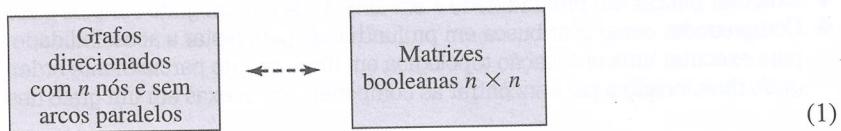


determinar qual desses ciclos corresponde ao percurso de menor distância. Lembre-se de que um ciclo é um caminho que termina onde começou e que não passa por nenhum dos outros nós mais de uma vez; assim, um tal ciclo daria um caminho eficiente para um vendedor visitar todas as cidades em sua região de vendas apenas uma vez e terminar em casa.

A Seção 6.3 fornece soluções algorítmicas em um grafo simples conexo para o problema de encontrar o caminho mínimo entre dois nós e para o problema de minimizar o número de arcos usados para conectar todos os nós. A Seção 6.4 discute algoritmos para percorrer grafos simples — “visitando” todos os nós de algum modo sistemático.

SEÇÃO 6.1 GRAFOS DIRECIONADOS E RELAÇÕES BINÁRIAS; O ALGORITMO DE WARSHALL

Restringiremos nossa atenção nesta seção a grafos direcionados sem arcos paralelos (e sem peso). (Em um grafo direcionado, dois arcos do nó a para o nó b seriam paralelos, mas um arco do nó a para b e outro do nó b para a não são paralelos.) Considere a matriz de adjacência do grafo (supondo uma ordem arbitrária dos n nós, o que sempre supomos ao discutir a matriz de adjacência de um grafo). Essa é uma matriz $n \times n$ que não é necessariamente simétrica. Além disso, como o grafo não tem arcos paralelos, a matriz de adjacência vai ser uma matriz booleana, isto é, uma matriz onde todos os elementos são iguais a 0 ou a 1. Reciprocamente, dada uma matriz booleana $n \times n$, podemos reconstruir o grafo direcionado representado por essa matriz, que não terá arcos paralelos. Dessa forma, temos uma correspondência um para um, que podemos ilustrar como



Vamos ver, agora, como as relações binárias entram nessa correspondência.

GRAFOS DIRECIONADOS E RELAÇÕES BINÁRIAS

Suponha que G é um grafo direcionado com n nós e sem arcos paralelos. Seja N o conjunto de nós. Se (n_i, n_j) é um par ordenado de nós, então, ou existe ou não existe um arco de n_i para n_j . Podemos usar essa propriedade para definir uma relação binária no conjunto N :

$$n_i \rho n_j \leftrightarrow \text{existe um arco em } G \text{ de } n_i \text{ para } n_j.$$

Essa relação é a **relação de adjacência** do grafo.

Para o grafo direcionado na Fig. 6.1, a relação de adjacência é $\{(1, 2), (1, 3), (3, 3), (4, 1), (4, 2), (4, 3)\}$.

❖ EXEMPLO 1

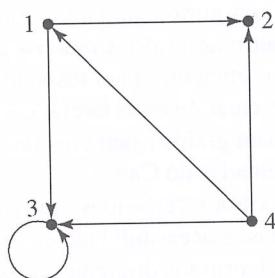


Fig. 6.1

Reciprocamente, se ρ é uma relação binária em um conjunto N , podemos definir um grafo direcionado G tendo N como o conjunto de nós e tendo um arco de n_i para n_j se, e somente se, $n_i \rho n_j$. G não vai ter arcos paralelos.

♦ EXEMPLO 2

Para o conjunto $N = \{1, 2, 3, 4\}$ e a relação binária $\{(1, 4), (2, 3), (2, 4), (4, 1)\}$ em N , obtemos o grafo direcionado associado ilustrado na Fig. 6.2.

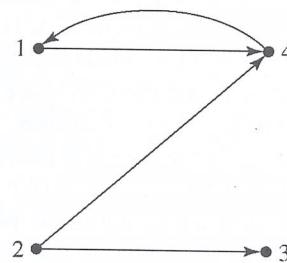
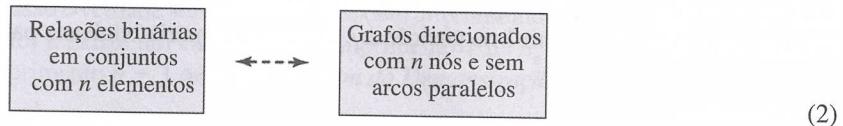
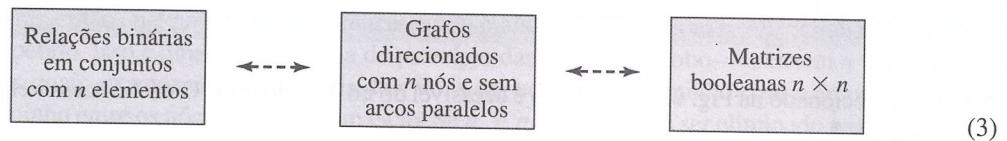


Fig. 6.2

Temos, agora, uma outra correspondência um para um:



É claro que uma correspondência um para um é o mesmo que uma bijeção. Se fizermos a composição das bijeções em (1) e (2), o resultado é uma bijeção que nos dá uma correspondência um para um entre relações binárias e matrizes. Dessa forma, temos três conjuntos equivalentes:



Um elemento em qualquer um dos três conjuntos tem representações correspondentes nos outros dois.

PROBLEMA
PRÁTICO 1

Dê a coleção de pares ordenados na relação de adjacência para a matriz booleana a seguir; desenhe, também, o grafo direcionado correspondente:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Lembre-se das propriedades de reflexividade, simetria, anti-simetria e transitividade de uma relação binária em um conjunto, que estudamos no Cap. 4. Se uma relação binária em um conjunto N tem uma determinada propriedade, isso se refletirá no grafo e na matriz booleana correspondente. Reciprocamente, determinadas características de um grafo direcionado ou de uma matriz booleana implicam certas propriedades das relações de adjacência correspondentes.

♦ EXEMPLO 3

Se ρ é uma relação reflexiva em um conjunto N , então, para cada $n_i \in N$, $n_i \rho n_i$. O grafo direcionado correspondente terá um laço em cada nó e a matriz booleana correspondente terá todos os elementos na diagonal principal iguais a 1.

PROBLEMA
PRÁTICO 2

Explique por que a relação binária associada ao grafo direcionado na Fig. 5.13 não é anti-simétrica.

Representamos ordens parcias no Cap. 4 usando diagramas de Hasse. Como essa representação difere da de um grafo direcionado? O diagrama de Hasse é uma simplificação da representação como grafo

direcionado. Suponha que G é um grafo direcionado que representa uma ordem parcial. Como uma ordem parcial é reflexiva, G tem um laço em cada nó. Podemos eliminar esses laços no diagrama de Hasse sem perda de informação, já que sabemos que cada nó tem um laço, isto é, cada nó está relacionado consigo mesmo. Como uma ordem parcial é transitiva, se $a \rho b$ e $b \rho c$, então $a \rho c$. O grafo direcionado teria um arco de a para b , um de b para c e um de a para c . No diagrama de Hasse podemos eliminar o arco de a para c sem perda de informação se mantivermos a transitividade em mente. Finalmente, o diagrama de Hasse não é um grafo direcionado mas impusemos a condição que, se a é um predecessor imediato de b , então o nó a aparece debaixo do nó b no diagrama de Hasse. Podemos, assim, obter um grafo direcionado do diagrama de Hasse fazendo com que todos os arcos apontem para cima. A anti-simetria impede qualquer conflito em potencial, onde o nó a deveria estar debaixo do nó b e b deveria estar debaixo de a .

No Cap. 4 definimos, também, operações em conjuntos que podem ser executadas em duas relações binárias ρ e σ em um conjunto N , $\rho \cup \sigma$ e $\rho \cap \sigma$. A relação $\rho \cup \sigma$ é a união dos pares ordenados em ρ ou σ , enquanto que $\rho \cap \sigma$ é a intersecção dos pares ordenados em ρ e em σ . Sejam \mathbf{R} e \mathbf{S} as matrizes booleanas associadas a ρ e σ , respectivamente. A matriz booleana associada a $\rho \cup \sigma$ vai ter 1 na posição i, j se, e somente se, \mathbf{R} tem 1 na posição i, j ou \mathbf{S} tem 1 na posição i, j . Cada elemento na matriz booleana associada a $\rho \cup \sigma$ é, portanto, o máximo entre os dois elementos correspondentes nas matrizes \mathbf{R} e \mathbf{S} , de modo que a matriz booleana associada a $\rho \cup \sigma$ é $\mathbf{R} \vee \mathbf{S}$ (veja a discussão sobre operações booleanas de matrizes booleanas na Seção 4.5). Analogamente, a matriz booleana associada a $\rho \cap \sigma$ vai ter 1 na posição i, j se, e somente se, ambas \mathbf{R} e \mathbf{S} têm 1 na posição i, j . Portanto, a matriz booleana para $\rho \cap \sigma$ é $\mathbf{R} \wedge \mathbf{S}$.

ACESSIBILIDADE

A propriedade de “acessibilidade” tem uma interpretação interessante em cada uma das três formas equivalentes em (3) — grafo direcionado, relação de adjacência e matriz de adjacência. Já definimos acessibilidade para grafos direcionados na Seção 5.1, mas enunciaremos novamente essa definição.

Definição: Nô Acessível Em um grafo direcionado, o nó n_j é **acessível** do nó n_i se existe um caminho de n_i para n_j .

No grafo direcionado da Fig. 6.2, o nó 3 não é acessível do nó 4 ou do nó 1. O nó 1 é acessível do nó 2 pelo caminho 2–4–1.

EXEMPLO 4

Em um sistema modelado por um grafo direcionado (um diagrama de fluxo de dados, por exemplo) com um “nó inicial”, qualquer nó que não é acessível do nó inicial nunca pode afetar o sistema e, portanto, pode ser eliminado. Se o grafo direcionado representa algo do tipo de rotas aéreas ou caminhos de comunicação em uma rede de computadores, não seria desejável ter algum nó que não fosse acessível de algum outro. Assim, a habilidade de testar acessibilidade tem muitas aplicações práticas.

A matriz de adjacência \mathbf{A} de um grafo direcionado G com n nós e sem arcos paralelos vai ter 1 na posição i, j se existe um arco de n_i para n_j . Esse seria um caminho de comprimento 1 de n_i para n_j . Portanto, a matriz de adjacência por si própria já nos dá uma forma limitada de acessibilidade, por caminhos de comprimento 1. Entretanto, vamos efetuar uma multiplicação booleana $\mathbf{A} \times \mathbf{A}$. Vamos denotar esse produto por $\mathbf{A}^{(2)}$ para distingui-lo de \mathbf{A}^2 , o resultado de $\mathbf{A} \cdot \mathbf{A}$ usando a multiplicação usual de matrizes. Lembrando a definição de multiplicação booleana da Seção 4.5, o elemento i, j de $\mathbf{A}^{(2)}$ é dado por

$$\mathbf{A}^{(2)}[i, j] = \bigvee_{k=1}^n (a_{ik} \wedge a_{kj}) \quad (4)$$

Se um termo do tipo $a_{ik} \wedge a_{kj}$ nessa soma for 0, então $a_{ik} = 0$ ou $a_{kj} = 0$ (ou ambos), e não existe caminho de comprimento 1 de n_i para n_k ou não existe caminho de comprimento 1 de n_k para n_j (ou ambos). Logo, não existe caminho de comprimento 2 de n_i para n_j passando pelo nó n_k . Se $a_{ik} \wedge a_{kj}$ não for 0, então $a_{ik} = 1$ e $a_{kj} = 1$. Portanto existe um caminho de comprimento 1 de n_i para n_k e existe um caminho de comprimento 1 de n_k para n_j , de modo que existe um caminho de comprimento 2 de n_i para n_j passando por n_k . Vai existir um caminho de comprimento 2 de n_i para n_j se, e somente se, existir um caminho de comprimento 2 passando por pelo menos um dos nós de 1 a n , isto é, se, e somente se, pelo menos um dos termos na soma em (4) é 1 e, portanto, $\mathbf{A}^{(2)}[i, j] = 1$. Assim, os elementos em $\mathbf{A}^{(2)}$ nos informam sobre acessibilidade por caminhos de comprimento 2.

LEMBRETE:
Para calcular $\mathbf{A}^{(2)}[i, j]$, escreva duas cópias de \mathbf{A} lado a lado. Mova um dedo da mão esquerda ao longo da linha i da cópia à esquerda e, ao mesmo tempo, move um dedo da mão direita ao longo da coluna j da cópia da direita. O valor é 1 se, e somente se, ambos os dedos encontram 1 ao mesmo tempo.

PROBLEMA PRÁTICO 3

Encontre \mathbf{A} para o grafo da Fig. 6.2 e calcule $\mathbf{A}^{(2)}$. O que o elemento 2, 1 indica?

A matriz $\mathbf{A}^{(2)}$ indica a presença ou ausência de caminhos de comprimento 2. Poderíamos inferir que esse resultado é válido para potências arbitrárias e comprimentos arbitrários.

Teorema sobre Matrizes Booleanas de Adjacência e Acessibilidade Se \mathbf{A} é a matriz booleana de adjacência de um grafo direcionado G com n nós e sem arcos paralelos, então $\mathbf{A}^{(m)}[i, j] = 1$ se, e somente se, existe um caminho de comprimento m do nó n_i para o nó n_j .

Demonstração: Uma demonstração por indução em m parece adequada. Já mostramos que o resultado é verdade para $m = 1$ (e $m = 2$). Suponha que $\mathbf{A}^{(p)}[i, j] = 1$ se, e somente se, existe um caminho de comprimento p de n_i para n_j . Sabemos que

$$\mathbf{A}^{(p+1)}[i, j] = \bigvee_{k=1}^n (\mathbf{A}^{(p)}[i, k] \wedge a_{kj})$$

Essa expressão vai ser 1 se, e somente se, pelo menos um dos termos, por exemplo $\mathbf{A}^{(p)}[i, q] \wedge a_{qj} = 1$, ou seja, $\mathbf{A}^{(p)}[i, q] = 1$ e $a_{qj} = 1$. Isso é verdade se, e somente se, existe um caminho de comprimento p de n_i para n_q (pela hipótese de indução) e existe um caminho de comprimento 1 de n_q para n_j , o que significa que existe um caminho de comprimento $p + 1$ de n_i para n_j . *Fim da Demonstração.*

PROBLEMA PRÁTICO 4

Do grafo da Fig. 6.2, qual você espera que seja o valor do elemento 2,1 em $\mathbf{A}^{(4)}$? Calcule $\mathbf{A}^{(4)}$ e verifique esse valor.

Se o nó n_j é acessível do nó n_i , o é ao longo de algum caminho. A existência de um tal caminho é indicada por um elemento 1 na posição i, j de algumas das matrizes $\mathbf{A}, \mathbf{A}^{(2)}, \mathbf{A}^{(3)}$, etc., mas não podemos calcular uma infinidade de produtos matriciais. Felizmente, existe um limite de até onde temos que ir. Se o grafo tem n nós, então qualquer caminho com n arcos ou mais, e, portanto, com $n + 1$ nós ou mais, tem que ter um nó repetido. Isso é uma consequência do princípio das casas de pombo — existem n “caixas” (nós distintos) nas quais estamos colocando mais de n itens (os nós em um caminho com n arcos ou mais). A seção do caminho entre os nós repetidos é um ciclo. Se $n_i \neq n_j$, o ciclo pode ser eliminado para se obter um caminho mais curto; então, se existe um caminho de n_i para n_j , vai existir um tal caminho de comprimento no máximo $n - 1$. Se $n_i = n_j$, então o ciclo poderia ser todo o caminho de n_i a n_i com comprimento máximo n ; embora possamos eliminar esse ciclo (observando que podemos considerar qualquer nó como sendo acessível de si mesmo), vamos conservá-lo para mostrar que existe um caminho não trivial de n_i para n_i .

Em consequência, independente de $n_i = n_j$ ou $n_i \neq n_j$, não precisamos nunca procurar um caminho de n_i a n_j de comprimento maior do que n . Logo, para determinar acessibilidade, precisamos apenas considerar os elementos i, j das matrizes $\mathbf{A}, \mathbf{A}^{(2)}, \dots, \mathbf{A}^{(n)}$. Como alternativa, podemos definir a **matriz R de acessibilidade** por

$$\mathbf{R} = \mathbf{A} \vee \mathbf{A}^{(2)} \vee \dots \vee \mathbf{A}^{(n)}$$

Então, n_j é acessível de n_i se, e somente se, o elemento i, j em \mathbf{R} é positivo.

Já vimos como a acessibilidade em um grafo pode ser expressa em termos da matriz de adjacência. E como representar a acessibilidade em termos da relação de adjacência associada ao grafo?

Se ρ é a relação de adjacência de um grafo G , vamos denotar por ρ^R a relação binária de acessibilidade, isto é, $(n_i, n_j) \in \rho^R$ exatamente quando existe um caminho em G de n_i para n_j . Podemos mostrar, então, que ρ^R é o fecho transitivo de ρ . Lembre, da definição de fecho de uma relação, que o fecho transitivo de ρ é uma relação transitiva que contém ρ e que está contida em qualquer relação transitiva contendo ρ .

Para ver que ρ^R é transitiva, suponha que (n_i, n_j) e (n_j, n_k) pertencem a ρ^R . Então existe um caminho em G de n_i para n_j e existe um caminho em G de n_j para n_k . Portanto, existe um caminho de n_i para n_k e (n_i, n_k) pertence a ρ^R . Para ver que ρ^R contém ρ , suponha que (n_i, n_j) pertence a ρ . Então existe um arco de n_i para n_j em G , o que significa que existe um caminho de comprimento 1 de n_i para n_j e $(n_i, n_j) \in \rho^R$. Finalmente, suponha que σ é uma relação transitiva no conjunto de nós de G que inclui ρ e suponha que $(n_i, n_j) \in \rho^R$. Isso significa que existe um caminho de n_i para n_j usando, por exemplo, os nós $n_i, n_x, \dots, n_w, n_j$. Então existe um arco de cada nó nesse caminho para o próximo e todos os pares ordenados $(n_i, n_x), (n_x, n_y), \dots, (n_w, n_j)$ pertencem a ρ , logo, pertencem a σ . Como σ é transitiva, (n_i, n_j) pertence a σ e ρ^R é um subconjunto de σ . Portanto, ρ^R é o fecho transitivo de ρ .

Resumindo, a correspondência entre as três representações equivalentes de relação de adjacência ρ , grafo direcionado G e matriz de adjacência A é dada por

$$(n_i, n_j) \text{ pertence ao fecho transitivo de } \rho \Leftrightarrow n_j \text{ é acessível de } n_i \text{ em } G \Leftrightarrow R[i, j] = 1, \text{ onde } R = A \vee A^{(2)} \vee \dots \vee A^{(n)}$$

Seja G o grafo direcionado na Fig. 6.3; G tem 5 nós.

♦ EXEMPLO 5

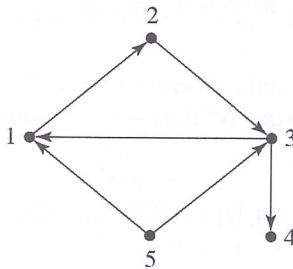


Fig. 6.3

A matriz de adjacência A de G é

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

A relação de adjacência ρ é $\rho = \{(1, 2), (2, 3), (3, 1), (3, 4), (5, 1), (5, 3)\}$.

As potências sucessivas de A são

$$A^{(2)} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \quad A^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

$$A^{(4)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix} \quad A^{(5)} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

LEMBRETE:
Não tente calcular os produtos matriciais sem escrever as duas matrizes lado a lado; você fará, certamente, algum erro.

Essas matrizes indicam, por exemplo, que existe um caminho de comprimento 2 de 2 para 1, pois $A^{(2)}[2, 1] = 1$ (o caminho é 2–3–1), e que existe um caminho de comprimento 4 de 5 para 3, já que $A^{(4)}[5, 3] = 1$ (o caminho é 5–3–1–2–3), mas que não existe caminho de comprimento 3 de 1 para 3, uma vez que $A^{(3)}[1, 3] = 0$.

A matriz de acessibilidade R é a soma booleana de A , $A^{(2)}$, $A^{(3)}$, $A^{(4)}$ e $A^{(5)}$:

$$R = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Os elementos iguais a 1 em \mathbf{R} indicam que existem caminhos em G dos nós 1, 2, 3 e 5 para todos os outros nós, com exceção do 5, mas que não existem caminhos do nó 4 para nenhum nó, o que pode ser confirmado observando-se a Fig. 6.3.

Já vimos que os elementos iguais a 1 em \mathbf{R} marcam os pares ordenados de nós que pertencem ao fecho transitivo de ρ . Portanto, o fecho transitivo é o seguinte conjunto de pares ordenados:

$$\{(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4), \\(3, 1), (3, 2), (3, 3), (3, 4), (5, 1), (5, 2), (5, 3), (5, 4)\}$$

Começando com ρ e seguindo o procedimento *ad hoc* descrito no Cap. 4 para se encontrar o fecho transitivo de uma relação, vemos que, para obter transitividade, precisamos primeiro acrescentar os pares $(1, 3)$, $(2, 1)$, $(2, 4)$, $(3, 2)$, $(5, 2)$ e $(5, 4)$. Revendo o novo conjunto, vemos que precisamos acrescentar também $(1, 1)$, $(1, 4)$, $(2, 2)$ e $(3, 3)$. A coleção resultante de pares ordenados é transitiva (e coincide com a obtida anteriormente). \diamond

PROBLEMA PRÁTICO 5

Calcule \mathbf{R} para o grafo direcionado da Fig. 6.2. Qual a informação dada pela coluna 2? \diamond

No Cap. 4, prometemos um algoritmo melhor para encontrar o fecho transitivo de uma relação. Aqui está: escreva a relação binária na forma de matriz de adjacência e calcule

$$\mathbf{R} = \mathbf{A} \vee \mathbf{A}^{(2)} \vee \dots \vee \mathbf{A}^{(n)}$$

Quanto trabalho é necessário para executar esse algoritmo? A expressão para \mathbf{R} indica que precisamos fazer as operações matriciais booleanas, mas essas operações matriciais necessitam, por sua vez, das operações booleanas **e** e **ou** nos elementos das matrizes. Usaremos, então, as operações booleanas **e** e **ou** para medir a quantidade de trabalho. Na Seção 4.5, observamos que a multiplicação usual de duas matrizes $n \times n$ precisa de $\Theta(n^3)$ multiplicações e somas; por um argumento semelhante, a multiplicação booleana de duas matrizes booleanas $n \times n$ necessita de $\Theta(n^3)$ operações booleanas **e/ou**. O algoritmo para calcular \mathbf{R} precisa de $n - 1$ multiplicações matriciais booleanas (para encontrar os produtos $\mathbf{A}^{(2)}, \mathbf{A}^{(3)}, \dots, \mathbf{A}^{(n)}$). Para calcular $n - 1$ desses produtos precisamos de $(n - 1)\Theta(n^3) = \Theta(n^4)$ operações booleanas. Para calcular $\mathbf{C} \vee \mathbf{D}$, onde \mathbf{C} e \mathbf{D} são duas matrizes booleanas $n \times n$, são necessárias n^2 operações booleanas **ou**. Para calcular \mathbf{R} , são necessárias $n - 1$ dessas operações, logo são executadas $(n - 1)n^2 = \Theta(n^3)$ operações **ou**. A quantidade total de trabalho é $\Theta(n^4) + \Theta(n^3) = \Theta(n^4)$.

Vamos discutir, à seguir, um algoritmo mais eficiente para calcular o fecho transitivo de uma relação (ou a matriz de acessibilidade de um grafo).

ALGORITMO DE WARSHALL

Para um grafo G com n nós, o algoritmo de Warshall calcula uma seqüência de $n + 1$ matrizes $\mathbf{M}_0, \mathbf{M}_1, \mathbf{M}_2, \dots, \mathbf{M}_n$. Para cada k , $0 \leq k \leq n$, $\mathbf{M}_k[i, j] = 1$ se, e somente se, existe um caminho em G de n_i para n_j cujos nós interiores (isso é, nós que não são extremidades do caminho) pertencem apenas ao conjunto de nós $\{n_1, n_2, \dots, n_k\}$.

Vamos examinar as “condições limítrofes”. Quando $k = 0$, o conjunto $\{n_1, n_2, \dots, n_0\}$ é o conjunto vazio, logo $\mathbf{M}_0[i, j] = 1$ se, e somente se, existe um caminho de n_i para n_j cujos nós interiores pertencem ao conjunto vazio, isto é, não existem nós interiores. O caminho de n_i para n_j tem que consistir apenas das extremidades e um arco, ou seja, n_i e n_j são nós adjacentes. Logo, $\mathbf{M}_0 = \mathbf{A}$. A outra condição limítrofe ocorre quando $k = n$. Então, o conjunto $\{n_1, n_2, \dots, n_n\}$ consiste em todos os nós de G , logo não existe, de fato, restrição alguma sobre os nós interiores do caminho e $\mathbf{M}_n[i, j] = 1$ se, e somente se, existe um caminho de n_i para n_j , o que significa que $\mathbf{M}_n = \mathbf{R}$.

Portanto, o algoritmo de Warshall começa com $\mathbf{A} = \mathbf{M}_0$ e calcula, sucessivamente, $\mathbf{M}_1, \mathbf{M}_2, \dots, \mathbf{M}_n = \mathbf{R}$. Esses cálculos podem ser definidos por indução. A base da indução é fazer $\mathbf{M}_0 = \mathbf{A}$. Suponha, agora, que \mathbf{M}_k foi calculado e vamos considerar como calcular \mathbf{M}_{k+1} ou, especificamente, $\mathbf{M}_{k+1}[i, j]$. Temos que $\mathbf{M}_{k+1}[i, j] = 1$ se, e somente se, existe um caminho de n_i para n_j cujos nós interiores pertencem ao conjunto $\{n_1, n_2, \dots, n_{k+1}\}$. Isso pode acontecer de duas maneiras:

1. Todos os nós interiores pertencem a $\{n_1, n_2, \dots, n_k\}$ e, nesse caso, $\mathbf{M}_k[i, j] = 1$. Portanto, devemos repetir, em \mathbf{M}_{k+1} , todos os elementos que são iguais a 1 em \mathbf{M}_k .
2. O nó n_{k+1} é um nó interior. Podemos supor que o nó n_{k+1} aparece apenas uma vez como nó interior, já que os ciclos podem ser eliminados do caminho. Então, tem que existir um caminho de n_i para n_{k+1} cujos nós interiores pertencem a $\{n_1, n_2, \dots, n_k\}$ e um caminho de n_{k+1} para n_j cujos nós interiores pertencem a

$\{n_1, n_2, \dots, n_k\}$. Isso significa que $M_k[i, k+1] = 1$ e $M_k[k+1, j] = 1$, o que significa que $M_k[i, k+1] \wedge M_k[k+1, j] = 1$; essa condição pode ser testada, uma vez que estamos supondo que M_k já foi calculada.

Na versão a seguir do algoritmo de Warshall, em pseudocódigo, o valor inicial da matriz M é A . Cada passagem pelo laço externo calcula a próxima matriz na seqüência $M_1, M_2, \dots, M_n = R$.

ALGORITMO WARSHALL

```

Warshall(matriz booleana n × n M)
//Inicialmente, M = matriz de adjacência de um grafo direcionado
//G sem arcos paralelos

para k = 0 até n - 1 faca
    para i = 1 até n faca
        para j = 1 até n faca
            M[i, j] = M[i, j]  $\vee$  (M[i, k + 1]  $\wedge$  M[k + 1, j])
        fim do para
    fim do para
    fim do para
    //ao final, M = matriz de acessibilidade de G
fim de Warshall
```

Isso nos dá uma boa descrição do algoritmo de Warshall, que pode ser implementado facilmente em alguma linguagem de programação. Entretanto, esses passos são confusos para se executar à mão e necessitam de anotações em paralelo para se guardar todos os índices. Podemos escrever o algoritmo de maneira informal, tornando-o mais fácil para execução manual. Suponha, novamente, que a matriz M_k na seqüência já foi calculada e que estamos tentando escrever a linha i da próxima matriz na seqüência. Isso significa que precisamos calcular a expressão

$$M[i, j] \vee (M[i, k + 1] \wedge M[k + 1, j]) \quad (5)$$

para os diversos valores de j . Se o elemento $M[i, k + 1]$ é 0, então o elemento $M[i, k + 1] \wedge M[k + 1, j] = 0$ para todo j . A expressão (5) se reduz, então, a

$$M[i, j] \vee 0 = M[i, j]$$

Em outras palavras, a linha i da matriz não muda. Se, por outro lado, o elemento $M[i, k + 1]$ é 1, então $M[i, k + 1] \wedge M[k + 1, j] = M[k + 1, j]$ para todo j . A expressão (5) fica

$$M[i, j] \vee M[k + 1, j]$$

Em outras palavras, a nova linha i da matriz vai ser a operação booleana **ou** da linha i atual com a linha $k + 1$ atual.

A Tabela 6.1 descreve os passos (informais) para se calcular os elementos de M_{k+1} a partir da matriz M_k .

TABELA 6.1

1. Considere a coluna $k + 1$ na matriz M_k .
2. Para cada linha com um elemento 0 nessa coluna, copie essa linha em M_{k+1} .
3. Para cada linha com um elemento 1 nessa coluna, execute a operação booleana **ou** dessa linha com a linha $k + 1$ e escreva a linha resultante em M_{k+1} .

❖ EXEMPLO 6 Para o grafo do Exemplo 5, a matriz inicial \mathbf{M}_0 é a matriz de adjacência.

$$\mathbf{M}_0 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Conhecemos \mathbf{M}_0 e queremos calcular \mathbf{M}_1 . Usando o passo 1 da Tabela 6.1, vamos considerar a coluna 1 de \mathbf{M}_0 . Usando o passo 2 da Tabela 6.1, as linhas 1, 2 e 4 de \mathbf{M}_0 contêm 0 na coluna 1, logo essas linhas são copiadas diretamente em \mathbf{M}_1 :

$$\mathbf{M}_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Terminamos agora usando o passo 3 da Tabela 6.1. O elemento na linha 3 coluna 1 de \mathbf{M}_0 é 1, logo a linha 3 de \mathbf{M}_1 é o resultado da operação booleana linha 3 de \mathbf{M}_0 ou linha 1 de \mathbf{M}_0 :

$$\mathbf{M}_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Como o elemento da linha 5 coluna 1 de \mathbf{M}_0 é 1, a linha 5 de \mathbf{M}_1 é o resultado da operação booleana linha 5 de \mathbf{M}_0 ou linha 1 de \mathbf{M}_0 :

$$\mathbf{M}_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Para calcular os elementos de \mathbf{M}_2 , considere a coluna 2. As linhas 2 e 4 (correspondentes aos zeros na coluna 2) são copiadas sem modificações. Executamos a operação booleana ou entre as linhas 1 e 2 para obter a nova linha 1, entre as linhas 3 e 2 para obter a nova linha 3 e entre as linhas 5 e 2 para obter a nova linha 5:

$$\mathbf{M}_2 = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

\mathbf{M}_3 é calculada de forma análoga:

$$\mathbf{M}_3 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

M_4 e M_5 vão ser iguais a M_3 ; a linha 4 contém apenas zeros, logo a operação booleana **ou** entre essa linha e outra qualquer não vai modificar a outra, e a coluna 5 também só tem zeros, logo todas as linhas são copiadas diretamente. Em termos do grafo, não se obtém nenhum novo elemento igual a 1 porque não existem caminhos do nó 4 para qualquer outro nó, nem caminhos de outro nó para o nó 5. Portanto, $M_3 = M_4 = M_5 = R$, como calculado no Exemplo 5. Observe, no entanto, que as matrizes calculadas pelo algoritmo de Warshall, exceto por A e R , não são iguais às potências de A usadas em nosso algoritmo anterior para calcular R .

Cada passagem pelo laço externo do algoritmo de Warshall modifica a matriz, colocando outra no lugar da que estava na passagem anterior. O algoritmo de Warshall não precisa de armazenagem adicional para outras matrizes, embora tenhamos escrito novas matrizes no nosso exemplo. Precisamos verificar mais uma coisa. Como vamos modificando a (única) matriz ao prosseguir, durante qualquer passagem pelo laço externo alguns elementos pertencerão a M_{k+1} enquanto outros ainda pertencem a M_k . Especificamente, no passo $k + 1$, podemos considerar $M[i, k + 1] \wedge M[k + 1, j]$ na expressão (5), onde esses valores já foram calculados nessa passagem e, portanto, representam $M_{k+1}[i, k + 1]$ e $M_{k+1}[k + 1, j]$ e não os valores $M_k[i, k + 1]$ e $M_k[k + 1, j]$ que usamos na nossa justificativa para esse algoritmo. Pode acontecer que $M_{k+1}[i, k + 1]$ e $M_{k+1}[k + 1, j]$ são iguais a 1, de modo que o valor 1 é colocado em $M_{k+1}[i, j]$, enquanto que os valores $M_k[i, k + 1]$ e $M_k[k + 1, j]$ são nulos? Não — se $M_{k+1}[i, k + 1] = 1$, existe um caminho de n_i para n_{k+1} cujos nós interiores pertencem ao conjunto $\{n_1, n_2, \dots, n_{k+1}\}$. No entanto, como n_{k+1} é uma extremidade e ciclos podem ser eliminados, também tem que existir um caminho com nós interiores no conjunto $\{n_1, n_2, \dots, n_k\}$, de modo que $M_k[i, k + 1] = 1$. Um argumento análogo é válido para $M_{k+1}[k + 1, j]$.

Use o algoritmo de Warshall (formal ou informalmente) para calcular R para o grafo da Fig. 6.2. Compare sua resposta com a encontrada no Problema Prático 5.

PROBLEMA PRÁTICO 6

Quanto trabalho é necessário para executar o algoritmo de Warshall, medido pelo número de operações booleanas **e/ou**? Considere o algoritmo formal. O único comando de atribuição nesse algoritmo está dentro de três laços encaixados, um dentro do outro; será executado n^3 vezes. Cada execução do comando de atribuição necessita de um **e** e um **ou**; portanto, o trabalho total é $2n^3 = \Theta(n^3)$. Lembre que nosso algoritmo anterior para calcular R era de ordem $\Theta(n^4)$.

SEÇÃO 6.1 REVISÃO

TÉCNICAS

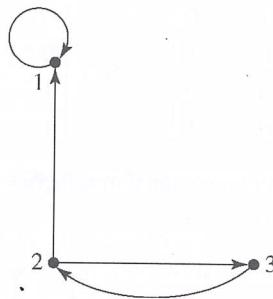
- ❖ Dada uma representação, encontrar qualquer das outras duas representações equivalentes entre a relação de adjacência, o grafo direcionado ou a matriz de adjacência.
- ❖ Calcular a matriz de acessibilidade R de um grafo G (ou, equivalentemente, encontrar o fecho transitivo da relação de adjacência em G) usando a fórmula $R = A \vee A^{(2)} \vee \dots \vee A^{(n)}$ e usando o algoritmo de Warshall.

IDÉIAS PRINCIPAIS

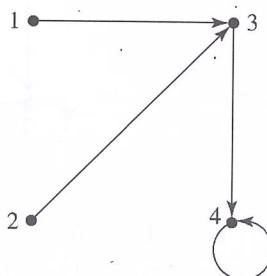
- ❖ Existe uma correspondência um para um entre um grafo direcionado G sem arcos paralelos, a relação de adjacência em G e a matriz de adjacência de G (em relação a alguma ordem arbitrária dos nós).
- ❖ A matriz de acessibilidade de um grafo G também representa o fecho transitivo da relação de adjacência em G .
- ❖ A matriz de acessibilidade de um grafo pode ser calculada com $\Theta(n^4)$ operações booleanas de **e/ou** somando-se as potências da matriz de adjacência A ou com $\Theta(n^3)$ operações booleanas de **e/ou** usando-se o algoritmo de Warshall.

EXERCÍCIOS 6.1

- ★1. Encontre a matriz de adjacência e a relação de adjacência para o grafo na figura a seguir:



2. Encontre a matriz de adjacência e a relação de adjacência para o grafo na figura a seguir:



- ★3. Encontre o grafo direcionado correspondente e a relação de adjacência para a matriz de adjacência a seguir:

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

4. Encontre o grafo direcionado correspondente e a relação de adjacência para a matriz de adjacência a seguir:

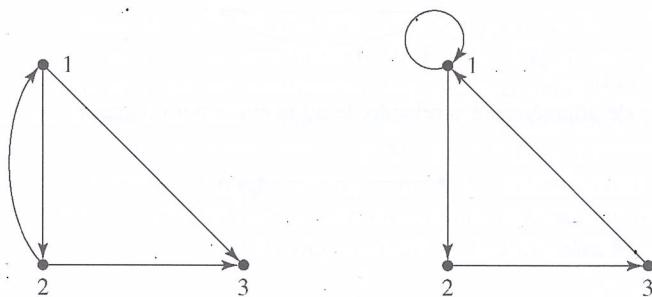
$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

5. Dada a relação de adjacência $\rho = \{(1, 4), (1, 5), (1, 6), (6, 2), (6, 3), (6, 5)\}$ no conjunto $N = \{1, 2, 3, 4, 5, 6\}$, encontre o grafo direcionado correspondente e a matriz de adjacência.
6. Dada a relação de adjacência $\rho = \{(2, 1), (3, 2), (3, 3), (3, 4), (4, 5), (6, 3), (6, 6)\}$ no conjunto $N = \{1, 2, 3, 4, 5, 6\}$, encontre o grafo direcionado correspondente e a matriz de adjacência.
7. Descreva uma propriedade de um grafo direcionado cuja matriz de adjacência é simétrica.
- ★8. Descreva o grafo direcionado cuja matriz de adjacência tem todos os elementos da linha 1 e todos os elementos da coluna 1 iguais a 1 e todos os outros elementos iguais a 0.
9. Descreva o grafo direcionado cuja matriz de adjacência tem todos os elementos nas posições $(i, i + 1)$ iguais a 1 para $1 \leq i \leq n - 1$, tem o elemento na posição $(n, 1)$ igual a 1 e tem todos os outros elementos iguais a 0.

10. Descreva uma propriedade da matriz de adjacência de um grafo cuja relação de adjacência é anti-simétrica.
11. As relações de adjacência ρ e σ têm as matrizes de adjacência associadas R e S dadas a seguir. Encontre as matrizes de adjacência associadas às relações $\rho \cup \sigma$ e $\rho \cap \sigma$.

$$R = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad S = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

- ★12. Os dois grafos direcionados na figura a seguir têm relações de adjacência ρ e σ . Desenhe os grafos associados às relações $\rho \cup \sigma$ e $\rho \cap \sigma$.

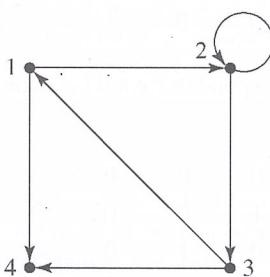


13. Seja A a matriz

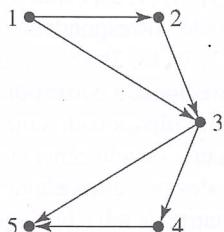
$$A = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Encontre os produtos A^2 e $A^{(2)}$.

14. A definição de *grafo conexo* pode ser estendida a grafos direcionados. Descreva a matriz de acessibilidade R para um grafo direcionado conexo.
- ★15. Para o grafo da figura a seguir, escreva a matriz de acessibilidade R simplesmente inspecionando o grafo.



16. Para o grafo da figura a seguir, escreva a matriz de acessibilidade R simplesmente inspecionando o grafo.



Para os Exercícios 17 a 22, calcule a matriz de acessibilidade R usando a fórmula $R = A \cup A^{(2)} \cup \dots \cup A^{(n)}$.

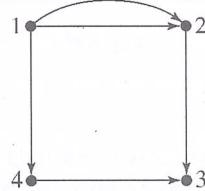
17. Exercício 1.
 18. Exercício 2.
 ★19. Exercício 3.
 20. Exercício 4.
 21. Exercício 5.
 22. Exercício 6.

Para os Exercícios 23 a 28, calcule a matriz de acessibilidade \mathbf{R} usando o algoritmo de Warshall.

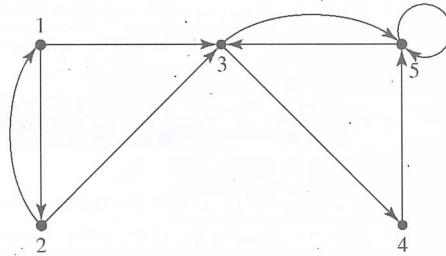
- ★23. Exercício 1.
 24. Exercício 2.
 25. Exercício 3.
 26. Exercício 4.
 ★27. Exercício 5.
 28. Exercício 6.

Os Exercícios 29 a 32 usam a multiplicação usual de matrizes para obter informação sobre um grafo.

- ★29. Seja G um grafo direcionado, podendo ter arcos paralelos, e seja \mathbf{A} sua matriz de adjacência. Então \mathbf{A} pode não ser uma matriz booleana. Prove que o elemento i,j da matriz \mathbf{A}^2 é igual ao número de caminhos de comprimento 2 do nó i para o nó j .
 30. Seja \mathbf{A} a matriz de adjacência de um grafo direcionado G , podendo ter arcos paralelos. Prove que o elemento i,j da matriz \mathbf{A}^n fornece o número de caminhos de comprimento n do nó i para o nó j .
 31. Para o grafo G a seguir, conte o número de caminhos de comprimento 2 do nó 1 para o nó 3. Verifique calculando \mathbf{A}^2 .



32. Para o grafo G a seguir, conte o número de caminhos de comprimento 4 do nó 1 para o nó 5. Verifique calculando \mathbf{A}^4 .



33. Seja ρ a relação binária definida no conjunto $\{0, \pm 1, \pm 2, \pm 4, \pm 16\}$ por $x \rho y \leftrightarrow y = x^2$. Desenhe o grafo direcionado associado.

SEÇÃO 6.2 CAMINHO DE EULER E CIRCUITO HAMILTONIANO

O PROBLEMA DO CAMINHO DE EULER

O problema do caminho de Euler (o problema de inspeção de rodovias) originou-se há muitos anos. O matemático suíço Leonhard Euler (pronuncia-se “óiler”) (1707–1783) ficou intrigado com um problema popular entre os habitantes de Königsberg (uma cidade na parte leste da antiga Prússia, mais tarde chamada de Caliningrado, na Rússia). O rio que atravessa a cidade bifurca em torno de uma ilha. Diversas pontes atravessam o rio, como ilustrado na Fig. 6.4.

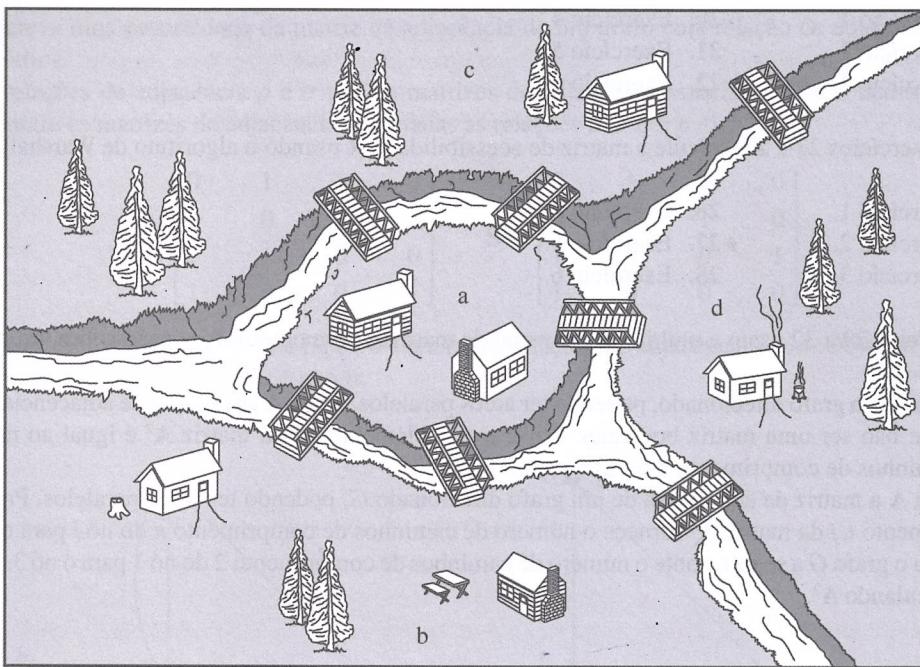


Fig. 6.4

O problema é decidir se uma pessoa poderia passear por toda a cidade cruzando cada ponte apenas uma vez. É possível resolver o problema por tentativa e erro, listando (ou andando) todos os caminhos possíveis, de modo que alguns habitantes dedicados poderiam ter resolvido esse problema particular. A idéia de Euler era representar a situação por um grafo (veja a Fig. 6.5), com as pontes representadas por arcos e as partes em terra da cidade (marcadas de a até d) representadas por nós. Ele então resolveu — por um mecanismo melhor do que tentativa e erro — a questão geral de quando existe um caminho de Euler em um grafo qualquer.

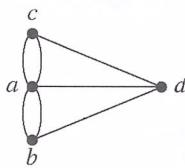


Fig. 6.5

Definição: Caminho de Euler Um **caminho de Euler** em um grafo G é um caminho que usa cada arco em G exatamente uma vez.

Existem caminhos de Euler para um dos grafos na Fig. 6.6? (Use tentativa e erro para responder. Essa é a velha brincadeira de criança, se é possível desenhar todo o grafo sem levantar o lápis do papel e sem desenhar duas vezes qualquer arco.)

PROBLEMA
PRÁTICO 7

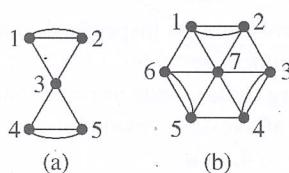


Fig. 6.6

Para essa discussão, vamos supor que todos os grafos são conexos, já que, caso contrário, um caminho de Euler não pode existir. A existência de um caminho de Euler em um determinado grafo depende dos graus de seus nós. Um nó é **par** se tem grau par e é **ímpar** se tem grau ímpar. Acontece que todo grafo tem um número par de nós ímpares. Para ver isso, escolha qualquer grafo e seja N o número de seus nós que são ímpares, $N(1)$ o número de nós de grau 1, $N(2)$ o número de nós de grau 2, e assim por diante. Então a soma S de todos os graus de todos os nós do grafo é

$$S = 1 \cdot N(1) + 2 \cdot N(2) + 3 \cdot N(3) + \dots + k \cdot N(k) \quad (1)$$

para algum k . Essa soma é, de fato, uma contagem do número total de extremidades de arco no grafo. Como o número de extremidades de arco é o dobro do número de arcas, S é um número par. Vamos reorganizar a equação (1), agrupando as parcelas correspondentes aos nós ímpares e as correspondentes aos nós pares:

$$\begin{aligned} S &= \underbrace{2 \cdot N(2) + 4 \cdot N(4) + \dots + 2m \cdot N(2m)}_{\text{nós pares}} \\ &\quad + \underbrace{1 \cdot N(1) + 3 \cdot N(3) + \dots + (2n+1) \cdot N(2n+1)}_{\text{nós ímpares}} \end{aligned}$$

A soma das parcelas que representam os nós pares é um número par. Subtraindo essa quantidade de ambos os lados da equação, obtemos uma nova equação

$$S' = 1 \cdot N(1) + 3 \cdot N(3) + \dots + (2n+1) \cdot N(2n+1) \quad (2)$$

onde S' (a diferença entre dois números pares) é um número par. Reescrevendo a Eq. (2) na forma

$$\begin{aligned} S' &= \underbrace{1 + 1 + \dots + 1}_{N(1) \text{ termos}} + \underbrace{3 + 3 + \dots + 3}_{N(3) \text{ termos}} + \dots \\ &\quad + \underbrace{(2n+1) + (2n+1) + \dots + (2n+1)}_{N(2n+1) \text{ termos}} \end{aligned}$$

vemos que essa soma tem N parcelas ao todo (o número de nós ímpares) e que cada parcela é um número ímpar. Para que a soma de N números ímpares seja par é preciso que N seja par. (Você pode provar isso?) Acabamos, então de provar o seguinte teorema:

Teorema sobre os Nós Ímpares em um Grafo O número de nós ímpares em qualquer grafo é par.

Suponha, agora, que um grafo tem um nó ímpar n de grau $2k+1$ e que existe um caminho de Euler no grafo que não começa em n . Então, para cada arco que usamos para chegar em n , existe um outro arco ainda não usado para sair de n , até que tenhamos usado os k pares de arcas. A próxima vez que chegarmos em n não haverá nenhum novo arco para sairmos. Assim, se nosso caminho não começa em n , ele tem que terminar em n . O caminho começa em n ou não, e, nesse último caso, ele termina em n , logo o caminho começa ou termina nesse nó ímpar arbitrário. Portanto, se existem mais de dois nós ímpares no grafo, não pode existir um caminho. Existem, então, dois casos possíveis onde um caminho de Euler pode existir — em um grafo sem nós ímpares ou com dois nós ímpares.

Considere o grafo sem nós ímpares. Pegue qualquer nó m e comece um caminho de Euler. Quando entrar em um nó diferente, sempre vai ter um outro arco para sair até chegar de volta a m . Se tiver usado todos os arcas do grafo, acabou. Se não, existe algum nó m' de seu caminho com arcas que não foram usados. Construa, então, um caminho de Euler que começa e termina em m' , de maneira análoga à anterior, usando todos os novos arcas. Esse ciclo pode ser adicionado ao caminho original como uma volta extra. Se tiver usado, agora, todos os arcas, acabou. Senão, continue esse processo até usar todos os arcas.

Se existem exatamente dois nós ímpares, pode-se começar um caminho de Euler em um deles e terminar em outro. Se o caminho não passou por todos os arcas, pode-se adicionar ciclos extras como no caso anterior.

Temos, agora, a solução completa do problema do caminho de Euler.

Teorema sobre Caminhos de Euler Existe um caminho de Euler em um grafo conexo se, e somente se, não existem nós ímpares ou existem exatamente dois nós ímpares. No caso em que não existem nós ímpares, o caminho pode começar em qualquer nó e terminar aí; no caso de dois nós ímpares, o caminho precisa começar em um deles e terminar no outro.

Usando o teorema precedente, faça novamente o Problema Prático 7.

❖ PROBLEMA PRÁTICO 8

O passeio de Königsberg é possível?

❖ PROBLEMA PRÁTICO 9

O teorema sobre caminhos de Euler é, de fato, um algoritmo para determinar se existe um caminho de Euler em um grafo conexo arbitrário. Para fazer com que ele pareça mais com um algoritmo, vamos escrevê-lo em pseudocódigo, mas primeiro vamos fazer uma hipótese que vai simplificar o problema, a de que o grafo não tem laços. Se o grafo G tiver laços, podemos retirá-los e considerar o grafo modificado H . Se H tiver um caminho de Euler, G também tem — sempre que chegarmos a um nó contendo um laço, percorremos o laço. Se H não tiver um caminho de Euler, G também não tem.

No algoritmo correspondente (algoritmo *CaminhoDeEuler*), os dados de entrada correspondem a um grafo conexo representado por uma matriz de adjacência A $n \times n$. A essência do algoritmo é contar o número de nós adjacentes a cada nó e determinar se esse é um número par ou ímpar. Se existem números ímpares demais, não existe um caminho de Euler. A variável *total* guarda o número de nós ímpares encontrados no grafo. O grau de um determinado nó, *grau*, é encontrado somando-se os números na linha da matriz de adjacência correspondente ao nó. (Essa é a razão da exclusão dos laços; um laço no nó i só adiciona 1 ao elemento $[i, j]$ da matriz de adjacência, mas esse laço contribui com 2 extremidades de arco.) A função *ímpar* tem valor “verdade” se, e somente se, o argumento é um inteiro ímpar.

ALGORITMO CAMINHODEEULER

CaminhoDeEuler (matriz $n \times n$ A)

//Determina se existe um caminho de Euler em um grafo conexo

//sem laços e com matriz de adjacência A .

Variáveis locais:

inteiro *total* //número de nós ímpares encontrados até agora
 inteiro *grau* //o grau de um nó
 inteiros *i, j* //índices da matriz

total = 0

i = 1

enquanto *total* <= 2 e *i* <= *n* **faça**

grau = 0

para *j* = 1 **até** *n* **faça**

grau = *grau* + *A*[*i, j*] //encontra o grau do nó *i*(*)

fim do para

se ímpar(*grau*) **então**

total = *total* + 1 //encontrou um outro nó ímpar

fim do se

i = *i* + 1

fim do enquanto

se *total* > 2 **então**

 escreva (“Não existe um caminho de Euler”)

senão

 escreva (“Existe um caminho de Euler”)

fim do se

fim de *CaminhoDeEuler*

❖ EXEMPLO 7

A matriz de adjacência do grafo da Fig. 6.6a é

$$\begin{bmatrix} 0 & 2 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 & 0 \end{bmatrix}$$

Quando o algoritmo entra pela primeira vez no laço de **enquanto**, *total* é 0 e *i* é 1. Então é atribuído a *grau* o valor inicial 0. Dentro do laço de **para**, são somados os elementos da linha 1 da matriz de adjacência ao *grau*, resultando em um valor 3 para *grau*. A função **ímpar** aplicada a *grau* tem valor “verdade”, logo o valor de *total* vai de 0 para 1; foi encontrado um nó de grau ímpar. Então o valor de *i* vai para 2. Não foram excedidos nem o limite sobre *total* nem o limite sobre o tamanho da matriz, de modo que o laço de **enquanto** é executado mais uma vez, dessa vez para a linha 2 da matriz. Mais uma vez, *grau* é ímpar, logo o valor *total* é mudado para 2. Quando o laço de **enquanto** é executado para a linha 3 da matriz, o valor de *grau* é par (4), logo o *total* não muda e o laço de **enquanto** é executado mais uma vez com *i* = 4. A linha 4 produz, novamente, um valor ímpar para *grau*, logo *total* sobe para 3. Isso encerra o laço de **enquanto**. Escreve-se a má notícia de que não existe um caminho de Euler porque o número de nós ímpares é maior do que 2. ❖

PROBLEMA PRÁTICO 10

Escreva a matriz de adjacência para o problema do passeio de Königsberg e execute o algoritmo *CaminhoDeEuler*.

Vamos analisar o algoritmo *CaminhoDeEuler*. A operação importante executada pelo algoritmo é um exame dos elementos na matriz de adjacência, que ocorre na linha (*). No pior caso, o laço de **enquanto** no algoritmo é executado n vezes, uma para cada linha. Dentro do laço de **enquanto**, o laço de **para**, contendo a linha (*), é executado n vezes, uma para cada coluna. Portanto, o algoritmo *CaminhoDeEuler* é de ordem $\Theta(n^2)$ no pior caso.

Podemos modificar o algoritmo, ao custo de algumas decisões lógicas extras, porque nunca precisamos examinar a última linha da matriz. Sabemos, do teorema sobre caminhos de Euler, que o número total de nós ímpares é par. Se o número de nós ímpares encontrados até a penúltima linha é ímpar, então a última linha tem que representar um nó ímpar; se esse número é par, então a última linha tem que representar um nó par. Essa modificação resulta na análise de $(n - 1)n$ elementos no pior caso, que ainda é $\Theta(n^2)$.

Se representarmos o grafo G por uma lista de adjacência ao invés de uma matriz de adjacência, então a versão correspondente do algoritmo teria que contar o comprimento da lista de adjacência de cada nó e guardar quantos deles têm comprimento ímpar. Existiriam n listas de adjacência para examinar, como existem n linhas da matriz de adjacência para examinar, mas o comprimento de cada lista pode ser menor do que n , o comprimento de uma linha da matriz. É possível reduzir a ordem de grandeza, ficando menor do n^2 , se o número de arcos no grafo for pequeno, mas o pior caso ainda é de ordem $\Theta(n^2)$.

O PROBLEMA DO CIRCUITO HAMILTONIANO

Um outro matemático famoso, William Rowan Hamilton (1805–1865), colocou um problema em teoria dos grafos muito semelhante ao de Euler. Ele perguntou como dizer se um grafo tem um **círculo hamiltoniano**, um ciclo contendo todos os nós do grafo.

PROBLEMA PRÁTICO 11

Existem ciclos hamiltonianos para os grafos na Fig. 6.6? (Use tentativa e erro para responder.) ❖

Como o problema do caminho de Euler, o problema do circuito hamiltoniano pode ser resolvido para um determinado grafo por tentativa e erro. O algoritmo é o seguinte: comece por um nó do grafo e tente alguns caminhos escolhendo diversos arcos. Se o caminho resulta em um nó repetido, não é um ciclo, jogue-o fora e tente um caminho diferente. Se o caminho pode ser completado para formar um ciclo, veja se todos os nós são visitados; se não, jogue-o fora e tente um caminho diferente. Continue assim até tentar todos os caminhos possíveis ou encontrar um circuito hamiltoniano. Isso vai envolver guardar cuidadosamente alguma informação de modo a não tentar um caminho mais de uma vez. A abordagem de tentativa e erro é, teoricamente, possível — mas, na prática, é impossível! Com exceção de grafos muito pequenos, vão existir simplesmente caminhos demais para se tentar.

Euler encontrou um algoritmo simples e eficiente para determinar, para um grafo arbitrário, se existe um caminho de Euler. Embora o problema do circuito hamiltoniano pareça bastante semelhante ao do caminho de Euler, existe uma diferença básica. Nunca se encontrou um algoritmo eficiente para determinar se existe um circuito hamiltoniano. De fato, existe alguma evidência (veja a Seção 8.3) de que um tal algoritmo nunca será encontrado.

Em determinados tipos de grafos, podemos determinar facilmente se existe um circuito hamiltoniano. Por exemplo, um grafo completo com $n > 2$ nós tem um circuito hamiltoniano porque, para qualquer nó no caminho, existe sempre um arco para se ir a qualquer nó não visitado e, finalmente, um arco para se voltar ao ponto de partida. Em geral, no entanto, não é possível determinar isso facilmente.

Suponha que estamos tratando de um grafo com peso. Se existe um circuito hamiltoniano para o grafo, podemos encontrar um de peso mínimo? Esse é o problema do caixeiro-viajante. Mais uma vez ele pode ser resolvido por tentativa e erro, traçando-se todos os caminhos possíveis e guardando-se os pesos dos caminhos que são circuitos hamiltonianos mas, novamente, esse não é um algoritmo eficiente. (Aliás, o problema do caixeiro-viajante para visitar todas as 48 capitais da parte continental contígua dos Estados Unidos foi resolvido — é necessário um total de aproximadamente 17.100 quilômetros!)

SEÇÃO 6.2 REVISÃO

TÉCNICA

- W** ♦ Usar o algoritmo *CaminhoDeEuler* para determinar se existe um caminho de Euler em um grafo.

IDÉIAS PRINCIPAIS

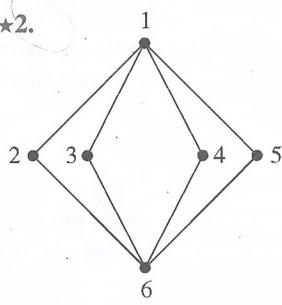
- ♦ Existe um critério simples para se determinar se existem caminhos de Euler em um grafo, mas não existe um tal critério para se determinar a existência de circuitos hamiltonianos.
♦ Um algoritmo da ordem de $\Theta(n^2)$ no pior caso pode determinar a existência de um caminho de Euler em um grafo conexo com n nós.

EXERCÍCIOS 6.2

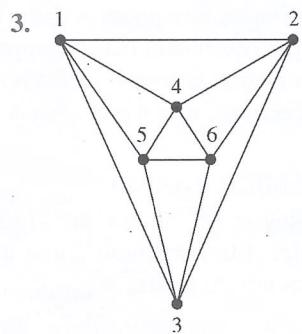
1. Refaça o Exemplo 3 do Cap. 2 usando o teorema sobre caminhos de Euler.

Para os Exercícios 2 a 10, determine se o grafo especificado tem um caminho de Euler usando o teorema sobre caminhos de Euler

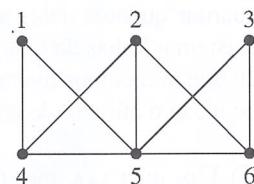
★2.



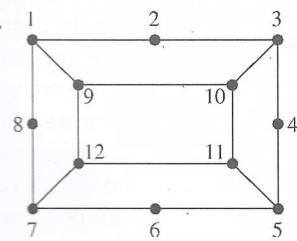
3.



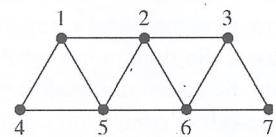
4.



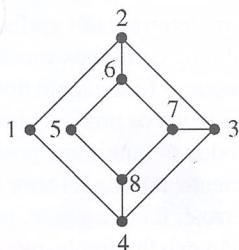
5.

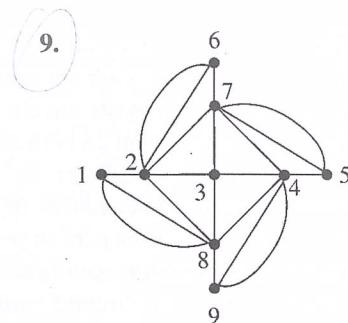
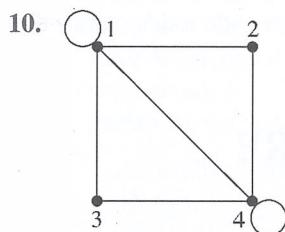
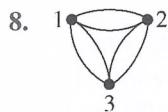


6.



★7.

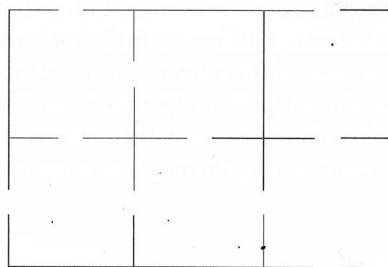




- ★11. Escreva a matriz de adjacência para o grafo do Exercício 2. Ao aplicar o algoritmo *CaminhoDeEuler*, qual o valor de *total* após a segunda passagem pelo laço **enquanto**?
- 12. Escreva a matriz de adjacência para o grafo do Exercício 4. Ao aplicar o algoritmo *CaminhoDeEuler*, qual o valor de *total* após a quarta passagem pelo laço **enquanto**?
- ★13. Escreva a matriz de adjacência para o grafo do Exercício 6. Ao aplicar o algoritmo *CaminhoDeEuler*, qual o valor de *i* após a saída do laço **enquanto**?
- 14. Escreva a matriz de adjacência para o grafo do Exercício 7. Ao aplicar o algoritmo *CaminhoDeEuler*, qual o valor de *i* após a saída do laço **enquanto**?

Para os Exercícios 15 a 22, decida por tentativa e erro se existem circuitos hamiltonianos para os grafos dos exercícios dados.

- ★15. Exercício 2 19. Exercício 6
- 16. Exercício 3 ★20. Exercício 7
- 17. Exercício 4 21. Exercício 8
- 18. Exercício 5 22. Exercício 9
- 23. Encontre um exemplo de um grafo não conexo que tem um caminho de Euler. (*Sugestão*: como isso parece ser intuitivamente contraditório, procure um caso trivial.)
- ★24. Prove que qualquer grafo que contém um circuito hamiltoniano é conexo.
- 25. Considere um grafo simples e completo com n nós. Podemos testar a existência de um circuito hamiltoniano por tentativa e erro selecionando um nó inicial fixo e depois gerando todos os possíveis caminhos de comprimento n a partir desse nó.
 - a. Quantos caminhos de comprimento n existem se são permitidas repetições de arcos e nós?
 - b. Quantos caminhos de comprimento n existem se são permitidas repetições de arcos e nós, mas um arco não pode ser usado em duas vezes sucessivas?
 - c. Quantos caminhos de comprimento n existem se não são permitidas repetições de arcos e nós com exceção do nó inicial? (Esses são os circuitos hamiltonianos.)
- 26. É possível entrar e sair de cada quarto na casa ilustrada na figura a seguir de modo que cada porta da casa seja usada exatamente uma vez? Por quê?



- ★27. Lembre-se de que K_n denota o grafo simples completo de ordem n .
 - a. Para que valores de n existe um caminho de Euler em K_n ?
 - b. Para que valores de n existe um circuito hamiltoniano em K_n ?

28. Lembre-se de que $K_{m,n}$ denota o grafo bipartido completo com $m + n$ nós.
- Para que valores de m e n existe um caminho de Euler em $K_{m,n}$?
 - Para que valores de m e n existe um circuito hamiltoniano em $K_{m,n}$?
29. Considere um grafo conexo com $2n$ vértices ímpares, $n \geq 2$. Pelo teorema sobre caminhos de Euler, esse grafo não tem caminhos de Euler.
- Qual o número mínimo de caminhos de Euler disjuntos, cada um viajando por alguns dos arcos no grafo, que são necessários para se percorrer cada arco exatamente uma vez?
 - Mostre que o número mínimo é suficiente.
- *30. Prove que sempre existe um circuito hamiltoniano em um grafo conexo onde todos os nós têm grau 2.

SEÇÃO 6.3 CAMINHO MÍNIMO E ÁRVORE GERADORA MÍNIMA

O PROBLEMA DO CAMINHO MÍNIMO

Suponha que temos um grafo simples conexo e com peso, onde os pesos são positivos. Então existe um caminho entre dois nós quaisquer x e y . De fato, podem existir muitos desses caminhos. A pergunta é: como encontrar um caminho com peso mínimo? Como o peso representa, muitas vezes, a distância, esse problema ficou conhecido como o problema do “caminho mínimo” (no sentido de “mais curto”). É um problema importante para uma rede de computadores ou de comunicação, onde a informação em um nó tem que ser enviada a outro nó do modo mais eficiente possível, ou para uma rede de transporte, onde os produtos de uma cidade têm que ser enviados a outra.

O problema do caixeiro-viajante é um problema de caminho de peso mínimo com restrições tão severas sobre a natureza do caminho que um tal caminho pode não existir. No problema de caminho mínimo, não há restrições (fora o peso mínimo) sobre a natureza do caminho e, como o grafo é conexo, sabemos que existe um tal caminho. Por essa razão podemos esperar encontrar um algoritmo eficiente para resolver o problema, embora não se conheça um tal algoritmo para o problema do caixeiro-viajante. Existe, de fato, um tal algoritmo.

O algoritmo para o caminho mínimo, conhecido como algoritmo de Dijkstra, funciona da seguinte maneira: Queremos encontrar o caminho de distância mínima de um nó x dado a outro nó y dado. Vamos construir um conjunto (que chamaremos de IN) que contém apenas x inicialmente mas que aumenta durante a execução do algoritmo. Em qualquer instante dado, IN contém todos os nós cujos caminhos mínimos a partir de x , usando apenas nós em IN , já foram determinados. Para todo nó z fora de IN , guardamos a menor distância $d[z]$ de x àquele nó usando um caminho cujo único nó não pertencente a IN é z . Guardamos, também, o nó adjacente a z nesse caminho, $s[z]$.

Como aumentamos IN , isto é, qual o próximo nó a ser incluído em IN ? Escolhemos o nó não pertencente a IN que tem a menor distância d . Uma vez incluído esse nó, que chamaremos de p , em IN , teremos que recalcular d para todos os outros nós restantes fora de IN , já que pode existir um caminho menor (mais curto) a partir de x contendo p do que antes de p pertencer a IN . Se existir um caminho menor, precisaremos atualizar também $s[z]$ de modo que p apareça como o nó adjacente a z no caminho mínimo atual. Assim que y for incluído em IN , IN pár de aumentar. O valor atual de $d[y]$ é a distância correspondente ao menor caminho, cujos nós podem ser encontrados procurando-se y , $s[y]$, $s[s[y]]$, e assim por diante, até percorrer todo o caminho de volta e chegar a x .

É dada, a seguir, uma forma em pseudocódigo desse algoritmo (algoritmo *CaminhoMínimo*). Os dados de entrada correspondem à matriz de adjacência de um grafo G simples e conexo com pesos positivos e nós x e y ; o algoritmo descreve o caminho mais curto entre x e y e a distância correspondente. Aqui, caminho mínimo significa caminho de peso mínimo. De fato, supomos que a matriz A é uma matriz de adjacência modificada, onde $A[i, j]$ é o peso do arco entre i e j , se existir, e $A[i, j]$ tem o valor ∞ se não existir um arco de i para j (o símbolo ∞ denota um número maior do que todos os pesos no grafo).

ALGORITMO CAMINHOMÍNIMO

CaminhoMínimo (matriz $n \times n$ A; nós x, y)

//Algoritmo de Dijkstra. A é uma matriz de adjacência modificada de um grafo simples //e conexo com pesos positivos; x e y são nós no grafo; o algoritmo escreve os nós do //caminho mínimo de x para y e a distância correspondente.

Variáveis locais:

conjunto de nós IN	//nós cujo caminho mínimo de x é conhecido
nós z, p	//nós temporários
vetor de inteiros d	//para cada nó, distância de x usando nós em IN
vetor de nós s	//para cada nó, nó anterior no caminho mínimo
inteiro DistânciaAnterior	//distância para comparar

//inicializa o conjunto IN e os vetores d e s

IN = {x}

d[x] = 0

para todos os nós z não pertencentes a IN faça

 d[z] = A[x, z]

 s[z] = x

fim do para

//coloca nós em IN

enquanto y não pertence a IN faça

 //adiciona o nó de distância mínima não pertencente a IN

 p = nó z não pertencente a IN com d[z] mínimo

 IN = IN ∪ {p}

//recalcula d para os nós não pertencentes a IN, ajusta s se necessário

para todos os nós não pertencentes a IN faça

 DistânciaAnterior = d[z]

 d[z] = min(d[z], d[p] + A[p, z])

 se d[z] ≠ DistânciaAnterior então

 s[z] = p

 fim do se

fim do para

fim do enquanto

//escreve os nós do caminho

escreva("Em ordem inversa, os nós do caminho são")

escreva(y)

z = y

repita

 escreva(s[z])

 z = s[z]

até z = x

//escreve a distância correspondente

escreva("A distância percorrida é", d[y])

fim de CaminhoMínimo

Considere o grafo na Fig. 6.7 e a matriz de adjacência modificada correspondente na Fig. 6.8

EXEMPLO 8

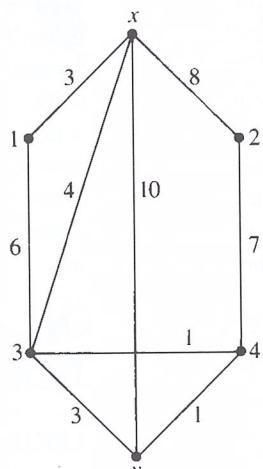


Fig. 6.7

	x	1	2	3	4	y
x	∞	3	8	4	∞	10
1	3	∞	∞	6	∞	∞
2	8	∞	∞	∞	7	∞
3	4	6	∞	∞	1	3
4	∞	∞	7	1	∞	1
y	10	∞	∞	3	1	∞

Fig. 6.8

Vamos seguir o algoritmo *CaminhoMínimo* para esse grafo. Ao final da fase de inicialização, IN só contém x , e d contém todas as distâncias diretas de x aos outros nós:

$$IN = \{x\}$$

	x	1	2	3	4	y
d	0	3	8	4	∞	10
s	—	x	x	x	x	x

Na Fig. 6.9, os nós dentro de círculos são os que estão no conjunto IN , as linhas mais grossas mostram os menores caminhos atuais e o valor de d para cada nó está escrito entre parênteses ao lado do nome do nó. A Fig. 6.9a ilustra a situação após a inicialização.

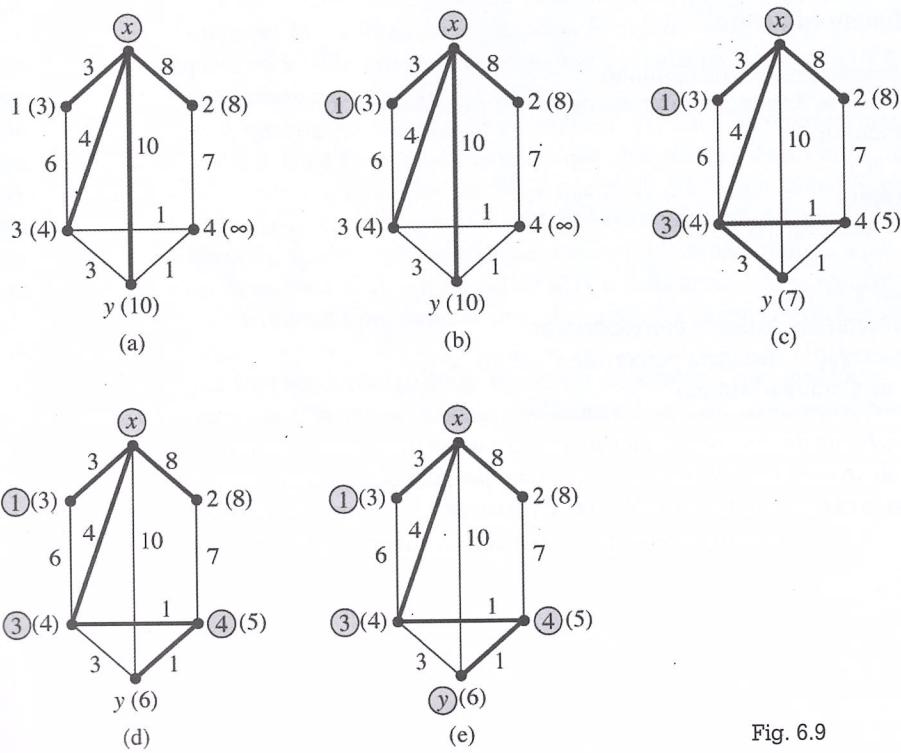


Fig. 6.9

Vamos entrar, agora, no laço de **enquanto** e procurar entre os valores de d o nó não pertencente a IN de distância mínima; esse é o nó 1, com $d[1] = 3$. Colocamos o nó 1 em IN e recalculamos, dentro do laço de **para**, os valores de d para os nós restantes 2, 3, 4 e y.

LEMBRETE:
As distâncias no algoritmo de Dijkstra são sempre recalculadas em relação ao último nó adicionado a IN .

$$\begin{aligned} p &= 1 \\ IN &= \{x, 1\} \\ d[2] &= \min(8, 3 + A[1, 2]) = \min(8, \infty) = 8 \\ d[3] &= \min(4, 3 + A[1, 3]) = \min(4, 9) = 4 \\ d[4] &= \min(\infty, 3 + A[1, 4]) = \min(\infty, \infty) = \infty \\ d[y] &= \min(10, 3 + A[1, y]) = \min(10, \infty) = 10 \end{aligned}$$

Não houve mudanças nos valores de d , logo não há mudanças nos valores de s (não havia nenhum caminho menor de x contendo o nó 1 do que indo diretamente de x). A Fig. 6.9b mostra que 1 agora pertence a IN .

A segunda passagem pelo laço de **enquanto** produz o seguinte:

$$\begin{aligned} p &= 3 \text{ (3 tem o menor valor de } d, \text{ a saber 4, entre 2, 3, 4, ou } y) \\ IN &= \{x, 1, 3\} \\ d[2] &= \min(8, 4 + A[3, 2]) = \min(8, 4 + \infty) = 8 \\ d[4] &= \min(\infty, 4 + A[3, 4]) = \min(\infty, 4 + 1) = 5 \text{ (uma mudança, atualize } s[4] \text{ para 3)} \\ d[y] &= \min(10, 4 + A[3, y]) = \min(10, 4 + 3) = 7 \text{ (uma mudança, atualize } s[y] \text{ para 3)} \end{aligned}$$

	x	1	2	3	4	y
d	0	3	8	4	5	7
s	-	x	x	x	3	3

Foram encontrados caminhos mais curtos de x aos nós 4 e y contendo o nó 3. A Fig. 6.9c reflete isso.

Na próxima passagem,

$$\begin{aligned} p &= 4 \text{ (valor de } d = 5) \\ IN &= \{x, 1, 3, 4\} \\ d[2] &= \min(8, 5 + 7) = 8 \\ d[y] &= \min(7, 5 + 1) = 6 \quad (\text{uma mudança, atualize } s[y]) \end{aligned}$$

	x	1	2	3	4	y
d	0	3	8	4	5	6
s	-	x	x	x	3	4

Veja a Fig. 6.9d.

Entrando no laço **enquanto** novamente, obtemos

$$\begin{aligned} p &= y \\ IN &= \{x, 1, 3, 4, y\} \\ d[2] &= \min(8, 6 + \infty) = 8 \end{aligned}$$

	x	1	2	3	4	y
d	0	3	8	4	5	6
s	-	x	x	x	3	4

Veja a Fig. 6.9e.

Agora que y pertence a IN , o laço **enquanto** termina. O caminho contém y , $s[y] = 4$, $s[4] = 3$ e $s[3] = x$. Logo, o caminho contém os nós x , 3, 4 e y . (O algoritmo nos dá esses nós em ordem inversa.) A distância correspondente é $d[y] = 6$. Examinando o grafo na Fig. 6.7 e verificando todas as possibilidades, vemos que esse é o caminho mínimo de x para y .

O algoritmo *CaminhoMínimo* termina quando y é colocado em IN , embora possam existir outros nós no grafo não pertencentes a IN (como o nó 2 no Exemplo 8). Como sabemos que não se pode encontrar um caminho mais curto de x a y contendo algum desses nós excluídos? Se continuarmos o algoritmo até incluir todos os nós em IN , os valores de d representarão os caminhos mínimos de x a qualquer nó usando todos os vértices em IN , isto é, usando todos os nós no grafo. Mas novos nós só podem ser colocados em IN para aumentar os valores de d . Um nó z que for adicionado ao conjunto após y tem que ter como seu caminho mínimo a partir de x um cuja distância é, pelo menos, tão grande quanto o valor de d para y quando y foi adicionado a IN . Portanto, não pode existir um caminho menor de x para y via z porque não existe nem um caminho menor só entre x e z .

Siga o algoritmo *CaminhoMínimo* para o grafo ilustrado na Fig. 6.10. Mostre os valores de p , o conjunto IN , e os valores dos vetores d e s em cada passagem do laço de **enquanto**. Escreva os nós do caminho mínimo e a distância percorrida.

PROBLEMA
PRÁTICO 12

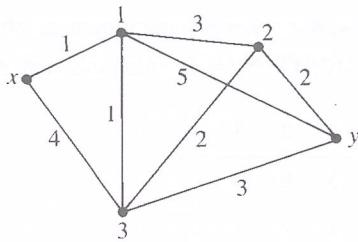


Fig. 6.10

Ao procurar o próximo nó para inclusão em IN no algoritmo *CaminhoMínimo*, mais de um nó p pode ter um valor mínimo em d , caso em que p pode ser selecionado arbitrariamente. Pode existir, também, mais de um caminho mínimo entre x e y em um grafo.

O algoritmo *CaminhoMínimo* também funciona para grafos direcionados se a matriz de adjacência estiver na forma apropriada. Também funciona para grafos não conexos; se x e y não estiverem na mesma componente conexa, então $d[y]$ vai permanecer igual a ∞ durante todo o tempo. Depois da inclusão de y em IN , o algoritmo termina e esse valor ∞ para $d[y]$ indica que não existe caminho entre x e y .

Podemos pensar no algoritmo *CaminhoMínimo* como sendo um algoritmo “mfope”. Ele não pode ver todo o grafo ao mesmo tempo para escolher os caminhos mínimos; escolhe apenas os caminhos mínimos em relação a IN em cada etapa. Um tal algoritmo é chamado de **algoritmo guloso** — faz o que parece ser melhor baseado em seu conhecimento imediato limitado. Nesse caso, o que parece melhor em determinado instante é, de fato, o melhor ao final.

Quão eficiente é o algoritmo de caminho mínimo? A maior parte do trabalho parece ser feito no laço de **para** que modifica os vetores d e s . Aí o algoritmo verifica todos os n nós para determinar quais nós z não estão em IN e recalcular $d[z]$ para esses, podendo, também, mudar $s[z]$. As quantidades necessárias $d[z]$, $d[p]$ e $A[p, z]$ para um dado z estão disponíveis imediatamente. O laço de **para**, portanto, necessita de $\Theta(n)$ operações. Além disso, a determinação do nó p que será incluído em IN pode, também, ser feita em $\Theta(n)$ operações verificando-se todos os nós. Com o pequeno trabalho adicional de incluir p em IN , cada execução do laço **enquanto** precisa de $\Theta(n)$ operações. No pior caso, y é o último nó de G ao ser incluído em IN e o laço de **enquanto** é executado $n - 1$ vezes. Portanto, o número total de operações envolvidas no laço de **enquanto** é $\Theta(n(n - 1)) = \Theta(n^2)$. A inicialização e escrita dos resultados, juntos, usam outras $\Theta(n)$ operações, de modo que o algoritmo precisa de $\Theta(n + n^2) = \Theta(n^2)$ operações no pior caso.

E se mantivermos IN (ou melhor, o complemento de IN) em uma espécie de lista encadeada, para não haver necessidade de examinar todos os nós do grafo para decidir quais deles não estão em IN ? Isso deveria tornar o algoritmo mais eficiente. Observe que o número de nós que não estão em IN é, inicialmente, $n - 1$ e esse número diminui de 1 em cada passagem do laço de **enquanto**. Dentro do laço de **enquanto**, o

algoritmo tem que executar, então, da ordem de $n - 1$ operações na primeira passagem, depois $n - 2$, depois $n - 3$, e assim por diante. Mas, como uma demonstração por indução vai mostrar,

$$(n - 1) + (n - 2) + \dots + 1 = (n - 1)n / 2 = \Theta(n^2)$$

Assim, a situação no pior caso ainda necessita de $\Theta(n^2)$ operações.

O PROBLEMA DA ÁRVORE GERADORA MÍNIMA

Um problema encontrado ao se projetar redes é como conectar todos os nós eficientemente, onde os nós podem ser computadores, telefones, depósitos, etc. Uma árvore geradora mínima pode fornecer uma solução econômica, uma que necessite da menor quantidade de cabos, tubos, ou qualquer que seja o material usado para a conexão. Por questões de segurança, no entanto, a árvore geradora mínima seria aumentada, em geral, com arcos adicionais de modo que, se alguma conexão for quebrada por alguma razão, uma rota alternativa poderia ser encontrada.

Definição: Árvore Geradora Uma **árvore geradora** para um grafo conexo é uma árvore sem raiz cujo conjunto de nós coincide com o conjunto de nós do grafo e cujos arcos são (alguns dos) arcos do grafo.

Uma árvore geradora, portanto, conecta todos os nós de um grafo sem arcos em excesso (e sem ciclos). Existem algoritmos para a construção de uma **árvore geradora mínima**, uma árvore geradora de peso mínimo, para um grafo dado simples e conexo com peso.

Um desses algoritmos, chamado de algoritmo de Prim, funciona de maneira muito parecida com o algoritmo de caminho mínimo. Existe um conjunto IN que contém, inicialmente, um nó arbitrário. Para cada nó z não pertencente a IN , guardamos a distância mínima $d[z]$ entre z e qualquer nó em IN . Incluímos, sucessivamente, nós em IN , onde o próximo nó a ser incluído é um que não pertence a IN e cuja distância $d[z]$ é mínima. O arco tendo essa distância mínima torna-se, então, parte da árvore geradora. Como pode haver distâncias mínimas iguais, a árvore geradora mínima pode não ser única. O algoritmo termina quando todos os nós pertencerem a IN .

A diferença fundamental entre as implementações dos dois algoritmos está no cálculo das novas distâncias para os nós que ainda não pertencem a IN . No algoritmo de Dijkstra, se p é o nó que acabou de ser incluído em IN , as distâncias para os nós que não estão em IN são recalculadas por

$$d[z] = \min(d[z], d[p] + A[p, z])$$

isto é, comparando a distância atual de x a z com a distância de x a p mais a distância de p a z . No algoritmo de Prim, se p é o nó que acabou de ser incluído em IN , as distâncias para os nós que não estão em IN são recalculadas por

$$d[z] = \min(d[z], A[p, z])$$

isto é, comparando a distância atual de z a IN com a distância de p a z .

Não escreveremos o algoritmo (que, como o algoritmo de caminho mínimo, necessita de $\Theta(n^2)$ operações no pior caso e é um algoritmo guloso); ilustraremos esse algoritmo simplesmente através de um exemplo.

EXEMPLO 9

Vamos encontrar uma árvore geradora mínima para o grafo da Fig. 6.7. Vamos tomar 1 como sendo o nó inicial arbitrário em IN . A seguir, consideramos todos os nós adjacentes a qualquer nó em IN , isto é, todos os nós adjacentes a 1, e selecionamos o mais próximo, que é x . Agora $IN = \{1, x\}$ e o arco entre 1 e x é parte da árvore geradora mínima. A seguir, consideramos todos os nós que não estão em IN e são adjacentes a 1 ou x . O mais próximo desses nós é o 3, que dista 4 de x . O arco entre 3 e x torna-se parte da árvore geradora mínima. Para $IN = \{1, x, 3\}$, o nó mais próximo a seguir é o 4, que dista 1 unidade de 3. Os nós restantes são incluídos, primeiro 4 e depois 2. A Fig. 6.11 mostra a árvore geradora mínima.

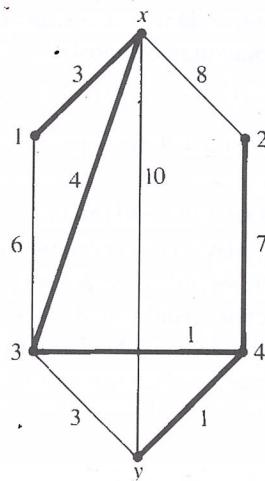


Fig. 6.11

Encontre uma árvore geradora mínima para o grafo da Fig. 6.10.

◆ PROBLEMA PRÁTICO 13

Instrumentos sísmicos devem ser distribuídos em uma zona de fendas vulcânicas, como ilustrado na Fig. 6.12a, onde estão indicadas as distâncias em metros entre os locais. (As distâncias entre alguns dos locais não são dadas devido a acidentes naturais que impediriam uma conexão direta.) A maneira mais econômica de colocar os dispositivos de modo que todos estejam conectados é criar uma árvore geradora mínima para o grafo, como ilustrado na Fig. 6.12b. A quantidade total de cabos envolvidos é de 975 metros.

◆ EXEMPLO 10

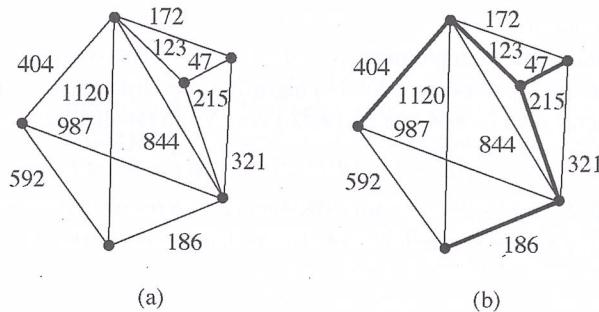


Fig. 6.12

SEÇÃO 6.3 REVISÃO

TÉCNICAS

- ◆ Encontrar um caminho mínimo de x a y em um grafo (usando o algoritmo de Dijkstra).
- ◆ Encontrar uma árvore geradora mínima para um grafo (usando o algoritmo de Prim).

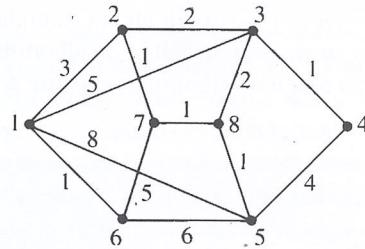
IDÉIA PRINCIPAL

- ◆ Pode-se usar algoritmos da ordem de $\Theta(n^2)$ no pior caso para encontrar um caminho mínimo entre dois nós ou uma árvore geradora mínima em um grafo simples e conexo com pesos positivos e n nós.

EXERCÍCIOS 6.3

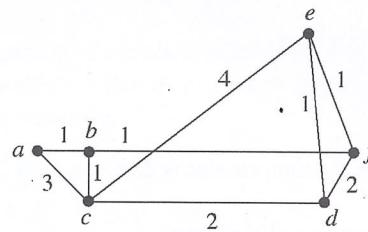
Para os Exercícios 1 a 4, use o grafo a seguir. Aplique o algoritmo *CaminhoMínimo* (algoritmo de Dijkstra) nos pares de nós dados; mostre os valores de p , o conjunto IN , os valores de d e os valores de s em cada

passagem do laço de **enquanto**. Escreva os nós no caminho de distância mínima e a distância correspondente.



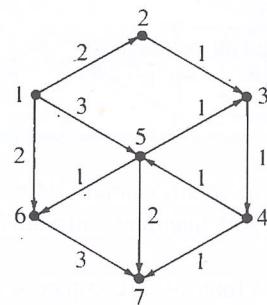
- ★1. De 2 para 5.
- 2. De 3 para 6.
- 3. De 1 para 5.
- 4. De 4 para 7.

Para os Exercícios 5 e 6, use o grafo a seguir. Aplique o algoritmo *CaminhoMínimo* (algoritmo de Dijkstra) nos pares de nós dados; mostre os valores de p , o conjunto IN , os valores de d e os valores de s em cada passagem do laço de **enquanto**. Escreva os nós no caminho de distância mínima e a distância correspondente.



- ★5. De a para e .
- 6. De d para a .

Para os Exercícios 7 e 8, use o grafo direcionado a seguir. Aplique o algoritmo *CaminhoMínimo* (algoritmo de Dijkstra) nos pares de nós dados; mostre os valores de p , o conjunto IN , os valores de d e os valores de s em cada passagem do laço de **enquanto**. Escreva os nós no caminho de distância mínima e a distância correspondente.



- ★7. De 1 para 7.
- 8. De 3 para 1.
- 9. a. Modifique o algoritmo *CaminhoMínimo* de modo a encontrar os menores caminhos de x para todos os outros nós no grafo.
b. Isso muda a ordem de grandeza do algoritmo no pior caso?
- 10. Dê um exemplo para mostrar que o algoritmo *CaminhoMínimo* não funciona quando são permitidos pesos negativos.

Um outro algoritmo para encontrar os caminhos mínimos a partir de um único nó para todos os outros em um grafo é o *algoritmo de Bellman-Ford*. Ao contrário do algoritmo de Dijkstra, que guarda um conjunto

de nós cujas distâncias mínimas de comprimento arbitrário (isto é, número de arcos) já foram determinadas, o algoritmo de Bellman-Ford executa uma série de cálculos procurando encontrar caminhos mínimos, sucessivamente, de comprimento 1, depois de comprimento 2, depois de comprimento 3, e assim por diante, até ao máximo de comprimento $n - 1$ (se existir algum caminho, então existe um de comprimento menor ou igual a $n - 1$). Uma descrição em pseudocódigo do algoritmo de Bellman-Ford é dada a seguir (algoritmo *OutroCaminhoMínimo*); ao usar esse algoritmo, a matriz A tem que satisfazer $A[i, i] = 0$ para todo i .

ALGORITMO OutroCaminhoMínimo

OutroCaminhoMínimo (matriz $n \times n A$; nó x ; vetor de inteiros d , vetor de nós s)
 //Algoritmo de Bellman-Ford. A é uma matriz de adjacência modificada de um grafo
 //simples e conexo com pesos; x é um nó no grafo; quando o algoritmo terminar, os nós
 //do caminho mínimo de x para um nó y são $y, s[y], s[s[y]], \dots, x$;
 //a distância correspondente é $d[y]$.

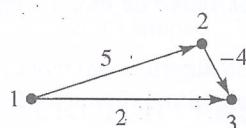
Variáveis locais:

nós z, p //nós temporários
 vetor de inteiros t //vetor de distâncias temporário, criado em cada iteração

```
//inicializa os vetores  $d$  e  $s$ ; isso estabelece os caminhos
//mínimos de comprimento 1 a partir de  $x$ 
 $d[x] = 0$ 
para todos os nós  $z$  diferentes de  $x$  faça
   $d[z] = A[x, z]$ 
   $s[z] = x$ 
fim do para
//encontra os caminhos míspimos de comprimentos 2, 3, etc.
para  $i = 2$  até  $n - 1$  faça
   $t = d$  //copia vetor atual  $d$  no vetor  $t$ 
  //modifica  $t$  para guardar os menores caminhos de comprimento  $i$ 
  para todos os nós  $z$  diferentes de  $x$  faça
    //encontra o caminho mínimo com mais um arco
     $p = \text{nó em } G \text{ para o qual } (d[p] + A[p, z]) \text{ é mínimo}$ 
     $t[z] = d[p] + A[p, z]$ 
    se  $p \neq z$  então
       $s[z] = p$ 
    fim do se
  fim do para
   $d = t$  //copia o vetor  $t$  de volta em  $d$ 
fim do para
fim de OutroCaminhoMínimo
```

Para os Exercícios 11 a 14, use o algoritmo *OutroCaminhoMínimo* (o algoritmo de Bellman-Ford) para encontrar o caminho mínimo do nó fonte para qualquer outro nó. Mostre os valores sucessivos de d e de s .

- ★11. O grafo para os Exercícios 1 a 4, nó fonte = 2 (compare sua resposta com o Exercício 1).
- 12. O grafo para os Exercícios 1 a 4, nó fonte = 1 (compare sua resposta com o Exercício 3).
- 13. O grafo para os Exercícios 7 e 8, nó fonte = 1 (compare sua resposta com o Exercício 7).
- 14. Grafo da figura a seguir, nó fonte = 1 (compare sua resposta com o Exercício 10).



Para calcular a distância correspondente ao caminho mínimo entre dois nós quaisquer em um grafo, o algoritmo *CaminhoMínimo* poderia ser usado repetidamente, com cada um dos nós sendo o nó fonte. Um algoritmo diferente, muito semelhante ao algoritmo de Warshall, também pode ser usado para resolver esse

problema de caminho mínimo “entre todos os pares”. Segue uma descrição desse algoritmo, onde A é a matriz de adjacência do grafo com $A[i, i] = 0$ para todo i .

```

ALGORITMO CAMINHOMÍNIMOENTRETODOSOSPARES
CaminhoMínimoEntreTodosOsPares (matriz n × n A)
//Algoritmo de Floyd — calcula o caminho mínimo entre dois nós
//quaisquer; inicialmente, A é a matriz de adjacência; ao final,
//A vai conter todas as distâncias dos caminhos mínimos

para k = 1 até n faca
    para i = 1 até n faca
        para j = 1 até n faca
            se A[i, k] + A[k, j] < A[i, j] então
                A[i, j] = A[i, k] + A[k, j]
            fim do se
        fim do para
    fim do para
    fim do para
fim de CaminhoMínimoEntreTodosOsPares

```

Para os Exercícios 15 e 16, use o algoritmo *CaminhoMínimoEntreTodosOsPares* (algoritmo de Floyd) para encontrar as distâncias correspondentes a todos os caminhos mínimos. Mostre os valores sucessivos da matriz A para cada passagem do laço externo.

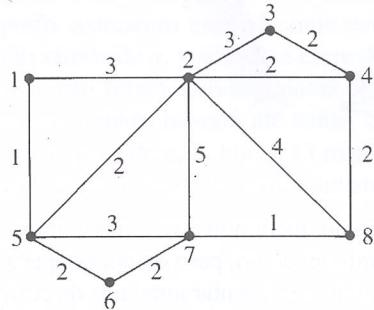
★15. Fig. 6.10.

16. Grafo para os Exercícios 1 a 4.

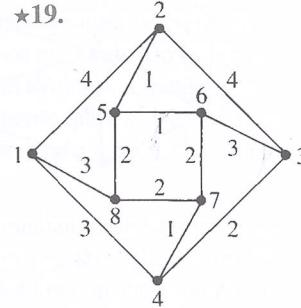
Para os Exercícios 17 a 20, use o algoritmo Prim para encontrar uma árvore geradora mínima para o grafo indicado.

17. Grafo para os Exercícios 1 a 4.

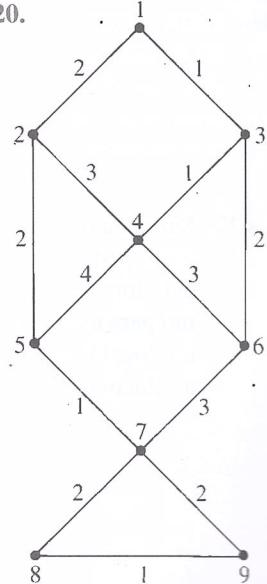
18.



19.



20.



O algoritmo de Kruskal é um outro algoritmo para encontrar uma árvore geradora mínima em um grafo conexo. Enquanto o algoritmo de Prim faz a árvore “crescer” a partir de um ponto arbitrário inicial incluindo arcos adjacentes associados a distâncias pequenas, o algoritmo de Kruskal inclui arcos em ordem

crescente de distância onde quer que estejam no grafo. Empates são resolvidos arbitrariamente. A única restrição é que um arco não é incluído se sua inclusão criar um ciclo. O algoritmo termina quando todos os nós estiverem incorporados em uma estrutura conexa. Uma descrição em pseudocódigo (bastante informal) é dada a seguir:

ALGORITMO OUTRAAGM

OutraAGM (matriz $n \times n$ A ; coleção de arcos T)

//Algoritmo de Kruskal para encontrar uma árvore geradora mínima;

//inicialmente, T é vazio; ao final, T = árvore geradora mínima.

ordene os arcos em G por distância em ordem crescente

repita

se próximo arco na ordem não completa um ciclo **então**

inclua esse arco em T

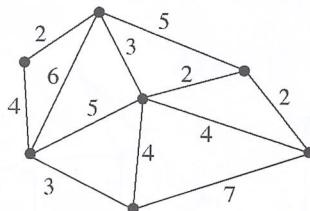
fim do se

até T ser conexo e conter todos os nós em G

fim de OutraAGM

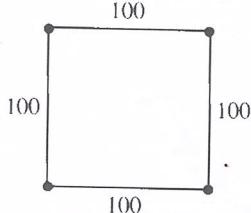
Para os Exercícios 21 a 24, use o algoritmo *OutraAGM* (algoritmo de Kruskal) para encontrar uma árvore geradora mínima.

- ★21. O grafo para os Exercícios 1 a 4.
- 23. O grafo para o Exercício 19.
- 22. O grafo para o Exercício 18.
- 24. O grafo para o Exercício 20.
- 25. Dê um exemplo para mostrar que a inclusão do nó mais próximo de IN em cada passo, como no algoritmo de Prim para encontrar a árvore geradora mínima, não garante um caminho menor.
- 26. Uma cidade está planejando construir caminhos para bicicletas ligando diversos parques municipais. A figura a seguir mostra um mapa com as distâncias entre os parques. (Alguns caminhos diretos teriam que cruzar ruas de muito movimento, de modo que essas distâncias não constam do mapa.) Encontre quais os caminhos que devem ser feitos de modo a ligar todos os parques com um custo mínimo.



- ★27. Suponha que os pesos dos arcos representam distâncias. Então, incluindo novos nós e arcos em um grafo pode resultar em uma árvore geradora para o novo grafo tendo um peso menor do que a árvore geradora para o grafo original. (A nova árvore geradora poderia representar uma rede de custo mínimo para uma rede de comunicação entre um grupo de cidades obtida adicionando-se uma chave em um local fora de todas as cidades.)

- Encontre uma árvore geradora de peso mínimo para o grafo com peso da figura. Qual é o peso?



- Coloque um nó no centro do quadrado. Inclua novos arcos do centro para os cantos. Encontre uma árvore geradora para o novo grafo e calcule seu peso (aproximado).

28. No início deste capítulo, você recebeu a seguinte tarefa:

Você é o administrador de uma rede, atuando em uma região extensa, que serve os diversos escritórios de sua companhia espalhados pelo país. As mensagens viajam através da rede roteadas de ponto a ponto até chegar ao seu destino. Cada nó na rede, portanto, funciona como uma estação distribuidora, recebendo e enviando mensagens para outros nós de acordo com um roteiro de distribuição mantido em cada nó. Algumas conexões na rede têm tráfego intenso, enquanto outras são menos usadas. A intensidade do tráfego pode variar dependendo da hora do dia; além disso, nós novos podem ser gerados e outros nós podem ser desativados. Portanto, você precisa atualizar periodicamente a informação contida em cada nó, de modo que ele possa transmitir mensagens ao longo do caminho mais eficiente (isto é, do que tem tráfego menos intenso).

Como calcular o roteiro de distribuição para cada nó?

Você percebe que pode representar a rede como um grafo com pesos, onde os arcos são as conexões entre os nós e os pesos dos arcos representam o tráfego nas conexões. O problema de roteamento torna-se, então, o de encontrar o caminho mínimo em um grafo de qualquer nó para qualquer outro nó. O algoritmo de Dijkstra pode ser usado para se obter o caminho mínimo de um dado nó arbitrário para todos os outros nós, de modo que você poderia usar o algoritmo repetidamente com nós iniciais diferentes. Ou você poderia usar o algoritmo *Caminho Mínimo Entre Todos Os Pares* (Exercício 15 acima), que parece ser mais curto e arrumado. Analise a ordem de grandeza de cada abordagem.

SEÇÃO 6.4 ALGORITMOS DE PERCURSO

Até agora este capítulo considerou diversos problemas de caminhos em um grafo G . Existe um caminho em G do nó x para o nó y ? Existe um caminho em G que usa todos os arcos uma vez? Existe um caminho em G que termina onde começa e passa por cada nó uma vez? Qual o caminho de peso mínimo entre x e y ? Nesta seção vamos tratar de um problema mais simples — só queremos escrever os nós de um grafo simples e conexo G em alguma ordem. Isso significa que precisamos encontrar um caminho que passa por cada nó pelo menos uma vez, mas podemos visitar um nó mais de uma vez se não o escrevermos de novo. Podemos também percorrer novamente arcos no grafo se necessário, e é claro que isso seria necessário se quiséssemos visitar cada nó em uma árvore. Esse processo é chamado de **percurso no grafo**. Já vimos diversos métodos para percursos em árvores (Seção 5.2). Os dois algoritmos nesta seção generalizam o percurso para ser aplicado em qualquer grafo.

BUSCA EM PROFUNDIDADE

No algoritmo de **busca em profundidade** para o percurso em um grafo, começamos em um nó arbitrário a do grafo, marcamos esse nó como tendo sido visitado e escrevemos esse nó. Percorremos, então, um caminho saindo de a , visitando e escrevendo os nós, indo tão longe quanto possível até não existirem nós que ainda não foram visitados nesse caminho. Voltamos, então, pelo caminho explorando, em cada nó, quaisquer caminhos laterais, até voltar, finalmente, para a . Depois exploramos quaisquer novos caminhos restantes a partir de a . A Fig. 6.13 mostra um grafo após a visita dos primeiros nós (marcados com um círculo) usando a busca em profundidade.

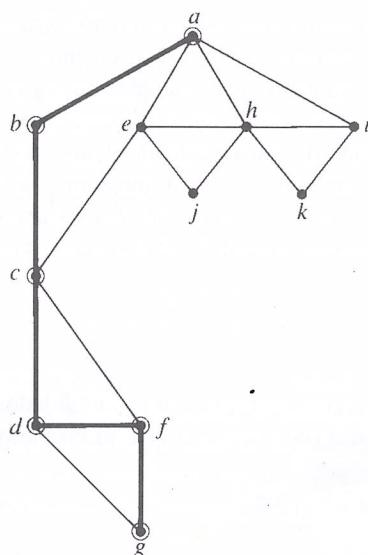


Fig. 6.13

Para uma descrição mais formal do algoritmo de busca em profundidade vamos usar recorrência, de modo que o algoritmo invoca a si mesmo durante sua execução. No algoritmo a seguir, os dados de entrada são um grafo simples e conexo G e um nó especificado a ; a saída é uma lista de todos os nós de G em ordem de profundidade de a .

```
ALGORITMO EmProfundidade
EmProfundidade(grafo G; nó a)
//Escreve os nós do grafo G em ordem de profundidade a partir do nó a.

    marque a como tendo sido visitado
    escreva(a)
    para cada nó n adjacente a a faça
        se nó n não tiver sido visitado então
            EmProfundidade(G, n)
        fim do se
    fim do para

fim de EmProfundidade
```

No passo de recorrência, o algoritmo é chamado com um novo nó especificado como ponto de partida. Não indicamos aqui como marcar os nós visitados ou como encontrar os nós n adjacentes a a .

Vamos aplicar a busca em profundidade ao grafo da Fig. 6.13, onde a é o nó inicial. Marcamos, primeiro, que visitamos a (ajuda, ao seguir a execução do algoritmo, colocar um círculo em um nó visitado) e depois escrevemos a . A seguir procuramos entre os nós adjacentes a a por um nó que ainda não tenha sido visitado. Aqui temos uma escolha (b , e , h e i); vamos escolher b . (Para todos obtermos as mesmas respostas, vamos combinar escolher o nó que venha primeiro em ordem alfabética quando tivermos uma escolha; na prática, a escolha seria determinada pelo modo de armazenamento dos vértices na representação do grafo.) Chamamos agora o algoritmo de busca em profundidade começando com o nó b .

Isso significa voltar para o início do algoritmo, onde o nó especificado agora é b ao invés de a . Então, marcamos primeiro b como tendo sido visitado e escrevemos b . Depois procuramos entre os nós adjacentes a b por um que não tenha sido marcado. O nó a é adjacente a b mas já foi marcado. O nó c satisfaz, logo chamamos o algoritmo começando com o nó c .

O nó c é marcado e escrito, e procuramos por nós adjacentes a c ainda não marcados. Por nossa convenção alfabética, escolhemos o nó d . Continuando dessa forma, visitamos a seguir o nó f e depois o g . Ao chegar no nó g , chegamos a um beco sem saída, já que não existem nós adjacentes ainda não visitados. Então, o laço de **para** do algoritmo chamado com o nó g está completo. (Nesse instante o grafo é como na Fig. 6.13.)

Terminamos o algoritmo para o nó g , mas o nó g era um dos nós adjacentes ao nó f , e ainda estamos dentro do laço de **para** dentro do algoritmo chamado com o nó f . Quando processamos f , g era o único nó adjacente ainda não visitado; completamos, então o laço de **para** e o algoritmo chamado com o nó f . Analogamente, voltando ao nó d , o algoritmo não encontra outros nós adjacentes não marcados e volta para a chamada do algoritmo com o nó c . Assim, após processar o nó d e tudo que veio depois dele até o beco sem saída, ainda estamos no laço de **para** chamado dentro do algoritmo com o nó c . Procuramos outros nós adjacentes a c ainda não marcados e encontramos um — o nó e . Aplicamos, então, o algoritmo de busca em profundidade ao nó e , o que nos leva aos nós h , i e k antes de chegar a outro beco sem saída. Voltando, temos um novo caminho a tentar a partir do nó h , o que nos leva ao nó j . A lista completa dos nós, na ordem em que foram escritos, é

$a, b, c, d, f, g, e, h, i, k, j$

O Exemplo 11 faz com que o processo de busca em profundidade pareça complexo, mas é mais fácil fazer a busca do que descrevê-la, como você vai descobrir no Problema Prático 14.

♦ EXEMPLO 11

PROBLEMA PRÁTICO 14

LEMBRETE:

Em uma busca em profundidade, vá tão longe quanto possível, depois volte, procurando qualquer caminho que tenha pulado na primeira passagem.

Escreva os nós em uma busca em profundidade no grafo da Fig. 6.14. Comece com o nó *a*.

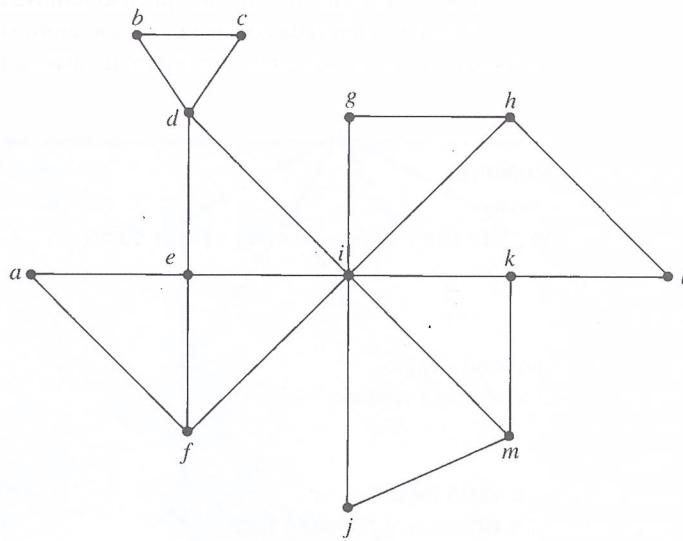


Fig. 6.14

BUSCA EM NÍVEL

Na **busca em nível**, começando em um nó arbitrário *a*, primeiro procuramos, a partir de *a*, todos os nós adjacentes, depois os nós adjacentes a esses, e assim por diante, quase como círculos concêntricos de ondas em um pequeno lago. A Fig. 6.15 mostra os primeiros nós visitados no mesmo grafo da Fig. 6.13, só que dessa vez usando busca em nível.

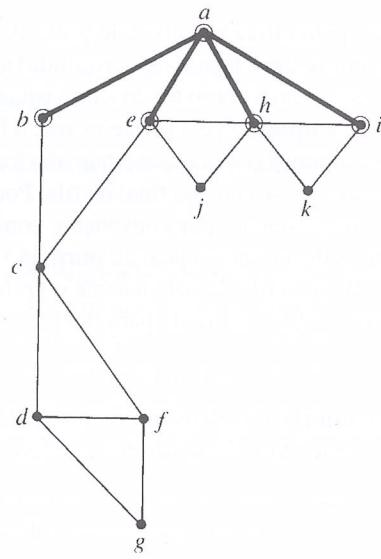


Fig. 6.15

Para escrever o algoritmo de busca em nível de maneira elegante, usaremos uma estrutura de **fila**. Uma fila é, simplesmente, uma lista onde os novos elementos entram no final e as saídas são pela frente. Uma fila em uma caixa de supermercado é um exemplo de uma fila de consumidores — um novo consumidor entra no final da fila e, à medida que os consumidores são atendidos, eles saem da frente da fila. A inclusão de um elemento no final de uma fila é chamada de uma operação de **inserção** e uma saída da frente da fila

é uma operação de **retirada**. Assim, o comando *insira(a, F)* inclui o elemento *a* no final da fila denominada *F*, e *retire(a, F)* retira o elemento atualmente na frente da fila *F*. Usaremos, também, uma função *frente(F)*, cujo resultado é o valor do elemento atualmente na frente da fila mas que não remove esses elementos. No algoritmo a seguir (algoritmo *EmNível*) os dados de entrada consistem em um grafo simples e conexo *G* e um nó especificado *a*; a saída é uma lista de todos os nós em *G* em ordem de nível a partir de *a*.

ALGORITMO EmNível

EmNível(grafo G; nó a)

//Escreve os nós do grafo *G* em ordem de nível a partir do nó *a*.

Variáveis locais:

fila de nós *F*

```

    initialize F como sendo vazio
    marque a como tendo sido visitado
    escreva(a)
    insira(a, F)
enquanto F não é vazio faça
    para cada nó n adjacente a frente(F) faça
        se n não foi visitado então
            marque n como tendo sido visitado
            escreva n
            insira(n, F)
        fim do se
    fim do para
    retire(F)
fim do enquanto
fim de EmNível
```

Vamos seguir os passos do algoritmo para busca em nível do grafo da Fig. 6.15 começando com o nó *a* (esse é o mesmo grafo em que fizemos uma busca em profundidade no Exemplo 11). Começamos inicializando uma fila vazia *F*, marcando o nó *a* como tendo sido visitado, escrevendo esse nó e inserindo-o na fila. Quando chegamos ao laço de **enquanto** pela primeira vez, a fila não está vazia e *a* é o elemento na frente da fila. No laço de **para**, procuramos por nós que ainda não foram visitados e que sejam adjacentes a *a*, escrevemos esses nós e colocamos esses nós no final da fila. Podemos ter uma escolha de nós para visitar aqui; como anteriormente, e simplesmente por convenção, concordamos em visitá-los em ordem alfabética. Assim, a primeira vez que completamos o laço de **para**, já visitamos e escrevemos os nós *b*, *e*, *h* e *i*, nessa ordem, e os inserimos no final da fila. Nesse instante o grafo fica como na Fig. 6.15. Removemos, então, *a* da frente da fila, que fica, então (da frente para trás)

b, e, h, i

Na próxima iteração do laço de **enquanto**, *b* é o elemento na frente da fila e o laço de **para** procura nós ainda não visitados adjacentes a *b*. O único que ainda não foi visitado é *c*, que é escrito e incluído na fila. Após a remoção de *b*, a fila contém

e, h, i, c

Executando o laço de **enquanto** novamente, *e* está na frente da fila. Uma busca dos nós adjacentes a *e* produz um nó novo, *j*. O grafo agora fica como na Fig. 6.16 e, após a retirada de *e*, a fila contém

h, i, c, j

Ao procurar nós adjacentes a *h*, encontramos um nó novo, *k*. Ao procurar nós adjacentes a *i*, não é inserido nenhum nó novo na fila. Quando *c* é o primeiro elemento da fila, uma busca de nós adjacentes a *c* produz dois nós novos, *d* e *f*. Após inserir esses na fila (e remover *c*), a fila contém

j, k, d, f

♦ EXEMPLO 12

Procurando nós adjacentes a j e depois a k não é inserido nenhum nó novo na fila. Quando d está na frente da fila, encontramos um nó novo g e a fila (após a remoção de d) fica

f, g

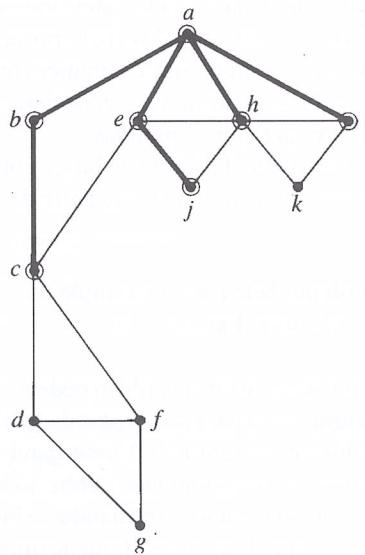


Fig. 6.16

Processando f e depois g , não encontramos nenhum nó novo. Após a retirada de g da fila, a fila fica vazia. O laço de **enquanto** — assim como o algoritmo — termina. A lista de nós escrita por esse processo, isto é, a lista de nós em ordem de nível a partir de a , é

$a, b, e, h, i, c, j, k, d, f, g$

Como a busca em profundidade, a busca em nível não é difícil de fazer; basta guardar os nós que já foram visitados e o conteúdo atual da fila.

PROBLEMA PRÁTICO 15

ANÁLISE

Quanto trabalho é feito pelas buscas em profundidade e em nível? Ambos os algoritmos procuram todos os nós ainda não visitados adjacentes a um determinado nó. Suponha que o grafo contém n nós e m arcos. Uma das vantagens de representar um grafo como uma lista de adjacência ao invés de uma matriz de adjacência é que essa operação particular é mais eficiente; para encontrar nós adjacentes ao nó n , é preciso percorrer a lista de adjacência de i , que pode ser curta, ao invés da linha i da matriz de adjacência, que tem que conter n elementos. Vamos supor, portanto, que o grafo é representado por uma lista de adjacência.

Na busca em nível, o algoritmo procura, na mesma etapa, todos os elementos da lista de adjacência inteira do nó na frente da fila, marcando, escrevendo e inserindo os nós ainda não visitados encontrados. Na busca em profundidade, o algoritmo pode ser interrompido muitas vezes ao percorrer a lista de adjacência de um dado nó (devido à recorrência) e processa partes das listas de adjacência de outros nós. Ao final, no entanto, todas as listas de adjacência foram completamente percorridas.

Percorrer as listas de adjacência do grafo nos dá a quantidade de trabalho feito por qualquer um dos algoritmos. Existem n listas de adjacência, de modo que o trabalho é, pelo menos, da ordem de $\Theta(n)$, já que todas as listas de adjacência têm que ser verificadas, mesmo que sejam vazias. Como existem m arcos, o trabalho em percorrer o comprimento total de todas as listas de adjacência é, pelo menos, $\Theta(m)$. Logo, ambos os algoritmos de busca em profundidade e de busca em nível são algoritmos da ordem de $\Theta(\max(n, m))$. Se existem mais arcos do que nós (o caso mais comum), então $\Theta(\max(n, m)) = \Theta(m)$.

APLICAÇÕES

Buscas em profundidade e em nível podem ser usadas como a base para a execução de outras tarefas relacionadas a grafos, algumas das quais resolvemos anteriormente. Pode-se associar, a cada busca, uma estrutura de árvore sem raiz que é um subgrafo do grafo original. Ao percorrer a lista de adjacência do nó i , se o nó j é adjacente ao nó i e ainda não foi visitado, então o arco $i-j$ é acrescentado a esse subgrafo. Como não são usados arcos que ligam a um nó já visitado anteriormente, são evitados ciclos e o subgrafo é uma árvore sem raiz. Como todos os nós são visitados (pela primeira vez), essas árvores são árvores geradoras para o grafo. Cada árvore tem $n - 1$ arcos, que é o número mínimo de arcos para conectar n nós. Estamos supondo aqui que os arcos não têm peso mas, se considerarmos os arcos com peso, todos eles iguais a 1, então essas árvores são árvores geradoras mínimas.

As linhas mais grossas na Fig. 6.13 são parte da árvore de busca em profundidade associada à busca do Exemplo 11 e as linhas mais grossas nas Fig. 6.15 e 6.16 são parte da árvore de busca em nível associada à busca do Exemplo 12.

- Complete a árvore de busca em profundidade para o Exemplo 11.
- Complete a árvore de busca em nível para o Exemplo 12.

PROBLEMA
PRÁTICO 16

Os algoritmos de busca em profundidade e em nível também podem ser aplicados em grafos direcionados e, nesse processo, fornecem um algoritmo novo para a acessibilidade. Para determinar se o nó j é acessível do nó i , faça uma busca em profundidade (ou em nível) começando com o nó i ; quando o algoritmo terminar, verifique se o nó j foi visitado. A acessibilidade “entre todos os pares”, isto é, quais nós são acessíveis de quais nós, pode ser determinada, então, executando-se buscas em profundidade ou em nível usando cada nó, por sua vez, como o nó fonte. Isso necessaria de um trabalho da ordem de $\Theta(n * \max(n, m))$. Se o grafo for muito esparsa, caso em que temos $\max(n, m) = n$, teríamos um algoritmo da ordem de $\Theta(n^2)$ para a acessibilidade. Lembre-se de que o algoritmo de Warshall (Seção 6.1) era de ordem $\Theta(n^3)$. Essa melhora ocorre porque, em um grafo esparsa, a maior parte das listas de adjacência serão curtas ou vazias, enquanto que o algoritmo de Warshall processa todos os elementos na matriz de adjacência, inclusive os nulos. Mas, se o grafo não for esparsa, o número de arcos pode ser de ordem $\Theta(n^2)$ e, nesse caso, $\Theta(n * \max(n, m)) = \Theta(n^3)$, da mesma ordem que o algoritmo de Warshall. Além disso, o algoritmo de Warshall tem a vantagem de ter uma implementação sucinta.

Na Seção 4.2 definimos ordenação topológica como uma maneira de estender uma ordem parcial em um conjunto finito a uma ordem total. Suponha que o conjunto parcialmente ordenado é representado por um grafo direcionado. A ordenação topológica será obtida contando-se os nós; vamos supor, então, que o valor inicial da contagem é 0. Escolha um nó como fonte e faça uma busca em profundidade a partir desse nó. Sempre que a busca volta de um nó pela última vez, atribua àquele nó o próximo número na contagem. Quando o algoritmo de busca em profundidade termina, escolha um nó ainda não visitado (se existir) como fonte para outra busca em profundidade e continue a incrementar o número de contagem. Continue esse processo até não existir nós que não tenham sido visitados no grafo. Uma ordenação topológica resulta ao se ordenar os nós na ordem inversa de seus números de contagem. Esse processo de ordenação topológica funciona porque atribuímos o número de contagem ao voltar de um nó pela última vez. Seu número de contagem será, então, maior do que todos os números de todos os nós acessíveis a partir dele, isto é, todos os nós dos quais ele é um predecessor na ordem parcial.

A Fig. 6.17a contém um grafo direcionado que representa uma ordem parcial. Escolhendo d (arbitrariamente) como o nó fonte e executando um algoritmo de busca em profundidade, visitamos e e f , quando temos que voltar. Ao nó f é atribuído o número 1, mas ainda não acabamos com e , pois podemos visitar g a partir dele. Voltando de g , atribuimos a g o número 2. Agora voltamos de e pela última vez e atribuímos o número 3 a e , e depois o número 4 a d . Escolha o nó a como fonte para uma outra busca. Visitamos o nó c e depois temos que voltar para a , logo são atribuídos a c e a , respectivamente, os números 5 e 6. Começando com b como nó fonte, não há nenhum lugar para se ir, logo atribuímos a b o número 7. Não há nós no grafo que ainda não tenham sido visitados, logo o processo pára. O esquema de numeração é mostrado na Fig. 6.17b.

EXEMPLO 13

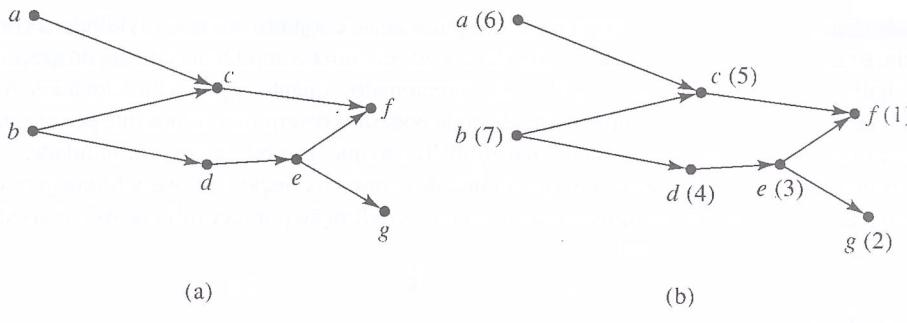


Fig. 6.17

Ordenando os números em ordem inversa, obtemos

$$\begin{array}{ccccccc} 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ b & a & c & d & e & g & f \end{array}$$

que é uma ordenação topológica.

PROBLEMA PRÁTICO 17

Use o algoritmo de busca em profundidade para executar uma ordenação topológica no grafo da Fig. 6.18. Indique os números de contagem no grafo.

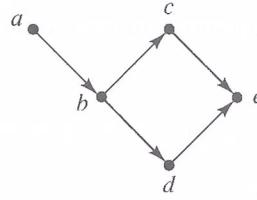


Fig. 6.18

Vamos considerar agora um grafo G (não direcionado) que não precisa ser conexo. Uma **componente conexa** de G é um subgrafo de G que é, ao mesmo tempo, conexo e não é subgrafo de nenhum subgrafo maior conexo. O grafo na Fig. 6.19 tem três componentes conexas. É claro que, se o grafo original for conexo, ele tem apenas uma componente conexa.

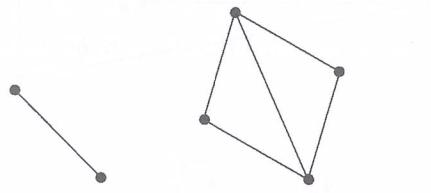


Fig. 6.19

Os algoritmos de busca em profundidade e em nível podem ser usados para encontrar as componentes conexas de um grafo. Escolhemos um nó arbitrário como fonte e efetuamos uma busca. Quando o algoritmo terminar, todos os nós visitados pertencem a uma componente. Encontramos, então, um nó no grafo que não tenha sido visitado e efetuamos outra busca, que vai produzir uma segunda componente. Continuamos esse processo até não haver nenhum nó não visitado no grafo.

Embora tenhamos definido acessibilidade apenas para grafos direcionados, o conceito também faz sentido para grafos não direcionados e não conexos. Vamos considerar apenas grafos simples não direcionados e não conexos, mas vamos usar a convenção de que, embora não exista nenhum laço, cada nó é acessível de si mesmo. A acessibilidade torna-se, então, uma relação de equivalência no conjunto de nós do grafo; nossa convenção impõe a reflexividade, enquanto que a simetria e a transitividade são válidas porque o grafo não

é direcionado. Essa relação de equivalência gera uma partição no conjunto dos nós, dividindo-o em classes de equivalência, e cada classe de equivalência consiste nos nós em uma componente conexa do grafo. O algoritmo de Warshall pode ser aplicado tanto a grafos não direcionados, quanto a grafos direcionados. A utilização do algoritmo de Warshall resulta em uma matriz de onde podemos determinar os nós que pertencem às várias componentes conexas, mas isso necessita de mais trabalho do que usar busca em profundidade.

Como observação final sobre busca em profundidade, vimos na Seção 1.5 que a linguagem de programação Prolog, ao processar uma pesquisa baseada em uma definição por recorrência, usa uma estratégia de busca em profundidade (Exemplo 40).

SEÇÃO 6.4 REVISÃO

TÉCNICAS

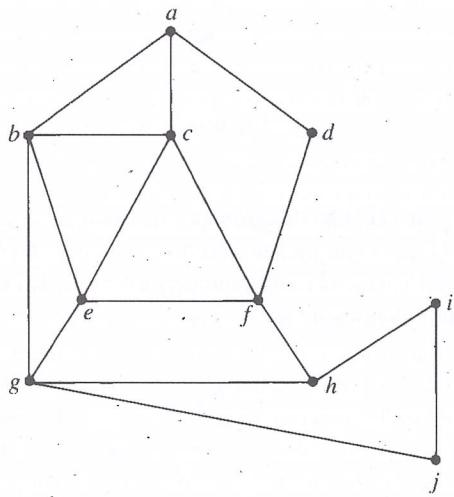
- W ♦ Conduzir uma busca em profundidade em um grafo.
- W ♦ Conduzir uma busca em nível em um grafo.

IDÉIAS PRINCIPAIS

- ♦ Existem algoritmos para visitar os nós de um grafo sistematicamente.
- ♦ Buscas em profundidade e em nível podem ser usadas como base para outras tarefas.

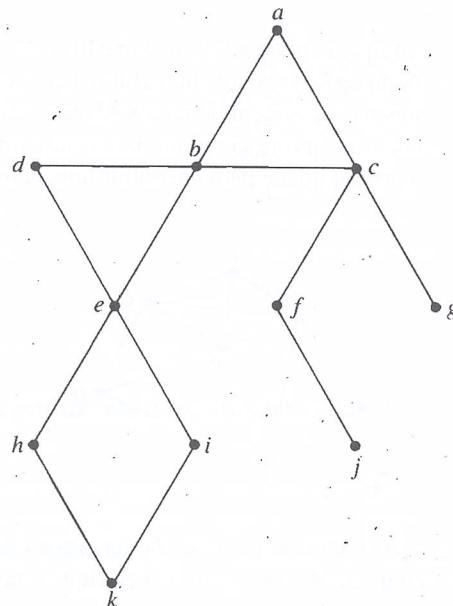
EXERCÍCIOS 6.4

Para os Exercícios 1 a 6, escreva os nós do grafo na figura a seguir em ordem de profundidade, começando com o nó especificado.



- ★1. a 2. c 3. d 4. g ★5. e 6. h

Para os Exercícios 7 a 10, escreva os nós do grafo na figura a seguir em ordem de profundidade, começando com o nó especificado.



★7. a 8. e ★9. f 10. h

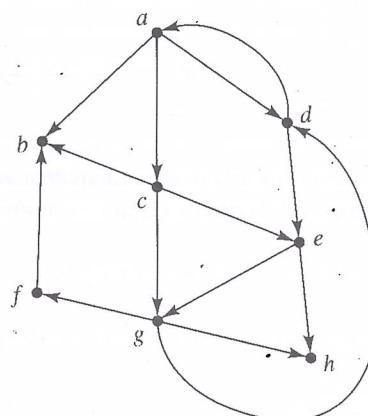
Para os Exercícios 11 a 16, escreva os nós do grafo na figura para os Exercícios 1 a 6 em ordem de nível, começando com o nó especificado.

★11. a 12. c 13. d
14. g 15. e 16. h

Para os Exercícios 17 a 20, escreva os nós do grafo na figura para os Exercícios 7 a 10 em ordem de nível, começando com o nó especificado.

★17. a 18. e
19. f 20. h

Para os Exercícios 21 a 23, escreva os nós do grafo na figura a seguir em ordem de profundidade, começando com o nó especificado.

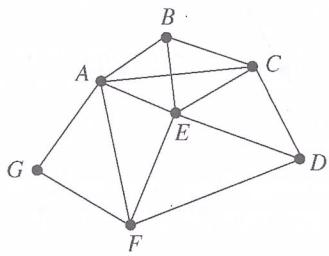


★21. a 22. g 23. f

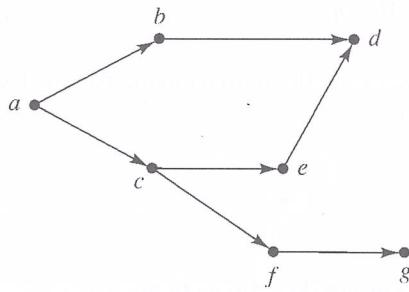
Para os Exercícios 24 a 26, escreva os nós do grafo na figura para os Exercícios 21 a 23 em ordem de nível, começando com o nó especificado.

★24. *a* 25. *g* 26. *f*

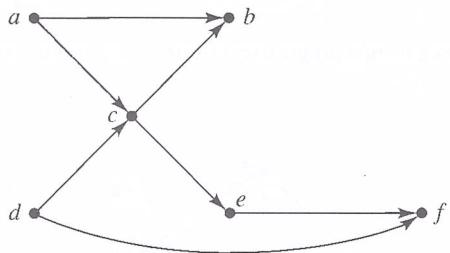
27. Na rede de computadores ilustrada na figura a seguir, a mesma mensagem deve ser enviada do nó *C* para os nós *A*, *E*, *F* e *G*. Um modo de fazer isso é encontrar o menor caminho de *C* a cada um dos nós e enviar diversas cópias da mesma mensagem. Uma abordagem mais eficiente é enviar uma cópia da mensagem de *C* ao longo de uma árvore geradora para o subgrafo contendo os nós envolvidos. Use o algoritmo de busca em profundidade para encontrar uma árvore geradora para o subgrafo.



- ★28. Use o algoritmo de busca em profundidade para fazer uma ordenação topológica no grafo da figura a seguir. Indique, no grafo, o número de contagem. Especifique o nó inicial ou nós para a busca.



29. Use o algoritmo de busca em profundidade para fazer uma ordenação topológica no grafo da figura a seguir. Indique, no grafo, o número de contagem. Especifique o nó inicial ou nós para a busca.



30. Encontre uma maneira de percorrer uma árvore em ordem de nível, isto é, de modo que todos os nós no mesmo nível são listados da esquerda para a direita, com profundidade crescente. (Sugestão: já temos um modo de fazer isso.)