zio diminui o valor efetivo por quilo de sua carga. No problema 0-1, quando consideramos um item para inclusão na mochila, devemos comparar a solução para o subproblema no qual o item é incluído com a solução para o subproblema no qual o item é excluído, antes de podermos fazer a escolha. O problema formulado desse modo ocasiona muitos subproblemas superpostos – um caso legítimo de programação dinâmica e, de fato, a programação dinâmica pode ser usada para resolver o problema 0-1. (Ver Exercício 16.2-2.)

Exercícios

16.2-1
Prove que o problema da mochila fracionária tem a propriedade de escolha gulosa.

16.2-2
Forneça uma solução de programação dinâmica para o problema da mochila 0-1 que seja executado no tempo $O(nW)$, onde n é número de itens e W é o peso máximo dos itens que o ladrão pode pôr em sua mochila.

16.2-3
Suponha que, em um problema de mochila 0-1, a ordem dos itens ordenados por peso crescente seja igual à sua ordem quando eles são ordenados por valor decrescente. Forneça um algoritmo eficiente com o objetivo de encontrar uma solução ótima para essa variante do problema da mochila e mostre que seu algoritmo é correto.

16.2-4
O professor Midas dirige um automóvel do Rio para São Paulo pela Via Dutra. O tanque de combustível de seu carro, quando completo, contém gasolina suficiente para viajar n quilômetros, e o mapa do professor fornece as distâncias entre os postos de gasolina em sua rota. O professor deseja fazer o mínimo possível de paradas para abastecimento ao longo do caminho. Forneça um método eficiente pelo qual o professor Midas possa determinar em quais postos de gasolina deve parar, e mostre que sua estratégia produz uma solução ótima.

16.2-5
Descreva um algoritmo eficiente que, dado um conjunto $\{x_1, x_2, ..., x_n\}$ de pontos na reta real, determine o menor conjunto de intervalos fechados de comprimento unitário que contenha todos os pontos dados. Mostre que seu algoritmo é correto.

16.2-6 ★
Mostre como resolver o problema da mochila fracionária no tempo $O(n)$. Suponha que você tenha uma solução para o Problema 9-2.

16.2-7
Suponha que você tem dois conjuntos A e B, cada um contendo n inteiros positivos. Você tem a possibilidade de reordenar cada conjunto como preferir. Depois da reordenação, seja a_i o i-ésimo elemento do conjunto A, e seja b_i o i-ésimo elemento do conjunto B. Você recebe então uma recompensa $\prod_{i=1}^{n} a_i^{b_i}$. Forneça um algoritmo que maximize sua recompensa. Prove que seu algoritmo maximiza a recompensa e enuncie seu tempo de execução.

16.3 Códigos de Huffman

Os códigos de Huffman constituem uma técnica amplamente utilizada e muito eficiente para comprimir dados, guardando a relação típica de 20 a 90%, dependendo das características dos dados que estão sendo comprimidos. O algoritmo guloso de Huffman utiliza uma tabela das freqüências de ocorrência dos caracteres para elaborar um modo ótimo de representar cada caractere como uma cadeia binária.

Vamos supor que temos um arquivo de dados de 100.000 caracteres que desejamos armazenar em forma compacta. Observamos que os caracteres no arquivo ocorrem com freqüências dadas pela Figura 16.3. Isto é, aparecem somente seis caracteres diferentes, e o caractere a ocorre 45.000 vezes.

Há muitas maneiras de representar um arquivo de informações como esse. Consideramos o problema de projetar um ***código de caracteres binários*** (ou ***código***, para abreviar) no qual cada caractere é representado por uma cadeia binária única. Se usarmos um ***código de comprimento fixo***, precisaremos de 3 bits para representar seis caracteres: a = 000, b = 001, ..., f = 101. Esse método exige 300.000 bits para codificar o arquivo inteiro. Podemos fazer algo melhor?

Um ***código de comprimento variável*** pode funcionar consideravelmente melhor que um código de comprimento fixo, fornecendo palavras de código curtas a caracteres freqüentes e palavras de código longas a caracteres pouco freqüentes. A Figura 16.3 mostra um código desse tipo; aqui, a cadeia de 1 bit 0 representa a, e a cadeia de 4 bits 1100 representa f. Esse código exige

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1.000 = 224.000 \text{ bits}$$

para representar o arquivo, uma economia de aproximadamente 25%. De fato, esse é um código de caracteres ótimo para esse arquivo, como veremos.

Códigos de prefixo

Consideramos aqui apenas códigos nos quais nenhuma palavra de código é também um prefixo de alguma outra palavra de código. Tais códigos são chamados ***códigos de prefixo***.[2] É possível mostrar (embora não o façamos aqui) que a compressão de dados ótima que se pode obter por meio de um código de caracteres sempre pode ser alcançada com um código de prefixo, e assim não há perda de generalidade em restringirmos nossa atenção a códigos de prefixo.

A codificação é sempre simples para qualquer código de caracteres binários; simplesmente concatenamos as palavras de código que representam cada caractere do arquivo. Por exemplo, com o código de prefixo de comprimento variável da Figura 16.3, codificamos o arquivo de 3 caracteres abc como 0 · 101 · 100 = 0101100, onde usamos " · " para denotar a concatenação.

	a	b	c	d	e	f
Freqüência (em milhares)	45	13	12	16	9	5
Palavra de código de comprimento fixo	000	001	010	011	100	101
Palavra de código de comprimento variável	0	101	100	111	1101	1100

FIGURA 16.3 Um problema de codificação de caracteres. Um arquivo de dados de 100.000 caracteres contém apenas os caracteres a-f, com as freqüências indicadas. Se for atribuída a cada caractere uma palavra de código de 3 bits, o arquivo poderá ser codificado em 300.000 bits. Usando-se o código de comprimento variável mostrado, o arquivo poderá ser codificado em 224.000 bits

[2] Talvez "códigos de prefixo livre" fosse um nome melhor, mas a expressão "códigos de prefixo" é padrão na literatura.

Os códigos de prefixo são desejáveis porque simplificam a decodificação. Como nenhuma palavra de código é um prefixo de qualquer outra, a palavra de código que inicia um arquivo codificado não é ambígua. Podemos simplesmente identificar a palavra de código inicial, traduzi-la de volta para o caractere original, removê-la do arquivo codificado e repetir o processo de decodificação no restante do arquivo codificado. Em nosso exemplo, a cadeia 001011101 é analisada unicamente como 0 · 0 · 101 · 1101, que é decodificada como aabe.

O processo de decodificação precisa de uma representação conveniente para o código de prefixo, de forma que a palavra de código inicial possa ser extraída com facilidade. Uma árvore binária cujas folhas são os caracteres dados fornece tal representação. Interpretamos a palavra de código binária para um caractere como o caminho desde a raiz até esse caractere, onde 0 significa "vá para o filho da esquerda" e 1 significa "vá para o filho da direita". A Figura 16.4 mostra as árvores para os dois códigos do nosso exemplo. Observe que elas não são árvores de pesquisa binária, pois as folhas não precisam aparecer em seqüência ordenada e os nós internos não contêm chaves de caracteres.

Um código ótimo para um arquivo é sempre representado por uma árvore binária *cheia*, na qual cada nó que não é uma folha tem dois filhos (ver Exercício 16.3-1). O código de comprimento fixo em nosso exemplo não é ótimo, pois sua árvore, mostrada na Figura 16.4(a), não é uma árvore binária cheia: existem palavras de código começando com 10..., mas nenhuma com 11.... Tendo em vista que agora podemos restringir nossa atenção a árvores binárias completas, dizemos que, se C é o alfabeto do qual os caracteres são obtidos e todas as freqüências de caracteres são positivas, então a árvore para um código de prefixo ótimo tem exatamente $|C|$ folhas, uma para cada letra do alfabeto, e exatamente $|C| - 1$ nós internos (veja o Exercício B.5-3).

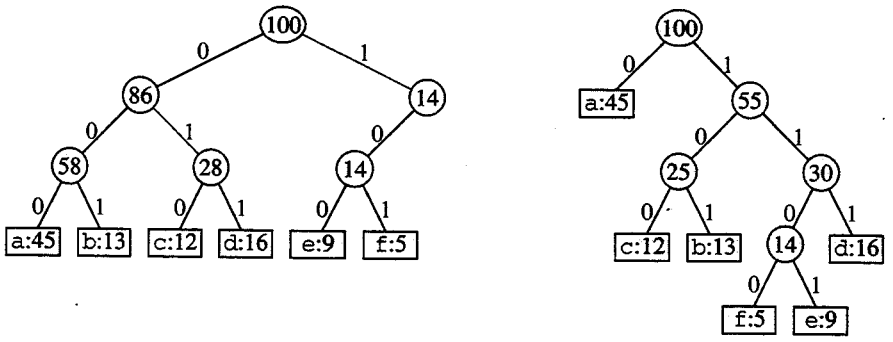

FIGURA 16.4 Árvores correspondentes aos esquemas de codificação na Figura 16.3. Cada folha é identificada com um caractere e sua freqüência de ocorrência. Cada nó interno é identificado com a soma das freqüências das folhas em sua subárvore. (**a**) A árvore correspondente ao código de comprimento fixo a = 000, ..., f = 101. (**b**) A árvore correspondente ao código de prefixo ótimo a = 0, b = 101, ..., f = 1100

Dada uma árvore T correspondente a um código de prefixo, é uma questão simples calcular o número de bits exigidos para codificar um arquivo. Para cada caractere c no alfabeto C, seja $f(c)$ a freqüência de c no arquivo e seja $d_T(c)$ a profundidade de folha de c na árvore. Observe que $d_T(c)$ é também o comprimento da palavra de código para o caractere c. Desse modo, o número de bits exigidos para codificar um arquivo é

$$B(T) = \sum_{c \in C} f(c) d_T(c), \tag{16.5}$$

que definimos como o *custo* da árvore T.

A construção de um código de Huffman

Huffman criou um algoritmo guloso que produz um código de prefixo ótimo chamado ***código de Huffman***. De acordo com nossas observações na Seção 16.2, a prova de sua correção se baseia na propriedade de escolha gulosa e subestrutura ótima. Em vez de demonstrar que essas propriedades são válidas e depois desenvolver pseudocódigo, apresentamos primeiro o pseudocódigo. Isso ajudará a esclarecer como o algoritmo faz escolhas gulosas.

No pseudocódigo a seguir, supomos que C é um conjunto de n caracteres e que cada caractere $c \in C$ é um objeto com uma freqüência definida $f[c]$. O algoritmo constrói de baixo para cima a árvore T correspondente ao código ótimo. Ele começa com um conjunto de $|C|$ folhas e executa uma seqüência de $|C| - 1$ operações de "intercalação" para criar a árvore final. Uma fila de prioridade mínima Q, tendo f como chave, é usada para identificar os dois objetos menos freqüentes a serem intercalados. O resultado da intercalação de dois objetos é um novo objeto cuja freqüência é a soma das freqüências dos dois objetos que foram intercalados.

```
HUFFMAN(C)
1 n ← |C|
2 Q ← C
3 for i ← 1 to n – 1
4     do alocar um novo nó z
5        esquerda[z] ← x ← EXTRACT-MIN(Q)
6        direita[z] ← y ← EXTRACT-MIN(Q)
7        f[z] ← f[x] + f[y]
8        INSERT(Q, z)
9 return EXTRACT-MIN(Q)        ▷ Retornar a raiz da árvore.
```

Em nosso exemplo, o algoritmo de Huffman procede como mostra a Figura 16.5. Tendo em vista que existem 6 letras no alfabeto, o tamanho da fila inicial é $n = 6$, e 5 passos de intercalação são exigidos para construir a árvore. A árvore final representa o código de prefixo ótimo. A palavra de código para uma letra é a seqüência de etiquetas de arestas no caminho da raiz até a letra.

A linha 2 inicializa a fila de prioridades Q com os caracteres em C. O loop **for** nas linhas 3 a 8 extrai repetidamente os dois nós x e y de freqüência mais baixa da fila e os substitui na fila por um novo nó z representando sua intercalação. A freqüência de z é calculada como a soma das freqüências de x e y na linha 7. O nó z tem x como seu filho da esquerda e y como seu filho da direita. (Essa ordem é arbitrária; a troca do filho da esquerda pelo da direita de qualquer nó produz um código diferente do mesmo custo.) Depois de $n - 1$ intercalações, o único nó da esquerda na fila – a raiz da árvore de código – é retornado na linha 9.

A análise do tempo de execução do algoritmo de Huffman supõe que Q é implementada como um heap binário (ver Capítulo 6). Para um conjunto C de n caracteres, a inicialização de Q na linha 2 pode ser executada em tempo $O(n)$ usando-se o procedimento BUILD-MIN-HEAP da Seção 6.3. O loop **for** nas linhas 3 a 8 é executado exatamente $n - 1$ vezes e, como cada operação de heap exige o tempo $O(\lg n)$, o loop contribui com $O(n \lg n)$ para o tempo de execução. Desse modo, o tempo de execução total de HUFFMAN em um conjunto de n caracteres é $O(n \lg n)$.

Correção do algoritmo de Huffman

Para provar que o algoritmo guloso HUFFMAN é correto, mostramos que o problema de determinar um código de prefixo ótimo exibe as propriedades de escolha gulosa e de subestrutura ótima. O próximo lema mostra que a propriedade de escolha gulosa é válida.

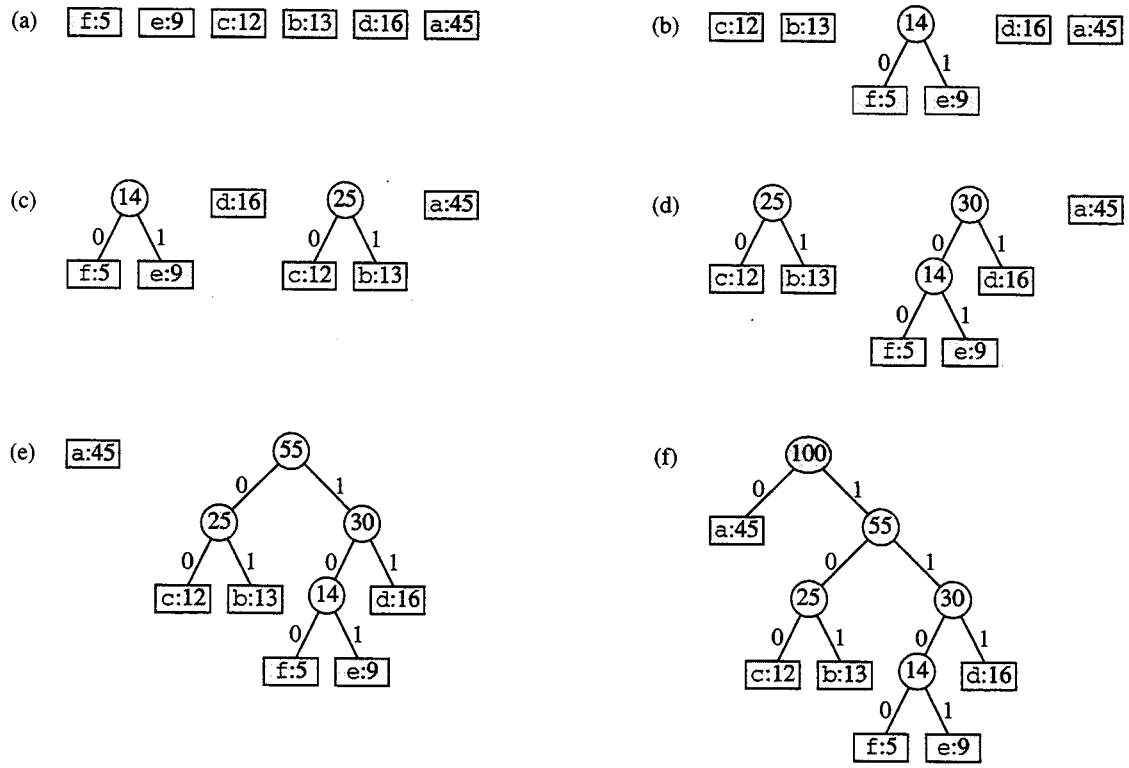

FIGURA 16.5 Os passos do algoritmo de Huffman para as freqüências dadas na Figura 16.3. Cada parte mostra o conteúdo da fila ordenado em ordem crescente por freqüência. Em cada passo, as duas árvores com freqüências mais baixas são intercaladas. As folhas são mostradas como retângulos contendo um caractere e sua freqüência. Nós internos são mostrados como círculos, contendo a soma das freqüências de seus filhos. Uma aresta conectando um nó interno a seus filhos é identificada por 0 se é uma aresta para um filho da esquerda e com 1, se é uma aresta para um filho da direita. A palavra de código para uma letra é a seqüência de etiquetas nas arestas que conectam a raiz à folha correspondente a essa letra. **(a)** O conjunto inicial de $n = 6$ nós, um para cada letra. **(b)–(e)** Estágios intermediários. **(f)** A árvore final

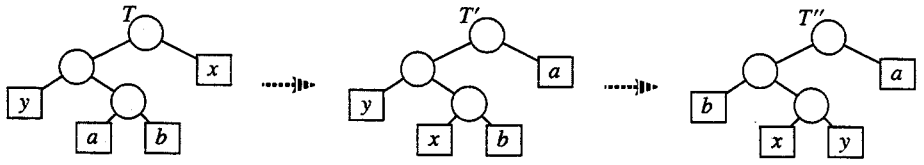

FIGURA 16.6 Uma ilustração do passo chave na prova do Lema 16.2. Na árvore ótima T, as folhas a e b são duas das folhas mais profundas e são irmãs. As folhas x e y são as duas folhas que o algoritmo de Huffman intercala primeiro; elas aparecem em posições arbitrárias em T. As folhas a e x são trocadas para se obter a árvore T'. Em seguida, as folhas b e y são trocadas para se obter a árvore T''. Como cada troca não aumenta o custo, a árvore resultante T'' também é uma árvore ótima

Lema 16.2

Seja C um alfabeto em que cada caractere $c \in C$ tem freqüência $f[c]$. Sejam x e y dois caracteres em C que têm as freqüências mais baixas. Então, existe um código de prefixo ótimo para C no qual as palavras de código para x e y têm o mesmo comprimento e diferem apenas no último bit.

Prova A idéia da prova é tomar a árvore T representando um código de prefixo ótimo arbitrário e modificá-la para criar uma árvore representando outro código de prefixo ótimo, tal que os caracteres x e y apareçam como folhas irmãs de profundidade máxima na nova árvore. Se pudermos fazer isso, então suas palavras de código terão o mesmo comprimento e serão diferentes apenas no último bit.

Sejam a e b dois caracteres que representam folhas irmãs de profundidade máxima em T. Sem perda de generalidade, supomos $f[a] \leq f[b]$ e $f[x] \leq f[y]$. Tendo em vista que $f[x]$ e $f[y]$ são as duas freqüências mais baixas de folhas, nessa ordem, e $f[a]$ e $f[B]$ são duas freqüências arbitrárias, nessa ordem, temos $f[x] \leq f[a]$ e $f[y] \leq f[b]$. Como mostra a Figura 16.6, permutamos as posições em T de a e x para produzir uma árvore T', e então permutamos as posições em T' de b e y para produzir uma árvore T''. Pela equação (16.5), a diferença de custo entre T e T'' é

$$\begin{aligned} B(T) - B(T'') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T''}(c) , \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_{T''}(x) - f[a] d_{T''}(a) \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_T(a) - f[a] d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \\ &\geq 0 , \end{aligned}$$

porque tanto $f[a] - f[x]$ quanto $d_T(a) - d_T(x)$ são não negativas. Mais especificamente, $f[a] - f[x]$ é não negativa porque x é uma folha de freqüência mínima, e $d_T(a) - d_T(x)$ é não negativa porque a é uma folha de profundidade máxima em T. De modo semelhante, como a troca de y e b não aumenta o custo, $B(T') - B(T'')$ é não negativa. Então, $B(T') \leq B(T)$ e, como T é ótima, $B(T) \leq B(T'')$, que implica $B(T'') = B(T)$. Desse modo, T'' é uma árvore ótima em que x e y aparecem como folhas irmãs de profundidade máxima, e disso decorre o lema. ■

O Lema 16.2 implica que o processo de construir uma árvore ótima por intercalações pode, sem perda de generalidade, começar com a escolha gulosa de intercalar esses dois caracteres de freqüência mais baixa. Por que essa é uma escolha gulosa? Podemos ver o custo de uma intercalação única como a soma das freqüências dos dois itens que estão sendo intercalados. O Exercício 16.3-3 mostra que o custo total da árvore construída é a soma dos custos de suas intercalações. De todas as intercalações possíveis em cada passo, HUFFMAN escolhe aquela que incorre no menor custo.

O próximo lema mostra que o problema de construir códigos de prefixo ótimos tem a propriedade de subestrutura ótima.

Lema 16.3

Seja C um dado alfabeto com freqüência $f[c]$ definida para cada caractere $c \in C$. Sejam dois caracteres x e y em C com freqüência mínima. Seja C' o alfabeto C com os caracteres x, y removidos e o (novo) caractere z adicionado, de forma que $C' = C - \{x, y\} \cup \{z\}$; defina f para C' como para C, exceto pelo fato de que $f[z] = f[x] + f[y]$. Seja T' qualquer árvore representando um código de prefixo ótimo para o alfabeto C'. Então a árvore T, obtida a partir de T' pela substituição do nó de folha para z por um nó interno que tem x e y como filhos, representa um código de prefixo ótimo para o alfabeto C.

Prova Primeiro, mostramos que o custo $B(T)$ da árvore T pode ser expresso em termos do custo $B(T')$ da árvore T', considerando-se os custos componentes na equação (16.5). Para cada $c \in C - \{x, y\}$, temos $d_T(c) = d_{T'}(c)$, e portanto $f[c] d_T(c) = f[c] d_{T'}(c)$. Como $d_T(x) = d_T(y) = d_{T'}(z) + 1$, temos

$$\begin{aligned} f[x] d_T(x) + f[y] d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= (f[z] d_{T'}(z) + f[x]) + f[y]) , \end{aligned}$$

da qual concluímos que

$$B(T) = B(T') + f[x] + f[y]$$

ou, de modo equivalente,

$$B(T') = B(T) - f[x] - f[y] .$$

Agora, provamos o lema por contradição. Suponha que T não representa um código de prefixo ótimo para C. Então, existe uma árvore T'' tal que $B(T'') < B(T)$. Sem perda de generalidade (pelo Lema 16.2), T'' tem x e y como irmãos. Seja T''' a árvore T'' com o pai comum de x e y substituído por uma folha z com freqüência $f[z] = f[x] + f[y]$. Assim

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T') , \end{aligned}$$

o que produz uma contradição para a hipótese de que T' representa um código de prefixo ótimo para C'. Desse modo, T deve representar um código de prefixo ótimo para o alfabeto C. ■

Teorema 16.4
O procedimento HUFFMAN produz um código de prefixo ótimo.

Prova Imediata, a partir dos Lemas 16.2 e 16.3. ■

Exercícios

16.3-1
Prove que uma árvore binária que não é completa não pode corresponder a um código de prefixo ótimo.

16.3-2
O que é um código de Huffman ótimo para o conjunto de freqüências a seguir, baseado nos primeiros 8 números de Fibonacci?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Você pode generalizar sua resposta para encontrar o código ótimo quando as freqüências forem os primeiros n números de Fibonacci?

16.3-3
Prove que o custo total de uma árvore para um código também pode ser calculado como a soma sobre todos os nós internos das freqüências combinadas dos dois filhos do nó.

16.3-4
Prove que, se ordenarmos os caracteres em um alfabeto de modo que suas freqüências sejam monotonicamente decrescentes, então existe um código ótimo cujos comprimentos de palavras de código são monotonicamente crescentes.

16.3-5
Suponha que temos um código de prefixo ótimo em um conjunto de caracteres $C = \{0, 1, ..., n-1\}$ e desejamos transmitir esse código usando o menor número de bits possível. Mostre como representar qualquer código de prefixo ótimo em C usando apenas $2n - 1 + n\lceil \lg n \rceil$ bits. (*Sugestão*: Use $2n - 1$ bits para especificar a estrutura da árvore, como se descobre por um percurso da árvore.)

16.3-6
Generalize o algoritmo de Huffman para palavras de código ternárias (isto é, palavras de código que utilizam os símbolos 0, 1 e 2), e prove que ele produz códigos ternários ótimos.

16.3-7
Suponha que um arquivo de dados contém uma seqüência de caracteres de 8 bits tais que todos os 256 caracteres são quase comuns: a freqüência máxima de caracteres é menor que duas vezes a freqüência mínima de caracteres. Prove que a codificação de Huffman nesse caso não é mais eficiente que o uso de um código de comprimento fixo de 8 bits comum.7

16.3-8
Mostre que nenhum esquema de compressão pode esperar comprimir um arquivo de caracteres de 8 bits escolhidos aleatoriamente por sequer um único bit. (*Sugestão*: Compare o número de arquivos com o número de arquivos codificados possíveis.)

★ 16.4 Fundamentos teóricos de métodos gulosos

Existe uma bela teoria sobre algoritmos gulosos, a qual descreveremos nesta seção. Essa teoria é útil para se determinar quando o método guloso produz soluções ótimas. Ela envolve estruturas combinatórias conhecidas como "matróides". Embora essa teoria não cubra todos os casos aos quais um método guloso se aplica (por exemplo, ela não abrange o problema de seleção de atividades da Seção 16.1 ou o problema de codificação de Huffman da Seção 16.3), ela envolve muitos casos de interesse prático. Além disso, essa teoria está sendo rapidamente desenvolvida e estendida para cobrir muitas outras aplicações; consulte as notas no final deste capítulo para referência.

Matróides

Um ***matróide*** é um par ordenado $M = (S, \ell)$ que satisfaz às condições a seguir.

1. S é um conjunto finito não vazio.
2. ℓ é uma família não vazia de subconjuntos de S, chamados subconjuntos ***independentes*** de S, tais que, se $B \in \ell$ e $A \subseteq B$, então $A \in \ell$. Dizemos que ℓ é ***hereditário*** se ele satisfaz a essa propriedade. Observe que o conjunto vazio $\emptyset$ é necessariamente um membro de ℓ.
3. Se $A \in \ell, B \in \ell$ e $|A| < |B|$, então existe algum elemento $x \in B - A$ tal que $A \cup \{x\} \in \ell$. Dizemos que M satisfaz à ***propriedade de troca***.

A palavra "matróide" se deve a Hassler Whitney. Ele estava estudando ***matróides de matrícula***, nos quais os elementos de S são as linhas de uma dada matriz, e um conjunto de linhas é independente se elas são linearmente independentes no sentido usual. É fácil mostrar que essa estrutura define um matróide (ver Exercício 16.4-2).

Como outro exemplo de matróides, considere o ***matróide gráfico*** $M_G = (S_G, \ell_G)$, definido em termos de um dado grafo não orientado $G = (V, E)$ como a seguir.

- O conjunto S_G é definido como E, o conjunto de arestas de G.
- Se A é um subconjunto de E, então $A \in \ell_G$ se e somente se A é acíclico. Ou seja, um conjunto de arestas A é independente se e somente se o subgrafo $G_A = (V, A)$ forma uma floresta.

O matróide gráfico M_G está intimamente relacionado com o problema da árvore de amplitude mínima, apresentado em detalhes no Capítulo 23.

Teorema 16.5
Se G é um grafo não orientado, então $M_G = (S_G, \ell_G)$ é um matróide.