkstra funciona em parte porque o caminho mínimo partindo de um ponto inicial s até qualquer nó v tem de passar exclusivamente por nós que estão mais próximos do que v . Isso não vale mais quando o comprimento das arestas pode ser negativo. Na Figura 4.12, o caminho mínimo de S até A passa por B , um nó que está mais distante!

O que precisa ser mudado para acomodar essa nova complicação? Para responder a isso, vamos tomar uma visão de alto-nível do algoritmo de Dijkstra. Uma invariante crucial é que os valores de $dist$ que ele mantém são sempre ou superestimados ou exatamente corretos. Eles iniciam em ∞ e a única forma pela qual mudam é na atualização

Figura 4.11 (a) Um heap binário com 10 elementos. Somente os valores das chaves são mostrados. (b)-(d) Os passos intermediários de “ascensão” na inserção de um elemento com chave 7. (e)-(g) Os passos de “descensão” em uma operação de remover-min.

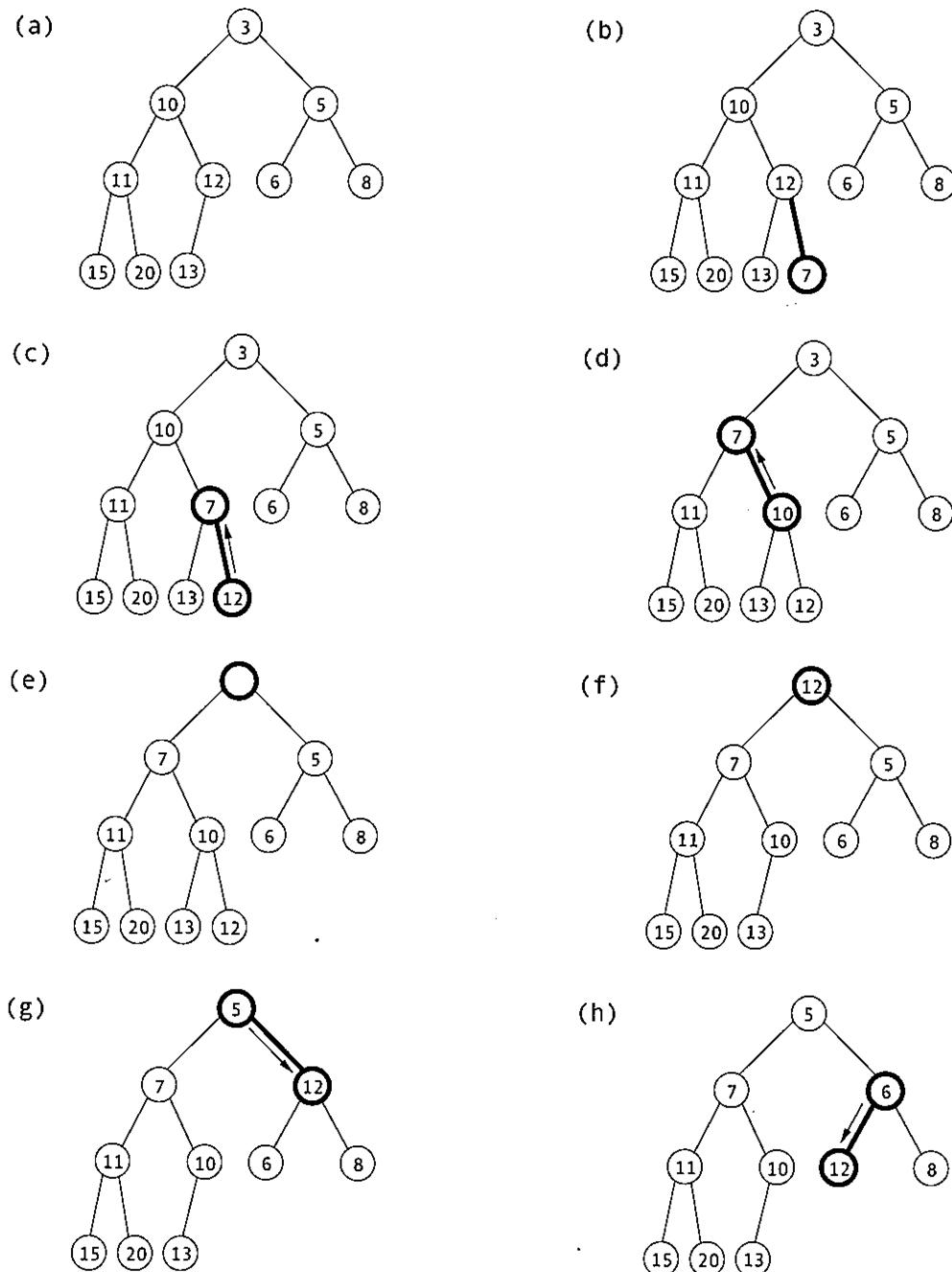
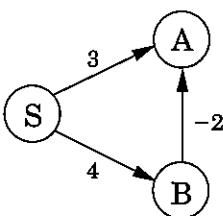


Figura 4.12 O algoritmo de Dijkstra não funcionará se existirem arestas negativas.



ao longo de uma aresta:

```

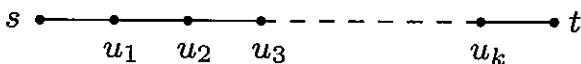
procedimento atualizar(( $u, v) \in E$ )
   $\text{dist}(v) = \min\{\text{dist}(v), \text{dist}(u) + l(u, v)\}$ 

```

A operação de *atualização* é simplesmente uma expressão do fato de que a distância para v não pode ser maior do que a distância para u , mais $l(u, v)$. Ela tem as seguintes propriedades.

1. Ela fornece a distância correta para v no caso particular onde u é o penúltimo nó no caminho mínimo para v e fornece o valor correto a $\text{dist}(u)$.
2. Ela nunca irá fazer $\text{dist}(u)$ pequeno demais e, neste sentido, é *segura*. Por exemplo, uma seqüência de atualizações redundantes não faz mal algum.

Essa operação é extremamente útil: é inofensiva e, usada com cuidado, irá calcular corretamente as distâncias. De fato, o algoritmo de Dijkstra pode ser imaginado simplesmente como uma seqüência de atualizações. Sabemos que essa particular seqüência não funciona com arestas negativas, mas será que existe alguma outra seqüência que funcione? Para ter uma idéia das propriedades que a seqüência tem de possuir, vamos tomar um nó t e examinar o caminho mais curto para ele partindo de s .



Esse caminho pode ter no máximo $|V| - 1$ arestas (você pode ver por quê?). Se a seqüência de atualizações realizadas inclui $(s, u_1), (u_1, u_2), (u_2, u_3), \dots, (u_k, t)$, *nesta ordem* (embora não necessariamente de maneira consecutiva), então pela primeira propriedade a distância para t será corretamente computada. Não importa que outras atualizações ocorram nessas arestas, ou o que acontece no restante do grafo, porque atualizações são *seguras*.

Mas mesmo assim, se não conhecemos todos os caminhos mínimos com antecedência, como podemos com certeza atualizar as arestas corretas na ordem correta? Aqui está uma solução fácil: simplesmente atualize *todas* as arestas, $|V| - 1$ vezes! O procedimento de tempo $O(|V| \cdot |E|)$ resultante é chamado de algoritmo de Bellman-Ford e é apresentado na Figura 4.13, com um exemplo de execução na Figura 4.14.

Figura 4.13 O algoritmo de Bellman-Ford para caminhos mínimos partindo de uma fonte em grafos gerais.

procedimento caminhos-mínimos(G, l, s)

Entrada: Grafo direcionado $G = (V, E)$;

comprimentos de arestas $\{l_e : e \in E\}$ sem ciclos negativos; vértice $s \in V$

Saída: Para todos os vértices u alcançáveis partindo de s , é atribuída a $\text{dist}(u)$ a distância de s até u .

para todo $u \in V$:

$\text{dist}(u) = \infty$

$\text{prev}(u) = \text{null}$

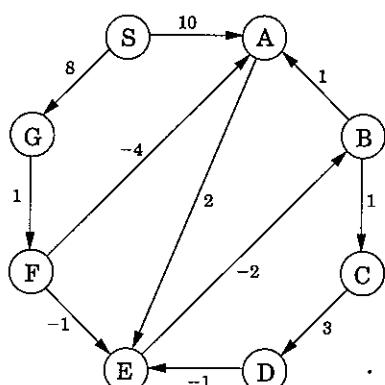
$\text{dist}(s) = 0$

repita $|V|-1$ vezes:

para toda $e \in E$:

atualizar(e)

Figura 4.14 O algoritmo de Bellman-Ford ilustrado em um grafo exemplo.



Nº	Iteração							
	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Uma nota sobre implementação: para muitos grafos, o número máximo de arestas em qualquer caminho mínimo é substancialmente menor do que $|V|-1$, com o resultado de que poucas rodadas de atualizações são necessárias. Portanto, faz sentido adicionar uma verificação extra ao algoritmo de caminhos mínimos, fazê-lo terminar imediatamente depois de qualquer rodada na qual nenhuma atualização ocorreu.

4.6.2 Ciclos negativos

Se o comprimento da aresta (E, B) na Figura 4.14 fosse trocado para -4 , o grafo teria um *ciclo negativo* $A \rightarrow E \rightarrow B \rightarrow A$. Em tais situações, não faz sentido sequer perguntar

por caminhos mínimos. Existe um caminho de comprimento 2 de A até E . Mas, percorrendo o ciclo, há também um caminho de comprimento 1 e, percorrendo múltiplas vezes, encontramos caminhos de comprimentos 0, -1, -2 e assim por diante.

O problema do caminho mínimo não pode ser bem-definido para grafos com ciclos negativos. Como esperado, nosso algoritmo da Seção 4.6.1 funciona apenas na ausência de tais ciclos. Mas onde essa hipótese apareceu na derivação do algoritmo? Bem, ela entrou sorrateiramente quando afirmamos a *existência* de um caminho mínimo de s até t .

Felizmente, é fácil detectar automaticamente ciclos negativos e dar um alerta. Um ciclo desse tipo possibilitaria a aplicação de infinitas rodadas de operações de atualização, reduzindo as estimativas de dist toda vez. Portanto, em vez de parar após $|V|-1$ iterações, realize uma rodada extra. Existirá um ciclo negativo se e somente se algum valor de dist for reduzido durante a rodada final.

4.7 Caminhos mínimos em dags

Existem duas subclasses de grafos que automaticamente excluem a possibilidade de ciclos negativos: grafos sem arestas negativas e grafos sem ciclos. Já sabemos como manusear eficientemente a primeira. Veremos agora como o problema de caminhos mínimos partindo de uma fonte pode ser resolvido em tempo apenas linear em grafos direcionados acíclicos.

Como antes, precisamos realizar uma seqüência de atualizações que inclua todos os caminhos mínimos como subseqüência. A fonte de eficiência é o fato de que

Em qualquer caminho de um dag, os vértices aparecem em ordem linearizada crescente.

Figura 4.15 Um algoritmo para caminhos mínimos partindo de uma fonte para grafos direcionados acíclicos.

procedimento caminho-mínimo-dag(G, l, s)

Entrada: Dag $G = (V, E)$;

comprimentos de aresta $\{l_e : e \in E\}$; vértice $s \in V$

Saída: Para todos os vértices u alcançáveis partindo de s , é atribuída a $\text{dist}(u)$ a distância de s até u .

para todo $u \in V$:

$\text{dist}(u) = \infty$

$\text{prev}(u) = \text{null}$

$\text{dist}(s) = 0$

Linearizar G

para cada $u \in V$, na ordem linearizada:

para toda $(u, v) \in E$:

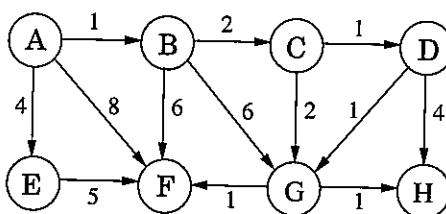
atualizar(u, v)

Portanto, é suficiente linearizar (ordenar topologicamente) o dag por busca em profundidade e, então, visitar os vértices na ordem, atualizando as arestas que saem de cada um. O algoritmo é dado na Figura 4.15.

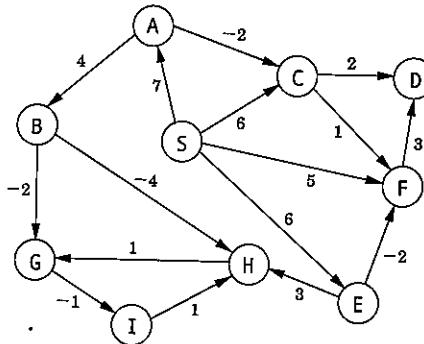
Note que nosso esquema não requer que as arestas sejam positivas. Em particular, podemos encontrar caminhos *mais longos* em um dag usando o mesmo algoritmo: apenas negue todos os comprimentos de arestas.

Exercícios

- 4.1. Suponha que o algoritmo de Dijkstra seja executado sobre o seguinte grafo, começando pelo nó A.



- (a) Desenhe uma tabela mostrando os valores intermediários de distância para todos os nós em cada iteração do algoritmo.
 (b) Mostre a árvore final de caminhos mínimos.
- 4.2. Faça o mesmo que no problema anterior, mas desta vez com o algoritmo de Bellman-Ford.



- 4.3. *Quadrados*. Projete e analise um algoritmo que tome como entrada um grafo não-direcionado $G = (V, E)$ e determine se G contém um ciclo simples (isto é, um ciclo que não intercepta a si mesmo) de comprimento quatro. O tempo de execução deve ser no máximo $O(|V|^3)$.

Você pode supor que o grafo de entrada seja representado tanto por uma matriz de adjacência quanto por listas de adjacência, o que quer que torne seu algoritmo mais simples.

- 4.4. Aqui está uma proposta para encontrar o comprimento do menor ciclo em um grafo não-direcionado com comprimentos unitários de arestas.

Quando uma aresta de retorno, digamos (v, w) , é encontrada durante uma busca em profundidade, ela forma um ciclo com as arestas de árvore de w para v . O comprimento

do ciclo é $\text{nível}[v] - \text{nível}[w] + 1$, onde o nível de um vértice é sua distância na árvore de DFS partindo do vértice raiz. Isso sugere o seguinte algoritmo:

- Faça uma busca em profundidade, anotando o nível de cada vértice.
- A cada vez que uma aresta de retorno é encontrada, compute o comprimento do ciclo e guarde-o se ele for menor do que o menor ciclo visto até então.

Mostre que essa estratégia nem sempre funciona apresentando um contra-exemplo e uma breve explicação (uma ou duas sentenças).

- 4.5. Freqüentemente existem múltiplos caminhos mínimos entre dois nós de um grafo. Forneça um algoritmo de tempo linear para a seguinte tarefa.

Entrada: Grafo não-direcionado $G = (V, E)$ com comprimentos de aresta unitários; nós $u, v \in V$.

Saída: O número de caminhos mínimos distintos de u até v .

- 4.6. Prove que para o vetor prev computado pelo algoritmo de Dijkstra, as arestas $\{u, \text{prev}[u]\}$ (para todo $u \in V$) formam uma árvore.

- 4.7. É dado um grafo direcionado $G = (V, E)$ com pesos (possivelmente negativos) nas arestas, juntamente com um nó específico $s \in V$ e uma árvore $T = (V, E')$, $E' \subseteq E$. Forneça um algoritmo que cheque se T é uma árvore de caminhos mínimos para G com ponto de partida s . Seu algoritmo deve rodar em tempo linear.

- 4.8. O professor F. Lake sugere o seguinte algoritmo para encontrar o caminho mínimo de um nó s para um nó t em um grafo direcionado com algumas arestas negativas: adicione uma mesma constante grande a cada peso de aresta de modo que todos os pesos tornem-se positivos, depois rode o algoritmo de Dijkstra começando no nó s e retorne o caminho mínimo encontrado para o nó t .

Esse método é válido? Prove que ele funciona corretamente ou apresente um contra-exemplo.

- 4.9. Considere um grafo direcionado no qual as únicas arestas negativas são aquelas que saem de s ; todas as outras são positivas. O algoritmo de Dijkstra pode falhar em um tal grafo, começando em s ? Prove sua resposta.

- 4.10. É dado um grafo direcionado com pesos (possivelmente negativos) nas arestas, no qual o caminho mínimo entre quaisquer dois vértices tem, garantidamente, no máximo k arestas. Forneça um algoritmo que encontre o caminho mínimo entre dois vértices u e v em tempo $O(k|E|)$.

- 4.11. Forneça um algoritmo que tome como entrada um grafo direcionado com comprimentos de aresta positivos e retorne o comprimento do menor ciclo no grafo (se o grafo é acíclico, ele o dirá). Seu algoritmo deve tomar tempo no máximo $O(|V|^3)$.

- 4.12. Forneça um algoritmo $O(|V|^2)$ para a seguinte tarefa.

Entrada: Um grafo não-direcionado $G = (V, E)$; comprimentos de aresta $l_e > 0$; uma aresta $e \in E$.

Saída: O comprimento do menor ciclo contendo a aresta e .

- 4.13. É dado um conjunto de cidades, junto com o padrão de rodovias entre elas, na forma de um grafo não-direcionado $G = (V, E)$. Cada trecho de rodovia $e \in E$ conecta duas das cidades e você sabe sua distância em quilômetros, l_e . Você deseja ir

da cidade s para a cidade t . Mas há um problema: o tanque do seu carro pode conter gasolina suficiente para cobrir apenas L quilômetros. Há postos de gasolina em cada cidade, mas não entre as cidades. Portanto, você pode tomar uma rota somente se cada uma de suas arestas tem comprimento $l_e \leq L$.

- (a) Dada a limitação da capacidade do seu tanque, mostre como determinar em tempo linear se existe ou não um caminho factível de s para t .
 - (b) Você agora planeja comprar um carro novo e quer saber a capacidade mínima do tanque que é necessária para viajar de s para t . Forneça um algoritmo de tempo $O((|V|+|E|)\log |V|)$ para determinar isso.
- 4.14. É dado um grafo direcionado fortemente conexo $G = (V, E)$ com pesos positivos nas arestas, juntamente com um particular nó $v_0 \in V$. Forneça um algoritmo eficiente para encontrar caminhos mínimos entre *todos os pares de nós*, com a restrição de que todos estes caminhos têm de passar por v_0 .
- 4.15. Caminhos mínimos não são sempre únicos: às vezes existem dois ou mais caminhos diferentes com o comprimento mínimo possível. Mostre como resolver o seguinte problema em tempo $O((|V|+|E|)\log |V|)$.

Entrada: Um grafo não-direcionado $G = (V, E)$; comprimentos de aresta $l_e > 0$; vértice inicial $s \in V$.

Saída: Um vetor booleano $\text{cmu}[\cdot]$: para cada nó u , a entrada $\text{cmu}[u]$ deve ser verdadeira se e somente se existir um caminho mínimo único de s até u . (*Dica:* $\text{cmu}[s] = \text{verdadeiro}$.)

- 4.16. A Seção 4.5.2 descreve uma maneira de guardar uma árvore binária completa de n nós em um vetor indexado por $1, 2, \dots, n$.

- (a) Considere o nó na posição j do vetor. Mostre que seu pai está na posição $\lfloor j/2 \rfloor$ e que seus filhos estão em $2j$ e $2j + 1$ (se estes números são $\leq n$).
- (b) Quais são os índices correspondentes quando uma árvore d -ária completa é guardada no vetor?

A Figura 4.16 mostra o pseudocódigo para um heap binário, modelado segundo uma exposição de R. E. Tarjan.² O heap é guardado como um vetor h , que suporta por hipótese duas operações de tempo constante:

- $|h|$, que retorna o número de elementos atualmente no vetor;
- h^{-1} , que retorna a posição de um elemento dentro do vetor.

A última operação sempre pode ser alcançada mantendo-se os valores de h^{-1} em um vetor auxiliar.

- (c) Mostre que o procedimento `construir-heap` toma tempo $O(n)$ quando chamado sobre um conjunto de n elementos. Qual a entrada de pior caso? (*Dica:* Comece mostrando que o tempo de execução é no máximo $\sum_{i=1}^n \log(n/i)$.)
- (d) O que precisa ser mudado para adaptar este pseudocódigo para heaps d -ários?

² Veja R. E. Tarjan, *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

Figura 4.16 Operações em um heap binário.

```

procedimento inserir( $h, x$ )
ascender( $h, x, |h|+1$ )

procedimento diminuir-chave( $h, x$ )
ascender( $h, x, h^{-1}(x)$ )

função remover-min( $h$ )
se  $|h| = 0$ :
    retornar null
senão:
     $x = h(1)$ 
    descender( $h, h(|h|), 1$ )
    retornar  $x$ 

função construir-heap( $S$ )
 $h = \text{vetor vazio de tamanho } |S|$ 
para  $x \in S$ :
     $h(|h|+1) = x$ 
para  $i = |S|$  até 1:
    descender( $h, h(i), i$ )
retornar  $h$ 

procedimento ascender( $h, x, i$ )
(colocar elemento  $x$  na posição  $i$  de  $h$  e ascender  $x$ )
 $p = \lceil i/2 \rceil$ 
enquanto  $i \neq 1$  e  $\text{chave}(h(p)) > \text{chave}(x)$ :
     $h(i) = h(p); \quad i = p; \quad p = \lceil i/2 \rceil$ 
 $h(i) = x$ 

procedimento descender( $h, x, i$ )
(colocar elemento  $x$  na posição  $i$  de  $h$  e descender  $x$ )
 $c = \text{menor-filho}(h, i)$ 
enquanto  $c \neq 0$  e  $\text{chave}(h(c)) < \text{chave}(x)$ :
     $h(i) = h(c); \quad i = c; \quad c = \text{menor-filho}(h, i)$ 
 $h(i) = x$ 

função menor-filho( $h, i$ )
(retorna o índice do menor filho de  $h(i)$ )
se  $2i > |h|$  :
    retornar 0 (nenhum filho)
senão:
    retornar o  $j$  que minimiza  $\{\text{chave}(h(j)): 2i \leq j \leq \min\{|h|, 2i+1\}\}$ 

```

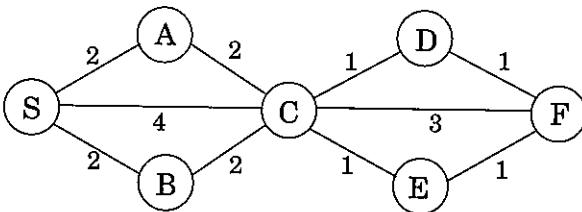
4.17. Suponha que queiramos rodar o algoritmo de Dijkstra sobre o um grafo cujas arestas têm pesos inteiros em um intervalo $0, 1, \dots, W$, onde W é um número relativamente pequeno.

- (a) Mostre como podemos fazer o algoritmo de Dijkstra executar em tempo $O(W|V| + |E|)$.
- (b) Apresente uma implementação alternativa que tome tempo apenas $O(|V| + |E|)\log W$.

4.18. Nos casos em que há vários caminhos mínimos diferentes entre dois nós (e arestas de comprimentos diferentes), o mais conveniente desses caminhos é *frequentemente aquele com o menor número de arestas*. Por exemplo, se nós representam cidades e comprimentos de aresta representam o custo de voar entre as cidades, pode haver muitas maneiras de ir da cidade s para a cidade t , todas com o mesmo custo. A mais conveniente dessas alternativas é aquela que envolve o menor número de escalas. De acordo com isso, para um nó inicial específico s , defina

$$\text{melhor}[u] = \text{número mínimo de arestas em um caminho mínimo de } s \text{ até } u.$$

No exemplo a seguir, os valores melhor para nós S, A, B, C, D, E, F são $0, 1, 1, 1, 2, 2, 3$, respectivamente.



Forneça um algoritmo eficiente para o seguinte problema.

Entrada: Grafo $G = (V, E)$; comprimentos de aresta positivos l_e ; nó inicial $s \in V$.

Saída: Os valores de $\text{melhor}[u]$ calculados para todos os nós $u \in V$.

4.19. *Problema generalizado do caminho mínimo.* No roteamento na Internet, existem atrasos nas linhas, mas também, mais significativamente, atrasos nos roteadores. Isso motiva um problema generalizado de caminhos mínimos.

Suponha que além de ter comprimentos de aresta $\{l_e : e \in E\}$, um grafo possua também custos nos vértices $\{c_v : v \in V\}$. Agora defina o custo de um caminho como a soma dos comprimentos das suas arestas, mais os custos de todos os vértices no caminho (incluindo os extremos). Dê um algoritmo eficiente para o seguinte problema.

Entrada: Um grafo direcionado $G = (V, E)$; comprimentos positivos de arestas l_e e custos positivos de vértices c_v ; um vértice inicial $s \in V$.

Saída: Um vetor $\text{custo}[\cdot]$ tal que para todo vértice u , $\text{custo}[u]$ é o menor custo entre qualquer caminho de s até u (ou seja, o custo do caminho mais barato), segundo a definição anterior.

Note que $\text{custo}[s] = c_s$.

4.20. Existe uma malha de rodovias $G = (V, E)$ conectando um conjunto de cidades V . Cada rodovia em E tem um comprimento associado l_e . Há uma proposta de se adicionar uma

rodovia a esta malha e existe uma lista E' de pares de cidades entre as quais a nova rodovia pode ser construída. Cada tal rodovia potencial $e' \in E'$ possui um comprimento associado. Como projetista do departamento de serviços públicos, você precisa determinar a rodovia $e' \in E'$ cuja adição à malha existente G resultaria no máximo decréscimo no caminho mínimo entre duas cidades fixas s e t na malha. Forneça um algoritmo eficiente para resolver este problema.

- 4.21. Algoritmos para caminhos mínimos podem ser aplicados no mercado de câmbio. Sejam c_1, c_2, \dots, c_n várias moedas; por exemplo, c_1 pode ser dólar, c_2 libra e c_3 real. Para quaisquer duas moedas c_i e c_j , existe uma taxa de câmbio $r_{i,j}$; isso significa que você pode comprar $r_{i,j}$ unidades da moeda c_j em troca de uma unidade de c_i . As taxas de câmbio satisfazem à condição de que $r_{i,j} \cdot r_{j,i} < 1$, de modo que se você começa com uma unidade da moeda c_i , troca pela moeda c_j e, depois, converte de volta para c_i , você termina com menos de uma unidade da moeda c_i (a diferença é o custo da transação).

- (a) Forneça um algoritmo eficiente para o seguinte problema: dado um conjunto de taxas de câmbio $r_{i,j}$ e duas moedas s e t , encontre a seqüência de trocas mais vantajosa para converter da moeda s para a moeda t . Para esse fim, você deve representar as moedas e taxas com um grafo cujos comprimentos de aresta são números reais.

As taxas de troca são atualizadas freqüentemente, refletindo a demanda e oferta das várias moedas. Ocionalmente as taxas de troca satisfazem à seguinte propriedade: existe uma seqüência de moedas $c_{i_1}, c_{i_2}, \dots, c_{i_k}$ tal que $r_{i_1, i_2} \cdot r_{i_2, i_3} \cdots r_{i_{k-1}, i_k} \cdot r_{i_k, i_1} > 1$. Isso significa que começando com uma unidade da moeda c_{i_1} e, então, sucessivamente convertendo para as moedas $c_{i_2}, c_{i_3}, \dots, c_{i_k}$ e finalmente de volta para c_{i_1} , você terminaria com mais de uma unidade da moeda c_{i_1} . Essas anomalias duram somente uma fração de minuto no mercado, mas elas dão a oportunidade para lucros sem riscos.

- (b) Forneça um algoritmo eficiente para detectar a presença de uma anomalia. Use a representação de grafo que você encontrou no item anterior.

- 4.22. *O problema do navio cargueiro.* Você é o dono de uma companhia de navegação que pode transportar cargas entre um grupo de cidades portuárias V . Você lucra em cada porto: uma visita à cidade i gera um lucro de p_i reais. Além disso, o custo de transporte do porto i para o porto j é $c_{ij} > 0$. Você deseja encontrar uma rota cíclica na qual a razão de lucro por custo é maximizada.

Para esse fim, considere um grafo direcionado $G = (V, E)$ cujos nós são portos e que tem arestas entre cada par de portos. Para qualquer ciclo C nesse grafo, a razão lucro-por-custo é

$$r(C) = \frac{\sum_{(i,j) \in C} p_j}{\sum_{(i,j) \in C} c_{ij}}.$$

Seja r^* a razão máxima alcançável por um ciclo simples. Uma maneira de determinar r^* é por busca binária: primeiro descobrindo alguma razão r e, depois, testando se ela é grande demais ou pequena demais.

Considere qualquer constante positiva $r > 0$. Dê a cada aresta (i, j) um peso de $w_{ij} = rc_{ij} - p_j$.

- (a) Mostre que se existe um ciclo de peso negativo, então $r < r^*$.
- (b) Mostre que se todos os ciclos no grafo têm pesos estritamente positivos, então $r > r^*$.
- (c) Forneça um algoritmo eficiente que tome como entrada uma precisão desejada $\epsilon > 0$ e retorne um ciclo simples C para o qual $r(C) \geq r^* - \epsilon$. Justifique a correção do seu algoritmo e analise seu tempo de execução em termos de $|V|$, ϵ , e $R = \max_{(i,j) \in E} (p_j/c_{ij})$.

Capítulo 5

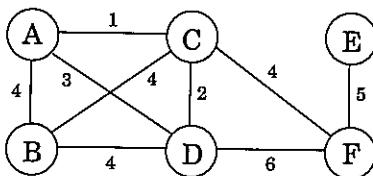
Algoritmos gulosos

Um jogo como xadrez somente pode ser vencido *antecipando-se jogadas*: um jogador concentrado inteiramente na vantagem imediata é fácil de derrotar. Mas em muitos outros jogos, como, por exemplo, Scrabble (o jogo de palavras cruzadas entre vários jogadores), é possível jogar muito bem simplesmente fazendo o movimento que parece melhor no momento, sem se preocupar muito com as consequências futuras.

Esse tipo de comportamento míope é fácil e conveniente, fazendo dele uma estratégia atrativa para algoritmos. Algoritmos *gulosos* constroem uma solução peça por peça, sempre escolhendo a próxima peça que oferece o benefício mais óbvio e imediato. Embora uma abordagem como essa possa ser desastrosa para algumas tarefas computacionais, existem muitas para as quais ela é ótima. Nossa primeiro exemplo é o de árvores espalhadas mínimas.

5.1 Árvores espalhadas mínimas

Suponha que você precise conectar uma coleção de computadores em rede, ligando pares selecionados deles. Isso se traduz em um problema em grafos no qual nós são computadores, arestas não-direcionadas são links em potencial e o objetivo é selecionar um número suficiente dessas arestas para que o grafo seja conexo. Mas não é tudo; cada link também tem um custo de manutenção, refletido no peso daquela aresta. Qual a rede mais barata possível?



Uma observação imediata é que o conjunto ótimo de arestas não pode conter um ciclo, porque ao removermos uma aresta deste ciclo o custo é reduzido sem comprometer a conectividade:

Propriedade 1 *Remover uma aresta de um ciclo não desconecta um grafo.*

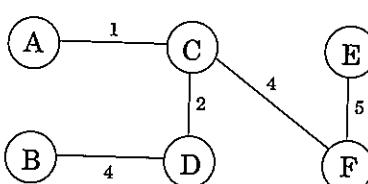
Portanto a solução tem de ser conexa e acíclica: grafos não-direcionados deste tipo são chamados *árvores*. A árvore particular que desejamos é aquela com o peso total mínimo, conhecida como a *árvore espalhada mínima*. Veja sua definição formal.

Entrada: Um grafo não-direcionado $G = (V, E)$; pesos de aresta w_e .

Saída: Uma árvore $T = (V, E')$, com $E' \subseteq E$, que minimiza

$$\text{peso}(T) = \sum_{e \in E'} w_e.$$

No exemplo anterior, a árvore espalhada mínima tem um custo de 16:



Entretanto, essa não é a única solução ótima. Você pode identificar outra?

5.1.1 Uma abordagem gulosa

O algoritmo de Kruskal para árvore espalhada mínima começa com o grafo vazio e, então, seleciona arestas de E de acordo com a seguinte regra.

Repetidamente adicione a próxima aresta mais leve que não produz um ciclo.

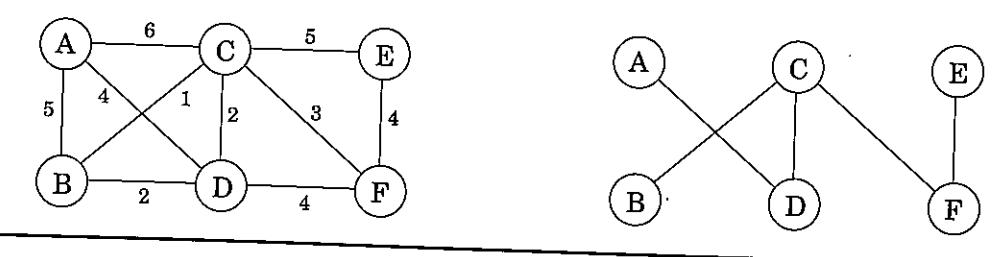
Em outras palavras, ele constrói a árvore aresta por aresta e , além de tomar o cuidado de evitar ciclos, simplesmente seleciona a aresta mais barata no momento. Esse é um algoritmo *guloso*: toda decisão que ele toma é aquela que traz a vantagem imediata mais óbvia.

A Figura 5.1 mostra um exemplo. Começamos com um grafo vazio e, então, tentamos adicionar arestas em ordem crescente de peso (empates são decididos arbitrariamente):

$B - C, C - D, B - D, C - F, D - F, E - F, A - D, A - B, C - E, A - C$.

As duas primeiras passam, mas a terceira, $B - D$, produziria um ciclo se adicionada. Portanto nós a ignoramos e vamos para a frente. O resultado final é uma árvore com custo 14, o mínimo possível.

Figura 5.1 A árvore espalhada mínima encontrada pelo algoritmo de Kruskal.



Árvores

Uma árvore é um grafo não-direcionado que é conexo e acíclico. Muito do que faz árvores tão úteis é a simplicidade da sua estrutura. Por exemplo,

Propriedade 2 *Uma árvore com n nós possui $n - 1$ arestas.*

Isso pode ser visto construindo-se a árvore uma aresta por vez, começando de um grafo vazio. Inicialmente cada um dos n nós está desconectado dos demais. Como cada aresta une duas componentes diferentes, exatamente $n - 1$ arestas são adicionadas até o momento em que a árvore está totalmente conectada.

Com um pouco mais de detalhes: quando uma particular aresta $\{u, v\}$ aparece, podemos estar certos de que u e v estão em componentes conexas separadas, pois, ao contrário, já haveria um caminho entre elas, e esta aresta criaria um ciclo. Adicionar a aresta, então, une as duas componentes, reduzindo com isso o número total de componentes conexas por um. Ao longo desse processo incremental, o número de componentes decresce de n até um, significando que $n - 1$ arestas têm de ter sido adicionadas durante esse caminho.

O oposto também é verdadeiro.

Propriedade 3 *Qualquer grafo não-direcionado, conexo, $G = (V, E)$ com $|E| = |V| - 1$ é uma árvore.*

Precisamos mostrar apenas que G é acíclico. Uma maneira de fazer isso é executar o seguinte procedimento iterativo sobre ele: enquanto o grafo contiver um ciclo, remova uma aresta deste ciclo. O processo termina com algum 'grafo $G' = (V, E')$, $E' \subseteq E$, que é acíclico e, pela Propriedade 1 (da página 127), é também conexo. Portanto G' é uma árvore, e assim $|E'| = |V| - 1$ pela Propriedade 2. Portanto $E' = E$, nenhuma aresta foi removida e G era acíclico desde o começo.

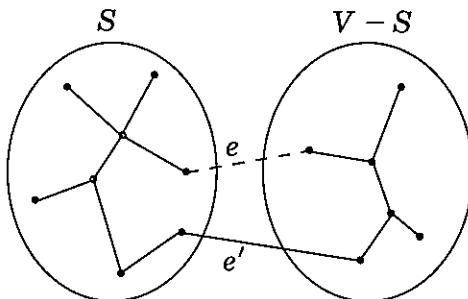
Em outras palavras, podemos dizer se um grafo conexo é uma árvore apenas contando quantas arestas ele tem. Aqui está outra caracterização.

Propriedade 4 *Um grafo não-direcionado é uma árvore se e somente se existe um único caminho entre qualquer par de nós.*

Em uma árvore, quaisquer dois nós podem ter apenas um caminho entre eles; pois se houvesse dois caminhos, a união deles conteria um ciclo.

Por sua vez, se um grafo tem um caminho entre quaisquer dois nós, então ele é conexo. Se esses caminhos são únicos, o grafo é também acíclico (pois um ciclo tem dois caminhos entre qualquer par de nós).

Figura 5.2 $T \cup \{e\}$. A adição de e (pontilhada) a T (linhas contínuas) produz um ciclo. Este ciclo tem de conter pelo menos uma outra aresta, mostrada aqui como e' , através do corte $(S, V - S)$.



A correção do método de Kruskal origina-se de uma certa *propriedade de corte*, que é geral o suficiente para justificar também uma gama inteira de outros algoritmos para árvore espalhada mínima.

5.1.2 A propriedade do corte

Digamos que no processo de construção de uma árvore espalhada mínima (AEM), já pegamos algumas arestas e estamos no caminho certo. Qual aresta devemos adicionar em seguida? O seguinte lema nos dá bastante flexibilidade para nossa escolha.

Propriedade do corte *Suponha que as arestas X sejam parte de uma árvore espalhada mínima de $G = (V, E)$. Selecione qualquer subconjunto de nós S para o qual X não atravessa entre S e $V - S$ e seja e a aresta mais leve através dessa partição. Então $X \cup \{e\}$ é parte de alguma AEM.*

Um *corte* é qualquer partição dos vértices em dois grupos, S e $V - S$. O que essa propriedade diz é que é sempre garantido adicionar a aresta mais leve que atravessa qualquer corte (isto é, entre um vértice de S e um de $V - S$), desde que X não tenha arestas atravessando o corte.

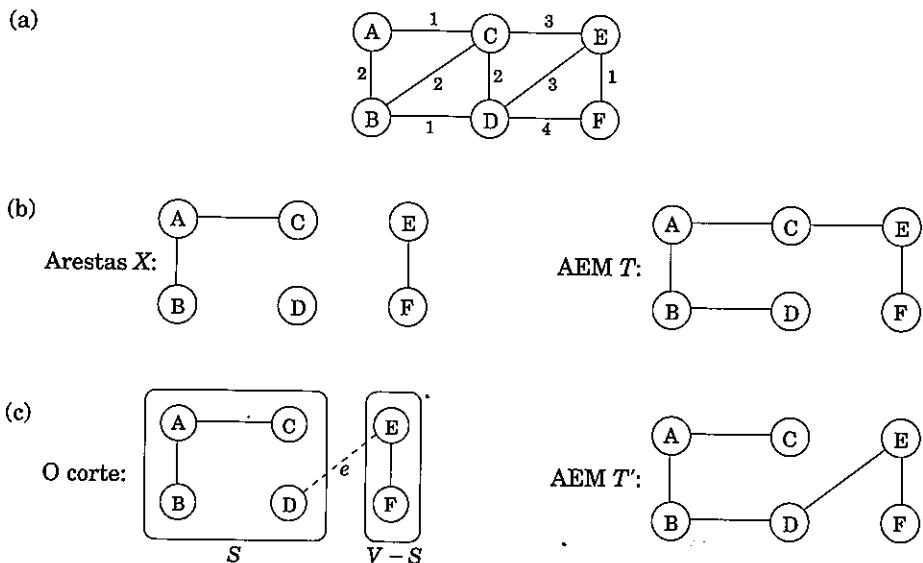
Vejamos por que isso vale. As arestas X são parte de alguma AEM T ; se a nova aresta e também for parte de T , então não há nada a provar. Portanto assuma que e não está em T . Vamos construir uma AEM diferente T' contendo $X \cup \{e\}$ alterando T levemente, mudando apenas uma de suas arestas.

Adicione a aresta e a T . Como T é conexa, ela já tem um caminho entre os extremos de e , portanto adicionar e cria um ciclo. Esse ciclo também tem de ter alguma outra aresta e' através do corte $(S, V - S)$ (Figura 5.2). Se agora removermos essa aresta, restará $T' = T \cup \{e\} - \{e'\}$, que mostraremos ser uma árvore. T' é conexa pela Propriedade 1, pois e' é uma aresta de ciclo. E ela tem o mesmo número de arestas que T ; portanto, pelas Propriedades 2 e 3, ela é também uma árvore.

Além disso, T' é uma árvore espalhada mínima. Compare seu peso com o de T :

$$\text{peso}(T') = \text{peso}(T) + w(e) - w(e').$$

Figura 5.3 A propriedade do corte funcionando. (a) Um grafo não-direcionado. (b) O conjunto X possui três arestas e é parte da AEM T da direita. (c) Se $S = \{A, B, C, D\}$, então uma das arestas de peso mínimo atravessando o corte $(S, V - S)$ é $e = \{D, E\}$. $X \cup \{e\}$ é parte da AEM T' , mostrada à direita.



Ambas e e e' atravessam entre S e $V - S$, e e é especificamente a aresta mais leve deste tipo. Portanto $w(e) \leq w(e')$ e $\text{peso}(T') \leq \text{peso}(T)$. Como T é uma AEM, tem que ser o caso que $\text{peso}(T') = \text{peso}(T)$ e T' é também uma AEM.

A Figura 5.3 apresenta um exemplo da propriedade do corte. Qual aresta é e ?

5.1.3 O algoritmo de Kruskal

Estamos prontos para justificar o algoritmo de Kruskal. A qualquer momento, as arestas que ele já escolheu formam uma solução parcial, uma coleção de componentes conexas cada uma delas com uma estrutura de árvore. A próxima aresta e a ser adicionada conecta duas dessas componentes; vamos chamá-las de T_1 e T_2 . Como e é a aresta mais leve que não produz um ciclo, ela garantidamente é a aresta mais leve entre T_1 e $V - T_1$, e, portanto, satisfaz a propriedade do corte.

Agora completamos alguns detalhes de implementação. Em cada estágio, o algoritmo escolhe uma aresta para adicionar à sua solução parcial atual. Para tanto, ele precisa testar cada aresta candidata $u - v$ para ver se seus extremos u e v estão em componentes diferentes; caso contrário, a aresta produz um ciclo. Assim que uma aresta é escolhida, as correspondentes componentes precisam ser unidas. Que tipo de estrutura de dados suporta tais operações?

Figura 5.4 O algoritmo de Kruskal para árvore espalhada mínima.

```

procedimento kruskal( $G, w$ )
Entrada: Um grafo não-direcionado conexo  $G = (V, E)$  com pesos de
aresta  $w_e$ 
Saída: Uma árvore espalhada mínima definida pelas arestas  $X$ 
para todo  $u \in V$ :
construir-conjunto( $u$ )
 $X = \{\}$ 
ordene as arestas  $E$  por peso
para todas as arestas  $\{u, v\} \in E$ , em ordem crescente de peso:
    se encontrar( $u$ ) ≠ encontrar( $v$ ):
        adicionar aresta  $\{u, v\}$  a  $X$ 
        unir( $u, v$ )

```

Modelaremos o estado do algoritmo como uma coleção de *conjuntos disjuntos*, cada um dos quais contém os nós de uma particular componente. Inicialmente cada nó está em uma componente sozinho:

construir-conjunto(x): cria um conjunto unitário contendo apenas x .

Testamos repetidamente pares de nós para ver se eles pertencem ao mesmo conjunto.

encontrar (x): a qual conjunto x pertence?

E sempre que adicionamos uma aresta, unimos duas componentes.

unir(x, y): une os conjuntos contendo x e y .

O algoritmo final é mostrado na Figura 5.4. Ele usa a operação **construir-conjunto** $|V|$ vezes, **encontrar** $2|E|$ vezes e **unir** $|V| - 1$ vezes.

5.1.4 Uma estrutura de dados para conjuntos disjuntos

União por rank:

Uma maneira de guardar um conjunto é usando uma árvore direcionada (Figura 5.5). Nós da árvore são elementos do conjunto, arranjados sem nenhuma ordem particular, e cada elemento tem um ponteiro para um elemento pai, o que o leva em algum momento até à raiz da árvore. O elemento raiz é um *representante* conveniente, ou um *nome*, para o conjunto. Ele se distingue dos demais pelo fato de que seu ponteiro pai é um loop para si mesmo.

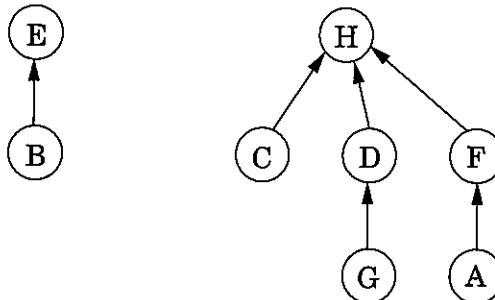
Além de um ponteiro pai π , cada nó tem também um *rank* que, por enquanto, deve ser interpretado como a altura da subárvore pendurada naquele nó.

```

procedimento construir-conjunto( $x$ )
 $\pi(x) = x$ 
rank( $x$ ) = 0
função encontrar( $x$ )
enquanto  $x \neq \pi(x)$ :  $x = \pi(x)$ 
retorna  $x$ 

```

Figura 5.5 Uma representação por árvore direcionada de dois conjuntos $\{B, E\}$ e $\{A, C, D, F, G, H\}$.



Como esperado, construir-conjunto é uma operação de tempo constante. Por sua vez, encontrar segue os ponteiros pai até a raiz da árvore e, portanto, toma tempo proporcional ao tamanho da árvore. A árvore na verdade é construída pela terceira operação unir, e assim precisamos garantir que este procedimento mantenha as árvores curtas.

Unir dois conjuntos é fácil: faça a raiz de uma apontar para a raiz da outra. Mas temos uma escolha aqui. Se os representantes (raízes) dos conjuntos são r_x e r_y , fazemos r_x apontar para r_y ou o contrário? Como a altura da árvore é o maior impedimento para eficiência computacional, uma boa estratégia é *fazer a raiz da árvore mais curta apontar para a raiz da mais alta*. Dessa maneira, a altura total cresce somente se as duas árvores unidas tiverem a mesma altura. Em vez de explicitamente computar alturas das árvores, usaremos os números de *rank* de seus nós raiz — que é a razão pela qual este esquema é chamado *união por rank*.

```

procedimento unir(x,y)
  rx = encontrar(x)
  ry = encontrar(y)
  se rx = ry: retornar
  se rank(rx) > rank(ry):
    π(ry) = rx
  senão:
    π(rx) = ry
    se rank(rx) = rank(ry): rank(ry) = rank(ry) + 1
  
```

Veja a Figura 5.6 para um exemplo.

Por definição, o *rank* de um nó é exatamente a altura da subárvore enraizada naquele nó. Isso significa, por exemplo, que à medida que nos movemos para cima em um caminho na direção do nó raiz, os valores de *rank* são estritamente crescentes.

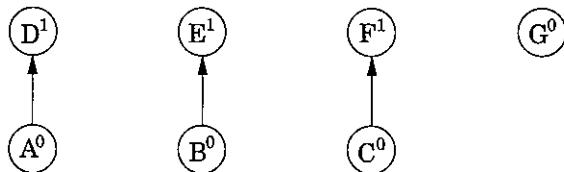
Propriedade 1 *Para todo x, $rank(x) < rank(\pi(x))$.*

Figura 5.6 Uma seqüência de operações em conjuntos disjuntos. Sobrescritos denotam rank.

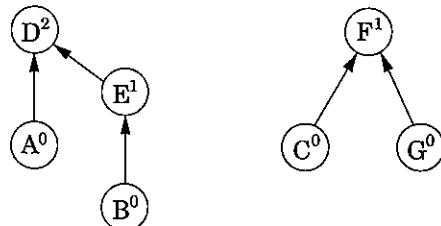
Depois de construir-conjunto(A), construir-conjunto(B),..., construir-conjunto(G):



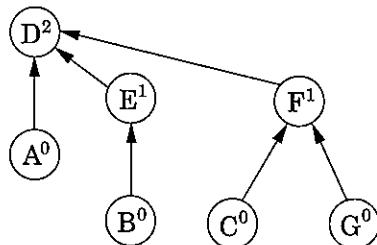
Depois de unir(A, D), unir(B, E), unir(C, F):



Depois de unir(C, G), unir(E, A):



Depois de unir(B, G):



Um nó raiz com rank k é criado pela união de duas árvores com rank na raiz de $k - 1$. Segue por indução (tente fazê-la!) que

Propriedade 2 Qualquer nó raiz de rank k tem pelo menos 2^k nós na sua árvore.

Isso se estende para nós internos (não raiz) também: um nó de rank k tem pelo menos 2^k descendentes. Ademais, qualquer nó foi uma raiz alguma vez e nem seu rank, nem seu número de descendentes mudou desde então. Além disso, nós de rank k dife-

rentes não podem ter descendentes comuns, pois pela Propriedade 1 qualquer elemento tem no máximo um ascendente de rank k . O que significa

Propriedade 3 *Se existem n elementos ao todo, pode haver no máximo $n/2^k$ nós de rank k .*

Essa última observação implica, crucialmente, que o rank máximo é $\log n$. Portanto, todas as árvores têm altura $\leq \log n$, e isso é uma cota superior sobre o tempo de execução de encontrar e unir.

Compressão de caminho:

Com a estrutura de dados apresentada até agora, o tempo total para o algoritmo de Kruskal torna-se $O(|E|\log|V|)$ para ordenar as arestas (lembre-se de que, $\log |E| \approx \log |V|$) mais outro $O(|E|\log|V|)$ para as operações de unir e encontrar que dominam o restante do algoritmo. Portanto parece haver pouco incentivo para fazer nossa estrutura de dados mais eficiente.

Mas e se as arestas nos forem dadas em ordem? Ou se os pesos são pequenos (digamos, $O(|E|)$) de modo que a ordenação possa ser feita em tempo linear? Então a parte da estrutura de dados torna-se o gargalo e vale a pena pensar em melhorar sua *performance* para algo melhor que $\log n$ por operação. Como se observa, a estrutura de dados aperfeiçoada é útil em muitas outras aplicações.

Mas como podemos realizar unir e encontrar mais rápido do que $\log n$? A resposta é: sendo um pouco mais cuidadosos para manter nossa estrutura de dados em boa forma. Como qualquer um que já cuidou de uma casa sabe, um pouco mais de esforço colocado na manutenção rotineira pode compensar bastante no longo prazo, prevenindo grandes reformas. Temos em mente uma particular operação de manutenção para nossa estrutura de dados unir-encontrar (*union-find*), destinada a manter as árvores curtas — durante cada encontrar, quando uma série de ponteiros pai é seguida até a raiz da árvore, nós mudaremos todos esses ponteiros para que apontem diretamente para a raiz (Figura 5.7). Essa heurística de *compressão de caminho* aumenta apenas levemente o tempo necessário para um encontrar e é fácil de codificar.

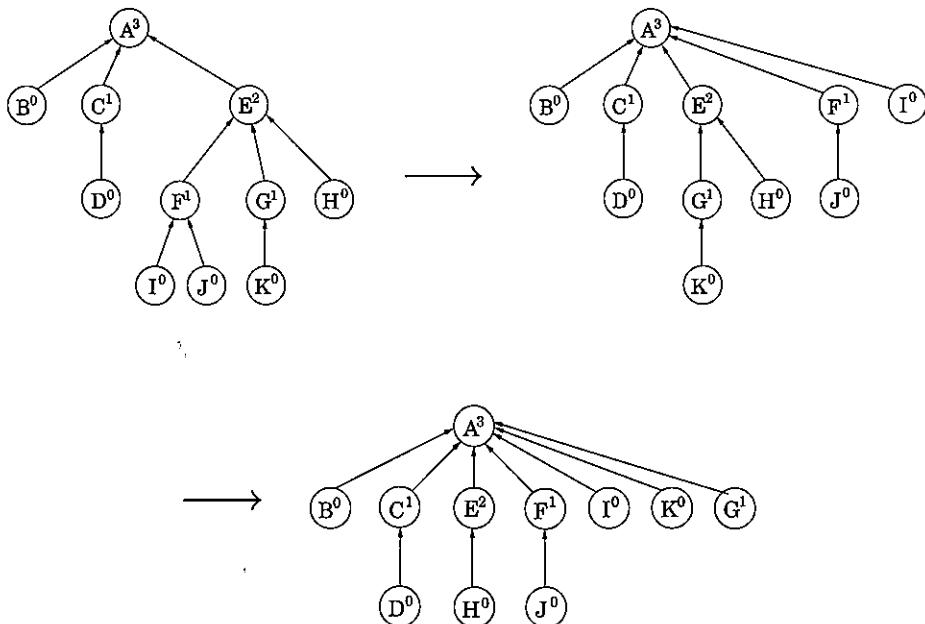
```
função encontrar( $x$ )
  se  $x \neq \pi(x)$ :  $\pi(x) = \text{encontrar}(\pi(x))$ 
  retornar  $\pi(x)$ 
```

O benefício dessa alteração simples é de longo prazo em vez de instantâneo e, assim, precisa de um tipo particular de análise: temos de examinar as *seqüências* de operações encontrar e unir, começando da estrutura de dados vazia e determinar o tempo médio por operação. Esse *custo amortizado* revela-se ligeiramente maior do que $O(1)$, bem abaixo do $O(\log n)$ anterior.

Pense na estrutura de dados como tendo um nível superior consistindo nos nós raiz e, abaixo desse nível, as partes internas das árvores. Existe uma divisão do trabalho: a operação de encontrar (com ou sem compressão de caminhos) apenas toca as partes internas das árvores, ao passo que unir somente examina o nível superior. Assim, compressão de caminho não tem qualquer efeito em operações unir e deixa o nível superior inalterado.

Agora sabemos que o rank dos nós raiz são inalterados, mas o que dizer de nós não-raiz? O ponto-chave aqui é que uma vez que um nó deixe de ser uma raiz, ele nunca

Figura 5.7 O efeito de compressão de caminho: encontrar(I) seguido de encontrar(K).



emerge novamente, e seu rank fica sempre fixo. Portanto o rank de todos os nós permanecem inalterados com compressão de caminho, muito embora esses números não possam mais ser interpretados como alturas de árvores. Em particular, as Propriedades 1–3 (da página 133) continuam valendo.

Se há n elementos, seus valores de rank podem variar de 0 a $\log n$ pela Propriedade 3. Vamos dividir a parte não-zero deste intervalo em certos intervalos cuidadosamente escolhidos, por razões que logo ficarão claras:

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, \dots, 16\}, \{17, 18, \dots, 2^{16} = 65536\}, \{65537, 65538, \dots, 2^{65536}\}, \dots$$

Cada grupo é da forma $\{k + 1, k + 2, \dots, 2^k\}$, onde k é uma potência de 2. O número de grupos é $\log^* n$, definido como o número operações log que têm de ser aplicadas sucessivamente a n até se obter 1 (ou algo menor do que 1). Por exemplo, $\log^* 1000 = 4$, pois $\log \log \log \log 1000 \leq 1$. Na prática, haverá apenas os primeiros cinco intervalos mostrados; mais intervalos são necessários somente se $n \geq 2^{65536}$, em outras palavras, nunca.

Em uma seqüência de operações `encontrar`, algumas podem tomar mais tempo do que outras. Vamos limitar o tempo de execução total usando uma contagem criativa. Especificamente, daremos a cada nó certa quantidade de dinheiro trocado, tal que o dinheiro total distribuído é no máximo $n \log^* n$ reais. Depois mostraremos que cada operação `encontrar` toma $O(\log^* n)$ passos, mais uma quantidade adicional de tempo

que pode ser “paga” usando o dinheiro trocado dos nós envolvidos — um real por unidade de tempo. Assim o tempo total para m operações de encontrar é $O(m \log^* n)$ mais no máximo $O(n \log^* n)$.

Em mais detalhes, um nó recebe seu dinheiro logo que ele deixa de ser uma raiz, ponto no qual seu rank é fixado. Se esse rank está no intervalo $\{k+1, \dots, 2^k\}$, o nó recebe 2^k reais. Pela Propriedade 3, o número de nós com rank $> k$ é limitado por

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \dots \leq \frac{n}{2^k}.$$

Portanto, o dinheiro total dado aos nós neste particular intervalo é no máximo n reais e, como existem $\log^* n$ intervalos, o dinheiro total desembolsado para todos os nós é $\leq n \log^* n$.

Agora, o tempo tomado por um encontrar específico é simplesmente o número de ponteiros seguidos. Considere os valores ascendentes de rank nesta cadeia de nós até a raiz. Nós x na cadeia caem em duas categorias: ou o rank de $\pi(x)$ está em um intervalo maior do que o rank de x , ou ele está no mesmo intervalo. Existem no máximo $\log^* n$ nós deste primeiro tipo (você pode ver por quê?), portanto o trabalho feito neles toma tempo $O(\log^* n)$. Os nós restantes — cujo rank de seus pais está no mesmo intervalo que os deles — têm de pagar um real do seu dinheiro trocado por seu tempo de processamento.

Isso funciona apenas se o dinheiro inicial de cada nó x é suficiente para cobrir todos os seus pagamentos na seqüência de operações encontrar. Aqui está a observação crucial: cada vez que x paga um real, seu pai muda para um rank superior. Portanto, se o rank de x está no intervalo $\{k+1, \dots, 2^k\}$, ele tem de pagar no máximo 2^k reais até que o rank do seu pai esteja em um intervalo superior; com o que, ele nunca tem de pagar novamente.

5.1.5 O algoritmo de Prim

Retornemos a nossa discussão sobre algoritmos para árvore espalhada mínima. O que a propriedade do corte nos diz em termos mais gerais é que qualquer algoritmo que se enquadre ao seguinte esquema guloso funciona com total garantia.

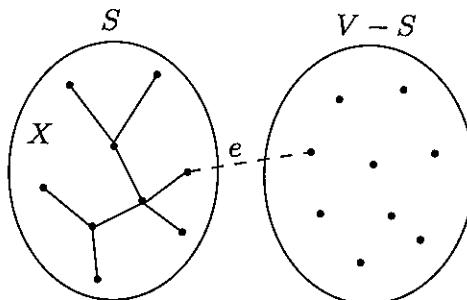
```
X = {} (arestas selecionadas até então)
repita até que |X| = |V| - 1:
    selecione um conjunto S ⊂ V para o qual X não tenha nenhuma
    aresta entre S e V - S
    seja e ∈ E a aresta de menor peso entre S e V - S
    X = X ∪ {e}
```

Uma alternativa popular ao algoritmo de Kruskal é o de Prim, no qual o conjunto de arestas X sempre forma uma subárvore e S é escolhido como o conjunto de vértices dessa árvore.

Em cada iteração, a subárvore definida por X cresce por uma aresta: a aresta mais leve entre um vértice de S e um vértice fora de S (Figura 5.8). Podemos pensar em S , de maneira equivalente, ou seja, crescendo ao incluir o vértice $v \notin S$ de menor custo:

$$\text{custo}(v) = \min_{u \in S} w(u, v).$$

Figura 5.8 O algoritmo de Prim: as arestas X formam uma árvore e S consiste nos seus vértices.



Isso lembra fortemente o algoritmo de Dijkstra e, de fato, o pseudocódigo é quase idêntico. A única diferença é nos valores de chave pelos quais a fila de prioridades é ordenada. No algoritmo de Prim, o valor de um nó é o peso da aresta mais leve que chega a ele vinda do conjunto S , ao passo que no de Dijkstra é o comprimento de um caminho inteiro para aquele nó partindo do ponto inicial. Não obstante, os dois algoritmos são similares o suficiente para terem o mesmo tempo de execução, que depende da particular implementação da fila de prioridades.

A Figura 5.9 mostra o algoritmo de Prim funcionando sobre um pequeno grafo de seis nós. Note como a AEM final é completamente especificada pelo vetor prev .

5.2 Codificação de Huffman

No esquema de compressão de áudio MP3, um sinal de som é codificado em três passos.

1. Ele é digitalizado por amostragem de intervalos regulares, gerando uma seqüência de números reais s_1, s_2, \dots, s_T . Por exemplo, a uma taxa de 44.100 amostras por segundo, uma sinfonia de 50 minutos corresponderia a $T = 50 \times 60 \times 44.100 \approx 130$ milhões de amostras.¹
2. Cada amostra de valor real é *quantizada*: aproximada para um número próximo de um conjunto finito Γ . Esse conjunto é cuidadosamente escolhido para explorar as limitações humanas de percepção, com a intenção de que a seqüência aproximada não seja distinguível de s_1, s_2, \dots, s_T pelo ouvido humano.
3. A *string* resultante de tamanho T sobre o alfabeto Γ é codificada em binário.

É no último passo que a codificação de Huffman é usada. Para entendermos seu papel, examinemos um exemplo simples no qual T é 130 milhões e o alfabeto Γ consiste em

¹Para o som estéreo, são necessários dois canais para dobrar o número de amostras.

Figura 5.9 Acima: o algoritmo de Prim para árvore espalhada mínima. Abaixo: uma ilustração do algoritmo de Prim, começando no nó A. Também é mostrada uma tabela de valores custo e prev e a AEM final.

procedimento prim(G, w)

Entrada: um grafo não-direcionado conexo $G = (V, E)$ com peso w_e nas arestas

Saída: uma árvore espalhada mínima definida pelo vetor prev

para todo $u \in V$:

custo(u) = ∞

prev(u) = null

selecione qualquer nó inicial u_0

custo(u_0) = 0

$H = \text{construir-fila}(V)$ (fila de prioridades, usando valores de custo como chaves)

enquanto H não está vazio:

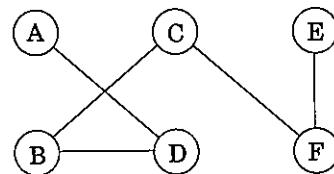
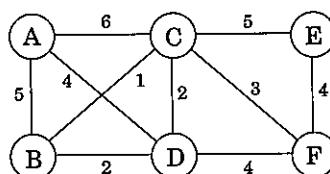
$v = \text{remover-min}(H)$

para cada $\{v, z\} \in E$:

se custo(z) > $w(v, z)$:

custo(z) = $w(v, z)$

prev(z) = v



Conjunto S	A	B	C	D	E	F
\emptyset	0/null	∞ /null				
A		5/A	6/A	4/A	∞ /null	∞ /null
A, D		2/D	2/D	4/A	∞ /null	4/D
A, D, B			1/B		∞ /null	4/D
A, D, B, C					5/C	3/C
A, D, B, C, F					4/F	

apenas quatro valores, denotados pelos símbolos A, B, C, D . Qual a forma mais econômica de escrever esta longa string em binário? A escolha óbvia é usar 2 bits por símbolo — digamos, palavra-código 00 para A , 01 para B , 10 para C e 11 D —, caso no qual 260 mega-bits são necessários no total. Será que existe uma codificação melhor do que essa?

Um algoritmo randomizado para o corte mínimo

Já vimos que árvores espalhadas e cortes estão intimamente relacionados. Agora abordaremos outra conexão. Vamos remover a última aresta que o algoritmo de Kruskal adiciona à árvore espalhada; isso quebra a árvore em duas componentes, portanto definindo um corte (S, \bar{S}) no grafo. O que podemos dizer sobre esse corte? Suponha que o grafo no qual estejamos trabalhando seja sem pesos e que suas arestas foram ordenadas uniformemente de maneira aleatória para o algoritmo de Kruskal processá-las. Veja um fato memorável: com probabilidade pelo menos $1/n^2$, (S, \bar{S}) é o corte mínimo no grafo, onde o tamanho de um corte é o número de arestas atravessando entre S e \bar{S} . Isso significa que repetindo o processo $O(n^2)$ vezes e apresentando o menor corte encontrado gera o corte mínimo com alta probabilidade: um algoritmo $O(mn^2 \log n)$ para cortes mínimos sem peso. Alguma sintonia adicional nos dá um algoritmo $O(n^2 \log n)$ para corte mínimo, inventado por David Karger, que é o algoritmo mais rápido que se conhece para este importante problema.

Assim, vamos ver por que o corte encontrado em cada iteração é o corte mínimo com probabilidade de pelo menos $1/n^2$. Em qualquer estágio do algoritmo de Kruskal, o conjunto de vértices V é partitionado em componentes conexas. As únicas arestas elegíveis para serem adicionadas à árvore têm suas extremidades em componentes distintas. O número de arestas incidentes a cada componente tem de ser pelo menos C , o tamanho do corte mínimo em G (pois poderíamos considerar um corte que separa esta componente do restante do grafo). Portanto, se há k componentes no grafo, o número de arestas elegíveis é pelo menos $kC/2$ (cada uma das k componentes tem no mínimo C arestas levando para fora dela, e precisamos compensar a contagem dupla de cada aresta). Como as arestas foram ordenadas aleatoriamente, a chance de que a próxima aresta elegível na lista seja do corte mínimo é de pelo menos $C/(kC/2) = 2/k$. Assim, com probabilidade ao menos $1 - 2/k = (k - 2)/k$, a escolha deixa o corte mínimo intato. Mas agora a chance de que o algoritmo de Kruskal deixe o corte mínimo intato por todo o caminho até a escolha da última aresta da árvore espalhada é de pelo menos

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} = \frac{1}{n(n-1)}.$$

Na busca por inspiração, examinemos nossa particular sequência e vemos que quatro símbolos não são igualmente abundantes.

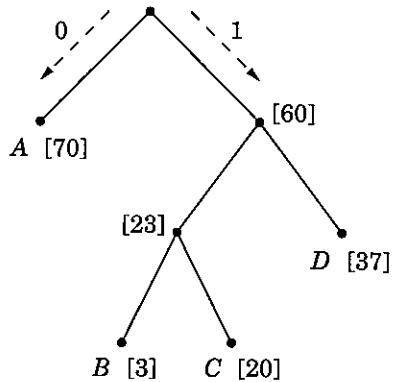
Símbolo	Freqüência
A	70 milhões
B	3 milhões
C	20 milhões
D	37 milhões

Será que existe alguma *codificação de tamanho variável*, na qual apenas *um* bit seja usado para o símbolo mais freqüente *A*, talvez à custa de precisar de três ou mais bits para os símbolos menos comuns?

Um perigo de ter palavras-código de comprimento variável é que a codificação resultante pode não ser unicamente decodificável. Por exemplo, se as palavras-código são

Figura 5.10 Uma codificação livre de prefixo. As freqüências são mostradas entre colchetes.

Símbolo	Palavra-símbolo
A	0
B	100
C	101
D	11



$\{0, 01, 11, 001\}$, a decodificação de *strings* como 001 é ambígua. Vamos evitar esse problema insistindo na propriedade de *ausência de prefixo*: nenhuma palavra-código pode ser um prefixo de outra palavra-código.

Qualquer codificação livre de prefixo pode ser representada por uma árvore binária *cheia* — quer dizer, uma árvore binária na qual cada nó tem ou zero ou dois filhos —, na qual os símbolos são as folhas e cada palavra-código é gerada por um caminho da raiz até uma folha, interpretando esquerda como 0 e direita como 1 (Exercício 5.29). A Figura 5.10 mostra um exemplo de uma codificação para os quatro símbolos A, B, C, D . A decodificação é única: a *string* de bits é decodificada começando na raiz, lendo a *string* da esquerda para a direita para mover para baixo na árvore, e, sempre que uma folha é alcançada, gerando o símbolo correspondente e retornando à raiz. É um esquema simples e compensa muito bem para nosso exemplo, em que (segundo os códigos da Figura 5.10) o tamanho total da *string* binária cai para 213 megabits, uma melhora de 17%.

Em geral, como podemos encontrar a árvore de codificação ótima, dadas as freqüências f_1, f_2, \dots, f_n de n símbolos? Para tornarmos o problema preciso, queremos uma árvore cujas folhas correspondam a um símbolo e que minimize o comprimento total da codificação,

$$\text{custo da árvore} = \sum_{i=1}^n f_i \cdot (\text{profundidade do } i\text{-ésimo símbolo na árvore})$$

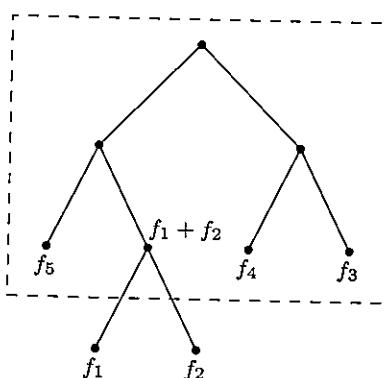
(o número de bits requeridos para um símbolo é exatamente sua profundidade na árvore).

Existe uma outra maneira de escrever essa função de custo que é muito útil. Embora sejam dadas apenas as freqüências para as folhas, podemos definir a freqüência para qualquer nó *interno* como a soma das freqüências de suas folhas descendentes; esse número é, afinal, o número de vezes que o nó interno é visitado durante a codificação ou decodificação. Durante o processo de codificação, cada vez que movemos para baixo na árvore, um bit é gerado para cada nó não-raiz pelo qual passamos. Portanto o custo total — o número total de bits gerados — pode também ser expresso assim:

O custo da árvore é a soma das freqüências de todas as suas folhas e nós internos, exceto a raiz.

A primeira formulação da função de custo nos diz que *os dois símbolos com as menores freqüências têm de estar no nível mais baixo da árvore ótima*, como filhos do nó interno mais baixo (o nó interno tem dois filhos, pois a árvore é cheia). Caso contrário, trocando os dois símbolos com o que quer que esteja mais baixo na árvore melhoraria a codificação.

Isso sugere que podemos começar a construir a árvore de maneira *gulosa*: encontre os dois símbolos com as menores freqüências, digamos i e j , e os faça filhos de um novo nó, que, então, tem freqüência $f_i + f_j$. Para mantermos a notação simples, consideremos que eles sejam f_1 e f_2 . Pela segunda formulação da função de custo, qualquer árvore na qual f_1 e f_2 são folhas irmãs tem custo $f_1 + f_2$ mais o custo para uma árvore com $n - 1$ folhas de freqüências $(f_1 + f_2), f_3, f_4, \dots, f_n$:



Esse último problema é simplesmente uma versão menor daquele com o qual começamos. Portanto, tiramos f_1 e f_2 para fora da lista de freqüências, inserimos $(f_1 + f_2)$ e repetimos. O algoritmo resultante pode ser descrito em termos de operações de fila de prioridades (como definidas na página 109) e toma tempo $O(n \log n)$ se um heap binário (Seção 4.5.2) é usado.

procedimento Huffman(f)

Entrada: um vetor $f[1 \dots n]$ de freqüências

Saída: uma árvore de codificação com n folhas

seja H a fila de prioridades de inteiros, ordenados por f
para $i = 1$ até n : inserir(H, i)

para $k = n + 1$ até $2n - 1$:

$i = \text{remover-min}(H)$, $j = \text{remover-min}(H)$

criar um nó de número k com filhos i, j

$f[k] = f[i] + f[j]$

inserir(H, k)

Retornando para o nosso exemplo simples: você pode dizer se a árvore da Figura 5.10 é ótima?

Entropia

A corrida de cavalos anual regional está trazendo três novos puros-sangues que nunca competiram um contra o outro. Animado, você estuda suas últimas 200 corridas e as resume como uma distribuição de probabilidades sobre quatro eventos: **primeiro** ("primeiro lugar"), **segundo**, **terceiro** e **outros**.

Evento	Aurora	Turbilhão	Fantasma
primeiro	0,15	0,30	0,20
segundo	0,10	0,05	0,30
terceiro	0,70	0,25	0,30
outros	0,05	0,40	0,20

Qual cavalo é o mais previsível? Uma abordagem quantitativa a esta questão é olhar a *compressibilidade*. Escreva o histórico de cada cavalo como uma *string* de 200 valores (**primeiro**, **segundo**, **terceiro**, **outros**). O número total de bits necessários para codificar essas *strings* de histórico pode, então, ser computado usando o algoritmo de Huffman. Isso resulta em 290 bits para Aurora, 380 para Turbilhão e 420 para Fantasma (verifique isto!).

A imprevisibilidade inerente, ou *aleatoriedade*, de uma distribuição de probabilidades pode ser tomada pela medida na qual é possível comprimir dados extraídos da distribuição.

$$\text{mais compressível} \equiv \text{menos aleatório} \equiv \text{mais previsível}$$

Suponha que haja n possíveis eventos, com probabilidades p_1, p_2, \dots, p_n . Se uma seqüência de m valores é tomada desta distribuição, então o i -ésimo evento vai aparecer aproximadamente mp_i vezes (se m é grande). Por simplicidade, considere que essas sejam exatamente as freqüências observadas e, além disso, que os p_i sejam todos potências de 2 (ou seja, da forma $1/2^k$). Podemos ver por indução (Exercício 5.19) que o número de bits necessários para codificar a seqüência é $\sum_{i=1}^n mp_i \log(1/p_i)$. Assim, o número médio de bits necessários para codificar uma única amostra da distribuição é

$$\sum_{i=1}^n p_i \log \frac{1}{p_i}.$$

Essa é a *entropia* da distribuição, a medida de quanta aleatoriedade ela contém.

Por exemplo, uma moeda justa tem dois eventos, cada um com probabilidade $1/2$. Portanto sua entropia é

$$\frac{1}{2} \log 2 + \frac{1}{2} \log 2 = 1.$$

Isso é bastante natural: um lance de moeda contém um bit de aleatoriedade. Mas o que dizer se a moeda não é justa, se ela tem uma chance de $3/4$ de dar cara? Então a entropia é

$$\frac{3}{4} \log \frac{4}{3} + \frac{1}{4} \log 4 = 0,81.$$

Uma moeda viciada é mais previsível do que uma moeda justa e, assim, tem uma entropia menor. À medida que o vício da moeda torna-se mais pronunciado, a entropia cai na direção de zero.

Exploraremos mais esses conceitos nos Exercícios 5.18 e 5.19.

5.3 Fórmulas Horn

Para apresentar inteligência de nível humano, um computador tem de ser capaz de realizar pelo menos um raciocínio lógico módico. Fórmulas Horn são um particular esquema para fazer isso, para expressar fatos lógicos e derivar conclusões.

O objeto mais primitivo em uma fórmula Horn é uma *variável booleana*, tomando valores **verdadeiro** ou **falso**. Por exemplo, variáveis x , y e z poderiam denotar as seguintes possibilidades.

- $x \equiv$ o assassinato aconteceu na cozinha
- $y \equiv$ o mordomo é inocente
- $z \equiv$ o coronel estava dormindo às 8 da noite

Um *literal* é ou uma variável x ou a sua negação \bar{x} (“**NÃO-** x ”). Em fórmulas Horn, conhecimento sobre as variáveis é representado por dois tipos de *cláusulas*:

1. *Implicações*, cujo lado esquerdo é um \wedge de qualquer número de literais positivos e cujo lado direito é um único literal positivo. Isso expressa sentenças da forma “se as condições à esquerda valem, então as da direita também têm de ser verdadeiras”. Por exemplo,

$$(z \wedge w) \Rightarrow u$$

poderia significar “se o coronel estava dormindo às 8 da noite e o assassinato aconteceu às 8 da noite, então o coronel é inocente”. Um tipo degenerado de implicação é a unidade *singleton* “ $\Rightarrow x$,” significando que x é **verdadeiro**: “o assassinato definitivamente aconteceu na cozinha”.

2. *Cláusulas puramente negativas*, consistindo em um ou de qualquer número de literais negativos, como em

$$(\bar{u} \vee \bar{v} \vee \bar{y})$$

(“eles não podem ser todos inocentes”).

Dado um conjunto de cláusulas desses dois tipos, o objetivo é determinar se existe uma explicação coerente: uma atribuição de valores **verdadeiro**/**falso** às variáveis que satisfaça todas as cláusulas. Isso é chamado também de *atribuição satisfatória*.

Os dois tipos de cláusulas nos levam para direções diferentes. As implicações nos dizem para tornar algumas das variáveis **verdadeiras**, enquanto as cláusulas negativas nos encorajam a torná-las **falsas**. Nossa estratégia para resolver uma fórmula Horn é esta: começamos com todas as variáveis **falsas**. Depois, procedemos tornando algumas delas **verdadeiras**, uma por uma, mas muito relutantemente e apenas se absolutamente tivermos de fazer porque uma implicação seria violada caso contrário. Depois que essa fase está terminada e todas as implicações estão satisfeitas, somente então nos voltamos para as cláusulas negativas e nos asseguramos de que sejam todas satisfeitas.

Em outras palavras, nosso algoritmo para cláusulas Horn é o seguinte esquema guloso (*avarento* seria talvez mais descriptivo):

Entrada: uma fórmula Horn
 Saída: uma atribuição satisfatória, se uma existe

faça todas as variáveis falsas

enquanto existe uma implicação que não está satisfeita:
 faça a variável do lado direito da implicação verdadeira

se todas as cláusulas puramente negativas estão satisfeitas:
 retornar a atribuição
 senão: retornar "fórmula não é satisfeita"

Por exemplo, suponha que a fórmula seja

$$(w \wedge y \wedge z) \Rightarrow x, \quad (x \wedge z) \Rightarrow w, \quad x \Rightarrow y, \quad \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad (\bar{w} \vee \bar{x} \vee \bar{y}), \quad (\bar{z})$$

Começamos com tudo falso e, então, notamos que x tem de ser verdadeiro, por conta da implicação unitária $\Rightarrow x$. Depois vemos que y tem de ser verdadeiro também, por causa de $x \Rightarrow y$. E assim por diante.

Para entender por que o algoritmo é correto, note que se ele retorna uma atribuição, esta atribuição satisfaz tanto as implicações quanto as cláusulas negativas e, portanto, ela é de fato uma atribuição satisfatória para a fórmula Horn da entrada. Portanto precisamos apenas nos convencer de que se o algoritmo não encontra nenhuma atribuição satisfatória, então, realmente não existe uma. Isso é o caso porque a nossa regra "avarenta" mantém a seguinte invariante:

Se um certo conjunto de variáveis são feitas verdadeiras, elas têm de ser verdadeiras em qualquer atribuição satisfatória.

Assim, se a atribuição encontrada depois do loop *enquanto* não satisfaz as cláusulas negativas, então, não pode haver nenhuma atribuição satisfatória.

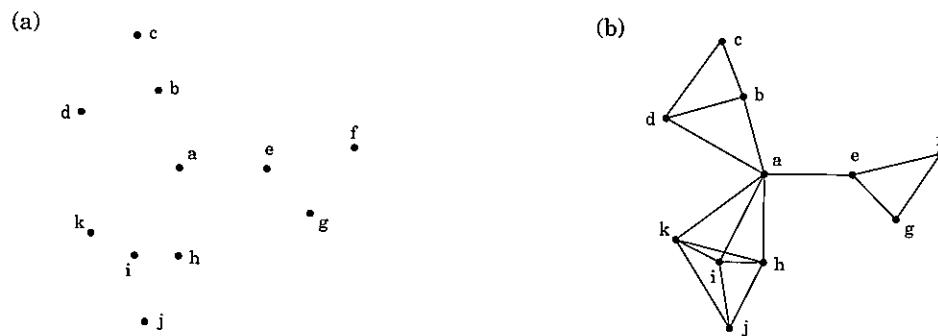
Fórmulas Horn estão no cerne de Prolog ("programming by logic"), uma linguagem na qual você programa especificando as propriedades desejadas da saída, usando expressões lógicas simples. A locomotiva dos interpretadores Prolog é o nosso algoritmo guloso para satisfação. Convenientemente, ele pode ser implementado em tempo linear no tamanho da fórmula, você pode ver como (Exercício 5.33)?

5.4 Cobertura de vértices

Os pontos na Figura 5.11 representam uma coleção de cidades. Esta região está nos estágios iniciais de planejamento e seus administradores estão decidindo onde colocar escolas. Há apenas duas restrições: cada escola deve ser em uma cidade e ninguém deve viajar mais do que 50 quilômetros para alcançar uma delas. Qual o número mínimo de escolas necessário?

Esse é um típico problema de cobertura de vértices. Para cada cidade x , seja S_x o conjunto de cidades a no máximo 50 quilômetros dela. Uma escola em x essencialmente

Figura 5.11 (a) Onze cidades. (b) Cidades que estão a no máximo 50 quilômetros umas das outras.



“cobriria” as outras cidades. A questão é, então, quantos conjuntos S_x têm de ser selecionados para cobrir todas as cidades da região?

Cobertura de vértices

Entrada: Um conjunto de elementos B ; conjuntos $S_1, \dots, S_m \subseteq B$.

Saída: Uma seleção dos S_i cuja união é B .

Custo: Número de conjuntos selecionados.

(No nosso exemplo, os elementos de B são as cidades.) Esse problema se presta imediatamente a uma solução gulosa:

Repetir até que todos os elementos de B estejam cobertos:

Selecionar o conjunto S_i com o maior número de elementos não cobertos.

Isso é extremamente natural e intuitivo. Vamos ver o que aconteceria no nosso exemplo anterior: primeiro seria posto uma escola na cidade a , pois tal procedimento cobre o maior número de outras cidades. Depois, três outras escolas seriam escolhidas — c, j e ou f ou g — totalizando quatro. Entretanto, existe uma solução com apenas três escolas, em b, e e i . O esquema guloso não é ótimo!

Mas felizmente, não está muito longe disso.

Afirmiação Suponha que B contenha n elementos e que a cobertura ótima consista em k conjuntos. Então o algoritmo guloso vai usar, no máximo, $k \ln n$ conjuntos.²

Seja n_t o número de elementos ainda não cobertos depois de t iterações do algoritmo guloso (portanto $n_0 = n$). Como os elementos restantes são cobertos pelos k conjuntos

² Significa “logaritmo natural”, isto é, para a base e .

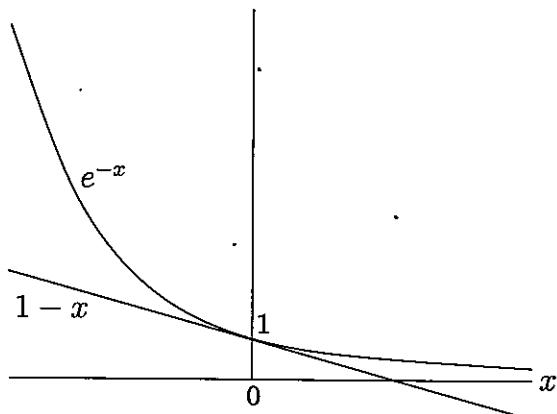
ótimos, tem de haver algum conjunto com pelo menos n_t/k deles. Portanto, a estratégia gulosa vai assegurar que

$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right),$$

que por aplicações sucessivas implica $n_t \leq n_0(1 - 1/k)^t$. Uma cota mais conveniente pode ser obtida da seguinte inequação útil

$$1 - x \leq e^{-x} \text{ para todo } x, \text{ com igualdade se e somente se } x = 0,$$

que é mais facilmente demonstrada por uma figura:



Assim

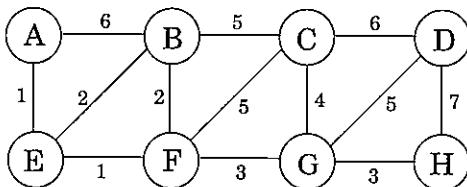
$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t < n_0 (e^{-1/k})^t = n e^{-t/k}$$

Em $t = k \ln n$, portanto, n_t é estritamente menor do que $n e^{-\ln n} = 1$, que significa que nenhum elemento resta para ser coberto.

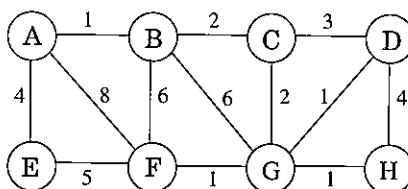
A razão entre a solução do algoritmo guloso e a solução ótima varia de entrada para entrada, mas é sempre menor do que $\ln n$. Há certas entradas para as quais a razão é muito próxima de $\ln n$ (Exercício 5.34). Chamamos a razão máxima de *fator de aproximação* do algoritmo guloso. Parece haver muito espaço para aperfeiçoamento, contudo, na verdade, tais esperanças são injustificadas: sob certas hipóteses de complexidade amplamente aceitas (que estarão claras quando alcançarmos o Capítulo 8), não existe, provavelmente, nenhum algoritmo de tempo polinomial com um fator de aproximação menor.

Exercícios

- 5.1. Considere o seguinte grafo.



- (a) Qual o custo da sua árvore espalhada mínima?
 - (b) Quantas árvores espalhadas mínimas ele tem?
 - (c) Suponha que o algoritmo de Kruskal seja executado sobre esse grafo. Em que ordem as arestas são adicionadas à AEM? Para cada aresta nessa seqüência, dê um corte que justifique sua adição.
- 5.2. Suponha que desejamos encontrar a árvore espalhada mínima do seguinte grafo.



- (a) Execute o algoritmo de Prim; sempre que houver uma escolha de nós, use a ordem alfabética (por exemplo, comece pelo nó A). Desenhe uma tabela mostrando os valores intermediários do vetor de custo.
 - (b) Execute o algoritmo de Kruskal no mesmo grafo. Mostre como fica a estrutura de dados para conjuntos disjuntos em cada estágio intermediário (incluindo a estrutura das árvores direcionadas), considerando que compressão de caminho seja usada.
- 5.3. Projete um algoritmo de tempo linear para a seguinte tarefa.

Entrada: Um grafo não-direcionado, conexo, G .

Pergunta: Existe uma aresta que você possa remover de G sem desconectá-lo?

Você pode reduzir o tempo de execução do seu algoritmo para $O(|V|)$?

- 5.4. Mostre que se um grafo não-direcionado com n vértices tem k componentes conexas, então ele tem pelo menos $n - k$ arestas.
- 5.5. Considere um grafo não-direcionado $G = (V, E)$ com pesos de aresta não-negativos $w_e \geq 0$. Suponha que você computou uma árvore espalhada mínima de G e que também computou os caminhos mínimos para todos os nós partindo de um particular nó $s \in V$. Agora suponha que cada peso de aresta seja aumentado em uma unidade: os novos pesos são $w'_e = w_e + 1$.
- (a) Será que a árvore espalhada mínima muda? Dê um exemplo para o qual ela muda ou prove que ela não pode mudar.

- (b) Será que os caminhos mínimos mudam? Dê um exemplo para o qual eles mudam ou prove que isso não pode ocorrer.
- 5.6. Seja $G = (V, E)$ um grafo não-direcionado. Prove que se todas as suas arestas são distintas, então ele tem uma única árvore espalhada mínima.
- 5.7. Mostre como encontrar a árvore espalhada *máxima* de um grafo, isto é, a árvore espalhada com o maior peso total.
- 5.8. Suponha que seja dado um grafo conexo com pesos $G = (V, E)$ com um determinado vértice s e onde todos os pesos de aresta sejam positivos e distintos. Será que é possível que uma árvore de caminhos mínimos partindo de s e uma árvore espalhada mínima de G não compartilhem nenhuma aresta? Se afirmativo, dê um exemplo. Do contrário, apresente uma razão.
- 5.9. As sentenças seguintes podem ou não estar corretas. Em cada caso, prove a sentença (se ela for correta) ou forneça um contra-exemplo (se não for correta). Sempre considere o grafo $G = (V, E)$ não-direcionado e conexo. Não considere que os pesos nas arestas sejam distintos, a menos que isso seja explicitamente afirmado.
- Se um grafo G tem mais do que $|V| - 1$ arestas e existe uma única aresta de maior peso, então esta aresta não pode ser parte da árvore espalhada mínima.
 - Se G tem um ciclo com uma aresta de maior peso única e , então e não pode ser parte de nenhuma AEM.
 - Seja e qualquer aresta de peso mínimo em G . Então e tem de ser parte de alguma AEM.
 - Se a aresta de menor peso em um grafo é única, então ela tem de ser parte da todas as AEM.
 - Se e é parte de alguma AEM de G , então ela tem de ser a aresta de menor peso através de algum corte de G .
 - Se G tem um ciclo com uma aresta mais leve única e , então e tem de ser parte de todas as AEM.
 - A árvore de caminhos mínimos computada pelo algoritmo de Dijkstra é necessariamente uma AEM.
 - O caminho mínimo entre dois nós é necessariamente parte de alguma AEM.
 - O algoritmo de Prim funciona corretamente quando existem arestas negativas.
 - (Para qualquer $r > 0$, defina um r -caminho cujas arestas tenham todos peso $< r$.) Se G contém um r -caminho do nó s até t , então toda AEM de G tem de conter também um r -caminho do nó s até o nó t .
- 5.10. Seja T uma AEM de um grafo G . Dado um subgrafo conexo H de G , mostre que $T \cap H$ está contido em alguma AEM de H .
- 5.11. Forneça o estado da estrutura de dados de conjunto disjuntos depois da seguinte seqüência de operações, começando com os conjuntos unitários $\{1\}, \dots, \{8\}$. Use compressão de caminho. Em caso de empate, sempre faça a raiz de menor número apontar para a raiz de maior número.

$\text{unir}(1, 2)$, $\text{unir}(3, 4)$, $\text{unir}(5, 6)$, $\text{unir}(7, 8)$, $\text{unir}(1, 4)$,
 $\text{unir}(6, 7)$, $\text{unir}(4, 5)$, $\text{encontrar}(1)$

- 5.12. Suponha que você implemente a estrutura de dados de conjuntos disjuntos usando união por rank, mas não compressão de caminho. Dê uma seqüência de m operações `unir` e `encontrar` sobre n elementos que tome tempo $\Omega(m \log n)$.
- 5.13. Considere uma longa *string* que consista nos quatro caracteres *A*, *C*, *G*, *T*; eles aparecem com freqüência 31%, 20%, 9% e 40%, respectivamente. Qual a codificação de Huffman desses quatro caracteres?
- 5.14. Suponha que os símbolos *a*, *b*, *c*, *d*, *e* ocorram com freqüências $1/2$, $1/4$, $1/8$, $1/16$, $1/16$, respectivamente.
- Qual a codificação de Huffman do alfabeto?
 - Se esta codificação é aplicada a um arquivo consistindo em 1.000.000 caracteres com as dadas freqüências, qual é o tamanho do arquivo codificado em bits?
- 5.15. Usamos o algoritmo de Huffman para obter uma codificação do alfabeto $\{a, b, c\}$ com freqüências f_a, f_b, f_c . Em cada um dos seguintes casos, dê um exemplo de freqüências (f_a, f_b, f_c) que levaria ao código especificado, ou explique por que o código não pode ser obtido (não importa quem sejam as freqüências).
- Código: $\{0, 10, 11\}$
 - Código: $\{0, 1, 00\}$
 - Código: $\{10, 01, 00\}$
- 5.16. Prove as seguintes duas propriedades do esquema de codificação de Huffman.
- Se algum caractere ocorre com freqüência maior do que $2/5$, então garantidamente existe uma palavra-código de tamanho 1.
 - Se todos os caracteres ocorrem com freqüência menor do que $1/3$, então garantidamente não existe nenhuma palavra-código de tamanho 1.
- 5.17. Sob uma codificação de Huffman de n símbolos com freqüências f_1, f_2, \dots, f_n , qual é o maior tamanho que uma palavra-código pode ter? Forneça um exemplo de conjunto de freqüências que produziria este caso.
- 5.18. A tabela seguinte dá as freqüências das letras da língua inglesa (incluindo o espaço em branco separador de palavras) em um corpo particular.

espaço	18,3 %	r	4,8 %	y	1,6 %
e	10,2 %	d	3,5 %	p	1,6 %
t	7,7 %	l	3,4 %	b	1,3 %
a	6,8 %	c	2,6 %	v	0,9 %
o	5,9 %	u	2,4 %	k	0,6 %
i	5,8 %	m	2,1 %	j	0,2 %
n	5,5 %	w	1,9 %	x	0,2 %
s	5,1 %	f	1,8 %	q	0,1 %
h	4,9 %	g	1,7 %	z	0,1 %

- Qual a codificação de Huffman ótima para este alfabeto?
- Qual o número esperado de bits por letra?

- (c) Suponha agora que calculemos a entropia dessas freqüências

$$H = \sum_{i=0}^{26} p_i \log \frac{1}{p_i}$$

(veja o quadro na página 143). Você esperaria que ela fosse maior ou menor do que a sua resposta acima? Explique.

- (d) Você acha que este é o limite de quanto um texto em inglês pode ser comprimido? Que características da língua inglesa, além de letras e suas freqüências, um esquema de compressão deveria levar em conta?

- 5.19. *Entropia*. Considere uma distribuição sobre n possíveis eventos, com probabilidades p_1, p_2, \dots, p_n .

- (a) Apenas para esta parte do problema, considere que cada p_i seja uma potência de 2 (ou seja, da forma $1/2^k$). Suponha que uma seqüência longa de m amostras seja extraída da distribuição e que para todo $1 \leq i \leq n$, o i -ésimo evento ocorra exatamente mp_i vezes na seqüência. Mostre que se o algoritmo de Huffman é aplicado a esta seqüência, a codificação resultante terá tamanho

$$\sum_{i=1}^n mp_i \log \frac{1}{p_i}$$

- (b) Agora considere distribuições arbitrárias — ou seja, as probabilidades p_i não estão restritas a potências de 2. A medida mais freqüentemente usada para o grau de aleatoriedade na distribuição é a *entropia*

$$\sum_{i=1}^n p_i \log \frac{1}{p_i}$$

Para que distribuição (sobre n eventos) a entropia é a máxima possível? E a mínima possível?

- 5.20. Forneça um algoritmo de tempo linear que tome como entrada uma árvore e determine se ela tem um *emparelhamento perfeito*: um conjunto de arestas que tocam cada vértice exatamente uma vez.

- 5.21. Um *conjunto feedback de arestas* de um grafo não-direcionado $G = (V, E)$ é subconjunto de arestas $E' \subseteq E$ que intercepta todos os ciclos do grafo. Assim, remover as arestas E' do grafo o torna acíclico.

Forneça um algoritmo eficiente para o seguinte problema:

Entrada: Grafo não-direcionado $G = (V, E)$ com pesos de aresta positivos w_e

Saída: Um conjunto feedback de arestas $E' \subseteq E$ de peso total mínimo $\sum_{e \in E'} w_e$

- 5.22. Neste problema, vamos desenvolver um novo algoritmo para encontrar árvores espalhadas mínimas. Ele é baseado na seguinte propriedade:

Selecione qualquer ciclo no grafo e seja e a aresta mais pesada deste ciclo. Então existe uma árvore espalhada mínima que não contém e .

- (a) Prove a propriedade cuidadosamente.
- (b) Aqui está o novo algoritmo para AEM. A entrada é algum grafo não-direcionado $G = (V, E)$ (em formato de lista de adjacência) com pesos nas arestas $\{w_e\}$. ordene as arestas de acordo com seus pesos para cada aresta $e \in E$, em ordem decrescente de w_e :
 se e é parte de um ciclo de G :
 $G = G - e$ (quer dizer, remova e de G)
 retornar G
- Prove que o algoritmo está correto.
- (c) Em cada iteração, o algoritmo tem de checar se existe um ciclo contendo uma aresta específica e . Forneça um algoritmo de tempo linear para esta tarefa e justifique sua correção.
- (d) Qual o tempo total tomado por este algoritmo, em termos de $|E|$?

- 5.23. É dado um grafo $G = (V, E)$ com pesos de aresta positivos e uma árvore espalhada mínima $T = (V, E')$ com relação a estes pesos; você pode considerar que G e T são dados como listas de adjacência. Agora suponha que o peso de uma particular aresta $e \in E$ seja modificado de $w(e)$ para um novo valor $\hat{w}(e)$. Você quer rapidamente atualizar a árvore espalhada mínima T para refletir a mudança, sem recomputar a árvore inteira do nada. Existem quatro casos. Em cada caso forneça um algoritmo de tempo linear para atualizar a árvore.
- (a) $e \notin E'$ e $\hat{w}(e) > w(e)$.
 - (b) $e \notin E'$ e $\hat{w}(e) < w(e)$.
 - (c) $e \in E'$ e $\hat{w}(e) < w(e)$.
 - (d) $e \in E'$ e $\hat{w}(e) > w(e)$.
- 5.24. Às vezes desejamos árvores espalhadas leves com certas propriedades especiais. Veja um exemplo.

Entrada: Grafo não-direcionado $G = (V, E)$; pesos de aresta w_e ; subconjunto de vértices $U \subset V$

Saída: A árvore espalhada mais leve na qual os nós de U são folhas (pode haver outras folhas na árvore também).

(A resposta não é necessariamente uma árvore espalhada mínima.)

Forneça um algoritmo para este problema que rode em tempo $O(|E| \log |V|)$. (Dica: Quando você remove os nós U da solução ótima, o que resta?)

- 5.25. Um contador binário de tamanho não especificado suporta duas operações: `incrementar` (que aumenta seu valor por um) e `zerar` (que faz seu valor ser zero). Mostre que, começando de um contador inicialmente zerado, qualquer seqüência de n operações `incrementar` e `zerar` toma tempo $O(n)$; isto é, o tempo amortizado por operação é $O(1)$.
- 5.26. Este é um problema que ocorre em análise automática de programas. Para um conjunto de variáveis x_1, \dots, x_n , são dadas algumas restrições de *igualdade*, da forma

$x_i = x_j$ " e algumas restrições de *desigualdade*, da forma " $x_i \neq x_j$ ". Será que é possível satisfazer todas elas?

Por exemplo, as restrições

$$x_1 = x_2, x_2 = x_3, x_3 = x_4, x_1 \neq x_4,$$

não podem ser satisfeitas. Forneça um algoritmo eficiente que tome como entrada m restrições sobre n variáveis e decida se as restrições podem ser satisfeitas.

- 5.27. *Grafos com seqüências de graus designadas.* Dada uma lista de n inteiros positivos d_1, d_2, \dots, d_n , queremos eficientemente determinar se existe um grafo $G = (V, E)$ cujos nós têm grau precisamente d_1, d_2, \dots, d_n , isto é, se $V = \{v_1, \dots, v_n\}$, então o grau de v_i deve ser exatamente d_i . Chamamos (d_1, d_2, \dots, d_n) de a *seqüência de graus* de G . O grafo G não deve ter ciclos unitários (arestas com ambos os extremos no mesmo nó) ou arestas múltiplas entre o mesmo par de nós.
- Dê um exemplo de d_1, d_2, d_3, d_4 no qual todo $d_i \leq 3$ e $d_1 + d_2 + d_3 + d_4$ é par, mas para a qual nenhum grafo com seqüência de graus (d_1, d_2, d_3, d_4) existe.
 - Suponha que $d_1 \geq d_2 \geq \dots \geq d_n$ e que exista um grafo $G = (V, E)$ com seqüência de graus (d_1, \dots, d_n) . Queremos mostrar que tem de haver um grafo que tenha essa seqüência de graus e, além disso, os vizinhos de v_1 são $v_2, v_3, \dots, v_{d_1+1}$. A idéia é transformar gradualmente G em um grafo com a propriedade adicional desejada.
 - Suponha que os vizinhos de v_1 em G não sejam $v_2, v_3, \dots, v_{d_1+1}$. Mostre que existe $i < j \leq n$ e $u \in V$ tal que $\{v_1, v_i\}, \{u, v_j\} \notin E$ e $\{v_1, v_j\}, \{u, v_i\} \in E$.
 - Especifique as mudanças que você faria em G para obter o novo grafo $G' = (V, E')$ com a mesma seqüência de graus de G e $(v_1, v_i) \in E'$.
 - Agora mostre que tem de existir um grafo com a dada seqüência de graus, mas no qual v_1 tem vizinhos $v_2, v_3, \dots, v_{d_1+1}$.
 - Usando o resultado da parte (b), descreva um algoritmo que sobre uma entrada d_1, \dots, d_n (não necessariamente ordenados) decida se existe um grafo com esta seqüência de graus. Seu algoritmo deve rodar em tempo polinomial em n .
- 5.28. Alice quer dar uma festa e está decidindo quem chamar. Ela tem n pessoas as quais escolher, e ela fez uma lista de quais pares dessas pessoas conhecem uma a outra. Ela quer selecionar o maior número de pessoas possível, sujeito a duas restrições: na festa, cada pessoa deve ter pelo menos outras cinco pessoas que ela conhece e outras cinco pessoas que ela não conhece.
- Forneça um algoritmo eficiente que tome como entrada a lista das n pessoas e a lista de pares de quem conhece quem e calcule a melhor escolha de convidados para a festa. Dê o tempo de execução em termos de n .
- 5.29. Uma *codificação livre de prefixo* de um alfabeto finito Γ atribui a cada símbolo em Γ uma palavra-código binária, tal que nenhuma palavra-código é um prefixo de outra

palavra-código. Uma codificação livre de prefixo é *minimal* se não é possível chegar à outra codificação livre de prefixo (dos mesmos símbolos) contraindo-se algumas das palavras-código. Por exemplo, a codificação $\{0, 101\}$ não é minimal, pois a palavra-código 101 pode ser contraída para 1 ainda mantendo a propriedade da ausência de prefixo.

Mostre que uma codificação livre de prefixo minimal pode ser representada por uma árvore binária cheia na qual cada folha corresponde a um elemento único de Γ , cuja palavra-código é gerada pelo caminho da raiz até aquela folha (interpretando uma ramificação à esquerda como 0 e uma ramificação à direita como 1).

- 5.30. *Huffman ternário*. Trimedia Disks Inc. desenvolveu discos rígidos "ternários". Cada célula em um disco pode agora guardar valores 0 , 1 , ou 2 (em vez de apenas 0 ou 1). Para aproveitar a nova tecnologia, desenvolva um algoritmo de Huffman modificado para comprimir seqüências de caracteres de um alfabeto de tamanho n , onde os caracteres ocorrem com freqüências conhecidas f_1, f_2, \dots, f_n . Seu algoritmo deve codificar cada caractere com uma palavra-código de tamanho variável sobre os valores $0, 1, 2$ tal que nenhuma palavra-código seja um prefixo de outra e para obter a máxima compressão possível. Prove que seu algoritmo é correto.
- 5.31. A intuição básica por trás do algoritmo de Huffman, que blocos freqüentes devem ter codificação mais curta e blocos pouco freqüentes, mais longas, também está presente em inglês, em que palavras típicas como *I*, *you*, *is*, *and*, *to*, *from* e *so* são curtas e palavras raramente usadas como *velociraptor* são mais longas.

Entretanto, palavras como *fire!*, *help!* e *run!* são curtas não porque são freqüentes, mas talvez porque o tempo é precioso em situações onde elas são usadas.

Para tornar a coisa mais teórica, suponha que tenhamos um arquivo composto de m diferentes palavras, com freqüências f_1, \dots, f_m . Suponha também que para a i -ésima palavra, o custo por bit de codificação seja c_i . Assim, se encontrarmos um código livre de prefixo onde a i -ésima palavra tem uma palavra-código de tamanho l_i , então o custo total da codificação será de $\sum_i f_i \cdot c_i \cdot l_i$.

Mostre como encontrar a codificação livre de custo total mínimo.

- 5.32. Um servidor tem n usuários esperando para serem servidos. O tempo de serviço requerido por usuário é conhecido previamente: é t_i minutos para o usuário i . Portanto se, por exemplo, os usuários são servidos em ordem crescente de i , então o i -ésimo usuário tem de esperar $\sum_{j=1}^i t_j$ minutos.

Queremos minimizar o tempo total de espera

$$T = \sum_{i=1}^n (\text{tempo gasto pelo usuário } i \text{ na espera}).$$

Forneça um algoritmo eficiente para computar a ordem ótima na qual processar os usuários.

- 5.33. Mostre como implementar um algoritmo avarento para satisfação de fórmulas Horn (Seção 5.3) em tempo linear no tamanho da fórmula (o número de ocorrências de literais nela).

5.34. Mostre que, para qualquer inteiro que é uma potência de 2, existe uma instância do problema de cobertura de vértices (Seção 5.4) com as seguintes propriedades:

- i. Existem n elementos no conjunto-base.
- ii. A cobertura ótima usa apenas dois conjuntos.
- iii. O algoritmo guloso seleciona pelo menos $\log n$ conjuntos.

Portanto a razão de aproximação que derivamos no Capítulo é justa.

5.35. Mostre que um grafo sem pesos com n nós tem no máximo $n(n-1)$ cortes mínimos distintos.

Capítulo 6

Programação dinâmica

Nos capítulos anteriores vimos alguns princípios elegantes de projeto — tal como divisão-e-conquista, exploração de grafos e escolha gulosa — que levam a algoritmos definitivos para uma variedade de importantes tarefas computacionais. A desvantagem dessas ferramentas é que somente podem ser usadas em tipos muito específicos de problemas. Nós agora nos voltamos para os dois *pesos pesados* da área de algoritmos, *programação dinâmica* e *programação linear*, técnicas de aplicabilidade muito ampla que podem ser invocadas quando métodos mais especializados falham. Previsivelmente, essa generalidade por vezes vem com um custo na eficiência.

6.1 Caminhos mínimos em dags, revisitados

Na conclusão do nosso estudo de caminhos mínimos (Capítulo 4), observamos que o problema é especialmente fácil em grafos direcionados acíclicos (dags). Recapitulemos esse caso, porque ele está no coração de programação dinâmica.

A característica especial que distingue um dag é que seus nós podem ser *linearizados*, eles podem ser arrumados em uma linha de modo que todas as arestas sigam da esquerda para a direita (Figura 6.1). Para entender de que maneira isso ajuda no assunto caminhos mínimos, suponha que queiramos descobrir as distâncias do nó S para todos os outros nós. Para sermos concretos, vamos nos concentrar no nó D . A única maneira de chegar a ele é pelos seus predecessores B ou C , portanto para encontrarmos o caminho mínimo para D , precisamos apenas comparar estas duas rotas:

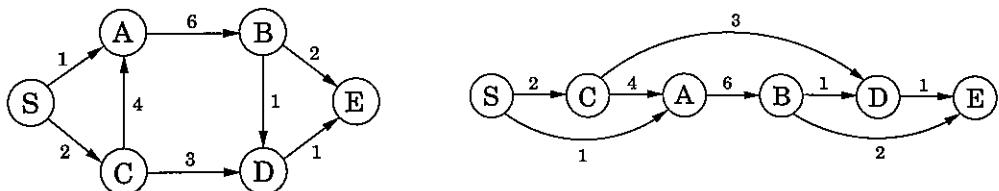
$$\text{dist}(D) = \min\{\text{dist}(B) + 1, \text{dist}(C) + 3\}.$$

Uma relação similar pode ser escrita para qualquer nó. Se computarmos os valores de dist na ordem da esquerda para a direita da Figura 6.1, podemos sempre estar certos de que, quando chegarmos ao nó v , já teremos toda a informação necessária para computar $\text{dist}(v)$. Portanto, somos capazes de computar todas as distâncias em uma única passada:

```
inicializa todos os valores  $\text{dist}(\cdot)$  como  $\infty$ 
 $\text{dist}(s) = 0$ 
para cada  $v \in V \setminus \{s\}$ , em ordem linearizada:
 $\text{dist}(v) = \min_{(u, v) \in E} \{\text{dist}(u) + l(u, v)\}$ 
```

Note que esse algoritmo está resolvendo uma coleção de *subproblemas*, $\{\text{dist}(u) : u \in V\}$. Começamos com o menor deles, $\text{dist}(s)$, pois imediatamente sabemos que

Figura 6.1 Um dag e sua linearização (ordenação topológica).



sua resposta é 0. Depois procedemos com subproblemas progressivamente “maiores” — distâncias para vértices que estão mais e mais longe na linearização; consideramos um subproblema grande quando temos de resolver muitos outros subproblemas antes de poder chegar a ele.

Essa é uma técnica muito geral. Em cada nó, computamos alguma função dos valores dos predecessores do nó. No nosso caso a função é uma soma de mínimos, mas poderíamos da mesma forma fazê-la uma soma de máximos, caso no qual obteríamos os caminhos *mais longos* no dag. Ou poderíamos usar um produto em vez de uma soma dentro dos parênteses, caso no qual terminaríamos computando o caminho com o menor produto dos comprimentos de aresta.

Programação dinâmica é um paradigma algorítmico muito poderoso no qual um problema é resolvido identificando-se uma coleção de subproblemas e lidando com eles um por um, primeiro os menores, usando as respostas aos problemas menores para ajudar a descobrir as respostas aos maiores, até que toda a coleção esteja solucionada. Em programação dinâmica não é dado um dag: o dag está *implícito*. Seus nós são os subproblemas que definimos e suas arestas são as dependências entre subproblemas: se para resolver o subproblema B precisamos da resposta ao subproblema A , então existe uma aresta (conceitual) de A para B . Nesse caso, A é considerado um subproblema menor do que B — e será sempre menor, em um sentido óbvio.

Mas é hora de vermos um exemplo.

6.2 Subseqüência crescente mais longa

No problema da *subseqüência crescente mais longa*, a entrada é uma seqüência de números a_1, \dots, a_n . Uma *subseqüência* é qualquer subconjunto desses números tomados em ordem, da forma, $a_{i1}, a_{i2}, \dots, a_{ik}$ onde $1 \leq i_1 < i_2 < \dots < i_k \leq n$, e uma *subseqüência crescente* é aquela na qual os números vão ficando estritamente maiores. A tarefa é encontrar a subseqüência crescente de maior comprimento. Por exemplo, a subseqüência crescente mais longa de 5, 2, 8, 6, 3, 6, 9, 7 é 2, 3, 6, 9:

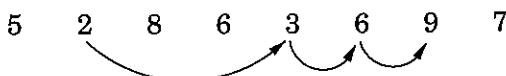
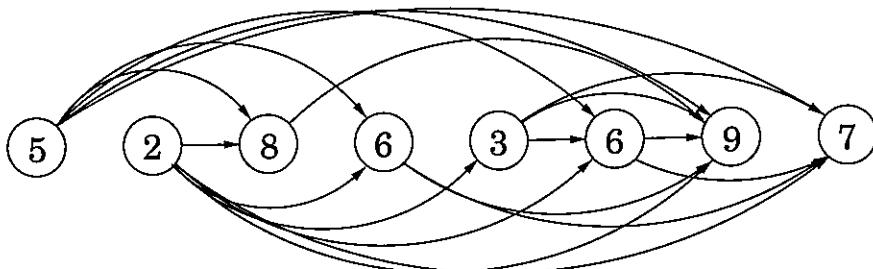


Figura 6.2 O dag das subseqüências crescentes.



Nesse exemplo, as setas denotam transições entre elementos consecutivos da solução ótima. De forma mais geral, para entender melhor o espaço de soluções, vamos criar um grafo de *todas* as transições permissíveis: crie um nó i para cada elemento a_i e adicione arestas (i, j) sempre que for possível para a_i e a_j serem elementos consecutivos em uma subseqüência crescente, ou seja, sempre que $i < j$ e $a_i < a_j$ (Figura 6.2).

Note que (1) este grafo $G = (V, E)$ é um dag, pois todas as arestas (i, j) têm $i < j$, e (2) existe uma correspondência um-para-um entre as subseqüências crescentes e os caminhos nesse dag. Portanto, nosso objetivo é simplesmente encontrar o caminho mais longo no dag!

Aqui está o algoritmo:

```

para  $j = 1, 2, \dots, n$ :
     $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$ 
    retornar  $\max_j L(j)$ 
  
```

$L(j)$ é o tamanho do caminho mais longo — a subseqüência crescente mais longa — terminando em j (mais 1, pois, estritamente falando, precisamos contar nós no caminho, não arestas). Raciocinando da mesma maneira que fizemos para caminhos mínimos, vemos que qualquer caminho para o nó j tem de passar por um de seus predecessores e, portanto, $L(j)$ é 1 mais o máximo valor $L(\cdot)$ entre esses predecessores. Se não há aresta alguma chegando em j , tomamos o máximo sobre o conjunto vazio, zero. E a resposta final é o maior $L(j)$, pois qualquer posição de término é permitida.

Isso é programação dinâmica. Para resolvemos nosso problema original, definimos uma coleção de subproblemas $\{L(j) : 1 \leq j \leq n\}$ com a seguinte propriedade-chave que permite que eles sejam resolvidos em uma única passada:

(*) Existe uma ordenação para os subproblemas e uma relação que mostra como resolver um subproblema dadas as respostas para subproblemas “menores”, ou seja, subproblemas que aparecem antes nesta ordenação.

No nosso caso, cada subproblema é resolvido usando a relação

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\},$$

uma expressão que envolve somente subproblemas menores. Quanto tempo toma este passo? Ele requer que os predecessores de j sejam conhecidos; para isso a lista de adjacência do grafo reverso G^R , possível de construir em tempo linear (reveja o Exercício 3.5), é útil. A computação de $L(j)$ toma tempo proporcional ao grau de entrada de j , levando a um tempo de execução total linear em $|E|$. Isto é no máximo $O(n^2)$, o máximo sendo quando o vetor de entrada está ordenado em ordem crescente. Assim, a solução de programação dinâmica é tanto simples, como eficiente.

Existe uma última questão a ser esclarecida: os valores de L somente nos dizem o *tamanho* da subseqüência ótima, desse modo, como recuperamos a própria subseqüência? Podemos resolver facilmente com o mesmo dispositivo de apontadores que usamos para caminhos mínimos no Capítulo 4. Ao computarmos $L(j)$, devemos também anotar $\text{prev}(j)$, o próximo nó anterior no caminho mais longo até j . A subseqüência ótima pode, então, ser reconstruída seguindo-se esses ponteiros.

6.3 Distância de edição

Quando o verificador de ortografia encontra um possível erro, ele procura no seu dicionário por outras palavras que estão próximas. Qual é o conceito apropriado de proximidade neste caso?

Uma medida natural da distância entre duas *strings* é a extensão segundo a qual elas podem ser *alinhas*, ou emparelhadas. Tecnicamente, um alinhamento é simplesmente uma maneira de escrever as *strings* uma sobre a outra. Por exemplo, aqui estão dois possíveis alinhamentos de SNOWY e SUNNY:

S	-	N	O	W	Y
S	U	N	N	-	Y

Custo: 3

-	S	N	O	W	-	Y
S	U	N	-	-	N	Y

Custo: 5

O “-” indica um “vazio”; qualquer número desses pode ser colocado em qualquer uma das *strings*. O custo de um alinhamento é o número de colunas nas quais as letras diferem. E a *distância de edição* entre duas *strings* é o custo do melhor alinhamento possível. Você pode ver que não existe nenhum alinhamento melhor de SNOWY e SUNNY do que aquele mostrado aqui com custo 3?

A distância de edição é assim chamada porque pode ser imaginada como o número mínimo de *edições* — inserções, remoções e substituições de caracteres — necessárias para transformar a primeira *string* na segunda. Por exemplo, o alinhamento mostrado na esquerda corresponde a três edições: inserir U, substituir O → N e remover W.

Em geral, existem tantos possíveis alinhamentos entre duas *strings* que seria terrivelmente ineficiente buscar entre todas elas pela melhor. Em vez disso, voltemos para programação dinâmica.

Uma solução por programação dinâmica

Ao resolver um problema por programação dinâmica, a questão mais crucial é *quais são os subproblemas*? Contanto que eles sejam escolhidos de forma que tenham

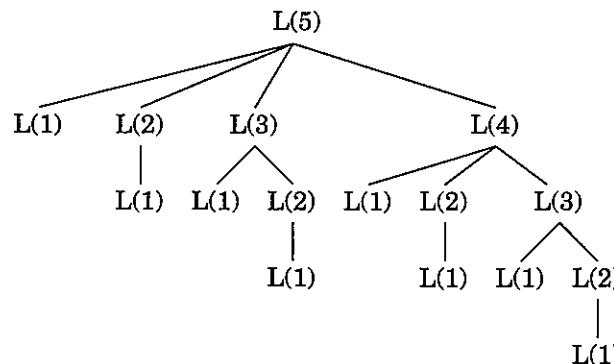
Recursão? Não, obrigado

Retornando à nossa discussão sobre a subseqüência crescente mais longa: a fórmula para $L(j)$ também sugere um algoritmo alternativo, recursivo. Não seria isso ainda mais simples?

Na verdade, recursão é uma idéia muito ruim: o procedimento resultante iria requerer tempo exponencial! Para entender por que, suponha que o dag contenha arestas (i, j) para todo $i < j$ — isto é, a dada seqüência de números a_1, a_2, \dots, a_n é ordenada. Nesse caso, a fórmula para o subproblema $L(j)$ torna-se

$$L(j) = 1 + \max\{L(1), L(2), \dots, L(j-1)\}.$$

A figura seguinte desdobra a recursão para $L(5)$. Note que os mesmos subproblemas são resolvidos repetidas vezes!



Para $L(n)$ essa árvore tem um número exponencial de nós (você pode dar uma cota?) e, portanto, a solução recursiva é desastrosa.

Então por que recursão funcionou tão bem com divisão-e-conquista? O ponto-chave é que, em divisão-e-conquista, um problema é expresso em termos de subproblemas que são *substancialmente menores*, digamos metade do tamanho. Por exemplo, mergesort ordena um vetor de tamanho n recursivamente ordenando dois subvetores de tamanho $n/2$. Em virtude da queda brusca no tamanho do problema, a árvore de recursão inteira tem apenas profundidade logarítmica e um número polinomial de nós.

Em contraste, em uma formulação típica de programação dinâmica, um problema é reduzido a subproblemas que são apenas ligeiramente menores — por exemplo, $L(j)$ se baseia em $L(j-1)$. Portanto a árvore de recursão inteira em geral tem profundidade polinomial e um número exponencial de nós. Entretanto, acontece que muitos desses nós são repetições, não existem muitos subproblemas *distintos* entre eles. Eficiência é, assim, obtida explicitamente enumerando os subproblemas e resolvendo-os na ordem correta.

Programação?

A origem do termo *programação dinâmica* tem muito pouco a ver com escrever código. Ele foi cunhado originalmente por Richard Bellman nos anos de 1950, um tempo em que programação de computadores era uma atividade esotérica praticada por tão pouca gente que nem merecia um nome. Naquela época, programação significava “planejamento” e “programação dinâmica” foi concebida para planejar otimizadamente processos multiestágio. O dag da Figura 6.2 pode ser imaginado como descrevendo os possíveis caminhos nos quais um tal processo pode evoluir: cada nó denota um estado, o nó mais à esquerda é o ponto de partida e as arestas saindo de um estado representam possíveis ações, levando a diferentes estados na próxima unidade de tempo.

A etimologia de *programação linear*, o assunto do Capítulo 7, é similar.

a propriedade (*) da página 158, escrever o algoritmo é uma questão simples: resolva iterativamente um subproblema depois do outro, em ordem crescente de tamanho.

Nosso objetivo é encontrar a distância de edição entre duas *strings* $x[1 \dots m]$ e $y[1 \dots n]$. O que seria um bom subproblema? Bem, ele deveria ser parte do caminho para resolver o problema inteiro, portanto, que tal observar a distância de edição entre algum *prefixo* da primeira string, $x[1 \dots i]$, e algum *prefixo* da segunda, $y[1 \dots j]$? Chame esse subproblema de $E(i, j)$ (veja a Figura 6.3). Nossa objetivo final, então, é computar $E(m, n)$.

Para isso funcionar, precisamos de alguma forma expressar $E(i, j)$ em termos de subproblemas menores. Vejamos — o que sabemos sobre o melhor alinhamento entre $x[1 \dots i]$ e $y[1 \dots j]$? Bem, sua coluna mais à direita pode ser apenas uma de três coisas:

$$\begin{array}{ccc} x[i] & \text{ou} & x[i] \\ & - & \\ & y[j] & \text{ou} \\ & - & y[j] \end{array}$$

O primeiro caso incorre em um custo de 1 para essa particular coluna e resta alinhar $x[1 \dots i - 1]$ com $y[1 \dots j]$. Mas isso é exatamente o subproblema $E(i - 1, j)$! Parece que estamos chegando a algum lugar. No segundo caso, também com custo 1, ainda precisamos alinhar $x[1 \dots i]$ com $y[1 \dots j - 1]$. Isso é novamente um subproblema, $E(i, j - 1)$. E no caso final, que ou custa 1 (se $x[i] \neq y[j]$) ou 0 (se $x[i] = y[j]$), o que resta é o subproblema $E(i - 1, j - 1)$. Em resumo, expressamos $E(i, j)$ em termos de três subproblemas

Figura 6.3 O subproblema $E(7, 5)$.

E	X	P	O	N	E	N	T	I	A	L
P	O	L	Y	N	O	M	I	A	L	

menores $E(i - 1, j)$, $E(i, j - 1)$, $E(i - 1, j - 1)$. Não temos qualquer idéia sobre qual dos três é o correto, portanto precisamos tentar todos eles e selecionar o melhor:

$$E(i, j) = \min\{1 + E(i - 1, j), 1 + E(i, j - 1), \text{dif}(i, j) + E(i - 1, j - 1)\}$$

onde por conveniência $\text{dif}(i, j)$ é definido como 0 se $x[i] = y[j]$ e 1, caso contrário.

Por exemplo, na computação da distância de edição entre EXPONENTIAL e POLYNOMIAL, o subproblema $E(4, 3)$ corresponde aos prefixos EXPO e POL. A coluna mais à direita de seu melhor alinhamento tem de ser uma das seguintes:

O	— ou —	— ou —	O
—	L	L	—

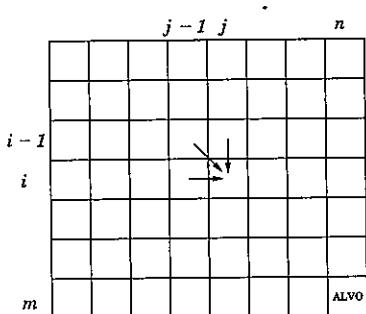
Assim, $E(4, 3) = \min\{1 + E(3, 3), 1 + E(4, 2), 1 + E(3, 2)\}$.

As respostas para todos os subproblemas $E(i, j)$ formam uma tabela bidimensional, como na Figura 6.4. Em que ordem os subproblemas devem ser resolvidos? Qualquer ordem é válida, desde que $E(i - 1, j)$, $E(i, j - 1)$, e $E(i - 1, j - 1)$ sejam considerados antes de $E(i, j)$. Por exemplo, poderíamos preencher a tabela uma linha por vez, da linha superior para a linha inferior e movendo da esquerda para a direita em cada linha. Ou, alternativamente, poderíamos preenchê-la coluna por coluna. Ambos os métodos asseguram que no momento em que vamos computar uma particular célula da tabela, todas as outras células de que precisamos já estão computadas.

Tanto os subproblemas quanto a ordem especificada estão quase prontos. Resta ainda apenas os “casos-base” da programação dinâmica, os menores subproblemas. Na situação presente, eles são $E(0, \cdot)$ e $E(\cdot, 0)$, ambos podem ser facilmente resolvidos.

Figura 6.4 (a) A tabela de subproblemas. Células $E(i - 1, j - 1)$, $E(i - 1, j)$ e $E(i, j - 1)$ são necessárias para preencher $E(i, j)$. (b) A tabela de valores final encontrada pela programação dinâmica.

(a)



(b)

	P	O	L	I	N	O	M	I	A	L
E	0	1	2	3	4	5	6	7	8	9
X	1	1	2	3	4	5	6	7	8	9
Z	2	2	2	3	4	5	6	7	8	9
P	3	2	3	3	4	5	6	7	8	9
O	4	3	2	3	4	5	5	6	7	8
N	5	4	3	3	4	4	5	6	7	8
E	6	5	4	4	4	5	5	6	7	8
N	7	6	5	5	5	4	5	6	7	8
C	8	7	6	6	6	5	5	6	7	8
I	9	8	7	7	7	6	6	6	7	8
A	10	9	8	8	8	7	7	7	6	7
L	11	10	9	8	9	8	8	8	7	6

$E(0, j)$ é a distância de edição entre o prefixo de x de tamanho 0, ou seja, a string vazia, e as primeiras j letras de y : claramente, j . E similarmente, $E(i, 0) = i$.

Nesse momento, o algoritmo para distância de edição surge basicamente sem esforço.

```

para i = 0, 1, 2, ..., m:
    E(i, 0) = i
para j = 1, 2, ..., n:
    E(0, j) = j
para i = 1, 2, ..., m:
    para j = 1, 2, ..., n:
        E(i, j) = min{E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + dif(i, j)}
retornar E(m, n)

```

Esse procedimento preenche a tabela linha por linha e da esquerda para a direita dentro de cada linha. Cada célula toma tempo constante no preenchimento, portanto o tempo total de execução é simplesmente o tamanho da tabela, $O(mn)$.

E no nosso exemplo, a distância de edição acaba sendo 6:

E	X	P	O	N	E	N	-	T	I	A	L
-	-	P	O	L	Y	N	0	M	I	A	L

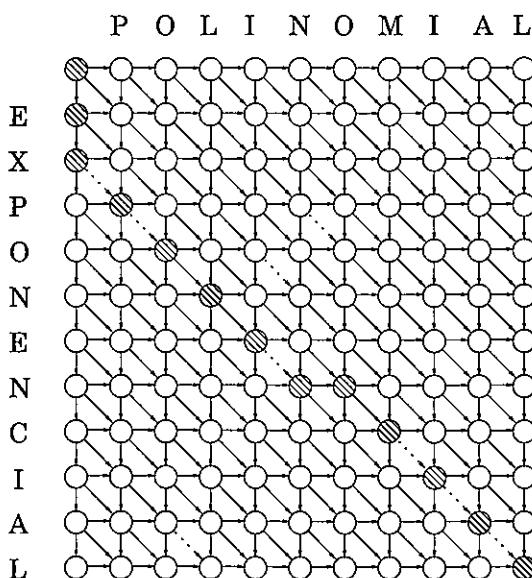
O dag subjacente

Todo programa dinâmico tem uma estrutura de dag subjacente: imagine cada nó como representando um subproblema e cada aresta como uma restrição de precedência na ordem segundo a qual os subproblemas devem ser manipulados. Ter nós u_1, \dots, u_k apontando para v significa “subproblema v pode ser resolvido desde que as respostas para u_1, \dots, u_k sejam conhecidas”.

Na nossa aplicação de distância de edição, os nós do dag subjacente correspondem aos subproblemas ou, equivalentemente, às posições (i, j) na tabela. Suas arestas são restrições de precedência, da forma $(i - 1, j) \rightarrow (i, j)$, $(i, j - 1) \rightarrow (i, j)$, e $(i - 1, j - 1) \rightarrow (i, j)$ (Figura 6.5). De fato, podemos levar as coisas mais a diante e colocar pesos nas arestas de modo que as distâncias de edição sejam dadas por caminhos mínimos no dag! Para tanto, faça todos os comprimentos de arestas iguais a 1, exceto para $\{(i - 1, j - 1) \rightarrow (i, j) : x[i] = y[j]\}$ (mostrado em pontilhado na figura), cujo tamanho é 0. A resposta final é, então, simplesmente a distância entre os nós $s = (0, 0)$ e $t = (m, n)$. Um possível caminho mínimo é mostrado, aquele que gera o alinhamento que encontramos antes. Nesse caminho, cada movimento para baixo é uma remoção, cada movimento para a direita é uma inserção e cada movimento diagonal é ou um emparelhamento ou uma substituição.

Alterando os pesos nesse dag, podemos permitir formas generalizadas de distâncias de edição, nas quais inserções, remoções e substituições têm custos associados diferentes.

Figura 6.5 O dag subjacente e um caminho de comprimento 6.



6.4 Mochila

Durante um roubo, o ladrão encontra muito mais objetos do que esperava, e tem de decidir o que levar. Sua bolsa (ou “mochila”) pode carregar um peso total de no máximo W quilos. Existem n itens entre os quais escolher, de pesos w_1, \dots, w_n e valor em reais v_1, \dots, v_n . Qual a combinação mais valiosa de itens que ele pode colocar na sua mochila?¹

Por exemplo, tome $W = 10$ e

Item	Peso	Valor
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

¹ Se essa aplicação parece fútil, substitua “peso” por “tempo de CPU” e “no máximo W quilos” por “no máximo W unidades de tempo de CPU”. Ou use “banda” no lugar de “tempo de CPU” etc. O problema da mochila generaliza uma ampla variedade de tarefas de seleção com restrição de recurso.

Subproblemas comuns

Encontrar os subproblemas certos demanda criatividade e experimentação. Mas há algumas poucas escolhas-padrão que aparecem repetidamente em programação dinâmica.

- i. A entrada é x_1, x_2, \dots, x_n e um subproblema é x_1, x_2, \dots, x_i .

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
-------	-------	-------	-------	-------	-------	-------	-------	-------	----------

O número de subproblemas é, portanto, linear.

- ii. A entrada é x_1, \dots, x_n e y_1, \dots, y_m . Um subproblema é x_1, \dots, x_i e y_1, \dots, y_j .

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
-------	-------	-------	-------	-------	-------	-------	-------	-------	----------

y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
-------	-------	-------	-------	-------	-------	-------	-------

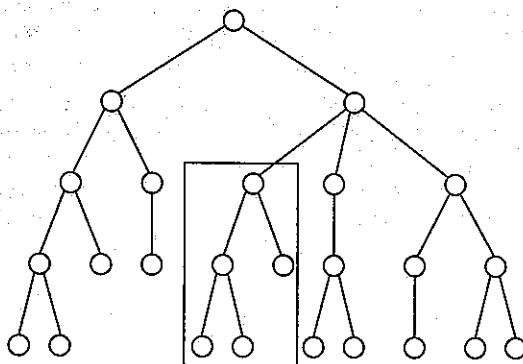
O número de subproblemas é $O(mn)$.

- iii. A entrada é x_1, \dots, x_n e um subproblema é x_i, x_{i+1}, \dots, x_j .

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
-------	-------	-------	-------	-------	-------	-------	-------	-------	----------

O número de subproblemas é $O(n^2)$.

- iv. A entrada é uma árvore enraizada. Um subproblema é uma subárvore enraizada.



Se a árvore tem n nós, quantos subproblemas existem?

Já encontramos os dois primeiros casos, os outros virão em breve.

De ratos e homens

Nossos corpos são máquinas extraordinárias: flexíveis em função, adaptáveis a novos ambientes e capazes de interagir e reproduzir. Todas essas capacidades são especificadas por um programa único para cada um de nós, uma *string* que tem 3 bilhões de caracteres de comprimento sobre o alfabeto {A, C, G, T} — nosso DNA.

As seqüências de DNA de quaisquer duas pessoas diferem por apenas cerca de 0,1%. Entretanto, isso ainda deixa 3 milhões de posições nas quais elas variam, mais do que suficiente para explicar a vasta gama de diversidade humana. Essas diferenças são de grande interesse médico e científico — por exemplo, elas podem ajudar a explicar quais pessoas são propensas a certas doenças.

O DNA é um vasto e aparentemente inescrutável programa, mas ele pode ser dividido em unidades menores, mais específicas no seu papel, semelhantes a sub-rotinas. São chamadas de *genes*. Os computadores tornaram-se uma ferramenta crucial no entendimento dos genes de humanos e outros organismos, tanto que *genómica computacional* é agora uma área em si mesma. Veja exemplos de questões típicas.

1. Quando um novo gene é descoberto, uma maneira de obter conhecimento sobre sua função é encontrar genes conhecidos que melhor se alinhavam com ele. Isso é particularmente útil na transferência de conhecimento de espécies bem-estudadas, como ratos, para seres humanos.

Uma primitiva básica no problema de busca é definir uma noção computacional eficiente de quando duas *strings* se alinhavam aproximadamente. A biologia sugere uma generalização da distância de edição, e programação dinâmica pode ser usada para computá-la.

Depois há o problema de buscar entre um vasto banco de genes conhecidos: o banco de dados GenBank já possui um comprimento total de mais de 10^{10} , e este número está crescendo rapidamente. O melhor método atual é o BLAST, uma combinação inteligente de truques algorítmicos e intuições biológicas que o fizeram o software mais usado em biologia computacional.

2. Métodos para *seqüenciar* DNA (determinar a *string* de caracteres que o constitui) tipicamente encontram somente fragmentos de 500 a 700 caracteres. Bilhões desses fragmentos espalhados aleatoriamente podem ser gerados, mas de que maneira podem ser montados como uma seqüência de DNA coerente? Por um problema, a posição de qualquer um desses fragmentos na seqüência final é desconhecida e tem de ser inferida juntando-se fragmentos que se sobreponham.

Um exemplo desses esforços é o rascunho do DNA humano completado em 2001 por dois grupos simultaneamente: o fundo público Human Genome Consortium e o privado Celera Genomics.

3. Quando um particular gene foi seqüenciado em cada uma de várias espécies, será que esta informação pode ser usada para reconstruir a história evolucionária dessas espécies?

Exploraremos esses problemas nos exercícios no final do Capítulo. Programação dinâmica se revelou uma ferramenta valiosa para alguns deles e para biologia computacional em geral.

Há duas versões desse problema. Se, por um lado, existem quantidades ilimitadas disponíveis de cada item, a escolha ótima é selecionar o item 1 e dois do item 4 (total: \$48). Por outro lado, se há apenas um de cada item (o ladrão invadiu uma galeria de arte, digamos), então a mochila ótima contém os itens 1 e 3 (total: \$46).

Como veremos no Capítulo 8, nenhuma das versões desse problema provavelmente tem um algoritmo polinomial. Entretanto, usando programação dinâmica, elas podem ser resolvidas em tempo $O(nW)$, o que é razoável quando W é pequeno, mas não é polinomial, pois o tamanho da entrada é proporcional a $\log W$ em vez de W .

Mochila com repetição

Vamos começar com a versão que permite repetições. Como sempre, a principal questão em programação dinâmica é: quais são os subproblemas? Nesse caso podemos reduzir o problema original de duas maneiras: examinar as mochilas de capacidade menor $w \leq W$, ou examinar menos itens (por exemplo, itens 1, 2, ..., j para $j \leq n$). Normalmente alguma experimentação é necessária para descobrir exatamente qual funciona.

A primeira restrição se refere a capacidades menores. De acordo com isso, defina

$$K(w) = \text{valor máximo alcançável por uma mochila de capacidade } w.$$

Podemos expressar em termos de subproblemas menores? Bem, se a solução ótima para $K(w)$ inclui o item i , então, ao removermos este item da mochila, resta-nos uma solução ótima para $K(w - w_i)$. Em outras palavras, $K(w)$ é simplesmente $K(w - w_i) + v_i$, para algum i . Não sabemos qual i , portanto temos de tentar todas as possibilidades.

$$K(w) = \max_{i: w_i \leq w} \{K(w - w_i) + v_i\},$$

onde, segundo nossa convenção, o máximo sobre um conjunto vazio é 0. Estamos prontos! O algoritmo agora surge por si mesmo e é caracteristicamente simples e elegante.

$$K(0) = 0$$

para $w = 1$ até W :

$$K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$$

retornar $K(W)$

Esse algoritmo preenche uma tabela unidimensional de comprimento $W + 1$, da esquerda para a direita. Cada célula pode tomar tempo até $O(n)$ para ser computada, portanto o tempo de execução total é $O(nW)$.

Como sempre, existe um dag subjacente. Tente construí-lo e você será recompensado com uma percepção surpreendente: esta particular variante da mochila se reduz a encontrar o caminho mais longo em um dag!

Mochila sem repetição

Vamos partir para a segunda variante: o que dizer se repetições não são permitidas? Nossos subproblemas anteriores agora se tornam completamente inúteis. Por exemplo, saber que o valor $K(w - w_n)$ é muito grande não nos ajuda, porque não sabemos se o

item n já foi usado ou não nesta solução parcial. Temos que, portanto, refinar nosso conceito de subproblema para contemplar informação adicional sobre os itens em uso. Adicionamos um segundo parâmetro, $0 \leq j \leq n$:

$K(w, j) =$ valor máximo alcançável usando uma mochila de capacidade w e itens $1, \dots, j$.

A resposta que procuramos é, então, $K(W, n)$.

Como podemos expressar um subproblema $K(w, j)$ em termos de subproblemas menores? Muito simples: ou o item j é necessário para alcançar o valor ótimo, ou não é:

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}.$$

(O primeiro caso é invocado somente se $w_j \leq w$.) Em outras palavras, podemos expressar $K(w, j)$ em termos de subproblemas $K(\cdot, j - 1)$.

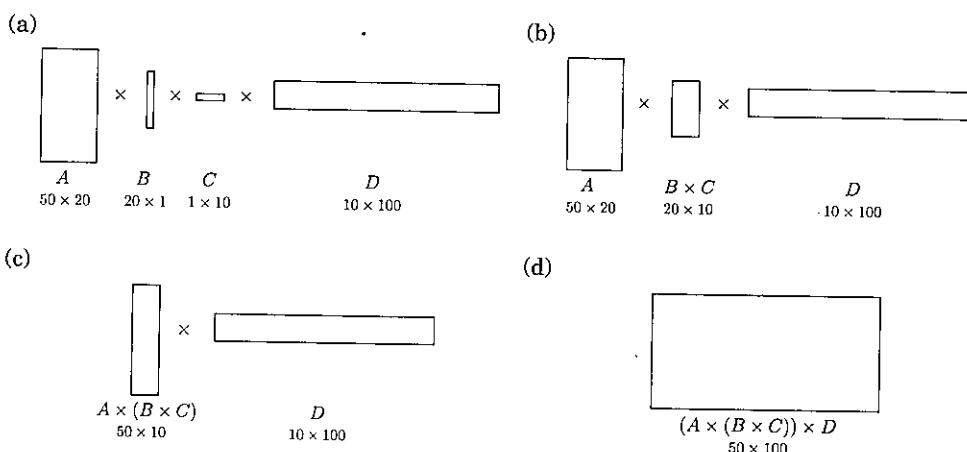
O algoritmo consiste então em preencher uma tabela bidimensional, com $W + 1$ linhas e $n + 1$ colunas. Cada célula da tabela toma tempo apenas constante, assim, muito embora a tabela seja muito maior do que no caso anterior, o tempo de execução permanece o mesmo, $O(nW)$. Veja o código.

```
Iniciar todo  $K(0, j) = 0$  e todo  $K(w, 0) = 0$ 
para  $j = 1$  até  $n$ :
    para  $w = 1$  até  $W$ :
        se  $w_j > w$ :  $K(w, j) = K(w, j - 1)$ 
        senão:  $K(w, j) = \max\{K(w, j - 1), K(w - w_j, j - 1) + v_j\}$ 
    retornar  $K(W, n)$ 
```

6.5 Multiplicação de cadeias de matrizes

Suponha que queiramos multiplicar quatro matrizes, $A \times B \times C \times D$, de dimensões 50×20 , 20×1 , 1×10 e 10×100 , respectivamente (Figura 6.6). Isso envolve multiplicar iterativamente duas matrizes por vez. Multiplicação de matrizes não é *comutativa*

Figura 6.6 $A \times B \times C \times D = (A \times (B \times C)) \times D$.



Memorização

Em programação dinâmica, escrevemos fórmulas recursivas que expressam problemas grandes em termos de problemas menores, e as usamos para preencher uma tabela de valores de soluções de baixo para cima, do menor para o maior subproblema.

A fórmula também sugere um algoritmo recursivo, mas vimos antes que recursão ingênua pode ser terrivelmente inefficiente, porque ela resolve os mesmos subproblemas repetidas vezes. O que dizer de uma implementação recursiva mais inteligente, uma que se lembre de suas invocações anteriores e, com isso, evita repeti-las?

No problema da mochila (com repetições), um tal algoritmo usaria uma tabela de espalhamento (reveja a Seção 1.5) para guardar os valores de $K(\cdot)$ que já foram computados. Em cada chamada recursiva requerendo algum $K(w)$, o algoritmo primeiro verificaria se a resposta já está na tabela e, então, procederia a sua computação somente se não estivesse. Esse recurso é chamado de *memorização (memoization)*:

Uma tabela de espalhamento, inicialmente vazia, contém os valores de $K(w)$ indexados por w

```
função mochila( $w$ )
se  $w$  está na tabela de espalhamento: retornar  $K(w)$ 
 $K(w) = \max\{\text{mochila}(w - w_i) + v_i : w_i \leq w\}$ 
inserir  $K(w)$  na tabela de espalhamento, com chave  $w$ 
retornar  $K(w)$ 
```

Como esse algoritmo nunca repete um subproblema, seu tempo de execução é $O(nW)$, assim como o programa dinâmico. Entretanto, o fator constante na notação O é substancialmente maior devido ao custo das chamadas recursivas.

Em alguns casos, entretanto, memorização vale a pena. Veja o porquê: programação dinâmica automaticamente resolve todo subproblema concebível, enquanto memorização resolve apenas aqueles realmente usados. Por exemplo, suponha que W e todos os pesos w_i sejam múltiplos de 100. Então um subproblema $K(w)$ é inútil se 100 não divide w . O algoritmo recursivo memorizado nunca irá olhar para essas células que não têm importância.

(em geral, $A \times B \neq B \times A$), mas *associativa*, o que significa, por exemplo, que $A \times (B \times C) = (A \times B) \times C$. Portanto podemos computar o nosso produto de quatro matrizes de muitas maneiras diferentes, dependendo de como organizamos os parênteses. Será que algumas dessas maneiras são melhores do que outras?

Multiplicar uma matriz $m \times n$ por uma matriz $n \times p$ toma mnp multiplicações, em uma aproximação suficiente. Usando essa fórmula, vamos comparar várias maneiras diferentes de avaliar $A \times B \times C \times D$:

Organização de parênteses	Computação do custo	Custo
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120.200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60.200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7.000

Como você pode ver, a ordem da multiplicação faz uma grande diferença no tempo de execução final! Além disso, a abordagem *gulosa* natural, de sempre realizar a multiplicação de matrizes mais barata disponível, leva à segunda organização de parênteses mostrada aqui e é, portanto, um fracasso.

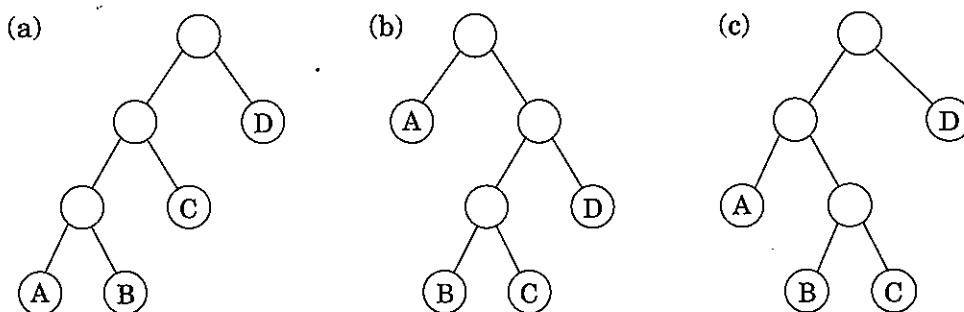
Como determinamos a ordem ótima, se queremos computar $A_1 \times A_2 \times \cdots \times A_n$, onde os A_i são matrizes com dimensões $m_0 \times m_1, m_1 \times m_2, \dots, m_{n-1} \times m_n$, respectivamente? A primeira coisa a notar é que uma particular organização de parênteses pode ser representada muito naturalmente por uma árvore binária na qual as matrizes individuais correspondem a folhas, a raiz é o produto final e nós interiores são produtos intermediários (Figura 6.7). As possíveis ordens, nas quais fazer a multiplicação, correspondem às várias árvores binárias cheias com n folhas, cujo número é exponencial em n (Exercício 2.13). Certamente não podemos tentar cada árvore e, com a força bruta está descartado, nos voltamos para programação dinâmica.

As árvores binárias da Figura 6.7 são sugestivas: para uma árvore ser ótima, suas subárvores também têm de ser ótimas. Quais são os subproblemas correspondentes às subárvores? Eles são produtos da forma $A_i \times A_{i+1} \times \cdots \times A_j$. Vamos ver se isso funciona: para $1 \leq i \leq j \leq n$, defina

$$C(i, j) = \text{custo mínimo de multiplicar } A_i \times A_{i+1} \times \cdots \times A_j.$$

O tamanho desse subproblema é o número de multiplicações de matrizes, $|j - i|$. O menor subproblema é quando $i = j$, caso em que não há nada a multiplicar, portanto $C(i, i) = 0$. Para $j > i$, considere a subárvore ótima para $C(i, j)$. A primeira ramificação nessa subárvore, aquela no topo, vai dividir o produto em duas partes, da forma $A_i \times \cdots \times A_k$ e $A_{k+1} \times \cdots \times A_j$, para algum k entre i e j . O custo da subárvore é, então, o custo destes dois produtos parciais, mais o custo de combiná-los: $C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j$. E precisamos apenas encontrar o ponto de

Figura 6.7 (a) $((A \times B) \times C) \times D$; (b) $A \times ((B \times C) \times D)$; (c) $(A \times (B \times C)) \times D$.



divisão k para o qual isso é mínimo:

$$C(i, j) = \min_{i \leq k < j} \{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j\}.$$

Estamos prontos para codificar! No que se segue, a variável s denota o tamanho do subproblema.

```

para  $i = 1$  até  $n$ :  $C(i, i) = 0$ 
para  $s = 1$  até  $n - 1$ :
    para  $i = 1$  até  $n - s$ :
         $j = i + s$ 
         $C(i, j) = \min\{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j : i \leq k < j\}$ 
    retornar  $C(1, n)$ 
```

Os subproblemas constituem uma tabela bidimensional, cujas células tomam cada uma tempo $O(n)$ para serem computadas. O tempo de execução total é, assim, $O(n^3)$.

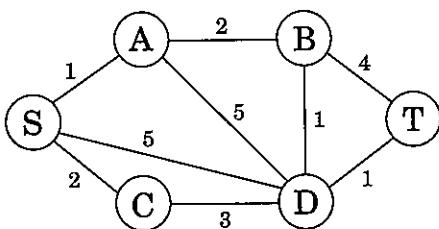
6.6 Caminhos mínimos

Começamos este capítulo com um algoritmo de programação dinâmica para a tarefa elementar de encontrar o caminho mínimo em um dag. Voltemos agora para problemas de caminho mínimo mais sofisticados e vejamos como também podem ser contemplados pela nossa técnica algorítmica poderosa.

Caminhos confiáveis mínimos

A vida é complicada e abstrações como grafos, comprimentos de arestas e caminhos mínimos raramente capturam toda a verdade. Em uma rede de comunicações, por exemplo, mesmo que os comprimentos de aresta fielmente reflitam os atrasos de transmissão, pode haver outras considerações envolvidas na escolha de um caminho. Por exemplo, cada aresta extra no caminho pode ser uma “preocupação” extra com as incertezas e perigos da perda de pacotes. Em tais casos, gostaríamos de evitar caminhos com arestas demais. A Figura 6.8 ilustra esse problema com um grafo no qual o caminho mínimo de S até T tem quatro arestas, enquanto existe um outro caminho que é um pouco mais longo, mas usa apenas duas arestas. Se quatro arestas traduzem-se em uma falta de confiabilidade proibitiva, poderíamos ter de escolher o segundo caminho.

Figura 6.8 Queremos um caminho de S até T que seja curto e tenha poucas arestas.



Suponha, então, que seja dado um grafo G com comprimentos nas arestas, juntamente com dois nós s e t e um inteiro k , e queiramos o caminho mínimo de s até t que use no máximo k arestas.

Será que existe uma maneira rápida de adaptar o algoritmo de Dijkstra para esta nova tarefa? Não exatamente: aquele algoritmo concentra-se no comprimento de cada caminho mínimo sem “se lembrar” do número de passos no caminho, o que é agora uma informação crucial.

Em programação dinâmica, o truque é escolher subproblemas de modo que toda a informação vital seja recordada e levada adiante. Neste caso, vamos definir, para cada vértice v e cada inteiro $i \leq k$, $\text{dist}(v, i)$ como o comprimento do menor caminho de s até v que usa i arestas. Os valores iniciais de $\text{dist}(v, 0)$ são ∞ para todos os vértices exceto s , para o qual é 0. E a equação geral de atualização é, como esperado naturalmente,

$$\text{dist}(v, i) = \min_{(u, v) \in E} \{\text{dist}(u, i - 1) + \ell(u, v)\}.$$

Precisamos dizer alguma coisa mais?

Caminhos mínimos para todos os pares

O que dizer se queremos encontrar o caminho mínimo não apenas entre s e t , mas entre *todos* os pares de vértices? Uma abordagem seria executar o algoritmo de caminhos mínimos geral da Seção 4.6.1 (pois pode haver arestas negativas) $|V|$ vezes, uma vez para cada nó inicial. O tempo de execução total seria, então, $O(|V|^2|E|)$. Veremos agora uma alternativa melhor, o algoritmo de tempo $O(|V|^3)$ baseado em programação dinâmica de *Floyd-Warshall*.

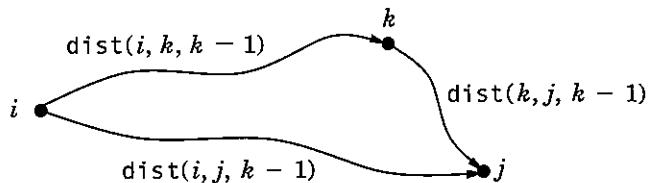
Será que há um bom subproblema para computar distâncias entre todos os pares de vértices em um grafo? Simplesmente resolver o problema para mais e mais pares ou pontos iniciais não ajuda, pois leva direto ao algoritmo $O(|V|^2|E|)$.

Uma idéia vem à mente: o caminho mínimo $u \rightarrow w_1 \rightarrow \dots \rightarrow w_l \rightarrow v$ entre u e v usa algum número de nós intermediários — possivelmente nenhum. Suponha que proibamos nós intermediários. Então podemos resolver os caminhos mínimos para todos os pares de uma só vez: o caminho mínimo entre u e v é a aresta direcionada (u, v) , se ela existe. O que dizer se agora gradualmente expandirmos o conjunto de nós intermediários permitidos? Podemos fazer isso, um nó por vez, atualizando os caminhos mínimos em cada estágio. Em algum momento, o conjunto torna-se o V inteiro, ponto no qual todos os vértices são permitidos em todos os caminhos e encontramos os caminhos mínimos reais entre vértices do grafo!

Mais concretamente, numere os vértices em V como $\{1, 2, \dots, n\}$, e denote por $\text{dist}(i, j, k)$ o comprimento do caminho mínimo de i até j no qual apenas os nós $\{1, 2, \dots, k\}$ podem ser usados como intermediários. Inicialmente, $\text{dist}(i, j, 0)$ é o comprimento entre a aresta direcionada entre i e j , se ela existe, e é ∞ , caso contrário.

O que acontece quando expandimos o conjunto intermediário para incluir um nó extra k ? Temos de reexaminar todos os pares i, j e verificar se usar k como ponto intermediário nos dá um caminho menor de i para j . Isso é fácil: um caminho mínimo de i até j que usa k junto com possivelmente outros nós intermediários de menor número passa por k apenas uma vez (por quê? porque assumimos que não existem ciclos nega-

tivos). E já calculamos o comprimento do caminho mínimo de i para k e de k para j usando apenas vértices de menor número:



Assim, usar k nos dá um caminho menor de i até j se e somente se

$$\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1) < \text{dist}(i, j, k - 1),$$

caso no qual $\text{dist}(i, j, k)$ deve ser atualizado de acordo.

Aqui está o algoritmo de Floyd-Warshall — e como você pode ver, ele toma tempo $O(|V|^3)$.

```

para  $i = 1$  até  $n$ :
    para  $j = 1$  até  $n$ :
         $\text{dist}(i, j, 0) = \infty$ 
    para todo  $(i, j) \in E$ :
         $\text{dist}(i, j, 0) = \ell(i, j)$ 
    para  $k = 1$  até  $n$ :
        para  $i = 1$  até  $n$ :
            para  $j = 1$  até  $n$ :
                 $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1), \text{dist}(i, j, k - 1)\}$ 

```

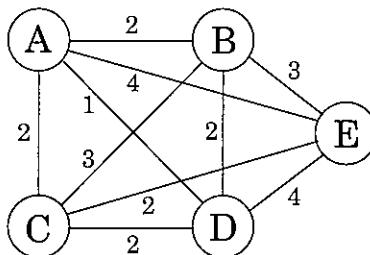
O problema do caixeiro-viajante

Um caixeiro-viajante está se preparando para uma grande jornada de vendas. Começando em sua cidade natal, maleta em mãos, ele vai conduzir uma jornada na qual cada uma das suas cidades-alvo será visitada exatamente uma vez antes que retorne para casa. Dadas as distâncias entre os pares de cidades, qual é a melhor ordem na qual visitá-las, para minimizar a distância total viajada?

Denote as cidades por $1, \dots, n$, a cidade natal do caixeiro-viajante sendo 1 e seja $D = (d_{ij})$ a matriz de distâncias entre as cidades. O objetivo é projetar uma jornada que comece e termine em 1, inclua todas as outras cidades exatamente uma vez e tenha comprimento total mínimo. A Figura 6.9 mostra um exemplo envolvendo cinco cidades. Você pode identificar a jornada ótima? Mesmo neste pequeno exemplo, é difícil para uma pessoa encontrar a solução; imagine o que acontece quando centenas de cidades estão envolvidas.

Acontece que esse problema também é difícil para computadores. De fato, o problema do caixeiro-viajante (TSP, do inglês *traveling salesman problem*) é uma das mais conhecidas tarefas computacionais. Há uma história longa de tentativas de resolvê-la, uma longa saga de fracassos e sucessos parciais e, ao longo do caminho, grandes avanços

Figura 6.9 A jornada ótima do caixeiro-viajante tem comprimento 10.



em algoritmos e teoria de complexidade. A notícia ruim sobre o TSP, que entenderemos melhor no Capítulo 8, é que é muito pouco provável que ele possa ser resolvido em tempo polinomial.

Quanto tempo ele toma, então? Bem, a abordagem de força bruta é avaliar todas as possíveis jornadas e retornar a melhor. Como há $(n - 1)!$ possibilidades, esta estratégia toma tempo $O(n!)$. Veremos agora que programação dinâmica leva a uma solução muito mais rápida, embora não polinomial.

Qual é o subproblema apropriado para o TSP? Subproblemas se referem a soluções parciais e, nesse caso, a solução parcial mais óbvia é a porção inicial da jornada. Suponha que começamos na cidade 1 como requerido, visitamos umas poucas cidades e estamos agora na cidade j . Que informação precisamos para estender essa jornada parcial? Certamente precisamos saber j , pois isso vai determinar quais as cidades mais convenientes para visitar depois. E precisamos também saber todas as cidades visitadas até agora, para não repetirmos nenhuma delas. Veja um subproblema apropriado:

Para um subconjunto de cidades $S \subseteq \{1, 2, \dots, n\}$ que inclui 1 e $j \in S$, seja $C(S, j)$ o comprimento do caminho mínimo visitando cada nó em S exatamente uma vez, começando em 1 e terminando em j .

Quando $|S| > 1$, definimos $C(S, 1) = \infty$, pois o caminho não pode começar e terminar em 1.

Agora, vamos expressar $C(S, j)$ em termos de subproblemas menores. Precisamos começar em 1 e terminar em j ; qual devemos escolher como a penúltima cidade? Ela tem de ser algum $i \in S$, tal que o comprimento total seja a distância de 1 até i , ou seja, $C(S - \{j\}, i)$, mais o comprimento da aresta final, d_{ij} . Temos de selecionar o melhor tal i :

$$C(S, j) = \min_{i \in S: i \neq j} C(S - \{j\}, i) + d_{ij}.$$

Sobre tempo e memória

A quantidade de tempo que leva para executar um algoritmo de programação dinâmica é facil de discernir a partir do dag de subproblemas: em muitos casos ele é simplesmente o número de arestas no dag! Tudo o que estamos realmente fazendo é visitar os nós na ordem linearizada, examinando as arestas que chegam a cada nó e, mais freqüentemente, fazendo uma quantidade constante de trabalho por aresta. No final, cada aresta do dag foi examinada uma vez.

Mas quanta memória de computador é requerida? Não há um parâmetro simples do dag caracterizando isso. É certamente possível fazer o trabalho com uma quantidade de memória proporcional ao número de vértices (subproblemas), mas podemos normalmente conseguir com muito menos. A razão é que o valor de um particular subproblema precisa ser lembrado somente até que os subproblemas maiores que dependem dele tenham sido resolvidos. Depois disso, a memória que ele usa pode ser liberada para reuso.

Por exemplo, no algoritmo de Floyd-Warshall o valor de $\text{dist}(i, j, k)$ não é necessário uma vez que os valores $\text{dist}(\cdot, \cdot, k+1)$ tenham sido computados. Portanto, precisamos apenas de dois vetores $|V| \times |V|$ para guardar os valores de dist , um para os valores ímpares de k e um para os valores pares: ao computarmos $\text{dist}(i, j, k)$, sobrescrevemos $\text{dist}(i, j, k-2)$.

(E não nos esqueçamos de que, como sempre em programação dinâmica, também precisamos de mais um vetor, $\text{prev}(i, j)$, guardando o penúltimo vértice no caminho mais curto atual de i até j , um valor que tem de ser atualizado com $\text{dist}(i, j, k)$. Omitimos este passo de contabilidade trivial, mas crucial dos nossos algoritmos de programação dinâmica.)

Você pode ver por que o dag da distância de edição na Figura 6.5 precisa apenas de memória proporcional ao comprimento da menor string?

Os subproblemas são ordenados por $|S|$. Veja o código.

```

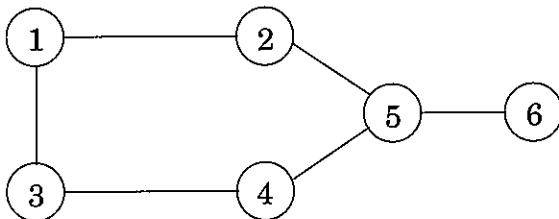
 $C(\{1\}, 1) = 0$ 
para  $s = 2$  até  $n$ :
    para todos os subconjuntos  $S \subseteq \{1, 2, \dots, n\}$  de tamanho  $s$  e con-
        tendo 1:
             $C(S, 1) = \infty$ 
            para todo  $j \in S, j \neq 1$ :
                 $C(S, j) = \min\{C(S - \{j\}, i) + d_{ij} : i \in S, i \neq j\}$ 
            retornar  $\min_j C(\{1, \dots, n\}, j) + d_{j1}$ 
```

Existem no máximo $2^n \cdot n$ subproblemas, e cada um é resolvido em tempo linear. O tempo de execução total é, portanto, $O(n^2 2^n)$.

6.7 Conjuntos independentes em árvores

Um subconjunto de nós $S \subseteq V$ é um *conjunto independente* de um grafo $G = (V, E)$ se não existem arestas entre eles. Por exemplo, na Figura 6.10 os nós $\{1, 5\}$ formam um conjunto independente, mas os nós $\{1, 4, 5\}$ não formam, por causa da aresta entre 4 e 5. O conjunto maior independente é $\{2, 3, 6\}$.

Figura 6.10 O maior conjunto independente neste grafo tem tamanho 3.



Como ocorre com vários outros problemas que vimos neste capítulo (mochila, caixeiro-viajante), acredita-se que encontrar o maior conjunto independente em um grafo é intratável. Entretanto, quando o grafo é uma *árvore*, o problema pode ser resolvido em tempo linear, usando programação dinâmica. E quais são os subproblemas apropriados? Já no problema da multiplicação de cadeia de matrizes notamos que a estrutura em níveis de uma árvore fornece uma definição natural de um subproblema — desde que um nó da árvore tenha sido identificado como uma raiz.

Aqui está o algoritmo: comece enraizando a árvore em qualquer nó r . Agora, cada nó define uma subárvore — aquela pendurada nele. Isso imediatamente sugere os subproblemas:

$$I(u) = \text{tamanho do maior conjunto independente da subárvore pendurada em } u.$$

Nosso objetivo final é $I(r)$.

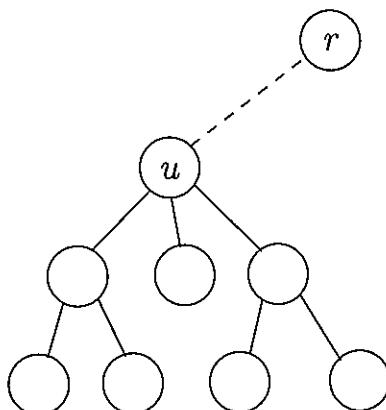
A programação dinâmica procede como sempre de subproblemas menores para os maiores, o que quer dizer de baixo para cima na árvore enraizada. Suponha que conheçamos os maiores conjuntos independentes para todas as subárvores debaixo de um certo nó u ; em outras palavras, suponha que conhecemos $I(w)$ para todos os descendentes w de u . Como podemos computar $I(u)$? Vamos dividir a computação em dois casos: qualquer conjunto independente ou inclui u ou não (Figura 6.11).

$$I(u) = \max \left\{ 1 + \sum_{\text{netos } w \text{ de } u} I(w), \sum_{\text{filhos } w \text{ de } u} I(w) \right\}.$$

Se, por um lado, o conjunto independente inclui u , ganhamos um ponto por ele, mas não podemos incluir os filhos de u — portanto prosseguimos com os netos. Esse é o primeiro caso na fórmula. Por outro lado, se não incluirmos u , não ganharemos um ponto por ele, mas podemos prosseguir com os seus filhos.

O número de subproblemas é exatamente o número de vértices. Com um pouco de cuidado, o tempo de execução pode ser linear, $O(|V| + |E|)$.

Figura 6.11 $I(u)$ é o tamanho do maior conjunto independente da subárvore enraizada em u . Dois casos: ou u está neste conjunto independente ou não está.



Exercícios

- 6.1. Uma *subseqüência contígua* de uma lista S é uma subseqüência feita de elementos consecutivos de S . Por exemplo, se S é

$$5, 15, -30, 10, -5, 40, 10,$$

então $15, -30, 10$ é uma subseqüência contígua, mas $5, 15, 40$ não é. Forneça um algoritmo de tempo linear para a seguinte tarefa:

Entrada: Uma lista de números a_1, a_2, \dots, a_n .

Saída: A subseqüência contígua de soma máxima (a subseqüência de tamanho zero tem soma zero).

Para o exemplo anterior, a resposta seria $10, -5, 40, 10$, com uma soma de 55.

(*Dica:* Para cada $j \in \{1, 2, \dots, n\}$ considere subseqüências contíguas terminando exatamente na posição j .)

- 6.2. Você vai sair em uma viagem longa, que começa na estrada no quilômetro 0. Ao longo do caminho existem n hotéis, nos quilômetros $a_1 < a_2 < \dots < a_n$, onde cada a_i é medido partindo do ponto inicial. Os únicos lugares onde você pode parar são esses hotéis, mas pode escolher dentre eles em qual parar. Você tem de parar no hotel final (no quilômetro a_n), que é o seu destino.

Você gostaria, idealmente, de viajar 200 quilômetros por dia, mas isso pode não ser possível (dependendo do espaçamento dos hotéis). Se você viaja x quilômetros durante um dia, a *penalização* para aquele dia é $(200 - x)^2$. Você quer planejar a sua viagem para minimizar a penalização total — ou seja, a soma das penalizações em todos os dias. Forneça um algoritmo eficiente que determine a seqüência ótima de hotéis nos quais parar.

- 6.3. Yuckdonald's está considerando abrir uma série de restaurantes ao longo da Quaint Valley Highway (QVH). Os n possíveis lugares estão ao longo de uma linha reta e as distâncias desses lugares partindo do começo da QVH são, em quilômetros e em ordem crescente, m_1, m_2, \dots, m_n . As restrições são as seguintes:

- Em cada lugar, Yuckdonald's pode abrir no máximo um restaurante. O lucro esperado de abrir um restaurante no lugar i é p_i , onde $p_i > 0$ e $i = 1, 2, \dots, n$.
- Quaisquer dois restaurantes devem estar a, pelo menos, k quilômetros um do outro, onde k é um inteiro positivo.

Forneça um algoritmo eficiente para computar o lucro total esperado máximo sujeito às dadas restrições.

- 6.4. É dado uma *string* de n caracteres $s[1 \dots n]$, que você acredita ser um documento-texto corrompido no qual toda a pontuação desapareceu (tal que ele parece algo como “erao-melhortempo...”). Você quer reconstruir o documento usando um dicionário, que está disponível na forma de uma função booleana $\text{dict}(\cdot)$: para qualquer *string* w ,

$$\text{dict}(w) = \begin{cases} \text{verdadeiro} & \text{se } w \text{ é uma palavra válida} \\ \text{falso} & \text{caso contrário.} \end{cases}$$

- (a) Forneça um algoritmo de programação dinâmica que determine se a *string* $s[\cdot]$ pode ser reconstituída como uma seqüência de palavras válidas. O tempo de execução deve ser no máximo $O(n^2)$, considerando que chamadas para dict tomam tempo unitário.
- (b) Se a *string* for válida, faça seu algoritmo apresentar a correspondente seqüência de palavras.
- 6.5. *Pedras em um tabuleiro quadriculado*. É dado um tabuleiro quadriculado com 4 linhas e n colunas e um número inteiro escrito em cada quadrado do tabuleiro. Também é dado um conjunto de $2n$ pedras e queremos colocar algumas delas ou todas elas no tabuleiro (cada pedra pode ser colocada em exatamente um quadrado) para maximizar a soma dos inteiros nos quadrados que são cobertos pelas pedras. Há uma restrição: para que disposição das pedras seja legal, nenhum par delas pode estar em quadrados adjacentes horizontal ou verticalmente (adjacência diagonal é permitida).

- (a) Determine o número de *padrões* legais que podem ocorrer em alguma coluna (isoladamente, ignorando as pedras nas colunas adjacentes) e descreva estes padrões.

Chame dois padrões de *compatíveis* se eles podem ser colocados em colunas adjacentes em uma disposição legal. Vamos considerar subproblemas consistindo nas primeiras k colunas $1 \leq k \leq n$. A cada subproblema pode ser atribuído um *tipo*, que é o padrão ocorrendo na última coluna.

(b) Usando as noções de compatibilidade e tipo, forneça um algoritmo de programação dinâmica de tempo $O(n)$ para computar uma disposição ótima.

- 6.6. Vamos definir uma operação de multiplicação sobre três símbolos a, b, c de acordo com a seguinte tabela; assim $ab = b$, $ba = c$ e assim por diante. Note que a operação de multiplicação definida pela tabela não é associativa nem comutativa.

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Encontre um algoritmo eficiente que examina uma *string* desses símbolos, digamos *bbbbac*, e decida se é possível ou não organizar parênteses na *string* para que o valor da expressão resultante seja a . Por exemplo, na entrada *bbbbac* seu algoritmo deve retornar *sim* porque $((b(bb))(ba))c = a$.

- 6.7. Uma subseqüência é um *palíndromo* se, ao ser lida da esquerda para a direita e da direita para a esquerda, tiver o mesmo sentido. Por exemplo, a seqüência

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

tem muitas subseqüências que são palíndromos, incluindo A, C, G, C, A e A, A, A, A (por sua vez, a subseqüência A, C, T não é um palíndromo). Projete um algoritmo que tome a seqüência $x[1 \dots n]$ e retorne o comprimento da maior subseqüência que é um palíndromo. Seu tempo de execução deve ser $O(n^2)$.

- 6.8. Dadas duas *strings* $x = x_1 x_2 \dots x_n$ e $y = y_1 y_2 \dots y_m$, desejamos encontrar o comprimento da maior substring comum delas, isto é, o maior k para o qual existem índices i e j com $x_i x_{i+1} \dots x_{i+k-1} = y_j y_{j+1} \dots y_{j+k-1}$. Mostre como fazer isso em tempo $O(mn)$.
- 6.9. Uma certa linguagem de processamento e *strings* oferece uma operação primitiva que divide uma *string* em dois pedaços. Como essa operação envolve copiar a *string* original, ela toma n unidades de tempo para uma *string* de tamanho n , não importa a posição do corte. Suponha, agora, que você queira quebrar a *string* em muitos pedaços. A ordem na qual os cortes são feitos pode afetar o tempo de execução total. Por exemplo, se você quiser cortar uma *string* de 20 caracteres nas posições 3 e 10, fazer o primeiro corte na posição 3 incorrerá em um custo total de $20 + 17 = 37$, enquanto fazer a posição 10 primeiro terá um custo melhor de $20 + 10 = 30$.

Forneça um algoritmo de programação dinâmica que, dadas as posições de m cortes em uma *string* de comprimento n , encontre o custo mínimo de dividir a *string* nos $m + 1$ pedaços.

- 6.10. *Contando caras.* Dados inteiros n e k , junto com $p_1, \dots, p_n \in [0, 1]$, você quer determinar a probabilidade de obter exatamente k caras quando n moedas viciadas são lançadas independentemente de maneira aleatória, onde p_i é a probabilidade de que a i -ésima

moeda dê cara. Forneça um algoritmo de tempo $O(nk)$ para esta tarefa². Considere que você pode multiplicar e adicionar dois números em $[0, 1]$ em tempo $O(1)$.

- 6.11. Dadas duas *strings* $x = x_1 x_2 \cdots x_n$ e $y = y_1 y_2 \cdots y_m$, queremos encontrar o comprimento da *maior subsequência comum* delas, isto é, o maior k para o qual existem índices $i_1 < i_2 < \cdots < i_k$ e $j_1 < j_2 < \cdots < j_k$ com $x_{i_1} x_{i_2} \cdots x_{i_k} = y_{j_1} y_{j_2} \cdots y_{j_k}$. Mostre como fazer isso em tempo $O(mn)$.
- 6.12. É dado um polígono convexo P sobre n vértices no plano (especificados por suas coordenadas x e y). Uma *triangulação* de P é uma coleção de $n - 3$ diagonais de P tal que nenhum par de diagonais se intercepta (exceto possivelmente nas suas extremidades). Note que uma triangulação divide o interior do polígono em $n - 2$ triângulos disjuntos. O custo de uma triangulação é a soma dos comprimentos das suas diagonais. Forneça um algoritmo eficiente para encontrar a triangulação de custo mínimo. (*Dica:* rotule os vértices de P com $1, \dots, n$, começando com um vértice arbitrário e andando em sentido horário. Para $1 \leq i < j \leq n$, o subproblema $A(i, j)$ denota a triangulação de custo mínimo do polígono gerado pelos vértices $i, i + 1, \dots, j$.)
- 6.13. Considere o seguinte jogo. A “banca” produz uma seqüência $s_1 \cdots s_n$ de “cartas”, com a face para cima, onde cada carta s_i tem valor v_i . Então dois jogadores se revezam selecionando uma carta da seqüência, mas podem selecionar apenas a primeira ou a última carta da seqüência (restante). O objetivo é coletar cartas de maior valor total. (Por exemplo, você pode pensar nas cartas como notas de dinheiro de diferentes valores.) Considere n par.
 - (a) Mostre uma seqüência de cartas tal que não é uma estratégia ótima para o primeiro jogador começar selecionando a carta de maior valor disponível, ou seja, a estratégia gulosa óbvia é subótima.
 - (b) Forneça um algoritmo $O(n^2)$ para computar uma estratégia ótima para o primeiro jogador. Dada a seqüência inicial, seu algoritmo deve pré-computar em tempo $O(n^2)$ alguma informação, e, então, o primeiro jogador deve ser capaz de fazer cada movimento de forma ótima em tempo $O(1)$ consultando a informação pré-computada.
- 6.14. *Cortando pano.* É dado uma peça retangular de pano com dimensões $X \times Y$, onde X e Y são inteiros positivos, e uma lista de n produtos que podem ser feitos usando o pano. Para cada produto $i \in [1, n]$ você sabe que um retângulo de pano de dimensões $a_i \times b_i$ é necessário e que o preço final de venda do produto é c_i . Considere que a_i , b_i e c_i são todos inteiros positivos. Você tem uma máquina que pode cortar qualquer peça retangular de pano em duas peças, ou horizontal ou verticalmente. Projete um algoritmo que determine o melhor retorno possível sobre a peça de pano de $X \times Y$, ou seja, uma estratégia para cortar o pano de forma que os produtos feitos das peças resultantes dêem a soma máxima de preços de venda. Você é livre para fazer quantas cópias de um produto quiser, ou mesmo nenhuma.
- 6.15. Suponha que dois times, A e B , estejam jogando uma partida para ver quem é o primeiro a ganhar n jogos (para algum particular n). Podemos supor que A e B sejam

² De fato, existe também um algoritmo $O(n \log^2 n)$ a seu alcance.

igualmente competentes, tal que cada um tem uma chance de 50% de ganhar cada particular jogo. Suponha que eles já jogaram $i + j$ jogos, dos quais A ganhou i e B ganhou j . Forneça um algoritmo eficiente para computar a probabilidade de que A virá a vencer a partida. Por exemplo, se $i = n - 1$ e $j = n - 3$, então, a probabilidade de que A ganhe a partida é $7/8$, pois ele tem de ganhar um dos próximos três jogos.

- 6.16. *O problema da venda de garagem.* Em uma manhã de domingo, existem n vendas de garagem acontecendo g_1, g_2, \dots, g_n . Para cada venda de garagem g_j , você tem uma estimativa de valor, v_j . Para cada duas vendas de garagem você tem uma estimativa do custo de transporte d_{ij} de ir de g_i para g_j . Também são dados os custos d_{0j} e d_{j0} de ir entre a sua casa e cada venda de garagem. Você quer encontrar um circuito por um subconjunto das dadas vendas de garagem, começando e terminando na sua casa, que maximize o seu benefício total menos o custo total de transporte.

Forneça um algoritmo que resolva este problema em tempo $O(n^2 2^n)$. (Dica: Ele está fortemente relacionado com o problema do caixeiro-viajante.)

- 6.17. Dado um estoque ilimitado de moedas de valores x_1, x_2, \dots, x_n , queremos dar um troco de valor v , ou seja, queremos encontrar um conjunto de moedas cujo valor total é v . Isso pode não ser possível: por exemplo, se os valores das moedas forem 5 e 10, então podemos dar um troco de 15, mas não de 12. Forneça um algoritmo de programação dinâmica de tempo $O(nv)$ para o seguinte problema.

Entrada: $x_1, \dots, x_n; v$.

Saída: É possível dar um troco de v usando moedas de valores x_1, \dots, x_n ?

- 6.18. Considere a seguinte variação do problema do troco (Exercício 6.17): são dados os valores x_1, x_2, \dots, x_n e você quer dar um troco de valor v , mas pode usar cada valor de moeda no máximo uma vez. Por exemplo, se os valores são 1, 5, 10, 20, então você pode dar um troco de $16 = 1 + 15$ e de $31 = 1 + 10 + 20$, mas não de 40 (porque não pode usar 20 duas vezes).

Entrada: Inteiros positivos x_1, x_2, \dots, x_n ; outro inteiro v .

Saída: Você pode dar um troco de v , usando cada valor x_i no máximo uma vez?

Mostre como resolver este problema em tempo $O(nv)$.

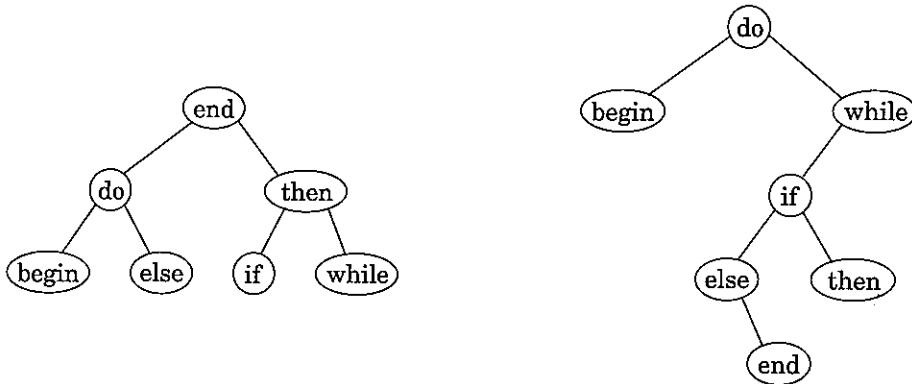
- 6.19. Aqui está mais uma variação do problema do troco (Exercício 6.17).

Dado um estoque ilimitado de moedas de valores x_1, x_2, \dots, x_n , queremos dar um troco de v usando no máximo k moedas; isto é, queremos encontrar um conjunto de $\leq k$ moedas cujo valor total é v . Isso pode não ser possível: por exemplo, se os valores são 5 e 10 e $k = 6$, então podemos dar um troco de 55, mas não de 65. Forneça um algoritmo de programação dinâmica eficiente para o seguinte problema.

Entrada: $x_1, \dots, x_n; k; v$.

Questão: Será que é possível dar um troco de v usando no máximo k moedas, de valores x_1, \dots, x_n ?

Figura 6.12 Duas árvores de busca binária para as palavras-chave de uma linguagem de programação.



- 6.20. *Árvores de busca binária ótimas.* Suponha que conheçamos as freqüências com as quais as palavras-chave ocorrem em programas de uma certa linguagem, por exemplo:

begin	5%
do	40%
else	8%
end	4%
if	10%
then	10%
while	23%

Queremos organizá-las em uma *árvore binária de busca*, para que a palavra-chave na raiz seja alfabeticamente maior do que todas as palavras-chave na subárvore da esquerda e menor do que todas as palavras-chave na subárvore da direita (e isso vale para todos os nós).

A Figura 6.12 tem um exemplo bem-balanceado na parte esquerda. Nesse caso, quando uma palavra-chave está sendo procurada, o número de comparações necessário é no máximo três: por exemplo, ao buscar "while", somente os três nós "end", "then" e "while" são examinados. Mas como sabemos as freqüências com as quais as palavras-chave são procuradas, podemos usar uma função de custo ainda mais refinada, o *número médio de comparações* ao buscar uma palavra. Para a árvore de busca na esquerda, ele é

$$\text{custo} = 1(0,04) + 2(0,40 + 0,10) + 3(0,05 + 0,08 + 0,10 + 0,23) = 2,42.$$

Por essa medida, a melhor árvore de busca é a da parte direita, que tem um custo de 2,18.

Forneça um algoritmo eficiente para a seguinte tarefa.

Entrada: n palavras (ordenadas alfabeticamente); as freqüências dessas palavras: p_1, p_2, \dots, p_n .

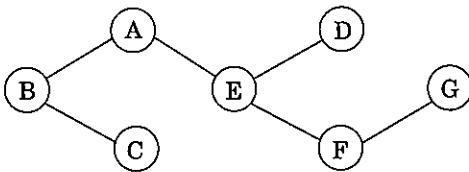
Saída: A árvore binária de busca com o menor custo (definido acima como o número esperado de comparações na busca por uma palavra).

- 6.21. Uma *cobertura de vértices* de um grafo $G = (V, E)$ é um subconjunto de vértices $S \subseteq V$ que inclui ao menos uma extremidade de cada aresta de E . Forneça um algoritmo de tempo linear para a seguinte tarefa.

Entrada: Uma árvore não-direcionada $T = (V, E)$.

Saída: O tamanho da menor cobertura de vértices de T .

Por exemplo, na árvore abaixo, as possíveis coberturas de vértice incluem $\{A, B, C, D, E, F, G\}$ e $\{A, C, D, F\}$, mas não $\{C, E, F\}$. A menor cobertura de vértice tem tamanho 3: $\{B, E, G\}$.



- 6.22. Forneça um algoritmo $O(nt)$ para a seguinte tarefa.

Entrada: Uma lista de n inteiros positivos a_1, a_2, \dots, a_n ; um inteiro positivo t .

Questão: Será que algum subconjunto de a_i tem soma total t ? (Você pode usar cada a_i no máximo uma vez.)

- 6.23. Um sistema de produção de missão crítica tem n estágios que têm de ser realizados seqüencialmente; o estágio i é realizado com a máquina M_i . Cada máquina M_i tem uma probabilidade r_i de funcionar confiavelmente e uma probabilidade $1 - r_i$ de falhar (e as falhas são independentes). Portanto, se implementarmos cada estágio com uma única máquina, a probabilidade de que o sistema como um todo funcione é $r_1 \cdot r_2 \cdots r_n$. Para aperfeiçoarmos essa probabilidade, adicionamos redundância, possuindo m_i cópias da máquina M_i que realiza o estágio i . A probabilidade de que todas as m_i cópias falhem simultaneamente é apenas $(1 - r_i)^{m_i}$, portanto a probabilidade de que o estágio i seja completado corretamente é $1 - (1 - r_i)^{m_i}$ e a probabilidade de que o sistema inteiro funcione é $\prod_{i=1}^n (1 - (1 - r_i)^{m_i})$. Cada máquina M_i tem um custo c_i , e existe um orçamento total B para comprar máquinas. (Considere que B e c_i são inteiros positivos.)

Dadas as probabilidades r_1, \dots, r_n , os custos c_1, \dots, c_n , e o orçamento B , encontre as redundâncias m_1, \dots, m_n , que estejam dentro do orçamento e que maximizem a probabilidade de que o sistema funcione corretamente.

- 6.24. *Complexidade de tempo e espaço para programação dinâmica.* Nossa algoritmo de programação dinâmica para computar a distância de edição entre strings de comprimento m e n cria uma tabela de tamanho $n \times m$ e, assim, precisa de tempo e espaço $O(mn)$. Na prática, ele vai esgotar a memória muito antes de esgotar o tempo. Como pode essa necessidade de espaço ser reduzida?

- (a) Mostre que se queremos apenas computar o valor da distância de edição (em vez da sequência ótima de edições), então é necessário espaço apenas $O(n)$,

porque somente uma porção pequena da tabela precisa ser mantida em qualquer dado momento.

- (b) Agora suponha que também queiramos a seqüência ótima de edições. Como vimos antes, este problema pode ser reformulado em termos de um correspondente dag em forma de reticulada, no qual o objetivo é encontrar o caminho ótimo do nó $(0, 0)$ até o nó (n, m) . Será conveniente trabalhar com essa formulação e, como estamos falando de conveniência, podemos também considerar que m é uma potência de 2.

Comecemos com uma pequena adição para o algoritmo de distância de edição que irá se revelar muito útil. O caminho ótimo no dag tem de passar por um nó intermediário $(k, m/2)$ para algum k ; mostre como o algoritmo pode ser modificado para também retornar este valor k .

- (c) Agora considere um esquema recursivo:

```
procedimento encontrar-caminho $((0, 0) \rightarrow (n, m))$ 
    computar o valor de  $k$  acima
    encontrar-caminho $((0, 0) \rightarrow (k, m/2))$ 
    encontrar-caminho $((k, m/2) \rightarrow (n, m))$ 
    concatenar estes dois caminhos, com  $k$  no meio
```

Mostre que o esquema pode ser executado em tempo $O(mn)$ e espaço $O(n)$.

- 6.25. Considere o seguinte problema 3-PARTIÇÃO. Dados inteiros a_1, \dots, a_n , queremos determinar se é possível partitionar $\{1, \dots, n\}$ em três subconjuntos disjuntos I, J, K tal que

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{1}{3} \sum_{i=1}^n a_i$$

Por exemplo, para a entrada $(1, 2, 3, 4, 4, 5, 8)$, a resposta é *sim*, porque existe a partição $(1, 8), (4, 5), (2, 3, 4)$. Por sua vez, para a entrada $(2, 2, 3, 5)$, a resposta é *não*.

Projete e analise um algoritmo de programação dinâmica para 3-PARTIÇÃO que rode em tempo polinomial em n e em $\sum_i a_i$.

- 6.26. *Alinhamento de seqüência.* Quando um novo gene é encontrado, uma abordagem-padrão para entender sua função é examinar um banco de dados de genes conhecidos e encontrar emparelhamentos próximos. A proximidade de dois genes é medida pela extensão na qual eles estão *alinhanados*. Para formalizar isso, pense em um gene como uma longa *string* sobre o alfabeto $\Sigma = \{A, C, G, T\}$. Considere dois genes (*strings*) $x = ATGCC$ e $y = TACGCA$. Um alinhamento de x e y é uma maneira de emparelhar essas duas *strings* ao escrevê-las em colunas, por exemplo:

-	A	T	-	G	C	C
T	A	-	C	G	C	A

Aqui o “—” indica um “espaço vazio”. Os caracteres de cada *string* têm de aparecer em ordem e cada coluna tem de conter um caractere de pelo menos uma das *strings*. O valor de um alinhamento é especificado por uma matriz δ de valores de tamanho $(|\Sigma| + 1) \times (|\Sigma| + 1)$, onde a linha e a coluna extras são para acomodar os espaços vazios. Por exemplo, o alinhamento anterior tem o seguinte valor:

$$\delta(-, T) + \delta(A, A) + \delta(T, -) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(C, A).$$

Forneça um algoritmo de programação dinâmica que tome como entrada duas *strings* $x[1 \dots n]$ e $y[1 \dots m]$ e uma matriz de valores δ , e retorne o alinhamento de maior valor. O tempo de execução deve ser $O(mn)$.

- 6.27. *Alinhamento com penalidades para espaços vazios.* O algoritmo do Exercício 6.26 ajuda a identificar seqüências de DNA que estão próximas umas das outras. As discrepâncias entre as seqüências próximas são freqüentemente causadas por erros na replicação do DNA. Entretanto, uma olhada mais cuidadosa no processo de replicação biológica revela que a função de valor que consideramos antes tem um problema qualitativo: a natureza, freqüentemente, insere ou remove substrings inteiras de nucleotídeos (criando longos espaços vazios), em vez de apenas editar uma posição por vez. Portanto, a penalização por um espaço vazio de tamanho 10 não deve ser 10 vezes a penalidade por um espaço vazio de tamanho 1, mas algo substancialmente menor.

Repita o Exercício 6.26, mas dessa vez use uma função de valor modificada na qual a penalização por um espaço vazio de tamanho k é $c_0 + c_1 k$, onde c_0 e c_1 são constantes dadas (e c_0 é maior do que c_1).

- 6.28. *Alinhamento local de seqüências.* Muitas vezes duas seqüências de DNA são significativamente diferentes, mas contêm regiões muito similares e *altamente conservadas*. Projete um algoritmo que tome como entrada duas *strings* $x[1 \dots n]$ e $y[1 \dots m]$ e uma matriz de valores δ (como definida no Exercício 6.26), e encontre as substrings x' e y' de x e y , respectivamente, que apresentam o alinhamento de maior valor sobre todos os pares de tais substrings. Seu algoritmo deve tomar tempo $O(mn)$.
- 6.29. *Encadeamento de exons.* Cada gene corresponde a uma sub-região do genoma total (a seqüência de DNA); entretanto, parte dessa região pode ser “*junk DNA*”. Freqüentemente, um gene consiste em várias peças chamadas exons, que são separadas por fragmentos junk chamados ítrons. Isso complica o processo de identificar genes em um genoma recentemente seqüenciado.

Suponha que tenhamos uma nova seqüência de DNA e que queiramos checar se um certo gene (uma string) está presente nele. Como não podemos esperar que o gene será uma subseqüência contígua, examinamos os emparelhamentos parciais — fragmentos do DNA que também estão presentes no gene (na verdade, mesmo esses emparelhamentos parciais serão aproximados, não perfeitos). Depois tentamos montar os fragmentos.

Denote por $x[1 \dots n]$ a seqüência de DNA. Cada emparelhamento parcial pode ser representado por uma tripla (l_i, r_i, w_i) , onde $x[l_i \dots r_i]$ é o fragmento e w_i é um peso

representando a força do emparelhamento (ela pode ser um valor de alinhamento local ou alguma outra quantidade estatística). Muitos desses potenciais emparelhamentos podem ser falsos, assim o objetivo é encontrar um subconjunto de triplas que são consistentes (não se sobrepõem) e têm um peso total máximo.

Mostre como fazer isso eficientemente.

- 6.30. *Reconstruindo árvores evolucionárias com parcimônia máxima.* Suponha que consigamos seqüenciar um particular gene por toda uma gama de espécies diferentes. Para sermos concretos, digamos que há n espécies, e as seqüências são *strings* de comprimento k sobre o alfabeto $\Sigma = \{A, C, G, T\}$. Como podemos usar essa informação para reconstruir a história evolucionária dessas espécies?

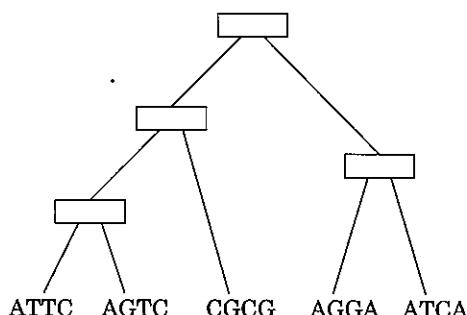
A história evolucionária é normalmente representada por uma árvore cujas folhas são as diferentes espécies, cuja raiz é seu ancestral comum e cujas ramificações internas representam eventos de especiação (ou seja, momentos nos quais uma nova espécie se origina de outra já existente). Portanto precisamos encontrar o seguinte:

- Uma árvore (binária) evolucionária com as dadas espécies nas folhas.
- Para cada nó interno, uma *string* de tamanho k : a seqüência de genes para aquele particular ancestral.

Para cada possível árvore T , anotada com as seqüências $s(u) \in \Sigma^k$ em cada um dos seus nós u , podemos atribuir um score baseado no princípio da *parcimônia*: um número menor de mutações é mais provável.

$$\text{score}(T) = \sum_{(u, v) \in E(T)} (\text{número de posições nas quais } s(u) \text{ e } s(v) \text{ concordam}).$$

Encontrar a árvore com o maior score é um problema difícil. Aqui vamos considerar apenas uma pequena parte dele: suponha que conhecemos a estrutura da árvore e que queremos preencher as seqüências $s(u)$ dos nós internos u . Veja um exemplo com $k = 4$ e $n = 5$:



- (a) Neste particular exemplo, há várias reconstruções com parcimônia máxima das seqüências dos nós internos. Encontre uma delas.
- (b) Forneça um algoritmo eficiente (em termos de n e k) para esta tarefa. (*Dica:* Muito embora as seqüências possam ser longas, você pode fazer uma posição só por vez.)

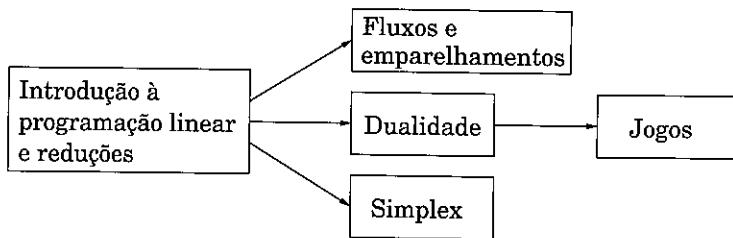
Capítulo 7

Programação linear e reduções

Muitos dos problemas para os quais queremos algoritmos são tarefas de *otimização*: o caminho *mais curto*, a árvore espalhada *mais barata*, a subseqüência crescente *mais longa* e assim por diante. Em tais casos, procuramos uma solução que (1) satisfaça certas restrições (por exemplo, o caminho tem de usar arestas do grafo e levar de s até t , a árvore tem de tocar todos os nós, a subseqüência tem de ser crescente); e (2) é a melhor possível, com relação a algum critério bem-definido, entre todas as soluções que satisfazem estas restrições.

Programação linear descreve uma ampla classe de tarefas de otimização nas quais tanto as restrições quanto os critérios são *funções lineares*. Acontece que um enorme número de problemas pode ser expresso desse jeito.

Dada a extensão deste tópico, o capítulo está dividido em várias partes, que podem ser lidas separadamente, mas de acordo com as seguintes dependências.



7.1 Uma introdução à programação linear

Em um problema de programação linear é dado um conjunto de variáveis às quais queremos atribuir valores reais de modo que (1) satisfaça um conjunto de equações lineares e/ou inequações lineares envolvendo essas variáveis e (2) maximize ou minimize determinada função objetivo linear.

7.1 Exemplo: maximização de lucro

Uma doceria tem dois produtos: seu carro-chefe, um sortimento de chocolates triangulares, chamados *Pyramide*, e outro, um tanto decadente e luxuoso, *Pyramide Nuit*. Quantas de cada um deles ela deve produzir para maximizar seus lucros? Digamos que produza x_1 caixas de *Pyramide* por dia, a um lucro de \$1 cada e x_2 caixas de *Nuit*, com um lucro mais substancial de \$6 por peça; x_1 e x_2 são valores desconhecidos que desejamos determinar. Mas isso não é tudo, há também algumas restrições sobre x_1 e x_2 que devem ser consideradas (além das óbvias, $x_1, x_2 \geq 0$). Primeiro, a demanda diária para esses chocolates exclusivos é limitada a no máximo 200 caixas de *Pyramide* e 300 caixas de *Nuit*. Também, a força de trabalho atual pode produzir um total de no máximo 400 caixas de chocolate por dia. Quais os níveis ótimos de produção?

Representamos a situação por um *programa linear*, como se segue.

$$\text{Função objetivo} \quad \max x_1 + 6x_2$$

$$\text{Restrições} \quad x_1 \leq 200$$

$$x_2 \leq 300$$

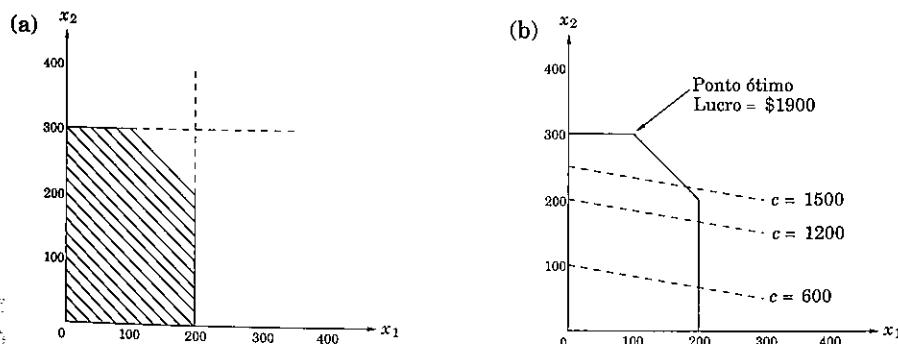
$$x_1 + x_2 \leq 400$$

$$x_1, x_2 \geq 0$$

Uma equação linear em x_1 e x_2 define uma linha no plano bidimensional (2D), e uma equação linear designa um *semi-espac*o, a região em um dos lados da linha. Assim, o conjunto de todas as *soluções factíveis* desse programa linear, isto é, os pontos (x_1, x_2) que satisfazem todas as restrições, é a intersecção de cinco semi-espacos. Ela é um polígono convexo, mostrado na Figura 7.1.

Queremos encontrar o ponto neste polígono no qual a função objetivo — o lucro — maximizada. Os pontos com um lucro de c dólares estão na linha $x_1 + 6x_2 = c$, que tem uma inclinação de $-1/6$ e é mostrada na Figura 7.1 para valores selecionados de c . A medida que c cresce, a “linha de lucro” move-se paralelamente a si mesma, para

Figura 7.1 (a) A região factível para um programa linear. (b) Linhas de contorno da função objetivo: $x_1 + 6x_2 = c$ para diferentes valores do lucro c .



cima e para a direita. Como o objetivo é maximizar c , temos de mover a linha o mais para cima possível, sem deixar de tocar a região factível. A solução ótima será o último ponto factível que a linha de lucro vê e, por isso, tem de ser um vértice do polígono, como mostra a figura. Se a inclinação da linha de lucro fosse diferente, então seu último contato com o polígono poderia ser uma aresta inteira em vez de um vértice só. Nesse caso, a solução ótima não seria única, mas haveria certamente um vértice ótimo.

É uma regra geral de programas lineares que o ótimo é alcançado em um vértice da região factível. As únicas exceções são casos nos quais o ótimo não existe; isso pode acontecer de duas maneiras:

1. O programa linear é *infactível*; ou seja, as restrições são tão justas que é impossível satisfazer todas elas. Por exemplo,

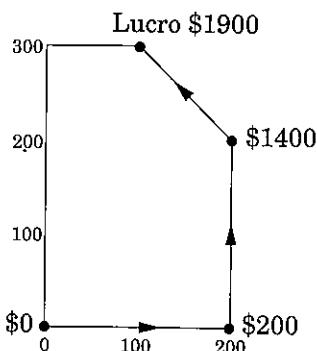
$$x \leq 1, x \geq 2.$$

2. As restrições são tão folgadas que a região factível é *ilimitada*, e é possível atingir valores de objetivo arbitrariamente altos. Por exemplo,

$$\begin{aligned} \max & x_1 + x_2 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Resolvendo programas lineares

Programas lineares (PLs) podem ser resolvidos pelo *método simplex*, criado por George Dantzig em 1947. Vamos explicá-lo em mais detalhes na Seção 7.6, mas resumidamente, este algoritmo começa em um vértice, no nosso caso talvez $(0, 0)$, e repetidamente procura por um vértice adjacente (conectado por uma aresta da região factível) de melhor valor objetivo. Dessa maneira, ele faz uma *escalada* nos vértices do polígono, andando de vizinho a vizinho para constantemente aumentar o lucro ao longo do caminho. Veja uma possível trajetória.



Ao alcançar um vértice que não tem um vizinho melhor, o simplex o declara como ótimo e encerra. Por que o teste *local* implica otimalidade *global*? Por simples geometria — pense na linha de lucro passando por esse vértice. Como todos os vizinhos do vértice estão abaixo da linha, o restante do polígono factível tem também de estar abaixo desta linha.

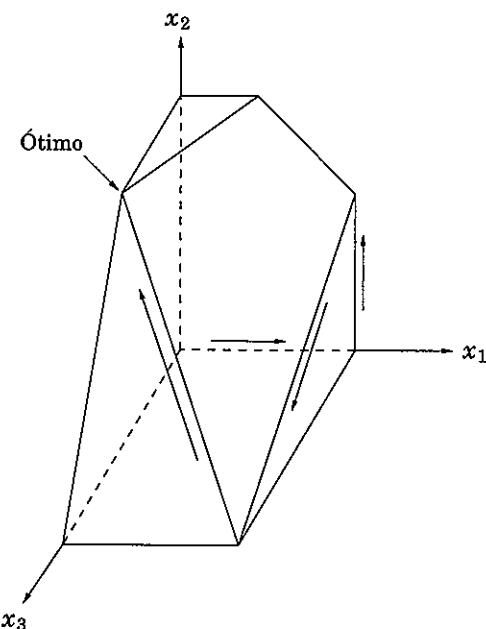
Mais produtos

Encorajada pela demanda do consumidor, a doceria decide introduzir uma terceira e ainda mais exclusiva linha de chocolates, chamada *Pyramide Luxe*. Uma caixa deles trará um lucro de \$13. Sejam x_1 , x_2 , x_3 o número de caixas de cada chocolate produzidas diariamente, com x_3 correspondendo ao Luxe. As antigas restrições sobre x_1 e x_2 persistem, entretanto a restrição de trabalho agora se estende a x_3 também: a soma de todas as três variáveis pode ser no máximo 400. Além disso, acontece que Nuit e Luxe requerem a mesma máquina de empacotamento, exceto que Luxe a usa três vezes mais, o que impõe uma outra restrição $x_2 + 3x_3 \leq 600$. Quais os melhores níveis de produção possíveis?

Veja o programa linear atualizado.

$$\begin{aligned} \max \quad & x_1 + 6x_2 + 13x_3 \\ \text{s.t.} \quad & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1 + x_2 + x_3 \leq 400 \\ & x_2 + 3x_3 \leq 600 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Figura 7.2 O poliedro factível para um programa linear de três variáveis.



O espaço de soluções agora é tridimensional. Cada equação linear define um plano 3D e cada inequação um semi-espaco de um lado do plano. A região factível é uma interseção de sete semi-espacos, um poliedro (Figura 7.2). Examinando a figura, você pode decifrar qual inequação corresponde a cada face do poliedro?

Um lucro de c corresponde ao plano $x_1 + 6x_2 + 13x_3 = c$. À medida que c cresce, o plano de lucro move-se paralelamente a si mesmo, mais e mais para o ortante positivo até que ele não toque mais a região factível. O ponto de contato final é o vértice ótimo: $(0, 300, 100)$, com um lucro total de \$3.100.

Como o algoritmo simplex se comportaria nesse problema modificado? Como antes, ele se moveria de vértice para vértice, ao longo das arestas do poliedro, constantemente aumentando o lucro. Uma possível trajetória é mostrada na Figura 7.2, correspondendo à seguinte seqüência de vértices e lucros:

$$\begin{array}{ccccc} (0, 0, 0) & (200, 0, 0) & (200, 200, 0) & (200, 0, 200) & (0, 300, 100) \\ \$0 & \rightarrow & \$200 & \rightarrow & \$1.400 & \rightarrow & \$2.800 & \rightarrow & \$3.100 \end{array}$$

Por fim, ao alcançar um vértice que não tem um vizinho melhor, ele terminaria e declararia o vértice como o ponto ótimo. Mais uma vez por geometria básica, se todos os vizinhos do vértice estão em um lado do plano de lucro, então também está o poliedro inteiro.

Um truque mágico chamado dualidade

Veja por que deve acreditar que $(0, 300, 100)$, com um lucro total de 3.100, é o ótimo: reexamine o programa linear. Adicione a segunda inequação à terceira, e adicione a elas a quarta multiplicada por 4. O resultado é a inequação $x_1 + 6x_2 + 13x_3 \leq 3.100$.

Você vê? Essa inequação informa que nenhuma solução factível (valores x_1, x_2, x_3 satisfazendo as restrições) pode ter um lucro maior do que 3.100. Portanto temos que, de fato, ter achado o ótimo! A única questão é de onde conseguimos estes multiplicadores misteriosos $(0, 1, 1, 4)$ para as quatro inequações?

Na Seção 7.4 veremos que sempre é possível obter tais multiplicadores resolvendo um outro PL! Exceto que (isso fica ainda melhor) não precisamos nem mesmo resolver outro PL, porque ele está tão intimamente conectado ao original — é chamado de *dual* —, que resolver o PL original resolve também o dual! Mas estamos adiantando muito a história.

O que dizer se adicionamos uma quarta linha de chocolates, ou uma centena mais deles? Então ele se torna um problema de alta dimensão e difícil de visualizar. O simplex continua a funcionar neste contexto geral, embora não possamos mais nos basear em intuição geométrica simples para a sua descrição e justificativa. Vamos estudar o algoritmo simplex completo na Seção 7.6.

Por enquanto, podemos estar tranqüilos sabendo de que existem muitas bibliotecas profissionais, de padrão industrial, que implementam o simplex e cuidam de todos os detalhes sutis como precisão numérica. Em uma aplicação típica, a principal tarefa é expressar corretamente o problema como um programa linear. A biblioteca, então, toma conta do restante.

Com isso em mente, vejamos uma aplicação de alta dimensão.

7.1.2 Exemplo: planejamento de produção

Desta vez, nossa companhia fabrica tapetes em teares manuais, um produto para o qual a demanda é extremamente sazonal. Nosso analista acabou de obter as estimativas de demanda para todos os meses do próximo ano: d_1, d_2, \dots, d_{12} . Como esperado, elas são muito desiguais, variando de 440 a 920.

Apresentemos um rápido perfil da companhia. Atualmente temos 30 empregados, cada um deles produz 20 tapetes por mês e recebe um salário mensal de 2.000. Não há nenhuma sobra inicial de tapetes.

Como podemos lidar com as flutuações na demanda? Veja três maneiras:

1. *Hora extra*, mas isto é caro, pois a hora extra paga 80% a mais do que o pagamento regular. Além disso, os trabalhadores podem fazer no máximo 30% de hora extra.
2. *Contratar e demitir*, mas isto custa \$320 e \$400, respectivamente, por trabalhador.
3. *Estocar a sobra de produção*, mas isto custa \$8 por tapete por mês. Atualmente não há nenhum tapete estocado e temos de terminar o ano sem nenhum tapete estocado.

Esse problema bastante elaborado pode ser formulado e resolvido como um programa linear!

Um passo inicial crucial é definir as variáveis.

w_i = número de trabalhadores durante i -ésimo mês; $w_0 = 30$.

x_i = número de tapetes feitos durante o i -ésimo mês.

o_i = número de tapetes produzidos por hora extra no mês i .

h_i, f_i = número de trabalhadores contratados e demitidos, respectivamente, no começo do mês i .

s_i = número de tapetes estocados no final do mês i ; $s_0 = 0$.

Ao todo, existem 72 variáveis (74 se você conta w_0 e s_0).

Agora escrevemos as restrições. Primeiro, todas as variáveis têm de ser não-negativas:

$$w_i, x_i, o_i, h_i, f_i, s_i \geq 0, \quad i = 1, \dots, 12.$$

O número total de tapetes feitos por mês consiste em produção regular mais hora extra:

$$x_i = 20w_i + o_i$$

(uma restrição para cada $i = 1, \dots, 12$). O número de trabalhadores pode potencialmente mudar no começo de cada mês:

$$w_i = w_{i-1} + h_i - f_i.$$

O número de tapetes estocados no final de cada mês é o que tínhamos inicialmente, mais o número que produzimos, menos a demanda do mês:

$$s_i = s_{i-1} + x_i - d_i.$$

E hora extra é limitada:

$$o_i \leq 6w_i.$$

Por fim, qual a função objetivo? É minimizar o custo total:

$$\min 2000 \sum_i w_i + 320 \sum_i h_i + 400 \sum_i f_i + 8 \sum_i s_i + 180 \sum_i o_i,$$

uma função linear das variáveis. Resolver esse programa linear com simplex deve levar menos do que um segundo e dará a estratégia de negócio ótima para nossa companhia.

Bem, quase. Pode acontecer de a solução ótima ser *fracional*, por exemplo, ela pode envolver contratar 10,6 trabalhadores no mês de março. Esse número seria arredondado para 10 ou 11 para fazer sentido, e o custo total, então, cresceria correspondentemente. No exemplo corrente, a maioria das variáveis assume valores bastante grandes (dois dígitos) e, assim, decisões de arredondamento devem ser feitas com muito cuidado para terminar com uma solução inteira de qualidade razoável.

Em geral, existe uma tensão em programação linear entre a facilidade de obter soluções fracionárias e o fato de que soluções inteiras são desejáveis. Como veremos no Capítulo 8, encontrar a solução inteira ótima de um PL é um problema importante, mas muito difícil, chamado *programação linear inteira*.

7.1.3 Exemplo: alocação ótima de largura de banda

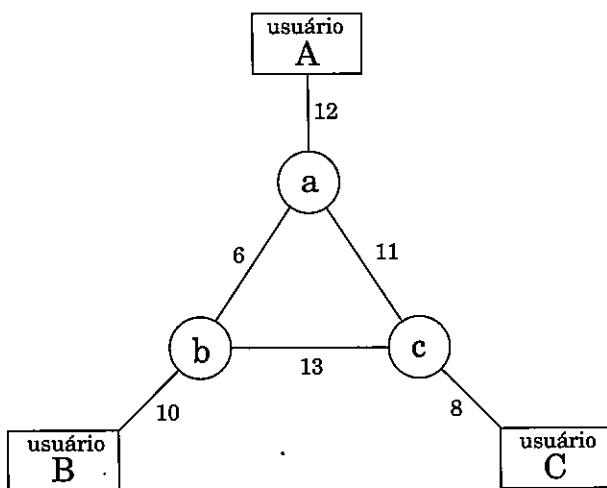
A seguir nos voltamos para uma versão em miniatura do tipo de problema que um provedor de serviço de rede pode enfrentar.

Suponha que estejamos gerenciando uma rede cujas linhas tenham as larguras de banda da Figura 7.3 e precisamos estabelecer três conexões: entre os usuários A e B, entre B e C, e entre A e C. Cada conexão requer pelo menos duas unidades de largura de banda, mas podem receber mais. A conexão A-B paga \$3 por unidade de largura de banda e as conexões B-C e A-C pagam \$2 e \$4, respectivamente.

Cada conexão pode ser roteada de duas maneiras, um caminho longo e um caminho curto, ou por uma combinação: por exemplo, duas unidades de banda pela rota curta, uma pela rota longa. Como roteamos essas conexões para maximizar a receita de nossa rede?

Esse é um programa linear. Temos variáveis para cada conexão e cada caminho (longo ou curto); por exemplo, x_{AB} é a largura de banda de caminho curto alocada para a conexão entre A e B, e x'_{AB} a largura de banda de caminho longo para esta mesma conexão. Demandamos que nenhuma largura de banda de uma aresta seja excedida e

Figura 7.3 Uma rede de comunicação entre três usuários A, B e C. Larguras de banda são mostradas.



que cada conexão obtenha uma largura de banda de pelo menos 2 unidades.

$$\begin{aligned}
 & \max 3x_{AB} + 3x'_{AB} + 2x_{BC} + 2x'_{BC} + 4x_{AC} + 4x'_{AC} \\
 & x_{AB} + x'_{AB} + x_{BC} + x'_{BC} \leq 10 \quad [\text{aresta } (b, B)] \\
 & x_{AB} + x'_{AB} + x_{AC} + x'_{AC} \leq 12 \quad [\text{aresta } (a, A)] \\
 & x_{BC} + x'_{BC} + x_{AC} + x'_{AC} \leq 8 \quad [\text{aresta } (c, C)] \\
 & x_{AB} + x'_{BC} + x'_{AC} \leq 6 \quad [\text{aresta } (a, b)] \\
 & x'_{AB} + x_{BC} + x'_{AC} \leq 13 \quad [\text{aresta } (b, c)] \\
 & x'_{AB} + x'_{BC} + x_{AC} \leq 11 \quad [\text{aresta } (a, c)] \\
 & x_{AB} + x'_{AB} \geq 2 \\
 & x_{BC} + x'_{BC} \geq 2 \\
 & x_{AC} + x'_{AC} \geq 2 \\
 & x_{AB}, x'_{AB}, x_{BC}, x'_{BC}, x_{AC}, x'_{AC} \geq 0
 \end{aligned}$$

Mesmo um pequeno exemplo como esse é difícil resolver à mão (tentel), e no entanto a solução ótima é obtida instantaneamente pelo simplex:

$$x_{AB} = 0, x'_{AB} = 7, x_{BC} = x'_{BC} = 1,5, x_{AC} = 0,5, x'_{AC} = 4,5.$$

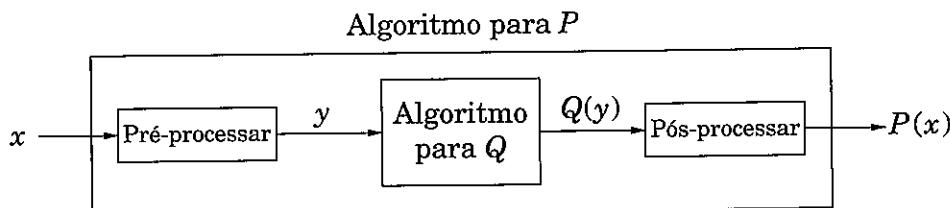
Essa solução não é integral, mas na aplicação atual não precisamos que ela seja e, assim, nenhum arredondamento é requerido. Reexaminado a rede original, vemos que todas as arestas exceto $a-c$ são usadas na capacidade total.

Reduções

Às vezes uma tarefa computacional é suficientemente geral tal que qualquer sub-rotina para ela pode ser usada também para resolver uma variedade de outras tarefas, que à primeira vista poderiam parecer não-relacionadas. Por exemplo, vimos no Capítulo 6 como um algoritmo para encontrar o caminho mais longo em um dag pode, surpreendentemente, também ser usado para encontrar subsequências crescentes mais longas. Descrevemos esse fenômeno dizendo que o problema da subsequência crescente mais longa *se reduz* ao problema do caminho mais longo em um dag. Por sua vez, o caminho mais longo em um dag se reduz ao caminho mínimo em um dag; veja como uma sub-rotina para o segundo caminho pode ser usada para resolver o primeiro:

```
função CAMINHO MAIS LONGO( $G$ )
    negar todos os pesos de arestas de  $G$ 
    retornar CAMINHO MÍNIMO( $G$ )
```

Vamos retroceder um pouco e tomar uma visão ligeiramente mais formal de reduções. Se qualquer sub-rotina para a tarefa Q pode ser usada também para resolver P , dizemos que P *se reduz a* Q . Muitas vezes, P pode ser resolvido com uma chamada única à sub-rotina de Q , o que significa que qualquer instância x de P pode ser transformada em uma instância y de Q tal que $P(x)$ pode ser deduzida de $Q(y)$:



(Você vê que a redução de $P = \text{CAMINHO MAIS LONGO}$ para $Q = \text{CAMINHO MÍNIMO}$ segue este esquema?) Se os procedimentos de pré e pós-processamento são eficientemente computáveis, então, isso cria um algoritmo eficiente para P a partir de *qualquer* algoritmo eficiente para Q !

Reduções aumentam o poder de algoritmos: uma vez que temos um algoritmo para o problema Q (que poderia ser o caminho mínimo, por exemplo), podemos usá-lo para resolver outros problemas. De fato, a maioria das tarefas computacionais que estudamos neste livro pode ser considerada problemas centrais em ciência da computação precisamente porque aparecem em tantas aplicações diferentes, o que é uma outra maneira de dizer que muitos problemas se reduzem a elas. Isso é especialmente verdade para programação linear.

Uma observação por precaução: nosso PL tem uma variável para cada caminho possível entre os usuários. Em uma rede maior, poderia haver facilmente um número exponencial de tais caminhos e, portanto, esta forma particular de traduzir o problema de redes em um PL não escalará bem. Veremos uma formulação mais engenhosa e mais escalável na Seção 7.2.

Uma questão final para você considerar. Suponha que removamos a restrição de que cada conexão deve receber pelo menos duas unidades de largura de banda. O ótimo mudaria?

7.1.4 Variantes de programação linear

Como evidenciado nos nossos exemplos, um programa linear geral tem muitos graus de liberdade.

1. Ele pode ser um problema de maximização ou minimização.
2. Suas restrições podem ser equações ou inequações.
3. As variáveis são freqüentemente restritas a serem não-negativas, mas elas podem também não ter restrição de sinal.

Vamos agora mostrar que essas várias opções de PL *podem todas ser reduzidas* *umas às outras* via transformações simples. Veja como.

1. Para transformar um problema de maximização em um de minimização (ou vice-versa), simplesmente multiplique os coeficientes da função objetivo por -1 .
- 2a. Para transformar uma restrição de inequação como $\sum_{i=1}^n a_i x_i \leq b$ em uma equação, introduza uma nova variável s e use

$$\begin{aligned}\sum_{i=1}^n a_i x_i + s &= b \\ s &\geq 0.\end{aligned}$$

Esta s é chamada de *variável de folga (slack variable)* para a inequação. Como justificativa, observe que o vetor (x_1, \dots, x_n) satisfaz a restrição e inequação original se e somente se existe algum $s \geq 0$ para o qual ele satisfaz a nova restrição de equação.

- 2b. Mudar uma restrição de equação em inequações é fácil: reescreva $ax = b$ como o par equivalente de restrições $ax \leq b$ e $ax \geq b$.
3. Finalmente, para lidar com a variável x que é irrestrita em sinal, faça o seguinte:
 - Introduza duas variáveis não-negativas, $x^+, x^- \geq 0$.
 - Substitua x , quer ela ocorra nas restrições ou na função objetivo, por $x^+ - x^-$.
 Dessa forma, x pode tomar qualquer valor real ajustando-se apropriadamente as novas variáveis. Mais precisamente, qualquer solução factível para o PL original envolvendo x pode ser mapeada em uma solução factível do novo PL envolvendo x^+, x^- , e vice-versa.

Aplicando essas transformações, podemos reduzir qualquer PL (maximização ou minimização, com inequações e equações, e com variáveis não-negativas e irrestritas) em um PL de um tipo muito mais restrito que chamamos de *forma-padrão*, na qual as variáveis são todas não-negativas, as restrições são todas equações, e a função objetivo deve ser minimizada.

Notação matriz-vetor

Uma função linear como $x_1 + 6x_2$ pode ser escrita como o produto interno de dois vetores

$$\mathbf{c} = \begin{pmatrix} 1 \\ 6 \end{pmatrix} \text{ e } \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

denotado $\mathbf{c} \cdot \mathbf{x}$ ou $\mathbf{c}^T \mathbf{x}$. De maneira similar, restrições lineares podem ser compiladas em forma: de matriz-vetor:

$$\begin{array}{l} x_1 \leq 200 \\ x_2 \leq 300 \\ x_1 + x_2 \leq 400 \end{array} \implies \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}}_{\mathbf{A}} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \underbrace{\begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix}}_{\mathbf{b}}.$$

Aqui, cada linha da matriz \mathbf{A} corresponde a uma restrição: seu produto interno com \mathbf{x} é no máximo o valor na linha correspondente de \mathbf{b} . Em outras palavras, se as linhas de \mathbf{A} são os vetores $\mathbf{a}_1, \dots, \mathbf{a}_m$, então a sentença $\mathbf{Ax} \leq \mathbf{b}$ é equivalente a

$$\mathbf{a}_i \cdot \mathbf{x} \leq b_i \text{ para todo } i = 1, \dots, m.$$

Com essas conveniências notacionais, um PL genérico pode ser expresso simplesmente como

$$\begin{aligned} & \max \mathbf{c}^T \mathbf{x} \\ & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq 0. \end{aligned}$$

Por exemplo, nosso primeiro programa linear é reescrito assim:

$$\begin{array}{ll} \max x_1 + 6x_2 & \min -x_1 - 6x_2 \\ x_1 \leq 200 & x_1 + s_1 = 200 \\ x_2 \leq 300 & x_2 + s_2 = 300 \\ x_1 + x_2 \leq 400 & x_1 + x_2 + s_3 = 400 \\ x_1, x_2 \geq 0 & x_1, x_2, s_1, s_2, s_3 \geq 0 \end{array} \implies$$

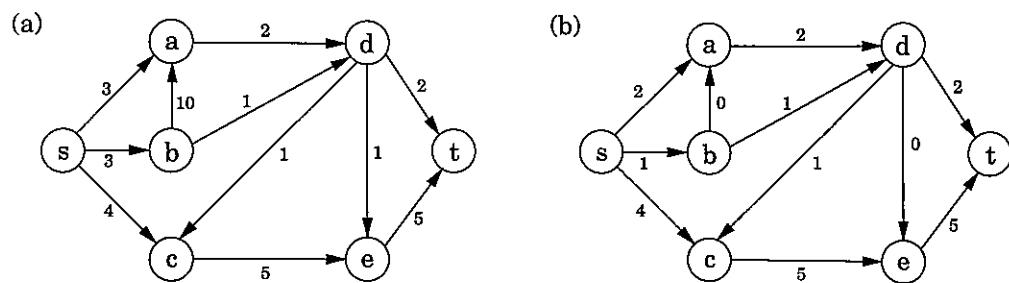
O original também estava em uma forma útil: maximizar um objetivo sujeito a certas inequações. Qualquer PL pode, da mesma forma, ser reformulado usando as reduções dadas antes.

7.2 Fluxo em redes

7.2.1 Bombeando óleo

A Figura 7.4(a) mostra um grafo direcionado representando uma rede de dutos ao longo dos quais pode-se bombeiar óleo. O objetivo é bombeiar o máximo possível de óleo de uma fonte s até um sorvedouro t . Cada duto tem uma capacidade máxima que

Figura 7.4 (a) Uma rede com capacidades nas arestas. (b) Um fluxo na rede.



pode suportar, e não há qualquer chance de estocar óleo no caminho. A Figura 7.4(b) mostra um *fluxo* possível de s até t , que bombeia 7 unidades ao todo. Será que isso é o melhor que pode ser feito?

7.2.2 Maximizando fluxo

As redes com as quais estamos lidando consistem em um grafo direcionado $G = (V, E)$; dois nós especiais $s, t \in V$, que são, respectivamente, uma fonte e um sorvedouro de G ; e *capacidades* $c_e > 0$ nas arestas.

Gostaríamos de bombear o máximo de óleo possível de s até t sem exceder as capacidades de nenhuma das arestas. Um esquema particular de bombeamento é chamado de um *fluxo* e consiste em uma variável f_e para cada aresta e da rede, satisfazendo as duas propriedades seguintes:

1. Ele não viola as capacidades nas arestas: $0 \leq f_e \leq c_e$ para toda $e \in E$.
2. Para todos os nós u exceto s e t , a quantidade de fluxo entrando em u é igual à quantidade saindo de u :

$$\sum_{(w,u) \in E} f_{wu} = \sum_{(u,z) \in E} f_{uz}.$$

Em outras palavras, o fluxo é *conservado*.

O *tamanho* de um fluxo é a quantidade total bombeada partindo de s até t e, pelo princípio da conservação, é igual à quantidade saindo de s :

$$\text{tamanho}(f) = \sum_{(s,u) \in E} f_{su}.$$

Em resumo, nosso objetivo é atribuir valores a $\{f_e : e \in E\}$ que satisfaçam um conjunto de restrições lineares e maximizem uma função objetivo linear. Mas isso é um problema linear! O problema do fluxo máximo se reduz à programação linear.

Por exemplo, para rede da Figura 7.4 o PL tem 11 variáveis, uma por aresta. Ele busca maximizar $f_{sa} + f_{sb} + f_{sc}$ sujeito a um total de 27 restrições: 11 para não-negatividade (tal como $f_{sa} \geq 0$), 11 para capacidade (tal como $f_{sa} \leq 3$), e 5 para conservação de fluxo (uma para cada nó do grafo além de s e t , tal como $f_{sc} + f_{dc} = f_{ce}$). O simplex não levaria

tempo algum para resolver corretamente o problema e para confirmar que, no nosso exemplo, um fluxo de 7 é de fato ótimo.

7.2.3 Um olhar mais próximo sobre algoritmo

Tudo o que sabemos até agora do algoritmo simplex é a intuição geométrica vaga de que ele prossegue fazendo movimentos locais na superfície da região convexa factível, sucessivamente melhorando a função objetivo até que por fim chegue à solução ótima. Uma vez que o tenhamos estudado em mais detalhes (Seção 7.6), estaremos em posição de entender exatamente como lida com PLs de fluxo, o que é útil como fonte de inspiração para projetar algoritmos de fluxo máximo *diretamente*.

De fato o comportamento do simplex tem uma interpretação elementar:

Comece com fluxo zero.

Repita: Escolha um caminho apropriado de s até t e aumente o fluxo ao longo das arestas deste caminho o máximo possível.

A Figura 7.5(a)-(d) mostra um pequeno exemplo no qual o simplex pára depois de duas iterações. O fluxo final tem tamanho 2, o que é visto facilmente como ótimo.

Existe apenas uma complicação. O que aconteceria se tivéssemos escolhido um caminho diferente, o da Figura 7.5(e)? Ele dá apenas uma unidade de fluxo e já parece bloquear todos os outros caminhos. O simplex contorna o problema permitindo também que caminhos *cancelem um fluxo existente*. Nesse caso particular, ele escolheria em seguida o caminho da Figura 7.5(f). A aresta (b, a) do caminho não está na rede original e tem o efeito de cancelar o fluxo previamente atribuído à aresta (a, b) .

Em resumo, para cada iteração o simplex procura por um caminho $s - t$ cujas arestas (u, v) podem ser de dois tipos:

1. (u, v) está na rede original e ainda não esgotou sua capacidade.
2. A aresta reversa (v, u) está na rede original e existe algum fluxo ao longo dela.

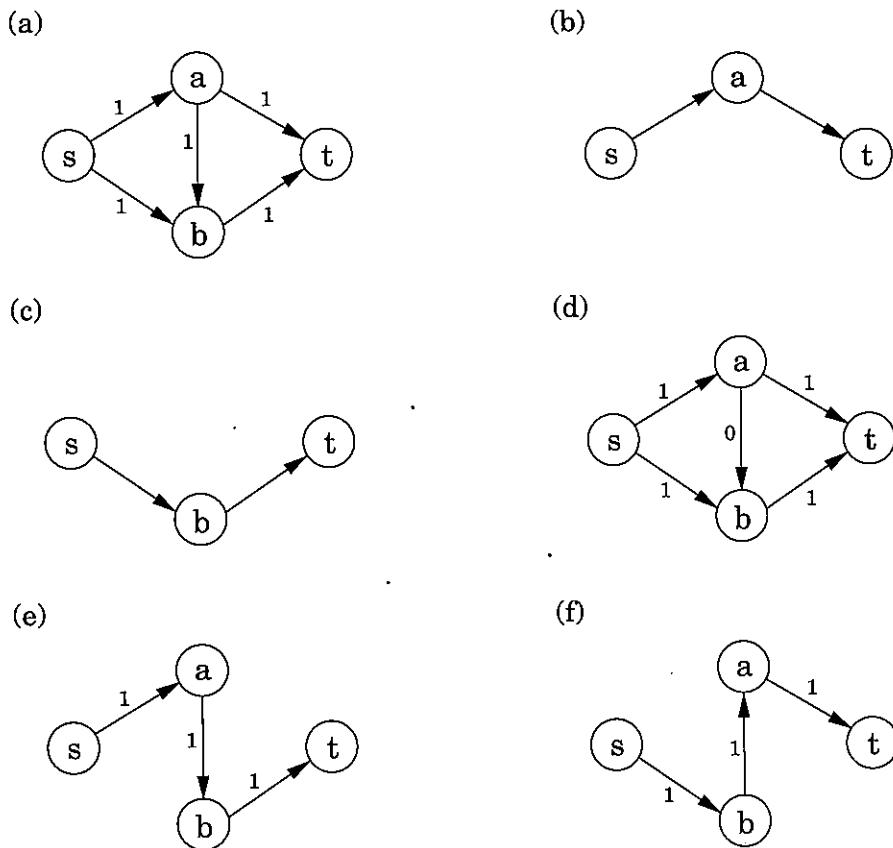
Se o fluxo atual é f , então no primeiro caso, a aresta (u, v) pode contribuir com $c_{uv} - f_{uv}$ unidades adicionais de fluxo e, no segundo caso, com f_{vu} unidades adicionais (cancelando todo ou parte do fluxo existente em (v, u)). Essas oportunidades de aumento de fluxo podem ser capturadas em uma *rede residual* $G^f = (V, E^f)$, que tem exatamente os dois tipos de arestas listados, com capacidades residuais c^f :

$$\begin{cases} c_{uv} - f_{uv} & \text{se } (u, v) \in E \text{ e } f_{uv} < c_{uv} \\ f_{vu} & \text{se } (v, u) \in E \text{ e } f_{vu} > 0. \end{cases}$$

Assim podemos de maneira equivalente pensar que o simplex escolhe um caminho $s - t$ na rede residual.

Simulando o comportamento do simplex, obtemos um algoritmo direto para resolver o fluxo máximo. Ele procede em iterações, a cada vez explicitamente construindo G^f ,

Figura 7.5 Uma ilustração do algoritmo de fluxo máximo. (a) Uma rede simples. (b) O primeiro caminho escolhido. (c) O segundo caminho escolhido. (d) O fluxo final. (e) Poderíamos ter escolhido este caminho primeiro. (f) E neste caso, teríamos de ter permitido este segundo caminho.



encontrando um caminho $s - t$ apropriado em G^f , usando, digamos, uma busca em largura de tempo linear, e parando se não existir mais nenhum tal caminho ao longo do qual o fluxo pode ser aumentado.

A Figura 7.6 ilustra o algoritmo no nosso exemplo do óleo.

7.2.4 Um certificado de otimalidade

Agora, vamos a um fato verdadeiramente memorável: o simplex não apenas computa o fluxo máximo, mas também gera uma prova curta da otimalidade deste fluxo!

Vejamos um exemplo do que queremos dizer. Particione os nós da rede de dutos (Figura 7.4) em dois grupos, $L = \{s, a, b\}$ e $R = \{c, d, e, t\}$:

Figura 7.6 O algoritmo de fluxo máximo aplicado à rede da Figura 7.4. Em cada iteração, o fluxo atual é apresentado na parte esquerda e a rede residual, na direita. Os caminhos escolhidos estão mostrados em negrito.

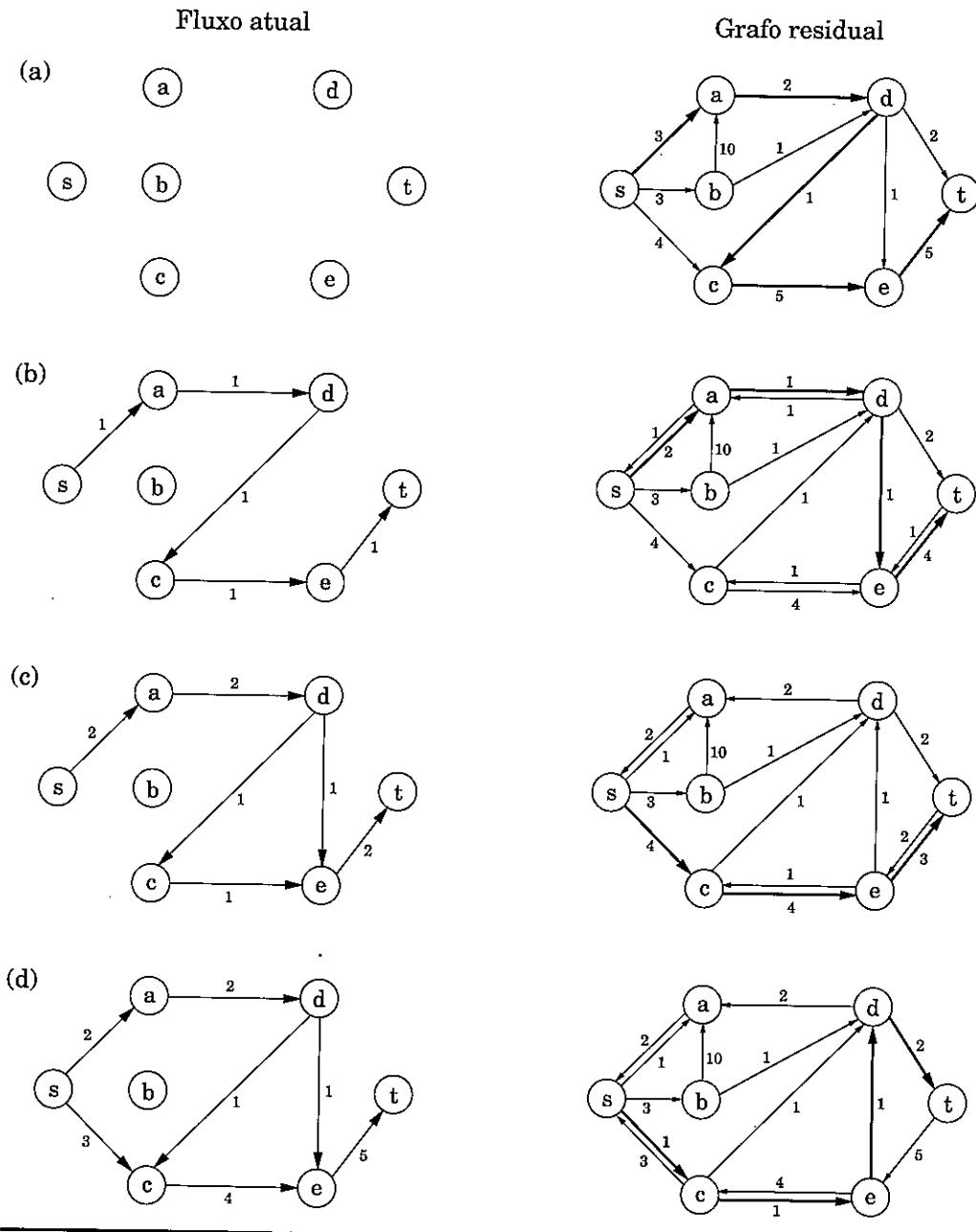
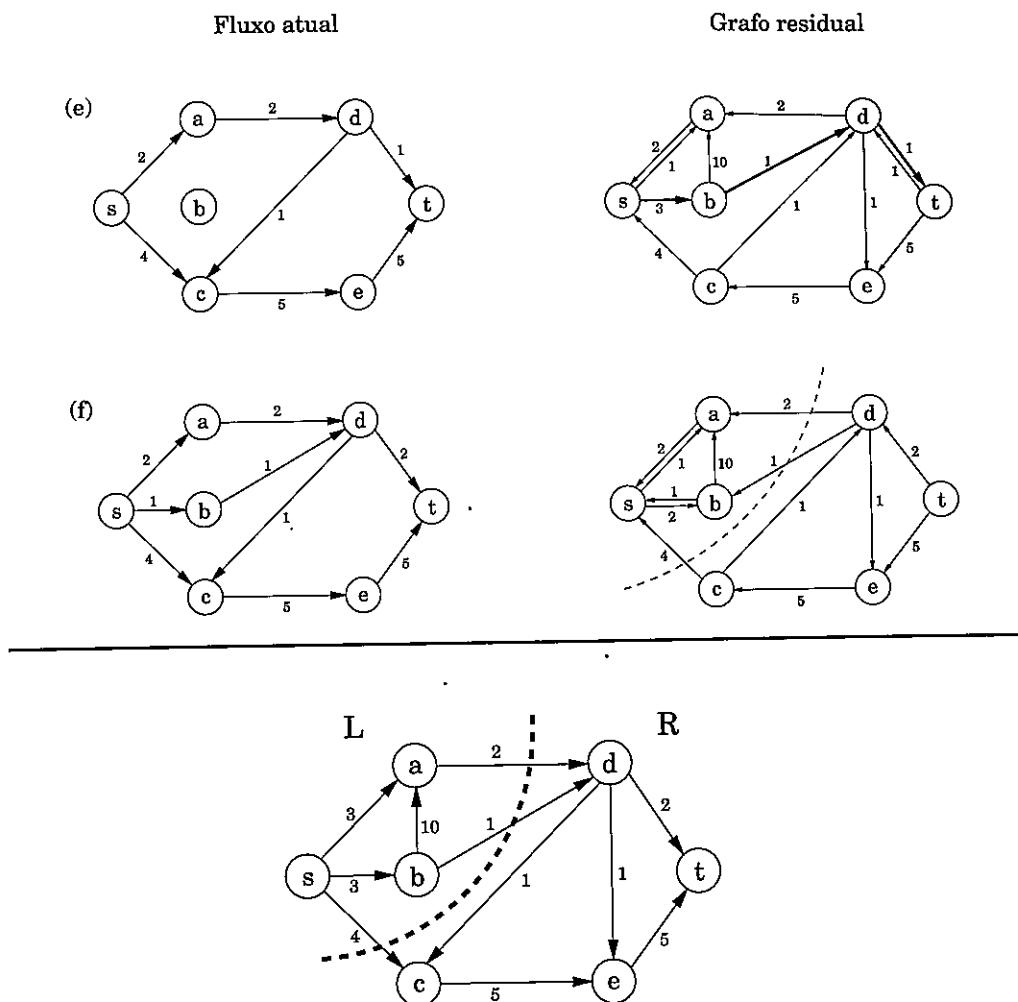


Figura 7.6 Continuação



Todo óleo bombeado tem de passar de L para R . Portanto, nenhum fluxo pode exceder a capacidade total das arestas de L para R , que é 7. Mas isso significa que o fluxo que achamos antes, de tamanho 7, tem de ser ótimo!

De modo mais geral, um corte (s, t) partitiona os vértices em dois grupos distintos L e R tal que s está em L e t está em R . Sua *capacidade* é a capacidade total das arestas de L para R e, como argumentado anteriormente, é uma cota superior sobre *qualquer* fluxo:

Selecione qualquer fluxo f e qualquer corte $(s, t) = (L, R)$. Então, $\text{tamanho}(f) \leq \text{capacidade}(L, R)$.

Alguns cortes são grandes e fornecem cotas superiores folgadas — o corte $\{\{s, b, c\}, \{a, d, e, t\}\}$ tem uma capacidade de 19. Mas existe também um corte de capacidade 7,

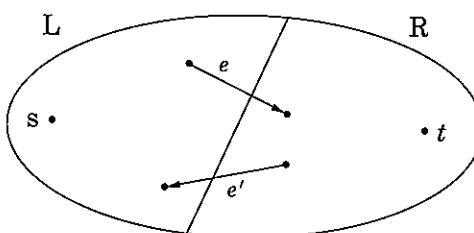
que é efetivamente um *certificado de optimidade* para o fluxo máximo. Isso não é apenas uma propriedade afortunada da nossa rede de dutos; tal corte *sempre* existe.

Teorema do corte-mínimo fluxo-máximo

O tamanho do fluxo máximo em uma rede é igual à capacidade do menor corte (s, t).

Além disso, nosso algoritmo automaticamente encontra este corte como um subproduto!

Vejamos por que isso é verdadeiro. Suponha que f seja o fluxo final quando o algoritmo termina. Sabemos que o nó t não é mais alcançável a partir de s na rede residual G^f . Seja L o conjunto de nós alcançáveis de s em G^f , e seja $R = V - L$ o restante dos nós. Então (L, R) é um corte no grafo G :



Afirmamos que

$$\text{tamanho}(f) = \text{capacidade}(L, R).$$

Para ver isso, observe que pela maneira com a qual L é definido, qualquer aresta saindo de L para R tem de estar na capacidade máxima (no fluxo atual f), e qualquer aresta de R para L tem de ter fluxo zero. (Assim, na figura, $f_e = c_e$ e $f_{e'} = 0$.) Portanto o fluxo líquido através de (L, R) é exatamente a capacidade do corte.

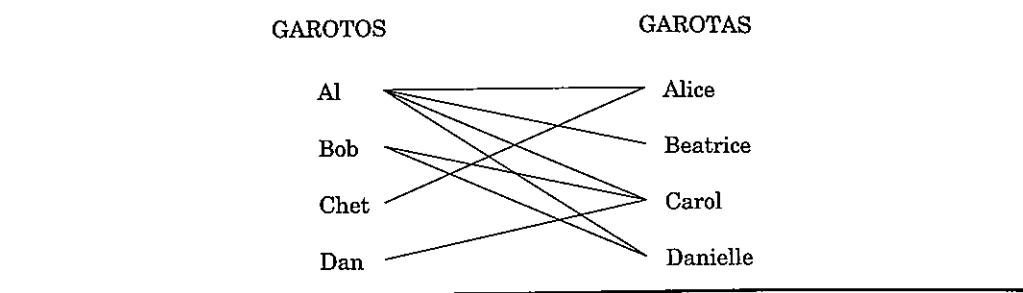
7.2.5 Eficiência

Cada iteração do nosso algoritmo de fluxo máximo é eficiente, requerendo tempo $O(|E|)$ se uma busca em profundidade ou largura é usada para encontrar um caminho $s - t$. Mas quantas iterações acontecem?

Suponha que todas as arestas na rede original tenham capacidade *inteira* $\leq C$. Então um argumento induutivo mostra que em cada iteração do algoritmo, o fluxo é sempre um inteiro e cresce por uma quantidade inteira. Portanto, como o fluxo máximo é no máximo $C|E|$ (por quê?), segue-se que o número de iterações é no máximo esse tanto. Mas isso é facilmente uma cota tranqüilizadora: o que dizer se C está na casa dos milhões?

Examinaremos novamente essa questão no Exercício 7.31. De fato é possível construir exemplos ruins nos quais o número de iterações é proporcional a C , se os caminhos $s - t$ não são escolhidos cuidadosamente. Entretanto, se os caminhos são escolhidos de maneira razoável — em particular, usando busca em largura, que encontra o caminho com menos arestas —, o número de iterações é no máximo $O(|V| \cdot |E|)$, não importa quais são as capacidades. Essa última cota leva a um tempo total de execução de $O(|V| \cdot |E|^2)$ para o fluxo máximo.

Figura 7.7 Uma aresta entre duas pessoas significa que elas gostam uma da outra. Será que é possível casar todo mundo com quem se gosta?

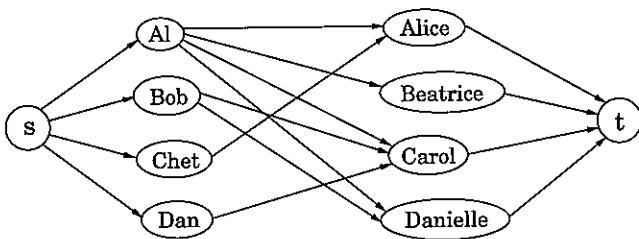


7.3 Emparelhamento bipartido

A Figura 7.7 mostra um grafo com quatro nós na parte esquerda representando garotos e quatro nós na parte direita representando garotas.¹ Existe uma aresta entre um garoto e uma garota se eles se gostam (por exemplo, Al gosta de todas as garotas). Será que é possível escolher casais tal que todos possuam exatamente um parceiro, de que eles gostam? Em jargão de teoria de grafos, existe um *emparelhamento perfeito*?

Esse jogo de emparelhamento pode ser reduzido ao problema do fluxo máximo e, com isso, à programação linear! Crie um novo nó fonte s , com arestas saindo para todos os garotos; um novo nó sorvedouro t , com arestas chegando de todas as garotas; e direcione todas as arestas no grafo original bipartido de garoto para garota (Figura 7.8). Por fim, dê a cada aresta a capacidade de 1. Então existe um emparelhamento perfeito se e somente se esta rede tem um fluxo cujo tamanho é igual ao número de casais. Você pode achar um tal fluxo no exemplo?

Figura 7.8 Uma rede de emparelhamento. Cada aresta tem a capacidade de um.



Na verdade, a situação é um pouco mais complicada do que foi dito: o que é fácil de ver é que o fluxo de *valor inteiro* ótimo corresponde ao emparelhamento ótimo.

¹Este tipo de grafo, no qual os nós podem ser particionados em dois grupos tal que todas as arestas são entre os grupos, é chamado *bipartido*.

Estaríamos um pouco perdidos interpretando um fluxo que bombeia 0,7 unidade ao longo da aresta Al-Carol, por exemplo! Felizmente, o problema do fluxo máximo tem a seguinte

Propriedade: se todas as capacidades de aresta são inteiras, o fluxo ótimo encontrado pelo algoritmo é inteiro. Podemos ver isso diretamente do algoritmo, que nesses casos incrementaria o fluxo por uma quantidade inteira em cada iteração.

Assim, integralidade acontece automaticamente no problema do fluxo máximo. Infelizmente, isto é a exceção em vez de a regra: como veremos no Capítulo 8, é um problema muito difícil encontrar a solução ótima (e a propósito, *qualquer* solução) de um programa linear geral, se também demandarmos que as variáveis sejam inteiras.

7.4 Dualidade

Vimos que em redes, fluxos são menores do que cortes, mas o fluxo máximo e o corte mínimo coincidem exatamente e cada um é, portanto, um certificado da otimalidade do outro. Mesmo já sendo um fenômeno memorável, agora o generalizamos de fluxo máximo para *qualquer* problema que possa ser resolvido por programação linear! Qualquer problema de maximização linear tem um problema de minimização *dual*, e eles se relacionam um com o outro da mesma maneira que fluxos e cortes.

Para entender o que é dualidade, reveja nosso PL introdutório com os dois tipos de chocolate:

$$\begin{aligned} \max \quad & x_1 + 6x_2 \\ \text{s.t.} \quad & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0. \end{aligned}$$

O simplex declara a solução ótima como $(x_1, x_2) = (100, 300)$, com valor objetivo de 1.900. Será que essa resposta pode ser verificada de algum jeito? Vejamos: suponha que peguemos a primeira inequação e a adicionemos seis vezes à segunda inequação. Obtemos

$$x_1 + 6x_2 \leq 2.000.$$

Isso é interessante, porque nos diz que é impossível alcançar um lucro de mais de 2.000. Será que podemos adicionar alguma outra combinação de restrições do PL e trazer essa cota superior ainda mais para perto de 1.900? Depois de alguma experimentação, encontramos que multiplicar as três inequações por 0, 5 e 1, respectivamente, e adicionando-as leva a

$$x_1 + 6x_2 \leq 1.900.$$

Portanto, 1.900 tem de ser mesmo o melhor valor possível! Os multiplicadores (0,5,1) magicamente constituem um *certificado de otimalidade*! É memorável que um tal certificado exista para este PL — e mesmo que soubéssemos que há um, como poderíamos sistematicamente encontrá-lo?

Vamos investigar o assunto descrevendo o que esperamos desses três multiplicadores, que chamaremos de y_1, y_2, y_3 .

Multiplicador	Inequação
y_1	$x_1 \leq 200$
y_2	$x_2 \leq 300$
y_3	$x_1 + x_2 \leq 400$

Para começar, estes y_i tem de ser não-negativos, pois caso contrário não se qualificam para multiplicar inequações (multiplicar uma inequação por um número negativo inverte-a \leq para \geq). Depois dos passos de multiplicação e adição, obtemos a cota:

$$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \leq 200y_1 + 300y_2 + 400y_3.$$

Queremos que o lado esquerdo se pareça com a nossa função objetivo $x_1 + 6x_2$ de modo que o lado direito seja uma cota superior sobre a solução ótima. Para tanto, precisamos que $y_1 + y_3$ seja 1 e $y_2 + y_3$ seja 6. Pensando bem, seria ótimo se $y_1 + y_3$ fosse maior do que 1 — o certificado resultante seria muito mais convincente. Assim, obtemos uma cota superior

$$x_1 + 6x_2 \leq 200y_1 + 300y_2 + 400y_3 \text{ se } \begin{cases} y_1, y_2, y_3 \geq 0 \\ y_1 + y_3 \geq 1 \\ y_2 + y_3 \geq 6 \end{cases}.$$

Podemos facilmente encontrar y que satisfaçam as inequações na direita simplesmente fazendo-os grandes o suficiente, por exemplo $(y_1, y_2, y_3) = (5, 3, 6)$. Mas esses multiplicadores particulares nos diriam que a solução ótima para o PL é no máximo $200 \cdot 5 + 300 \cdot 3 + 400 \cdot 6 = 4.300$, uma cota que é frouxa demais para ser de interesse. O que queremos é uma cota que seja o mais firme possível, assim devemos minimizar $200y_1 + 300y_2 + 400y_3$ sujeito às inequações anteriores. *E isso é um novo programa linear!*

Portanto, encontrar o conjunto de multiplicadores que dá a melhor cota superior para o nosso PL é equivalente a resolver um novo PL:

$$\begin{aligned} \min \quad & 200y_1 + 300y_2 + 400y_3 \\ \text{s.t.} \quad & y_1 + y_3 \geq 1 \\ & y_2 + y_3 \geq 6 \\ & y_1, y_2, y_3 \geq 0 \end{aligned}$$

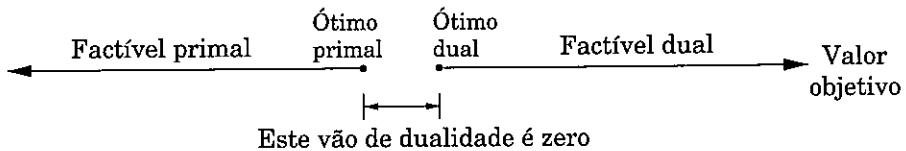
Por projeto, qualquer valor factível desse PL *dual* é uma cota superior sobre o PL *primal* original. Assim, se de alguma forma encontrarmos um par de valores factíveis primal e dual que sejam iguais, eles têm de ser ambos ótimos. Veja exatamente um tal par:

$$\text{Primal : } (x_1, x_2) = (100, 300); \quad \text{Dual : } (y_1, y_2, y_3) = (0, 5, 1).$$

Os dois têm valor 1.900 e, portanto, certificam a otimalidade um do outro (Figura 7.9).

Impressionantemente, isso não é um exemplo de sorte, mas um fenômeno geral. Para começar, a construção anterior — criar um multiplicador para cada restrição primal;

Figura 7.9 Por projeto, valores factíveis duais \geq valores factíveis primais. O teorema da dualidade nos diz que, além disso, seus valores ótimos coincidem.



escrever uma restrição no dual para cada variável do primal, na qual a soma tem de ser maior do que o coeficiente objetivo da variável primal correspondente; e otimizar a soma dos multiplicadores ponderados pelo lado direito do primal — pode ser realizada para qualquer PL, como mostra a Figura 7.10, e em generalidade ainda maior na Figura 7.11. A segunda figura tem uma adição notável: se o primal tem uma restrição de igualdade, o correspondente multiplicador (ou *variável dual*) não precisa ser não-negativo, porque a validade de equações é preservada quando multiplicadas por números negativos. Assim, os multiplicadores das equações são variáveis sem restrições. Note também a simetria simples entre os dois PLs, na qual a matriz $A = (a_{ij})$ define uma restrição primal com cada uma de suas *linhas*, e uma restrição dual com cada uma de suas *colunas*.

Por construção, qualquer solução factível do dual é uma cota superior sobre qualquer solução factível do primal. Mas, além disso, seus ótimos coincidem!

Teorema da dualidade: *Se um programa linear tem um ótimo limitado, então seu dual também tem, e os dois valores ótimos coincidem.*

Quando o primal é o PL que expressa o problema de fluxo máximo, é possível atribuir interpretações às variáveis duais que mostram que o dual é nenhum outro senão o problema do corte mínimo (Exercício 7.25). A relação entre fluxos e cortes é apenas uma instância específica do teorema da dualidade. E, de fato, a prova do teorema surge do algoritmo simplex, da mesma maneira com a qual o teorema do corte-mínimo fluxo-máximo surge da análise do algoritmo de fluxo máximo.

Figura 7.10 Um PL primal genérico na forma de matriz-vetor e seu dual.

PL Primal:

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

PL Dual:

$$\begin{aligned} \min \quad & y^T b \\ \text{s.t.} \quad & y^T A \geq c^T \\ & y \geq 0 \end{aligned}$$

Figura 7.11 No caso mais geral de programação linear, temos um conjunto I de inequações e um conjunto E de equações (um total de $m = |I| + |E|$ restrições) sobre n variáveis, das quais um subconjunto N possui restrição de não-negatividade. O dual tem $m = |I| + |E|$ variáveis, das quais somente aquelas correspondentes à I possuem restrição de não-negatividade.

PL Primal:

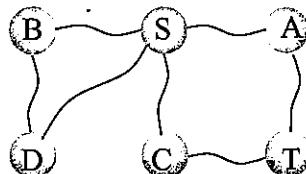
$$\begin{aligned} \max \quad & c_1x_1 + \cdots + c_nx_n \\ a_{i1}x_1 + \cdots + a_{in}x_n \leq & b_i \quad \text{para } i \in I \\ a_{i1}x_1 + \cdots + a_{in}x_n = & b_i \quad \text{para } i \in E \\ x_j \geq 0 \quad & \text{para } j \in N \end{aligned}$$

PL Dual:

$$\begin{aligned} \min \quad & b_1y_1 + \cdots + b_my_m \\ a_{1j}y_1 + \cdots + a_{mj}y_m \geq & c_j \quad \text{para } j \in N \\ a_{1j}y_1 + \cdots + a_{mj}y_m = & c_j \quad \text{para } j \notin N \\ y_i \geq 0 \quad & \text{para } i \in I \end{aligned}$$

Visualizando dualidade

Podemos resolver o problema do caminho mínimo com o seguinte dispositivo “análogico”: dado um grafo não-direcionado e com pesos nas arestas, construa um *modelo físico* dele no qual cada aresta é uma corda de comprimento igual ao peso da aresta e cada vértice é um nó de amarração onde as extremidades apropriadas das cordas são unidas. Então para encontrar o caminho mínimo de s para t , apenas fixe t e puxe s para longe de t , até que a estrutura fique tensionada.



Não há nada de memorável ou surpreendente sobre tudo isso até que notemos o seguinte: o problema do caminho mínimo é de *minimização*, certo? Então por que estamos *puxando* s para longe de t , uma ação cujo propósito é, obviamente, *maximização*? Resposta: ao puxar s para longe de t resolvemos o *dual* do problema do caminho mínimo! Esse dual tem uma forma bastante simples (Exercício 7.28), com uma variável x_u para cada nó u :

$$\begin{aligned} \max \quad & x_s - x_t \\ |x_u - x_v| \leq & w_{uv} \quad \text{para todas as arestas } \{u, v\}. \end{aligned}$$

Em palavras, o problema dual é afastar s de t o máximo possível, sujeito à restrição de que as extremidades de cada aresta $\{u, v\}$ estejam separadas por uma distância de no máximo w_{uv} .

7.5 Jogos de soma-zero

Podemos representar várias situações de conflito na vida por *jogos de matriz*. Por exemplo, o jogo infantil da *pedra-papel-tesoura* é especificado pela *matriz de perdas e ganhos* ilustrada aqui. Há dois jogadores, chamados Linha e Coluna, e cada um deles

seleciona um movimento de $\{r, p, s\}$. Eles, então, verificam a célula correspondente aos seus movimentos na matriz, e Coluna paga para Linha esse montante. É ganho para Linha e perda para Coluna.

		Coluna		
		r	p	s
Linha	r	0	-1	1
	p	1	0	-1
	s	-1	1	0

Agora suponha que os dois joguem repetidamente. Se Linha sempre fizer o mesmo movimento, Coluna vai rapidamente perceber e fará sempre o movimento contrário, ganhando sempre. Portanto, Linha deve embaralhar as coisas: podemos modelar isso permitindo que Linha tenha uma *estratégia mista*, na qual toda vez ela jogue r com probabilidade x_1 , p com probabilidade x_2 e s com probabilidade x_3 . Essa estratégia é especificada pelo vetor $\mathbf{x} = (x_1, x_2, x_3)$, números positivos que totalizam 1. De maneira similar, a estratégia mista de Coluna é algum $\mathbf{y} = (y_1, y_2, y_3)$.²

Em qualquer dada rodada do jogo, há uma chance de $x_j y_i$ de que Linha e Coluna joguem o i -ésimo e j -ésimo movimento, respectivamente. Portanto, o retorno *esperado* (médio) é

$$\sum_{i,j} G_{ij} \cdot \text{Prob}[\text{Linha joga } i, \text{Coluna joga } j] = \sum_{i,j} G_{ij} x_i y_j.$$

Linha quer *maximizar* isso, enquanto Coluna quer *minimizar*. Quais retornos eles podem esperar alcançar no pedra-papel-tesoura? Bem, suponha, por exemplo, que Linha jogue a estratégia “completamente aleatória” $\mathbf{x} = (1/3, 1/3, 1/3)$. Se Coluna jogar r , o retorno médio (lendo a primeira coluna da matriz do jogo) será

$$\frac{1}{3} \cdot 0 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot -1 = 0.$$

Isso também é verdadeiro se Coluna jogar p ou s . E como o retorno de qualquer estratégia mista (y_1, y_2, y_3) é simplesmente uma média ponderada dos retornos individuais de jogar r , p e s , ele tem de ser também zero. Isto pode ser visto diretamente da fórmula anterior,

$$\sum_{i,j} G_{ij} x_i y_j = \sum_{i,j} G_{ij} \cdot \frac{1}{3} y_j = \sum_j y_j \left(\sum_i \frac{1}{3} G_{ij} \right) = \sum_j y_j \cdot 0 = 0,$$

onde a penúltima equação é a observação de que cada coluna de G totaliza zero. Portanto, ao jogar a estratégia “completamente aleatória”, Linha força um retorno esperado de zero, *não importando o que Coluna faça*, ou seja, Coluna não pode ter esperança de um retorno (médio) negativo (lembre-se de que ele quer que o retorno seja o menor possível). Mas de maneira simétrica, se Coluna joga a estratégia completamente aleatória, ele também força um retorno esperado de zero e, assim, Linha não pode ter esperança de um retorno (médio) positivo. Em resumo, o melhor que cada jogador

² Também são de interesse cenários nos quais os jogadores alteram as estratégias de rodada para rodada. Contudo, situações desse tipo podem ficar muito complicadas e são um assunto bastante complexo.

pode fazer é jogar de maneira completamente aleatória, com um retorno esperado de zero. Confirmamos matematicamente aquilo que você já sabia desde muito sobre pedra-papel-tesoura!

Vamos pensar de uma forma ligeiramente diferente, considerando dois cenários:

1. Primeiro Linha anuncia sua estratégia e, depois, Coluna seleciona a sua.
2. Primeiro Coluna anuncia sua estratégia e, depois, Linha escolhe a dela.

Vimos que o retorno médio é o mesmo (zero) nos dois casos se as duas partes jogam de maneira ótima. Mas isso pode muito bem ser devido ao alto nível de simetria em pedra-papel-tesoura. Em jogos gerais, esperaríamos que a primeira opção favorecesse Coluna, pois ele sabe a estratégia de Linha e pode explorar tal fato ao escolher sua própria estratégia. De modo semelhante, esperaríamos que a segunda opção favorecesse Linha. Surpreendentemente, não é o caso: se ambos jogarem de maneira ótima, não prejudicará, ao jogador, anunciar sua estratégia antecipadamente! Além disso, essa propriedade memorável é uma consequência de — e, de fato, equivalente a — dualidade em programação linear.

Investiguemos com um jogo não-simétrico. Imagine um cenário de *eleição presidencial* no qual há dois candidatos para o cargo e os movimentos que eles fazem correspondem aos assuntos de campanha aos quais podem dar ênfase (as iniciais são de *economia, sociedade, moralidade e corte de impostos*). As células de retorno são milhões de votos perdidos por Coluna.

	m	t
e	3	-1
s	-2	1

Suponha que Linha anuncie que vai jogar a estratégia mista $x = (1/2, 1/2)$. O que Coluna deve fazer? Jogar m incorrerá em uma perda esperada de $1/2$, enquanto t incorrerá em uma perda de 0 . A melhor resposta de Coluna é, portanto, a estratégia *pura* $y = (0, 1)$.

Mas geralmente, uma vez que a estratégia de Linha $x = (x_1, x_2)$ esteja fixada, existe sempre uma estratégia *pura* que é ótima para Coluna: ou jogar m , com retorno $3x_1 - 2x_2$, ou t , com retorno $-x_1 + x_2$, o que for menor. Aliás, qualquer estratégia mista y é uma média ponderada dessas duas estratégias puras e, assim, não pode bater a melhor das duas.

Portanto, se Linha é forçada a anunciar x antes que Coluna jogue, Linha sabe que a melhor resposta de Coluna vai alcançar um retorno esperado de $\min\{3x_2 - 2x_1, -x_1 + x_2\}$. Linha deve escolher x *defensivamente* para maximizar seu retorno contra essa melhor resposta:

Selecione (x_1, x_2) que maximize

$$\min\{3x_2 - 2x_1, -x_1 + x_2\}$$

retorno da melhor resposta de Coluna para x

A escolha de x_i dá a Linha a melhor *garantia* possível sobre o seu retorno esperado. E veremos agora que pode ser encontrada por um PL! O principal truque é notar que para

x_1 e x_2 fixos, os seguintes termos são equivalentes:

$$\begin{array}{ll} \max z & \\ z = \min\{3x_1 - 2x_2, -x_1 + x_2\} & z \leq 3x_1 - 2x_2 \\ & z \leq -x_1 + x_2 \end{array}$$

E Linha precisa escolher x_1 e x_2 para maximizar este z .

$$\begin{array}{ll} \max z & \\ -3x_1 + 2x_2 + z \leq 0 & \\ x_1 - x_2 + z \leq 0 & \\ x_1 + x_2 = 1 & \\ x_1, x_2 \geq 0 & \end{array}$$

Simetricamente, se Coluna tem de anunciar sua estratégia primeiro, a melhor aposta é escolher a estratégia mista y que minimize sua perda sob a melhor resposta de Linha, em outras palavras,

$$\text{Selecionar } (y_1, y_2) \text{ que minimiza } \underbrace{\min\{3y_1 - y_2, -2y_1 + y_2\}}_{\text{retorno da melhor resposta de Linha para } y}$$

Na forma de PL,

$$\begin{array}{ll} \min w & \\ -3y_1 + y_2 + w \geq 0 & \\ 2y_1 - y_2 + w \geq 0 & \\ y_1 + y_2 = 1 & \\ y_1, y_2 \geq 0 & \end{array}$$

A observação crucial agora é que esses dois PL são duais um do outro (veja a Figura 7.11)! Assim, eles têm o mesmo ótimo, que chamaremos de V .

Vamos resumir. Resolvendo um PL, Linha (o maximizador) pode determinar uma estratégia para si que garante um retorno esperado de pelo menos V , não importando o que Coluna faça. E, resolvendo o PL dual, Coluna (o minimizador) pode garantir um retorno esperado de no máximo V , não importando o que Linha faça. Segue daí que essa é a única jogada ótima definida: *a priori* não havia sequer certeza de que uma jogada existia. V é conhecido como o *valor* do jogo. No nosso exemplo, ele é $1/7$ e é obtido quando Linha joga com sua estratégia mista ótima $(3/7, 4/7)$ e Coluna joga com sua estratégia mista ótima $(2/7, 5/7)$.

Esse exemplo é facilmente generalizado para jogos arbitrários e mostra a existência de estratégias mistas que são ótimas para ambos os jogadores e alcançam o mesmo valor — um resultado fundamental de teoria dos jogos chamado *teorema min-max*. Ele pode ser escrito na forma de equação como se segue:

$$\max_x \min_y \sum_{i,j} G_{ij} x_i y_j = \min_y \max_x \sum_{i,j} G_{ij} x_i y_j.$$

Isso é surpreendente, porque o lado esquerdo, no qual Linha tem de escolher sua estratégia primeiro, deveria presumidamente ser melhor para Coluna do que o lado direito, no qual ele tem de ir primeiro. Dualidade iguala os dois, como fez com fluxo máximo e corte mínimo.

7.6 O algoritmo simplex

O poder e expressividade extraordinárias de programas lineares seriam de pouco consolo se não tivéssemos uma maneira de resolvê-los eficientemente. Esse é o papel do algoritmo simplex.

Em alto nível, o algoritmo simplex toma um conjunto de inequações lineares e uma função objetivo linear e encontra o ponto factível com a seguinte estratégia:

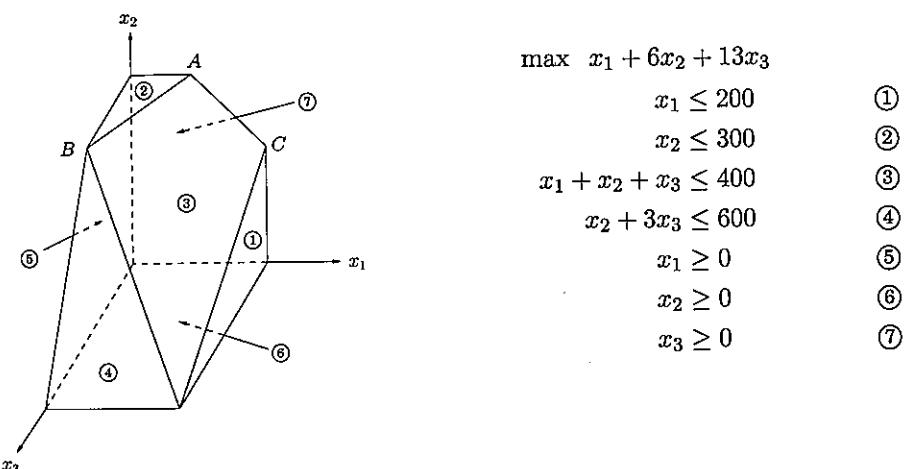
seja v qualquer vértice da região factível
enquanto existe um vizinho v' de v com valor objetivo melhor:
faça $v = v'$

Nos nossos exemplos 2D e 3D (Figura 7.1 e Figura 7.2), a situação era simples de visualizar e fez sentido intuitivo. Mas o que dizer se existirem n variáveis, x_1, \dots, x_n ?

Qualquer atribuição dos x_i pode ser representada por uma n -tupla de números reais e plotada no espaço n -dimensional. Uma equação linear envolvendo os x_i define um *hiperplano* neste mesmo espaço \mathbb{R}^n , e a correspondente inequação linear define um *semi-espacô*, todos os pontos que estão precisamente sobre o hiperplano ou estão em um particular lado dele. Por fim, a região factível do programa linear é especificada por um conjunto de inequações e é, portanto, a interseção dos correspondentes semi-espacos, um poliedro convexo.

Mas o que os conceitos de *vértice* e *vizinho* significam no contexto geral?

Figura 7.12 Um poliedro definido por sete inequações.



7.6.1 Vértices e vizinhos no espaço n-dimensional

A Figura 7.12 recorda um exemplo anterior. Examinando com atenção, vemos que *cada vértice é o ponto único no qual algum subconjunto de hiperplanos se encontra*. O vértice A, por exemplo, é o único ponto no qual as restrições ②, ③ e ⑦ são satisfeitas com igualdade. Por sua vez, os hiperplanos correspondentes às inequações ④ e ⑥ não definem um vértice, porque sua interseção não é apenas um único ponto, mas uma linha inteira.

Vamos tornar essa definição precisa.

Selecione um subconjunto das inequações. Se existir um único ponto que satisfaça todas elas com igualdade, e este ponto for factível, então ele será um vértice.

Quantas equações são necessárias para identificar unicamente um ponto? Quando há n variáveis, por um lado, precisamos de pelo menos n equações lineares se queremos uma solução única. Por outro lado, ter mais do que n equações é redundante: pelo menos uma delas pode ser reescrita como uma combinação linear das outras e pode, portanto, ser descartada. Em resumo,

Cada vértice é especificado por um conjunto de n inequações.³

Uma noção de *vizinho* segue agora naturalmente.

Dois vértices são *vizinhos* se eles têm em comum $n - 1$ das inequações que os definem.

Na Figura 7.12, por exemplo, vértices A e C compartilham as duas inequações $\{③, ⑦\}$ e são assim vizinhos.

7.6.2 O algoritmo

Em cada iteração, o simplex tem duas tarefas:

1. Checar se o vértice atual é ótimo (e se for, parar).
2. Determinar para onde se mover em seguida.

Como veremos, ambas as tarefas são fáceis se o vértice estiver na origem. E se o vértice estiver em outro lugar, vamos transformar o sistema de coordenadas para movê-lo para a origem!

Primeiro vejamos por que a origem é tão conveniente. Suponha que tenhamos algum PL genérico

$$\begin{aligned} \max \quad & \mathbf{c}^T \mathbf{x} \\ \text{Ax} \leq & \mathbf{b} \\ \mathbf{x} \geq & 0 \end{aligned}$$

onde x é o vetor de variáveis, $x = \{x_1, \dots, x_n\}$. Suponha que a origem seja factível. Então ela é certamente um vértice, pois é o único ponto no qual as n inequações $\{x_1 \geq 0, \dots, x_n \geq 0\}$ são *justas*. Agora vamos resolver nossas duas tarefas. Tarefa 1:

³Existe uma dificuldade aqui. É possível que o mesmo vértice seja gerado por diferentes subconjuntos de inequações. Na Figura 7.12, o vértice B é gerado por $\{②, ③, ④\}$, mas também por $\{②, ④, ⑥\}$. Tais vértices são chamados *degenerados* e requerem consideração especial. Consideraremos, por enquanto, que eles não existam, e retornaremos a eles mais tarde.

A origem é ótima se e somente se todos os $c_i \leq 0$.

Se todos os $c_i \leq 0$, então, considerando as restrições $x \geq 0$, não podemos ter esperança de um valor objetivo melhor. Pela direção contrária, se algum $c_i > 0$, então, a origem não é ótima, pois podemos aumentar a função objetivo aumentando x_i .

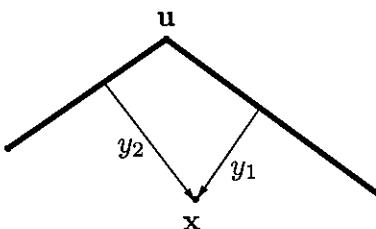
Assim, para a tarefa 2, podemos proceder aumentando algum x_i para o qual $c_i > 0$. Quanto podemos aumentá-lo? Até que atinjamos alguma outra restrição, ou seja, nós afrouxamos a restrição justa $x_i \geq 0$ e aumentamos x_i até que outra inequação, previamente frouxa, agora se torne justa. Neste ponto, temos novamente n inequações justas, portanto estamos em um novo vértice.

Por exemplo, suponha que estejamos lidando com o seguinte programa linear.

$$\begin{aligned} \max \quad & 2x_1 + 5x_2 \\ \text{s.t.} \quad & 2x_1 - x_2 \leq 4 \quad ① \\ & x_1 + 2x_2 \leq 9 \quad ② \\ & -x_1 + x_2 \leq 3 \quad ③ \\ & x_1 \geq 0 \quad ④ \\ & x_2 \geq 0 \quad ⑤ \end{aligned}$$

O simplex pode ser iniciado na origem, especificada pelas restrições ④ e ⑤. Para movermos, podemos afrouxar a restrição justa $x_2 \geq 0$. À medida que x_2 é gradualmente aumentado, a primeira restrição que encontra é $-x_1 + x_2 \leq 3$, e assim tem de parar em $x_2 = 3$, ponto no qual a restrição é justa. O novo vértice é, portanto, dado por ③ e ④.

Desse modo sabemos o que fazer se estamos na origem. Mas o que dizer se nosso vértice atual u está em outro lugar? O truque é transformar u na origem, transformando o sistema de coordenadas de (x_1, \dots, x_n) para a "vista local" partindo de u . Essas coordenadas locais consistem em (escaladas apropriadamente) distâncias y_1, \dots, y_n para os n hiperplanos (inequações) que definem e contêm u :



Especificamente, se uma dessas inequações é $a_i \cdot x \leq b_i$, então a distância de um ponto x para aquela particular "parede" é

$$y_i = b_i - a_i \cdot x.$$

As n equações desse tipo, uma por parede, definem os y_i como funções lineares dos x_i , e este relacionamento pode ser invertido para expressar os x_i como uma função linear dos y_i . Assim, podemos reescrever o PL inteiro em termos dos y . Isso não muda fundamentalmente (por exemplo, o valor ótimo permanece o mesmo), mas o expressa em um esquema de coordenadas diferente. O PL renovado, "local", tem as seguintes três propriedades:

1. Ele inclui as inequações $y \geq 0$, que são simplesmente as versões transformadas das inequações definindo u .
2. O próprio u é a origem do y -espaço.
3. A função de custo torna-se $\max c_u + \tilde{c}^T y$, onde c_u é o valor da função objetivo em u e \tilde{c} é um vetor de custo transformado.

Em resumo, estamos de volta à situação com a qual sabemos lidar! A Figura 7.13 mostra o algoritmo em ação, continuando nosso exemplo anterior.

O algoritmo simplex está completamente definido agora. Ele se move de um vértice para o vértice vizinho, parando quando a função objetivo é localmente ótima, isto é, quando as coordenadas do vetor de custo local são todas zero ou negativas. Como acabamos de ver, um vértice com essa propriedade tem de ser também globalmente ótimo. Por sua vez, se o vértice atual não é localmente ótimo, então seu sistema de coordenadas local inclui alguma dimensão ao longo da qual a função objetivo pode ser melhorada, assim nós nos movemos nesta direção — ao longo da aresta do poliedro — até atingir um vértice vizinho. Pela hipótese da ausência de degenerações (veja a nota de rodapé 3 na Seção 7.6.1), esta aresta tem tamanho não-nulo, e portanto estritamente aumentamos o valor objetivo. Dessa maneira, o processo tem de parar em algum momento.

7.6.3 Questões a explicar

Existem várias questões importantes no algoritmo simplex que ainda não mencionamos.

O vértice inicial

Como encontramos um vértice no qual começar o simplex? Em nossos exemplos 2D e 3D sempre começamos pela origem, o que funcionou porque aconteceu de os programas lineares terem inequações com o lado direito positivo. Em um PL geral não seremos sempre tão afortunados. Contudo encontrar um vértice inicial *pode ser reduzido a um PL e resolvido pelo simplex!*

Para entender como isso é feito, comece com um programa linear na forma-padrão (releia a Seção 7.1.4), pois sabemos que PLs podem sempre ser reescritos daquela maneira.

$$\min c^T x \text{ tal que } Ax = b \text{ e } x \geq 0.$$

Primeiro nos asseguramos de que os lados direitos das inequações sejam todos não-negativos: se $b_i < 0$, simplesmente multiplique ambos os lados da i -ésima equação por -1 .

Então criamos um novo PL como se segue:

- Criamos m novas *variáveis artificiais* $z_1, \dots, z_m \geq 0$, onde m é o número de equações.
- Adicionamos z_i ao lado esquerdo da i -ésima equação.
- Fazemos $z_1 + z_2 + \dots + z_m$ ser o objetivo a ser *minimizado*.

Para esse novo PL, é fácil obter um vértice inicial, ou seja, aquele com $z_i = b_i$ para todo i e todas as outras variáveis zeradas. Portanto podemos resolvê-lo com o simplex, para obter a solução ótima.

Figura 7.13 O simplex em ação.

<p>PL inicial:</p> $\begin{aligned} \max & 2x_1 + 5x_2 \\ 2x_1 - x_2 &\leq 4 \quad ① \\ x_1 + 2x_2 &\leq 9 \quad ② \\ -x_1 + x_2 &\leq 3 \quad ③ \\ x_1 &\geq 0 \quad ④ \\ x_2 &\geq 0 \quad ⑤ \end{aligned}$	<p>Vértice atual: {④, ⑤} (origem). Valor objetivo: 0.</p> <p>Movimento: aumentar x_2. ⑤ é afrouxada, ③ torna-se justa. Parar em $x_2 = 3$.</p> <p>Novo vértice: {④, ③} tem coordenadas locais (y_1, y_2):</p> $y_1 = x_1, \quad y_2 = 3 + x_1 - x_2$
<p>PL reescrito:</p> $\begin{aligned} \max & 15 + 7y_1 - 5y_2 \\ y_1 + y_2 &\leq 7 \quad ① \\ 3y_1 - 2y_2 &\leq 3 \quad ② \\ y_2 &\geq 0 \quad ③ \\ y_1 &\geq 0 \quad ④ \\ -y_1 + y_2 &\leq 3 \quad ⑤ \end{aligned}$	<p>Vértice atual: {④, ③}. Valor objetivo: 15.</p> <p>Movimento: aumentar y_1. ④ é afrouxada, ② torna-se justa. Parar em $y_1 = 1$.</p> <p>Novo vértice: {②, ③} tem coordenadas locais (z_1, z_2):</p> $z_1 = 3 - 3y_1 + 2y_2, \quad z_2 = y_2$
<p>PL reescrito:</p> $\begin{aligned} \max & 22 - \frac{7}{3}z_1 - \frac{1}{3}z_2 \\ -\frac{1}{3}z_1 + \frac{5}{3}z_2 &\leq 6 \quad ① \\ z_1 &\geq 0 \quad ② \\ z_2 &\geq 0 \quad ③ \\ \frac{1}{3}z_1 - \frac{2}{3}z_2 &\leq 1 \quad ④ \\ \frac{1}{3}z_1 + \frac{1}{3}z_2 &\leq 4 \quad ⑤ \end{aligned}$	<p>Vértice atual: {②, ③}. Valor objetivo: 22.</p> <p>Ótimo: todos os $c_i < 0$.</p> <p>Resolver ②, ③ (no PL original) para obter a solução ótima $(x_1, x_2) = (1, 4)$.</p>

Há dois casos. Se o valor ótimo de $z_1 + \dots + z_m$ é zero, então todos os z_i obtidos pelo simplex são zero e, assim, do vértice ótimo do novo PL nós obtemos um vértice factível inicial do PL original, simplesmente ignorando os z_i . Podemos por fim começar o simplex!

Mas o que dizer se o objetivo ótimo for positivo? Vamos pensar. Tentamos minimizar a soma dos z_i , mas o simplex decidiu que ela não pode ser zero. Mas, isso significa que o programa linear original é infactível: ele *precisa* de alguns z_i não-nulos para se tornar factível. É desse modo que o simplex descobre e reporta que um PL é infactível.

Degeneração

No poliedro da Figura 7.12 o vértice B é *degenerado*. Geometricamente, significa que ele é a interseção de mais do que $n = 3$ faces do poliedro (neste caso, ②, ③, ④, ⑤). Algebricamente, significa que se escolhermos qualquer um de quatro conjuntos de três inequações ($\{②, ③, ④\}$, $\{②, ③, ⑤\}$, $\{②, ④, ⑤\}$ e $\{③, ④, ⑤\}$) e resolvêrmos o sistema correspondente de três equações lineares em três variáveis obteremos a mesma solução em todos os quatro casos: $(0, 300, 100)$. Isso é um problema sério: o simplex pode retornar um vértice degenerado subótimo simplesmente porque todos os seus vizinhos são idênticos a ele e, assim, não têm objetivo melhor. E se modificarmos o simplex de modo que ele detecte degeneração e continue a pular de vértice para vértice apesar da falta de melhora no custo, ele pode terminar entrando em loop.

Uma maneira de consertar isso é com uma *perturbação*: mude cada b_i por uma quantidade aleatória minúscula para $b_i \pm \epsilon_i$. Isso não muda a essência do PL pois os ϵ_i são minúsculos, mas têm o efeito de diferenciar entre as soluções dos sistemas lineares. Para ver por que geometricamente, imagine que os quatro planos ②, ③, ④, ⑤ sejam movidos um pouquinho. O vértice B não iria se dividir em dois vértices, bem perto um do outro?

Ausência de limite

Em alguns casos um PL é ilimitado, tal que a sua função objetivo pode ser feita arbitrariamente grande (ou pequena, se é um problema de minimização). Se esse é o caso, o simplex vai perceber: ao explorar a vizinhança de um vértice, ele vai notar que retirar uma inequação e adicionar outra leva a um sistema não-determinado de equações que tem infinitas soluções. E de fato (isto é um teste fácil) o espaço de soluções contém uma linha inteira ao longo da qual o objetivo pode se tornar maior e maior, até ∞ . Nesse caso o simplex pára e reclama.

7.6.4 O tempo de execução do simplex

Qual é o tempo de execução do simplex para um programa linear genérico

$$\max \mathbf{c}^T \mathbf{x} \text{ tal que } \mathbf{A}\mathbf{x} \leq \mathbf{0} \text{ e } \mathbf{x} \geq \mathbf{0},$$

onde existem n variáveis e \mathbf{A} contém m restrições de inequações? Como ele é um algoritmo iterativo que procede de vértice para vértice, vamos começar computando o tempo tomado por uma única iteração. Suponha que o vértice atual seja \mathbf{u} . Por definição, ele é o ponto único no qual n restrições de inequação são satisfeitas com igualdade. Cada um de seus vizinhos compartilham $n - 1$ dessas inequações, portanto \mathbf{u} pode ter no máximo $n \cdot m$ vizinhos: escolha qual inequação retirar e qual nova adicionar.

Uma maneira ingênua de realizar a iteração seria checar cada vizinho em potencial para ver se ele realmente é um vértice do poliedro e determinar seu custo.

Eliminação gaussiana

Sob a nossa definição algébrica, meramente escrever as coordenadas de um vértice envolve resolver um sistema de equações lineares. Como isso é feito?

É dado um sistema de n equações lineares em n variáveis, digamos $n = 4$ e

$$\begin{array}{rcl} x_1 & - 2x_3 & = 2 \\ x_2 + x_3 & = 3 \\ x_1 + x_2 & - x_4 & = 4 \\ x_2 + 3x_3 + x_4 & = 5 \end{array}$$

O método da escola para resolver tais sistemas é aplicar repetidamente a seguinte regra: se adicionamos um múltiplo de uma equação a uma outra equação, o sistema de equações total permanece equivalente. Por exemplo, adicionando -1 vezes a primeira equação à terceira, obtemos o sistema equivalente

$$\begin{array}{rcl} x_1 & - 2x_3 & = 2 \\ x_2 + x_3 & = 3 \\ x_1 + 2x_3 - x_4 & = 2 \\ x_2 + 3x_3 + x_4 & = 5 \end{array}$$

Essa transformação é inteligente no seguinte sentido: ela *elimina* a variável x_1 da terceira equação, deixando apenas uma equação com x_1 . Em outras palavras, ignorando a primeira equação, temos um sistema de três equações e três variáveis: diminuímos n por $1!$ Podemos resolver este sistema menor para obter x_2, x_3, x_4 e então substituí-los na primeira equação para obter x_1 .

Isso sugere um algoritmo — mais uma vez devido a Gauss.

procedimento gauss(E, X)

Entrada: Um sistema $E = \{e_1, \dots, e_n\}$ de equações em n variáveis $X = \{x_1, \dots, x_n\}$:

$e_1: a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1; \dots; e_n: a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n$

Saída: Uma solução do sistema, se existir

Se todos os coeficientes a_{ij} são zero:

pare com a mensagem "infactível ou não linearmente independente"

se $n = 1$: retornar b_1/a_{11}

escolha o coeficiente a_{p1} de maior magnitude e troque as equações e_1, e_p
para $i = 2$ até n :

$$e_i = e_i - (a_{ii}/a_{p1}) \cdot e_p$$

$(x_2, \dots, x_n) = \text{gauss}(E - \{e_1\}, X - \{x_1\})$

$$x_1 = (b_1 - \sum_{j>1} a_{1j}x_j)/a_{11}$$

retornar (x_1, \dots, x_n)

(Ao escolhermos a equação a ser trocada pela primeira, selecionamos aquela com o maior $|a_{p1}|$ por razões de *precisão numérica*, porque vamos dividir por a_{p1} .)

Eliminação gaussiana usa $O(n^2)$ operações aritméticas para reduzir o tamanho do problema de n para $n - 1$ e, assim, usa $O(n^3)$ operações ao todo. Para mostrarmos que isso é também uma boa estimativa do tempo de execução total, precisamos argumentar que os números envolvidos permanecem polinomialmente limitados — por exemplo, que a solução (x_1, \dots, x_n) não requer muito mais precisão para ser representada do que os coeficientes originais a_{ij} e b_i . Você entende por quê?

Encontrar o custo é rápido, apenas um produto interno, mas checar se é um vértice de verdade envolve resolver um sistema de n equações e n variáveis (isto é, satisfazendo as n inequações escolhidas exatamente) e checar se o resultado é factível. Por eliminação gaussiana (veja o quadro anterior) isto toma tempo $O(n^3)$, levando a um tempo de execução não desejado de $O(mn^4)$ por iteração.

Felizmente, existe uma maneira muito melhor, e o fator mn^4 pode ser aperfeiçoado para mn , fazendo do simplex um algoritmo prático. Lembre-se de nossa discussão anterior (Seção 7.6.2) sobre a *visão local* partindo do vértice \mathbf{u} . Acontece que o custo extra por iteração de reescrever o PL em termos das coordenadas locais atuais é apenas $O((m+n)n)$; isso explora o fato de que a visão local muda apenas ligeiramente entre iterações, em apenas uma das inequações que a definem.

Depois, para selecionar o melhor vizinho, lembramos que a (visão local da) função objetivo é da forma “ $\max c_{\mathbf{u}} + \tilde{\mathbf{c}} \cdot \mathbf{y}$ ” onde $c_{\mathbf{u}}$ é o valor da função objetivo em \mathbf{u} . Isso imediatamente identifica uma direção promissora para o movimento: selecionamos qualquer $\tilde{c}_i > 0$ (se não existe nenhum, então o vértice atual é ótimo e o simplex pára). Como o restante do PL agora foi reescrito em termos das coordenadas \mathbf{y} , é fácil determinar quanto y_i pode ser aumentado antes de alguma inequação ser violada. (E se podemos aumentar y_i indefinidamente, sabemos que o PL é ilimitado.)

Segue daí que o tempo de execução por iteração do simplex é apenas $O(mn)$. Mas quantas iterações poderia haver? Naturalmente, não pode haver mais do que $\binom{m+n}{n}$, uma cota superior sobre o número de vértices. Mas esta cota é exponencial em n . E, de fato, há exemplos de PLs para os quais o simplex realiza mesmo um número exponencial de iterações. Em outras palavras, o simplex é um algoritmo de tempo exponencial. Entretanto, tais exemplos exponenciais não ocorrem na prática e é este fato que faz o simplex tão valioso e tão amplamente usado.

Programação linear em tempo polinomial

O simplex não é um algoritmo polinomial. Certos tipos raros de programas lineares fazem com que ele vá de um canto da região factível para um canto melhor e, então, para um melhor ainda e assim por diante por um número exponencial de passos. Por muito tempo, programação linear foi considerada um paradoxo, um problema que pode ser resolvido na prática, mas não em teoria!

Então, em 1979, um jovem matemático soviético Leonid Khachiyan descobriu o *algoritmo do elipsóide*, que é bastante diferente do simplex: extremamente simples na sua concepção (mas sofisticado em sua prova) e, ainda assim, resolve qualquer programa linear em tempo polinomial. Em vez de procurar a solução de um canto do poliedro para o próximo, o algoritmo de Khachiyan o confina em elipsóides (bolas alongadas e/ou achatadas em alta dimensão) cada vez menores. Quando o algoritmo foi anunciado, tornou-se uma espécie de “Sputnik matemático”, um feito bombástico que preocupou o status americano, em plena Guerra Fria, sobre a possível superioridade científica da União Soviética. O algoritmo do elipsóide se revelou um importante avanço teórico, mas não competiu bem com o simplex na prática. O paradoxo da programação linear se aprofundou: um problema com dois algoritmos, um que é eficiente em teoria e um que é eficiente na prática!

Programação linear em tempo polinomial (*continuação*)

Alguns poucos anos depois, Narendra Karmarkar, um estudante de pós-graduação na U.C. Berkeley, apareceu com uma idéia completamente diferente, que levou a outro algoritmo provadamente polinomial para programação linear. O algoritmo de Karmarkar é conhecido como o *método do ponto interior*, porque faz exatamente isto: busca o canto ótimo não pulando de canto em canto na superfície do poliedro, como o simplex faz, mas desenhando um caminho inteligente no interior do poliedro. E ele tem de fato uma boa performance na prática.

Mas talvez o maior avanço em algoritmos para programação linear não tenha sido a descoberta teórica de Khachiyan ou a abordagem inovadora de Karmarkar, mas uma consequência inesperada dessa última: a competição feroz entre as duas abordagens, simplex e ponto interior, que resultou no desenvolvimento de código muito rápido para programação linear.

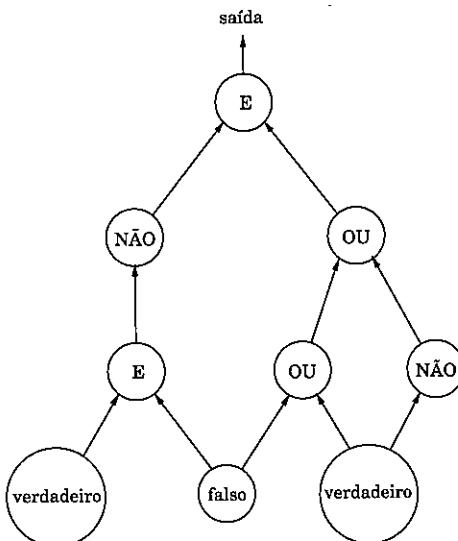
7.7 Pós-escrito: avaliação de circuito

A importância de programação linear se origina da variedade impressionante de problemas que se reduzem a ela e, por isso, dão testemunha do seu poder de expressão. Em certo sentido, a próxima aplicação é a de *maior ordem*.

É dado um *círcuito booleano*, ou seja, um dag de portas dos seguintes tipos.

- Portas de *entrada* têm grau de entrada zero, com valor *verdadeiro* ou *falso*.
- Portas *E* e portas *OU* têm grau de entrada 2.
- Portas *NÃO* têm grau de entrada 1.

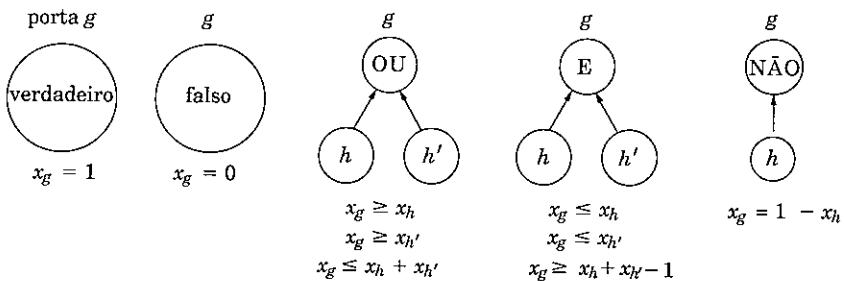
Além disso, uma das portas é designada como a *saída*. Veja um exemplo.



O problema do **VALOR DE CIRCUITO** é o seguinte: quando as leis da lógica booleana são aplicadas às portas em ordem topológica, a saída avalia para *verdadeiro*?

Há uma maneira simples, automática, de traduzir esse problema em um programa linear. Crie uma variável x_g para cada porta g , com restrições $0 \leq x_g \leq 1$. Adicione

as demais restrições para cada tipo de porta:



Tais restrições forçam todas as portas a tomar exatamente os valores corretos — 0 para **falso**, e 1 para **verdadeiro**. Não precisamos maximizar ou minimizar coisa alguma e podemos ler a resposta na variável x_0 correspondente à porta de saída.

Isso é uma redução direta à programação linear de um problema que pode não parecer muito interessante à primeira vista. Entretanto, o problema do **VALOR DE CIRCUITO** é em certo sentido *o problema mais geral solúvel em tempo polinomial!* Afinal, todo algoritmo em algum momento rodará em um computador, e o computador, em último sentido, é um circuito combinacional booleano implementado em um chip. Se o algoritmo executa em tempo polinomial, ele pode ser realizado como um circuito booleano consistindo em um número polinomial de cópias do circuito do computador, uma por unidade de tempo, com os valores das portas em uma camada usados para computar os valores para a próxima. Assim, o fato de o **VALOR DE CIRCUITO** se reduzir à programação linear significa que *todos os problemas que podem ser resolvidos em tempo polinomial também se reduzem!*

No nosso próximo tópico, **NP-completude**, veremos que muitos problemas *difíceis* se reduzem da mesma maneira à *programação inteira*, o gêmeo difícil de programação linear.

Outra consideração final: de que outro modo o problema da avaliação de circuito pode ser resolvido? Vamos pensar — um circuito é um dag. E que técnica algorítmica é mais adequada para resolver problemas em dags? Isso mesmo: uma programação dinâmica! Juntamente com programação linear, as duas técnicas algorítmicas mais gerais do mundo.

Exercícios

7.1. Considere o seguinte programa linear.

$$\begin{aligned}
 & \text{maximizar } 5x + 3y \\
 & 5x - 2y \geq 0 \\
 & x + y \leq 7 \\
 & x \leq 5 \\
 & x \geq 0 \\
 & y \geq 0
 \end{aligned}$$

Desenhe a região factível e identifique a solução ótima.

- 7.2. Trigo é produzido em Kansas e no México e é consumido em Nova York e na Califórnia. Kansas produz 15 mil toneladas de trigo e o México, 8. Por sua vez, Nova York consome 10 mil toneladas e a Califórnia, 13. O custo de transporte por mil toneladas são \$4 do México para Nova York, \$1 do México para Califórnia, \$2 do Kansas para Nova York, e \$3 do Kansas para Califórnia.

Escreva um programa linear que decida as quantidades de trigo (em milhares de toneladas e frações de milhares de toneladas) a serem transportados de cada produtor para cada consumidor, de modo que minimize o custo total de transporte.

- 7.3. Um avião cargueiro pode carregar um peso máximo de 100 toneladas e um volume máximo de 60 metros cúbicos. Há três materiais a serem transportados e a companhia cargueira pode escolher carregar qualquer quantidade de cada, até os limites máximos disponíveis dados a seguir.

- Material 1 tem densidade 2 ton/metro cúbico, quantidade máxima disponível 40 metros cúbicos, e receita de \$1.000 por metro cúbico.
- Material 2 tem densidade 1 ton/metro cúbico, quantidade máxima disponível 30 metros cúbicos, e receita de \$1.200 por metro cúbico.
- Material 3 tem densidade 3 ton/metro cúbico, quantidade máxima disponível 20 metros cúbicos, e receita de \$12.000 por metro cúbico.

Escreva um programa linear que otimize a receita dentro das restrições.

- 7.4. Moe está decidindo quanta cerveja Duff regular e quanta cerveja Duff forte encomendar a cada semana. Duff regular custa a Moe \$1 por caneco e ele a vende por \$2 por caneco; Duff Forte custa \$1,50 por caneco e ele vende por \$3 por caneco. Entretanto, como parte de uma complicada fraude de marketing, a companhia Duff somente vende um caneco de Duff Forte para cada dois canecos ou mais de Duff regular que Moe compra. Além disso, devido a eventos passados sobre os quais é melhor nem comentar, Duff não venderá a Moe mais do que 3.000 canecos por semana. Moe sabe que ele pode vender tanta cerveja quanto tiver. Formule um programa linear para decidir quanto de Duff regular e quanto de Duff Forte comprar, para maximizar o lucro de Moe. Resolva o problema geometricamente.

- 7.5. A companhia de produtos caninos oferece duas comidas para cachorro: Frisky Pup e Husky Hound, que são feitas de uma mistura de cereais e carne. Um pacote de Frisky Pup requer 1 quilo de cereal e 1,5 quilo de carne, e é vendido por \$7. Um pacote de Husky Hound usa 2 quilos de cereal e 1 quilo de carne, e é vendido por \$6. O cereal bruto custa \$1 por quilo e a carne bruta, \$2 por quilo. Há também o custo de \$1,40 para empacotar o Frisky Pup e \$0,60 para o Husky Hound. Um total de 240.000 quilos de cereal e 180.000 quilos de carne estão disponíveis a cada mês. O único gargalo de produção está no fato de a fábrica poder empacotar apenas 110.000 pacotes de Frisky Pup por mês. Desnecessário dizer, a gerência gostaria de maximizar o lucro.

- Formule o problema como um programa linear em duas variáveis.
- Desenhe a região factível, dê as coordenadas de cada vértice, e circule o vértice que maximiza o lucro. Qual o lucro máximo possível?

- 7.6. Dê um exemplo de um programa linear em duas variáveis cuja região factível é infinita, mas que exista uma solução ótima de custo limitado.
- 7.7. Encontre condições necessárias e suficientes sobre os reais a e b sob as quais o programa linear

$$\begin{aligned} & \max x + y \\ & ax + by \leq 1 \\ & x, y \geq 0 \end{aligned}$$

- (a) É infactível.
 (b) É ilimitado.
 (c) Tem uma solução ótima única e finita.

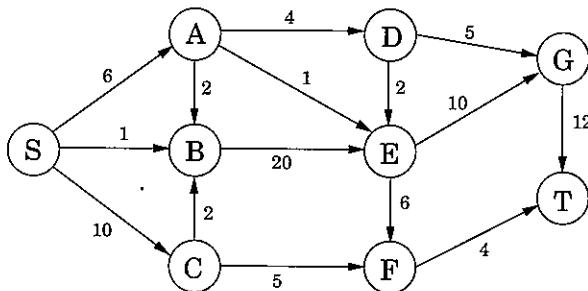
- 7.8. São dados os seguintes pontos no plano:

$$(1, 3), (2, 5), (3, 7), (5, 11), (7, 14), (8, 15), (10, 19).$$

Você quer encontrar uma linha $ax + by = c$ que passe aproximadamente por esses pontos (nenhuma linha passará perfeitamente por eles). Escreva um programa linear (você não precisa resolvê-lo) para encontrar a linha que minimize o *erro máximo absoluto*,

$$\max_{1 \leq i \leq 7} |ax_i + by_i - c|.$$

- 7.9. Um problema de programação quadrática busca maximizar uma função objetivo quadrática (com termos como $3x_1^2$ ou $5x_1x_2$) sujeito a um conjunto de restrições lineares. Dê um exemplo de um programa quadrático em duas variáveis x_1, x_2 , tal que a região factível seja não-vazia e limitada e, ainda assim, nenhum dos vértices dessa região otimize o objetivo (quadrático).
- 7.10. Para a seguinte rede, com capacidades nas arestas como mostrado, encontre o fluxo máximo de S até T , e também um corte correspondente a esse fluxo.



- 7.11. Escreva o dual do seguinte programa linear.

$$\begin{aligned} & \max x + y \\ & 2x + y \leq 3 \\ & x + 3y \leq 5 \\ & x, y \geq 0 \end{aligned}$$

Encontre as soluções ótimas do PL primal e do dual.

7.12. Para o programa linear

$$\begin{aligned} \max & x_1 - 2x_3 \\ & x_1 - x_2 \leq 1 \\ & 2x_2 - x_3 \leq 1 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

prove que a solução $(x_1, x_2, x_3) = (3/2, 1/2, 0)$ é ótima.

- 7.13. *Moedas casadas.* Neste jogo simples de dois jogadores (chame-os de R e C) cada um escolhe um resultado, *cara* ou *coroa*. Se os dois resultados são iguais, C dá um real a R; se os resultados são diferentes, R dá um real a C.

(a) Represente as perdas e ganhos com uma matriz 2×2 .

(b) Qual o valor deste jogo, e quais as estratégias ótimas para os dois jogadores?

- 7.14. O mercado de pizza de uma pequena cidade é dividido entre dois rivais, Tony e Joey. Eles estão ambos investigando estratégias para roubar mercado um do outro. Joey está considerando baixar os preços ou cortar fatias maiores. Tony está visando começar uma linha de pizzas gourmet, ou ofertar assentos do lado de fora, ou dar refrigerante grátis na hora do almoço. Os efeitos dessas várias estratégias estão resumidos na seguinte matriz de perdas e ganhos (células são dúzias de pizzas, ganho para Joey e perda para Tony).

		TONY		
		Gourmet	Assentos	Refrigerante grátis
JOEY	Preço baixo	+2	0	-3
	Fatias maiores	-1	-2	+1

Por exemplo, se Joey reduz os preços e Tony fornece a opção Gourmet, Tony perde 2 dúzias de pizzas de mercado para Joey.

Qual o valor deste jogo, e quais as estratégias ótimas para Tony e Joey?

- 7.15. Encontre o valor do jogo especificado pela seguinte matriz de perdas e ganhos.

$$\begin{matrix} 0 & 0 & -1 & -1 \\ 0 & 1 & -2 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & 0 & 0 & 1 \\ 1 & -2 & 0 & -3 \\ 0 & -3 & 2 & -1 \\ 0 & -2 & 1 & -1 \end{matrix}$$

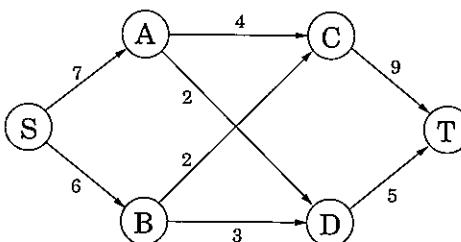
(Dica: Considere as estratégias mistas $(1/3, 0, 0, 1/2, 1/6, 0, 0)$ e $(2/3, 0, 0, 1/3)$.)

- 7.16. Uma salada é qualquer combinação dos seguintes ingredientes: (1) tomate, (2) alface, (3) espinafre, (4) cenoura e (5) óleo. Toda salada tem de conter: (A) pelo menos 15 gramas de proteína, (B) pelo menos 2 e no máximo 6 gramas de gordura, (C) pelo menos 4 gramas de carboidratos, (D) no máximo 100 miligramas de sódio. Além disso, (E) você não quer que sua salada seja mais do que 50% verde, em massa. O conteúdo nutricional desses ingredientes (por 100 gramas) é

ingrediente	energia (kcal)	proteína (gramas)	gordura (gramas)	carboidrato (gramas)	sódio (miligramas)
tomate	21	0,85	0,33	4,64	9,00
alface	16	1,62	0,20	2,37	8,00
espinafre	371	12,78	1,58	74,69	7,00
cenoura	346	8,39	1,39	80,70	508,20
óleo	884	0,00	100,00	0,00	0,00

Encontre um applet para programação linear na Web e use-o para fazer a salada com a quantidade mínima de calorias sob aquelas restrições nutricionais. Descreva sua formulação de programação linear e a solução ótima (a quantidade de cada ingrediente e o valor). Cite os recursos da Web que você usou.

- 7.17. Considere a seguinte rede (os números são capacidades de arestas).



- (a) Encontre o fluxo máximo f e um corte mínimo.
- (b) Desenhe o grafo residual G_f (junto com suas capacidades de arestas). Nesta rede residual, marque os vértices alcançáveis a partir de S e os vértices a partir dos quais T é alcançável.
- (c) Uma aresta da rede é chamada de *aresta de gargalo*, se aumentar sua capacidade resulta em um aumento no fluxo máximo. Liste todas as arestas de gargalo na rede apresentada.
- (d) Dê um exemplo bastante simples (contendo no máximo quatro nós) de uma rede que não tenha nenhuma aresta de gargalo.
- (e) Forneça um algoritmo eficiente para identificar todas as arestas de gargalo em uma rede. (Dica: Comece usando o algoritmo usual de fluxo em redes e, então, examine o grafo residual.)
- 7.18. Existem muitas variações comuns do problema do fluxo máximo. Veja quatro delas.
- (a) Existem muitas fontes e muitos sorvedouros, e queremos maximizar o fluxo total de todas as fontes para todos os sorvedouros.
- (b) Cada vértice também tem uma capacidade de fluxo máximo que pode entrar nele.
- (c) Cada aresta tem não somente uma capacidade, mas também uma *cota inferior* sobre o fluxo que ela tem de carregar.

(d) O fluxo que sai de cada nó u não é o mesmo que o fluxo que entra, mas é menor por um fator de $(1 - \epsilon_u)$, onde ϵ_u é um coeficiente de perda associado ao nó u .

Cada um desses pode ser resolvido eficientemente. Mostre isso reduzindo (a) e (b) para o problema de fluxo máximo original e reduzindo (c) e (d) à programação linear.

- 7.19. Suponha que alguém apresente a você a solução de um problema de fluxo máximo em alguma rede. Forneça um algoritmo de tempo *linear* para determinar se a solução de fato leva a um fluxo máximo.
- 7.20. Considere a seguinte generalização do problema do fluxo máximo.

É dada uma rede direcionada $G = (V, E)$ com capacidades nas arestas $\{c_e\}$. Em vez de um único par (s, t) , são fornecidos múltiplos pares $(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)$ onde os s_i são fontes de G e os t_i são sorvedouros de G . Também são dadas k demandas d_1, \dots, d_k . O objetivo é encontrar k fluxos $f^{(1)}, \dots, f^{(k)}$ com as seguintes propriedades:

- $f^{(i)}$ é um fluxo válido de s_i para t_i .
- Para cada aresta e , o fluxo total $f_e^{(1)} + f_e^{(2)} + \dots + f_e^{(k)}$ não excede a capacidade c_e .
- O tamanho de cada fluxo $f^{(i)}$ é pelo menos a demanda d_i .
- O tamanho do fluxo *total* (a soma dos fluxos) é tão grande quanto possível.

Como você resolveria este problema?

- 7.21. Uma aresta de uma rede de fluxo é chamada *crítica* se ao diminuir a capacidade desta aresta o fluxo máximo é diminuído. Forneça um algoritmo eficiente que encontre uma aresta crítica em uma rede.
- 7.22. Em uma particular rede $G = (V, E)$ cujas arestas têm capacidades inteiras c_e , já encontramos o fluxo máximo f do nó s para o nó t . Entretanto, agora percebemos que um dos valores de capacidade que usamos estava errado: para arestas (u, v) usamos c_{uv} ao passo que deveríamos ter usado $c_{uv} - 1$. Isso é lamentável porque o fluxo f usa aquela particular aresta na capacidade máxima: $f_{uv} = c_{uv}$.

Poderíamos refazer a computação do fluxo do início, mas existe um meio mais rápido. Mostre como um novo fluxo ótimo pode ser computado em tempo $O(|V| + |E|)$.

- 7.23. Uma *cobertura de vértice* de um grafo não-direcionado $G = (V, E)$ é um subconjunto dos vértices que toca todas as arestas — isto é, um subconjunto $S \subset V$ tal que para toda aresta $\{u, v\} \in E$, um ou ambos u, v estão em S .

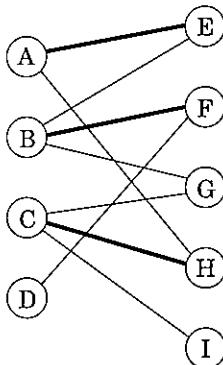
Mostre que o problema de encontrar a cobertura de vértice mínima em um grafo *bipartido* se reduz a fluxo máximo. (*Dica:* Você pode relacionar este problema ao corte mínimo em uma rede apropriada?)

- 7.24. *Emparelhamento bipartido direto.* Vimos como encontrar um emparelhamento máximo em um grafo bipartido via redução ao problema do fluxo máximo. Agora vamos desenvolver um algoritmo direto.

Seja $G = (V_1 \cup V_2, E)$ um grafo bipartido (tal que cada aresta tem uma extremidade em V_1 e a outra em V_2) e seja $M \subseteq E$ um emparelhamento no grafo (um conjunto de arestas que não se tocam). Um vértice é *coberto* por M se ele é a extremidade de algu-

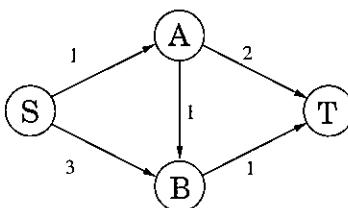
ma aresta em M . Um *caminho alternativo* é um caminho de tamanho ímpar que começa e termina em um vértice não-coberto, e cujas arestas alternam entre M e $E - M$.

- (a) No grafo bipartido a seguir, um emparelhamento M é apresentado em negrito. Encontre um caminho alternante.



- (b) Prove que um emparelhamento M é máximo se e somente se não existir um caminho alternado com relação a ele.
 (c) Projete um algoritmo que encontre um caminho alternado em tempo $O(|V| + |E|)$ usando uma variante de busca em largura.
 (d) Forneça um algoritmo direto $O(|V| \cdot |E|)$ para encontrar um emparelhamento máximo em um grafo bipartido.

7.25. *O dual de fluxo máximo.* Considere a seguinte rede com capacidades nas arestas.



- (a) Escreva o problema de encontrar o fluxo máximo a partir de S até T como um programa linear.
 (b) Escreva o dual deste programa linear. Deve haver uma variável dual para cada aresta da rede e para cada vértice com exceção de S, T .

Agora vamos resolver o mesmo problema com total generalidade. Lembre-se do programa linear para um problema de fluxo máximo geral (Seção 7.2).

- (c) Escreva o dual deste PL geral de fluxo, usando uma variável y_e para cada aresta e x_u para cada vértice $u \neq s, t$.
 (d) Mostre que qualquer solução para o PL geral dual tem de satisfazer a seguinte propriedade: para cada caminho direcionado de s até t na rede, a soma dos valores y_e ao longo deste caminho tem de ser pelo menos 1.

- (e) Quais são os significados intuitivos das variáveis duais? Mostre que qualquer corte $s - t$ na rede pode ser traduzido em uma solução factível dual cujo custo é exatamente a capacidade daquele corte.

7.26. Em um sistema de inequações lineares satisfatória

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &\leq b_1 \\ &\vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &\leq b_m \end{aligned}$$

descrevemos a j -ésima inequação como *forçada-a-igualdade* se ela é satisfeita com igualdade por *toda* solução $\mathbf{x} = (x_1, \dots, x_n)$ do sistema. Equivalentemente, $\sum_i a_{ji}x_i \leq b_j$ não é *forçada-a-igualdade* se existir um \mathbf{x} que satisfaça o sistema inteiro e tal que $\sum_i a_{ji}x_i < b_j$.

Por exemplo, em

$$\begin{array}{rcl} x_1 + x_2 &\leq & 2 \\ -x_1 - x_2 &\leq & -2 \\ x_1 &\leq & 1 \\ -x_2 &\leq & 0 \end{array}$$

as duas primeiras inequações são *forçadas-a-igualdade*, enquanto a terceira e a quarta não são. Uma solução \mathbf{x} para o sistema é chamada *característica* se, para toda inequação I que não é *forçada-a-igualdade*, \mathbf{x} satisfaz I sem igualdade. Na instância anterior, uma tal solução é $(x_1, x_2) = (-1, 3)$, para a qual $x_1 < 1$ e $-x_2 < 0$ enquanto $x_1 + x_2 = 2$ e $-x_1 - x_2 = -2$.

- (a) Mostre que qualquer sistema satisfatório tem uma solução característica.
 (b) Dado um sistema de inequações lineares satisfatório, mostre como usar programação linear para determinar quais inequações são *forçadas-a-igualdade* e para encontrar uma solução característica.
- 7.27. Mostre que o *problema do troco* (Exercício 6.17) pode ser formulado como um programa linear inteiro. Será que podemos resolver este programa como um PL, na certeza de que a solução irá se revelar integral (como no caso de emparelhamento bipartido)? Prove ou dê um contra-exemplo.
- 7.28. *Um programa linear para caminho mínimo.* Suponha que queiramos computar o caminho mínimo a partir de um nó s para o nó t em um grafo direcionado com comprimentos de aresta $l_e > 0$.
- (a) Mostre que isto é equivalente a encontrar um fluxo $s - t$, f , que minimize $\sum_e l_e f_e$ sujeito a $\text{tamanho}(f) = 1$. Não há nenhuma restrição de capacidade.
 (b) Escreva o problema do caminho mínimo como um programa linear.
 (c) Mostre que o PL dual pode ser escrito como
- $$\begin{aligned} \max x_s - x_t \\ x_u - x_v \leq l_{uv} \text{ para toda } (u, v) \in E \end{aligned}$$
- (d) Uma interpretação para o dual é dada no quadro da página 209. Por que o nosso PL dual não é idêntico ao daquela página?

- 7.29. *Hollywood.* Um produtor de filmes está procurando atores e investidores para seu novo filme. Existem n atores disponíveis; o ator i cobra s_i dólares. Para financiamento, existem m investidores disponíveis. O investidor j dará p_j reais, mas somente com a condição de que certos atores $L_j \subseteq \{1, 2, \dots, n\}$ estejam incluídos no elenco (*todos* esses atores L_j têm de ser escolhidos para que seja recebido financiamento do investidor j).

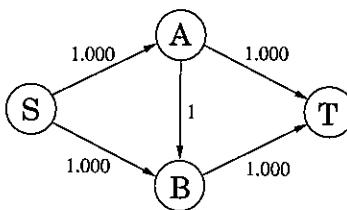
O lucro do produtor é a soma dos pagamentos dos investidores menos o pagamento dos atores. O objetivo é maximizar este lucro.

- Expresse este problema como um programa linear inteiro no qual as variáveis tomam valores $\{0, 1\}$.
- Agora relaxe para um programa linear, e mostre que tem de haver de fato uma solução inteira ótima (como é o caso, por exemplo, com fluxo máximo e emparelhamento bipartido).

- 7.30. *Teorema de Hall.* Retornando ao cenário da formação de casais da Seção 7.3, suponha que tenhamos um grafo bipartido com garotos na esquerda e um número igual de garotas na direita. O teorema de Hall informa que existe um emparelhamento perfeito se e somente se a seguinte condição é satisfeita: qualquer subconjunto S de garotos está conectado a pelo menos $|S|$ garotas.

Prove este teorema. (Dica: O teorema do corte-mínimo fluxo-máximo deve ser útil.)

- 7.31. Considere a seguinte rede simples com capacidades nas arestas como apresentado.



- Mostre que, se o algoritmo de Ford-Fulkerson é executado sobre este grafo, uma escolha sem cuidado de atualizações pode fazê-lo realizar 1.000 iterações. Imagine se as capacidades fossem um milhão em vez de 1.000!

Vamos agora encontrar uma estratégia para escolher caminhos sob a qual o algoritmo garantidamente termina em um número razoável de iterações.

Considere uma rede direcionada arbitrária $(G = (V, E), s, t, \{c_e\})$ na qual queremos encontrar o fluxo máximo. Assuma por simplicidade que todas as capacidades de arestas são pelo menos 1, e defina a capacidade de um caminho $s \rightarrow t$ como a menor capacidade entre as arestas que o constituem. O *caminho mais rápido* de s para t é aquele com a maior capacidade.

- Mostre que o caminho $s \rightarrow t$ mais rápido em um grafo pode ser computado por uma variante do algoritmo de Dijkstra.
- Mostre que o fluxo máximo em G é a soma dos fluxos individuais ao longo de no máximo $|E|$ caminhos de s até t .
- Agora mostre que se sempre aumentamos o fluxo ao longo do caminho mais rápido no grafo residual, então o algoritmo de Ford-Fulkerson terminará em no

máximo $O(|E| \log F)$ iterações, onde F é o tamanho do fluxo máximo. (*Dica:* Rever a prova para o algoritmo guloso para cobertura de conjuntos da Seção 5.4.)

De fato, uma regra ainda mais simples — encontrar um caminho no grafo residual usando busca em largura — garante que no máximo $O(|V| \cdot |E|)$ iterações serão necessárias.

Capítulo 8

Problemas NP-completos

8.1 Problemas de busca

Durante os sete capítulos anteriores desenvolvemos algoritmos para encontrar caminhos mínimos e árvores espalhadas mínimas em grafos, emparelhamentos em grafos bipartidos, subseqüência crescente máxima, fluxo máximo em redes e assim por diante. Todos esses algoritmos são *eficientes*, porque em cada caso o requisito de tempo cresce como uma função polinomial (tal como n , n^2 ou n^3) no tamanho da entrada.

Para melhor apreciar tais algoritmos eficientes, considere uma alternativa: em todos esses problemas buscamos uma solução (caminho, árvore, emparelhamento etc.) dentro uma população *exponencial* de possibilidades. De fato, n garotos podem ser casados com n garotas de $n!$ maneiras diferentes, um grafo com n vértices tem n^{n-2} árvores espalhadas, e um grafo típico tem um número exponencial de caminhos de s para t . Todos esses problemas poderiam ser, em princípio, resolvidos em tempo exponencial checando entre todas as soluções candidatas, uma por uma. Mas um algoritmo cujo tempo de execução é 2^n , ou pior, é qualquer coisa menos útil na prática (veja o próximo quadro). A busca por algoritmos eficientes tem a ver com encontrar maneiras inteligentes de contornar o processo de busca exaustiva, usando dicas vindas da entrada para radicalmente reduzir o espaço de busca.

Até agora, neste livro, temos visto os mais brilhantes sucessos desta busca, técnicas algorítmicas que derrotam o espectro da exponencialidade: algoritmos gulosos, programação dinâmica, programação linear (enquanto divisão-e-conquista tipicamente gera algoritmos rápidos para problemas que já conseguíamos resolver em tempo polinomial). Chegou a hora de encontrar as falhas mais persistentes e constrangedoras desta busca. Veremos alguns outros “problemas de busca”, nos quais novamente estamos procurando uma solução com propriedades particulares entre um caos exponencial de alternativas. Mas para esses novos problemas nenhum atalho parece ser possível. Os algoritmos mais rápidos que conhecemos para eles são todos exponenciais — não substancialmente melhores dos que uma busca exaustiva. A partir de agora introduziremos alguns exemplos importantes.

Satisfatibilidade

SATISFATIBILIDADE, ou SAT (reveja o Exercício 3.28 e a Seção 5.3), é um problema de grande importância prática, com aplicações variando de teste de chip e de projeto de

A história de Sissa e Moore

De acordo com a lenda, o jogo de xadrez foi inventado pelo brâmane Sissa para divertir e educar o seu rei. Perguntado pelo grande monarca o que ele queria em troca, o homem sábio pediu que o rei colocasse um grão de arroz no primeiro quadrado do tabuleiro, dois no segundo, quatro no terceiro e assim por diante, dobrando a quantidade de arroz até a 64^a quadrado. O rei concordou no ato, e como resultado ele foi a primeira pessoa a aprender a valiosa — embora limitante — lição de crescimento exponencial. O pedido de Sissa correspondia a $2^{64} - 1 = 18.446.744.073.709.551.615$ grãos de arroz, arroz suficiente para pavimentar a Índia várias vezes!

Por toda a natureza, de colônias de bactérias a células em um feto, vemos sistemas que crescem exponencialmente — por um tempo. Em 1798, o filósofo britânico T. Robert Malthus publicou um ensaio no qual previu que o crescimento exponencial (chamou-o de “crescimento geométrico”) da população humana iria em breve esgotar os recursos, que são linearmente crescentes, um argumento que influenciou Charles Darwin profundamente. Malthus sabia o fato fundamental de que uma exponencial cedo ou tarde supera qualquer polinomial.

Em 1965, o pioneiro de chips de computadores Gordon E. Moore notou que a densidade de transistores em chips havia dobrado a cada ano no começo dos anos de 1960, e previu que a tendência continuaria. Essa previsão, moderada para uma duplicação a cada 18 meses e estendida para a velocidade dos computadores, é conhecida como a *lei de Moore*. Ela tem valido memoravelmente por 40 anos. Estas são as duas causas na base da explosão de tecnologia da informação nas décadas passadas: a *lei de Moore* e *algoritmos eficientes*.

Pareceria que a lei de Moore desencoraja o desenvolvimento de algoritmos polinomiais. Afinal, se um algoritmo é exponencial, por que não esperar que a lei de Moore o torne factível? Mas na realidade exatamente o oposto ocorre: a lei de Moore é um tremendo incentivo para o desenvolvimento de algoritmos eficientes, porque tais algoritmos são necessários para obtermos benefícios do crescimento exponencial na velocidade dos computadores.

Veja o porquê. Se, por exemplo, um algoritmo $O(2^n)$ para satisfatibilidade booleana (SAT) roda por uma hora, ele resolveria instâncias com 25 variáveis em 1975, 31 variáveis nos computadores mais rápidos de 1985, 38 variáveis em 1995, e cerca de 45 variáveis com as máquinas de hoje. Bastante progresso — exceto que cada variável extra requer um ano e meio de espera, enquanto o apetite de aplicações (muitas das quais estão, ironicamente, relacionadas com projeto de computadores) cresce muito mais rápido. Em contraste, o tamanho de instâncias resolvidas por um algoritmo $O(n)$ ou $O(n \log n)$ seria *multiplicado por um fator de cerca de 100* a cada década. No caso de um algoritmo $O(n^2)$, o tamanho da instância solúvel em um tempo fixo seria multiplicado por cerca de 10 a cada década. Mesmo um algoritmo $O(n^6)$, polinomial apesar de não convidativo, mais que dobraria o tamanho das instâncias resolvidas a cada década. Quando se trata do crescimento do tamanho dos problemas que podemos resolver com um algoritmo, temos uma inversão: algoritmos exponenciais fazem um progresso lento, polinomial, enquanto algoritmos polinomiais avançam rápido, exponencialmente! Para que a lei de Moore se reflita no mundo, precisamos de algoritmos eficientes.

A história de Sissa e Moore (continuação)

Como Sissa e Malthus sabiam muito bem, expansão exponencial não pode ser sustentada indefinidamente no nosso mundo finito. Colônias de bactérias ficam sem comida; os chips encontram a barreira da escala atômica. A lei de Moore vai parar de dobrar a velocidade dos nossos computadores dentro de uma ou duas décadas. E, então, o progresso dependerá de genialidade algorítmica — ou talvez ao contrário de novas idéias como *computação quântica*, explorada no Capítulo 10.

computadores a análise de imagens e engenharia de software. Ele é também um problema difícil canônico. Veja a aparência de uma instância de SAT:

$$(x \vee y \vee z) \ (x \vee \bar{y}) \ (y \vee \bar{z}) \ (z \vee \bar{x}) \ (\bar{x} \vee \bar{y} \vee \bar{z}).$$

Isso é uma *fórmula booleana na forma normal conjuntiva* (CNF, do inglês *conjunctive normal form*). Ela é uma coleção de *cláusulas* (os parênteses), cada uma consistindo em uma disjunção (*ou* lógico, denotado \vee) de vários *literais*, onde um literal é uma variável booleana (tal como x) ou a negação de uma (tal como \bar{x}). Uma *atribuição de valor verdade satisfatória* é uma atribuição de falso ou verdadeiro a cada variável de modo que cada cláusula contenha um literal cujo valor seja verdadeiro. O problema SAT é o seguinte: dada uma fórmula booleana na forma normal conjuntiva, encontre uma atribuição satisfatória ou então declare que nenhuma existe.

Na instância mostrada anteriormente, tornar todas as variáveis verdadeiro satisfaz todas as cláusulas exceto a última. Será que existe uma atribuição que satisfaça *todas* as cláusulas?

Com um pouco de reflexão, não é difícil argumentar que nesse caso nenhuma atribuição desse tipo existe. (*Dica:* As três cláusulas do meio restringem todas as três variáveis a ter o mesmo valor.) Mas como decidimos isso em geral? Claro, podemos sempre buscar entre todas as possíveis atribuições, uma por uma, mas para fórmulas com n variáveis, o número de atribuições possíveis é exponencial, 2^n .

SAT é um típico *problema de busca*. É fornecida uma *instância I* (ou seja, algum dado de entrada especificando o problema em mãos, neste caso uma fórmula booleana na forma normal conjuntiva), e nos pedem para encontrarmos uma *solução S* (um objeto que satisfaça uma especificação, neste caso uma atribuição que satisfaça cada cláusula). Se nenhuma solução desse tipo existe, então devemos afirmar tal fato.

Mais especificamente, um problema de busca tem de ter a propriedade de que qualquer solução S proposta para uma instância I pode ser *rapidamente checada* pela sua correção. O que isso implica? Claramente, S tem de pelo menos ser concisa (rápida de ler), com tamanho polinomialmente limitado por aquele de I . Isso é claramente verdade no caso de SAT, para o qual S é uma atribuição para as variáveis. Para formalizarmos a noção de checagem rápida, diremos que existe um algoritmo de tempo polinomial que toma como entrada I e S e decide se S é ou não uma solução para I . Para SAT, isso é fácil, pois apenas envolve checar que a atribuição especificada por S satisfaz mesmo cada cláusula de I .

Mais adiante neste capítulo será útil elevar tal ponto de vista e pensar neste algoritmo eficiente para checar soluções propostas como *definindo* o problema de busca. Assim:

Um problema de busca é especificado por um algoritmo \mathcal{C} que toma duas entradas, uma instância I e uma solução S proposta, e roda em tempo polinomial em $|I|$. Dizemos que S é uma solução para I se e somente se $\mathcal{C}(I, S) = \text{verdadeiro}$.

Dada a importância do problema de busca SAT, os pesquisadores durante os últimos 50 anos tentaram duramente encontrar maneiras eficientes de resolvê-lo, mas sem sucesso. Os algoritmos mais rápidos que temos são ainda exponenciais nas suas entradas de pior caso.

Ainda assim, interessantemente, existem duas variantes naturais de SAT para as quais temos de fato bons algoritmos. Se todas as cláusulas contêm no máximo um literal positivo, então a fórmula booleana é chamada de *fórmula Horn*, e uma atribuição satisfatória, se existe, pode ser encontrada pelo algoritmo guloso da Seção 5.3. Como alternativa, se todas as cláusulas têm apenas *dois* literais, a teoria de grafos entra em cena, e SAT pode ser resolvido em tempo linear encontrando as componentes fortemente conexas de um particular grafo construído da instância (reveja o Exercício 3.28). De fato, no Capítulo 9 veremos um algoritmo polinomial diferente para este mesmo caso especial, chamado 2SAT.

Por sua vez, se formos um pouco mais permissivos e aceitarmos cláusulas com *três* literais, então o problema resultante, conhecido como 3SAT (um exemplo do qual vimos antes), de novo se torna difícil de resolver!

O problema do caixeiro-viajante

No problema do caixeiro-viajante (TSP, do inglês *traveling salesman problem*) são dados n vértices $1, \dots, n$ e todas as $n(n - 1)/2$ distâncias entre eles, bem como um orçamento b . Temos de encontrar um *círculo*, um ciclo que passe por todos os vértices exatamente uma vez, de custo total b ou menos — ou declarar que nenhum circuito desse tipo existe, ou seja, buscamos uma permutação $\tau(1), \dots, \tau(n)$ dos vértices tal que quando eles são percorridos nesta ordem, a distância total coberta é no máximo b :

$$d_{\tau(1),\tau(2)} + d_{\tau(2),\tau(3)} + \cdots + d_{\tau(n),\tau(1)} \leq b.$$

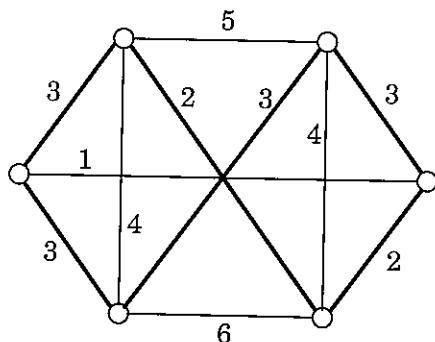
Veja a Figura 8.1 para um exemplo (apenas algumas das distâncias são mostradas; considere que as restantes sejam muito grandes).

Note como definimos o TSP como um *problema de busca*: dada uma instância, encontre um circuito dentro do orçamento (ou declare que nenhum existe). Mas por que estamos expressando o problema do caixeiro viajante dessa maneira, quando na realidade ele é um *problema de otimização*, no qual o *menor* circuito possível é procurado? Por que vesti-lo com outra roupa?

Por uma boa razão. Nossa plano neste capítulo é comparar e relacionar problemas. O enquadramento em problemas de busca é útil neste sentido, porque engloba problemas de otimização como o TSP, além de verdadeiros problemas de busca como SAT.

Transformar um problema de otimização em um de busca não muda sua dificuldade em absoluto, porque as duas versões se *reduzem uma a outra*. Qualquer algoritmo que resolva o TSP de otimização, prontamente resolve o problema de busca: encontre o circuito ótimo e se ele estiver dentro do orçamento, apresente-o; senão, não há solução.

Figura 8.1 O circuito ótimo para o caixeiro-viajante, mostrado em negrito, tem comprimento 18.



Na direção contrária, um algoritmo para o problema de busca também pode ser usado para resolver o problema de otimização. Para entender por que, primeiro suponha que, de alguma maneira, conhecemos o custo do circuito ótimo; então, poderíamos encontrar esse circuito chamando o algoritmo para o problema de busca, usando o custo ótimo como orçamento. Muito bom, mas como encontramos o custo ótimo? Fácil: por busca binária! (Veja o Exercício 8.1.)

A propósito, existe uma sutileza aqui: por que introduzimos um orçamento? Todo problema de otimização também não é um problema de busca no sentido de que estamos buscando uma solução que tem a propriedade de ser ótima? O problema é que a solução para um problema de busca deve ser fácil de reconhecer ou, como colocamos antes, verificável em tempo polinomial. Dada uma solução para o TSP, é fácil checar as propriedades “é um circuito” (simplesmente verifique que cada vértice é visitado exatamente uma vez) e “tem comprimento total $\leq b$ ”. Mas como alguém poderia verificar a propriedade “é ótimo”?

Como com SAT, não há nenhum algoritmo de tempo polinomial conhecido para o TSP, apesar de muito esforço de pesquisadores durante quase um século. Claro, existe um algoritmo exponencial para resolvê-lo, tentando todos os $(n - 1)!$ circuitos, e na Seção 6.6 vimos um algoritmo mais rápido, embora ainda exponencial, de programação dinâmica.

O problema da árvore espalhada mínima (AEM), para o qual de fato temos um algoritmo eficiente, possibilita um contraste nítido aqui. Para refraseá-lo como um problema de busca, novamente é dada uma matriz de distâncias e uma cota b , e temos de encontrar a árvore T com peso total $\sum_{(i, j) \in T} d_{ij} \leq b$. O TSP pode ser imaginado como um primo difícil do problema AEM, no qual não é permitido à árvore ramificar e, portanto, ela tem de ser um caminho.¹ Essa restrição extra na estrutura da árvore resulta em um problema muito mais difícil.

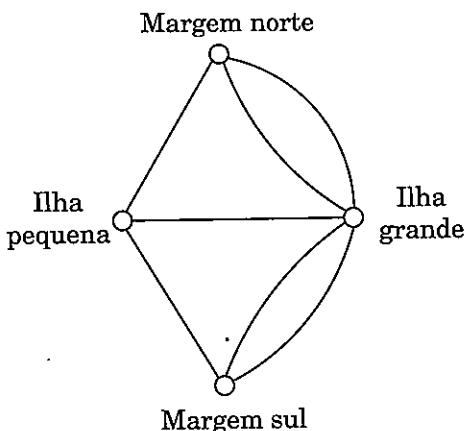
Euler e Rudrata

No verão de 1735, Leonhard Euler (pronuncia-se “Oiler”), o famoso matemático suíço, estava andando pelas pontes do parque da cidade de Königsberg, na Prússia Oriental.

¹Na verdade, o TSP demanda um circuito, mas podemos definir uma versão alternativa que busca um caminho, e não é difícil perceber que isso é tão difícil quanto o próprio TSP.

Depois de um tempo, ele notou frustrado que, não importava onde começava sua caminhada, não importava quão inteligentemente ele prosseguia, era impossível cruzar cada ponte exatamente uma vez. E dessa ambição boba, a área de teoria dos grafos nasceu.

Euler identificou prontamente as raízes da deficiência do parque. Primeiro, você transforma um mapa do parque em um grafo cujos vértices são as quatro áreas de terra (duas ilhas, duas margens) e cujas arestas são as setes pontes:



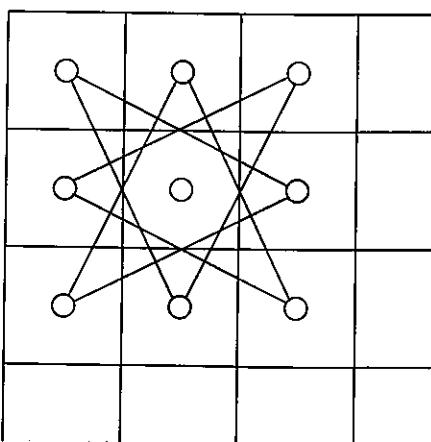
Esse grafo possui múltiplas arestas entre dois vértices — uma característica que não temos permitido até agora neste livro, mas que é significativa para este problema em particular, pois cada ponte tem de ser contada separadamente. Estamos procurando por um caminho que passe cada aresta exatamente uma vez (o caminho pode repetir vértices). Em outras palavras, estamos fazendo esta pergunta: *quando um grafo pode ser desenhado sem levantar o lápis do papel?*

A resposta descoberta por Euler é simples, elegante e intuitiva: se e somente se (a) o grafo é conexo e (b) cada vértice, com a possível exceção de dois vértices (os vértices inicial e final da caminhada), tem grau par (Exercício 3.26). Essa é a razão pela qual o parque de Königsberg não podia ser percorrido: todos os quatro vértices têm grau ímpar.

Para colocarmos isso nos nossos interesses atuais, vamos definir um problema de busca chamado CAMINHO EULERIANO: dado um grafo, encontre um caminho que contenha cada aresta exatamente uma vez. Segue da observação de Euler, e de um pouco mais de raciocínio, que este problema de busca pode ser resolvido em tempo polinomial.

Quase um milênio antes do fatídico verão de Euler na Prússia Oriental, um poeta da Kashemira chamado Rudrata fez esta pergunta: será possível visitar todos os quadrados do tabuleiro de xadrez, sem repetir nenhum quadrado, em uma longa caminhada que termine no quadrado em que se começou e que em cada passo faça um movimento legal de cavalo? Isso é novamente um problema em grafos: este grafo tem agora 64 vértices, e dois quadrados estão conectados por uma aresta se um cavalo puder ir de um para o outro quadrado em um único movimento (ou seja, se suas coordenadas diferirem

Figura 8.2 Os movimentos do cavalo em um canto do tabuleiro de xadrez.



por 2 em uma dimensão e por 1 na outra). Veja a Figura 8.2 para a porção do grafo correspondente ao canto superior esquerdo do tabuleiro. Você pode encontrar um circuito para o cavalo no seu tabuleiro?

Esse é um tipo diferente de problema de busca em grafos: queremos um ciclo que passe por todos os *vértices* (em oposição a todas as arestas no problema de Euler), sem repetir nenhum vértice. E não há qualquer razão para ficar somente com tabuleiros; essa questão pode ser feita para qualquer grafo. Vamos definir o problema de busca CICLO RUDRATA como o seguinte: dado um grafo, encontre um ciclo que visita cada vértice exatamente uma vez — ou afirme que não existe ciclo deste tipo.² Esse problema infelizmente faz lembrar o TSP e, de fato, nenhum algoritmo polinomial é conhecido para ele.

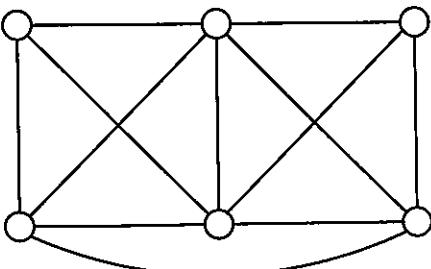
Há duas diferenças entre as definições dos problemas de Euler e Rudrata. A primeira é que o problema de Euler visita todas as *arestas* enquanto o de Rudrata visita todos os *vértices*. Mas há também a questão de que um deles demanda um caminho e o outro requer um ciclo. Qual dessas duas diferenças justifica a disparidade enorme na complexidade computacional entre os dois problemas? Tem de ser a primeira, porque a segunda diferença pode ser vista como puramente cosmética. De fato, defina o problema do CAMINHO RUDRATA exatamente como o CICLO RUDRATA, exceto que o objetivo agora é encontrar um *caminho* que passe por todos os vértices exatamente uma vez. Como veremos em breve, existe uma equivalência precisa entre as duas versões do problema de Rudrata.

Cortes e bissecções

Um *corte* é um conjunto de arestas cuja remoção deixa o grafo desconexo. Freqüentemente é interessante encontrar cortes pequenos e o problema do CORTE MÍNIMO é, dado

²Na literatura este problema é conhecido como o problema do *ciclo hamiltoniano*, em homenagem ao grande matemático irlandês que o descobriu no século XIX.

Figura 8.3 Qual o menor corte neste grafo?



um grafo e um orçamento b , encontrar um corte com no máximo b arestas. Por exemplo, o menor corte na Figura 8.3 tem tamanho 3. Esse problema pode ser resolvido em tempo polinomial com $n - 1$ computações de fluxo máximo: dê a cada aresta a capacidade de 1 e encontre o fluxo máximo entre algum nó fixo e todos os outros nós. O menor fluxo desse tipo corresponderá (via teorema do fluxo-máximo corte-mínimo) ao menor corte. Você entende por quê? Nós também vimos um algoritmo randomizado, bastante diferente, para este problema (página 140).

Em muitos grafos, como naquele da Figura 8.3, o menor corte deixa apenas um vértice de um lado — ele consiste em todas as arestas adjacentes a este vértice. Muito mais interessantes são cortes pequenos que particionam os vértices do grafo em conjuntos de tamanhos quase iguais. Mais precisamente, este é o problema do CORTE BALANCEADO: dado um grafo com n vértices e um orçamento b , particione os vértices em dois conjuntos S e T tal que $|S|, |T| \geq n/3$ e tal que haja no máximo b arestas entre S e T . Outro problema difícil.

Cortes balanceados aparecem em uma variedade de aplicações importantes, tais como *agrupamento* (*clustering*). Considere, por exemplo, o problema de segmentar uma imagem nos seus componentes constituintes (digamos, um elefante em pé em um gramado plano com um céu azul acima). Um boa maneira de fazer isso é criar um grafo com um nó por pixel da imagem e colocar uma aresta entre nós cujos pixels correspondentes estão espacialmente perto um do outro e também têm cores similares. Um objeto único na imagem (como o elefante, digamos) então corresponde a um conjunto de vértices altamente conectados no grafo. Um corte balanceado, portanto, vai provavelmente dividir os pixels em dois grupos sem separar nenhum dos constituintes primários da imagem. O primeiro corte pode, por exemplo, separar o elefante de um lado do céu e da grama no outro lado. Um corte posterior seria necessário para separar o céu da grama.

Programação linear inteira

Muito embora o algoritmo simplex não seja de tempo polinomial, mencionamos no Capítulo 7 que *existe* um algoritmo diferente, polinomial, para programação linear. Portanto, programação linear é solúvel eficientemente tanto na prática quanto na teoria. Mas a situação muda completamente se, além de especificarmos uma função linear

objetiva e inequações lineares, também restringimos a solução (os valores das variáveis) a números *inteiros*. Esse problema é chamado PROGRAMAÇÃO LINEAR INTEIRA (PLI). Vamos ver como podemos formulá-lo como um problema de busca. É dado um conjunto de inequações lineares $Ax \leq b$, onde A é uma matriz $m \times n$ e b é um vetor de dimensão m ; uma função objetivo especificada por um vetor c de dimensão n ; e por fim um *objetivo* g (o análogo do orçamento em problemas de maximização). Queremos encontrar um vetor x *inteiro* não-negativo de dimensão n , tal que $Ax \leq b$ e $c \cdot x \geq g$.

Mas há uma redundância aqui: a última restrição $c \cdot x \geq g$ é ela própria uma inequação linear e pode ser absorvida em $Ax \leq b$. Portanto, definimos PLI como o seguinte problema de busca: dados A e b , encontre um vetor x inteiro não-negativo que satisfaça as inequações $Ax \leq b$, ou afirme que nenhum vetor deste tipo existe. A despeito das muitas aplicações cruciais desse problema, e do interesse intenso dos pesquisadores, nenhum algoritmo eficiente é conhecido para ele.

Existe um caso especial particularmente claro de PLI que é muito difícil em si mesmo: o objetivo é encontrar um vetor x de 0 e 1 satisfazendo $Ax = 1$, onde A é uma matriz $m \times n$ com células 0 – 1 e 1 é o vetor de 1 de dimensão m . Deve ser evidente das reduções na Seção 7.1.4 que isso é de fato um caso especial de PLI. Chamamos de EQUAÇÕES ZERO-UM (EZU).

Introduzimos um número de importantes problemas de busca, alguns dos quais são conhecidos de capítulos anteriores e para os quais há algoritmos eficientes, e outros que são diferentes em pequenas, mas cruciais, formas que os fazem problemas computacionais muito difíceis. Para completar nossa história vamos introduzir mais alguns poucos problemas difíceis, que terão um papel mais adiante neste capítulo, quando vamos relacionar a dificuldade computacional de todos estes problemas. O leitor está convidado a ir direto para a Seção 8.2 e, então, retornar para as definições desses problemas quando for necessário.

Casamento tridimensional

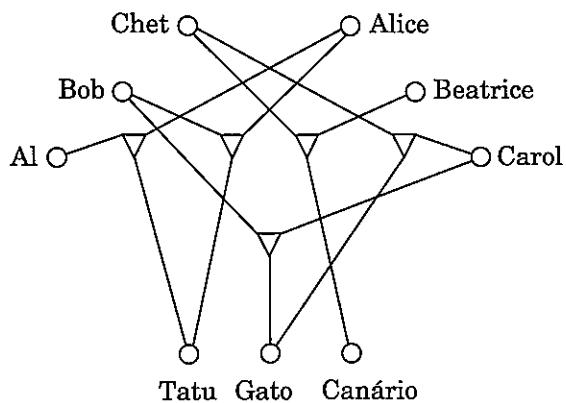
Lembre-se do problema do EMPARELHAMENTO BIPARTIDO: dado um grafo bipartido com n nós de cada lado (os *garotos* e as *garotas*), encontre um conjunto de n arestas disjuntas, ou afirme que nenhum conjunto deste tipo existe. Na Seção 7.3, vimos como resolver esse problema eficientemente com uma redução para fluxo máximo. Entretanto, existe uma generalização interessante, chamada CASAMENTO 3D (3D MATCHING), para o qual nenhum algoritmo polinomial é conhecido. Neste novo contexto, existem n garotos e n garotas, mas também n bichos de estimação, e as compatibilidades entre eles são especificadas por um conjunto de *triplas*, cada uma contendo um garoto, uma garota e um bicho. Intuitivamente, uma tripla (b, g, p) significa que o garoto b , a garota g e o bicho p se dão bem juntos. Queremos encontrar n tripas disjuntas e, assim, criar n domicílios harmoniosos.

Você pode identificar uma solução na Figura 8.4?

Conjunto independente, cobertura de vértice e clique

No problema do CONJUNTO INDEPENDENTE (veja a Seção 6.7) é dado um grafo e um inteiro g , e o objetivo é encontrar g vértices que são independentes, isto é, não existe arestas entre nenhum par deles. Você pode encontrar um conjunto independente de três vértices na Figura 8.5? O que dizer de quatro vértices? Vimos na Seção 6.7 que este

Figura 8.4 Um cenário de casamento mais elaborado. Cada tripla é mostrada como um nó em forma de triângulo ligando um garoto, uma garota e um bicho de estimação.

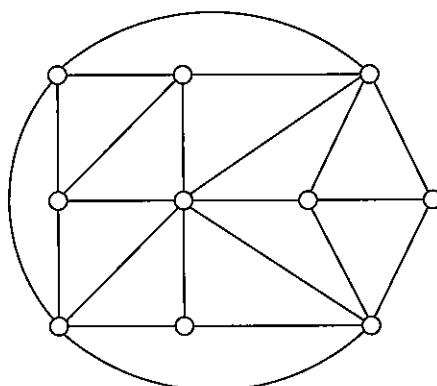


problema pode ser resolvidó eficientemente em árvores, mas para grafos gerais nenhum algoritmo polinomial é conhecido.

Existem muitos outros problemas de busca sobre grafos. Na COBERTURA DE VÉRTICE, por exemplo, a entrada é um grafo e um orçamento b , e a idéia é encontrar b vértices que cubram (toquem) cada aresta. Você pode cobrir todas as arestas da Figura 8.5 com sete vértices? Com seis? (E você vê a conexão íntima com o problema do CONJUNTO INDEPENDENTE?)

COBERTURA DE VÉRTICE é um caso especial de COBERTURA DE CONJUNTO, que encontramos no Capítulo 5. Naquele problema, é dado um conjunto E e vários subconjuntos

Figura 8.5 Qual o tamanho do maior conjunto independente neste grafo?



dele S_1, \dots, S_m , juntamente com um orçamento b . Precisamos selecionar b destes subconjuntos tal que a união deles seja E . COBERTURA DE VÉRTICE é o caso especial no qual E consiste nas arestas de um grafo e há um subconjunto S_i para cada vértice, contendo as arestas adjacentes àquele vértice. Você entende por que CASAMENTO 3D também é um caso especial de COBERTURA DE CONJUNTO?

E por fim há o problema da CLIQUE: dado um grafo e um objetivo g , encontre um conjunto de g vértices tal que todas as possíveis arestas entre eles estejam presentes. Qual é a maior clique na Figura 8.5?

Caminho mais longo

Sabemos que o problema do caminho mínimo pode ser resolvido muito eficientemente, mas o que dizer do problema do CAMINHO MAIS LONGO (máximo)? Aqui é dado um grafo G com pesos de aresta não-negativos e dois vértices s e t determinados, junto com um objetivo g . Precisamos encontrar um caminho de s para t com o peso total de pelo menos g . Naturalmente, para evitar soluções triviais exigimos que o caminho seja *simples*, não contendo vértices repetidos.

Nenhum algoritmo eficiente é conhecido para esse problema (que às vezes também é conhecido pelo nome de TAXICAB RIP-OFF, ou taxista desonesto).

Mochila e soma de subconjunto

Lembre-se do problema da MOCHILA (Seção 6.4): são dados pesos inteiros w_1, \dots, w_n e valores inteiros v_1, \dots, v_n para n itens. Também é dado uma capacidade de peso W e um objetivo g (a capacidade está presente no problema de otimização original, o objetivo é adicionado para se obter um problema de busca). Buscamos um conjunto de itens cujo peso total é no máximo W e cujo valor total é pelo menos g . Como sempre, se nenhum conjunto deste tipo existir, devemos declará-lo.

Na Seção 6.4, desenvolvemos um esquema de programação dinâmica para MOCHILA com tempo de execução $O(nW)$, que notamos ser exponencial no tamanho da entrada, pois envolve W em vez de $\log W$. E também temos o algoritmo exaustivo usual, que inspeciona todos os subconjunto de itens — todos os 2^n deles. Será que existe um algoritmo polinomial para MOCHILA? Ninguém conhece um.

Mas suponhamos que estejamos interessados na variante do problema da mochila no qual os inteiros são codificados em *unário* — por exemplo, escrevendo *IIIIIIIIIIII* para 12. Isso é reconhecidamente uma maneira exponencial de representar inteiros, um desperdício, mas ela de fato define um problema legítimo, que poderíamos chamar de MOCHILA UNÁRIA. Segue da nossa discussão que este problema um tanto artificial de fato possui um algoritmo polinomial.

Uma variante diferente: suponha agora que cada valor de item seja igual a seu peso (todos dados em binário), e para completar, o objetivo g é o mesmo que a capacidade W . (Para adaptar a historinha com a qual introduzimos o problema da mochila, os itens são todos pepitas de ouro, e o ladrão quer encher sua mochila até o topo.) Esse caso especial é equivalente a encontrar um subconjunto de um certo conjunto de inteiros que some exatamente W . Como ele é um caso especial da MOCHILA, não pode ser mais difícil. Mas será que poderia ser polinomial? Como se verifica, esse problema, chamado SOMA DE SUBCONJUNTO, também é muito difícil.

Nesse momento alguém poderia perguntar: se SOMA DE SUBCONJUNTO é um caso especial, tão difícil quanto o problema geral MOCHILA, por que estamos interessados nele? A razão é *simplicidade*. No cálculo complicado de reduções entre problemas de busca que desenvolveremos neste capítulo, problemas conceitualmente simples como SOMA DE SUBCONJUNTO e 3SAT são valiosos.

8.2 Problemas NP-completos

Problemas difíceis, problemas fáceis:

Em resumo, o mundo está cheio de problemas de busca, alguns podem ser resolvidos eficientemente, enquanto outros parecem ser muito difíceis. Isso é apresentado na seguinte tabela.

Problemas difíceis (NP-completos)	Problemas fáceis (em P)
3SAT	2SAT, HORN SAT
CAIXEIRO-VIAJANTE	ÁRVORE ESPALHADA MÍNIMA
CAMINHO MAIS LONGO	CAMINHO MÍNIMO
CASAMENTO 3D	EMPARELHAMENTO BIPARTIDO
MOCHILA	MOCHILA UNÁRIA
CONJUNTO INDEPENDENTE	CONJUNTO INDEPENDENTE em árvores
PROGRAMAÇÃO LINEAR INTEIRA	PROGRAMAÇÃO LINEAR
CAMINHO RUDRATA	CAMINHO EULERIANO
CORTE BALANCEADO	CORTE MÍNIMO

Vale a pena contemplar a tabela. Na direita temos problemas que podem ser resolvidos com eficiência. Na esquerda, temos um cacho de nozes duras que escaparam de uma solução eficiente por muitas décadas ou séculos.

Os vários problemas na direita podem ser resolvidos por diversos e especializados algoritmos: programação dinâmica, fluxo em redes, busca em grafos, guloso. Esses problemas são fáceis por várias razões diferentes.

Em um contraste forte, os problemas da esquerda são *todos difíceis pela mesma razão!* No seu núcleo, são todos o mesmo problema, apenas com roupagens diferentes! Eles são todos *equivalentes*: como veremos na Seção 8.3, cada um pode ser reduzido para qualquer um dos outros — e vice-versa.

P e NP

Chegou a hora de introduzirmos alguns conceitos importantes. Sabemos o que um problema de busca é: a característica que o define é que qualquer solução proposta pode ter sua correção rapidamente verificada, ou seja, existe um algoritmo de verificação eficiente C que toma como entrada a dada instância I (os dados especificando o problema a ser resolvido), bem como a solução proposta S , e responde verdadeiro se e

Por que P e NP?

OK, P vem de “polinomial”. Mas por que usamos as iniciais NP (a abreviação comum para “no problem”) para descrever uma classe de problemas de busca, alguns dos quais são terrivelmente difíceis?

NP vem de “tempo polinomial não determinístico”, um termo que vem das raízes da teoria de complexidade. Intuitivamente, isso significa que uma solução para qualquer problema de busca pode ser encontrada e verificada em tempo polinomial por um tipo especial (e bastante irreal) de algoritmo, chamado *algoritmo não-determinístico*. Tal algoritmo tem o poder de *adivinhar* corretamente a cada passo.

A propósito, a definição original de NP (e seu uso mais comum até os dias atuais) não era de uma classe de problemas de busca, mas de uma classe de *problemas de decisão*: questões algébricas que podem ser respondidas com sim ou não. Exemplo: “será que existe uma atribuição que satisfaz a fórmula booleana?”. Mas isso também reflete uma realidade histórica: na época em que a teoria de NP-completude estava sendo desenvolvida, os pesquisadores da teoria da computação estavam interessados em linguagens formais, um domínio no qual problemas de decisão são de importância central.

somente se S realmente é uma solução para a instância I . Além disso, o tempo de execução de $C(I, S)$ é limitado por um polinômio em $|I|$, o comprimento da instância. Denotamos a classe de todos os problemas de busca por NP.

Vimos muitos exemplos de problemas de busca em NP solúveis em tempo polinomial. Nesses casos, há um algoritmo que toma como entrada uma instância I e tem tempo de execução polinomial em $|I|$. Se I tem uma solução, o algoritmo apresenta uma tal solução; e se I não tem solução, o algoritmo corretamente aponta isso. A classe de todos os problemas de busca que podem ser resolvidos em tempo polinomial é denotada P. Assim, todos os problemas de busca na parte direita da tabela estão em P.

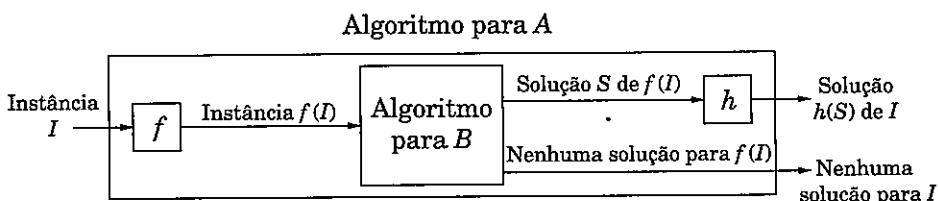
Será que existem problemas de busca que não podem ser resolvidos em tempo polinomial? Em outras palavras, será que $P \neq NP$? A maioria dos pesquisadores em algoritmos pensa que sim. É difícil acreditar que busca exponencial pode sempre ser evitada, que um simples truque vai solucionar todos os problemas difíceis, notoriamente sem solução por décadas e séculos. E existe uma boa razão para os matemáticos acreditarem que $P \neq NP$ — a tarefa de encontrar uma prova para determinada asserção matemática é um problema de busca e está, portanto, em NP (afinal, quando uma prova formal de uma sentença matemática é escrita em todo detalhe, ela pode ser verificada mecanicamente, linha por linha, por um algoritmo eficiente). Portanto, se $P = NP$, haveria uma maneira eficiente de provar teoremas, eliminando assim a necessidade de haver matemáticos! Juntando tudo, há uma variedade de razões pelas quais se acredita amplamente que $P \neq NP$. Entretanto, demonstrar revelou-se extremamente difícil, um dos mais profundos e mais importantes quebra-cabeças não resolvidos da matemática.

Reduções, novamente

Mesmo que aceitemos que $P \neq NP$, o que dizer dos problemas específicos no lado esquerdo da tabela? Com base em que evidência acreditamos que esses particulares pro-

blemas não têm nenhum algoritmo eficiente (além, claro, do fato histórico que muitos matemáticos e cientistas da computação inteligentes tentaram duramente e falharam na busca por algum)? Tal evidência é dada por *reduções*, que traduzem um problema de busca em outro. O que elas demonstram é que os problemas do lado esquerdo da tabela são todos, em certo sentido, *o mesmo problema*, exceto que são formulados em diferentes linguagens. Mais que isso, nós também usaremos reduções para mostrar que esses problemas são os de busca *mais difíceis* em NP — se um deles tiver um algoritmo de tempo polinomial, *todos* os problemas em NP terão um algoritmo polinomial. Assim, se acreditamos que $P \neq NP$, todos esses problemas de busca são difíceis.

Definimos reduções no Capítulo 7 e vimos muitos exemplos delas. Vamos agora especializar esta definição para problemas de busca. Uma *redução* do problema de busca *A* para o problema de busca *B* é um algoritmo de tempo polinomial *f* que transforma qualquer instância *I* de *A* em uma instância *f(I)* de *B*, junto com um outro algoritmo de tempo polinomial *h* que mapeia qualquer solução *S* de *f(I)* de volta para uma solução *h(S)* de *I*; veja o diagrama seguinte. Se *f(I)* não tem solução, então *I* também não tem. Os dois procedimentos de tradução *f* e *h* implicam que qualquer algoritmo para *B* pode ser convertido em um algoritmo para *A* quando colocado entre os “parênteses” *f* e *h*.



E agora podemos finalmente definir a classe dos problemas de busca mais difíceis.

Um problema de busca é NP-completo se todos os outros problemas de busca se reduzem para ele.

Esse requerimento é mesmo muito forte. Para um problema ser NP-completo, ele tem de ser útil na solução de qualquer problema de busca do mundo! É notável que tais problemas existam. Mas eles existem, e a primeira coluna da tabela que vimos antes está preenchida com os exemplos mais famosos. Na Seção 8.3 veremos como todos estes problemas se reduzem uns aos outros, e também por que todos os outros problemas de busca se reduzem a eles.

Fatoração

Um último ponto. Começamos este livro introduzindo outro problema de busca notoriamente famoso: **FATORAÇÃO**, a tarefa de encontrar todos os fatores primos de um dado inteiro. Mas a dificuldade de **FATORAÇÃO** é de uma natureza diferente daquela dos outros problemas de busca difíceis que acabamos de ver. Por exemplo, ninguém acredita que **FATORAÇÃO** seja NP-completo. Uma diferença grande é que, no caso de **FATORAÇÃO**, a definição não contém a já familiar cláusula “ou diga que nenhum existe”. Um número pode *sempre* ser fatorado em primos.

As duas maneiras de usar reduções

Até agora neste livro, o propósito de uma redução de um problema A para um problema B foi direto e honroso: sabemos como resolver B eficientemente e queremos usar este conhecimento para resolver A . Neste capítulo, entretanto, reduções de A para B servem a um objetivo de certo modo perverso: sabemos que A é difícil e usamos a redução para provar que B é difícil também!

Se denotamos uma redução de A para B por

$$A \longrightarrow B$$

então podemos dizer que a *dificuldade* flui na direção da seta, enquanto *algoritmos eficientes* movem-se na direção oposta. É por essa propagação de dificuldade que sabemos que problemas NP-completos são difíceis: todos os demais problemas de busca se reduzem a eles e, assim, cada problema NP-completo contém a complexidade de todos os problemas de busca. Se um problema NP-completo estiver em P, então $P = NP$.

Reduções também têm a propriedade conveniente de que podem ser *compostas*.

$$\text{Se } A \longrightarrow B \text{ e } B \longrightarrow C, \text{ então } A \longrightarrow C.$$

Para tanto, observe antes de mais nada que qualquer redução é completamente especificada pelas funções de pré e pós-processamento f e h (veja o diagrama de redução). Se (f_{AB}, h_{AB}) e (f_{BC}, h_{BC}) definem as reduções de A para B e de B para C , respectivamente, então uma redução de A para C é dada pela composição dessas funções: $f_{BC} \circ f_{AB}$ mapeia uma instância de A em uma instância de C e $h_{AB} \circ h_{BC}$ envia uma solução de C de volta para uma solução de A .

Isso significa que uma vez que saibamos que um problema A é NP-completo, podemos usá-lo para provar que um novo problema B também é NP-completo, simplesmente reduzindo A para B . Uma redução como essa estabelece que todos os problemas em NP se reduzem para B , via A .

Figura 8.6 O espaço NP de todos os problemas de busca, considerando que $P \neq NP$.

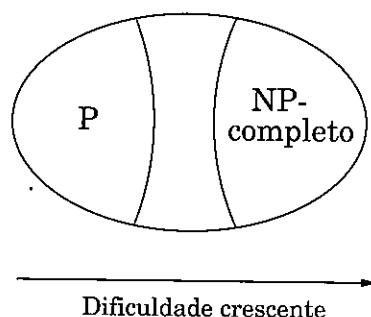
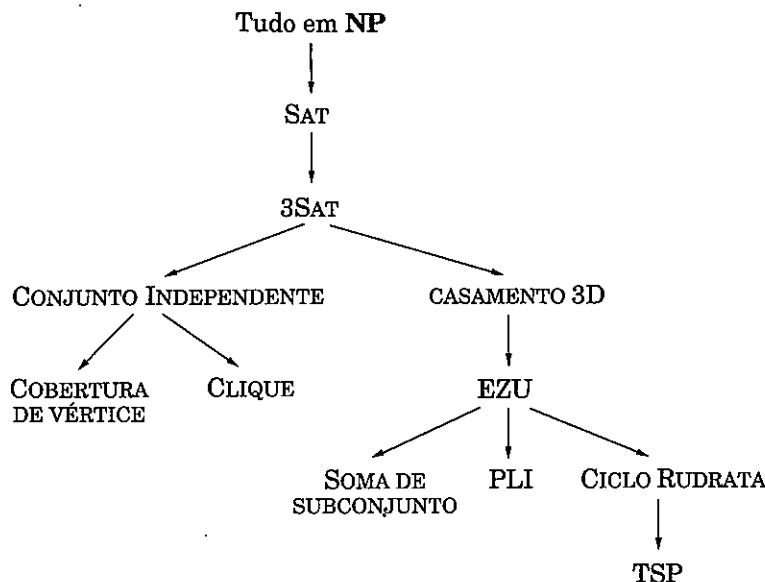


Figura 8.7 Reduções entre problemas de busca.



Outra diferença (possivelmente não completamente sem relação) é esta: como vemos no Capítulo 10, FATORAÇÃO sucumbe ao poder de *computação quântica* — enquanto SAT, TSP e os outros problemas NP-completos parecem não sucumbir.

8.3 As reduções

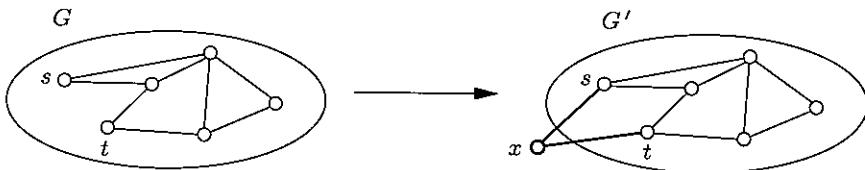
Veremos agora que os problemas de busca da Seção 8.1 podem ser reduzidos entre eles como mostra a Figura 8.7. Como consequência, eles são todos NP-completos.

Antes de abordar as reduções específicas na árvore, vamos nos aquecer relacionando duas versões do problema de Rudrata.

CAMINHO RUDRATA (s, t) \longrightarrow CICLO RUDRATA

Lembre-se do problema do CICLO RUDRATA: dado um grafo, será que existe um ciclo que passe por cada vértice exatamente uma vez? Podemos também formular o problema do CAMINHO RUDRATA (s, t), fortemente relacionado, no qual dois vértices s e t são especificados e queremos um caminho começando em s e terminando em t que passe por cada vértice exatamente uma vez. Será que é possível que o CICLO RUDRATA seja mais fácil do que o CAMINHO RUDRATA (s, t)? Vamos mostrar com uma redução que a resposta é não.

A redução mapeia uma instância ($G = (V, E)$, s, t) de CAMINHO RUDRATA (s, t) em uma instância $G' = (V', E')$ de CICLO RUDRATA da seguinte forma: G' é simplesmente G com um vértice adicional x e duas novas arestas $\{s, x\}$ e $\{x, t\}$. Por exemplo:



Assim $V' = V \cup \{x\}$ e $E' = E \cup \{\{s, x\}, \{x, t\}\}$. Como recuperamos um *caminho* Rudrata (s, t) em G dado qualquer *ciclo* Rudrata em G' ? Fácil, apenas removemos as arestas $\{s, x\}$ e $\{x, t\}$ do ciclo.



Para confirmarmos a validade dessa redução, temos de mostrar que ela funciona nos casos das duas saídas apresentadas:

1. Quando a instância do CICLO RUDRATA tem uma solução.

Como o novo vértice x tem apenas dois vizinhos, s e t , qualquer ciclo em G' tem de percorrer consecutivamente as arestas $\{t, x\}$ e $\{x, s\}$. O restante do ciclo, então, percorre todos os outros vértices no caminho de s para t . Assim, remover as duas arestas $\{t, x\}$ e $\{x, s\}$ do ciclo Rudrata leva a um caminho Rudrata de s para t no grafo original G .

2. Quando a instância do CICLO RUDRATA não tem uma solução.

Nesse caso, temos de mostrar que a instância original do CAMINHO RUDRATA (s, t) também não pode ter uma solução. Em geral é mais fácil provar a contrapositiva, isto é, mostrar que se existe um caminho Rudrata (s, t) em G , então existe também um ciclo Rudrata em G . Mas isto é fácil: apenas adicione as duas arestas $\{t, x\}$ e $\{x, s\}$ ao caminho Rudrata para fechar o ciclo.

Um último detalhe, crucial, mas tipicamente fácil de checar, é que as funções de pré e pós-processamento tomam tempo polinomial no tamanho da instância (G, s, t).

Também é possível ir na outra direção e reduzir CICLO RUDRATA ao CAMINHO RUDRATA (s, t). Juntas, estas reduções demonstram que as duas variantes do Rudrata são, em essência, o mesmo problema — o que não é muito surpreendente, dado que suas descrições

são quase iguais. Mas a maioria das outras reduções que veremos são entre pares de problemas que, à primeira vista, parecem bastante diferentes. Para mostrar que eles são essencialmente o mesmo, nossas reduções terão de traduzir inteligentemente entre eles.

3SAT → CONJUNTO INDEPENDENTE

Dificilmente poderíamos pensar em dois problemas mais diferentes. Na entrada do 3SAT é um conjunto de cláusulas, cada uma com três ou menos literais, por exemplo

$$(\bar{x} \vee y \vee \bar{z}) (x \vee \bar{y} \vee z) (x \vee y \vee z) (\bar{x} \vee \bar{y}),$$

e o objetivo é encontrar uma atribuição satisfatória. No CONJUNTO INDEPENDENTE a entrada é um grafo e um número g e o problema é encontrar um conjunto de g vértices não-adjacentes dois a dois. Temos que de alguma maneira relacionar lógica booleana com grafos!

Vamos pensar. Para formar uma atribuição satisfatória temos de selecionar um literal de cada cláusula e dar a ele o valor de verdadeiro. Mas nossas escolhas têm de ser coerentes: se escolhemos \bar{x} em uma cláusula, não podemos escolher x em outra. Qualquer escolha coerente de literais, um por cada cláusula, especifica uma atribuição (variáveis para as quais nenhum literal foi escolhido podem assumir qualquer valor).

Assim, vamos representar uma cláusula, digamos $(x \vee \bar{y} \vee z)$, por um triângulo, com vértices rotulados x , \bar{y} , z . Por que triângulo? Porque um triângulo tem seus vértices todos conectados e, assim, nos força a selecionar somente um deles para o conjunto independente. Repita essa construção para todas as cláusulas — uma cláusula com dois literais será representada simplesmente por uma aresta ligando os literais. (Uma cláusula com um literal é trivial e pode ser removida em um passo de pré-processamento, pois o valor da variável está determinado.) No grafo resultante, um conjunto independente tem de selecionar no máximo um literal de cada grupo (cláusula). Para forçar exatamente uma escolha por cada cláusula, tome o objetivo g como o número de cláusulas; no nosso exemplo, $g = 4$.

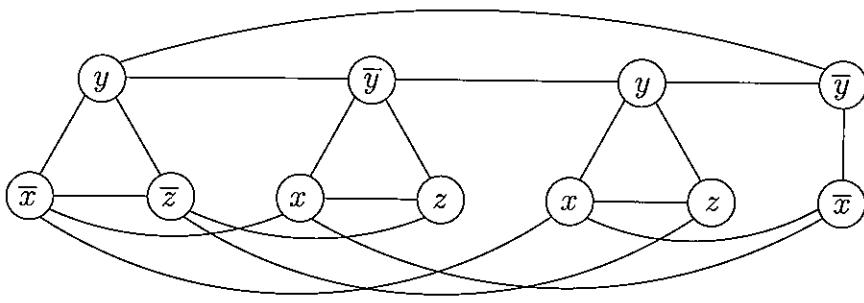
Tudo o que está faltando agora é uma maneira de evitar que escolhamos literais opostos (ambos x e \bar{x}) em cláusulas diferentes. Mas isso é fácil: coloque uma aresta entre quaisquer dois vértices que correspondam a literais opostos. O grafo resultante para o nosso exemplo é mostrado na Figura 8.8.

Vamos recapitular a construção. Dada uma instância i de 3SAT, criamos uma instância (G, g) do CONJUNTO INDEPENDENTE como se segue.

- O grafo G tem um triângulo para cada cláusula (ou apenas uma aresta, se a cláusula tem dois literais), com vértices rotulados pelos literais da cláusula, e tem arestas adicionais entre quaisquer dois vértices que representem literais opostos.
- O objetivo g é o número de cláusulas.

Claramente, essa construção toma tempo polinomial. Entretanto, lembre-se de que para uma redução não precisamos apenas de uma maneira eficiente de mapear instâncias do primeiro problema em instâncias do segundo (a função f no diagrama da página 245),

Figura 8.8 O grafo correspondendo a $(\bar{x} \vee y \vee \bar{z})$ $(x \vee \bar{y} \vee z)$ $(x \vee y \vee z)$ $(\bar{x} \vee \bar{y})$.



mas também de uma forma de reconstruir a solução da primeira instância de qualquer solução da segunda (a função h). Como sempre, há dois pontos a mostrar.

1. Dado um conjunto independente S de g vértices em G , é possível recuperar eficientemente uma atribuição satisfatória para I .

Para qualquer variável x , o conjunto S não pode conter vértices rotulados tanto x quanto \bar{x} , porque qualquer tal par de vértices está conectado por uma aresta. Assim, atribua a x um valor de **verdadeiro** se S contém um vértice rotulado x , e um valor de **falso** se S contém um vértice rotulado \bar{x} (se S não contém nenhum, então atribua qualquer valor a x). Como S tem g vértices, ele tem de ter um vértice por cláusula; esta atribuição satisfaz àqueles particulares literais e, assim, satisfaz a todas as cláusulas.

2. Se o grafo G não tem nenhum conjunto independente de tamanho g , a fórmula booleana I é insatisfatória.

Normalmente é mais claro provar a contrapositiva, que informa que se I tem uma atribuição satisfatória, G tem um conjunto independente de tamanho g . Isto é fácil: para cada cláusula, selecione qualquer literal cujo valor na atribuição satisfatória seja **verdadeiro** (tem de haver pelo menos um literal desse tipo), e adicione o vértice correspondente a S . Você entende por que o conjunto S tem de ser independente?

SAT \longrightarrow 3SAT

Trata-se de um tipo comum e interessante de redução de um problema para um *caso especial* dele mesmo. Queremos mostrar que o problema permanece difícil mesmo se suas entradas são restritas de alguma maneira — no caso presente, mesmo se todas as cláusulas forem restritas a ter ≤ 3 literais. As reduções modificam a instância para eliminar a característica proibida (cláusulas com ≥ 4 literais), mas mantendo a instância essencialmente a mesma, para que possamos ler uma solução para a instância original por meio de qualquer solução da modificada.

Aqui está o truque para reduzir SAT a 3SAT: dada uma instância I de SAT, use exatamente a mesma instância para 3SAT, exceto que qualquer cláusula com mais do que três literais, $(a_1 \vee a_2 \vee \dots \vee a_k)$ (onde os a_i são literais e $k > 3$), é substituída por um conjunto de cláusulas,

$$(a_1 \vee a_2 \vee y_1) (\bar{y}_1 \vee a_3 \vee y_2) (\bar{y}_2 \vee a_4 \vee y_3) \dots (\bar{y}_{k-3} \vee a_{k-1} \vee a_k),$$

onde os y_i são novas variáveis. Chame a instância 3SAT resultante de I' . A conversão de I para I' é claramente polinomial.

Por que essa redução funciona? I' é equivalente a I em termos de satisfatibilidade, porque para qualquer atribuição aos a_i ,

$$\left\{ \begin{array}{l} (a_1 \vee a_2 \vee \dots \vee a_k) \\ \text{é satisfeita} \end{array} \right\} \iff \left\{ \begin{array}{l} \text{existe uma valoração dos } y_i \text{ para a qual} \\ (a_1 \vee a_2 \vee y_1) (\bar{y}_1 \vee a_3 \vee y_2) \dots (\bar{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{são todos satisfeitos} \end{array} \right\}.$$

Para entender, primeiramente suponha que as cláusulas da direita sejam todas satisfeitas. Então pelo menos um literal a_1, \dots, a_k tem de ser verdadeiro — caso contrário y_1 teria de ser verdadeiro, o que forçaria y_2 a ser verdadeiro e assim por diante, até o momento em que a última cláusula seria falsificada. Mas isso significa que $(a_1 \vee a_2 \vee \dots \vee a_k)$ também é satisfeita.

Para a direção oposta, se $(a_1 \vee a_2 \vee \dots \vee a_k)$ é satisfeita, algum a_i tem de ser verdadeiro. Faça y_1, \dots, y_{i-2} iguais a verdadeiro e o restante a falso. Isso assegura que as cláusulas na direita sejam todas satisfeitas.

Assim, qualquer instância de SAT pode ser transformada em uma instância equivalente de 3SAT. De fato, 3SAT permanece difícil mesmo com a restrição adicional de que nenhuma variável apareça em mais do que três cláusulas. Para tanto, precisamos nos livrar de qualquer variável que apareça demasiadamente.

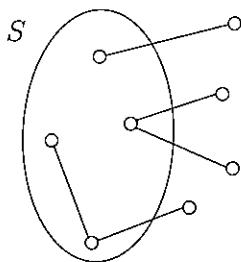
Veja uma redução de 3SAT para sua versão reduzida. Suponha que na instância do 3SAT, a variável x apareça em $k > 3$ cláusulas. Depois, substitua sua primeira ocorrência por x_1 , sua segunda ocorrência por x_2 e assim por diante, trocando cada uma de suas k ocorrências por uma nova e diferente variável. Por fim, adicione as cláusulas

$$(\bar{x}_1 \vee x_2) (\bar{x}_2 \vee x_3) \dots (\bar{x}_k \vee x_1).$$

E repita para toda variável que apareça mais de três vezes.

É fácil entender que na nova fórmula nenhuma variável aparece mais do que três vezes (e, de fato, nenhum literal aparece mais do que duas vezes). Além disso, as cláusulas extras envolvendo x_1, x_2, \dots, x_k restringem essas variáveis a terem o mesmo valor; você percebe o porquê? Assim a instância original de 3SAT é satisfatória se e somente se a instância restrita é satisfatória.

Figura 8.9 S é uma cobertura de vértice se e somente se $V - S$ é um conjunto independente.



CONJUNTO INDEPENDENTE \rightarrow COBERTURA DE VÉRTICE

Algumas reduções dependem de engenhosidade para relacionar dois problemas muito diferentes. Outras apenas aproveitam o fato de que um problema é um só pequeno disfarce do outro. Para reduzir CONJUNTO INDEPENDENTE a COBERTURA DE VÉRTICE, precisamos apenas notar que um conjunto de nós S é uma cobertura de vértices do grafo $G = (V, E)$ (ou seja, S toca cada aresta de E) se e somente se os nós restantes, $V - S$, forem um conjunto independente de G (Figura 8.9).

Portanto, para resolver uma instância (G, g) do CONJUNTO INDEPENDENTE, procure por uma cobertura de vértice de G com $|V| - g$ nós. Se tal cobertura existir, tome todos os nós que *não* estejam nela. Se nenhuma cobertura existir, G não pode ter um conjunto independente de tamanho g .

CONJUNTO INDEPENDENTE \rightarrow CLIQUE

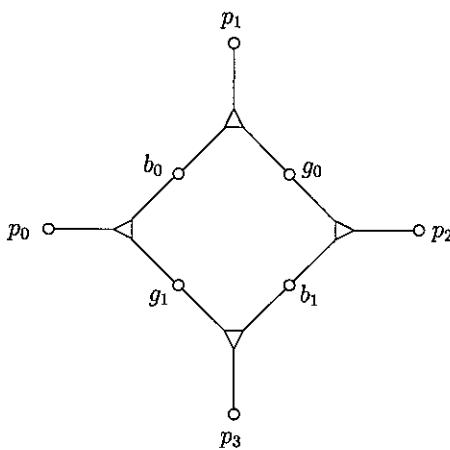
CONJUNTO INDEPENDENTE e CLIQUE também se reduzem facilmente um ao outro. Defina o *complemento* de um grafo $G = (V, E)$ como $\bar{G} = (V, \bar{E})$, onde \bar{E} contém precisamente aqueles pares não ordenados dos vértices que não estão em E . Em outras palavras, esses nós não têm nenhuma aresta entre eles em G se e somente se eles têm todas as possíveis arestas entre eles em \bar{G} .

Portanto, podemos reduzir CONJUNTO INDEPENDENTE a CLIQUE mapeando uma instância (G, g) de CONJUNTO INDEPENDENTE à instância correspondente (\bar{G}, g) de CLIQUE; as soluções de ambos são idênticas.

3SAT \rightarrow CASAMENTO 3D

Novamente, dois problemas muitos diferentes. Temos de reduzir 3SAT ao problema de encontrar, entre o conjunto de triplas garoto-garota-bicho, um subconjunto que conte-nha cada garoto, cada garota e cada bicho exatamente uma vez. Em resumo, temos de projetar conjuntos de triplas garoto-garota-bicho que, de alguma forma, se comportem como variáveis booleanas e portas!

Considere o seguinte conjunto de quatro triplas, cada uma representada por um nó triangular ligando um garoto, uma garota e um bicho:



Suponha que os dois garotos b_0 e b_1 e as duas garotas g_0 e g_1 não estejam envolvidos em nenhuma tripla. (Os quatro bichos p_0, \dots, p_3 vão, claro, pertencer às outras triplas também; caso contrário a instância trivialmente não teria qualquer solução.) Então qualquer casamento tem de conter uma das triplas (b_0, g_1, p_0) , (b_1, g_0, p_2) , ou as duas triplas (b_0, g_0, p_1) , (b_1, g_1, p_3) , porque estas são as únicas maneiras nas quais estes dois garotos e garotas podem encontrar algum casamento. Portanto, a “peça” tem dois possíveis estados: ela se comporta como uma variável booleana!

Para, então, transformar uma instância de 3SAT a uma de CASAMENTO 3D, começamos criando uma cópia da peça precedente para *cada* variável x . Chame os nós resultantes p_{x1}, b_{x0}, g_{x1} e assim sucessivamente. A interpretação pretendida é que o garoto b_{x0} seja casado com a garota g_{x1} se $x = \text{verdadeiro}$, e com a garota g_{x0} se $x = \text{falso}$.

Em seguida, precisamos criar triplas que de alguma maneira mimetizem cláusulas. Para cada cláusula, digamos $c = (x \vee \bar{y} \vee z)$, introduza um novo garoto b_c e uma nova garota g_c . Eles estarão envolvidos em três triplas, uma para cada literal na cláusula. E os bichos nessas triplas têm de refletir as três maneiras com as quais a cláusula pode ser satisfeita: (1) $x = \text{verdadeiro}$, (2) $y = \text{falso}$, (3) $z = \text{verdadeiro}$. Para (1), temos a tripla (b_c, g_c, p_{x1}) , onde p_{x1} é o bicho p_1 na peça de x . Veja por que escolhemos p_1 : se $x = \text{verdadeiro}$, então b_{x0} é casado com g_{x1} e b_{x1} com g_{x0} , e assim os bichos p_{x0} e p_{x2} são tomados. Caso no qual b_c e g_c podem ser casados com p_{x1} . Mas se $x = \text{falso}$, então p_{x1} e p_{x3} são tomados, e assim g_c e b_c não podem ser acomodados deste jeito. Fazemos a mesma coisa para os outros dois literais da cláusula, o que gera triplas envolvendo b_c e g_c com p_{y0} ou p_{y2} (para a variável negada y) e com p_{z1} ou p_{z3} (para a variável z).

Temos de nos assegurar de que para toda ocorrência de um literal em uma cláusula c exista um bicho diferente para casar com b_c e g_c . Mas isso é fácil: por uma redução anterior, podemos considerar que nenhum literal aparece mais do que duas vezes e, assim, cada peça de variável tem bichos suficientes, dois para ocorrências negadas e dois para não negadas.

A redução parece agora completa: de qualquer casamento podemos recuperar uma atribuição satisfatória simplesmente examinando em cada peça de variável e vendo com qual garota b_{x0} foi casado. E de qualquer atribuição satisfatória podemos casar a peça correspondente a cada variável x de modo que as triplas (b_{x0}, g_{x1}, p_{x0}) e (b_{x1}, g_{x0}, p_{x2}) sejam escolhidas se $x = \text{verdadeiro}$ e as triplas (b_{x0}, g_{x0}, p_{x1}) e (b_{x1}, g_{x1}, p_{x3}) sejam escolhidas se $x = \text{falso}$; e para cada cláusula c , casar b_c e g_c com o bicho que corresponda a um de seus literais satisfatórios.

Mas um último problema permanece: no casamento definido no final do último parágrafo, *alguns bichos podem ser deixados sem casamentos*. De fato, se existem n variáveis e m cláusulas, exatamente $2n - m$ bichos ficarão sem casamento (você pode checar que este número com certeza será positivo, porque temos no máximo três ocorrências de cada variável, e pelo menos dois literais em cada cláusula). Mas isso é fácil de corrigir: adicione $2n - m$ novos pares de garoto-garota que sejam “amantes genéricos de animais”, e case-os por triplas com todos os bichos!

CASAMENTO 3D → EZU

Lembre-se de que em EZU é dado uma matriz \mathbf{A} , $m \times n$, com células 0 – 1, e temos de encontrar um vetor 0 – 1 $\mathbf{x} = \{x_1, \dots, x_n\}$ tal que as m equações

$$\mathbf{Ax} = \mathbf{1}$$

sejam satisfeitas, onde denotamos por 1 o vetor coluna de 1. Como podemos expressar o problema CASAMENTO 3D neste esquema?

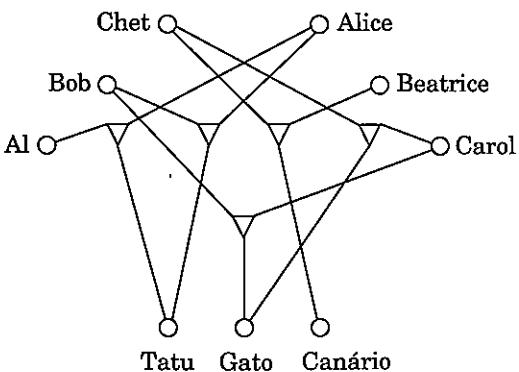
EZU e PLI são problemas muito úteis precisamente porque fornecem um formato no qual muitos problemas combinatoriais podem ser expressos. Em tal formulação pensamos nas variáveis 0 – 1 como descrevendo uma solução, e escrevemos equações expressando as restrições do problema.

Por exemplo, aqui está como expressamos uma instância de CASAMENTO 3D (m garotos, m garotas, m bichos e n triplas garoto-garota-bicho) na linguagem de EZU. Temos variáveis 0 – 1 x_1, \dots, x_n uma por tripla, onde $x_i = 1$ significa que a i -ésima tripla é escolhida para o casamento, e $x_i = 0$ significa que ela não é escolhida.

Agora tudo o que temos de fazer é escrever equações afirmando que a solução descrita pelos x_i é um casamento legítimo. Para cada garoto (ou garota, ou bicho), suponha que as triplas contendo ele (ou ela, ou o bicho) sejam aquelas numeradas j_1, j_2, \dots, j_k ; a equação apropriada é então

$$x_{j_1} + x_{j_2} + \dots + x_{j_k} = 1,$$

que afirma que exatamente uma dessas triplas tem de ser incluída no casamento. Por exemplo, aqui está a matriz \mathbf{A} para uma instância de CASAMENTO 3D que vimos antes.



$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

As cinco colunas de A correspondem às cinco triplas, enquanto as nove linhas são para Al, Bob, Chet, Alice, Beatrice, Tatu, Gato e Canário, respectivamente.

É simples argumentar que soluções para as duas instâncias se traduzem entre elas.

EZU → SOMA DE SUBCONJUNTO

Esta é uma redução entre dois casos especiais de PLI: um com muitas equações, mas somente coeficientes 0 – 1, e o outro com uma equação única, mas coeficientes inteiros arbitrários. A redução é baseada em uma idéia simples e já consagrada pelo tempo: vetores 0 – 1 podem codificar números!

Por exemplo, dada esta instância de EZU:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix},$$

estamos procurando por um conjunto de colunas de A que, adicionadas, resultam no vetor de 1. Mas se pensamos nas colunas como inteiros binários (lidos de cima para baixo), estamos procurando por um subconjunto dos inteiros 18, 5, 4, 8 cuja soma seja o inteiro binário $1111_2 = 31$. E isso é uma instância da SOMA DE SUBCONJUNTOS. A redução está completa!

Exceto por um detalhe, aquele que normalmente estraga a conexão próxima entre vetores 0 – 1 e inteiros binários: o *dígito de excesso*. Em virtude do excesso, inteiros binários de 5 bits podem somar até 31 (por exemplo, $5 + 6 + 20 = 31$ ou, em binário, $00101_2 + 00110_2 + 10100_2 = 11111_2$) mesmo quando a soma dos vetores correspondentes não é $(1, 1, 1, 1, 1)$. Mas isso é fácil de resolver: pense nos vetores-coluna não como inteiros em base 2, mas como inteiros na base $n + 1$ — um mais do que o número de colunas. Dessa forma, como no máximo n inteiros são adicionados, e todos os seus dígitos são 0 e 1, não pode haver excesso, e nossa redução funciona.

EZU → PLI

3SAT é um caso especial de SAT — ou SAT é uma generalização de 3SAT . Por *caso especial* queremos dizer que as instâncias de 3SAT são um subconjunto das instâncias de SAT (em particular, aquelas sem cláusulas longas), e a definição de solução é a mesma nos dois problemas (uma atribuição satisfazendo a todas as cláusulas). Conseqüentemente, existe uma redução de 3SAT para SAT , na qual a entrada não passa por qualquer transformação, e a solução para a instância objetivo também é deixada sem alteração. Em outras palavras, as funções f e h do diagrama de redução (na página 245) são ambas a identidade.

Isso parece trivial o suficiente, mas é uma maneira muito útil e comum de estabelecer que um problema é **NP-completo**: simplesmente notar que ele é uma generalização de um problema **NP-completo** conhecido. Por exemplo, o problema da **COBERTURA DE CONJUNTO** é **NP-completo** porque é uma generalização de **COBERTURA DE VÉRTICE** (e também, a propósito, de **CASAMENTO 3D**). Veja o Exercício 8.10 para mais exemplos.

Muitas vezes dá um pouco de trabalho estabelecer que um problema é um caso especial de outro. A redução de EZU para PLI é um caso destes. Em PLI estamos procurando por um vetor inteiro x que satisfaça $\mathbf{Ax} \leq \mathbf{b}$, para dada matriz A e vetor b . Para escrever uma instância de EZU nesta forma exata, precisamos reescrever cada equação da instância de EZU como duas inequações (veja as transformações da Seção 7.1.4) e adicionar para cada variável x_i as inequações $x_i \leq 1$ e $-x_i \leq 0$.

EZU → CICLO RUDRATA

No problema do **CICLO RUDRATA** procuramos um ciclo em um grafo que visite todos os vértices exatamente uma vez. Vamos provar que ele é **NP-completo** em dois estágios: primeiro reduziremos EZU para uma generalização do **CICLO RUDRATA**, chamada **CICLO RUDRATA COM ARESTAS EMPARELHADAS** e, então, veremos como nos

Figura 8.10 Ciclo Rudrata com arestas emparelhadas: $C = \{(e_1, e_3), (e_5, e_6), (e_4, e_5), (e_3, e_7), (e_3, e_8)\}$.

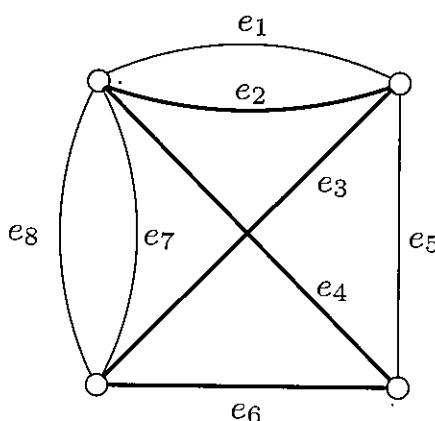
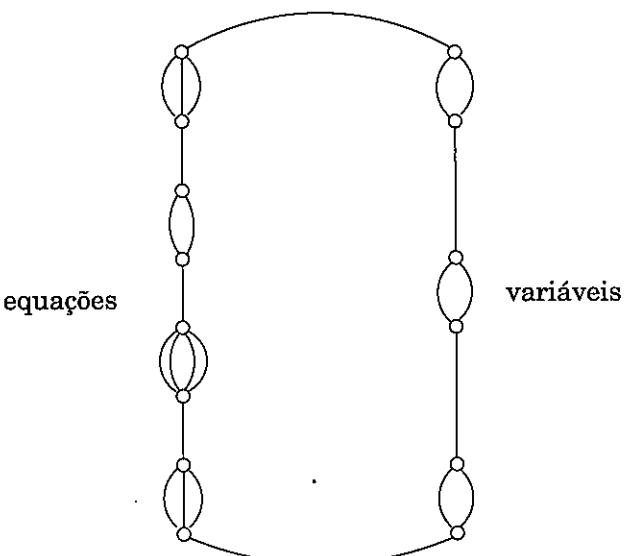


Figura 8.11 Reduzindo EZU para CICLO RUDRATA COM ARESTAS EMPARELHADAS.



livrar das características extras desse último problema e reduzi-lo para o problema do CICLO RUDRATA puro.

Em uma instância do CICLO RUDRATA COM ARESTAS EMPARELHADAS é dado um grafo $G = (V, E)$ e um conjunto $C \subseteq E \times E$ de pares de arestas. Procuramos um ciclo que (1) visite todos os vértices uma vez, como um ciclo Rudrata deve ser, e (2) para cada par de arestas (e, e') em C , passe pela aresta e , ou pela aresta e' , — *exatamente uma delas*. No exemplo da Figura 8.10 uma solução é apresentada em negrito. Note que permitimos duas ou mais arestas paralelas entre dois nós — uma característica que não faz sentido em muitos problemas em grafos —, pois agora cópias diferentes de uma aresta podem ser emparelhadas com outras cópias de arestas em maneiras que, de fato, fazem diferença.

Agora vejamos a redução de EZU para CICLO RUDRATA COM ARESTAS EMPARELHADAS. Dada uma instância de EZU, $Ax = 1$ (onde A é uma matriz $m \times n$ com células 0 — 1 e, assim, descreve m equações em n variáveis), o grafo que construímos tem a estrutura muito simples mostrada na Figura 8.11: um ciclo que conecta $m + n$ coleções de arestas paralelas. Para cada variável x_i , temos duas arestas paralelas (correspondendo a $x_i = 1$ e $x_i = 0$). E para cada equação $x_{j_1} + \dots + x_{j_k} = 1$ envolvendo k variáveis, temos k arestas paralelas, uma para cada variável aparecendo na equação. Esse é o grafo inteiro. Evidentemente, qualquer ciclo Rudrata neste grafo tem de percorrer as $m + n$ coleções de arestas paralelas uma por uma, escolhendo uma aresta de cada coleção. Assim, o ciclo “escolhe” para cada variável um valor — 0 ou 1 — e, para cada equação, uma variável aparecendo nela.

A redução inteira não pode ser tão simples assim, claro. A estrutura da matriz A (e não apenas as suas dimensões) tem de estar refletida de algum jeito, e há um lugar restante: o conjunto C de pares de arestas tal que exatamente uma aresta em cada par é percorrida. Para cada equação (lembre-se de que há m no total), e para cada variável x_i aparecendo nela, adicionamos a C o par (e, e') , onde e é a aresta correspondente à ocorrência de x_i naquela particular equação (no lado esquerdo da Figura 8.11), e e' é a aresta correspondente à atribuição de variável $x_i = 0$ (no lado direito da figura). Isso completa a construção.

Tome qualquer solução dessa instância do CICLO RUDRATA COM ARESTAS EMPARELHADAS. Como discutido anteriormente, ela seleciona um valor para cada variável e uma variável para cada equação. Afirmamos que os valores assim escolhidos são uma solução para a instância original de EZU. Se uma variável x_i tem valor 1, então a aresta $x_i = 0$ não é percorrida e, assim, todas as arestas associadas a x_i no lado da equação têm de ser percorridas (pois elas estão emparelhadas em C com a aresta $x_i = 0$). Portanto, em cada equação exatamente uma variável aparecendo nela tem valor 1 — que é o mesmo que dizer que todas as equações são satisfeitas. A outra direção é simples também: de uma solução da instância de EZU obtém-se facilmente um ciclo Rudrata apropriado.

Eliminando os pares de arestas

Até agora temos uma redução de EZU para CICLO RUDRATA COM ARESTAS EMPARELHADAS, mas estamos realmente interessados em CICLO RUDRATA, que é um caso especial do problema com arestas emparelhadas: aquele no qual o conjunto de pares C é vazio. Para alcançarmos nosso objetivo precisamos, como sempre, encontrar uma maneira de nos livrar da característica indesejada — neste caso, os pares de arestas.

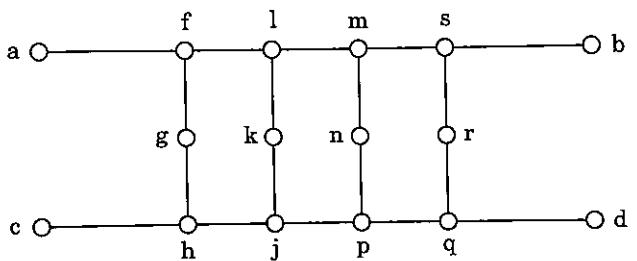
Considere o grafo da Figura 8.12 e suponha que seja parte de um grafo maior G de uma forma tal que somente as quatro extremidades a, b, c, d toquem o restante do grafo. Afirmamos que esse grafo tem a seguinte propriedade importante: *em qualquer ciclo Rudrata de G o subgrafo mostrado tem de ser percorrido de uma das duas maneiras mostradas em negrito na Figura 8.12(b) e (c)*. Veja o porquê. Suponha que o ciclo primeiro entre no subgrafo pelo vértice a e vá até f . Então ele tem de ir para o vértice g , porque g tem grau 2 e, portanto, tem de ser visitado imediatamente após um de seus nós adjacentes ser visitado — caso contrário, não existe maneira de incluí-lo no ciclo. Assim temos de ir para o vértice h e aqui parece que temos uma escolha. Poderíamos continuar até j , ou retornar para c . Mas se tomamos a segunda opção, como iremos visitar o restante do subgrafo? (Um ciclo Rudrata não pode deixar nenhum vértice não visitado.) É fácil ver que isso seria impossível e, assim, de h nós não temos escolha senão continuar até j e de lá visitar o resto do grafo como mostra a Figura 8.12(b). Por simetria, se o ciclo Rudrata entra neste subgrafo por c , ele tem de percorrê-lo como na Figura 8.12(c). E essas são as duas únicas formas.

Mas essa propriedade nos informa algo importante: a peça se comporta exatamente como duas arestas $\{a, b\}$ e $\{c, d\}$ que estão emparelhadas no problema do CICLO RUDRATA COM ARESTAS EMPARELHADAS [veja a Figura 8.12(d)].

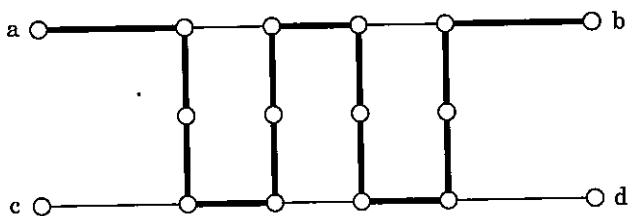
O resto da redução está claro agora: para reduzir CICLO RUDRATA COM ARESTAS EMPARELHADAS para CICLO RUDRATA, passamos pelos pares em C um por um. Para nos

Figura 8.12 Uma peça para forçar o comportamento de emparelhamento.

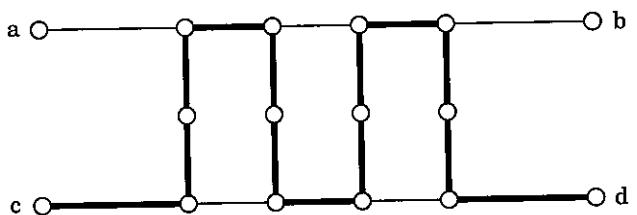
(a)



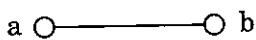
(b)



(c)



(d)



$$C = \{(\{a, b\}, \{c, d\})\}$$

livrar de cada par ($\{a, b\}$, $\{c, d\}$), substituímos as duas arestas pela peça da Figura 8.12(a). Para qualquer outro par de C que envolva $\{a, b\}$, substituímos a aresta $\{a, b\}$ pela nova aresta $\{a, f\}$, onde f é da peça: o percurso por $\{a, f\}$ é, de agora em diante, uma indicação de que a aresta $\{a, b\}$ no grafo antigo seria percorrido. De maneira similar, $\{c, h\}$ substitui $\{c, d\}$. Depois de $|C|$ tais substituições (realizadas em tempo polinomial, pois cada substituição adiciona somente 12 vértices ao grafo) estamos prontos, e os ciclos Rudrata no grafo resultante estarão em correspondência um-para-um com os ciclos Rudrata no grafo original que está em conformidade com as restrições de C .

CICLO RUDRATA \rightarrow TSP

Dado um grafo $G = (V, E)$, construa a seguinte instância do TSP: o conjunto de cidades é o mesmo que V , e a distância entre cidades u e v é 1 se $\{u, v\}$ é uma aresta de G e $1 + \alpha$ caso contrário, para algum $\alpha > 1$ ser determinado. O orçamento da instância TSP é igual ao número de nós, $|V|$.

É fácil ver que se G tem um ciclo Rudrata, o mesmo ciclo é também um circuito dentro do orçamento da instância do TSP; e que pelo lado oposto, se G não tem um ciclo Rudrata, não existe solução: o circuito TSP mais barato possível tem um custo de pelo menos $n + \alpha$ (ele tem de usar pelo menos uma aresta de tamanho $1 + \alpha$, e o comprimento total de todos os outros $n - 1$ é pelo menos $n - 1$). Assim CICLO RUDRATA se reduz a TSP.

Nessa redução, introduzimos o parâmetro α porque variando-o, podemos obter dois resultados interessantes. Se $\alpha = 1$, então todas as distâncias são ou 1 ou 2 e, assim, a instância do TSP satisfaz à inequação do triângulo: se i, j, k são cidades, então $d_{ij} + d_{jk} \geq d_{ik}$ (prova: $a + b \geq c$ vale para quaisquer números $1 \leq a, b, c \leq 2$). Isso é um caso especial do TSP que é de importância prática e que, como veremos no Capítulo 9, é em certo sentido mais fácil, porque ele pode ser eficientemente *aproximado*.

Se, por sua vez, α é grande, a instância resultante do TSP pode não satisfazer a inequação do triângulo, mas tem uma outra propriedade importante: ou ele tem uma solução de custo n ou menos, ou todas as suas soluções têm custo pelo menos $n + \alpha$ (que agora pode ser arbitrariamente maior do que n). Não pode haver nada no meio! Como veremos no Capítulo 9, essa importante propriedade *do não* implica que, a menos que $P=NP$, nenhum algoritmo de aproximação é possível.

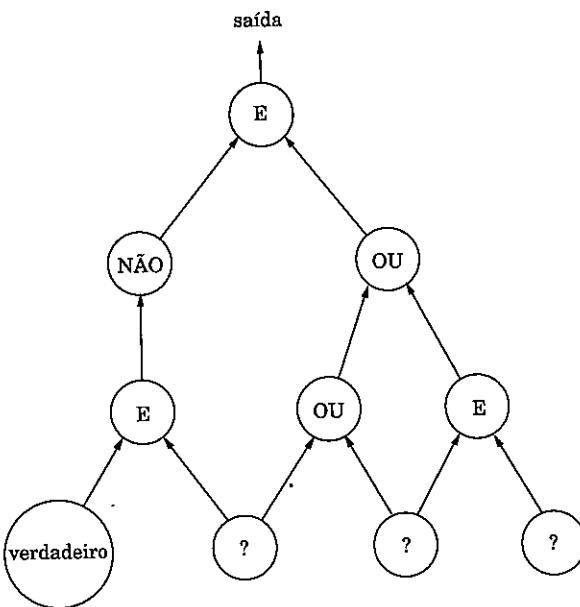
QUALQUER PROBLEMA EM NP \rightarrow SAT

Reduzimos SAT aos vários problemas de busca na Figura 8.7. Agora fechamos o círculo e argumentamos que todos os problemas — e de fato todos os problemas em NP — se reduzem a SAT.

Em particular, mostraremos que todos os problemas em NP podem ser reduzidos a uma generalização de SAT que chamamos SAT CIRCUITO. Em SAT CIRCUITO é dado um *circuito booleano* (veja a Figura 8.13 e a Seção 7.7), um dag cujos vértices são *portas* de cinco tipos diferentes:

- Portas E e portas ou têm grau de entrada 2.
- Portas NÃO têm grau de entrada 1.

Figura 8.13 Uma instância de SAT CIRCUITO.



- Portas de *entrada conhecidas* não têm nenhuma aresta de entrada e são rotuladas como **falso** ou **verdadeiro**.
- Portas de *entrada desconhecidas* não têm nenhuma aresta de entrada e são rotuladas como “?”.

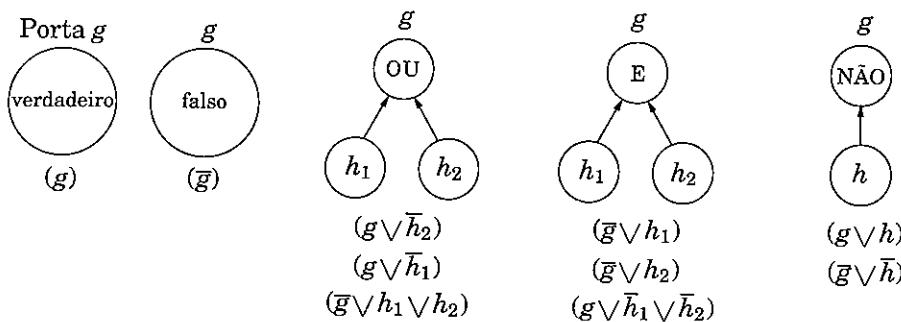
Um dos sorvedouros do dag é designado como a porta de *saída*.

Dada uma atribuição de valores para as entradas desconhecidas, podemos avaliar as portas do circuito em ordem topológica, usando as regras da lógica booleana (tais como **falso** \vee **verdadeiro** = **verdadeiro**), até obtermos o valor da porta de saída. Esse é o valor do circuito para a particular atribuição às entradas. Por exemplo, o circuito na Figura 8.13 avalia para **falso** sob a atribuição **verdadeiro**, **falso**, **verdadeiro** (da esquerda para a direita).

SAT CIRCUITO é, então, o seguinte problema de busca: dado um circuito, encontre uma atribuição para as entradas desconhecidas tal que a porta de saída avalie para **verdadeiro**, ou declare que nenhuma atribuição existe. Por exemplo, para o circuito na Figura 8.13, poderíamos ter retornado a atribuição (**falso**, **verdadeiro**, **verdadeiro**) porque, se substituirmos esses valores nas entradas desconhecidas (da esquerda para a direita), a saída irá se tornar **verdadeiro**.

SAT CIRCUITO é uma generalização de SAT. Para entender, note que SAT pergunta por uma atribuição satisfatória para um circuito que tem esta estrutura simples: um grupo de portas **E** une as cláusulas no topo, e o resultado deste grande **E** é a saída. Cada cláusula é ou de seus literais. E cada literal é ou uma entrada desconhecida ou o **NÃO** de uma. Não existem portas de entrada conhecidas.

Indo na outra direção, SAT CIRCUITO também pode ser reduzido a SAT. Veja como podemos reescrever qualquer circuito em forma normal conjuntiva (o E de cláusulas): para cada porta g no circuito, criamos uma variável g , e modelamos o efeito da porta usando umas poucas cláusulas:



(Você entende por que essas cláusulas forçam mesmo exatamente o efeito desejado?) E para terminar, se g é a porta de saída, nós forçamos o valor *verdadeiro* para ela adicionando a cláusula (g) . A instância resultante de SAT é equivalente à dada instância de SAT CIRCUITO: as atribuições satisfatórias desta forma normal conjuntiva estão em correspondência um-para-um com as do circuito.

Agora que sabemos que SAT CIRCUITO se reduz a SAT, voltemos a nossa tarefa principal, mostrar que *todos* os problemas de busca se reduzem a SAT CIRCUITO. Assim, suponha que A seja um problema em NP. Temos de descobrir uma redução de A para SAT CIRCUITO. Isso parece muito difícil, *porque não sabemos quase nada sobre A!*

Tudo o que sabemos sobre A é que ele é um problema de busca, portanto precisamos desse conhecimento para trabalhar. A principal característica de um problema de busca é que qualquer solução para ele pode ser rapidamente verificada: existe um algoritmo C que checa, dada uma instância I e uma solução proposta S , se S é ou não uma solução para I . Além disso, C toma essa decisão em tempo polinomial no tamanho de I (podemos considerar que a própria S é codificada como uma *string* binária, e sabemos que o comprimento desta *string* é polinomial no tamanho de I).

Lembre-se agora de nosso argumento da Seção 7.7 de que qualquer algoritmo polinomial pode ser representado por um circuito, cujas portas de entrada codificam a entrada do algoritmo. Naturalmente, para qualquer comprimento de entrada (número de bits de entrada) o circuito vai ser escalado para o número apropriado de entradas, mas o número total de portas do circuito será polinomial no número de entradas. Se o algoritmo polinomial em questão resolve um problema que requer uma resposta sim ou não (como é a situação com C : “Será que S codifica uma solução para a instância codificada por I ?”), então essa resposta é dada na porta de saída.

Concluímos que, dada qualquer instância I do problema A , podemos construir em tempo polinomial um circuito cujas entradas conhecidas são os bits de I , e cujas entradas desconhecidas são os bits de S , tal que a saída é verdadeiro se e somente se as saídas desconhecidas representam uma solução S de I . Em outras palavras, *as atribuições satisfatórias para as entradas desconhecidas do circuito estão em correspondência um-para-um com as soluções da instância I de A*. A redução está completa.

Problemas insolúveis

Um problema NP-completo ao menos pode ser solucionado por *algum* algoritmo — o problema é que este algoritmo será exponencial. Mas acontece que existem problemas computacionais perfeitamente decentes para os quais *nenhum algoritmo existe em absoluto*!

Um problema famoso desse tipo é uma versão aritmética de SAT. Dada uma equação polinomial em muitas variáveis, talvez

$$x^3yz + 2y^4z^2 - 7xy^5z = 6,$$

será que existem valores inteiros de x, y, z que a satisfazem? Não há *nenhum algoritmo* que solucione esse problema. Nenhum algoritmo em absoluto, polinomial, exponencial, duplamente exponencial, ou pior! Tais problemas são chamados *insolúveis (indecidíveis)*.

O primeiro problema insolúvel foi descoberto em 1936 por Alan M. Turing, na época uma estudante de matemática em Cambridge, Inglaterra. Quando Turing o apresentou, não havia computadores ou linguagens de programação (pode-se de fato afirmar que essas tecnologias apareceram depois *exatamente porque* este pensamento brilhante ocorreu a Turing). Mas hoje podemos formulá-lo em termos familiares.

Suponha que seja dado um programa na sua linguagem de programação favorita, junto com uma particular entrada. Será que o programa vai terminar, uma vez começado com essa entrada? Essa é uma questão muito razoável. Muitos de nós acharíamos fantástico se tivéssemos um algoritmo, chame-o `termina(p, x)`, que tomasse como entrada um arquivo contendo um programa p , um arquivo de dados x e, depois de trabalhar, finalmente nos informasse se p iria ou não parar se começado com x .

Mas como você faria para escrever o programa `termina`? (Se você nunca viu isto antes, vale a pena pensar sobre isso por um momento, para apreciar a dificuldade de escrever um tal “detector universal de loop infinito”.)

Bem, você não pode fazer. *Esse algoritmo não existe!*

E aqui está a prova: suponha que, de fato, tenhamos um tal programa `termina(p, x)`. Então poderíamos usá-lo como uma sub-rotina do seguinte programa malicioso:

```
função paradoxo(z: arquivo)
1: se termina(z,z) goto 1
```

Problemas insolúveis (*continuação*)

Note o que paradoxo faz: ele termina se e somente se o programa *z* não termina quando a ele é dado o seu próprio código como entrada.

Você deve perceber o problema. O que dizer se colocarmos esse programa em um arquivo com o nome de paradoxo e se executarmos paradoxo(paradoxo)? Será que a execução vai finalmente parar? Ou não? Nenhuma resposta é possível. Como chegamos a essa contradição assumindo que existe um algoritmo que nos informa se programas terminam, temos de concluir que este problema não pode ser solucionado por nenhum algoritmo.

A propósito, isso tudo nos diz algo importante sobre programação: ela nunca será automatizada, ela dependerá sempre de disciplina, engenhosidade e *hackery*. Agora sabemos que não se pode dizer se um programa tem um loop infinito. Mas você pode dizer se ele tem um estouro de buffer? Você sabe como usar a indecidibilidade do “problema da parada” para mostrar que isto também é insolúvel?

Exercícios

- 8.1. *Otimização versus busca.* Consulte o problema do caixeiro-viajante:

TSP

Entrada: Uma matriz de distâncias; um orçamento b

Saída: Um circuito que passe por todas as cidades e que tenha comprimento $\leq b$, se um tal circuito existir.

A versão de otimização deste problema pergunta diretamente pelo menor circuito.

TSP-OPT

Entrada: Uma matriz de distâncias

Saída: O menor circuito que passe por todas as cidades.

Mostre que se TSP pode ser resolvido em tempo polinomial, então TSP-OPT também pode.

- 8.2. *Busca versus decisão.* Suponha que você tenha um procedimento que execute em tempo polinomial e que informe se um grafo tem ou não um caminho Rudrata. Mostre que você pode usá-lo para desenvolver um algoritmo de tempo polinomial para o CAMINHO RUDRATA (que retorna o caminho propriamente, se ele existe).

- 8.3. SAT AVARENTO é o seguinte problema: dado um conjunto de cláusulas (cada uma delas uma disjunção de literais) e um inteiro k , encontre uma atribuição satisfatória na qual no máximo k variáveis são verdadeiro, se tal atribuição existir. Prove que SAT AVARENTO é NP-completo.

- 8.4. Considere o problema da CLIQUE restrito a grafos nos quais todo vértice tem grau no máximo 3. Chame este problema de CLIQUE-3.

- (a) Prove que CLIQUE-3 está em NP.

(b) O que está errado com a seguinte prova de NP-completude para CLIQUE-3? Sabemos que o problema da CLIQUE em grafos gerais é NP-completo, portanto é suficiente apresentar uma redução de CLIQUE-3 para CLIQUE. Dado um grafo G com vértices de grau ≤ 3 , e um parâmetro g , a redução deixa o grafo e o parâmetro sem modificação: claramente a saída da redução é uma entrada possível para o problema da CLIQUE. Além disso, a resposta para ambos os problemas é idêntica. Isso prova a correção da redução e, portanto, a NP-completude de CLIQUE-3.

(c) É verdade que o problema da COBERTURA DE VÉRTICE permanece NP-completo mesmo quando restrito a grafos nos quais todo vértice tem grau no máximo 3. Chame este problema de cv-3. O que está errado com a seguinte prova de NP-completude para CLIQUE-3?

Apresentamos uma redução de cv-3 para CLIQUE-3. Dado um grafo $G = (V, E)$ com graus de nós limitados a 3, e um parâmetro b , construímos uma instância de CLIQUE-3 deixando o grafo inalterado e mudando o parâmetro para $|V| - b$. Agora, um subconjunto $C \subseteq V$ é uma cobertura de vértice em G se e somente se o conjunto complementar $V - C$ é uma clique em G . Portanto, G tem uma cobertura de vértice de tamanho $\leq b$ se e somente se ele tem uma clique de tamanho $\geq |V| - b$. Isso prova a correção da redução e, consequentemente, a NP-completude de CLIQUE-3.

(d) Descreva um algoritmo $O(|V|)$ para CLIQUE-3.

- 8.5. Forneça uma redução simples de CASAMENTO 3D para SAT, e outra de CICLO RUDRATA para SAT. (Dica: No segundo caso você pode usar variáveis x_{ij} cujo significado intuitivo é “vértice i é o j -ésimo vértice do ciclo Rudrata”; então, você precisa escrever cláusulas que expressem as restrições do problema.)
- 8.6. Na página 251, vimos que 3SAT permanece NP-completo mesmo quando restrito a fórmulas nas quais cada literal aparece no máximo duas vezes.
 - (a) Mostre que se cada literal aparece no máximo *uma vez*, o problema é solúvel em tempo polinomial.
 - (b) Mostre que o CONJUNTO INDEPENDENTE permanece NP-completo mesmo no caso especial quando todos os nós do grafo têm grau no máximo 4.
- 8.7. Considere um caso especial de 3SAT no qual todas as cláusulas têm exatamente três literais e cada variável aparece exatamente três vezes. Mostre que este problema pode ser resolvido em tempo polinomial. (Dica: Crie um grafo bipartido com cláusulas na parte esquerda, variáveis na parte direita, e arestas sempre que uma variável aparece em uma cláusula. Use o Exercício 7.30 para mostrar que este grafo tem um emparelhamento.)
- 8.8. No problema 4SAT EXATO, a entrada é um conjunto de cláusulas, cada uma é uma disjunção de exatamente quatro literais, e cada variável ocorre no máximo uma vez em cada cláusula. O objetivo é encontrar uma atribuição satisfatória, se existir. Prove que 4SAT EXATO é NP-completo.

- 8.9. No problema CONJUNTO INCIDENTE, é dada uma família de conjuntos $\{S_1, S_2, \dots, S_n\}$ e um orçamento b , e desejamos encontrar um conjunto H de tamanho $\leq b$ que intercepte todos os S_i , se um tal H existir. Em outras palavras, queremos $H \cap S_i \neq \emptyset$ para todo i .

Mostre que CONJUNTO INCIDENTE é NP-completo.

- 8.10. *Provando NP-completude por generalização.* Para cada um dos problemas a seguir, prove que é NP-completo mostrando que ele é uma *generalização* de algum problema NP-completo que vimos neste capítulo.

- (a) ISOMORFISMO DE GRAFO: dados como entrada dois grafos não-direcionados G e H , determine se G é um subgrafo de H (isto é, se ao removermos certos vértices e arestas de H obtemos um grafo que é, a menos de renomeação de vértices, idêntico a G), e se for o caso, retorne o mapeamento correspondente de $V(G)$ em $V(H)$.
- (b) CAMINHO MAIS LONGO: dado um grafo G e um inteiro g , encontre em G um caminho simples de comprimento g .
- (c) MAX SAT: dada uma fórmula CNF e um inteiro g , encontre uma atribuição que satisfaça pelo menos g cláusulas.
- (d) SUBGRAFO DENSO: dados um grafo G e dois inteiros a e b , encontre um conjunto de a vértices de G tal que exista pelo menos b arestas entre eles.
- (e) SUBGRAFO ESPARSO: dados um grafo G e dois inteiros a e b , encontre um conjunto de a vértices de G tal que exista no máximo b arestas entre eles.
- (f) COBERTURA DE CONJUNTO. (Este problema generaliza *dois* problemas NP-completos conhecidos.)
- (g) REDE CONFIÁVEL: é dado duas matrizes $n \times n$, uma matriz de distância d_{ij} e uma matriz de *requerimento de conectividade* r_{ij} , bem como um orçamento b ; temos de encontrar um grafo $G = (\{1, 2, \dots, n\}, E)$ tal que (1) o custo total de todas as arestas seja b ou menos e (2) entre quaisquer dois vértices distintos i e j existam r_{ij} caminhos vértice-disjuntos. (*Dica:* Suponha que todos os d_{ij} sejam 1 ou 2, $b = n$, e que todos os r_{ij} sejam 2. Qual problema NP-completo bem conhecido é este?)

- 8.11. Existem muitas variantes do problema de RUDRATA, dependendo de o grafo ser não-direcionado ou direcionado, e de um ciclo ou caminho ser pedido. Reduza o problema do CAMINHO RUDRATA DIRECIONADO a cada um dos seguintes.

- (a) O problema (não-direcionado) do CAMINHO RUDRATA.
- (b) O problema não-direcionado do CAMINHO RUDRATA (s, t) , que é exatamente como o CAMINHO RUDRATA, exceto que as extremidades do caminho são especificadas na entrada.

- 8.12. O problema da k -ÁRVORE ESPALHADA é o seguinte.

Entrada: Um grafo não-direcionado $G = (V, E)$

Saída: Uma árvore espalhada de G na qual cada nó tem grau $\leq k$, se uma árvore desse tipo existe.

Mostre que para todo $k \geq 2$:

- (a) *k*-ÁRVORE ESPALHADA é um problema de busca.
- (b) *k*-ÁRVORE ESPALHADA é NP-completo. (*Dica:* Comece com $k = 2$ e considere a relação entre este problema e o CAMINHO RUDRATA.)
- 8.13. Determine quais dos seguintes problemas são NP-completos e quais são solúveis em tempo polinomial. Em cada problema, é dado um grafo não-direcionado $G = (V, E)$, além de:
- Um conjunto de nós $L \subseteq V$, e você tem de encontrar uma árvore espalhada tal que o conjunto de folhas dela contém o conjunto L .
 - Um conjunto de nós $L \subseteq V$, e você tem de encontrar uma árvore espalhada tal que o conjunto de folhas dela é precisamente o conjunto L .
 - Um conjunto de nós $L \subseteq V$, e você tem de encontrar uma árvore espalhada tal que o conjunto de folhas dela está contido no conjunto L .
 - Um inteiro k , e você tem de encontrar uma árvore espalhada com k ou menos folhas.
 - Um inteiro k , e você tem de encontrar uma árvore espalhada com k ou mais folhas.
 - Um inteiro k , e você tem de encontrar uma árvore espalhada com exatamente k folhas.
- (*Dica:* Todas as provas de NP-completude são por generalização, exceto uma.)
- 8.14. Prove que o seguinte problema é NP-completo: dados um grafo não-direcionado $G = (V, E)$ e um inteiro k , retorne uma clique de tamanho k e também um conjunto independente de tamanho k , desde que ambos existam.
- 8.15. Mostre que o seguinte problema é NP-completo.
- SUBGRAFO COMUM MÁXIMO
- Entrada:* Dois grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$; um orçamento b .
- Saída:* Dois conjuntos de nós $V'_1 \subseteq V_1$ e $V'_2 \subseteq V_2$ cuja remoção deixa pelo menos b nós em cada grafo, e torna os dois grafos idênticos.
- 8.16. Estamos com vontade de experimentar e queremos criar uma nova receita. Existem vários ingredientes que podemos escolher e gostaríamos de usar o maior número possível deles, mas alguns ingredientes não combinam bem com outros. Se há n ingredientes possíveis (numerados de 1 a n), escrevemos uma matriz $n \times n$ dando a *incompatibilidade* entre qualquer par de ingredientes. Essa *incompatibilidade* é um número real entre 0,0 e 1,0, onde 0,0 significa “eles se combinam perfeitamente” e 1,0 significa “eles são realmente incompatíveis”. A seguir está uma matriz de exemplo para quando há cinco possíveis ingredientes:

	1	2	3	4	5
1	0,0	0,4	0,2	0,9	1,0
2	0,4	0,0	0,1	1,0	0,2
3	0,2	0,1	0,0	0,8	0,5
4	0,9	1,0	0,8	0,0	0,2
5	1,0	0,2	0,5	0,2	0,0

Neste caso, os ingredientes 2 e 3 se combinam muito bem, enquanto 1 e 5 são uma combinação muito ruim. Note que a matriz é necessariamente simétrica e que as células diagonais são sempre 0,0. Qualquer conjunto de ingredientes incorre em uma *penalidade* que é a soma de todos os valores de incompatibilidade entre pares de ingredientes. Por exemplo, o conjunto de ingredientes {1, 3, 5} incorre em uma penalidade de $0,2 + 1,0 + 0,5 = 1,7$. Queremos que esta penalidade seja pequena.

COZINHA EXPERIMENTAL

Entrada: n , o número de ingredientes disponíveis; D , a matriz $n \times n$ de “incompatibilidade”; algum número $p \geq 0$

Saída: O número máximo de ingredientes que podemos escolher com penalidade $\leq p$.

Mostre que se COZINHA EXPERIMENTAL é solúvel em tempo polinomial, então 3SAT também é.

- 8.17. Mostre que para qualquer problema Π em NP, existe um algoritmo que resolve Π em tempo $O(2^{p(n)})$, onde n é o tamanho da instância de entrada e $p(n)$ é um polinômio (que pode depender de Π).
- 8.18. Mostre que se P=NP, o sistema de criptografia RSA (Seção 1.4.2) pode ser quebrado em tempo polinomial.
- 8.19. Uma *pipa* é um grafo sobre um número par de vértices, digamos $2n$, nos quais n dos vértices formam uma clique e os restantes n vértices são conectados em um “rabo” que consiste em um caminho incidente a um dos vértices da clique. Dado um grafo e um objetivo g , o problema da PIPA pede que se encontre um subgrafo que seja uma pipa e que contenha $2g$ nós. Prove que PIPA é NP-completo.
- 8.20. Em um grafo não-direcionado $G = (V, E)$, dizemos que $D \subseteq V$ é um *conjunto dominante* se qualquer $v \in V$ ou está em D ou é adjacente a pelo menos um membro de D . No problema do CONJUNTO DOMINANTE, a entrada é um grafo e um orçamento b , e o objetivo é encontrar um conjunto dominante no grafo de tamanho no máximo b , se, assim, existir. Prove que este problema é NP-completo.
- 8.21. *Seqüenciamento por hibridização.* Um procedimento experimental para identificar uma nova seqüência de DNA atesta repetidamente para determinar quais substrings de comprimento k ela contém. Baseado nessas substrings, a seqüência total tem de ser reconstruída.

Vamos agora formular como um problema combinatório. Para qualquer string x (a seqüência de DNA), seja $\Gamma(x)$ o multiconjunto de todas as suas substrings de comprimento k . Em particular, $\Gamma(x)$ contém exatamente $|x| - k + 1$ elementos.

O problema da reconstrução é fácil de formular agora: dado um multiconjunto de strings de tamanho k , encontre a string x tal que $\Gamma(x)$ é exatamente este multiconjunto.

- (a) Mostre que o problema da reconstrução se reduz a CAMINHO RUDRATA. (*Dica:* Construa um grafo direcionado com um nó para cada substring de tamanho k , e com uma aresta de a para b se os últimos $k - 1$ caracteres de a são iguais aos $k - 1$ primeiros caracteres de b .)

(b) Mas, de fato, há notícias muito melhores. Mostre que o mesmo problema também se reduz a CAMINHO EULERIANO. (Dica: Desta vez, use uma aresta direcionada para cada substring de tamanho k .)

- 8.22. Em escalonamento de tarefas, é comum usar uma representação de grafo com um nó para cada tarefa e uma aresta direcionada da tarefa i para a tarefa j se i é uma pré-condição para j . Esse grafo direcionado representa as restrições de precedência no problema de escalonamento. Claramente, um escalonamento é possível se e somente se o grafo for acíclico; se ele não for, gostaríamos de encontrar o menor número de restrições que têm de ser removidas para torná-lo acíclico.

Dado um grafo direcionado $G = (V, E)$, um subconjunto $E' \subseteq E$ é chamado de *conjunto de arco de retorno* se a remoção das arestas E' torna G acíclico.

CONJUNTO DE ARCO DE RETORNO (CAR): dado um grafo direcionado $G = (V, E)$ e um orçamento b , encontre um conjunto de arco de retorno $\leq b$ arestas, se, assim, existir.

- (a) Mostre que CAR está em NP.

Podemos mostrar que CAR é NP-completo com uma redução a partir de COBERTURA DE VÉRTICE. Dada uma instância (G, b) de COBERTURA DE VÉRTICE, onde G é um grafo não-direcionado e queremos uma cobertura de vértice de tamanho $\leq b$, construímos uma instância (G', b) de CAR como se segue. Se $G = (V, E)$ tem n vértices v_1, \dots, v_n , então faça $G' = (V', E')$ um grafo direcionado com $2n$ vértices $w_1, w'_1, \dots, w_n, w'_n$, e $n + 2|E|$ arestas (direcionadas):

- (w_i, w'_i) para todo $i = 1, 2, \dots, n$.
- (w'_i, w_j) e (w'_j, w_i) para toda $(v_i, v_j) \in E$.

(b) Mostre que se G contém uma cobertura de vértice de tamanho b , então G' contém um conjunto de arco de retorno de tamanho b .

(c) Mostre que se G' contém um conjunto de arco de retorno de tamanho b , então G tem uma cobertura de vértice de tamanho (no máximo) b . (Dica: Dado um conjunto de arco de retorno de tamanho b em G' , você pode ter de modificá-lo ligeiramente para obter um outro que tenha uma forma mais conveniente, mas é do mesmo tamanho ou menor. Então, argumente que G tem de conter uma cobertura de vértice do mesmo tamanho que o conjunto de arco de retorno modificado.)

- 8.23. No problema dos CAMINHOS NÓS-DISJUNTOS, a entrada é um grafo não-direcionado no qual alguns vértices foram especialmente marcados: um certo número de “fontes” s_1, s_2, \dots, s_k e um número igual de “destinos” t_1, t_2, \dots, t_k . O objetivo é encontrar k caminhos nós-disjuntos (isto é, caminhos que não têm nenhum nó em comum) onde o i -ésimo caminho vai de s_i para t_i . Mostre que este problema é NP-completo.

Veja uma seqüência de dicas progressivamente mais fortes.

- (i) Reduza de 3SAT.
- (ii) Para uma fórmula 3SAT com m cláusulas e n variáveis, use $k = m + n$ fontes e destinos. Introduza um par fonte/destino (s_x, t_x) para cada variável x , e um par fonte/destino (s_c, t_c) para cada cláusula c .

- (iii) Para cada cláusula 3SAT, introduza 6 novos vértices intermediários, um para cada literal ocorrendo naquela cláusula e um para o seu complemento.
- (iv) Note que se o caminho de s_c para t_c passa por algum vértice intermediário representando, digamos, uma ocorrência da variável x , então nenhum outro caminho pode passar por aquele vértice. Por qual outro vértice você gostaria de forçar o outro caminho a passar?

Capítulo 9

Lidando com NP-completude

Você é o membro júnior de um time antigo de projetistas. Sua tarefa atual é escrever código para resolver um problema de aparência simples envolvendo grafos e números. O que você deve fazer?

Se tiver sorte, seu problema estará entre aquela meia dúzia de problemas sobre grafos com pesos (caminho mínimo, árvore espalhada mínima, fluxo máximo etc.), que resolvemos neste livro. Mesmo se esse for o caso, reconhecer um tal problema em seu habitat natural — obscurecido pela realidade e o contexto — requer prática e habilidade. É mais provável que você tenha de reduzir seu problema a um destes afortunados — ou resolvê-lo usando programação dinâmica ou programação linear.

Mas há uma chance de que nada disso aconteça. O mundo dos problemas de busca é uma paisagem negra. Existem uns poucos pontos de luz — idéias algorítmicas brilhantes — cada um iluminando uma pequena área em seu redor (os problemas que se reduzem a ele; duas dessas áreas, programação dinâmica e linear são, de fato, bastante grandes). Mas a vasta extensão restante é escura como breu: NP-completa. O que você deve fazer?

Comece provando que seu problema é na verdade NP-completo. Freqüentemente uma prova por generalização (veja a discussão da página 256 e o Exercício 8.10) é tudo de que você precisa; e às vezes uma redução simples partindo de 3SAT ou EZU não é tão difícil de achar. Isso soa como um exercício teórico, mas, se completado com sucesso, traz realmente recompensas tangíveis: agora o seu status no time foi elevado, você não é mais a criança que não sabe resolver nada: tornou-se o nobre cavaleiro com a missão impossível.

Mas, infelizmente, um problema não desaparece quando é demonstrado NP-completo. A questão real é: *o que fazer depois?*

Esse é o assunto deste capítulo e também a inspiração para algumas das pesquisas modernas mais importantes sobre algoritmos e complexidade. NP-completude não é um atestado de óbito — é apenas o começo de uma aventura fascinante.

A prova de NP-completude do seu problema provavelmente constrói grafos complicados e estranhos, muito pouco prováveis de aparecer na sua aplicação. Por exemplo, muito embora SAT seja NP-completo, atribuições satisfatórias para SAT HORN

(as instâncias de SAT que aparecem em programação lógica) podem ser encontradas eficientemente (veja a Seção 5.3). Ou, suponha que os grafos que aparecem na sua aplicação sejam árvores. Nesse caso, muitos problemas NP-completos, como CONJUNTO INDEPENDENTE, podem ser resolvidos em tempo linear por programação dinâmica (veja a Seção 6.7).

Infelizmente, essa abordagem nem sempre funciona. Por exemplo, sabemos que 3SAT é NP-completo. E o problema do CONJUNTO INDEPENDENTE, junto com muitos outros problemas NP-completos, permanece assim mesmo para grafos planares (grafos que podem ser desenhados no plano sem cruzar arestas). Além disso, muitas vezes você não consegue caracterizar claramente as instâncias que aparecem na sua aplicação. Terá de depender de alguma forma de *busca exponencial inteligente* — procedimentos como *backtracking* e *branch-and-bound* que têm tempo exponencial no pior caso, mas que, com o projeto correto, podem ser muito eficientes nas instâncias típicas que aparecem na sua aplicação. Discutiremos esses métodos na Seção 9.1.

Ou você pode desenvolver um algoritmo para o seu problema de otimização NP-completo que não alcance o ótimo, *mas que também nunca fique muito longe dele*. Por exemplo, na Seção 5.4 vimos que o algoritmo guloso sempre produz uma cobertura de vértice que não é mais do que $\log n$ vezes a cobertura de vértice ótima. Um algoritmo que atinja uma tal garantia é chamado de *algoritmo de aproximação*. Como veremos na Seção 9.2, esses algoritmos são conhecidos para muitos problemas de otimização NP-completos e estão entre os mais inteligentes e sofisticados algoritmos que existem. E a teoria de NP-completude pode novamente ser usada como um guia para essa empreitada, mostrando que, para alguns problemas, há até limites severos para o quanto podem ser aproximados — a menos, claro, que $P = NP$.

Por fim, existem *heurísticas*, algoritmos sem qualquer garantia tanto no tempo de execução quanto no grau de aproximação. Heurísticas se baseiam em engenhosidade, intuição, um bom entendimento da aplicação, experimentação meticolosa, e muitas vezes, idéias vindas da física ou biologia, para atacar um problema. Veremos alguns tipos comuns na Seção 9.3.

9.1 Busca exaustiva inteligente

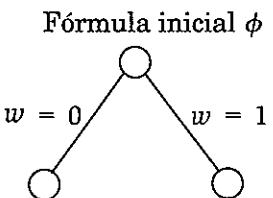
9.1.1 Backtracking

Backtracking é baseado na observação de que é freqüentemente possível rejeitar uma solução examinando apenas uma pequena parte dela. Por exemplo, se uma instância de SAT contém a cláusula $(x_1 \vee x_2)$, todas as atribuições com $x_1 = x_2 = 0$ (ou seja, falso) podem ser instantaneamente eliminadas. Em outras palavras, checando e descartando rapidamente a *atribuição parcial*, somos capazes de podar um quarto de todo o espaço de busca. Uma direção promissora, mas será que pode ser sistematicamente explorada?

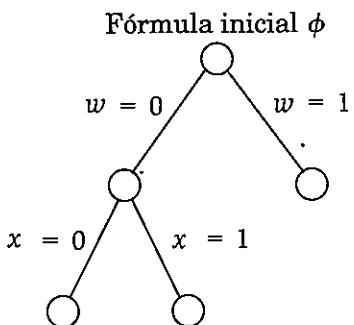
Veja como isso é feito. Considere a fórmula booleana $\phi(w, x, y, z)$ especificada pelo conjunto de cláusulas

$$(w \vee x \vee y \vee z), (w \vee \bar{x}), (x \vee \bar{y}), (y \vee \bar{z}), (z \vee \bar{w}), (\bar{w} \vee \bar{z}).$$

Vamos manter incrementalmente uma árvore de soluções parciais. Começamos ramificando sobre qualquer uma das variáveis, digamos w :



Plugando $w = 0$ e $w = 1$ em ϕ , vemos que nenhuma cláusula é imediatamente violada e, assim, nenhuma dessas duas atribuições parciais pode ser eliminada de pronto. Portanto precisamos continuar ramificando. Podemos expandir qualquer um dos dois nós disponíveis e sobre qualquer variável da nossa escolha. Vamos tentar esta:

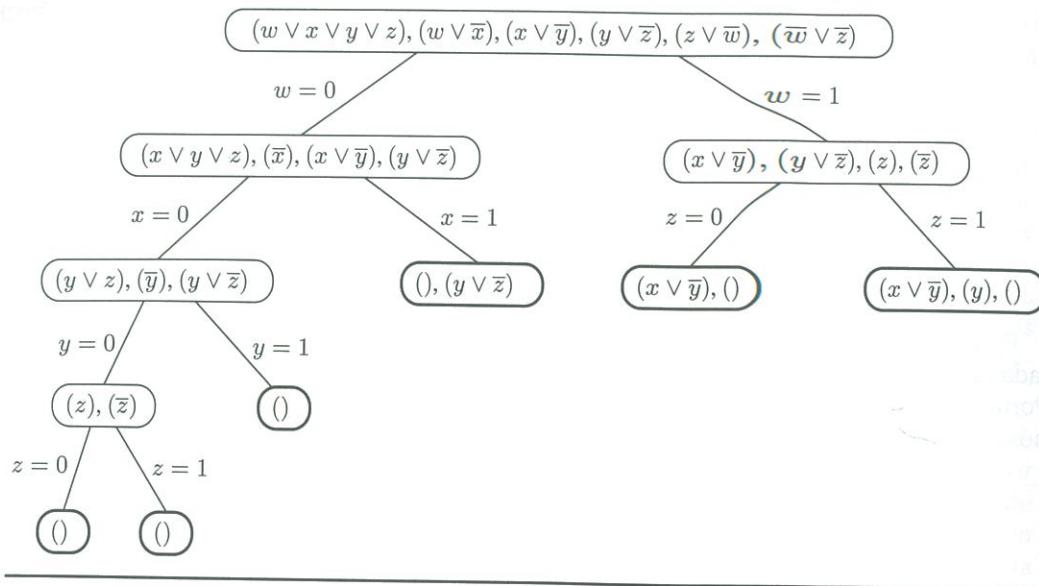


Dessa vez, tivemos sorte. A atribuição parcial $w = 0, x = 1$ viola a cláusula $(w \vee \bar{x})$ e pode, então, ser terminada, limitando com isso um bom pedaço do espaço de busca. Fazemos *backtracking* para fora deste beco sem saída e continuamos nossas explorações em um dos dois nós ativos restantes.

Dessa maneira, *backtracking* explora o espaço de atribuições, crescendo a árvore apenas nos nós onde existe incerteza sobre o resultado e parando se, em qualquer estágio, uma atribuição satisfatória for encontrada.

No caso de satisfatibilidade booleana, cada nó da árvore de busca pode ser descrito ou por uma atribuição parcial ou pelas cláusulas que restam quando aqueles valores são plugados na fórmula original. Por exemplo, se $w = 0$ e $x = 0$, então qualquer cláusula com \bar{w} ou \bar{x} é instantaneamente satisfeita e qualquer literal w ou x não é satisfeito e pode ser removido. O que resta é

$$(y \vee z), (\bar{y}), (y \vee \bar{z}).$$

Figura 9.1 Backtracking revela que ϕ não é satisfatória.

Da mesma maneira, $w = 0$ e $x = 1$ deixa

$$(), (y \vee z),$$

com a “cláusula vazia” $()$ descartando satisfatibilidade. Assim os nós da árvore de busca, representando atribuições parciais, são eles próprios *subproblemas SAT*.

Essa representação alternativa é útil para fazer as duas decisões que aparecem repetidamente: qual subproblema expandir em seguida, e qual variável usar na ramificação. Como o benefício do *backtracking* está na sua habilidade de eliminar porções do espaço de busca, e como isso acontece apenas quando uma cláusula **vazia** é encontrada, faz sentido escolher o subproblema que contenha a *menor cláusula* e, então, ramificar sobre uma variável naquela cláusula. Se essa cláusula acontece de ter apenas um literal, pelo menos um dos ramos resultantes será terminado. (Se existe um empate na escolha do subproblema, uma política razoável é selecionar o mais baixo na árvore, na esperança de que ele esteja próximo de uma atribuição satisfatória.) Veja a Figura 9.1 para a conclusão do nosso exemplo anterior.

De maneira mais abstrata, um algoritmo de *backtracking* requer um teste que olhe para um subproblema e rapidamente declare um de três resultados:

1. Falha: o subproblema não tem solução.
2. Sucesso: uma solução para o subproblema foi encontrada.
3. Incerteza.

No caso de SAT, o teste declara falha se existe uma cláusula **vazia**, sucesso se não há nenhuma cláusula, e incerteza caso contrário. O procedimento de *backtracking* tem o seguinte formato:

Comece com algum problema P_0

Seja $S = \{P_0\}$ o conjunto de subproblemas ativos

Repita enquanto S é não vazio:

escolher um subproblema $P \in S$ e removê-lo de S

expandir P em subproblemas menores P_1, P_2, \dots, P_k

Para cada P_i :

Se teste(P_i) tem sucesso: pare e anuncie esta solução

Se teste(P_i) falha: descarte P_i

Caso contrário: adicione P_i a S

Anuncie que não existe nenhuma solução

Para SAT, o procedimento escolher seleciona uma cláusula e expandir seleciona uma variável dentro daquela cláusula. Já discutimos algumas maneiras razoáveis de fazer tais escolhas.

Com as rotinas teste, expandir e escolher certas, backtracking pode ser memoravelmente efetivo na prática. O algoritmo de backtracking que mostramos para SAT é a base de muitos programas para satisfatibilidade bem-sucedidos. Outro sinal de qualidade é este: se apresentado com uma instância 2SAT, ele vai sempre encontrar uma atribuição satisfatória, se existir uma, em tempo polinomial (Exercício 9.1)!

9.1.2 Branch-and-bound

O mesmo princípio pode ser generalizado de problemas de busca tais como SAT para problemas de otimização. Para sermos concretos, digamos que tenhamos um problema de minimização; maximização seguirá o mesmo padrão.

Como antes, lidaremos com soluções parciais, cada uma das quais representa um *subproblema*, ou seja, qual é (o custo da) a melhor maneira de completar esta solução? E como antes, precisamos de uma base para eliminar soluções parciais, pois não há nenhuma outra fonte de eficiência no nosso método. Para rejeitarmos um subproblema, temos de ter certeza de que seu custo excede o de alguma outra solução que já encontramos. Mas seu custo exato é desconhecido para nós e em geral não pode ser computado eficientemente. Portanto, como alternativa, usamos uma *cota inferior* rápida sobre este custo.

Comece com algum problema P_0

Seja $S = \{P_0\}$ o conjunto de subproblemas ativos

$\text{melhor} = \infty$

Repita enquanto S é não vazio:

escolher um subproblema $P \in S$ e removê-lo de S

expandir P em subproblemas menores P_1, P_2, \dots, P_k

Para cada P_i :

Se P_i é uma solução completa: atualize o melhor

senão, se cota inferior(P_i) < melhor: adicionar P_i a S

retornar melhor

Vejamos como isso funciona para o problema do caixeiro-viajante sobre um grafo $G = (V, E)$ com comprimentos de aresta $d_e > 0$. Uma solução parcial é um caminho

simples $a \rightsquigarrow b$ passando por alguns vértices $S \subseteq V$, onde S inclui as extremidades a e b . Podemos denotar uma tal solução parcial por uma tupla $[a, S, b]$ — de fato, a será fixo por todo o algoritmo. O subproblema correspondente é encontrar a melhor maneira de completar o circuito, ou seja, o caminho complementar mais barato $b \rightsquigarrow a$ com nós intermediários $V - S$. Note que o problema inicial é da forma $[a, \{a\}, a]$ para qualquer $a \in V$ de nossa escolha.

Em cada passo do algoritmo de *branch-and-bound*, estendemos uma particular solução $[a, S, b]$ por uma única aresta (b, x) , onde $x \in V - S$. Pode haver até $|V - S|$ maneiras de fazer isso, e cada um desses ramos leva a um subproblema da forma $[a, S \cup \{x\}, x]$.

Como podemos dar uma cota inferior para o custo de completar um circuito parcial $[a, S, b]$? Muitos métodos sofisticados foram desenvolvidos para isso, mas examinemos um bastante simples. O restante do circuito consiste em um caminho por $V - S$, mais arestas de a e b para $V - S$. Portanto, seu custo é pelo menos a soma do seguinte:

1. A aresta mais leve de a para $V - S$.
2. A aresta mais leve de b para $V - S$.
3. A árvore espalhada mínima de $V - S$.

(Você entende por quê?) Esta cota inferior pode ser computada rapidamente por um algoritmo para árvore espalhada mínima. A Figura 9.2 ilustra um exemplo: cada nó da árvore representa um circuito parcial (especificamente, o caminho da raiz para aquele nó) que em algum estágio é considerado pelo procedimento de *branch-and-bound*. Note como apenas 28 soluções parciais são consideradas, em vez de $7! = 5.040$ que apareceriam em uma busca por força-bruta.

9.2 Algoritmos de aproximação

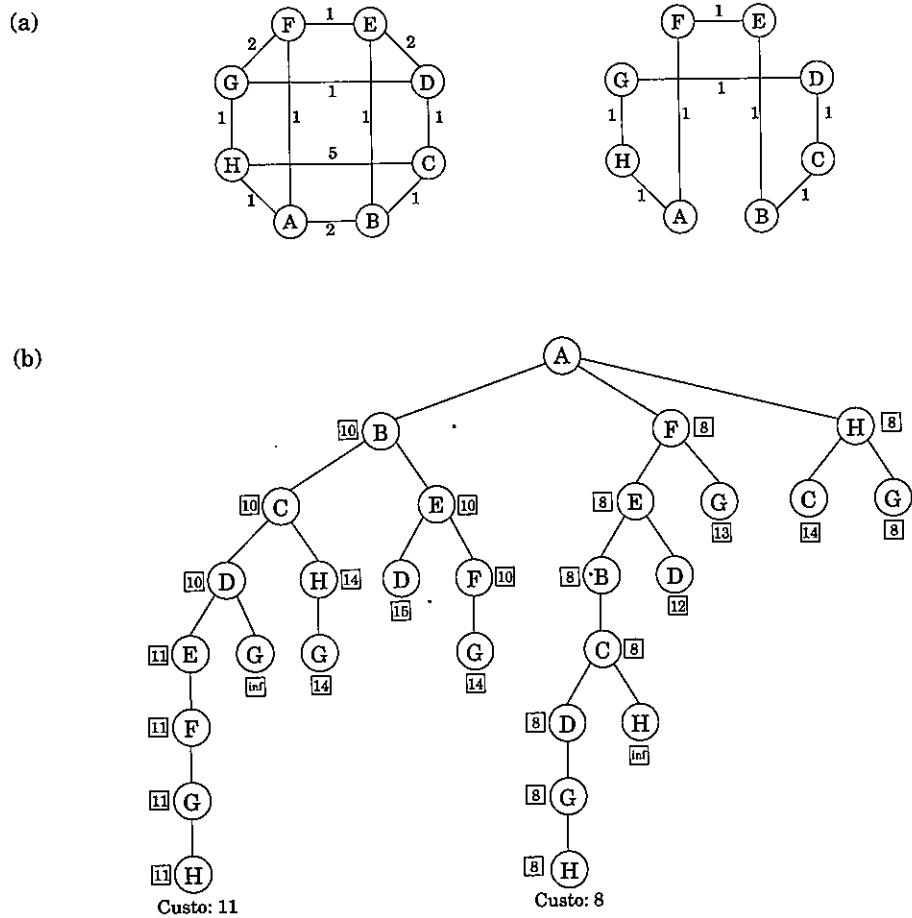
Em um problema de otimização é dada uma instância I e precisamos encontrar a solução ótima — aquela com o ganho máximo se temos um problema de maximização como CONJUNTO INDEPENDENTE, ou o custo mínimo se estamos lidando com um problema de minimização tal como TSP. Para cada instância I , vamos denotar por $\text{OPT}(I)$ o valor (benefício ou custo) da solução ótima. A matemática fica um pouco mais simples se *considerarmos que $\text{OPT}(I)$ é sempre um inteiro positivo* (e não está muito longe da verdade).

Já vimos um exemplo de um (famoso) algoritmo de aproximação na Seção 5.4: o esquema guloso para COBERTURA DE VÉRTICE. Para toda instância I de tamanho n , mostramos que este algoritmo guloso, com certeza, encontra rapidamente uma cobertura de vértice de cardinalidade no máximo $\text{OPT}(I) \log n$. Esse fator $\log n$ é conhecido como a garantia de aproximação do algoritmo.

De maneira mais geral, considere qualquer problema de minimização. Suponha agora que tenhamos um algoritmo \mathcal{A} para nosso problema que, dada uma instância I , retorna uma solução com valor $\mathcal{A}(I)$. A razão de aproximação do algoritmo \mathcal{A} é definida como

$$\alpha_{\mathcal{A}} = \max_I \frac{\mathcal{A}(I)}{\text{OPT}(I)}.$$

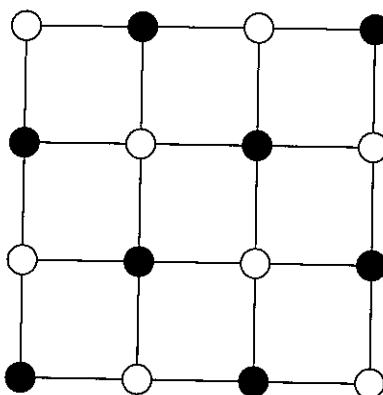
Figura 9.2 (a) Um grafo e seu circuito ótimo para o caixeiro-viajante. (b) A árvore de busca do *branch-and-bound*, explorada da esquerda para a direita. Os números nos quadrados indicam cotas inferiores sobre o custo.



Em outras palavras, $\alpha_{\mathcal{A}}$ mede o fator pelo qual a saída do algoritmo \mathcal{A} excede a solução ótima, na entrada de pior caso. A razão de aproximação também pode ser definida para problemas de maximização, tais como CONJUNTO INDEPENDENTE, da mesma maneira — exceto que para obtermos um número maior do que 1, tomamos o recíproco.

Portanto, quando confrontados com um problema de otimização NP-completo, um objetivo razoável é procurar por um algoritmo de aproximação \mathcal{A} cuja $\alpha_{\mathcal{A}}$ seja tão pequena quanto possível. Mas esse tipo de garantia pode parecer um pouco misterioso: como podemos chegar perto do ótimo se não podemos determinar o ótimo? Vamos ver um exemplo simples.

Figura 9.3 Um grafo cuja cobertura de vértice ótima, mostrada sombreada, é de tamanho 8.



9.2.1 Cobertura de vértice

Sabemos que o problema da COBERTURA DE VÉRTICE é **NP-difícil**.

Cobertura de Vértice

Entrada: Um grafo não-direcionado $G = (V, E)$.

Saída: Um subconjunto de vértices $S \subseteq V$ que toca todas as arestas.

Objetivo: Minimizar $|S|$.

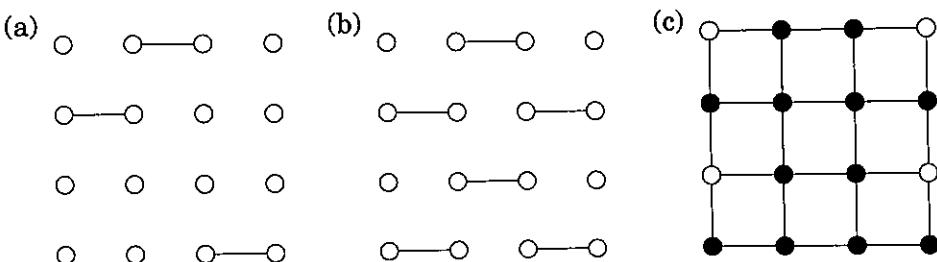
Veja a Figura 9.3 para um exemplo.

Como COBERTURA DE VÉRTICE é um caso especial de COBERTURA DE CONJUNTO, sabemos do Capítulo 5 que ele pode ser aproximado dentro de um fator de $O(\log n)$ pelo algoritmo guloso: remova repetidamente o vértice de maior grau e inclua-o na cobertura de vértice. E existem grafos sobre os quais o algoritmo guloso retorna uma cobertura de vértice que é de fato $\log n$ vezes o ótimo.

Um algoritmo de aproximação melhor para COBERTURA DE VÉRTICE é baseado na noção de *emparelhamento*, um subconjunto de arestas que não têm nenhum vértice em comum (Figura 9.4). Um emparelhamento é *maximal* se nenhuma nova aresta pode ser adicionada a ele. Emparelhamentos maximais vão nos ajudar a encontrar boas coberturas de vértice, além disso eles são fáceis de gerar: repetidamente selecionam arestas disjuntas daquelas já escolhidas, até que isso não seja mais possível.

Qual a relação entre emparelhamento e cobertura de vértice? Aqui está o fato crucial: qualquer cobertura de vértice de um grafo G tem de ser pelo menos tão grande quanto o número de arestas em qualquer emparelhamento em G ; ou seja, *qualquer emparelhamento fornece uma cota inferior sobre OPT*. Isso é simplesmente porque cada aresta do emparelhamento tem de estar coberta por uma de suas extremidades em uma cobertura de vértice! Encontrar uma cota inferior é o passo-chave ao projetarmos um algoritmo de aproximação, porque temos de comparar a qualidade da solução encontrada por nosso algoritmo com OPT , cuja computação é **NP-completa**.

Figura 9.4 (a) Um emparelhamento, (b) sua extensão a um emparelhamento maximal e (c) a cobertura de vértice resultante.



Mais uma observação completa o projeto do nosso algoritmo de aproximação: seja S um conjunto que contenha ambas as extremidades de cada aresta em um emparelhamento maximal M . Então S tem de ser uma cobertura de vértice — do contrário, ou seja, se não tocar alguma aresta $e \in E$, então M não poderá ser maximal, pois poderemos ainda adicionar e a ele. Mas nossa cobertura S tem $2|M|$ vértices. E do parágrafo anterior sabemos que qualquer cobertura de vértice tem de ter tamanho pelo menos $|M|$. Portanto, tudo está feito.

Veja o algoritmo para COBERTURA DE VÉRTICE.

Encontrar um emparelhamento maximal $M \subseteq E$

Retornar $S = \{\text{todas as extremidades de arestas em } M\}$

Esse procedimento simples sempre retorna uma cobertura de vértice cujo tamanho é no máximo duas vezes o ótimo!

Resumindo, muito embora não tenhamos nenhuma maneira de encontrar a melhor cobertura de vértice, podemos facilmente encontrar outra estrutura, um emparelhamento maximal, com duas propriedades-chave:

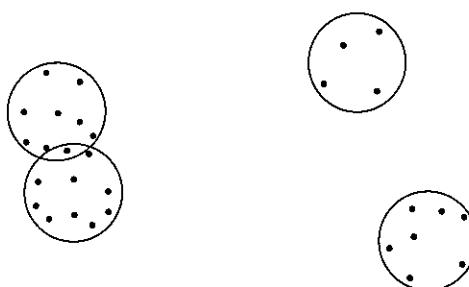
1. Seu tamanho nos dá uma cota inferior sobre a cobertura de vértice ótima.
2. Ele pode ser usado para construir uma cobertura de vértice, cujo tamanho pode ser relacionado com o da cobertura ótima usando a propriedade 1.

Assim, esse algoritmo simples tem uma razão de aproximação de $\alpha_A \leq 2$. De fato, não é difícil encontrar exemplos nos quais ele gera um erro de 100%; assim $\alpha_A = 2$.

9.2.2 Agrupamento

Abordaremos agora um problema de *agrupamento*, no qual temos alguns dados (documentos de texto, digamos, ou imagens, ou amostras de voz) que queremos dividir em grupos. É muitas vezes útil definir “distâncias” entre esses pontos de dados, números que capturam quanto perto ou longe eles estão um do outro. Freqüentemente os dados são pontos reais em algum espaço de alta dimensão e as distâncias são a distância euclidiana usual; em outros casos, as distâncias são o resultado de alguns “testes de similaridade” aos quais submetemos os pontos de dados. Considere que temos tais distâncias e que elas satisfaçam às propriedades métricas usuais:

1. $d(x, y) \geq 0$ para todo x, y .

Figura 9.5 Alguns pontos de dados e os $k = 4$ grupos ótimos.

2. $d(x, y) = 0$ se e somente se $x = y$.
3. $d(x, y) = d(y, x)$.
4. (Desigualdade triangular) $d(x, y) \leq d(x, z) + d(z, y)$.

Gostaríamos de partitionar os pontos de dados em grupos que sejam compactos no sentido de terem diâmetro pequeno.

k-grupo

Entrada: Pontos $X = \{x_1, \dots, x_n\}$ com distância métrica subjacente $d(\cdot, \cdot)$; inteiro k .

Saída: Uma partição dos pontos em k grupos C_1, \dots, C_k .

Objetivo: Minimizar o diâmetro dos grupos,

$$\max_j \max_{x_a, x_b \in C_j} d(x_a, x_b).$$

Uma maneira de visualizar essa tarefa é imaginar n pontos no espaço, que devem ser cobertos por k esferas de tamanho igual. Qual o menor diâmetro possível das esferas? A Figura 9.5 mostra um exemplo.

Esse problema é NP-difícil, mas tem um algoritmo muito simples. A idéia é selecionar k dos pontos de dados como *centros de grupos* e, então, atribuir cada um dos pontos restantes ao centro mais próximo dele, criando assim k grupos. Os centros são selecionados um por vez, usando uma regra intuitiva: sempre selecione o próximo centro o mais longe possível dos centros já escolhidos (veja a Figura 9.6).

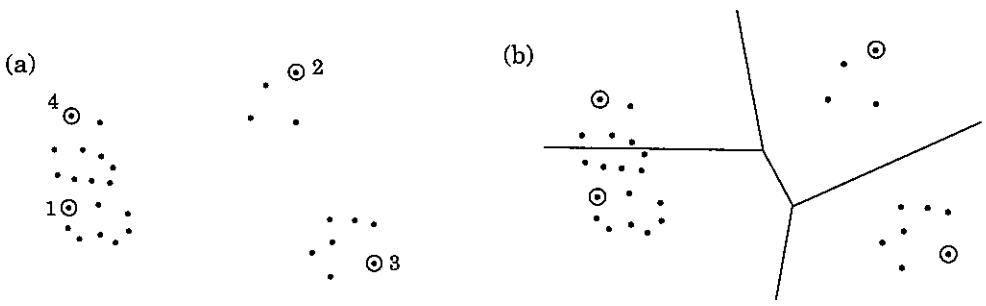
Selecionar qualquer ponto $\mu_1 \in X$ como o primeiro centro de grupo para $i = 2$ até k :

Seja μ_i o ponto em X mais longe de μ_1, \dots, μ_{i-1}
(isto é, que maximiza $\min_{j < i} d(\cdot, \mu_j)$)

Criar k grupos: $C_i = \{\text{todos os } x \in X \text{ cujo centro mais próximo é } \mu_i\}$

Está claro que esse algoritmo retorna uma partição válida. Além disso, o diâmetro resultante é, garantidamente, no máximo duas vezes o ótimo.

Figura 9.6 (a) Quatro centros escolhidos por um percurso que toma primeiro quem está mais longe. (b) Os grupos resultantes.



Veja este argumento: seja $x \in X$ o ponto mais distante de μ_1, \dots, μ_k (em outras palavras, o próximo centro que escolheríamos, se quiséssemos $k + 1$ centros), e seja r sua distância para o centro mais próximo. Então todos os pontos em X têm de estar à distância no máximo r do centro do seu grupo. Pela desigualdade triangular, isso significa que todo grupo tem diâmetro de no máximo $2r$.

Mas como o r se relaciona com o diâmetro do agrupamento ótimo? Bem, identificamos $k + 1$ pontos $\{\mu_1, \mu_2, \dots, \mu_k, x\}$ que estão todos a uma distância de pelo menos r um do outro (por quê?). Qualquer partição em k grupos tem de colocar dois destes pontos no mesmo grupo e, portanto, tem de ter diâmetro de pelo menos r .

Esse algoritmo tem uma certa similaridade de alto nível com o nosso esquema para COBERTURA DE VÉRTICE. Em vez de um emparelhamento maximal, usamos uma estrutura fácil de ser computada diferente — um conjunto de k pontos que cobrem todo o X por um raio r , estando ao mesmo tempo mutuamente separados por uma distância de pelo menos r . Essa estrutura é usada tanto para gerar um agrupamento quanto para dar uma cota inferior para o agrupamento ótimo.

Não se conhece nenhum algoritmo de aproximação melhor para este problema.

9.2.3 TSP

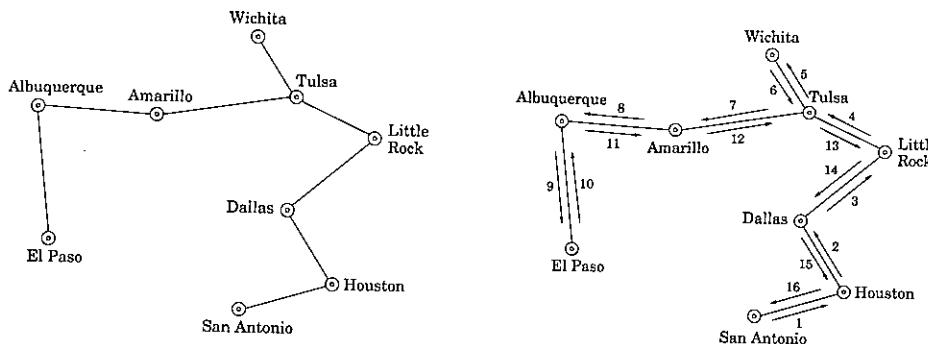
A desigualdade triangular teve um papel crucial ao mostrarmos que o problema k -GRUPO é aproximável. Ela também ajuda com o PROBLEMA DO CAIXEIRO-VIAJANTE: se as distâncias entre as cidades satisfazem propriedades métricas, existe um algoritmo que apresenta um circuito de comprimento no máximo 1,5 vez o ótimo. Vejamos agora um resultado ligeiramente mais fraco que alcança um fator de 2.

Continuando com os processos de raciocínio dos nossos dois algoritmos de aproximação anteriores, podemos perguntar se existe alguma estrutura que seja fácil de computar e que esteja plausivelmente relacionada com o melhor circuito para o caixeiro-viajante (além de fornecer uma boa cota inferior sobre OPT). Um pouco de raciocínio e experimentação revelam que a resposta é a *árvore espalhada mínima*.

Vamos entender esta relação. Remover qualquer aresta de um circuito do caixeiro-viajante deixa um caminho por todos os vértices, que é uma árvore espalhada. Portanto,

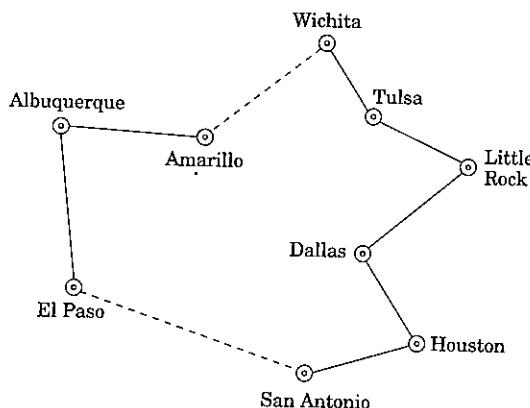
Custo do TSP \geq custo deste caminho \geq custo da AEM.

Agora, precisamos de alguma maneira usar a AEM para construir um circuito do caixeiro-viajante. Se pudéssemos usar cada aresta *duas vezes*, seguindo o contorno da AEM terminaríamos com um circuito que visita todas as cidades, algumas delas mais de uma vez. Aqui está um exemplo, com a AEM na parte esquerda e o circuito resultante na direita (os números mostram a ordem na qual as arestas são tomadas).



Portanto, esse circuito tem tamanho no máximo duas vezes o custo da AEM, que como já vimos é no máximo duas vezes o custo do TSP.

Esse é o resultado que desejávamos, mas ainda não está pronto porque o nosso circuito visita algumas cidades múltiplas vezes e, portanto, não é legal. Para consertar esse problema, o circuito deve simplesmente pular qualquer cidade que esteja prestes a revisitar e, ao contrário, ir diretamente para a próxima *nova* cidade na sua lista:



Pela desigualdade triangular, esses atalhos só podem tornar o circuito total menor.

TSP Geral

Mas o que dizer se estivermos interessados em instâncias do TSP que não satisfazem à desigualdade triangular? O fato é que esse é um problema *muito mais* difícil de aproximar.

Aqui está a razão: na página 260 apresentamos uma redução de tempo polinomial na qual dado qualquer grafo G e qualquer inteiro $C > 0$ produz uma instância $I(G, C)$ do TSP tal que:

- (i) Se G tem um caminho Rudrata, então $\text{OPT}(I(G, C)) = n$, o número de vértices em G .
- (ii) Se G não tem nenhum caminho Rudrata, então $\text{OPT}(I(G, C)) \geq n + C$.

Isso significa que mesmo uma solução aproximada do TSP nos possibilitaria resolver o CAMINHO RUDRATA! Elaboremos os detalhes.

Considere um algoritmo de aproximação \mathcal{A} para TSP e denote por $\alpha_{\mathcal{A}}$ a sua razão de aproximação. De qualquer instância G do CAMINHO RUDRATA, vamos criar uma instância $I(G, C)$ do TSP usando a constante específica $C = n\alpha_{\mathcal{A}}$. O que acontece quando o algoritmo \mathcal{A} é executado sobre essa instância do TSP? No caso (i), ele tem de apresentar um circuito de tamanho no máximo $\alpha_{\mathcal{A}}\text{OPT}(I(G, C)) = n\alpha_{\mathcal{A}}$, enquanto no caso (ii) tem de apresentar um circuito de tamanho pelo menos $\text{OPT}(I(G, C)) > n\alpha_{\mathcal{A}}$. Assim, podemos descobrir se G tem um caminho Rudrata! Veja o procedimento resultante:

Dado qualquer grafo G :

```
computar  $I(G, C)$  (com  $C = n \cdot \alpha_{\mathcal{A}}$ ) e executar o algoritmo  $\mathcal{A}$ 
sobre ela
se o circuito resultante tem comprimento  $\leq n\alpha_{\mathcal{A}}$ :
    concluir que  $G$  tem um caminho Rudrata
senão: concluir que  $G$  não tem nenhum caminho Rudrata
```

Isso nos diz se G tem ou não um caminho Rudrata; chamando o procedimento um número polinomial de vezes, podemos encontrar o caminho propriamente dito (Exercício 8.2).

Mostramos que se TSP tiver um algoritmo de aproximação de tempo polinomial, então existe um algoritmo polinomial para o problema NP-completo CAMINHO RUDRATA. Assim, a menos que $P = NP$, não pode existir um algoritmo de aproximação eficiente para o TSP.

9.2.4 Mochila

Nosso último algoritmo de aproximação é para um problema de maximização e tem uma garantia muito impressionante: dado qualquer $\epsilon > 0$, ele vai retornar uma solução de valor pelo menos $(1 - \epsilon)$ vezes o valor ótimo, em um tempo que cresce apenas polynomialmente no tamanho da entrada e em $1/\epsilon$.

O problema é a MOCHILA, que encontramos pela primeira vez no Capítulo 6. Existem n itens, com pesos w_1, \dots, w_n e valores v_1, \dots, v_n (todos inteiros positivos), e o objetivo é selecionar a combinação mais valiosa de itens sujeita à restrição de que o peso total deles seja no máximo W .

Anteriormente vimos uma solução com programação dinâmica para este problema com tempo de execução $O(nW)$. Usando uma técnica similar, um tempo de execução de $O(nV)$ também pode ser alcançado, onde V é a soma dos valores. Nenhum desses tempos de execução é polinomial, porque W e V podem ser muito grandes, exponenciais no tamanho da entrada.

Consideremos o algoritmo $O(nV)$. No caso ruim quando V é grande, o que dizer se simplesmente diminuirmos em escala os valores de alguma forma? Por exemplo, se

$$v_1 = 117.586.003, v_2 = 738.493.291, v_3 = 238.827.453,$$

poderíamos simplesmente descartar alguma precisão e, ao contrário, usar 117, 738 e 238. Isto não muda tanto o problema e vamos tornar o algoritmo muito, muito mais rápido!

Agora para os detalhes. Junto com a entrada, consideraremos que o usuário especificou algum fator de aproximação $\epsilon > 0$.

Descartar qualquer item com peso $> W$

Seja $v_{\max} = \max_i v_i$

Reescalar os valores $\hat{v}_i = \left\lfloor v_i \cdot \frac{n}{\epsilon v_{\max}} \right\rfloor$

Executar o algoritmo de programação dinâmica com valores $\{\hat{v}_i\}$
Retornar a escolha resultante de itens

Vamos ver por que isso funciona. Em primeiro lugar, como os valores reescalados \hat{v}_i são todos no máximo n/ϵ , o programa dinâmico é eficiente, rodando em tempo $O(n^3/\epsilon)$.

Agora suponha que a solução ótima para o problema original seja selecionar algum subconjunto de itens S , com valor total K^* . Os valores reescalados dessa mesma atribuição são

$$\sum_{i \in S} \hat{v}_i = \sum_{i \in S} \left\lfloor v_i \cdot \frac{n}{\epsilon v_{\max}} \right\rfloor \geq \sum_{i \in S} \left(v_i \cdot \frac{n}{\epsilon v_{\max}} - 1 \right) \geq K^* \cdot \frac{n}{\epsilon v_{\max}} - n.$$

Portanto, a atribuição ótima para o problema reduzido, chame-o de \widehat{S} , tem um valor reescalado de pelo menos esse tanto. Em termos dos valores originais, a atribuição \widehat{S} tem um valor de pelo menos

$$\sum_{i \in \widehat{S}} v_i \geq \sum_{i \in \widehat{S}} \hat{v}_i \cdot \frac{\epsilon v_{\max}}{n} \geq \left(K^* \cdot \frac{n}{\epsilon v_{\max}} - n \right) \cdot \frac{\epsilon v_{\max}}{n} = K^* - \epsilon v_{\max} \geq K^*(1 - \epsilon).$$

9.2.5 A hierarquia de approximabilidade

Dado qualquer problema de otimização NP-completo, buscamos o melhor algoritmo de aproximação possível. Se isso falhar, tentamos provar *cotas inferiores* sobre as razões de aproximação que podem ser alcançadas em tempo polinomial (acabamos de dar uma tal prova para o TSP geral). Dito tudo isso, problemas de otimização NP-completos são classificados como se segue:

- Aqueles para os quais, como o TSP, nenhuma razão de aproximação finita é possível.
- Aqueles para os quais uma razão de aproximação é possível, mas há limites para o quanto pequena ela pode ser. COBERTURA DE VÉRTICE, k -GRUPOS e o TSP com desigualdade triangular pertencem a essa classe. (Para esses problemas não estabelecemos limites para a sua aproximação, mas os limites de fato existem, e suas provas constituem alguns dos mais sofisticados resultados da área.)
- Abaixo disso temos uma classe mais afortunada de problemas NP-completos para os quais a aproximação não tem nenhum limite, e algoritmos de aproximação polinomiais com razões de erro arbitrariamente próximas de zero existem. A MOCHILA está aqui.

- Por fim, há uma outra classe de problemas, entre a primeira e a segunda apreendidas aqui, para a qual a razão de aproximação é cerca de $\log n$. COBERTURA DE VÉRTICE é um exemplo.

(Um lembrete: tudo isso é dependente da hipótese de que $P \neq NP$. Se isso falha, esta hierarquia colapsa em P e todos os problemas de otimização NP-completos podem ser resolvidos exatamente em tempo polinomial.)

Um ponto final em algoritmos de aproximação: muitas vezes esses algoritmos, ou suas variantes, têm uma performance muito melhor em instâncias típicas do que as suas razões de aproximação nos poderiam fazer acreditar.

9.3 Heurísticas de busca local

Nossa próxima estratégia para lidar com NP-completude é inspirada pela evolução (que é, afinal, o procedimento de otimização mais testado do mundo) — por seu processo incremental de introduzir pequenas mutações, testá-las, e mantê-las se elas funcionarem bem. Esse paradigma é chamado *busca local* e pode ser aplicado a qualquer tarefa de otimização. Veja como ela se parece a um problema de minimização.

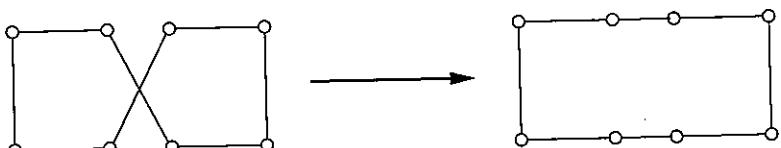
```
Seja  $s$  qualquer solução inicial
Enquanto existir alguma solução  $s'$  na vizinhança de  $s$ 
    para a qual  $\text{custo}(s') < \text{custo}(s)$ : substitua  $s$  por  $s'$ 
    retornar  $s$ 
```

Em cada iteração, a solução atual é substituída por uma melhor próxima a ela, em sua *vizinhança*. A estrutura de vizinhança é algo que impomos ao problema e é a decisão central de projeto em busca local. Como uma ilustração, revisitemos o problema do caixeiro-viajante.

9.3.1 Caixeiro-viajante, uma vez mais

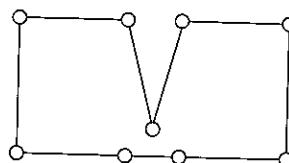
Consideremos que tenhamos todas as distâncias entre pares de n cidades, levando a um espaço de busca de $(n - 1)!$ circuitos diferentes. O que é um bom conceito de vizinhança?

A noção mais óbvia é considerar dois circuitos como próximos se eles diferirem em apenas algumas poucas arestas. Eles não podem diferir em apenas uma aresta (você entende por quê?), assim vamos considerar diferenças de duas arestas. Definimos uma vizinhança de *mudança-2* do circuito s como o conjunto de todos os circuitos que podem ser obtidos removendo duas arestas de s e, então, colocando duas outras arestas. Veja um exemplo de movimento local:

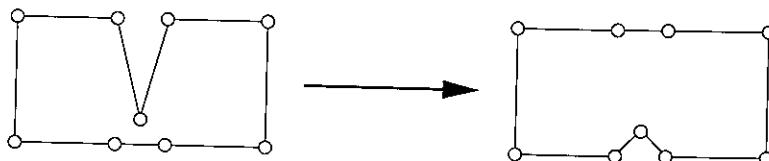


Agora temos um procedimento de busca local bem-definido. Como ele se classifica segundo os nossos dois critérios padrão para algoritmos — qual o seu tempo total de execução e será que ele sempre retorna a melhor solução?

Embaraçosamente, nenhuma dessas questões tem uma resposta satisfatória. Cada iteração é certamente rápida, porque um circuito tem apenas $O(n^2)$ vizinhos. Entretanto, não está claro quantas iterações serão necessárias: se, por exemplo, poderia haver um número exponencial delas. Da mesma maneira, tudo o que podemos facilmente dizer sobre o circuito final é que ele é *localmente ótimo* — isto é, ele é superior aos circuitos da sua vizinhança imediata. Pode haver soluções melhores um pouco mais longe. Por exemplo, a seguinte figura mostra uma possível resposta final que é claramente subótima; o conjunto de movimentos locais é simplesmente muito limitado para aperfeiçoá-la.



Para superarmos isso, podemos tentar uma vizinhança mais generosa, por exemplo uma *mudança-3*, consistindo em circuitos que diferem em até três arestas. E, de fato, o caso ruim anterior é resolvido:



Mas há uma desvantagem, a de que o tamanho de uma vizinhança torna-se $O(n^3)$, fazendo cada iteração mais dispendiosa. Além disso, pode ainda haver mínimos locais subótimos, embora menos do que antes. Para evitar isso, teríamos de subir para *mudança-4*, ou mais alto. Dessa maneira, eficiência e qualidade freqüentemente se revelam considerações competidoras em uma busca local. Eficiência demanda vizinhanças que possam ser buscadas rapidamente, mas vizinhanças menores podem aumentar a quantidade de ótimos locais de qualidade baixa. O compromisso apropriado é tipicamente determinado por experimentação.

A Figura 9.7 mostra um exemplo específico de busca local em funcionamento. A Figura 9.8 é uma ilustração estilizada, mais abstrata, de busca local. As soluções povoam a área não sombreada e o custo decresce quando nos movemos para baixo. Começando com uma solução inicial, o algoritmo move-se montanha abaixo até que um ótimo local seja alcançado.

Em geral, o espaço de busca pode estar crivado de ótimos locais e alguns deles podem ser de qualidade muito pobre. A esperança é que com uma escolha judiciosa de estrutura de vizinhança, a maioria dos ótimos locais sejam razoáveis. Seja isto a realidade ou

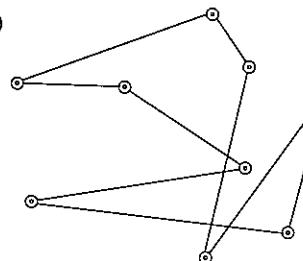
Figura 9.7 (a) Nove cidades americanas. (b) Busca local, começando em um circuito aleatório e usando mudança-3. O circuito do caixeiro-viajante é encontrado depois de três movimentos.

(a)

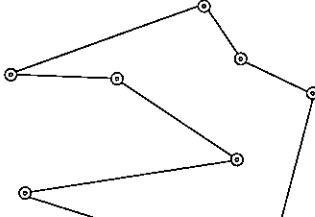


(b)

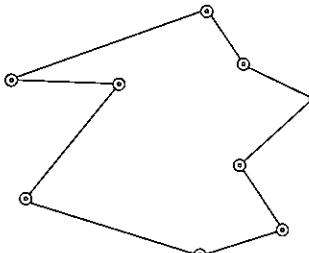
(i)



(ii)



(iii)



(iv)

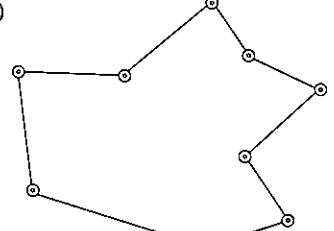
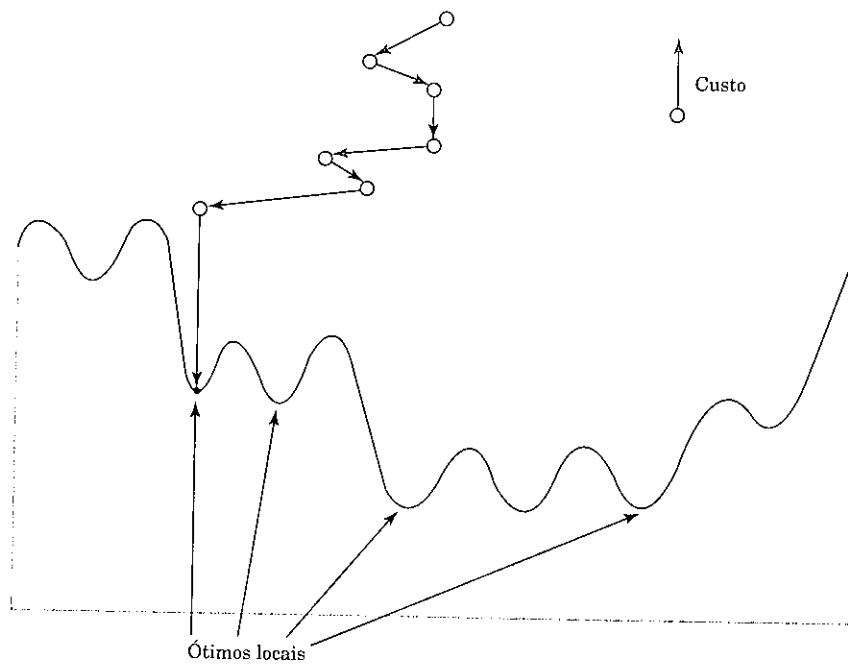


Figura 9.8 Busca local.

meramente crença injustificada, o fato empírico é que algoritmos de busca local são os de melhor performance em uma ampla gama de problemas de otimização. Vejamos outro exemplo.

9.3.2 Particionamento de grafo

O problema de particionamento de grafos aparece em uma diversidade de aplicações, de layout de circuito à análise de programas e segmentação de imagens. Vimos um caso especial dele, CORTE BALANCEADO, no Capítulo 8.

Particionamento de grafo

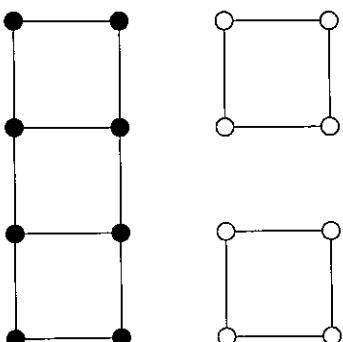
Entrada: Um grafo não-direcionado $G = (V, E)$ com pesos de aresta não-negativos; um número real $\alpha \in [0, 1/2]$.

Saída: Uma partição dos vértices em dois grupos A e B , cada um com tamanho de pelo menos $\alpha|V|$.

Objetivo: Minimizar a capacidade do corte (A, B) .

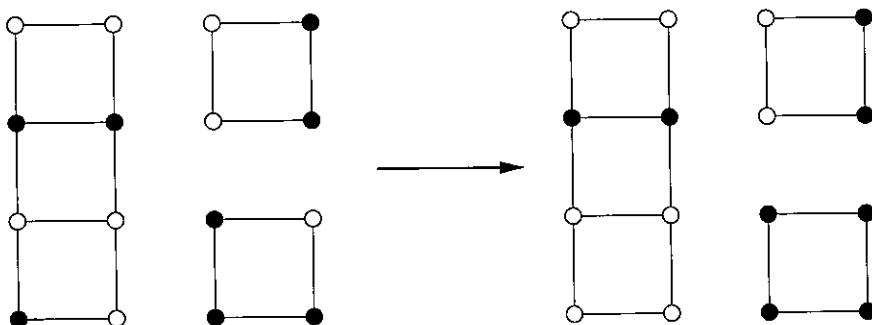
A Figura 9.9 mostra um exemplo no qual o grafo tem 16 nós, todos os pesos de aresta são 0 ou 1, e a solução ótima tem custo 0. Remover a restrição sobre os tamanhos de A e B levaria ao problema do CORTE MÍNIMO, que sabemos ser solúvel eficientemente

Figura 9.9 Uma instância de PARTICIONAMENTO DE GRAFO, com a partição ótima para $\alpha = 1/2$. Vértices de um lado do corte estão sombreados.

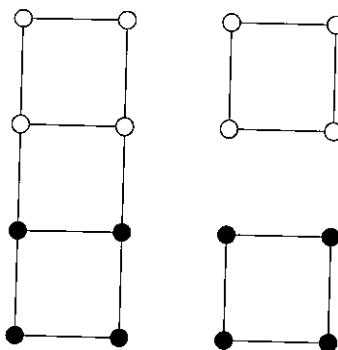


usando de técnicas de fluxo. A presente variante, entretanto, é NP-difícil. No projeto de algoritmo de busca local, será uma grande conveniência focar no caso especial $\alpha = 1/2$, no qual A e B são forçados a conter exatamente metade dos vértices. A perda de generalidade aparente é puramente cosmética, pois PARTICIONAMENTO DE GRAFO se reduz a este caso especial.

Precisamos decidir sobre uma estrutura de vizinhança para nosso problema, e existe uma maneira óbvia de fazer isso. Seja (A, B) , com $|A| = |B|$, uma solução candidata; vamos definir seus vizinhos como todas as soluções que podem ser obtidas trocando-se um par de vértices através do corte, ou seja, todas as soluções da forma $(A - \{a\} + \{b\}, B - \{b\} + \{a\})$ onde $a \in A$ e $b \in B$. Veja um exemplo de movimento local:



Agora temos um procedimento razoável de busca local e poderíamos simplesmente parar por aqui. Mas há ainda muito espaço para melhoria em termos da *qualidade* das soluções produzidas. O espaço de busca inclui alguns ótimos locais que estão bem longe da solução global. Veja um que tem custo 2.



O que pode ser feito a respeito dessas soluções subótimas? Poderíamos expandir o tamanho das suas vizinhanças para permitir duas substituições por vez, mas esta particular instância ruim teimosamente ainda resistiria. Em vez disso, vamos olhar alguns outros esquemas genéricos para aperfeiçoar procedimentos de busca local.

9.3.3 Lidando com ótimos locais

Randomização e recomeço

Randomização pode ser um aliado valioso em busca local. Ela é tipicamente usada de duas maneiras: para selecionar uma solução inicial aleatória, por exemplo, uma partição de grafo aleatória; e para escolher um movimento local quando vários estão disponíveis.

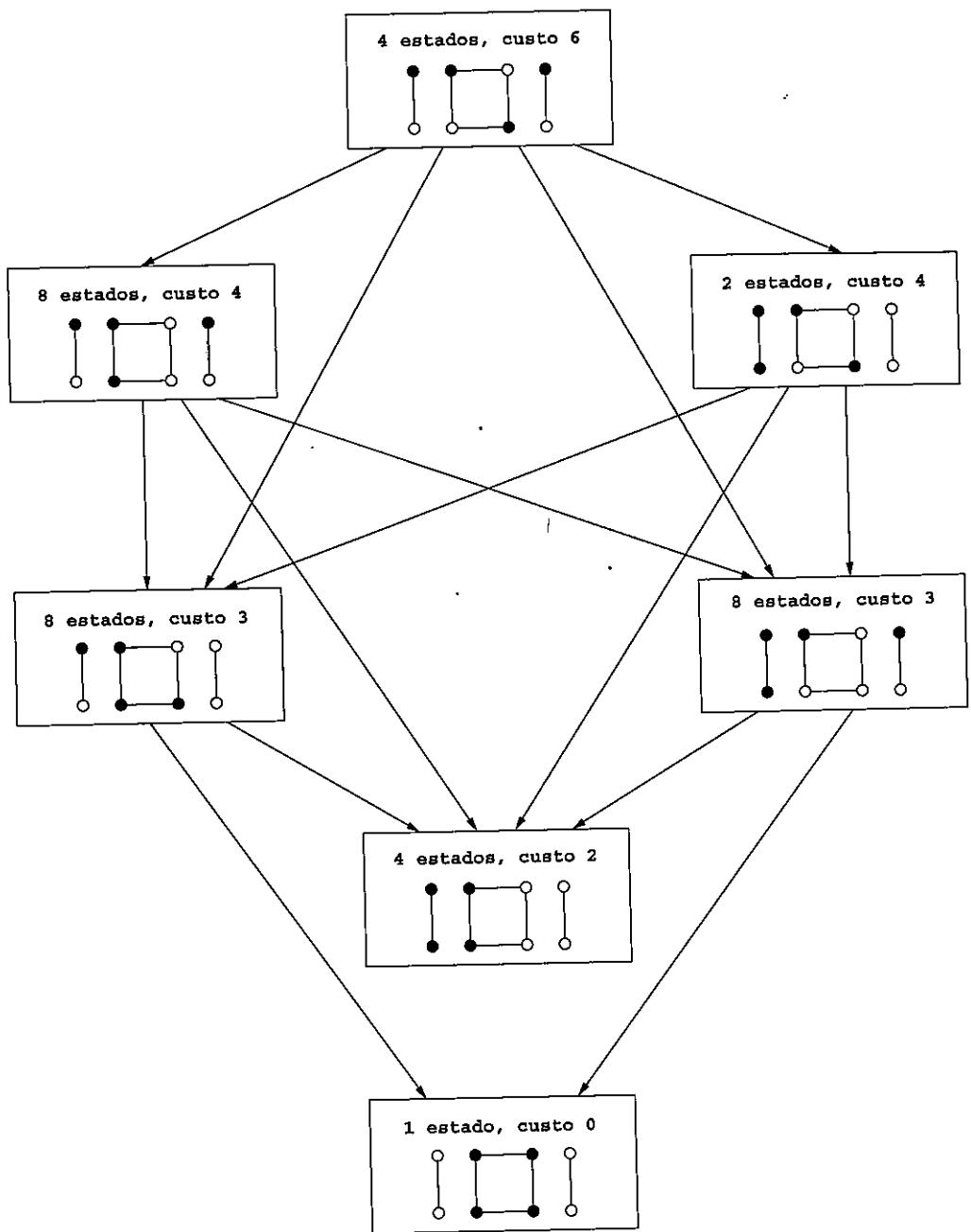
Quando há muitos ótimos locais, randomização é uma maneira de assegurar que pelo menos existe alguma probabilidade de obter a correta. A busca local pode, então, ser repetida várias vezes, com uma semente aleatória diferente em cada invocação, e retornar a melhor solução. Se a probabilidade de alcançar um bom ótimo local em qualquer dada execução é p , então dentro de $O(1/p)$ execuções uma solução provavelmente será encontrada (reveja o Exercício 1.34).

A Figura 9.10 mostra uma pequena instância de particionamento de grafo, junto com o espaço de busca de soluções. Há um total de $\binom{8}{4} = 70$ possíveis estados, mas como cada um deles tem um gêmeo idêntico no qual os lados esquerdo e direito estão trocados, com efeito há apenas 35 soluções. Na figura, elas estão organizadas em sete grupos para efeito de clareza. Existem cinco ótimos locais, dos quais quatro são ruins, com custo 2, e um é bom, com custo 0. Se busca local é iniciada em uma solução aleatória e, em cada passo um vizinho aleatório de custo menor é selecionado, então a probabilidade de a busca selecionar uma solução ruim é, no máximo, quatro vezes maior do que a de selecionar uma boa. Assim, apenas algumas poucas repetições é necessário.

Reconhecimento simulado

No exemplo da Figura 9.10, cada execução da busca local tem uma chance razoável de encontrar o ótimo global. Isto não é sempre verdade. À medida que o tamanho do problema cresce, a razão entre ótimos locais ruins e bons freqüentemente cresce, às vezes até o ponto de ter tamanho exponencial. Em tais casos, simplesmente repetir a busca local poucas vezes não é efetivo.

Figura 9.10 O espaço de estados para um grafo de oito nós. O espaço contém 35 soluções, que foram particionadas em sete grupos por clareza. Um exemplo de cada é mostrado. Existem cinco ótimos locais.



Uma abordagem diferente é permitir ocasionalmente movimentos que de fato aumentem o custo, na esperança de que eles vão puxar a busca para fora de becos sem saída. Isso seria muito útil no ótimo local ruim da Figura 9.10, por exemplo. O método de *recozimento simulado* (*simulated annealing*) redefine a busca local introduzindo a noção de uma *temperatura* T .

```

seja  $s$  qualquer solução inicial
repetir
    escolher aleatoriamente uma solução  $s'$  na vizinhança de  $s$ 
    se  $\Delta = \text{custo}(s') - \text{custo}(s)$  é negativo:
        substituir  $s$  por  $s'$ 
    senão:
        substituir  $s$  por  $s'$  com probabilidade  $e^{-\Delta/T}$ .

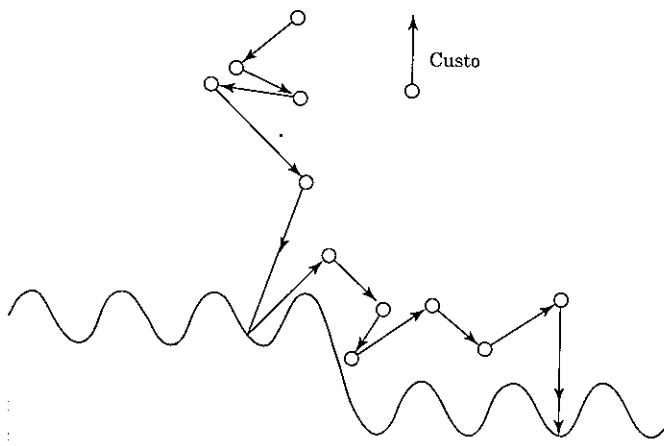
```

Se T é zero, isso se torna idêntico à nossa busca local anterior. Mas se T é grande, movimentos que aumentam o custo são ocasionalmente aceitos. Qual valor de T deve ser usado?

O truque é começar com um T grande e, então, gradualmente reduzi-lo até zero. Assim inicialmente, a busca local pode vagar bem livremente, com uma preferência apenas pequena por soluções de baixo custo. À medida que o tempo passa, essa preferência torna-se mais forte, e o sistema permanece preferencialmente na região de custo mais baixo do espaço de estados, com excursões ocasionais para fora dela para escapar de ótimos locais. Em algum dado momento, quando a temperatura cair ainda mais, o sistema converge para uma solução. A Figura 9.11 mostra o processo esquematicamente.

Recozimento simulado é inspirado pela física de cristalização. Quando uma substância deve ser cristalizada, ela começa em estado líquido, com suas partículas em movimento relativamente descontrolado. Depois ela é lentamente resfriada e, à medida que

Figura 9.11 Recozimento simulado.



isso ocorre, as partículas gradualmente se movem para configurações mais regulares. Esta regularidade torna-se mais e mais pronunciada até que por fim um cristal é formado.

Os benefícios de reconhecimento simulado vêm a um custo significativo: em virtude da temperatura variável e da liberdade inicial de movimento, muito mais movimentos locais são necessários até a convergência. Além disso, é uma arte e tanto escolher um bom escalonamento segundo o qual decrescer a temperatura, chamado de *escalonamento de reconhecimento*. Mas em muitos casos em que a qualidade das soluções melhora significativamente, o compromisso é recompensador.

Exercícios

- 9.1. No algoritmo de *backtracking* para SAT, suponha que sempre escolhamos um subproblema (fórmula CNF) que tenha uma cláusula tão pequena quanto possível; e expandamos ao longo de uma variável que aparece nesta cláusula pequena. Mostre que se a fórmula de entrada contém somente cláusulas com dois literais (ou seja, é uma instância de 2SAT), então uma atribuição satisfatória, se existir, será achada em tempo polinomial.
- 9.2. Projete um algoritmo de *backtracking* para o problema do CAMINHO RUDRATA de um vértice fixo s . Para especificar completamente um tal algoritmo você tem que definir:
 - (a) O que é um subproblema?
 - (b) Como escolher um subproblema.
 - (c) Como expandir um subproblema.

Argumente brevemente por que suas escolhas são razoáveis.

- 9.3. Projete um algoritmo de *branch-and-bound* para o problema da COBERTURA DE CONJUNTO. Isso implica decidir:
 - (a) O que é um subproblema?
 - (b) Como você escolhe um subproblema para expandir?
 - (c) Como você expande um subproblema?
 - (d) O que é uma cota inferior apropriada?

Você acredita que suas escolhas vão funcionar bem em instâncias típicas do problema? Por quê?

- 9.4. Dado um grafo não-direcionado $G = (V, E)$ no qual cada nó tem grau $\leq d$, mostre como eficientemente encontrar um conjunto independente cujo tamanho é pelo menos $1/(d + 1)$ vezes o tamanho do maior conjunto independente.
- 9.5. *Busca local para árvores espalhadas mínimas.* Considere o conjunto de todas as árvores espalhadas (não apenas as mínimas) de um grafo não-direcionado, conexo, com pesos, $G = (V, E)$.

Lembre-se da Seção 5.1 que adicionar uma aresta e a uma árvore espalhada T cria um ciclo único, e subsequentemente remover qualquer outra aresta $e' \neq e$ desse ciclo fornece uma árvore espalhada T' diferente. Dizemos que T e T' diferem por uma única *troca de aresta* (e, e') e que elas são *vizinhas*.

- (a) Mostre que é possível mover de qualquer árvore espalhada T para qualquer outra árvore espalhada T' realizando uma série de trocas de aresta, ou seja, movendo de vizinho em vizinho. No máximo, quantas trocas de aresta são necessárias?
- (b) Mostre que se T' é uma AEM, então é possível escolher essas trocas de modo que os custos das árvores espalhadas encontradas pelo caminho sejam não crescentes. Em outras palavras, se a seqüência de árvores espalhadas encontrada é

$$T = T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k = T',$$

o custo(T_{i+1}) \leq custo(T_i) para todo $i < k$.

- (c) Considere o seguinte algoritmo de busca local ao qual é dado como entrada um grafo não-direcionado com pesos de aresta distintos.

Seja T qualquer árvore espalhada de G
enquanto existe uma troca de aresta (e, e') que reduz custo(T):

$$T \leftarrow T + e - e'$$

retornar T

Mostre que esse procedimento sempre retorna uma árvore espalhada mínima.
No máximo, quantas iterações ele toma?

- 9.6. No problema da ÁRVORE DE STEINER MÍNIMA, a entrada consiste em: um grafo completo $G = (V, E)$ com distâncias d_{uv} entre todos os pares de nós; e um determinado conjunto de vértices terminais $V' \subseteq V$. O objetivo é encontrar uma árvore de custo mínimo que inclua os vértices V' . Essa árvore pode ou não incluir nós em $V - V'$.



Suponha que as distâncias na entrada sejam uma métrica (reveja a definição da página 279). Mostre que um algoritmo de aproximação eficiente de razão 2 para ÁRVORE DE STEINER MÍNIMA pode ser obtida ignorando os nós não terminais e simplesmente retornando a árvore espalhada mínima em V' . (Dica: Reveja nosso algoritmo de aproximação para o TSP.)

- 9.7. No problema do CORTE MÚLTIPLO, a entrada é um grafo não-direcionado $G = (V, E)$ e um conjunto de nós terminais $s_1, s_2, \dots, s_k \in V$. O objetivo é encontrar um conjunto mínimo de arestas em E cuja remoção deixa todos os terminais em componentes diferentes.

- (a) Mostre que este problema pode ser resolvido exatamente em tempo polinomial quando $k = 2$.
- (b) Forneça um algoritmo de aproximação com razão no máximo 2 para o caso $k = 3$.
- (c) Projete um algoritmo de busca local para o corte múltiplo.

- 9.8. No problema MAX SAT, é dado um conjunto de cláusulas e queremos encontrar uma atribuição que satisfaça o número máximo possível delas.

(a) Mostre que se este problema pode ser resolvido em tempo polinomial, então SAT também pode.

(b) A seguir, um algoritmo bastante ingênuo.

para cada variável:

atribua valor 0 ou 1 a ela, lançando uma moeda

Suponha que a entrada tenha m cláusulas, das quais a j -ésima tem k_j literais. Mostre que o número esperado de cláusulas satisfeitas por este algoritmo simples é

$$\sum_{j=1}^m \left(1 - \frac{1}{2^{k_j}}\right) \geq \frac{m}{2}.$$

Em outras palavras, esta esperança é uma aproximação de fator 2! E se todas as cláusulas contêm k literais, então este fator de aproximação melhora para $1 + 1/(2^k - 1)$.

(c) Você pode tornar este algoritmo determinístico? (Dica: Em vez de lançar uma moeda para cada variável, selecione o valor que satisfaz o maior número de cláusulas ainda não satisfeitas. Qual fração das cláusulas é satisfeita no final?)

- 9.9. No problema do CORTE MÁXIMO é dado um grafo não-direcionado $G = (V, E)$ com um peso $w(e)$ em cada aresta e gostaríamos de separar os vértices em dois conjuntos S e $V - S$ tal que o peso total das arestas entre os dois conjuntos seja tão grande quanto possível.

Para cada $S \subseteq V$, defina $w(S)$ como a soma de todos os w_{uv} sobre todas as arestas $\{u, v\}$ tal que $|S \cap \{u, v\}| = 1$. Obviamente, CORTE MÁXIMO é maximizar $w(S)$ sobre todos os subconjuntos de V .

Considere o seguinte algoritmo de busca local para CORTE MÁXIMO:

começar com qualquer $S \subseteq V$
 enquanto existir um subconjunto $S' \subseteq V$ tal que
 $|S' - S| \cup (S - S')| = 1$ e $w(S') > w(S)$ faça:
 $S = S'$

(a) Mostre que é um algoritmo de aproximação para CORTE MÁXIMO com razão 2.

(b) Mas será que é um algoritmo polinomial?

- 9.10. Vamos chamar um algoritmo de busca local *exato* quando ele sempre produz a solução ótima. Por exemplo, o algoritmo de busca local para o problema da árvore espalhada mínima introduzido no Problema 9.5 é exato. Para um outro exemplo, o simplex pode ser considerado um algoritmo de busca local exato para programação linear.

(a) Mostre que o algoritmo de busca local de mudança-2 para o TSP não é exato.

(b) Repita para o algoritmo de busca local de mudança- $\lceil \frac{n}{2} \rceil$, onde n é o número de cidades.

(c) Mostre que o algoritmo de busca local de mudança- $(n - 1)$ é exato.

(d) Se A é um problema de otimização, defina APERFEIÇOAMENTO- A como o seguinte problema de busca: dada uma instância x de A e uma solução s de A , encontre outra solução de x com custo melhor (ou declare que nenhuma existe, e que

assim s é ótima). Por exemplo, no APERFEIÇOAMENTO TSP é dada uma matriz de distância e um circuito, e precisamos encontrar um circuito melhor. Mostra-se que APERFEIÇOAMENTO TSP é NP-completo e APERFEIÇOAMENTO COBERTURA DE VÉRTICE também é. Prove esse último resultado.

- (e) Dizemos que um algoritmo de busca local tem *iteração polinomial* se cada execução do loop requer tempo polinomial. Por exemplo, as implementações óbvias do algoritmo de busca local de mudança- $(n - 1)$ para o TSP, definidas acima, não apresentam iteração polinomial. Mostre que, a menos que $P = NP$, não existe nenhum algoritmo de busca local exato com iteração polinomial para os problemas TSP e COBERTURA DE VÉRTICE.

Capítulo 10

Algoritmos quânticos

Este livro começou com os algoritmos mais amplamente usados e mais velhos do mundo (aqueles para adicionar e multiplicar números) e um problema antigo e difícil (**FATORAÇÃO**). Neste capítulo final viramos a mesa: apresentamos um dos mais novos algoritmos — e ele é um algoritmo eficiente para **FATORAÇÃO**!

Há um problema, claro: este algoritmo precisa de um *computador quântico* para ser executado.

Física quântica é uma teoria bonita e misteriosa que descreve a Natureza no detalhe, no nível de partículas elementares. Uma das maiores descobertas dos anos de 1990 foi que computadores quânticos — computadores baseados em princípios da física quântica — são radicalmente diferentes daqueles que operam segundo os princípios mais familiares da física clássica. Surpreendentemente, eles podem ser exponencialmente mais eficientes: como veremos, computadores quânticos podem resolver **FATORAÇÃO** em tempo polinomial! Como resultado, em um mundo com computadores quânticos, os sistemas que atualmente garantem transações comerciais na Internet (e são baseados no sistema de criptografia RSA) não serão mais seguros.

10.1 Qubits, superposição e medida

Nesta seção introduzimos as características básicas de física quântica que são necessárias para entender o funcionamento dos computadores quânticos.¹

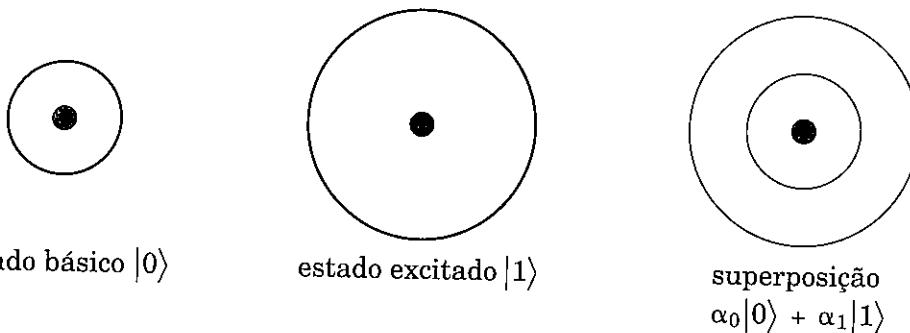
Em chips de computadores comuns, os bits são fisicamente representados por voltagens baixa ou alta em fios. Mas há muitas outras maneiras com as quais um bit pode ser guardado — por exemplo, no estado de um átomo de hidrogênio. O elétron único neste átomo pode estar ou no estado básico (a configuração de energia mais baixa) ou pode estar em um estado excitado (uma configuração de alta energia). Podemos usar os dois estados para codificar, respectivamente, os valores 0 e 1 do bit.

Introduziremos agora alguma notação de física quântica. Denotamos o estado básico do nosso elétron por $|0\rangle$, pois ele codifica o valor 0 do bit e, da mesma maneira, o estado

¹Este campo é tão estranho que o famoso físico Richard Feynman é citado como tendo dito: “Acho que posso seguramente dizer que ninguém entende física quântica”. Portanto existe pouca chance de que você vá entender a teoria em profundidade depois de ler esta seção! Mas se estiver interessado em aprender mais, veja a leitura recomendada no final do livro.

Figura 10.1 Um elétron pode estar em um estado básico ou em um estado excitado.

Na notação de Dirac usada em física quântica, eles são denotados $|0\rangle$ e $|1\rangle$. Mas o princípio da superposição informa que, na verdade, o elétron está em um estado que é uma combinação linear dos dois: $\alpha_0|0\rangle + \alpha_1|1\rangle$. Isso faria sentido imediato se α fossem probabilidades, números reais não-negativos somando 1. Contudo o princípio da superposição insiste em que eles podem ser *números complexos arbitrários*, desde que os quadrados das suas normas somem 1!

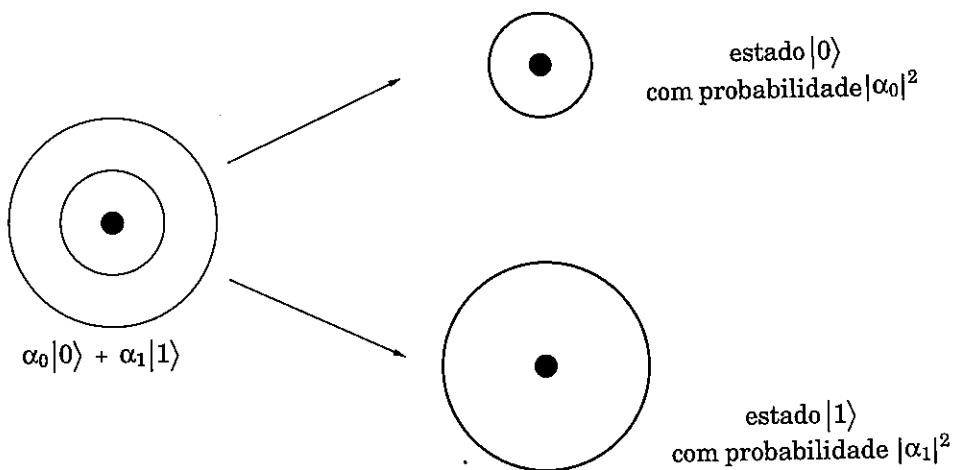


excitado por $|1\rangle$. Esses são os dois estados possíveis do elétron em física clássica. Muitos dos aspectos mais contra-intuitivos da física quântica surgem do *princípio da superposição* que afirma que se um sistema quântico pode estar em um de dois estados, então ele também pode estar em *qualquer superposição linear* daqueles dois estados. Por exemplo, o estado do elétron poderia bem ser $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ ou $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$; ou um número infinito de outras combinações na forma $\alpha_0|0\rangle + \alpha_1|1\rangle$. O coeficiente α_0 é chamado de a *amplitude* do estado $|0\rangle$, e similarmente com α_1 . E — como se as coisas já não estejam estranhas o suficiente — os α podem ser números complexos, *desde que* estejam normalizados de forma que $|\alpha_0|^2 + |\alpha_1|^2 = 1$. Por exemplo, $\sqrt{5}|0\rangle + \sqrt{5}|1\rangle$ (onde i é a unidade imaginária, $\sqrt{-1}$) é um estado quântico perfeitamente válido! Tal superposição, $\alpha_0|0\rangle + \alpha_1|1\rangle$, é a unidade básica de informação codificada em computadores quânticos (Figura 10.1). Ela é chamada um *qubit* (pronunciado “quiubits”).

O conceito todo de superposição sugere que o elétron não se decide se está em um estado básico ou em um estado excitado, e a amplitude α_0 é a medida da sua inclinação para o estado básico. Continuando por essa linha de pensamento, é tentador pensar em α_0 como a *probabilidade* de que o elétron esteja no estado básico. Mas como damos sentido ao fato de que α_0 pode ser negativo ou, ainda pior, imaginário? Esse é um dos mais misteriosos aspectos de física quântica, um que parece se estender para além da nossa intuição sobre o mundo físico.

A superposição linear, entretanto, é o mundo privado do elétron. Para termos uma visão do estado do elétron, precisamos fazer uma *medida* e, quando fazemos isso, obtemos um bit único de informação — 0 ou 1. Se o estado do elétron é $\alpha_0|0\rangle + \alpha_1|1\rangle$, a saída da medida é 0 com probabilidade $|\alpha_0|^2$ e 1 com probabilidade $|\alpha_1|^2$ (felizmente

Figura 10.2 A medida de uma superposição tem o efeito de forçar o sistema a decidir por um particular estado, com probabilidades determinadas pelas amplitudes.



normalizamos, assim $|\alpha_0|^2 + |\alpha_1|^2 = 1$). Além disso, o ato da medida faz o sistema mudar seu estado: se o resultado da medida é 0, então o novo estado do sistema é $|0\rangle$ (o estado básico), e se o resultado é 1, o novo estado é $|1\rangle$ (o estado excitado). Esta característica de física quântica, de que uma medida perturba o sistema e o força a decidir (neste caso por estado básico ou excitado), é outro fenômeno estranho sem um análogo clássico.

O princípio da superposição vale não somente para sistemas de dois níveis como o que acabamos de descrever, mas, em geral, para sistemas de k níveis. Por exemplo, na realidade o elétron em um átomo de hidrogênio pode estar em um de vários níveis de energia, começando com o estado básico, o primeiro estado excitado, o segundo estado excitado e assim por diante. Poderíamos considerar um sistema de k níveis consistindo em um estado básico e os $k-1$ primeiros estados excitados, e poderíamos denotá-los por $|0\rangle, |1\rangle, |2\rangle, \dots, |k-1\rangle$. O princípio da superposição diria, então, que o estado quântico geral do sistema é $\alpha_0|0\rangle + \alpha_1|1\rangle + \dots + \alpha_{k-1}|k-1\rangle$, onde $\sum_{j=0}^{k-1} |\alpha_j|^2 = 1$. Medir o estado do sistema revelaria agora um número entre 0 e $k-1$, e o resultado j ocorreria com probabilidade $|\alpha_j|^2$. Como antes, a medida perturbaria o sistema e o novo estado *iria se tornar de fato* $|j\rangle$ ou o j -ésimo estado excitado.

Como codificamos n bits de informação? Poderíamos escolher $k=2^n$ níveis do átomo de hidrogênio. Mas uma opção mais promissora é usar n qubits.

Vamos começar considerando o caso de dois qubits, ou seja, o estado dos elétrons de *dois* átomos de hidrogênio. Como cada elétron pode estar ou no estado básico ou no estado excitado, na física clássica os dois elétrons têm um total de quatro possíveis estados — 00, 01, 10 ou 11 — e são, portanto, adequados para guardar 2 bits de informação. Mas em

Emaranhamento

Suponha que tenhamos dois qubits, o primeiro no estado $\alpha_0|0\rangle + \alpha_1|1\rangle$ e o segundo no estado $\beta_0|0\rangle + \beta_1|1\rangle$. Qual é o estado conjunto dos dois qubits? A resposta é, o produto (tensorial) dos dois: $\alpha_0\beta_0|00\rangle + \alpha_0\beta_1|01\rangle + \alpha_1\beta_0|10\rangle + \alpha_1\beta_1|11\rangle$.

Dado um estado arbitrário de dois qubits, será que podemos especificar o estado de cada qubit individual dessa maneira? Não, em geral os dois qubits estão *emaranhados* e não podem ser decompostos nos estados dos qubits individuais. Por exemplo, considere o estado $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$, um dos famosos estados Bell. Ele não pode ser decomposto em estados dos dois qubits individuais (veja o Exercício 10.1). Emaranhamento é um dos mais misteriosos aspectos de mecânica quântica e é, em última análise, a fonte do poder da computação quântica.

física quântica, o princípio da superposição nos informa que o estado quântico dos dois elétrons é uma combinação linear dos quatro estados clássicos,

$$|\alpha\rangle + \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle,$$

normalizados de modo que $\sum_{x \in \{0,1\}^2} |\alpha_x|^2 = 1$.² Medir o estado do sistema agora revela 2 bits de informação e a probabilidade do resultado $x \in \{0,1\}^2$ é $|\alpha_x|^2$. Além disso, como antes, se o resultado da medida é jk , o novo estado do sistema é $|jk\rangle$: se $jk = 10$, por exemplo, o primeiro elétron está no estado excitado e o segundo está no estado básico.

Uma questão interessante aparece aqui: o que dizer se fizermos uma *medida parcial*? Por exemplo, se medirmos somente o primeiro qubit, qual é a probabilidade de que o resultado seja 0? É simples. É exatamente a mesma que seria se tivéssemos medido ambos os qubits, ou seja: $\Pr\{\text{primeiro bit} = 0\} = \Pr\{00\} + \Pr\{01\} = |\alpha_{00}|^2 + |\alpha_{01}|^2$. Ótimo, mas quanto essa medida parcial perturba o estado do sistema?

A resposta é elegante. Se o resultado da medida do primeiro bit for 0, a nova superposição é obtida cortando-se todos os termos de $|\alpha\rangle$ que são inconsistentes com esse resultado (cujo primeiro bit é 1). Claro que a soma dos quadrados das amplitudes não é mais 1, portanto precisamos normalizar. No nosso exemplo, o novo estado seria

$$|\alpha_{\text{novo}}\rangle = \frac{\alpha_{00}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} |00\rangle + \frac{\alpha_{01}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} |01\rangle.$$

Por fim, vamos considerar o caso geral de n átomos de hidrogênio. Pense em n como um número razoavelmente pequeno de átomos, digamos $n = 500$. Classicamente, os estados de 500 elétrons poderiam ser usados para guardar 500 bits de informação da maneira

²Lembre-se de que $\{0, 1\}^2$ denotam o conjunto consistindo nas quatro strings binárias de dois bits e, em geral, $\{0, 1\}^n$ denota o conjunto de todas as strings binárias de n bits.

óbvia. Mas o estado quântico dos 500 qubits é uma superposição linear de todos os 2^{500} estados clássicos possíveis:

$$\sum_{x \in \{0,1\}^n} \alpha_x |x\rangle.$$

É como se a Natureza tivesse 2^{500} pedaços de papel com ela, cada um com um número complexo escrito nele, apenas para rastrear o estado deste sistema de 500 átomos de hidrogênio! Além disso, em cada momento, à medida que o estado do sistema evolui, é como se a Natureza riscasse o número complexo em cada pedaço de papel e o substituisse por um novo valor.

Vamos considerar o esforço envolvido em fazer tudo isso. O número 2^{500} é muito maior do que as estimativas para o número de partículas elementares no universo. Onde, então, a Natureza guarda essa informação? Como podem sistemas quânticos microscópicos de uns poucos átomos conter mais informação do que podemos guardar no universo clássico inteiro? Certamente isso é uma teoria bastante extravagante sobre a quantidade de esforço feita pela Natureza apenas para manter um sistema minúsculo evoluindo no tempo.

Nesse fenômeno se encontra a motivação básica para computação quântica. Afinal, se a Natureza é tão extravagante no nível quântico, por que deveríamos basear nossos computadores em física clássica? Por que não fazer uso dessa quantidade gigantesca de esforço que é gasta no nível quântico?

Mas há um problema fundamental: a superposição linear de tamanho exponencial é o mundo privado dos elétrons. Medir o sistema somente revela n bits de informação. Como antes, a probabilidade de que o resultado seja uma particular *string* de 500 bits x é $|\alpha_x|^2$. E o novo estado após a medida é apenas $|x\rangle$.

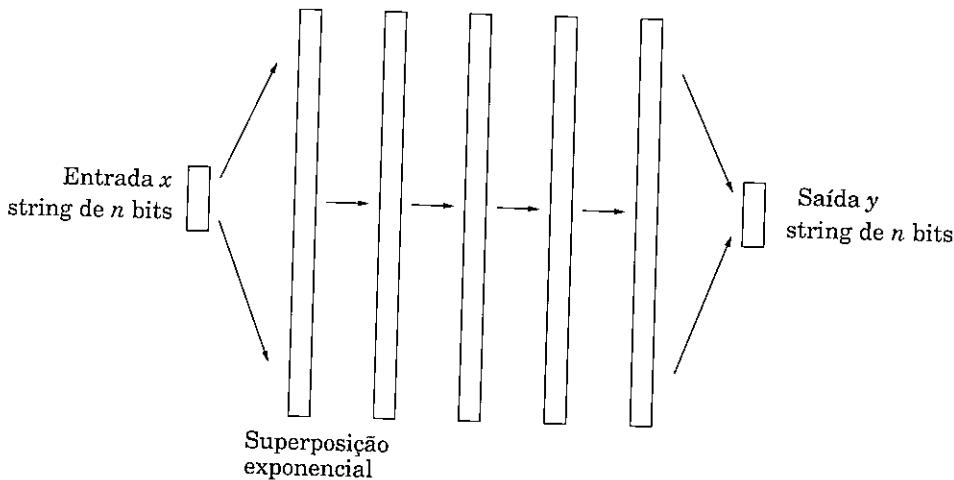
10.2 O plano

Um algoritmo quântico não é como nenhum outro que você viu até agora. Sua estrutura reflete a tensão entre o “ambiente de trabalho privado” exponencial de um sistema de n qubits e os meros n bits que podem ser obtidos pela medida.

A entrada para um algoritmo quântico consiste em n bits clássicos, e a saída também consiste em n bits clássicos. É quando o sistema quântico não está sendo observado que os efeitos quânticos operam e temos os benefícios da Natureza trabalhando com esforço exponencial para nós.

Se a entrada é uma *string* x de n bits, o computador quântico toma como entrada n qubits no estado $|x\rangle$. Depois uma série de operações quânticas é realizada, ao final da qual o estado dos n qubits foi transformado para alguma superposição $\sum_y \alpha_y |y\rangle$. Finalmente, uma medida é realizada e a saída é a *string* y de n bits com probabilidade $|\alpha_y|^2$. Observe que essa saída é *aleatória*. Mas isso não é um problema, como vimos antes com algoritmos randomizados, como aquele para teste de primalidade. Desde que y corresponda à resposta correta com probabilidade alta o suficiente, podemos repetir o processo inteiro algumas vezes para tornar a chance de falha minúscula.

Figura 10.3 Um algoritmo quântico toma n bits “clássicos” como sua entrada, os manipula para criar uma superposição de seus 2^n estados possíveis, manipula a superposição de tamanho exponencial para obter o resultado quântico final e, então, mede o resultado para obter (com a distribuição de probabilidade apropriada) os n bits de saída. Para a fase do meio, existem operações elementares que contam como um passo, mas que manipulam todas as amplitudes, uma quantidade exponencial da superposição.



Vejamos a parte quântica do algoritmo. Algumas das operações-chave quânticas (que iremos discutir em breve) podem ser imaginadas como uma busca por certos tipos de *padrão* em uma superposição de estados. Por causa disso, é útil imaginar o algoritmo com dois estágios. No primeiro estágio, os n bits clássicos da entrada são “desempacotados” em uma superposição de tamanho exponencial, que é expressamente preparada para ter um padrão subjacente ou regularidade que, se detectada, resolveria a tarefa em questão. O segundo estágio consiste em um conjunto adequado de operações quânticas, seguidas de uma medida, que revela o padrão escondido.

Tudo isso provavelmente soa bastante misterioso no momento, mas mais detalhes virão logo. Na Seção 10.3 daremos uma descrição em alto nível da operação mais importante que pode ser eficientemente realizada por um computador quântico: uma versão quântica da transformada rápida de Fourier (TRF). Depois vamos descrever certos padrões cuja detecção pode ser feita de forma ideal por esta TRF quântica, e mostraremos como reformular o problema de fatorar um inteiro N em termos da detecção de precisamente um tal padrão. Por fim, veremos como preparar o estágio inicial do algoritmo quântico, que converte a entrada N em uma superposição de tamanho exponencial com o tipo certo de padrão.

O algoritmo para fatorar um inteiro grande N pode ser visto como uma seqüência de reduções (e tudo o que é mostrado aqui em itálico será definido no tempo adequado):

- **FATORAÇÃO** é reduzida a encontrar uma *raiz quadrada não trivial* de 1 módulo N .
- Encontrar uma tal raiz é reduzido a computar a *ordem* de um inteiro aleatório módulo N .
- A ordem de um inteiro é precisamente o *período* de uma *particular superposição periódica*.
- Finalmente, períodos de superposições podem ser encontrados pela *TRF quântica*.

Começaremos com o último passo.

10.3 A transformada de Fourier quântica

Lembre-se da transformada rápida de Fourier (TRF) do Capítulo 2. Ela toma como entrada um vetor α de dimensão M e valores complexos (onde M é uma potência de 2, digamos $M = 2^m$), e devolve como saída um vetor β de dimensão M e valores complexos:

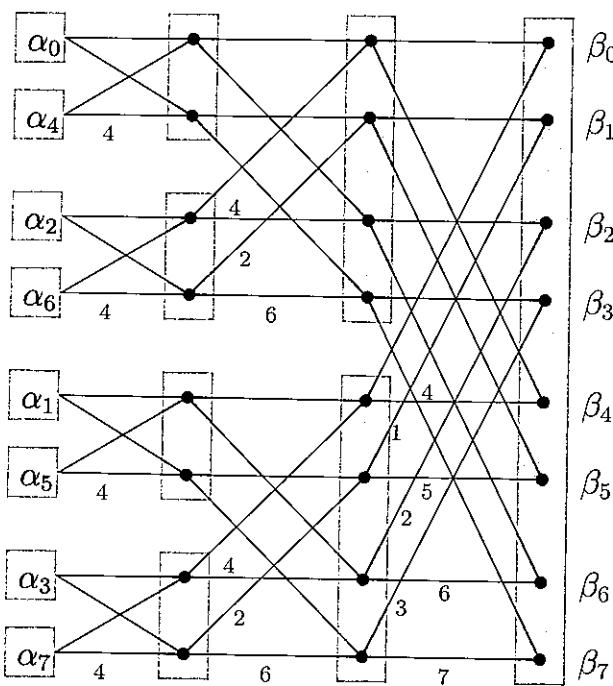
$$\begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{M-1} \end{bmatrix} = \frac{1}{\sqrt{M}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{M-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(M-1)} \\ \vdots & & & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(M-1)j} \\ \vdots & & & & \\ 1 & \omega^{(M-1)} & \omega^{2(M-1)} & \cdots & \omega^{(M-1)(M-1)} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{M-1} \end{bmatrix},$$

onde ω é uma M -ésima raiz complexa da unidade (o fator extra de \sqrt{M} é novo e tem o efeito de assegurar que se os $|\alpha_i|^2$ totalizam 1, então também o fazem os $|\beta_i|^2$). Embora a equação anterior sugira um algoritmo $O(M^2)$, a TRF clássica pode realizar este cálculo em apenas $O(M \log M)$ passos e é este ganho que provocou o efeito profundo de fazer processamento digital de sinais viável na prática. Veremos agora que computadores quânticos podem implementar a TRF *exponencialmente* mais rápido, em tempo $O(\log^2 M)$!

Mas espere um pouco, como pode algum algoritmo tomar tempo menor do que M , o comprimento da entrada? A questão é que podemos codificar a entrada em uma superposição de apenas $m = \log M$ qubits: afinal, a superposição consiste em 2^m valores de amplitude. Na notação que introduzimos antes, escreveríamos a superposição como $|\alpha\rangle = \sum_{j=0}^{M-1} \alpha_j |j\rangle$, onde α_i é a amplitude da string binária de m bits correspondente ao número i da maneira natural. Isso explica um problema importante: a notação $|j\rangle$ é, realmente, apenas uma outra maneira de escrever um vetor, em que o índice de cada célula do vetor é escrito explicitamente nesse símbolo de colchetes especial.

Começando dessa superposição de entrada $|\alpha\rangle$, a *transformada de Fourier quântica* (TFQ) a manipula apropriadamente em $m = \log M$ estágios. Em cada estágio a superposição evolui para que possa codificar os resultados intermediários do mesmo estágio da

Figura 10.4 O circuito clássico da TRF do Capítulo 2. Os vetores de entrada de M bits são processados em uma seqüência de $m = \log M$ níveis.

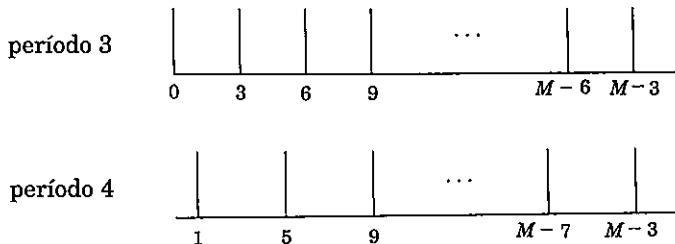


TRF clássica (cujo circuito, com $m = \log M$ estágios, é reproduzido do Capítulo 2 na Figura 10.4). Como veremos na Seção 10.5, isso pode ser alcançado com m operações quânticas por estágio. No final, depois de m estágios e $m^2 = \log^2 M$ operações elementares, obtemos a superposição $|\beta\rangle$ que corresponde à saída desejada da TFQ.

Até agora consideramos apenas as boas notícias sobre a TFQ: sua velocidade impressionante. Agora é o momento de examinar os detalhes. O algoritmo TRF clássico retorna, de fato, os M números complexos $\beta_0, \dots, \beta_{M-1}$. Em contraste, a TFQ somente prepara uma superposição $|\beta = \sum_{j=0}^{M-1} \beta_j |j\rangle$. E, como vimos antes, as amplitudes são parte do “mundo privado” deste sistema quântico.

Assim, a única maneira de colocar as mãos nesse resultado é medindo-o! E medir o estado do sistema somente gera $m = \log M$ bits clássicos: especificamente, a saída é o índice j com probabilidade $|\beta_j|^2$.

Portanto, em vez de TFQ, seria mais acurado chamar este algoritmo de *amostragem de Fourier quântica*. Além disso, muito embora tenhamos restringido nossa atenção ao caso $M = 2^m$ nesta seção, o algoritmo pode ser implementado para valores arbitrários de M e pode ser resumido da seguinte maneira:

Figura 10.5 Exemplos de superposições periódicas.

Entrada: Uma superposição de $m = \log M$ qubits, $|\alpha\rangle = \sum_{j=0}^{M-1} \alpha_j |j\rangle$.

Método: Usar $O(m^2) = O(\log^2 M)$ operações quânticas realiza a TRF quântica para obter a superposição $|\beta\rangle = \sum_{j=0}^{M-1} \beta_j |j\rangle$.

Saída: Um número aleatório de m bits j (isto é, $0 \leq j \leq M-1$), a partir da distribuição de probabilidade $\Pr[j] = |\beta_j|^2$.

Amostragem Fourier quântica é basicamente uma maneira rápida de obter uma idéia vaga sobre a saída da TRF clássica, apenas detectando uma das componentes maiores do vetor de resposta. De fato, nem sequer vemos o valor dessa componente — somente vemos o seu índice. Como podemos usar essa informação escassa? Em quais aplicações da TRF, o índice apenas das componentes grandes é suficiente? Isso é o que exploraremos em seguida.

10.4 Periodicidade

Suponha que a entrada para a TFQ, $|\alpha\rangle = (\alpha_0, \alpha_1, \dots, \alpha_{M-1})$, seja tal que $\alpha_i = \alpha_j$ sempre que $i \equiv j \pmod k$, onde k é um particular inteiro que divide M , ou seja, o vetor α consiste em M/k repetições de alguma seqüência $(\alpha_0, \alpha_1, \dots, \alpha_{k-1})$ de comprimento k . Além disso, suponha que exatamente um dos k números $\alpha_0, \dots, \alpha_{k-1}$ seja não-zero, digamos α_j . Então dizemos que $|\alpha\rangle$ é *periódica com período k e deslocamento j* (Figura 10.4).

Acontece que quando a entrada é periódica, podemos usar a amostragem de Fourier quântica para computar o seu período! Isso é baseado no seguinte fato, provado no próximo quadro:

Suponha que a entrada da amostragem de Fourier quântica seja periódica com período k , para algum k que divide M . Então a saída será um múltiplo de M/k , e é igualmente provável que ela seja qualquer um dos k múltiplos de M/k .

Agora um pouco de raciocínio nos diz que repetindo a amostragem umas poucas vezes (repetidamente preparando a superposição periódica e fazendo a amostragem de Fourier),

A transformada de Fourier de um vetor periódico

Suponha que o vetor $|\alpha\rangle = (\alpha_0, \alpha_1, \dots, \alpha_{M-1})$ seja periódico com período k e com nenhum deslocamento (isto é, os termos não-nulos são $\alpha_k, \alpha_{2k}, \dots$). Assim,

$$|\alpha\rangle = \sum_{j=0}^{M/k-1} \sqrt{\frac{k}{M}} |jk\rangle.$$

Mostraremos que a sua transformada de Fourier $|\beta\rangle = (\beta_0, \beta_1, \dots, \beta_{M-1})$ também é periódica, com período M/k e nenhum deslocamento.

Proposição $|\beta\rangle = \frac{1}{\sqrt{k}} \sum_{i=0}^{k-1} |\frac{jm}{k}\rangle$.

Prova. No vetor de entrada, o coeficiente α_ℓ é $\sqrt{k/M}$ se k divide ℓ e é zero caso contrário. Podemos conectar isto à fórmula para o j -ésimo coeficiente de $|\beta\rangle$:

$$\beta_j = \frac{1}{\sqrt{M}} \sum_{\ell=0}^{M-1} \omega^{j\ell} \alpha_\ell = \frac{\sqrt{k}}{M} \sum_{i=0}^{M/k-1} \omega^{jik}.$$

A somatória é uma série geométrica, $1 + \omega^{jk} + \omega^{2jk} + \omega^{3jk} + \dots$, contendo M/k termos e com razão ω^{jk} (lembre-se de que ω é uma raiz complexa M -ésima da unidade). Há dois casos. Se a razão é exatamente 1, o que acontece se $jk \equiv 0 \pmod{M}$, então a soma da série é simplesmente o número de termos. Se a razão não é 1, podemos aplicar a fórmula usual para séries geométricas para descobrir que a soma é $\frac{1 - \omega^{jk(M/k)}}{1 - \omega^{jk}} = \frac{1 - \omega^{Mj}}{1 - \omega^{jk}} = 0$.

Portanto β_j é $1/\sqrt{k}$ se M divide jk e é zero, caso contrário. ■

De maneira mais geral, podemos considerar a superposição original periódica com período k , mas com algum deslocamento $l < k$:

$$|\alpha\rangle = \sum_{j=0}^{M/k-1} \sqrt{\frac{k}{M}} |jk+l\rangle.$$

Então, como antes, a transformada de Fourier $|\beta\rangle$ terá amplitudes não-nulas precisamente nos múltiplos de M/k :

Proposição $|\beta\rangle = \frac{1}{\sqrt{k}} \sum_{i=0}^{k-1} \omega^{ijM/k} |\frac{jm}{k}\rangle$.

A prova dessa proposição é bastante semelhante à da anterior (Exercício 10.5).

Concluímos que a TFQ de qualquer superposição periódica com período k é um vetor, todo composto de zeros, exceto nos índices múltiplos de M/k , e todos estes k coeficientes não-nulos têm valores absolutos iguais. Portanto, se amostrarmos a entrada, obteremos um índice múltiplo de M/k e cada um dos k tais índices ocorrem com probabilidade $1/k$.

e então tomado o maior divisor comum de todos índices retornados, iremos, com probabilidade muito alta, obter o número M/k — e dele o período k da entrada!

Tornemos isso mais preciso.

Lema *Suponha que s amostras independentes sejam tomadas uniformemente de*

$$0, \frac{M}{k}, \frac{2M}{k}, \dots, \frac{(k-1)M}{k}.$$

Então, com probabilidade de pelo menos $1 - k/2^s$, o máximo divisor comum dessas amostras é M/k .

Prova. A única maneira disso falhar é se todas as amostras forem múltiplas de $j \cdot M/k$, onde j é algum inteiro maior do que 1. Assim, fixe qualquer inteiro $j \geq 2$. A chance de que uma particular amostra seja um múltiplo de jM/k é no máximo de $1/j \leq 1/2$; e, portanto, a chance de que todas as amostras sejam múltiplos de jM/k é de no máximo $1/2^s$.

Até agora raciocinamos sobre um particular número j ; a probabilidade de que este evento ruim aconteça para *algum* $j \leq k$ é no máximo igual à soma dessas probabilidades sobre os diferentes valores de j , que não é maior do que $k/2^s$. ■

Podemos tornar a probabilidade de falha tão pequena quanto desejarmos fazendo de s um múltiplo apropriado de $\log M$.

10.5 Circuitos quânticos

Então computadores quânticos podem calcular uma transformada de Fourier exponencialmente mais rápido do que computadores clássicos. Mas como exatamente são esses computadores? Do que é feito um *círculo quântico* e exatamente como ele computa transformadas de Fourier tão rápido?

10.5.1 Portas quânticas elementares

Uma operação quântica elementar é análoga a uma porta elementar como as portas E e NÃO em um circuito clássico. Ela opera sobre um qubit único ou dois qubits. Um dos exemplos mais importantes é a porta de Hadamard, denotada por H, que opera um qubit único. Sobre a entrada $|0\rangle$, ela resulta em $H(|0\rangle) = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. E para entrada $|1\rangle$, $H(|1\rangle) = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. Em figuras:



Note que nos dois casos, medir o qubit resultante gera 0 com probabilidade 1/2, e 1 com probabilidade 1/2. Mas o que acontece se a entrada para a porta Hadamard for uma superposição arbitrária $\alpha_0|0\rangle + \alpha_1|1\rangle$? A resposta, ditada pela linearidade da

física quântica, é a superposição $\alpha_0 H(|0\rangle) + \alpha_1 H(|1\rangle) = \frac{\alpha_0 + \alpha_1}{\sqrt{2}} |0\rangle + \frac{\alpha_0 - \alpha_1}{\sqrt{2}} |1\rangle$. E assim, se aplicarmos a porta de Hadamard à saída de uma porta de Hadamard, ela restaura o estado original do qubit!

Outra porta básica é o NÃO controlado, ou CNÃO. Ela opera sobre dois qubits, com o primeiro atuando como um qubit de controle e o segundo como o qubit-alvo. A porta CNÃO inverte o segundo bit se e somente se o primeiro qubit for um 1. Assim, CNÃO ($|00\rangle = |00\rangle$) e CNÃO ($|10\rangle = |11\rangle$):



Mais uma porta básica, a porta de fase controlada, é descrita na subseção que descreve o circuito quântico para a TFQ.

Consideremos a seguinte questão: suponha que tenhamos um estado quântico sobre n qubits, $|\alpha\rangle = \sum_{x \in \{0,1\}^n} \alpha_x |x\rangle$. Quantas destas 2^n amplitudes mudam se aplicarmos a porta de Hadamard ao primeiro qubit apenas? A resposta surpreendente é — todas elas!

A nova superposição torna-se $|\beta\rangle = \sum_{x \in \{0,1\}^n} \beta_x |x\rangle$, onde $\beta_{0y} = \frac{\alpha_{0y} + \alpha_{1y}}{\sqrt{2}}$ e $\beta_{1y} = \frac{\alpha_{0y} - \alpha_{1y}}{\sqrt{2}}$. Examinando os resultados com atenção, a operação quântica sobre o primeiro qubit lida com cada um dos y sufixos de $n - 1$ bits separadamente. O par de amplitudes α_{0y} e α_{1y} são transformadas em $(\alpha_{0y} + \alpha_{1y})/\sqrt{2}$ e $(\alpha_{0y} - \alpha_{1y})/\sqrt{2}$. Isso é exatamente a característica que irá nos dar um ganho exponencial na transformada de Fourier quântica.

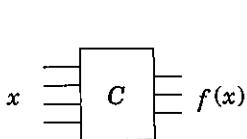
10.5.2 Dois tipos básicos de circuitos quânticos

Um circuito quântico seleciona algum número n de qubits como entrada e resulta no mesmo número de qubits. No diagrama, os n qubits são carregados pelos n fios que vão da esquerda para a direita. O circuito quântico consiste na aplicação de uma seqüência de portas quânticas elementares (do tipo descrito anteriormente) a qubits únicos e pares de qubits.

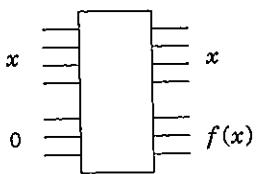
Em alto nível, existem duas funcionalidades básicas de circuitos quânticos que usamos no projeto de algoritmos quânticos:

Transformada de Fourier Quântica Estes circuitos quânticos selecionam como entrada n qubits em algum estado $|\alpha\rangle$ e devolvem na saída $|\beta\rangle$ resultante da aplicação da TFQ à $|\alpha\rangle$.

Funções Clássicas Considere uma função f com n bits de entrada e m bits de saída e suponha que haja um circuito clássico que devolva na saída $f(x)$. Então existe um circuito quântico que, sobre uma entrada de uma string x de n bits mais um sufixo de m 0, devolve na saída $x \cdot f(x)$:



Círculo clássico



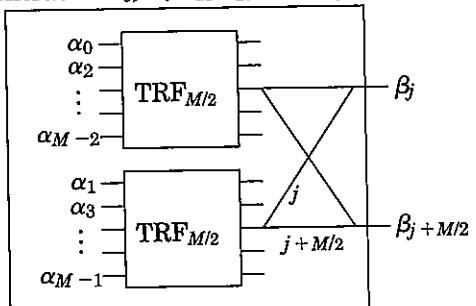
Círculo quântico

Agora a entrada para este circuito quântico poderia ser uma superposição sobre as *strings* x de n bits, $\sum_x |x, 0^k\rangle$, caso em que a saída tem de ser $\sum_x |x, f(x)\rangle$. O Exercício 10.7 explora a construção de tais circuitos por meio de portas quânticas elementares.

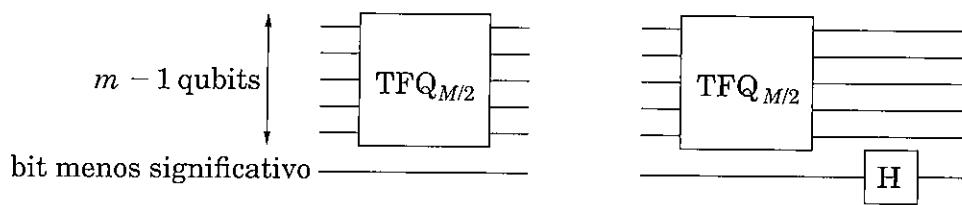
Entender circuitos quânticos neste nível é suficiente para acompanhar o restante do capítulo. A próxima subseção sobre circuitos quânticos para a TFQ pode, portanto, ser ignorada com segurança por alguém que não queira entrar nestes detalhes.

10.5.3 O circuito da transformada de Fourier quântica

Aqui reproduzimos o diagrama (da Seção 2.6.4) mostrando como o circuito da TRF clássica para vetores de tamanho M é composto de dois circuitos da TRF para vetores de tamanho $(M/2)$ seguidos de algumas portas simples.

TRF _{M} (entrada: $\alpha_0, \dots, \alpha_{M-1}$, saída: $\beta_0, \dots, \beta_{M-1}$)

Vejamos como simular em um sistema quântico. A entrada está agora codificada nas 2^m amplitudes de $m = \log M$ qubits. Assim, a decomposição das entradas em pares e ímpares, como mostrado na figura anterior, é claramente determinada por um dos qubits — o qubit menos significativo. Como separamos as entradas pares e ímpares e aplicamos os circuitos recursivos para computar $TRF_{M/2}$ em cada metade? A resposta é memorável: simplesmente aplique o circuito quântico $TFQ_{M/2}$ aos $m - 1$ qubits restantes. O efeito disso é aplicar $TFQ_{M/2}$ à superposição de todas as *strings* de m bits da forma $x0$ (das quais existem $M/2$) e, separadamente, à superposição de todas as *strings* de m bits da forma $x1$. Assim, os dois circuitos recursivos clássicos podem ser emulados por um único circuito quântico — um ganho exponencial quando desdobramos a recursão!



Consideremos agora as portas no circuito da TRF clássica *depois* das chamadas recursivas à $\text{TRF}_{M/2}$: os fios emparelham j com $M/2 + j$ e, ignorando por enquanto a fase que é aplicada ao conteúdo do $(M/2 + j)$ -ésimo fio, temos de adicionar e subtrair essas duas quantidades para obter a j -ésima e a $(M/2 + j)$ -ésima saídas, respectivamente. Como um circuito quântico alcançaria o resultado dessas M portas clássicas? Simples: apenas aplique a porta Hadamard ao primeiro qubit! Lembre-se de que, da discussão anterior (Seção 10.5.1), para qualquer possível configuração dos restantes $m - 1$ qubits x , isso emparelha as strings $0x$ e $1x$. Traduzindo a partir do binário, significa que estamos emparelhando x e $M/2 + x$. Além disso, o resultado da porta de Hadamard é que para cada tal par, as amplitudes são substituídas pela soma e diferença (normalizadas por $1/\sqrt{2}$), respectivamente. Até agora a TFQ não requer quase nenhuma porta!

A fase que tem de ser aplicada ao $(M/2 + j)$ -ésimo fio para cada j requer um pouco mais de trabalho. Note que a fase de ω^j tem de ser aplicada somente se o primeiro qubit é 1. Agora se j é representado pelos primeiros $m - 1$ bits $j_1 \dots j_{m-1}$, então $\omega^j = \prod_{l=1}^{m-1} \omega^{2^l j_l}$. Assim, a fase ω^j pode ser utilizada aplicando para o l -ésimo fio (para cada l) uma fase de ω^{2^l} se o l -ésimo qubit for um 1 e o primeiro qubit for um 1. Esta tarefa pode ser cumprida por outra porta quântica de dois bits — a porta de fase controlada. Ela deixa os dois qubits inalterados a menos que ambos sejam 1, caso em que aplica determinado fator de fase.

O circuito TFQ está especificado agora. O número de portas quânticas é dado pela fórmula $S(m) = S(m - 1) + O(m)$, que resulta em $S(m) = O(m^2)$. A TFQ sobre entradas de tamanho $M = 2^m$ requer, portanto, $O(m^2) = O(\log^2 M)$ operações quânticas.

10.6 Fatoração como periodicidade

Vimos como a transformada de Fourier quântica pode ser usada para encontrar o período de uma superposição periódica. Agora mostramos, por meio de uma seqüência de reduções simples, como fatoração pode ser reformulada como um problema de busca de período.

Fixe um inteiro N . Uma *raiz quadrada não trivial de 1 módulo N* (veja os Exercícios 1.36 e 1.40) é qualquer inteiro $x \not\equiv \pm 1 \pmod{N}$ tal que $x^2 \equiv 1 \pmod{N}$. Se pudermos encontrar uma raiz quadrada não trivial de 1 mod N , então será fácil decompor N em um produto de dois fatores não triviais (e repetir o processo fatoraria N).

Lema *Se x é uma raiz quadrada não trivial de 1 módulo N , então $\text{mdc}(x + 1, N)$ é um fator não trivial de N .*

Prova. $x^2 \equiv 1 \pmod{N}$ implica que N divide $(x^2 - 1) = (x + 1)(x - 1)$. Mas N não divide nenhum desses termos individuais, pois $x \not\equiv \pm 1 \pmod{N}$. Portanto N tem de ter um fator não trivial em comum com ambos $(x + 1)$ e $(x - 1)$. Em particular, $\text{mdc}(N, x + 1)$ é um fator não trivial de N . ■

Exemplo. Seja $N = 15$. Então $4^2 \equiv 1 \pmod{15}$, mas $4 \not\equiv \pm 1 \pmod{15}$. Ambos $\text{mdc}(4 - 1, 15) = 3$ e $\text{mdc}(4 + 1, 15) = 5$ são fatores não triviais de 15.

Para completarmos a conexão com periodicidade, precisamos de mais um conceito. Defina a *ordem* de x módulo N como o menor inteiro positivo r tal que $x^r \equiv 1 \pmod{N}$. Por exemplo, a ordem de 2 mod 15 é 4.

Computar a ordem de um número *aleatório* x mod N é fortemente relacionado ao problema de encontrar raízes quadradas não triviais e, com isso, à fatoração. Veja a ligação.

Lema *Seja N um composto ímpar, com pelo menos dois fatores primos distintos, e seja x um número escolhido aleatoriamente e uniformemente entre 0 e $N - 1$. Se $\text{mdc}(x, N) = 1$, então com probabilidade de pelo menos $1/2$, a ordem r de x mod N é par e, além disso, $x^{r/2}$ é uma raiz quadrada não trivial de $1 \pmod{N}$.*

A prova desse lema é deixada como exercício. O que ela implica é que se pudermos computar a ordem r de um elemento x mod N , escolhido aleatoriamente, então haverá uma boa chance de que essa ordem seja par e que $x^{r/2}$ seja uma raiz quadrada não trivial de 1 módulo N . Caso em que $\text{mdc}(x^{r/2} + 1, N)$ será um divisor de N .

Exemplo. Se $x = 2$ e $N = 15$, então a ordem de 2 é 4, pois $2^4 \equiv 1 \pmod{15}$. Elevando 2 à metade dessa potência, obtemos uma raiz não trivial de 1: $2^2 \equiv 4 \not\equiv \pm 1 \pmod{15}$. Portanto obtemos um divisor de 15 computando $\text{mdc}(4 + 1, 15) = 5$.

Portanto, reduzimos FATORAÇÃO ao problema de BUSCA DE ORDEM. A vantagem deste último problema é que ele tem uma função periódica natural associada: fixe N e x , e considere a função $f(a) = x^a \pmod{N}$. Se r é a ordem de x , então $f(0) = f(r) = f(2r) = \dots = 1$, e de maneira similar, $f(1) = f(r + 1) = f(2r + 1) = \dots = x$. Assim f é periódica, com período r . E podemos computá-la eficientemente com o algoritmo de quadrados sucessivos da Seção 1.2.2. Portanto, para fatorar N , tudo o que precisamos fazer é descobrir como usar a função f para preparar uma superposição periódica com período r , por meio da qual podemos usar a amostragem quântica de Fourier como na Seção 10.3 para encontrar r . Isso é descrito no próximo quadro.

10.7 O algoritmo quântico para fatoração

Podemos agora juntar todas as peças do algoritmo quântico para FATORAÇÃO (veja a Figura 10.6). Como podemos testar em tempo polinomial se a entrada é um primo ou uma potência de primo, consideraremos que já fizemos isso e que a entrada é um número composto ímpar com pelo menos dois fatores primos distintos.

Entrada: Um inteiro composto ímpar N .

Saída: Um fator de N .

Preparando uma superposição periódica

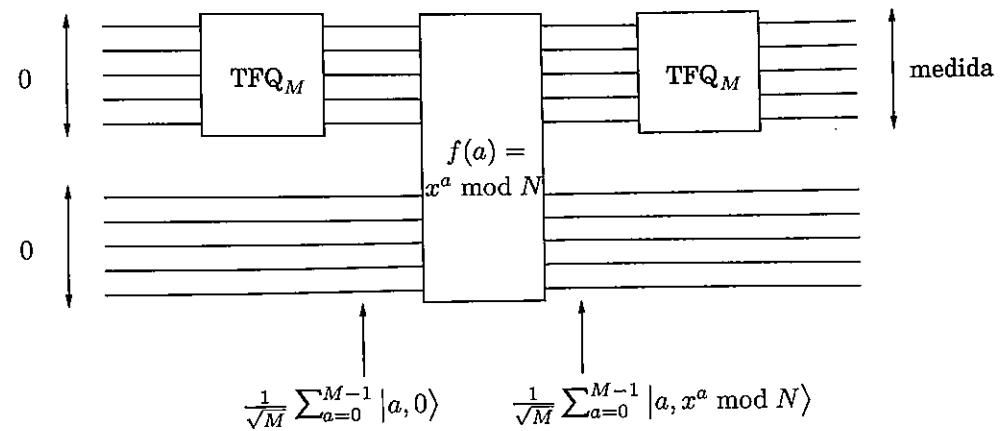
Vejamos como usar a nossa função periódica $f(a) = x^a \bmod N$ para preparar uma superposição periódica:

- Começamos com dois registradores quânticos, ambos inicialmente com 0.
- Computamos uma transformada de Fourier quântica do primeiro registrador módulo M , para obter uma superposição sobre todos os números entre 0 e $M - 1$: $\frac{1}{\sqrt{M}} \sum_{a=0}^{M-1} |a, 0\rangle$. Isso acontece porque a superposição inicial pode ser imaginada como periódica com período M , portanto a transformada é periódica com período 1.
- Computamos a função $f(a) = x^a \bmod N$. O circuito quântico para fazer isso considera o conteúdo do primeiro registrador a como entrada para f , e o segundo registrador (que é inicialmente 0) como o registrador resposta. Depois de aplicar esse circuito quântico, o estado dos dois registradores é: $\sum_{a=0}^{M-1} \frac{1}{\sqrt{M}} |a, f(a)\rangle$.
- Agora medimos o segundo registrador. Isso dá uma superposição periódica no primeiro registrador, com período r , o período de f . Entenda por quê.

Como f é uma função periódica com período r , para cada r -ésimo valor no primeiro registrador, o conteúdo do segundo registrador é o mesmo. A medida do segundo registrador gera $f(k)$ para algum k aleatório entre 0 e $r - 1$. Qual é o estado do registrador depois dessa medida? Para responder a essa questão, lembre-se das regras para medida parcial esboçadas neste capítulo. O primeiro registrador está agora em uma superposição dos valores de a que são compatíveis com o resultado da medida no segundo registrador. Mas os valores de a são exatamente $k, k + r, k + 2r, \dots, k + M - r$. Portanto o estado resultante do primeiro registrador é uma superposição periódica $|\alpha\rangle$ com período r , o que é exatamente a ordem de x que queremos encontrar!

1. Escolha x aleatória e uniformemente no intervalo $1 \leq x \leq N - 1$.
2. Seja M uma potência de 2 perto de N (por razões que não podemos abordar aqui, é melhor escolher $M \approx N^2$).
3. Repita $s = 2\log N$ vezes:
 - (a) Comece com dois registradores quânticos, ambos inicialmente 0, o primeiro grande o suficiente para guardar um número módulo M e o segundo, módulo N .
 - (b) Use a função periódica $f(a) = x^a \bmod N$ para criar uma superposição periódica $|\alpha\rangle$ de comprimento M como se segue (veja o quadro anterior para detalhes):
 - i. Aplique a TFQ ao primeiro registrador para obter a superposição $\sum_{a=0}^{M-1} \frac{1}{\sqrt{M}} |a, 0\rangle$.
 - ii. Compute $f(a) = x^a \bmod N$ usando um circuito quântico, para obter a superposição $\sum_{a=0}^{M-1} \frac{1}{\sqrt{M}} |a, x^a \bmod N\rangle$.

Figura 10.6 Fatoração quântica.



iii. Meça o segundo registrador. Agora o primeiro registrador contém a superposição periódica $|\alpha\rangle = \sum_{j=0}^{M/r-1} \frac{1}{\sqrt{M}} |jr + k\rangle$ onde k é um deslocamento aleatório entre 0 e $r - 1$ (lembre-se de que r é a ordem de x módulo N).

- (c) Faça uma amostragem de Fourier na superposição $|\alpha\rangle$ para obter um índice entre 0 e $M - 1$.

Seja g o mdc dos índices resultantes j_1, \dots, j_s .

4. Se M/g for par, então compute $\text{mdc}(N, x^{M/2g} + 1)$ e devolva isso como saída se ele for um fator não trivial de N ; caso contrário retorne ao passo 1.

De lemas anteriores, sabemos que esse método funciona para pelo menos metade das escolhas de x e, assim, o procedimento inteiro tem de ser repetido somente um par de vezes na média até que um fator seja encontrado.

Mas há um aspecto deste algoritmo, relacionado com o número M , que ainda não está muito claro: M , o tamanho da nossa TRF, tem de ser uma potência de dois. E para a nossa idéia de detecção de período funcionar, o período tem de dividir M — assim ele também tem de ser uma potência de 2. Mas o período no nosso caso é a ordem de x , definitivamente não uma potência de 2!

A razão pela qual tudo funciona de qualquer modo é a seguinte: *a transformada de Fourier quântica pode detectar o período de um vetor periódico mesmo que ele não divida M*. Mas a derivação não é tão simples como no caso em que o período de fato divide M , portanto não vamos nos aprofundar nesse ponto.

Seja $n = \log N$ o número de bits da entrada N . O tempo de execução do algoritmo é dominado pelas $2 \log N = O(n)$ repetições do passo 3. Como exponenciação modular toma $O(n^3)$ passos (como vimos na Seção 1.2.2) e a transformada de Fourier quântica toma $O(n^2)$ passos, o tempo total de execução para o algoritmo de fatoração quântica é $O(n^3 \log n)$.

Implicações para ciência da computação e física quântica

Nos primórdios da ciência da computação, as pessoas se perguntavam se haveria computadores muito mais poderosos do que aqueles feitos de circuitos compostos de portas elementares. Mas desde os anos de 1970 essa questão tem sido considerada resolvida. Computadores implementando a arquitetura de von Neumann em silício eram os vencedores óbvios e era amplamente aceito que qualquer outra maneira de implementar computadores seria polinomialmente equivalente a eles, isto é, uma computação de T passos em qualquer computador toma no máximo um número polinomial em T de passos em algum outro. Esse princípio fundamental é chamado de *tese estendida de Church-Turing*. Computadores quânticos violam essa tese fundamental e questionam algumas das nossas hipóteses mais básicas sobre computadores.

Será que computadores quânticos podem ser construídos? Esse é o desafio que mantém ocupados muitos grupos de pesquisa de físicos e cientistas da computação ao redor do mundo. O principal problema é que superposições quânticas são muito frágeis e precisam ser protegidas de qualquer medida espúria feita pelo ambiente. Existe progresso, mas ele é muito lento: até agora, a computação quântica relatada mais ambiciosa foi a fatoração do número 15 em seus fatores 3 e 5 usando ressonância magnética nuclear (RMN). E mesmo nesse experimento, há dúvidas sobre quanto fielmente o algoritmo quântico para fatoração foi implementado. A próxima década promete ser realmente excitante em termos de nossa habilidade de manipulação física de bits quânticos e de implementação de computadores quânticos.

Mas existe outra possibilidade: o que dizer se todos os esforços para implementar um computador quântico falharem? Isso seria ainda mais interessante, porque revelaria alguma falha fundamental em física quântica, uma teoria que permanece há um século sem ser desafiada.

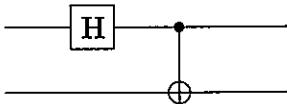
Computação quântica é motivada tanto por tentar clarificar a natureza misteriosa da física quântica quanto por tentar criar computadores inovadores e superpoderosos.

Exercícios

10.1. $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ é um dos mais famosos “estados Bell”, um estado altamente entrelançado dos seus dois qubits. Nesta questão examinamos algumas de suas estranhas propriedades.

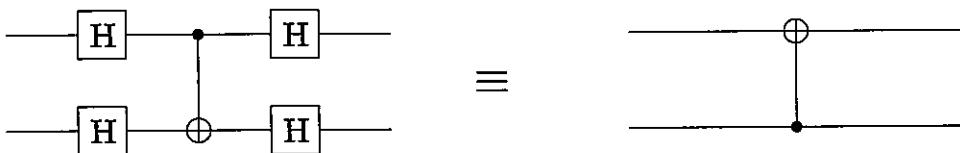
- Suponha que este estado pudesse ser decomposto como o produto (tensorial) de dois qubits (reveja o quadro da página 300), o primeiro no estado $\alpha_0|0\rangle = \alpha_1|1\rangle$ e o segundo no estado $\beta_0|0\rangle = \beta_1|1\rangle$. Escreva quatro equações que as amplitudes α_0 , α_1 , β_0 e β_1 tenham de ser satisfeitas. Conclua que o estado Bell não pode ser decomposto daquela maneira.
- Qual o resultado de medir o primeiro qubit de $|\psi\rangle$?
- Qual o resultado de medir o segundo qubit depois de medir o primeiro qubit?
- Se os dois qubits no estado $|\psi\rangle$ estão muito longe um do outro, você entende por que a resposta para (c) é surpreendente?

- 10.2. Mostre que o seguinte circuito quântico prepara o estado Bell $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ sobre a entrada $|00\rangle$: aplique uma porta de Hadamard ao primeiro qubit seguida de uma CNÃO com o primeiro qubit como o controle e o segundo qubit como o alvo.



O que o circuito devolve como saída nas entradas 10, 01 e 11? A resposta representa os restantes estados-base Bell.

- 10.3. Qual a transformada de Fourier quântica módulo M da superposição uniforme $\frac{1}{\sqrt{M}}\sum_{j=0}^{M-1}|j\rangle$?
- 10.4. Qual a TFQ módulo M de $|j\rangle$?
- 10.5. *Convolução-multiplicação.* Suponha que desloquemos uma superposição $|\alpha\rangle = \sum_j \alpha_j |j\rangle$ por l para obter a superposição $|\alpha'\rangle = \sum_j \alpha_j |j+l\rangle$. Se a TFQ de $|\alpha\rangle$ é $|\beta\rangle$, mostre que a TFQ de $|\alpha'\rangle$ é $|\beta'\rangle$, onde $\beta'_j = \beta_j \omega^{lj}$. Conclua que se $|\alpha'\rangle = \sum_{j=0}^{Mk-1} \sqrt{\frac{k}{M}} |jk+l\rangle$, então $|\beta'\rangle = |\beta'\rangle = \frac{1}{\sqrt{k}} \sum_{j=0}^{k-1} \omega^{ljM/k} |jM/k\rangle$.
- 10.6. Mostre que se você aplicar a porta de Hadamard às entradas e saídas de uma porta CNÃO, o resultado é uma porta CNÃO com qubits de controle e alvo trocados:



- 10.7. A porta TROCA CONTROLADA (C-SWAP) toma como entrada 3 qubits e troca o segundo com o terceiro se e somente se o primeiro qubit for um 1.

- Mostre que cada uma das portas NÃO, CNÃO e C-SWAP são suas próprias inversas.
- Mostre como implementar uma porta E usando uma porta C-SWAP, isto é, quais entradas a, b, c você deve dar a uma porta C-SWAP para que uma das saídas seja $a \wedge b$?
- Como você alcançaria “fan-out” usando apenas essas três portas? Ou seja, fazer que sobre a entrada a e 0, devolva na saída a e a .
- Conclua, portanto, que para qualquer circuito clássico C existe um circuito quântico equivalente Q usando apenas as portas NÃO e C-SWAP no seguinte sentido: se C devolve na saída y com entrada x , então Q devolve $|x, y, z\rangle$ com entrada $|x, 0, 0\rangle$. (Aqui z é algum conjunto de junk bits gerados durante a computação.)
- Agora mostre que existe um circuito quântico Q^{-1} que devolva na saída $|x, 0, 0\rangle$ sobre a entrada $|x, y, z\rangle$.
- Mostre que existe um circuito quântico Q' feito de portas NÃO, CNÃO e C-SWAP que devolva na saída $|x, y, 0\rangle$ com entrada $|x, 0, 0\rangle$.

10.8. Neste problema vamos mostrar que se $N = pq$ é o produto de dois primos ímpares e se x é escolhido aleatória e uniformemente entre 0 e $N - 1$, tal que o $\text{mdc}(x, N) = 1$, então com probabilidade de pelo menos $3/8$, a ordem r de $x \bmod N$ é par e, além disso, $x^{r/2}$ é uma raiz quadrada não trivial de 1 mod N .

- (a) Seja p um primo ímpar e seja x um número uniformemente aleatório módulo p . Mostre que a ordem de $x \bmod p$ é par com probabilidade de pelo menos $1/2$. (*Dica:* Use o pequeno teorema de Fermat (Seção 1.3).)
- (b) Use o teorema chinês do resto (Exercício 1.37) para mostrar que com probabilidade de pelo menos $3/4$, a ordem r de $x \bmod N$ é par.
- (c) Se r é par, mostre que a probabilidade de que $x^{r/2} \equiv \pm 1$ é de no máximo $1/2$.

Notas históricas e leitura adicional

Capítulos 1 e 2

O livro clássico sobre teoria dos números é

G. H. Hardy e E. M. Wright, Introduction to the Theory of Numbers. Oxford University Press, 1980.

O algoritmo para primalidade foi descoberto por Robert Solovay e Volker Strassen em meados dos anos de 1970, enquanto o sistema criptográfico RSA apareceu alguns anos depois. Veja

D. R. Stinson, Cryptography: Theory and Practice. Chapman and Hall, 2005

para muito mais sobre criptografia. Para algoritmos randomizados, veja

R. Motwani e P. Raghavan, Randomized Algorithms. Cambridge University Press, 1995.

Funções de espalhamento universal foram propostas em 1979 por Larry Carter e Mark Wegman. O algoritmo rápido para multiplicação de matrizes é fruto do trabalho de Volker Strassen (1969). Também em consequência do trabalho de Strassen, com Arnold Schönhage, está o algoritmo mais rápido para multiplicação inteira. Ele usa uma variante da TRF para multiplicar inteiros de n bits em $O(n \log n \log \log n)$ operações de bits.

Capítulo 3

Busca por profundidade e suas muitas aplicações foram articuladas por John Hopcroft e Bob Tarjan em 1973 — eles foram agraciados por essa contribuição com o prêmio Turing, a distinção mais alta para Ciência da Computação. O algoritmo de duas fases para encontrar componentes fortemente conexas deve-se a Rao Kosaraju.

Capítulos 4 e 5

O algoritmo de Dijkstra foi descoberto em 1959 por Edsger Dijkstra (1930-2002), enquanto o primeiro algoritmo para computar árvores espalhadas mínimas pode ser rastreado até um artigo de 1926 do matemático tcheco Otakar Boruvka. A análise da estrutura de dados unir-encontrar (*union-find*) (que é, na verdade, um pouco mais justa do que a nossa cota $\log^* n$) deve-se a Bob Tarjan. Por fim, David Huffman descobriu em 1952, quando ainda estudante de pós-graduação, o algoritmo de codificação que leva o seu nome.

Capítulo 7

O método simplex foi descoberto em 1947 por George Danzig (1914-2005) e o teorema min-max para jogos de soma-zero, em 1928, por John von Neumann (que também é considerado o pai do computador). Um livro muito bom sobre programação linear é

V. Chvátal, Linear Programming. W. H. Freeman, 1983.

E para teoria dos jogos, veja

Martin J. Osborne e Ariel Rubinstein, A course in game theory. MIT Press, 1994.

Capítulos 8 e 9

A noção de NP-completude foi primeiro identificada no trabalho de Steve Cook, que provou em 1971 que SAT é NP-completo; um ano depois Dick Karp apresentou uma lista de 23 problemas NP-completos (incluindo todos os que provamos no Capítulo 8), estabelecendo sem sombra de dúvida a aplicabilidade do conceito (os dois foram agraciados com o prêmio Turing). Leonid Levin, trabalhando na União Soviética, independentemente provou um teorema similar.

Para um excelente tratamento de NP-completude veja

M. R. Garey e D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness. W. H. Freeman, 1979.

E para o assunto mais geral de Complexidade veja

C. H. Papadimitriou, Computational Complexity. Addison-Wesley, Reading Massachusetts, 1995.

Capítulo 10

O algoritmo quântico para primalidade foi descoberto em 1994 por Peter Shor. Para uma introdução inovadora à mecânica quântica para cientistas da computação veja

<http://www.cs.berkeley.edu/~vazirani/quantumphysics.html>

e, para uma introdução à computação quântica, veja as notas de aula para o curso “Qubits, Quantum Mechanics, and Computers” em

<http://www.cs.berkeley.edu/~vazirani/cs191.html>

Índice Remissivo

- $O(\cdot)$, 6
 $\Theta(\cdot)$, 8
 $\Omega(\cdot)$, 8
 $|\cdot|$, 297
adição, 11
advanced encryption standard (AES), 32
AEM, *veja* árvore espalhada mínima
agrupamento, 239, 279
Alan M. Turing, 263
algoritmo do elipsóide, 220
algoritmo Bellman-Ford, 117
algoritmo de aproximação, 276
algoritmo de Dijkstra, 108
algoritmo de Kruskal, 128-132
algoritmo de Prim, 139
algoritmo guloso, 128
algoritmo simplex, 192
 vértice degenerado, 218
 vértice, 213
 vizinho, 213
amostragem de Fourier quântica, 304
arestas negativas em grafos, 115
aritmética modular, 16-23
 adição, 17
 divisão, 18, 23
 exponenciação, 18
 inverso multiplicativo, 23
 multiplicação, 18
árvore, 129
árvore binária
 completa, 12
 cheia, 73, 141
árvore espalhada mínima, 127, 236
 busca local, 293
ascendente, 88
assinatura digital, 43
átomo de hidrogênio, 297

backtracking, 272
base de Fourier, 65
bases, 12
biologia computacional, 166
branch-and-bound, 275
busca binária, 50
busca em profundidade, 83
 aresta de árvore, 85
 aresta de retorno, 85
busca exaustiva, 232

caminho euleriano, 100, 237
caminho mais longo, 242-243, 266
caminho mínimo, 105
 caminhos confiáveis, 171
 todos os pares, 172
caminhos e ciclos de Rudrata, 264-265
 caminho de Rudrata, 238, 247, 264
 ciclo de Rudrata, 238, 247, 256
ciclo, 89
ciclo negativo, 118
cifra de uso único, 31
circuito booleano, 221, 260
circuito euleriano, 100
circuito quântico, 307
clique, 242, 252
CNF, *veja* fórmula booleana
cobertura de vértices, 145, 241, 252
 algoritmo de aproximação, 278
codificação de Huffman, 138
codificação livre de prefixo, 141
complemento de dois, 17
componentes biconexas, 102
componentes fortemente conexas, 91
compressão de caminho, *veja*
 conjuntos disjuntos
compressão MP3, 138
computador quântico, 297
conectividade
 direcionada, 91
 não-direcionada, 86
conjunto independente, 240, 249, 252
 em árvores, 175
conjuntos disjuntos, 132
 compressão de caminho, 135
 união por rank, 133
corte máximo, 295
corte múltiplo, 294
corte, 130
 corte balanceado, 239
 corte máximo, 295
 corte mínimo, 140, 238
 corte s-t, 203
 e fluxo, 203
criptografia
 chave privada, 31-32
 chave pública, 31-33

dag, *veja* grafo direcionado acíclico
Dantzig, George, 190

descendentes, 88
DFS, *veja* busca em profundidade
distância de edição, 159
distâncias em grafos, 104
divisão, 15
dualidade, 192, 206
 caminho mínimo, 229
fluxo, 228

eliminação gaussiana, 219
emaranhamento, 300
emparelhamento
 casamento 3D, 240-243, 253-254
 emparelhamento bipartido, 205,
 228, 240
 maximal, 278
 perfeito, 205
entropia, 143, 151
equações zero-um, 240
EZU, *veja* equações zero-um

fatoração, 24, 245, 297, 310
Feynman, Richard, 297
fila de prioridade, 109-110, 113-114
fluxo, 86
fórmula booleana, 144
 atribuição satisfatória, 144, 234
 forma normal conjuntiva, 234
 implicação, 144
 literal, 144
 variável, 144
fórmula Horn, 144
função de espalhamento, 35
 para varredura na Web, 94
universal, 38

Gauss, Carl Friedrich, 45, 70
Gordon Moore, 233
grafo, 80
 aresta, 80
 denso, 82
 direcionado, 81
 esparsa, 82
 fonte, 90
 não-direcionada, 81
 nó, 80
 reverso, 96
 sorvedouro, 90
 vértices, 80

- grafo bipartido, 96
- grafo direcionado acíclico, 89
 - caminho mínimo, 119, 156
 - caminhos *mais longos*, 120
- grau, 96
- heap, 109, 114
 - binário, 114, 122
 - d-ário, 114-115, 122
 - Fibonacci, 114
- hiperplano, 213
- indecidibilidade, 264
- inequação linear, 189
- interpolação, 58, 62
- jogos
 - estratégia mista, 210
 - estratégia pura, 210
 - perdas e ganhos, 209
 - teorema min-max*, 212
- John Tukey, 70
- k-grupo, 280
- Lei de Moore, 4, 233
- Leonardo Fibonacci, 2
- Leonhard Euler, 100, 236
- linearização, 90
- lista de adjacência, 82
- \log^* , 137
- logaritmo, 12
- matriz de adjacência, 81
- matriz de Vandermonde, 64
- max SAT, *veja* satisfatibilidade
- mcd, *veja* máximo divisor comum
- mediana, 53
- medida, 298
 - parcial, 300
- método do ponto interior, 221
- mochila, 242
 - algoritmo de aproximação, 284
 - com repetições, 167
 - mochila unária, 242-243
 - sem repetições, 167
- multiplicação, 130
 - divisão-e-conquista, 45-48
- multiplicação de matrizes, 56, 168
- multiplicação polinomial, 58
- notação O , 6-8
- NP, 244
- números complexos, 62, 298
 - raízes da unidade, 63
- números de Carmichael, 26-28
- números de Fibonacci, 2
- ordem módulo N, 311
- ordenação
 - cota inferior, 52
 - mergesort iterativo, 51
 - mergesort, 50-51
 - quicksort, 56, 75
- ordenação topológica, *veja* linearização P, 244
 - particionamento de grafo, 288
 - algoritmo de Euclides, 20
 - função Euclides estendido, 21
- passos básicos de computação, 5
- pequeno teorema de Fermat, 23
- PLI, *veja* programação linear inteira
- poliedro, 192, 213
- porta de Hadamard, 307
- porta NÃO controlada, 308
- porta quântica, 310
- primalidade, 23-27
- primo relativo, 23
- primos aleatórios, 28
- princípio da superposição, 298
- problema da parada, 264
- problema de busca, 234-235
- problema do caixeiro-viajante, 173, 235
 - algoritmo de aproximação, 281
 - branch-and-bound, 276
 - busca local, 285
- problemas de otimização, 188
- problemas NP-completos, 244
- processamento de sinais, 59
- programa linear, 189
 - dual, 207
 - forma matriz-vetor, 198
 - forma padrão, 197
 - infactível, 190
 - primal, 207
- programação dinâmica
 - subproblemas comuns, 165
 - subproblemas, 158
 - versus* divisão-e-conquista, 160
- programação linear inteira, 194, 239, 243
- Prolog, 145
- propriedade do corte, 130
- qubit, 298
- raiz quadrada não trivial, 28, 43, 310
- razão de aproximação, 276
- recozimento simulado, 290
- recursão, 160
- rede residual, 200
- rede, 199
- redução, 196, 245
- relação de equivalência, 102
- relação de recorrência, 46
- SAT circuito, *veja* satisfatibilidade
 - satisfatibilidade, 232
 - 2SAT, 101, 235
 - 3SAT, 235, 249, 250, 252
 - backtracking*, 272, 293
 - Fórmula Horn, 144-145, 235
 - max SAT, 266, 295
 - SAT circuito, 260
 - SAT, 250
- seleção, 54
- semi-espaco, 189, 213
- série geométrica 9, 49
- série harmônica, 39
- sistema de criptografia RSA, 33-34, 268
- soluções factíveis, 189
- soma de subconjunto, 242, 255
- subseqüência crescente mais longa, 157
- superposição, 298
 - periódica, 305
- tempo exponencial, 4, 232
- tempo polinomial, 234
- teorema chinês do resto, 42
- teorema corte-mínimo fluxo-máximo, 204
- teorema da dualidade, 208
- teorema de Hall, 230
- teorema de Wilson, 42
- teorema dos números primos de Lagrange, 28
- teorema Master de recorrências, 49
- teorema Master, 49
- teoria de grupos, 27
- teoria dos números, 35
- tese estendida de Church-Turing, 314
- teste de Fermat, 25
- transformada de Fourier quântica, 303
- transformada rápida de Fourier, 57
 - algoritmo, 68
- TSP, *veja* problema do caixeiro-viajante
- valor de circuito, 221
- Volker Strassen, 56
- World Wide Web, 82, 94

