24.1 The Bellman-Ford algorithm

The ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

Figure 24.4 shows the execution of the Bellman-Ford algorithm on a graph with 5 vertices. After initializing the d and π values of all vertices in line 1, the algorithm makes $|V| - 1$ passes over the edges of the graph. Each pass is one iteration of the **for** loop of lines 2–4 and consists of relaxing each edge of the graph once. Figures 24.4(b)–(e) show the state of the algorithm after each of the four passes over the edges. After making $|V| - 1$ passes, lines 5–8 check for a negative-weight cycle and return the appropriate boolean value. (We'll see a little later why this check works.)

The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(E)$ time, and the **for** loop of lines 5–7 takes $O(E)$ time.

To prove the correctness of the Bellman-Ford algorithm, we start by showing that if there are no negative-weight cycles, the algorithm computes correct shortest-path weights for all vertices reachable from the source.

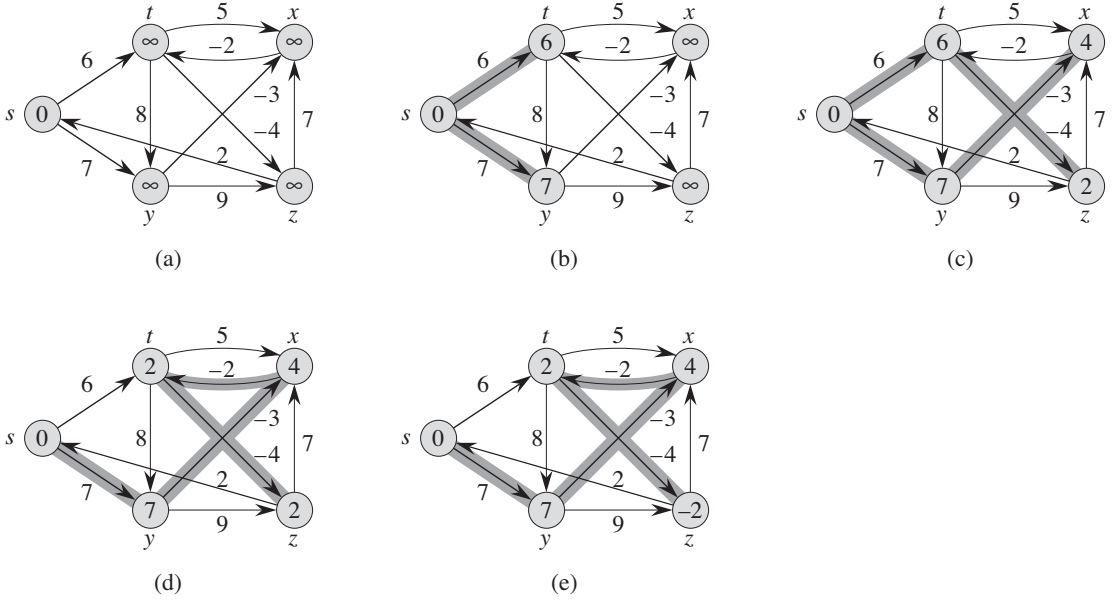

Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

Lemma 24.2

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the **for** loop of lines 2–4 of BELLMAN-FORD, we have $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

Proof We prove the lemma by appealing to the path-relaxation property. Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V| - 1$ edges, and so $k \le |V| - 1$. Each of the $|V| - 1$ iterations of the **for** loop of lines 2–4 relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i). By the path-relaxation property, therefore, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ■

Corollary 24.3
Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, for each vertex $v \in V$, there is a path from s to v if and only if BELLMAN-FORD terminates with $v.d < \infty$ when it is run on G.

Proof The proof is left as Exercise 24.1-2. ■

Theorem 24.4 (Correctness of the Bellman-Ford algorithm)
Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof Suppose that graph G contains no negative-weight cycles that are reachable from the source s. We first prove the claim that at termination, $v.d = \delta(s, v)$ for all vertices $v \in V$. If vertex v is reachable from s, then Lemma 24.2 proves this claim. If v is not reachable from s, then the claim follows from the no-path property. Thus, the claim is proven. The predecessor-subgraph property, along with the claim, implies that G_π is a shortest-paths tree. Now we use the claim to show that BELLMAN-FORD returns TRUE. At termination, we have for all edges $(u, v) \in E$,

$$\begin{aligned} v.d &= \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \quad \text{(by the triangle inequality)} \\ &= u.d + w(u, v) , \end{aligned}$$

and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

Now, suppose that graph G contains a negative-weight cycle that is reachable from the source s; let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. Then,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 . \tag{24.1}$$

Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE. Thus, $v_i.d \le v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing the inequalities around cycle c gives us

$$\begin{aligned}\sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k}(v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) \ .\end{aligned}$$

Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d$, and so

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d \ .$$

Moreover, by Corollary 24.3, $v_i.d$ is finite for $i = 1, 2, \ldots, k$. Thus,

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) \ ,$$

which contradicts inequality (24.1). We conclude that the Bellman-Ford algorithm returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise. ■

Exercises

24.1-1

Run the Bellman-Ford algorithm on the directed graph of Figure 24.4, using vertex z as the source. In each pass, relax edges in the same order as in the figure, and show the d and π values after each pass. Now, change the weight of edge (z, x) to 4 and run the algorithm again, using s as the source.

24.1-2

Prove Corollary 24.3.

24.1-3

Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

24.1-4

Modify the Bellman-Ford algorithm so that it sets $v.d$ to $-\infty$ for all vertices v for which there is a negative-weight cycle on some path from the source to v.

24.1-5 ⋆

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$. Give an $O(VE)$-time algorithm to find, for each vertex $v \in V$, the value $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$.

24.1-6 ⋆

Suppose that a weighted, directed graph $G = (V, E)$ has a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

24.2 Single-source shortest paths in directed acyclic graphs

By relaxing the edges of a weighted dag (directed acyclic graph) $G = (V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\Theta(V + E)$ time. Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

The algorithm starts by topologically sorting the dag (see Section 22.4) to impose a linear ordering on the vertices. If the dag contains a path from vertex u to vertex v, then u precedes v in the topological sort. We make just one pass over the vertices in the topologically sorted order. As we process each vertex, we relax each edge that leaves the vertex.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

Figure 24.5 shows the execution of this algorithm.

The running time of this algorithm is easy to analyze. As shown in Section 22.4, the topological sort of line 1 takes $\Theta(V + E)$ time. The call of INITIALIZE-SINGLE-SOURCE in line 2 takes $\Theta(V)$ time. The **for** loop of lines 3–5 makes one iteration per vertex. Altogether, the **for** loop of lines 4–5 relaxes each edge exactly once. (We have used an aggregate analysis here.) Because each iteration of the inner **for** loop takes $\Theta(1)$ time, the total running time is $\Theta(V + E)$, which is linear in the size of an adjacency-list representation of the graph.

The following theorem shows that the DAG-SHORTEST-PATHS procedure correctly computes the shortest paths.

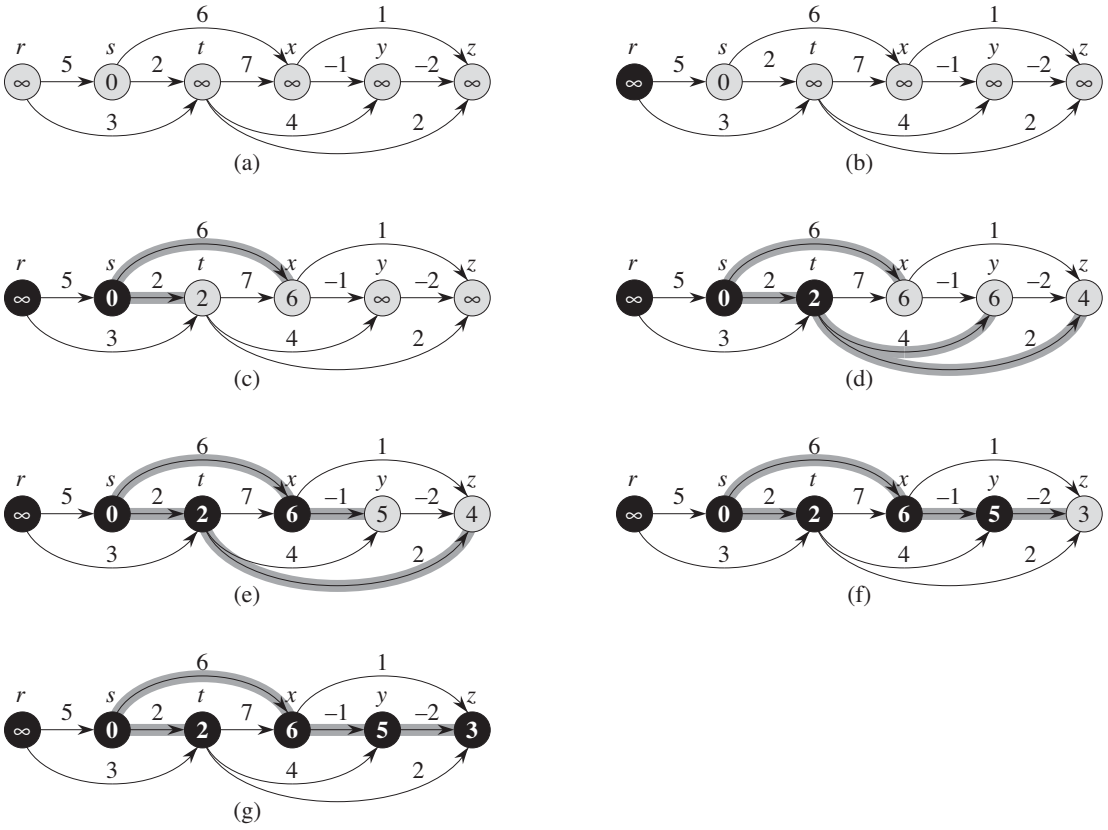

Figure 24.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values appear within the vertices, and shaded edges indicate the π values. **(a)** The situation before the first iteration of the **for** loop of lines 3–5. **(b)–(g)** The situation after each iteration of the **for** loop of lines 3–5. The newly blackened vertex in each iteration was used as u in that iteration. The values shown in part (g) are the final values.

Theorem 24.5

If a weighted, directed graph $G = (V, E)$ has source vertex s and no cycles, then at the termination of the DAG-SHORTEST-PATHS procedure, $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree.

Proof We first show that $v.d = \delta(s, v)$ for all vertices $v \in V$ at termination. If v is not reachable from s, then $v.d = \delta(s, v) = \infty$ by the no-path property. Now, suppose that v is reachable from s, so that there is a shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$. Because we pro-

cess the vertices in topologically sorted order, we relax the edges on p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. The path-relaxation property implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 0, 1, \ldots, k$. Finally, by the predecessor-subgraph property, G_π is a shortest-paths tree. ■

An interesting application of this algorithm arises in determining critical paths in ***PERT chart***[2] analysis. Edges represent jobs to be performed, and edge weights represent the times required to perform particular jobs. If edge (u, v) enters vertex v and edge (v, x) leaves v, then job (u, v) must be performed before job (v, x). A path through this dag represents a sequence of jobs that must be performed in a particular order. A ***critical path*** is a *longest* path through the dag, corresponding to the longest time to perform any sequence of jobs. Thus, the weight of a critical path provides a lower bound on the total time to perform all the jobs. We can find a critical path by either

- negating the edge weights and running DAG-SHORTEST-PATHS, or
- running DAG-SHORTEST-PATHS, with the modification that we replace "∞" by "$-\infty$" in line 2 of INITIALIZE-SINGLE-SOURCE and "$>$" by "$<$" in the RELAX procedure.

Exercises

24.2-1
Run DAG-SHORTEST-PATHS on the directed graph of Figure 24.5, using vertex r as the source.

24.2-2
Suppose we change line 3 of DAG-SHORTEST-PATHS to read

3 **for** the first $|V| - 1$ vertices, taken in topologically sorted order

Show that the procedure would remain correct.

24.2-3
The PERT chart formulation given above is somewhat unnatural. In a more natural structure, vertices would represent jobs and edges would represent sequencing constraints; that is, edge (u, v) would indicate that job u must be performed before job v. We would then assign weights to vertices, not edges. Modify the DAG-SHORTEST-PATHS procedure so that it finds a longest path in a directed acyclic graph with weighted vertices in linear time.

---

[2] "PERT" is an acronym for "program evaluation and review technique."

本身存放在一起，这样在遍历每条邻接链表时，我们可以在 $O(1)$时间内获得边的权重。 650

24.1　Bellman-Ford 算法

Bellman-Ford 算法解决的是一般情况下的单源最短路径问题，在这里，边的权重可以为负值。给定带权重的有向图 $G=(V, E)$和权重函数 w：$E\rightarrow\mathbf{R}$，Bellman-Ford 算法返回一个布尔值，以表明是否存在一个从源结点可以到达的权重为负值的环路。如果存在这样一个环路，算法将告诉我们不存在解决方案。如果没有这种环路存在，算法将给出最短路径和它们的权重。

Bellman-Ford 算法通过对边进行松弛操作来渐近地降低从源结点 s 到每个结点 v 的最短路径的估计值 $v.d$，直到该估计值与实际的最短路径权重 $\delta(s, v)$相同时为止。该算法返回 TRUE 值当且仅当输入图不包含可以从源结点到达的权重为负值的环路。

```
BELLMAN-FORD(G,w,s)
1  INITIALIZE-SINGLE-SOURCE(G,s)
2  for i=1to|G.V|-1
3      for each edge(u,v)∈G.E
4          RELAX(u,v,w)
5  for each edge(u,v)∈G.E
6      if v.d>u.d+w(u,v)
7          return FALSE
8  return TRUE
```

图 24-4 描述的是在有 5 个结点的图上运行 Bellman-Ford 算法的过程。在算法第 1 行对所有结点的 d 值和 π 值进行初始化后，算法对图的每条边进行 $|V|-1$ 次处理。每一次处理对应的是算法第 2～4 行 **for** 循环的一次循环，该循环对图的每条边进行一次松弛操作。图 24-4(b)～(e)描述的是对边进行 4 次松弛操作时，每一次松弛后的算法状态。在进行了 $|V|-1$ 次松弛操作后，算法第 5～8 行负责检查图中是否存在权重为负值的环路并返回与之相适应的布尔值。(我们将在稍后的篇幅里看到该检查为什么是正确的。)

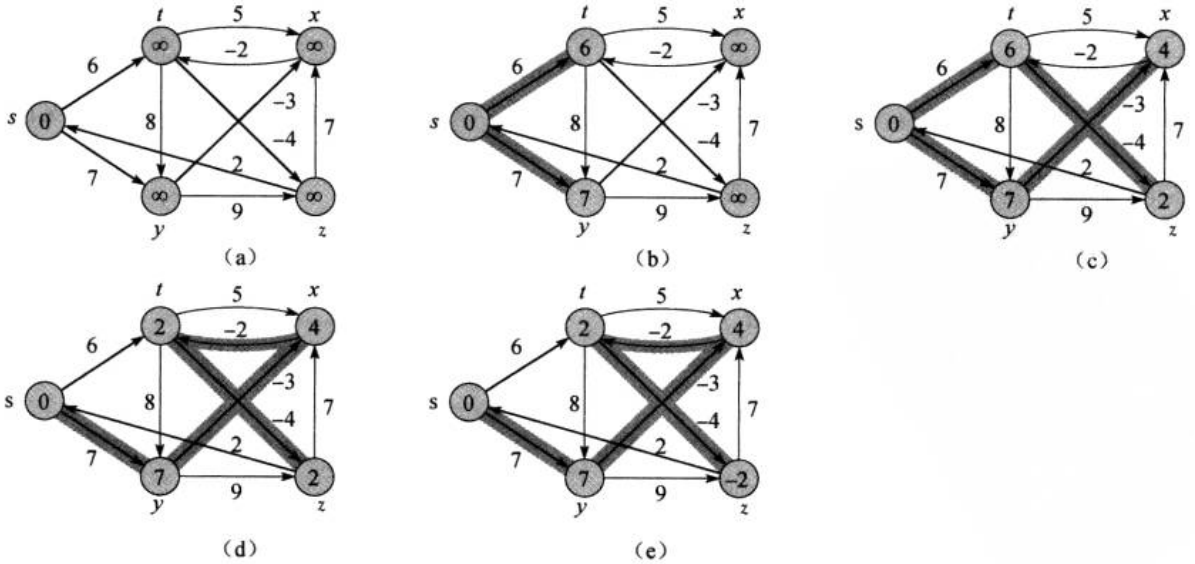

图 24-4　Bellman-Ford 算法的执行过程。源结点为 s，结点中的数值为该结点的 d 值，加了阴影的边表示前驱值：如果边(u, v)加了阴影，则 $v.\pi=u$。在本图的例子中，每一次的松弛操作对边的处理次序都是：(t, x)，(t, y)，(t, z)，(y, x)，(y, z)，(z, x)，(z, s)，(s, t)，(s, y)。(a)在第 1 次松弛操作前的场景。(b)～(e)在对边进行每次松弛操作后的场景。图(e)中的 d 值和 π 值为最终取值。在本例中，Bellman-Ford 算法返回的值为 TRUE

由于算法第1行的初始化操作所需时间为 $\Theta(V)$，第2～4行循环的运行时间为 $\Theta(E)$，且一共要进行 $|V|-1$ 次循环，第5～7行的 **for** 循环所需时间为 $O(E)$，Bellman-Ford 算法的总运行时间为 $O(VE)$。

要证明 Bellman-Ford 算法的正确性，首先证明在没有权重为负值的环路的情况下，该算法正确计算出从源结点可以到达的所有结点之间的最短路径权重。

引理 24.2 设 $G=(V, E)$ 为一个带权重的源结点为 s 的有向图，其权重函数为 $w: E\to\mathbf{R}$。假定图 G 不包含从源结点 s 可以到达的权重为负值的环路。那么在算法 BELLMAN-FORD 的第2～4行的 **for** 循环执行了 $|V|-1$ 次之后，对于所有从源结点 s 可以到达的结点 v，我们有 $v.d=\delta(s, v)$。

证明 我们通过使用路径松弛性质来证明本引理。考虑任意从源结点 s 可以到达的结点 v，设 $p=\langle v_0, v_1, \cdots, v_k\rangle$ 为从源结点 s 到结点 v 之间的任意一条最短路径，这里 $v_0=s$，$v_k=v$。因为最短路径都是简单路径，p 最多包含 $|V|-1$ 条边，因此 $k\leqslant|V|-1$。算法第2～4行的 **for** 循环每次松弛所有的 $|E|$ 条边。在第 i 次松弛操作时，这里 $i=1, 2, \cdots, k$，被松弛的边中包含边 (v_{i-1}, v_i)。根据路径松弛性质，$v.d=v_k.d=\delta(s, v_k)=\delta(s, v)$。 ■

651 ～ 652

推论 24.3 设 $G=(V, E)$ 是一带权重的源结点为 s 的有向图，其权重函数为 $w: E\to\mathbf{R}$。假定图 G 不包含从源结点 s 可以到达的权重为负值的环路，则对于所有结点 $v\in V$，存在一条从源结点 s 到结点 v 的路径当且仅当 BELLMAN-FORD 算法终止时有 $v.d<\infty$。

证明 该证明留给读者作为练习(请参阅练习 24.1-2)。 ■

定理 24.4(Bellman-Ford 算法的正确性) 设 BELLMAN-FORD 算法运行在一带权重的源结点为 s 的有向图 $G=(V, E)$ 上，该图的权重函数为 $w: E\to\mathbf{R}$。如果图 G 不包含从源结点 s 可以到达的权重为负值的环路，则算法将返回 TRUE 值，且对于所有结点 $v\in V$，前驱子图 G_π 是一棵根结点为 s 的最短路径树。如果图 G 包含一条从源结点 s 可以到达的权重为负值的环路，则算法将返回 FALSE 值。

证明 假定图 G 不包含从源结点 s 可以到达的权重为负值的环路。我们首先证明，对于所有结点 $v\in V$，在算法 BELLMAN-FORD 终止时，我们有 $v.d=\delta(s, v)$。如果结点 v 是从 s 可以到达的，则引理 24.2 证明了本论断。如果结点 v 不能从 s 到达，则该论断可以从非路径性质获得。因此，该论断得到证明。综合前驱子图性质和本论断可以推导出 G_π 是一棵最短路径树。现在，我们使用这个论断来证明 BELLMAN-FORD 算法返回的是 TRUE 值。在算法 BELLMAN-FORD 终止时，对于所有的边 $(u, v)\in E$，我们有

$$\begin{aligned} v.d &= \delta(s,v) \\ &\leqslant \delta(s,u)+w(u,v) \quad \text{(根据三角不等式)} \\ &= u.d+w(u,v) \end{aligned}$$

因此，算法第6行中没有任何测试可以让 BELLMAN-FORD 算法返回 FALSE 值。因此，它一定返回的是 TRUE 值。

现在，假定图 G 包含一个权重为负值的环路，并且该环路可以从源结点 s 到达；设该环路为 $c=\langle v_0, v_1, \cdots, v_k\rangle$，这里 $v_0=v_k$，则有

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \tag{24.1}$$

653 下面使用反证法。假设 Bellman-Ford 算法返回的是 TRUE 值，则 $v_i.d\leqslant v_{i-1}.d+w(v_{i-1}, v_i)$，这里 $i=1, 2, \cdots, k$。将环路 c 上的所有这种不等式加起来，我们有

$$\sum_{i=1}^{k} v_i.d \leqslant \sum_{i=1}^{k} (v_{i-1}.d+w(v_{i-1}, v_i)) = \sum_{i=1}^{k} v_{i-1}.d+\sum_{i=1}^{k} w(v_{i-1}, v_i)$$

由于 $v_0=v_k$，环路 c 上面的每个结点在上述求和表达式 $\sum_{i=1}^{k} v_i.d$ 和 $\sum_{i=1}^{k} v_{i-1}.d$ 中刚好各出现一次，因此有

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$$

而且，根据推论 24.3，$v_i.d$ 对于 $i=1, 2, \cdots, k$ 来说取的都是有限值，因此有

$$0 \leqslant \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

而这与不等式(24.1)矛盾。因此，我们得出结论，如果图 G 不包含从源结点 s 可以到达的权重为负值的环路，则 Bellman-Ford 算法返回 TRUE 值，否则返回 FALSE 值。 ■

练习

24.1-1 在图 24-4 上运行 Bellman-Ford 算法，使用结点 z 作为源结点。在每一遍松弛过程中，以图中相同的次序对每条边进行松弛，给出每遍松弛操作后的 d 值和 π 值。然后，把边 (z, x) 的权重改为 4，再次运行该算法，这次使用 s 作为源结点。

24.1-2 证明推论 24.3。

24.1-3 给定 $G=(V, E)$ 是一带权重且没有权重为负值的环路的有向图，对于所有结点 $v \in V$，从源结点 s 到结点 v 之间的最短路径中，包含边的条数的最大值为 m。(这里，判断最短路径的根据是权重，不是边的条数。)请对算法 BELLMAN-FORD 进行简单修改，可以让其在 $m+1$ 遍松弛操作之后终止，即使 m 不是事先知道的一个数值。

24.1-4 修改 Bellman-Ford 算法，使其对于所有结点 v 来说，如果从源结点 s 到结点 v 的一条路径上存在权重为负值的环路，则将 $v.d$ 的值设置为 $-\infty$。 654

***24.1-5** 设 $G=(V, E)$ 为一带权重的有向图，其权重函数为 $w: E \rightarrow \mathbf{R}$。请给出一个时间复杂度为 $O(VE)$ 的算法，对于每个结点 $v \in V$，计算出数值 $\delta^*(v)=\min_{u \in V}\{\delta(u, v)\}$。

***24.1-6** 假定 $G=(V, E)$ 为一带权重的有向图，并且图中存在一个权重为负值的环路。给出一个有效的算法来列出所有属于该环路上的结点。请证明算法的正确性。

24.2 有向无环图中的单源最短路径问题

根据结点的拓扑排序次序来对带权重的有向无环图 $G=(V, E)$ 进行边的松弛操作，我们便可以在 $\Theta(V+E)$ 时间内计算出从单个源结点到所有结点之间的最短路径。在有向无环图中，即使存在权重为负值的边，但因为没有权重为负值的环路，最短路径都是存在的。

我们的算法先对有向无环图进行拓扑排序(请参阅 22.4 节)，以便确定结点之间的一个线性次序。如果有向无环图包含从结点 u 到结点 v 的一条路径，则 u 在拓扑排序的次序中位于结点 v 的前面。我们只需要按照拓扑排序的次序对结点进行一遍处理即可。每次对一个结点进行处理时，我们对从该结点发出的所有的边进行松弛操作。

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4     for each vertex v ∈ G.Adj[u]
5        RELAX(u, v, w)
```

图 24-5 描述的是算法 DAG-SHORTEST-PATHS 的执行过程。

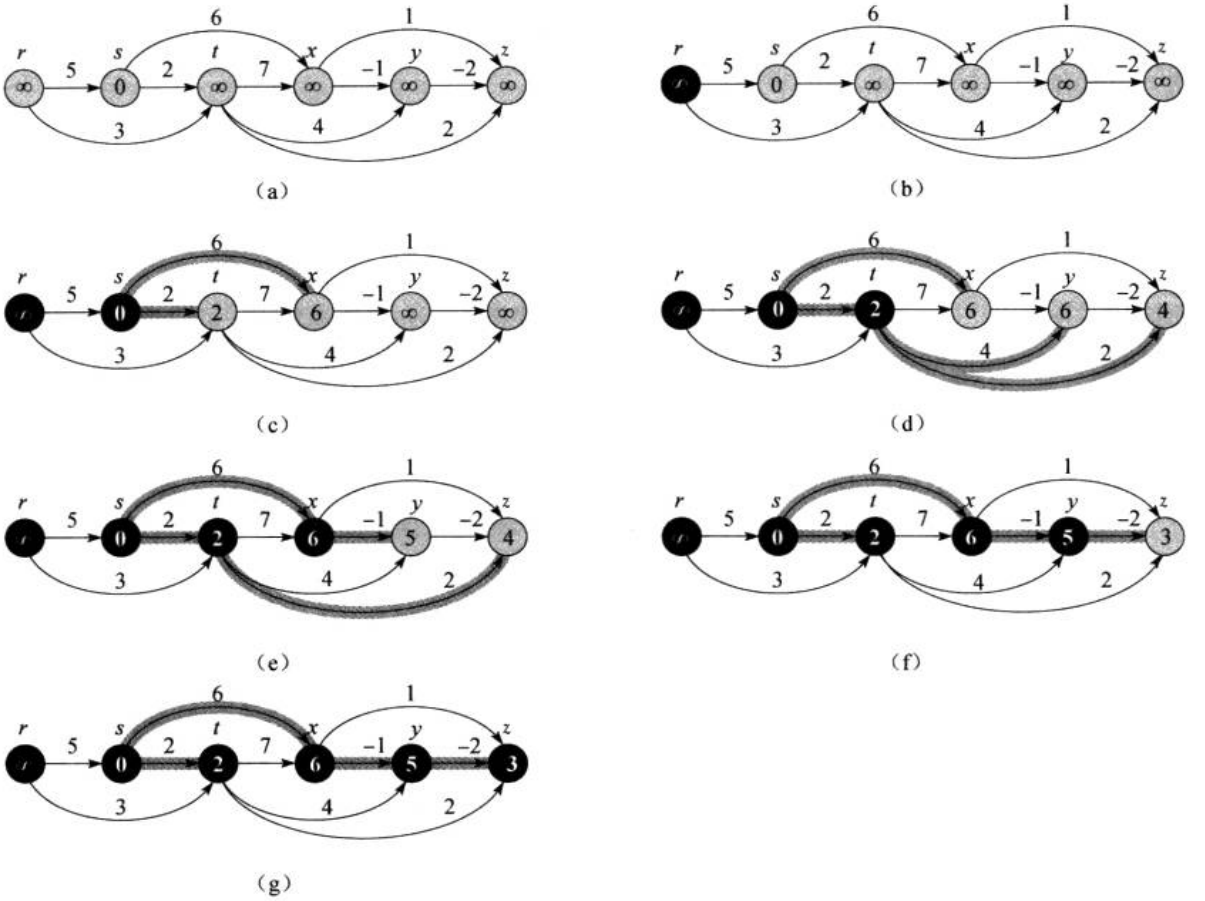

图 24-5 在有向无环图上执行最短路径算法 DAG-SHORTEST-PATHS 的过程。图中的结点从左至右以拓扑排序的次序排列。源结点为 s，每个结点中的数值为 d 值，加了阴影的边表示 π 值。(a)在算法第 3～5 行 **for** 循环开始前的场景。(b)～(g)第 3～5 行 **for** 循环在每次执行后的场景。每次循环时新变为黑色的结点作为该次循环里的 u 结点。图(g)中所显示的各种值都是最后的取值

该算法的运行时间非常容易分析。如 22.4 节所描述的，算法第 1 行的拓扑排序时间为 $\Theta(V+E)$。第 2 行对 INITIALIZE-SINGLE-SOURCE 的调用所需时间为 $\Theta(V)$。第 3～5 行的 **for** 循环(外循环)对于每个结点执行一遍，因此，第 4～5 行的 **for** 循环(内循环)对每条边刚好松弛一次。(注意，我们这里使用了聚合分析。)因为内循环每次的运行时间为 $\Theta(1)$，算法的总运行时间为 $\Theta(V+E)$。对于以邻接链表法表示的图来说，这个时间为线性级。

下面的定理将证明 DAG-SHORTEST-PATHS 过程正确计算出所有的最短路径。

定理 24.5 *如果带权重无环路的有向图 $G=(V, E)$ 有一个源结点 s，则在算法 DAG-SHORTEST-PATHS 终止时，对于所有的结点 $v\in V$，我们有 $v.d=\delta(s, v)$，且前驱子图 G_π 是一棵最短路径树。*

证明 首先证明对于所有的结点 $v\in V$，在算法 DAG-SHORTEST-PATHS 终止时都有 $v.d=\delta(s, v)$。如果结点 v 不能从源结点 s 到达，则根据非路径性质有 $v.d=\delta(s, v)=\infty$。现在假定结点 v 可以从结点 s 到达，因此，图中存在一条最短路径 $p=\langle v_0, v_1, \cdots, v_k\rangle$，这里 $v_0=s$，$v_k=v$。因为算法是按照拓扑排序的次序来对结点进行处理，所以对路径 p 上的边的放松次序为 (v_0, v_1)，(v_1, v_2)，$\cdots$，(v_{k-1}, v_k)。根据路径松弛性质，对于 $i=0, 1, \cdots, k$，在算法终止时有 $v_i.d=\delta(s, v_i)$。最后，根据前驱子图性质，G_π 是一棵最短路径树。 ■

655
~
656

算法 DAG-SHORTEST-PATHS 的一个有趣的应用是在 **PERT** 图[⊖]的分析中进行关键路径的判断。PERT 图是一个有向无环图，在这种图中，每条边代表需要进行的工作，边上的权重代表执行该工作所需要的时间。如果边(u, v)进入结点 v，边(v, x)离开结点 v(从结点 v 发出)，则工作(u, v)必须在工作(v, x)前完成。PERT 图中的一条路径代表的是一个工作执行序列。**关键路径**则是该有向无环图中一条最长的路径，该条路径代表执行任何工作序列所需要的最长时间。因此，关键路径上的权重提供的是执行所有工作所需时间的下界。我们可以使用下面两种办法中的任意一种来找到 PERT 图中的关键路径：

- 将所有权重变为负数，然后运行 DAG-SHORTEST-PATHS。
- 运行 DAG-SHORTEST-PATHS，但进行如下修改：在 INITIALIZE-SINGLE-SOURCE 的第 2 行将∞替换为$-\infty$，在 RELAX 过程中将“$>$”替换为“$<$”。

练习

24.2-1　请在图 24-5 上运行 DAG-SHORTEST-PATHS，使用结点 r 作为源结点。

24.2-2　假定将 DAG-SHORTEST-PATHS 的第 3 行改为：

3　**for** the first $|V|-1$ vertices, taken in topologically sorted order

证明：该算法的正确性保持不变。

24.2-3　上面描述的 PERT 图的公式有一点不太自然。在一个更自然的结构下，图中的结点代表要执行的工作，边代表工作之间的次序限制，即边(u, v)表示工作 u 必须在工作 v 之前执行。在这种结构的图中，我们将权重赋给结点，而不是边。请修改 DAG-SHORTEST-PATHS 过程，使得其可以在线性时间内找出这种有向无环图中一条最长的路径。

657

24.2-4　给出一个有效的算法来计算一个有向无环图中的路径总数。分析你自己的算法。

24.3　Dijkstra 算法

Dijkstra 算法解决的是带权重的有向图上单源最短路径问题，该算法要求所有边的权重都为非负值。因此，在本节的讨论中，我们假定对于所有的边$(u, v) \in E$，都有 $w(u, v) \geqslant 0$。我们稍后将看到，如果所采用的实现方式合适，Dijkstra 算法的运行时间要低于 Bellman-Ford 算法的运行时间。

Dijkstra 算法在运行过程中维持的关键信息是一组结点集合 S。从源结点 s 到该集合中每个结点之间的最短路径已经被找到。算法重复从结点集 $V-S$ 中选择最短路径估计最小的结点 u，将 u 加入到集合 S，然后对所有从 u 发出的边进行松弛。在下面给出的实现方式中，我们使用一个最小优先队列 Q 来保存结点集合，每个结点的关键值为其 d 值。

```
DIJKSTRA.(G,w,s)
1  INITIALIZE-SINGLE-SOURCE(G,s)
2  S=∅
3  Q=G.V
4  while Q≠∅
5      u=EXTRACT-MIN(Q)
6      S=S∪{u}
7      for each vertex v∈G.Adj[u]
8          RELAX(u,v,w)
```

⊖ “PERT”是“Program Evaluation and Review Technique”的缩写。