24.3 Algoritmo de Dijkstra

O algoritmo de Dijkstra resolve o problema de caminhos mais curtos de única origem em um grafo orientado ponderado $G = (V, E)$ para o caso no qual todos os pesos de arestas são não negativos. Então, nesta seção, iremos supor que $w(u, v) \geq 0$ para cada aresta $(u, v) \in E$. Como veremos, com uma boa implementação, o tempo de execução do algoritmo de Dijkstra é inferior ao do algoritmo de Bellman-Ford.

O algoritmo de Dijkstra mantém um conjunto S de vértices cujos pesos finais de caminhos mais curtos desde a origem s já foram determinados. O algoritmo seleciona repetidamente o vértice $u \geq V - S$ com a estimativa mínima de caminhos mais curtos, adiciona u a S e relaxa todas as arestas que saem de u. Na implementação a seguir, manteremos uma fila de prioridade mínima Q de vértices, tendo como chaves seus valores de d.

```
DIJKSTRA(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 S ← ∅
3 Q ← V[G] → fila de prioridade de todos os vértices de G
4 while Q ≠ ∅
5    do u ← EXTRACT-MIN(Q) → menor d[]
6       S ← S ∪ {u}
7       for cada vértice v ∈ Adj[u]
8           do RELAX(u, v, w)
```

O algoritmo de Dijkstra relaxa arestas como mostra a Figura 24.6. A linha 1 executa a inicialização habitual dos valores de d e π, e a linha 2 inicializa o conjunto S como o conjunto vazio. O algoritmo mantém o invariante de que $Q = V - S$ no início de cada iteração do loop **while** das linhas 4 a 8. A linha 3 inicializa a fila de prioridade mínima Q para conter todos os vértices em V; tendo em vista que $S = \emptyset$ nesse momento, o invariante é verdadeiro após a linha 3. Em cada passagem pelo loop **while** das linhas 4 a 8, um vértice u é extraído de $Q = V - S$ e inserido no conjunto S, mantendo assim o invariante. (Na primeira passagem por esse loop, $u = s$.) Então, o vértice u tem a menor estimativa de caminhos mais curtos em comparação com qualquer vértice em $V - S$. Em seguida, as linhas 7 e 8 relaxam cada aresta (u, v) que saem de u, atualizando assim a estimativa $d[v]$ e o predecessor $\pi[v]$ se o caminho mais curto até v pode ser melhorado mediante a passagem por u. Observe que os vértices nunca são inseridos em Q após a linha 3, e que cada vértice é extraído de Q e inserido em S exatamente uma vez, de modo que o loop **while** das linhas 4 a 8 iterage exatamente $|V|$ vezes.

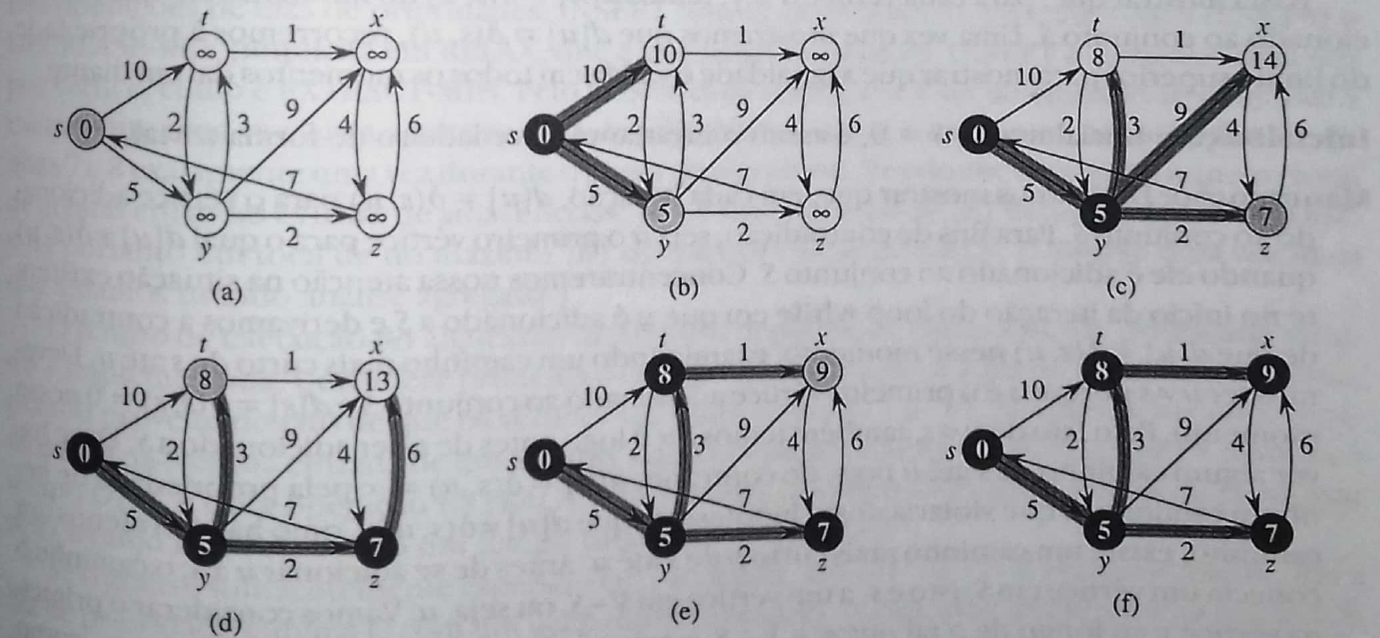

FIGURA 24.6 A execução do algoritmo de Dijkstra. A origem s é o vértice mais à esquerda. As estimativas de caminhos mais curtos são mostradas dentro dos vértices, e as arestas sombreadas indicam valores de predecessores. Vértices pretos estão no conjunto S, e vértices brancos estão na fila de prioridade mínima $Q = V - S$. **(a)** A situação imediatamente antes da primeira iteração do loop **while** das linhas 4 a 8. O vértice sombreado tem o valor de d mínimo e é escolhido como vértice u na linha 5. **(b)–(f)** A situação após cada iteração sucessiva do loop **while**. O vértice sombreado em cada parte é escolhido como vértice u na linha 5 da próxima iteração. Os valores de d e π mostrados na parte (f) são os valores finais

Pelo fato do algoritmo de Dijkstra sempre escolher o vértice "mais leve" ou "mais próximo" em $V-S$ para adicionar ao conjunto S, dizemos que ele utiliza uma estratégia gulosa. As estratégias gulosas são apresentadas em detalhes no Capítulo 16, mas você não precisa ler aquele capítulo para entender o algoritmo de Dijkstra. As estratégias gulosas nem sempre produzem resultados ótimos em geral, mas, como mostram o teorema a seguir e seu corolário, o algoritmo de Dijkstra realmente calcula caminhos mais curtos. A chave é mostrar que cada vez que um vértice u é inserido no conjunto S, temos $d[u] = \delta(s, u)$.

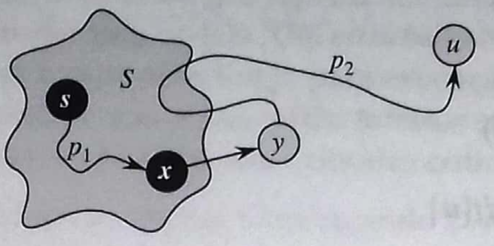

FIGURA 24.7 A prova do Teorema 24.6. O conjunto S é não vazio imediatamente antes do vértice u ser inserido nele. Um caminho mais curto p desde a origem s até o vértice u pode ser decomposto em $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, onde y é o primeiro vértice sobre o caminho que não está em S e $x \in S$ precede imediatamente y. Os vértices x e y são distintos, mas podemos ter $s = x$ ou $y = u$. O caminho p_2 pode reentrar ou não no conjunto S

***Teorema 24.6* (*Correção do algoritmo de Dijkstra*)**
Se executarmos o algoritmo de Dijkstra sobre um grafo orientado ponderado $G = (V, E)$ com função peso não negativa w e origem s, ele termina com $d[u] = \delta(s, u)$ para todos os vértices $u \in V$.

Prova Usamos o loop invariante a seguir:

No início de cada iteração do loop **while** das linhas 4 a 8, $d[v] = \delta(s, v)$ para cada vértice $v \in S$.

Basta mostrar que, para cada vértice $u \in V$, temos $d[u] = \delta(s, u)$ no momento em que u é adicionado ao conjunto S. Uma vez que mostramos que $d[u] = \delta(s, u)$, recorremos à propriedade do limite superior para mostrar que a igualdade é válida em todos os momentos daí em diante.

Inicialização: Inicialmente, $S = \emptyset$, e assim o invariante é verdadeiro de forma trivial.

Manutenção: Desejamos mostrar que, em cada iteração, $d[u] = \delta(s, u)$ para o vértice adicionado ao conjunto S. Para fins de contradição, seja u o primeiro vértice para o qual $d[u] \neq \delta(s, u)$ quando ele é adicionado ao conjunto S. Concentraremos nossa atenção na situação existente no início da iteração do loop **while** em que u é adicionado a S e derivamos a contradição de que $d[u] = \delta(s, u)$ nesse momento, examinando um caminho mais curto de s até u. Devemos ter $u \neq s$ porque s é o primeiro vértice adicionado ao conjunto S e $d[s] = \delta(s, s) = 0$ nesse momento. Pelo fato de $u \neq s$, também temos $S \neq \emptyset$ logo antes de u ser adicionado a S. Deve haver algum caminho de s até u pois, do contrário, $d[u] = \delta(s, u) = \infty$ pela propriedade de nenhum caminho, o que violaria nossa hipótese de que $d[u] \neq \delta(s, u)$. Como há pelo menos um caminho, existe um caminho mais curto p de s até u. Antes de se adicionar u a S, o caminho p conecta um vértice em S, isto é s, a um vértice em $V-S$, ou seja, u. Vamos considerar o primeiro vértice y ao longo de p tal que $y \in V-S$, e seja $x \in S$ o predecessor de y. Portanto, como mostra a Figura 24.7, o caminho p pode ser decomposto como $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$. (Um dos caminhos p_1 ou p_2 pode não ter nenhuma aresta.)

Afirmamos que $d[y] = \delta(s, y)$ quando u é adicionado a S. Para provar essa afirmação, observe que $x \in S$. Assim, como u foi escolhido como o primeiro vértice para o qual $d[u] \neq \delta(s, u)$ quando foi adicionado a S, tínhamos $d[x] = \delta(s, x)$ quando x foi adicionado a S. A aresta (x, y) foi relaxado nesse momento, e assim a afirmação decorre da propriedade de convergência.

Podemos agora obter uma contradição para provar que $d[u] = \delta(s, u)$. Como y ocorre antes de u em um caminho mais curto de s para u e todos os pesos de arestas são não negativos (especialmente das arestas do caminho p_2), temos $\delta(s, y) \le \delta(s, u)$ e, desse modo,

$$
\begin{aligned}
d[y] &= \delta(s, y) \\
&\le \delta(s, u) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (24.2) \\
&\le d[u] \qquad \text{(pela propriedade do limite superior)} .
\end{aligned}
$$

Porém, como ambos os vértices u e y estavam em $V - S$ quando u foi escolhido na linha 5, temos $d[u] \le d[y]$. Desse modo, as duas desigualdades em (24.2) são de fato igualdades, dando

$$d[y] = \delta(s, y) = \delta(s, u) = d[u] .$$

Conseqüentemente, $d[u] = \delta(s, u)$, o que contradiz nossa escolha de u. Concluímos que $d[u] = \delta(s, u)$ quando u é adicionado a S, e que essa igualdade é mantida em todos os momentos daí em diante.

Término: No término, $Q = \emptyset$ e, juntamente com nosso invariante anterior de que $Q = V - S$, isso implica que $S = V$. Desse modo, $d[u] = \delta(s, u)$ para todos os vértices $u \in V$. ■

Corolário 24.7

Se executarmos o algoritmo de Dijkstra sobre um grafo orientado ponderado $G = (V, E)$ com função peso não negativa w e origem s, então, no término, o subgrafo predecessor G_π será uma árvore de caminhos mais curtos com raiz em s.

Prova Imediata a partir do Teorema 24.6 e da propriedade de subgrafo predecessor. ■

Análise

Qual é a rapidez do algoritmo de Dijkstra? Ele mantém a fila de prioridade mínima Q chamando três operações de filas de prioridades: INSERT (implícita na linha 3), EXTRACT-MIN (linha 5) e DECREASE-KEY (implícita em RELAX, que é chamado na linha 8). INSERT é invocado uma vez por vértice, como é EXTRACT-MIN. Pelo fato de cada vértice $v \in V$ ser adicionado ao conjunto S exatamente uma vez, cada aresta na lista de adjacências $Adj[v]$ é examinada no loop **for** das linhas 7 e 8 exatamente uma vez durante o curso do algoritmo. Tendo em vista que o número total de arestas em todas as listas de adjacências é $|E|$, existe um total de $|E|$ iterações desse loop **for**, e há portanto um total de no máximo $|E|$ operações DECREASE-KEY. (Observe uma vez mais que estamos usando análise agregada.)

O tempo de execução do algoritmo de Dijkstra depende de como a fila de prioridade mínima é implementada. Considere primeiro o caso no qual mantemos a fila de prioridade mínima, tirando proveito do fato de que os vértices são numerados de 1 a $|V|$. Simplesmente armazenamos $d[v]$ na v-ésima entrada de um arranjo. Cada operação INSERT e DECREASE-KEY demora o tempo $O(1)$, e cada operação EXTRACT-MIN demora o tempo $O(V)$ (pois temos de pesquisar pelo arranjo inteiro), dando um tempo total $O(V^2 + E) = O(V^2)$.

Se o grafo é suficientemente esparso – em particular, $E = o(V^2/\lg V)$ – é prático implementar a fila de prioridade mínima Q com um heap mínimo binário. (Conforme discutimos na Seção 6.5, um detalhe de implementação importante é que os vértices e os elementos do heap correspondentes devem manter descritores um para o outro.) Cada operação EXTRACT-MIN demora então o tempo $O(\lg V)$. Como antes, existem $|V|$ dessas operações. O tempo para construir o heap mínimo binário é $O(V)$. Cada operação DECREASE-KEY demora o tempo $O(\lg V)$, e ainda há no máximo $|E|$ de tais operações. Então, o tempo de execução total é $O((V + E) \lg V)$, que é $O(E \lg V)$ se todos os vértices são acessíveis a partir da origem. Esse tempo de execução é uma melhoria sobre o tempo $O(V^2)$ de implementação direta se $E = o(V^2/\lg V)$.

Podemos de fato alcançar um tempo de execução igual a $O(V \lg V + E)$ implementando a fila de prioridade mínima Q com um heap de Fibonacci (ver Capítulo 20). O custo amortizado de cada uma das $|V|$ operações EXTRACT-MIN é $O(\lg V)$, e cada chamadas de DECREASE-KEY, das quais existe no máximo $|E|$, demora apenas o tempo amortizado $O(1)$. Historicamente, o desenvolvimento de heaps de Fibonacci foi motivado pela observação de que, no algoritmo de Dijkstra, existem tipicamente muito mais chamadas DECREASE-KEY que chamadas EXTRACT-MIN; assim, qualquer método de redução do tempo amortizado de cada operação DECREASE-KEY para $o(\lg V)$ sem aumentar o tempo amortizado de EXTRACT-MIN produziria uma implementação assintoticamente mais rápida do que aquela que utiliza heaps binários.

O algoritmo de Dijkstra exibe alguma semelhança, tanto em relação à busca em largura (ver Seção 22.2) quanto em relação ao algoritmo de Prim para calcular árvores espalhadas mínimas (ver Seção 23.2). Ele é semelhante à busca em largura no fato de que o conjunto S corresponde ao conjunto de vértices pretos em uma busca em largura; exatamente como os vértices em S têm seus pesos finais de caminhos mais curtos, os vértices pretos em uma busca em largura têm suas distâncias corretas primeiro na extensão. O algoritmo de Dijkstra é semelhante ao algoritmo de Prim no fato de que ambos os algoritmos usam uma fila de prioridade mínima para encontrar o vértice "mais leve" fora de um conjunto dado (o conjunto S no algoritmo de Dijkstra, e a árvore que está sendo aumentada no algoritmo de Prim), inserem esse vértice no conjunto e ajustam os pesos dos vértices restantes fora do conjunto de acordo com ele.

Exercícios

24.3-1

Execute o algoritmo de Dijkstra sobre o grafo orientado da Figura 24.2, primeiro usando o vértice s como origem, e depois usando o vértice y como origem. No estilo da Figura 24.6, mostre os valores de d e π e os vértices no conjunto S após cada iteração do loop **while**.

24.3-2

Forneça um exemplo simples de um grafo orientado com arestas de peso negativo para o qual o algoritmo de Dijkstra produza respostas incorretas. Por que a prova do Teorema 24.6 não é válida quando são permitidas arestas de peso negativo?

24.3-3

Suponha que mudamos a linha 4 do algoritmo de Dijkstra para o seguinte:

4 **while** $|Q| > 1$

Essa mudança faz o loop **while** ser executado $|V| - 1$ vezes em lugar de $|V|$ vezes. Esse algoritmo proposto é correto?

24.3-4

Temos um grafo orientado $G = (V, E)$ no qual cada aresta $(u, v) \in E$ tem um valor associado $r(u, v)$, o qual é um número real no intervalo $0 \leq r(u, v) \leq 1$ que representa a confiabilidade de um canal de comunicação do vértice u até o vértice v. Interpretamos $r(u, v)$ como a probabilidade de que o canal de u até v não venha a falhar, e supomos que essas probabilidades são independentes. Forneça um algoritmo eficiente para encontrar o caminho mais confiável entre dois vértices dados.

24.3-5

Seja $G = (V, E)$ um grafo orientado ponderado com função peso $w : E \rightarrow \{1, 2, ..., W\}$ para algum inteiro positivo W, e suponha que não existam dois vértices com os mesmos pesos de caminhos mais curtos a partir do vértice de origem s. Agora, suponha que definimos um grafo orientado não ponderado $G' = (V \cup V', E')$ substituindo cada aresta $(u, v) \in E$ por $w(u, v)$ arestas de peso unitário em série. Quantos vértices G' tem? Suponha agora que executamos uma busca em