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 $w : E \rightarrow \mathbb{R}$, 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.

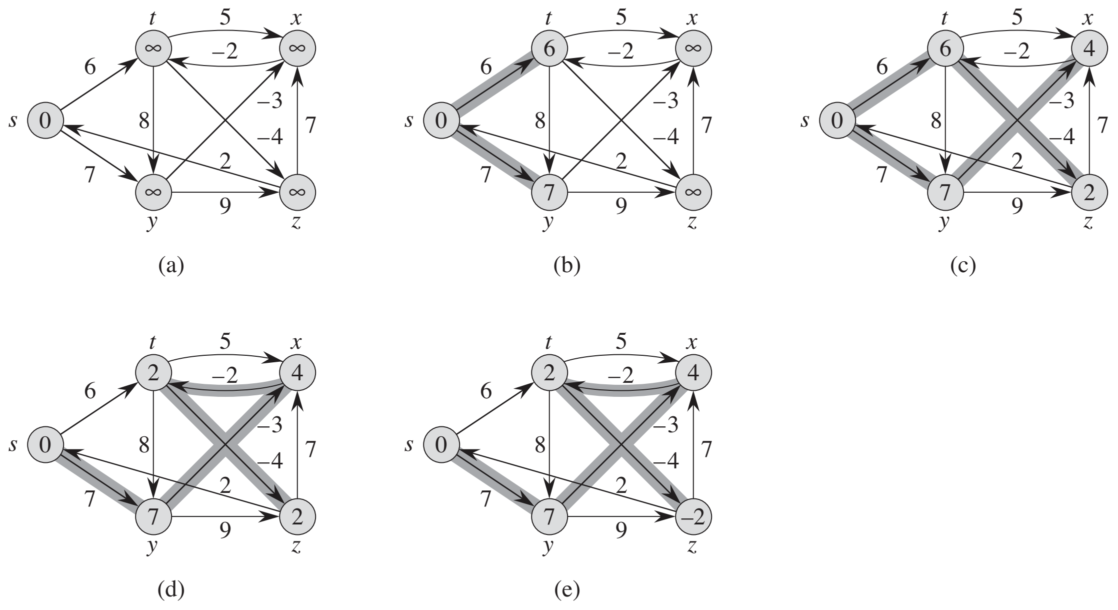

Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(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 $(u, v) \in E$,

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

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, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing the inequalities around cycle c gives us

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

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,

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

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.

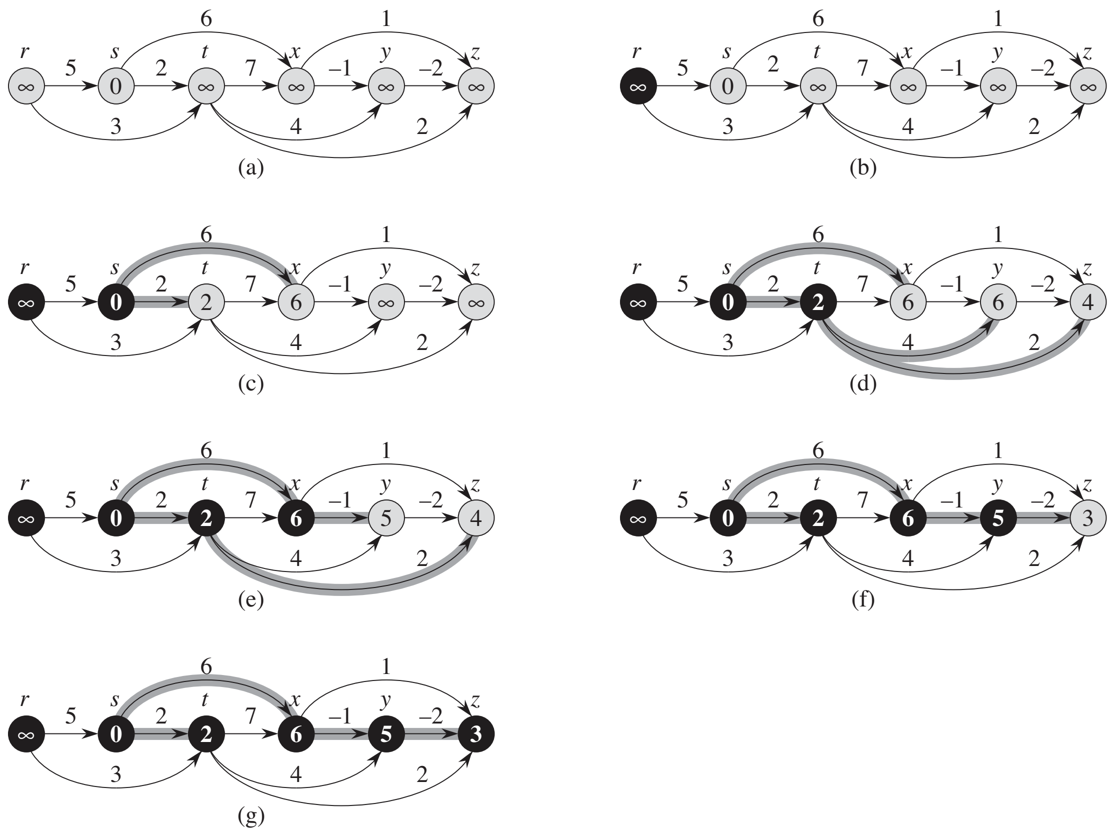

Figure 24.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values appear within the vertices, and shaded edges indicate the π values. **(a)** The situation before the first iteration 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 process 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."

24.2-4

Give an efficient algorithm to count the total number of paths in a directed acyclic graph. Analyze your algorithm.

24.3 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

```
DIJKSTRA($G, w, s$)
1  INITIALIZE-SINGLE-SOURCE($G, s$)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN($Q$)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX($u, v, w$)
```

Dijkstra's algorithm relaxes edges as shown in Figure 24.6. Line 1 initializes the d and π values in the usual way, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 4–8. Line 3 initializes the min-priority queue Q to contain all the vertices in V; since $S = \emptyset$ at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4–8, line 5 extracts a vertex u from $Q = V - S$ and line 6 adds it to set S, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 7–8 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if we can improve the shortest path to v found so far by going through u. Observe that the algorithm never inserts vertices into Q after line 3 and that each vertex is extracted from Q

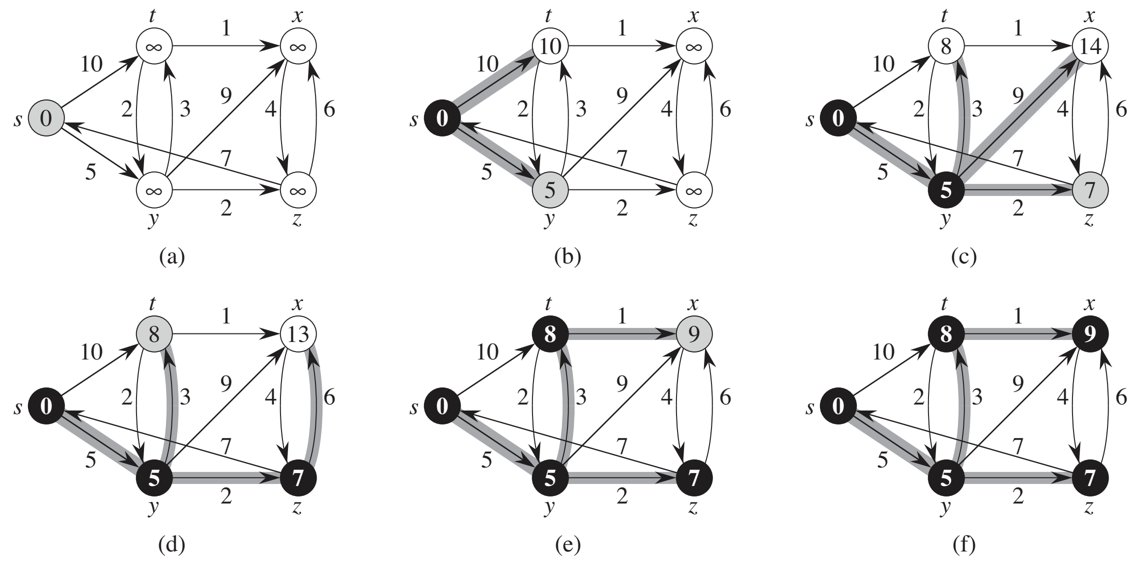

Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

and added to S exactly once, so that the **while** loop of lines 4–8 iterates exactly $|V|$ times.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy. Chapter 16 explains greedy strategies in detail, but you need not have read that chapter to understand Dijkstra's algorithm. Greedy strategies do not always yield optimal results in general, but as the following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths. The key is to show that each time it adds a vertex u to set S, we have $u.d = \delta(s, u)$.

Theorem 24.6 (Correctness of Dijkstra's algorithm)

Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function w and source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

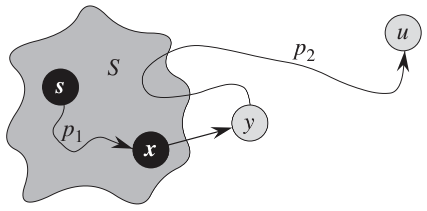

Figure 24.7 The proof of Theorem 24.6. Set S is nonempty just before vertex u is added to it. We decompose a shortest path p from source s to vertex u into $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, where y is the first vertex on the path that is not in S and $x \in S$ immediately precedes y. Vertices x and y are distinct, but we may have $s = x$ or $y = u$. Path p_2 may or may not reenter set S.

Proof We use the following loop invariant:

> At the start of each iteration of the **while** loop of lines 4–8, $v.d = \delta(s, v)$ for each vertex $v \in S$.

It suffices to show for each vertex $u \in V$, we have $u.d = \delta(s, u)$ at the time when u is added to set S. Once we show that $u.d = \delta(s, u)$, we rely on the upper-bound property to show that the equality holds at all times thereafter.

Initialization: Initially, $S = \emptyset$, and so the invariant is trivially true.

Maintenance: We wish to show that in each iteration, $u.d = \delta(s, u)$ for the vertex added to set S. For the purpose of contradiction, let u be the first vertex for which $u.d \neq \delta(s, u)$ when it is added to set S. We shall focus our attention on the situation at the beginning of the iteration of the **while** loop in which u is added to S and derive the contradiction that $u.d = \delta(s, u)$ at that time by examining a shortest path from s to u. We must have $u \neq s$ because s is the first vertex added to set S and $s.d = \delta(s, s) = 0$ at that time. Because $u \neq s$, we also have that $S \neq \emptyset$ just before u is added to S. There must be some path from s to u, for otherwise $u.d = \delta(s, u) = \infty$ by the no-path property, which would violate our assumption that $u.d \neq \delta(s, u)$. Because there is at least one path, there is a shortest path p from s to u. Prior to adding u to S, path p connects a vertex in S, namely s, to a vertex in $V - S$, namely u. Let us consider the first vertex y along p such that $y \in V - S$, and let $x \in S$ be y's predecessor along p. Thus, as Figure 24.7 illustrates, we can decompose path p into $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$. (Either of paths p_1 or p_2 may have no edges.)

We claim that $y.d = \delta(s, y)$ when u is added to S. To prove this claim, observe that $x \in S$. Then, because we chose u as the first vertex for which $u.d \neq \delta(s, u)$ when it is added to S, we had $x.d = \delta(s, x)$ when x was added to S. Edge (x, y) was relaxed at that time, and the claim follows from the convergence property.

We can now obtain a contradiction to prove that $u.d = \delta(s, u)$. Because y appears before u on a shortest path from s to u and all edge weights are nonnegative (notably those on path p_2), we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq u.d \qquad \text{(by the upper-bound property)} \ . \end{aligned} \tag{24.2}$$

But because both vertices u and y were in $V - S$ when u was chosen in line 5, we have $u.d \leq y.d$. Thus, the two inequalities in (24.2) are in fact equalities, giving

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

Consequently, $u.d = \delta(s, u)$, which contradicts our choice of u. We conclude that $u.d = \delta(s, u)$ when u is added to S, and that this equality is maintained at all times thereafter.

Termination: At termination, $Q = \emptyset$ which, along with our earlier invariant that $Q = V - S$, implies that $S = V$. Thus, $u.d = \delta(s, u)$ for all vertices $u \in V$. ■

Corollary 24.7
If we run Dijkstra's algorithm on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source s, then at termination, the predecessor subgraph G_π is a shortest-paths tree rooted at s.

Proof Immediate from Theorem 24.6 and the predecessor-subgraph property. ■

Analysis

How fast is Dijkstra's algorithm? It maintains the min-priority queue Q by calling three priority-queue operations: INSERT (implicit in line 3), EXTRACT-MIN (line 5), and DECREASE-KEY (implicit in RELAX, which is called in line 8). The algorithm calls both INSERT and EXTRACT-MIN once per vertex. Because each vertex $u \in V$ is added to set S exactly once, each edge in the adjacency list *Adj*$[u]$ is examined in the **for** loop of lines 7–8 exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is $|E|$, this **for** loop iterates a total of $|E|$ times, and thus the algorithm calls DECREASE-KEY at most $|E|$ times overall. (Observe once again that we are using aggregate analysis.)

The running time of Dijkstra's algorithm depends on how we implement the min-priority queue. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to $|V|$. We simply store $v.d$ in the vth entry of an array. Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT-MIN operation takes $O(V)$ time (since we have to search through the entire array), for a total time of $O(V^2 + E) = O(V^2)$.

If the graph is sufficiently sparse—in particular, $E = o(V^2/\lg V)$—we can improve the algorithm by implementing the min-priority queue with a binary min-heap. (As discussed in Section 6.5, the implementation should make sure that vertices and corresponding heap elements maintain handles to each other.) Each EXTRACT-MIN operation then takes time $O(\lg V)$. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. Each DECREASE-KEY operation takes time $O(\lg V)$, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E)\lg V)$, which is $O(E \lg V)$ if all vertices are reachable from the source. This running time improves upon the straightforward $O(V^2)$-time implementation if $E = o(V^2/\lg V)$.

We can in fact achieve a running time of $O(V \lg V + E)$ by implementing the min-priority queue with a Fibonacci heap (see Chapter 19). The amortized cost of each of the $|V|$ EXTRACT-MIN operations is $O(\lg V)$, and each DECREASE-KEY call, of which there are at most $|E|$, takes only $O(1)$ amortized time. Historically, the development of Fibonacci heaps was motivated by the observation that Dijkstra's algorithm typically makes many more DECREASE-KEY calls than EXTRACT-MIN calls, so that any method of reducing the amortized time of each DECREASE-KEY operation to $o(\lg V)$ without increasing the amortized time of EXTRACT-MIN would yield an asymptotically faster implementation than with binary heaps.

Dijkstra's algorithm resembles both breadth-first search (see Section 22.2) and Prim's algorithm for computing minimum spanning trees (see Section 23.2). It is like breadth-first search in that set S corresponds to the set of black vertices in a breadth-first search; just as vertices in S have their final shortest-path weights, so do black vertices in a breadth-first search have their correct breadth-first distances. Dijkstra's algorithm is like Prim's algorithm in that both algorithms use a min-priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in Prim's algorithm), add this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

Exercises

24.3-1

Run Dijkstra's algorithm on the directed graph of Figure 24.2, first using vertex s as the source and then using vertex z as the source. In the style of Figure 24.6, show the d and π values and the vertices in set S after each iteration of the **while** loop.

24.3-2

Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why doesn't the proof of Theorem 24.6 go through when negative-weight edges are allowed?

24.3-3

Suppose we change line 4 of Dijkstra's algorithm to the following.

```
4  while $|Q| > 1$
```

This change causes the **while** loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm correct?

24.3-4

Professor Gaedel has written a program that he claims implements Dijkstra's algorithm. The program produces $v.d$ and $v.\pi$ for each vertex $v \in V$. Give an $O(V + E)$-time algorithm to check the output of the professor's program. It should determine whether the d and π attributes match those of some shortest-paths tree. You may assume that all edge weights are nonnegative.

24.3-5

Professor Newman thinks that he has worked out a simpler proof of correctness for Dijkstra's algorithm. He claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path, and therefore the path-relaxation property applies to every vertex reachable from the source. Show that the professor is mistaken by constructing a directed graph for which Dijkstra's algorithm could relax the edges of a shortest path out of order.

24.3-6

We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \le r(u, v) \le 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

24.3-7

Let $G = (V, E)$ be a weighted, directed graph with positive weight function $w : E \to \{1, 2, \ldots, W\}$ for some positive integer W, and assume that no two vertices have the same shortest-path weights from source vertex s. Now suppose that we define an unweighted, directed graph $G' = (V \cup V', E')$ by replacing each edge $(u, v) \in E$ with $w(u, v)$ unit-weight edges in series. How many vertices does G' have? Now suppose that we run a breadth-first search on G'. Show that