CHAPTER 25: SINGLE-SOURCE SHORTEST PATHS

Previous Chapter Return to Table of Contents Next Chapter

A motorist wishes to find the shortest possible route from Chicago to Boston. Given a road map of the United States on which the distance between each pair of adjacent intersections is marked, how can we determine this shortest route?

One possible way is to enumerate all the routes from Chicago to Boston, add up the distances on each route, and select the shortest. It is easy to see, however, that even if we disallow routes that contain cycles, there are millions of possibilities, most of which are simply not worth considering. For example, a route from Chicago to Houston to Boston is obviously a poor choice, because Houston is about a thousand miles out of the way.

In this chapter and in Chapter 26, we show how to solve such problems efficiently. In a ***shortest-paths problem***, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow \mathbf{R}$ mapping edges to real-valued weights. The ***weight*** of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

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

We define the ***shortest-path weight*** from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} \\ \infty \end{cases}$$ if there is a path from u to v, otherwise.

A ***shortest path*** from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$.

In the Chicago-to-Boston example, we can model the road map as a graph: vertices represent intersections, edges represent road segments between intersections, and edge weights represent road distances. Our goal is to find a shortest path from a given intersection in Chicago (say, Clark St. and Addison Ave.) to a given intersection in Boston (say, Brookline Ave. and Yawkey Way).

Edge weights can be interpreted as metrics other than distances. They are often used to represent time, cost, penalties, lossage, or any other quantity that accumulates linearly along a path and that one wishes to minimize.

The breadth-first-search algorithm from Section 23.2 is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight. Because many of the concepts from breadth-first search arise in the study of shortest paths in weighted graphs, the reader is encouraged to review Section 23.2 before proceeding.

Variants

In this chapter, we shall focus on the ***single-source shortest-paths problem*:** given a graph $G = (V, E)$, we want to find a shortest path from a given ***source*** vertex $s \in V$ to every vertex $v \in V$. Many other problems can be solved by the algorithm for the single-source problem, including the following variants.

Single-destination shortest-paths problem: Find a shortest path to a given ***destination*** vertex t from every vertex v. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also. Moreover, no algorithms for this problem are known that run asymptotically faster than the best single-source algorithms in the worst case.

All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v. This problem can be solved by running a single-source algorithm once from each vertex; but it can usually be solved faster, and its structure is of interest in its own right. Chapter 26 addresses the all-pairs problem in detail.

Negative-weight edges

In some instances of the single-source shortest-paths problem, there may be edges whose weights are negative. If the graph $G = (V, E)$ contains no negative-weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value. If there is a negative-weight cycle reachable from s, however, shortest-path weights are not well defined. No path from s to a vertex on the cycle can be a shortest path--a lesser-weight path can always be found that follows the proposed "shortest" path and then traverses the negative-weight cycle. If there is a negative-weight cycle on some path from s to v, we define $\delta(s,v) = -\infty$.

Figure 25.1 illustrates the effect of negative weights on shortest-path weights. Because there is only one path from s to a (the path $\langle s, a\rangle$), $\delta(s, a = w(s, a) = 3$. Similarly, there is only one path from s to b, and so $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = - 1$. There are infinitely many paths from s to c: $\langle s, c\rangle$, $\langle s, c, d, c\rangle$, $\langle s, c, d, c, d, c\rangle$, and so on. Because the cycle $\langle c, d, c\rangle$ has weight $6 + (-3) = 3 > 0$, the shortest path from s to c is $\langle s, c\rangle$, with weight $\delta(s, c) = 5$. Similarly, the shortest path from s to d is $\langle s, c, d\rangle$, with weight $\delta(s, d) = w(s, c) + w(c, d) = 11$. Analogously, there are infinitely many paths from s to e: $\langle s, e\rangle$, $\langle s, e, f, e\rangle$, $\langle s, e, f, e, f, e\rangle$, and so on. Since the cycle $\langle e, f, e\rangle$ has weight $3 + (-6) = -3 < 0$, however, there is no shortest path from s to e. By traversing the negative-weight cycle $\langle e, f, e\rangle$ arbitrarily many times, we can find paths from s to e with arbitrarily large negative weights, and so $\delta(s, e) = -\infty$. Similarly, $\delta(s, f) -\infty$. Because g is reachable from f, we can also find paths with arbitrarily large negative weights from s to g, and $\delta(s, g) = -\infty$. Vertices h, i, *and* j also form a negative-weight cycle. They are not reachable from s, however, and so $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

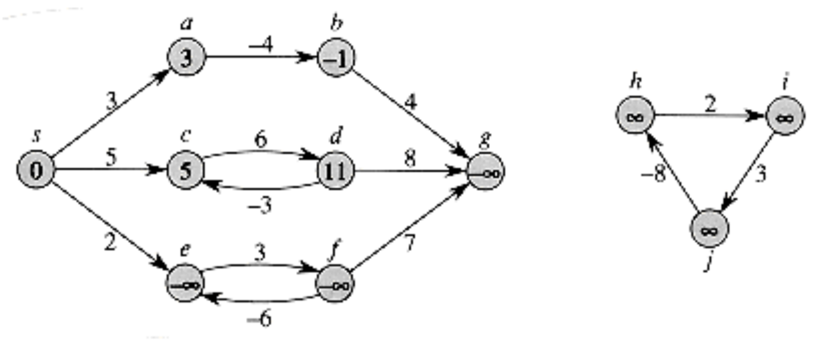

Figure 25.1 Negative edge weights in a directed graph. Shown within each vertex is its shortest-path weight from source s. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of -∞. Because vertex g is reachable from a vertex whose shortest-path weight is -∞, it, too, has a shortest-path weight of -∞. Vertices such as h, i, and j are not reachable from s, and so their shortest-path weights are ∞, even though they lie on a negative-weight cycle.

Some shortest-paths algorithms, such as Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative, as in the road-map example. Others, such as the Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

Representing shortest paths

We often wish to compute not only shortest-path weights, but the vertices on the shortest paths as well. The representation we use for shortest paths is similar to the one we used for breadth-first trees in Section 23.2. Given a graph $G = (V, E)$, we maintain for each vertex $v \in V$ a ***predecessor*** $\Pi[v]$ that is either another vertex or NIL. The shortest-paths algorithms in this chapter set the Π attributes so that the chain of predecessors originating at a vertex v runs backwards along a shortest path from s to v. Thus, given a vertex v for which $\Pi[v] \neq$ NIL, the procedure PRINT-PATH (G, s, v) from Section 23.2 can be used to print a shortest path from s to v.

During the execution of a shortest-paths algorithm, however, the Π values need not indicate shortest paths. As in breadth-first search, we shall be interested in the ***predecessor subgraph*** $G\Pi = (V_{\Pi}, E_{\pi})$ induced by the Π values. Here again, we define the vertex set $V\Pi$, to be the set of vertices of G with non-NIL predecessors, plus the source s:

$$V\Pi = \{v \in V : \Pi[v] \neq \text{NIL}\} \cup \{s\} .$$

The directed edge set $E\Pi$ is the set of edges induced by the Π values for vertices in $V\Pi$:

$$E\Pi = \{(\Pi[v], v) \in E : v \in V\Pi - \{\Sigma\}\} .$$

We shall prove that the Π values produced by the algorithms in this chapter have the property that at termination $G\Pi$ is a "shortest-paths tree"--informally, a rooted tree containing a shortest path from a source s to every vertex that is reachable from s. A shortest-paths tree is like the breadth-first tree from Section 23.2, but it contains shortest paths from the source defined in terms of edge weights instead of numbers of edges. To be precise, let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$, and assume that G contains no negative-weight cycles reachable from the source vertex $s \in V$, so that shortest paths are well defined. A ***shortest-paths tree*** rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G,

2. G' forms a rooted tree with root s, and

3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Shortest paths are not necessarily unique, and neither are shortest-paths trees. For example, Figure 25.2 shows a weighted, directed graph and two shortest-paths trees with the same root.

Chapter outline

The single-source shortest-paths algorithms in this chapter are all based on a technique known as relaxation. Section 25.1 begins by proving some important properties of shortest paths in general and then proves some important facts about relaxation-based algorithms. Dijkstra's algorithm, which solves the single-source shortest-paths problem when all edges have nonnegative weight, is given in Section 25.2. Section 25.3 presents the Bellman-Ford algorithm, which is used in the more general case in which edges can have negative weight. If the graph contains a negative-weight cycle reachable from the source, the Bellman-Ford algorithm detects its presence. Section 25.4 gives a linear-time algorithm for computing shortest paths from a single source in directed acyclic graphs. Finally, Section 25.5 shows how the Bellman-Ford algorithm can be used to solve a special case of "linear programming."

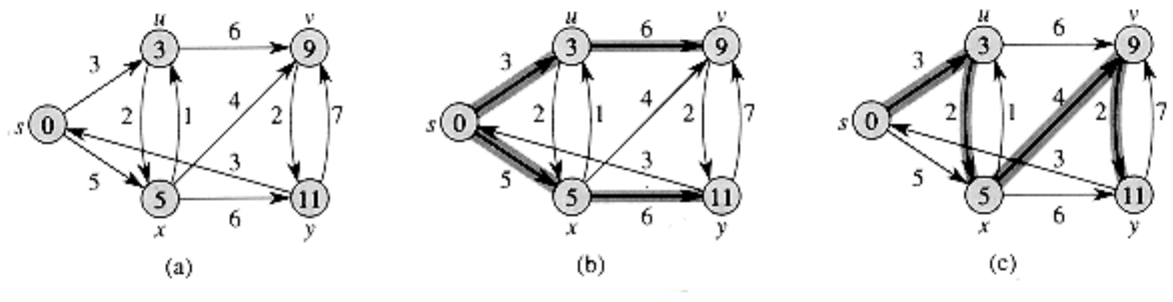

Figure 25.2 (a) A weighted, directed graph with shortest-path weights from source s. (b) The shaded edges form a shortest-paths tree rooted at the source s. (c) Another shortest-paths tree with the same root.

Our analysis will require some conventions for doing arithmetic with infinities. We shall assume that for any real number $a \neq -\infty$, we have $a + \infty = \infty + a = \infty$. Also, to make our proofs hold in the presence of negative-weight cycles, we shall assume that for any real number $a \neq \infty$, we have $a + (-\infty) = (-\infty) + a = -\infty$.

25.1 Shortest paths and relaxation

To understand single-source shortest-paths algorithms, it is helpful to understand the techniques that they use and the properties of shortest paths that they exploit. The main technique used by the algorithms in this chapter is relaxation, a method that repeatedly decreases an upper bound on the actual shortest-path weight of each vertex until the upper bound equals the shortest-path weight. In this section, we shall see how relaxation works and formally prove several properties it maintains.

On a first reading of this section, you may wish to omit proofs of theorems--reading only their statements--and then proceed immediately to the algorithms in Sections 25.2 and 25.3. Pay particular attention, however, to Lemma 25.7, which is a key to understanding the shortest-paths algorithms in this chapter. On a first reading, you may also wish to ignore completely the lemmas concerning predecessor subgraphs and shortest-paths trees (Lemmas 25.8 and 25.9), concentrating instead on the earlier lemmas, which pertain to shortest-path weights.

Optimal substructure of a shortest path

Shortest-paths algorithms typically exploit the property that a shortest path between two vertices contains other shortest paths within it. This optimal-substructure property is a hallmark of the applicability of both dynamic programming (Chapter 16) and the greedy method (Chapter 17). In fact, Dijkstra's algorithm is a greedy algorithm, and the Floyd-Warshall algorithm, which finds shortest paths between all pairs of vertices (see Chapter 26), is a dynamic-programming algorithm. The following lemma and its corollary state the optimal-substructure property of shortest paths more precisely.

Lemma 25.1

Given a weighted, directed graph $G = (V, E)$ with weight function $w: E \rightarrow \mathbf{R}$, *let* $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from vertex v_1, to vertex v_k and, for any i and j such that $1 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_i+1, \ldots, v_i \rangle$ be the subpath of p from vertex v_i, to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof If we decompose path p into $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$, then $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ is a path from v_1 to v_k whose weight $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the premise that p is a shortest path from v_1 to v_k..

In studying breadth-first search (Section 23.2), we proved as Lemma 23.1 a simple property of shortest distances in unweighted graphs. The following corollary to Lemma 25.1 generalizes the property to weighted graphs.

Corollary 25.2

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$. Suppose that a shortest path p from a source s to a vertex v can be decomposed into $s \overset{p'}{\leadsto} u \rightarrow v$ for some vertex u and path p'. Then, the weight of a shortest path from s to v is $\delta(s, v) = \delta(s, u) + w(u, v)$.

Proof By Lemma 25.1, subpath p' is a shortest path from source s to vertex u. Thus,

$$\begin{aligned}\delta(s, v) &= w(p) \\ &= w(p') + w(u, v) \\ &= \delta(s, u) + w(u, v).\end{aligned}$$

The next lemma gives a simple but useful property of shortest-path weights.

Lemma 25.3

Let $G = (V, E)$ be a weighted, directed graph $G = (V, E)$ with weight function $w: E \rightarrow \mathbf{R}$ and source vertex s. 0Then, for all edges $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Proof A shortest path p from source s to vertex v has no more weight than any other path from s to v. Specifically, path p has no more weight than the particular path that takes a shortest path from source s to vertex u and then takes edge (u, v).

Relaxation

The algorithms in this chapter use the technique of ***relaxation***. For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source s to v. We call $d[v]$ a ***shortest-path estimate***. We initialize the shortest-path estimates and predecessors by the following procedure.

```
INITIALIZE-SINGLE-SOURCE(G,s)
1  for each vertex $v \in V[G]$
2       do $d[v] \leftarrow \infty$
3          $\pi[v] \leftarrow$ NIL
4  $d[s] \leftarrow 0$
```

After initialization, $\pi[v]$ = NIL for all $v \in V$, $d[v] = 0$ for $v = s$, and $d[v] = \infty$ for $v \in V - \{s\}$.

The process of ***relaxing***[1] an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $d[v]$ and $\pi[v]$. A relaxation step may decrease the value of the shortest-path estimate $d[v]$ and update v's predecessor field $\pi[v]$. The following code performs a relaxation step on edge (u, v).

[1]It may seem strange that the term "relaxation" is used for an operation that tightens an upper bound. The use of the term is historical. The outcome of a relaxation step can be viewed as a relaxation of the constraint $d[v] \le d[u] + w(u, v)$, which, by Lemma 25.3, must be satisfied if $d[u] = \delta(s, u)$ *and* $d[v] = \delta(s, v)$. That is, if $d[v] \le d[u] + w(u, v)$, there is no

"pressure" to satisfy this constraint, so the constraint is "relaxed."

```
RELAX(u, v, w)

1 if d[v] > d[u] + w(u,v)

2    then d[v] ← d[u] + w(u,v)

3         Π[v] ← u
```

Figure 25.3 shows two examples of relaxing an edge, one in which a shortest-path estimate decreases and one in which no estimate changes.

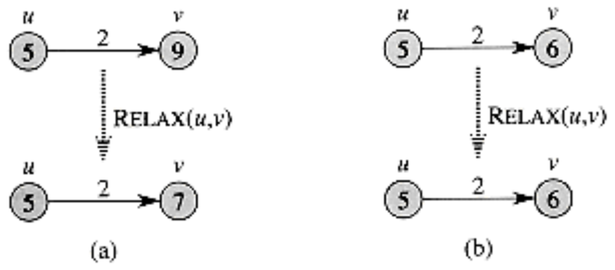

Figure 25.3 Relaxation of an edge (u, v). The shortest-path estimate of each vertex is shown within the vertex. (a) Because d[v] > d[u] + w(u, v) prior to relaxation, the value of d[v] decreases. (b) Here, d[v] ≤ d[u] + w(u, v) before the relaxation step, so d[v] is unchanged by relaxation.

Each algorithm in this chapter calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges. Moreover, relaxation is the only means by which shortest-path estimates and predecessors change. The algorithms in this chapter differ in how many times they relax each edge and the order in which they relax edges. In Dijkstra's algorithm and the shortest-paths algorithm for directed acyclic graphs, each edge is relaxed exactly once. In the Bellman-Ford algorithm, each edge is relaxed several times.

Properties of relaxation

The correctness of the algorithms in this chapter depends on important properties of relaxation that are summarized in the next few lemmas. Most of the lemmas describe the outcome of executing a sequence of relaxation steps on the edges of a weighted, directed graph that has been initialized by INITIALIZE-SINGLE-SOURCE. Except for Lemma 25.9, these lemmas apply to *any* sequence of relaxation steps, not just those that produce shortest-path values.

Lemma 25.4

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$, and let $(u, v) \in E$. Then, immediately after relaxing edge (u, v) by executing RELAX(u, v, w), we have $d[v] \leq d[u] + w(u, v)$.

Proof If, just prior to relaxing edge (u, v), we have $d[v] > d[u] + w(u, v)$, then $d[v] = d[u] + w(u, v)$ afterward. If, instead, $d[v] \leq d[u] + w(u, v)$ just before the relaxation, the neither

$d[u]$ nor $d[v]$ changes, and so $d[v] \le d[u] + w(u, v)$ afterward.

Lemma 25.5

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$. Let $s \in V$ be the source vertex, and let the graph be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Then, $d[v] \ge \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps on the edges of G. Moreover, once $d[v]$ achieves its lower bound $\delta(s, v)$, it never changes.

Proof The invariant $d[v] \ge \delta(s, v)$ is certainly true after initialization, since $d[s] = 0 \ge \delta(s, s)$ (note that $\delta(s, s)$ is $-\infty$ if s is on a negative-weight cycle and 0 otherwise) and $d[v] = \infty$ implies $d[v] \ge \delta(s, v)$ for all $v \in V - \{s\}$. We shall use proof by contradiction to show that the invariant is maintained over any sequence of relaxation steps. Let v be the first vertex for which a relaxation step of an edge (u, v) causes $d[v] < \delta(s, v)$. Then, just after relaxing edge (u, v), we have

$$\begin{aligned} d[u] + w(u, v) &= d[v] \\ &< \delta(s,v) \\ &\le \delta(s,u) + w(u, v) \quad \text{(by Lemma 25.3)}, \end{aligned}$$

which implies that $d[u] < \delta(s, u)$. But because relaxing edge (u, v) does not change $d[u]$, this inequality must have been true just before we relaxed the edge, which contradicts the choice of v as the first vertex for which $d[v] < \delta(s, v)$. We conclude that the invariant $d[v] \ge \delta(s, v)$ is maintained for all $v \in V$.

To see that the value of $d[v]$ never changes once $d[v] = \delta(s, v)$, note that having achieved its lower bound, $d[v]$ cannot decrease because we have just shown that $d[v] \ge \delta(s, v)$, and it cannot increase because relaxation steps do not increase d values.

Corollary 25.6

Suppose that in a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbf{R}$, no path connects a source vertex $s \in V$ to a given vertex $v \in V$. Then, after the graph is initialized by INITIALIZE-SINGLE-SOURCE(G, s), we have $d[v] = \delta(s, v)$, and this equality is maintained as an invariant over any sequence of relaxation steps on the edges of G.

Proof By Lemma 25.5, we always have $\infty = \delta(s, v) \le d[v]$; thus, so $d[v] = \infty = \delta(s, v)$.

The following lemma is crucial to proving the correctness of the shortest-paths algorithms that appear later in this chapter. It gives sufficient conditions for relaxation to cause a shortest-path estimate to converge to a shortest-parth weight.

Lemma 25.7

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$, let $s \in V$ be a source vertex, and let $s \leadsto u \rightarrow v$ be a shortest path in G for some vertices $u, v \in V$. Suppose that G is initialized by INITIALIZE-SINGLE-SOURCE(G, s) and then a sequence of relaxation steps that includes the call RELAX(u, v, w) is executed on the edges of G. If $d[u] = \delta$

(*s, u*) at any time prior to the call, then $d[v] = \delta(s, v)$ at all times after the call.

Proof By Lemma 25.5, if $d[u] = \delta(s, u)$ at some point prior to relaxing edge (*u, v*), then this equality holds thereafter. In particular, after relaxing edge (*u, v)* we have

```
d[v]  ≤  d[u] + w(u,v)    (by Lemma 25.4)

=  δ(s,u) + w(u,v)

=  δ(s,v)           (by Corollary 25.2).
```

By Lemma 25.5, $\delta(s, v)$ bounds $d[v]$ from below, from which we conclude that $d[v] = \delta(s, v)$, and this equality is maintained thereafter.

Shortest-paths trees

So far, we have shown that relaxation causes the shortest-path estimates to descend monotonically toward the actual shortest-path weights. We would also like to show that once a sequence of relaxations has computed the actual shortest-path weights, the predecessor subgraph G_π induced by the resulting π values is a shortest-paths ree for *G* . e start with the following lemma, which shows that the predecessor subgraph always forms a rooted tree whose root is the source.

Lemma 25.8

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$ and source vertex $s \in V$, and assume that *G* contains no negative-weight cycles that are reachable from *s*. Then, after the graph is initialized by INITIALIZE-SINGLE-SOURCE(*G, s*), the predecessor subgraph G_π forms a rooted tree with root *s*, and any sequence of relaxation steps on edges of *G* maintains this property as an invariant.

Proof Initially, the only vertex in G_π is the source vertex, and the lemma is trivially true. Consider a predecessor subgraph G_π that arises after a sequence of relaxation steps. We shall first prove that G_π is acyclic. Suppose for the sake of contradiction that some relaxation step creates a cycle in the graph G_π. Let the cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_k = v_0$. Then, $\pi[v_i] = v_{i-1}$ for $i = 1, 2, \ldots, k$ and, without loss of generality, we can assume that it was the relaxation of edge (v_{k-1}, v_k) that created the cycle in G_π.

We claim that all vertices on cycle *c* are reachable from the source *s*. Why? Each vertex on *c* has a non-NIL predecessor, and so each vertex on *c* was assigned a finite shortest-path estimate when it was assigned its non-NIL π value. By Lemma 25.5, each vertex on cycle *c* has a finite shortest-path weight, which implies that it is reachable from *s*.

We shall examine the shortest-path estimates on *c* just prior to the call RELAX (v_{k-1}, v_k, w) and show that *c* is a negative-weight cycle, thereby contradicting the assumption that *G* contains no negative-weight cycles that are reachable from the source. Just before the call, we have $\pi[v_i] = v_{i-1}$ for $i = 1, 2, \ldots, k - 1$. Thus, for $i = 1, 2, \ldots, k - 1$, the last update to $d[v_i]$ was by the assignment $d[v_i] \leftarrow d[v_{i-1}] + w(v_i, v_{i-1})$. If $d[v_{i-1}]$ changed since then, it decreased. Therefore, just before the call RELAX (v_{k-1}, v_k, w), we have

$d[v_i] \geq d[v_{i-1}]+w(v_{i-1},v_i)$ for all $i = 1, 2,\ldots,k - 1$.

(25.1)

Because $\pi[v_k]$ is changed by the call, immediately beforehand we also have the strict inequality

$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k)$.

Summing this strict inequality with the k - 1 inequalities (25.1), we obtain the sum of the shortest-path estimates around cycle c:

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

But

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] ,$$

since each vertex in the cycle c appears exactly once in each summation. This implies

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

Thus, the sum of weights around the cycle c is negative, thereby providing the desired contradiction.

We have now proved that G_π is a directed, acyclic graph. To show that it forms a rooted tree with root s, it sufiices (see Exercise 5.5-3) to prove that for each vertex $v \in V_\pi$, there is a unique path from s to v in G_π.

We first must show that a path from s exists for each vertex in V_π. The vertices in V_π are those with non-NIL π values, plus s. The idea here is to prove by induction that a path exists from s to all vertices in V_π. The details are left as Exercise 25.1-6.

To complete the proof of the lemma, we must now show that for any vertex $v \in V_\pi$, there is at most one path from s to v in the graph G_π. Suppose otherwise. That is, suppose that there are two simple paths from s to some vertex v: p_1, which can be decomposed into $s \rightsquigarrow u \rightsquigarrow x \to z \rightsquigarrow v,$ and p_2, which can be decomposed into $s \rightsquigarrow u \rightsquigarrow y \to z \rightsquigarrow v$, where $x \neq y$. (See Figure 25.4.) But then, $\pi[z] = x$ and $\pi[z] = y$, which implies the contradiction that $x = y$. We conclude that there exists a unique simple path in G_π from s to v, and thus G_π forms a rooted tree with root s.

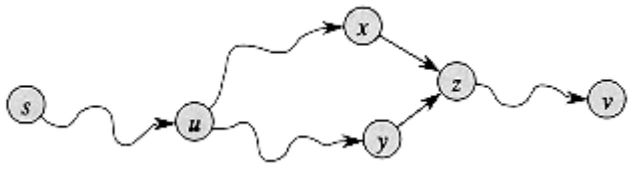

Figure 25.4 Showing that a path in G_π from source s to vertex v is unique. If there are two paths $p_1\ (s \leadsto u \leadsto x \to z \leadsto v)$ and $p_2\ (s \leadsto u \leadsto y \to z \leadsto v)$, where x ≠ y, then π[z] = x and π[z] = y, a contradiction.

We can now show that if, after we have performed a sequence of relaxation steps, all vertices have been assigned their true shortest-path weights, then the predecessor subgraph G_π is a shortest-paths tree.

Lemma 25.9

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$ and source vertex $s \in V$, and assume that G contains no negative-weight cycles that are reachable from s. Let us call INITIALIZE-SINGLE-SOURCE(G, s) and then execute any sequence of relaxation steps on edges of G that produces $d[v] = \delta(s, v)$ for all $v \in V$. Then, the predecessor subgraph G_π is a shortest-paths tree rooted at s.

Proof We must prove that the three properties of shortest-paths trees hold for G_π. To show the first property, we must show that V_π is the set of vertices reachable from s. By definition, a shortest-path weight $\delta(s, v)$ is finite if and only if v is reachable from s, and thus the vertices that are reachable from s are exactly those with finite d values. But a vertex $v \in V - \{s\}$ has been assigned a finite value for $d[v]$ if and only if $\pi[v] \neq$ NIL. Thus, the vertices in V_π are exactly those reachable from s.

The second property follows directly from Lemma 25.8.

It remains, therefore, to prove the last property of shortest-paths trees: for all $v \in V_\pi$, the unique simple path $s \overset{p}{\leadsto} v$ in G_π is a shortest path from s to v in G. Let $p = \langle v_0, v_1, \ldots, v_k\rangle$, where $v_0 = s$ and $v_k = v$. For $i = 1, 2, \ldots, k$, we have both $d[v_i] = \delta(s, v_i)$ and $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$, from which we conclude $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. Summing the weights along path p yields

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

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

$$= \delta(s, v_k) - \delta(s, v_0)$$

$$= \delta(s, v_k).$$

The third line comes from the telescoping sum on the second line, and the fourth line follows from $\delta(s, v_0) = \delta(s, s) = 0$. Thus, $w(p) \le \delta(s, v_k)$. Since $\delta(s, v_k)$ is a lower bound on the weight of any path from s to v_k, we conclude that $w(p) = \delta(s, v_k)$, and thus p is a shortest path from s to $v = v_k$.

Exercises

25.1-1

Give two shortest-paths trees for the directed graph of Figure 25.2 other than the two shown.

25.1-2

Give an example of a weighted, directed graph $G = (V, E)$ with weight function $w\colon E \rightarrow \mathbf{R}$ and source s such that G satisfies the following property: For every edge $(u, v) \in E$, there is a shortest-paths tree rooted at s that contains (u, v) and another shortest-paths tree rooted at s that does not contain (u, v).

25.1-3

Embellish the proof of Lemma 25.3 to handle cases in which shortest-path weights are ∞ or $-\infty$.

25.1-4

Let $G = (V, E)$ be a weighted, directed graph with source vertex s, and let G be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Prove that if a sequence of relaxation steps sets $\pi[s]$ to a non-NIL value, then G contains a negative-weight cycle.

25.1-5

Let $G = (V, E)$ be a weighted, directed graph with no negative-weight edges. Let $s \in V$ be the source vertex, and let us define $\pi[v]$ as usual: $\pi[v]$ is the predecessor of v on some shortest path to v from source s if $v \in V - \{s\}$ is reachable from s, and NIL otherwise. Give an example of such a graph G and an assignment of π values that produces a cycle in G_π. (By Lemma 25.8, such an assignment cannot be produced by a sequence of relaxation steps.)

25.1-6

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$ and no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Prove that for every vertex $v \in V_\pi$, there exists a path from s to v in G_π and that this property is maintained as an invariant over any sequence of relaxations.

25.1-7

Let $G = (V, E)$ be a weighted, directed graph that contains no negative-weight cycles. Let s

$\in V$ be the source vertex, and let G be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Prove that there is a sequence of $|V| - 1$ relaxation steps that produces $d[v] = \delta(s, v)$ for all $v \in V$.

25.1-8

Let G be an arbitrary weighted, directed graph with a negative-weight cycle reachable from the source vertex s. Show that an infinite sequence of relaxations of the edges of G can always be constructed such that every relaxation causes a shortest-path estimate to change.

25.2 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 $(u, v) \in E$.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. That is, for all vertices $v \in S$, we have $d[v] = \delta(s, v)$. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, inserts u into S, and relaxes all edges leaving u. In the following implementation, we maintain a priority queue Q that contains all the vertices in $V - S$, keyed by their d values. The implementation assumes that graph G is represented by adjacency lists.

```
DIJKSTRA(G,w,s)

1  INITIALIZE-SINGLE-SOURCE (G,s)

2  $S \leftarrow \emptyset$

3  $Q \leftarrow V[G]$

4  while $Q \neq \emptyset$

5       do $u \leftarrow$ EXTRACT-MIN(Q)

6          $S \leftarrow S \cup \{u\}$

7          for each vertex $v \in Adj[u]$

8              do RELAX (u,v,w)
```

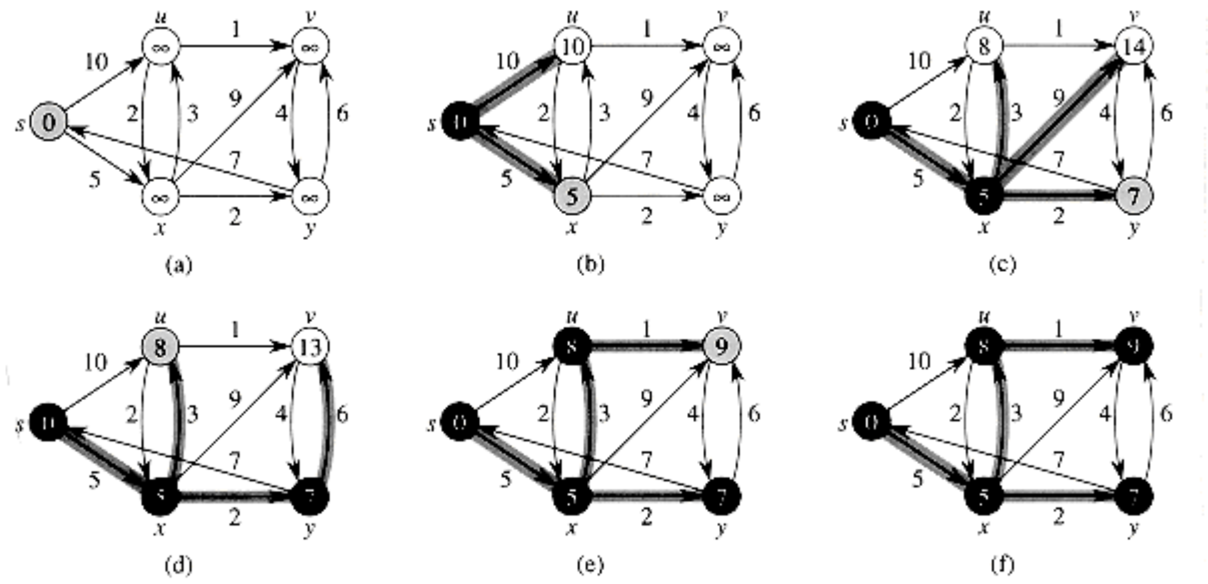

Figure 25.5 The execution of Dijkstra's algorithm. The source is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values: if edge (u,v) is shaded, then π[v] = u. Black vertices are in the set S, and white vertices are in the priority 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 and π values shown in part (f) are the final values.

Dijkstra's algorithm relaxes edges as shown in Figure 25.5. Line 1 performs the usual initialization of *d* and π values, and line 2 initializes the set *S* to the empty set. Line 3 then initializes the priority queue *Q* to contain all the vertices in $V - S = V - \emptyset = V$. Each time through the **while** loop of lines 4-8, a vertex *u* is extracted from *Q* = *V* - *S* and inserted into set *S*. (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 *d*[*v*] and the predecessor π[*v*] if the shortest path to *v* can be improved by going through *u*. Observe that vertices are never inserted into *Q* after line 3 and that each vertex is extracted from *Q* and inserted into *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 insert into set *S*, we say that it uses a greedy strategy. Greedy strategies are presented in detail in Chapter 17, 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 a vertex *u* is inserted into set *S*, we have *d*[*u*] = δ(s, u).

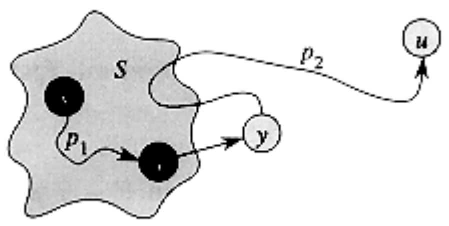

Figure 25.6 The proof of Theorem 25.10. Set S is nonempty just before vertex u is inserted into it. A shortest path p from source s to vertex u can be decomposed into $s \overset{p_1}{\leadsto} x \rightarrow y \overset{p_2}{\leadsto} u$ where y is the first vertex on the path that is not in V - S and 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.**

Theorem 25.10

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, *d*[*u*] = δ(s, u*) for all vertices* u $\in$ V*.*

Proof We shall show that for each vertex *u* $\in$ *V,* we have *d*[*u*] = δ(*s*, *u*) at the time when *u* is inserted into set *S* and that this equality is maintained thereafter.

For the purpose of contradiction, let *u* be the first vertex for which *d*[*u*] $\neq$ δ(*s*, *u*) when it is inserted into set *S*. We shall focus our attention on the situation at the beginning of the iteration of the **while** loop in which *u* is inserted into *S* and derive the contradiction that *d*[*u*] = δ(*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 inserted into set *S* and *d*[*s*] = δ(*s, s*) = 0 at that time. Because *u* $\neq$ *s*, we also have that $S \neq \emptyset$ just before *u* is inserted into *S*. There must be some path from *s* to *u*, for otherwise *d*[*u*] = δ(*s*, *u*) = $\infty$by Corollary 25.6, which would violate our assumption that *d*[*u*] $\neq$ δ(*s*, *u*). Because there is at least one path, there is a shortest path *p* *fr*om *s* to *u*. 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* $\in$ *V* be *y*'s predecessor. Thus, as shown in Figure 25.6, path *p* can be decomposed as $s \overset{p_1}{\leadsto} x \rightarrow y \overset{p_2}{\leadsto} u$

We claim that *d*[*y*] = δ(*s*, *y*) when *u* is inserted into *S*. To prove this claim, observe that *x* $\in$ *S*. Then, because *u* is chosen as the first vertex for which *d*[*u*] $\neq$ δ(*s*, *u*) when it is inserted into *S*, we had *d*[*x*] = δ(*s*, *x*) when x was inserted into *S*. Edge (*x*, *y*) was relaxed at that time, so the claim follows from Lemma 25.7.

We can now obtain a contradiction to prove the theorem. Because *y* occurs before *u* on a shortest path from *s* to *u* and all edge weights are nonnegative (notably those on path p_2), we have δ(*s*, *y*) $\le$ δ(*s*, *u*), and thus

$$
\begin{aligned}
d[y] &= \delta(s, y) \\
&\le \delta(s, u) \\
&\le d[u] \quad \text{(by Lemma 25.5).}
\end{aligned}
$$

(25.2)

But because both vertices u and y were in $V - S$ when u was chosen in line 5, we have $d[u] \le d[y]$. Thus, the two inequalities in (25.2) are in fact equalities, giving

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

Consequently, $d[u] = \delta(s, u)$, which contradicts our choice of u. We conclude that at the time each vertex $u \in V$ is inserted into set S, we have $d[u] = (s, u)$, and by Lemma 25.5, this equality holds thereafter.

Corollary 25.11

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 25.10 and Lemma 25.9.

Analysis

How fast is Dijkstra's algorithm? Consider first the case in which we maintain the priority queue $Q = V - S$ as a linear array. For such an implementation, each EXTRACT-MIN operation takes time $O(V)$, and there are $|V|$ such operations, for a total EXTRACT-MIN time of $O(V^2)$. Each vertex $v \in V$ is inserted into set S exactly once, so each edge in the adjacency list $Adj[v]$ is examined in the **for** loop of lines 4-8 exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is $|E|$, there are a total of $|E|$ iterations of this **for** loop, with each iteration taking $O(1)$ time. The running time of the entire algorithm is thus $O(V^2 + E) = O(V^2)$.

If the graph is sparse, however, it is practical to implement the priority queue Q with a binary heap. The resulting algorithm is sometimes called the ***modified Dijkstra algorithm***. Each EXTRACT-MIN operation then takes time $O(\lg V)$. As before, there are $|V|$ such operations. The time to build the binary heap is $O(V)$. The assignment $d[v] \leftarrow d[u] + w(u, v)$ in RELAX is accomplished by the call DECREASE-KEY$(Q, v, d[u] + w(u, v))$, which takes time $O(\lg V)$ (see Exercise 7.5-4), 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.

We can in fact achieve a running time of $O(V \lg V + E)$ by implementing the priority queue Q with a Fibonacci heap (see Chapter 21). The amortized cost of each of the $|V|$ EXTRACT-MIN operations is $O(\lg V)$, and each of the $|E|$ DECREASE-KEY calls takes only $O(1)$ amortized time. Historically, the development of Fibonacci heaps was motivated by the observation that in the modified Dijkstra algorithm there are potentially many more DECREASE-KEY calls than EXTRACT-MIN calls, so 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.

Dijkstra's algorithm bears some similarity to both breadth-first search (see Section 23.2) and Prim's algorithm for computing minimum spanning trees (see Section 24.2). 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 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 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), insert this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

Exercises

25.2-1

Run Dijkstra's algorithm on the directed graph of Figure 25.2, first using vertex s as the source and then using vertex y as the source. In the style of Figure 25.5, show the d and π values and the vertices in set S after each iteration of the **while** loop.

25.2-2

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 25.10 go through when negative-weight edges are allowed?

25.2-3

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?

25.2-4

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.

25.2-5

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \{0, 1, . . . , W - 1\}$ for some nonnegative integer W. Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

25.2-6

Modify your algorithm from Exercise 25.2-5 to run in $O((V + E) \lg W)$ time. (*Hint*: How many distinct shortest-path estimates can there be in $V - S$ at any point in time?)

25.3 The Bellman-Ford algorithm

The ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the more general case in which edge weights can be negative. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow \mathbf{R}$, 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.

Like Dijkstra's algorithm, the Bellman-Ford algorithm uses the technique of relaxation, progressively decreasing an estimate $d[v]$ 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 $i \leftarrow 1$ to $|V[G]| - 1$
3      do for each edge $(u, v) \in E[G]$
4              do RELAX($u$, $v$, $w$)
5 for each edge $(u, v) \in E[G]$
6      do if $d[v] > d[u] + w(u, v)$
7            then return FALSE
8 return TRUE
```

Figure 25.7 shows how the execution of the Bellman-Ford algorithm works on a graph with 5 vertices. After performing the usual initialization, 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 25.7(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 shall see a little later why this check works.)

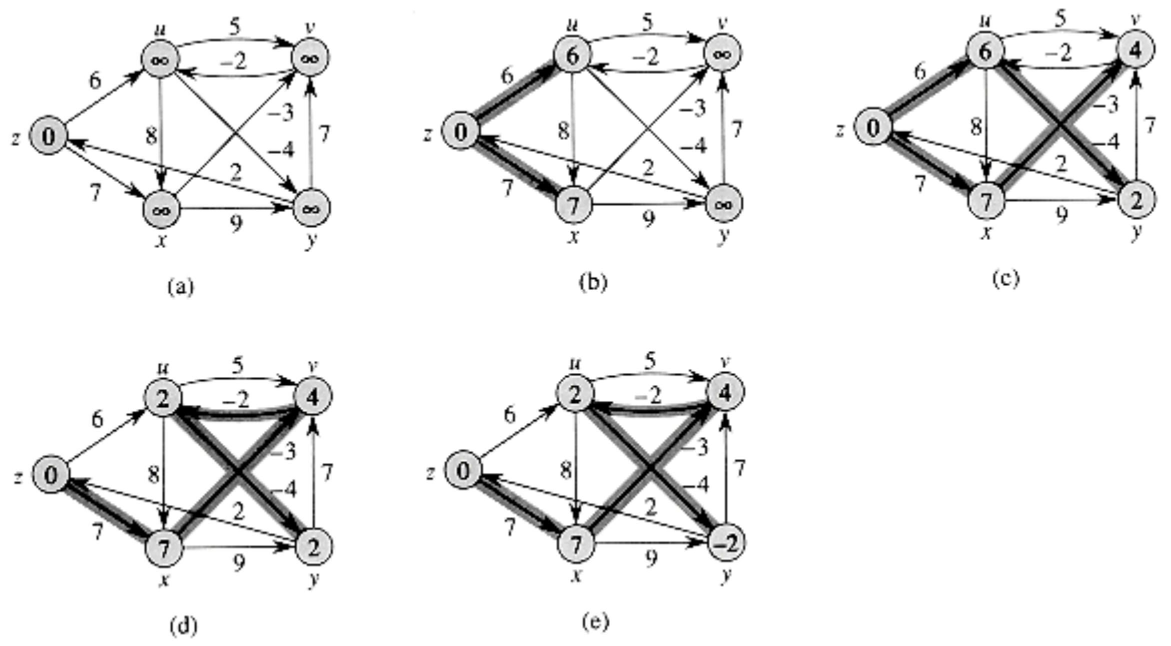

Figure 25.7 The execution of the Bellman-Ford algorithm. The source is vertex z. The d values are shown within the vertices, and shaded edges indicate the π values. In this particular example, each pass relaxes the edges in lexicographic order: (u, v), (u, x),(u, y),(v, u),(x, v),(x, y),(y, v),(y, z),(z, u),(z, x). (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.

The Bellman-Ford algorithm runs in time $O(V E)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V|$ - 1 passes over the edges in lines 2-4 takes $O(E)$ 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. The proof of this lemma contains the intuition behind the algorithm.

Lemma 25.12

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \rightarrow \mathbf{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, at the termination of BELLMAN-FORD, we have $d[v] = \delta(s, v)$ for all vertices v that are reachable from s.

Proof Let v be a vertex reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v, where $v_0 = s$ and $v_k = v$. The path p is simple, and so $k \le |V| - 1$. We want to prove by induction that for $i = 0, 1, \ldots, k$, we have $d[v_i] = \delta(s, v_i)$ after the ith pass over the edges of G and that this equality is maintained thereafter. Because there are $|V|$ - 1 passes, this claim suffices to prove the lemma.

For the basis, we have $d[v_0] = \delta(s, v_0) = 0$ after initialization, and by Lemma 25.5, this equality is maintained thereafter.

For the inductive step, we assume that $d[v_{i-1}] = \delta(s, v_{i-1})$ after the (i - 1)st pass. Edge (v_{i-1}, v_i) is relaxed in the ith pass, so by Lemma 25.7, we conclude that $d[v_i] = \delta(s, v_i)$ after the ith pass and at all subsequent times, thus completing the proof.

Corollary 25.13

Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \rightarrow \mathbf{R}$. Then for each vertex $v \in V$, there is a path from s to v if and only if BELLMAN-FORD terminates with $d[v] < \infty$ when it is run on G.

Proof The proof is similar to that of Lemma 25.12 and is left as Exercise 25.3-2.

Theorem 25.14

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow \mathbf{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $d[v] = \delta(s,v)$ 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, $d[v] = \delta(s, v)$ for all vertices $v \in V$. If vertex v is reachable from s, then Lemma 25.12 proves this claim. If v is not reachable from s, then the claim follows from Corollary 25.6. Thus, the claim is proven. Lemma 25.9, 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 $(u, v) \in E$,

$$d[v] = \delta(s,v)$$

$$\leq \delta(s,u) + w(u,v) \quad \text{(by Lemma 25.3)}$$

$$= d[u] + w(u,v),$$

and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. It therefore returns TRUE.

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

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 . \qquad (25.3)$$

(25.3)

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

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

As in the proof of Lemma 25.8, each vertex in *c* appears exactly once in each of the first two summations. Thus,

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

Moreover, by Corollary 25.13, $d\ [v_i]$ is finite for i = 1, 2, . . . , *k*. Thus,

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

which contradicts inequality (25.3). 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

25.3-1

Run the Bellman-Ford algorithm on the directed graph of Figure 25.7, using vertex *y* as the source. Relax edges in lexicographic order in each pass, and show the *d* and π values after each pass. Now, change the weight of edge (*y, v*) to 4 and run the algorithm again, using *z* as the source.

25.3-2

Prove Corollary 25.13.

25.3-3

Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let *m* be the maximum over all pairs of vertices $u, v \in V$ of the minimum number of edges in a shortest path from *u* 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.

25.3-4

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

25.3-5

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

25.3-6

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.

25.4 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 23.4) to impose a linear ordering on the vertices. If there is 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 each vertex is processed, all the edges that leave the vertex are relaxed.

```
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      do for each vertex v ∈ Adj[u]

5             do RELAX(u,v,w)
```

An example of the execution of this algorithm is shown in Figure 25.8.

The running time of this algorithm is determined by line 1 and by the **for** loop of lines 3-5. As shown in Section 23.4, the topological sort can be performed in $\Theta(V + E)$ time. In the **for** loop of lines 3-5, as in Dijkstra's algorithm, there is one iteration per vertex. For each vertex, the edges that leave the vertex are each examined exactly once. Unlike Dijkstra's algorithm, however, we use only $O(1)$ time per edge. The running time is thus $\Theta(V + E)$, 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.

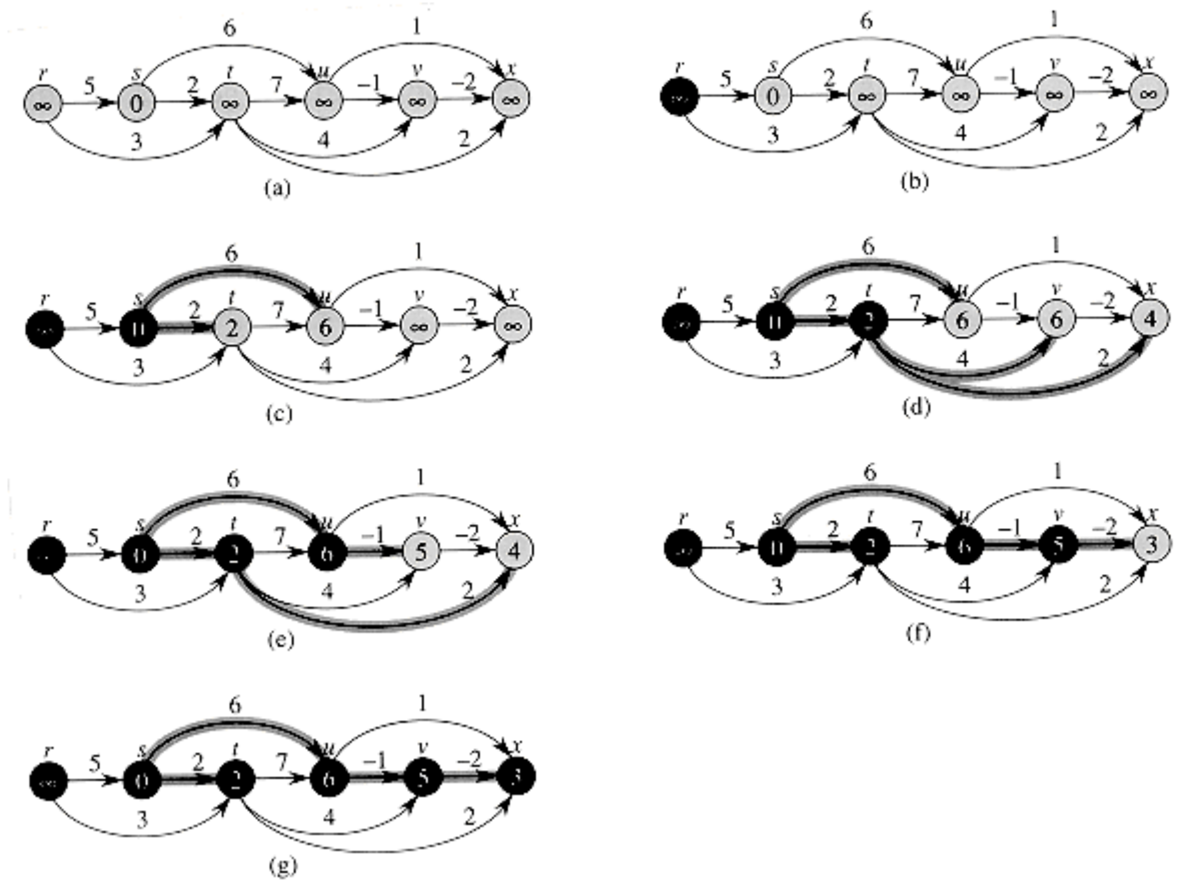

Figure 25.8 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values are shown within the vertices, and shaded edges indicate the π values. (a) The situation before the first iteration 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 v in that iteration. The values shown in part (g) are the final values.

Theorem 25.15

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, $d[v] = \delta(s, v)$ *for all vertices* $v \in V$, *and the predecessor subgraph* G_π is a shortest-paths tree.

Proof We first show that $d[v] = \delta(s, v)$ *for all vertices* $v \in V$ at termination. If v is not reachable from s, then $d[v] = \delta(s, v) = \infty$ *by Corollary 25.6. 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, the edges on p are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. A simple induction using Lemma 25.7 (as in the proof of Lemma 25.12) shows that $d[v_i] = \delta(s, v_i)$ at termination for $i = 0, 1, \ldots, k$. Finally, by Lemma 25.9, 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 prior to 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 an ordered sequence of jobs. The weight of a critical path is 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, replacing "∞" by "-∞" in line 2 of INITIALIZE-SINGLE-SOURCE and ">" by "< " in the RELAX procedure.

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

Exercises

25.4-1

Run DAG-SHORTEST-PATHS on the directed graph of Figure 25.8, using vertex r as the source.

25.4-2

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.

25.4-3

The PERT chart formulation given above is somewhat unnatural. It would be more natural for vertices to represent jobs and edges to represent sequencing constraints; that is, edge (u, v) would indicate that job u must be performed before job v. Weights would then be assigned 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.

25.4-4

Give an efficient algorithm to count the total number of paths in a directed acyclic graph. Analyze your algorithm and comment on its practicality.

25.5 Difference constraints and shortest paths

In the general linear-programming problem, we wish to optimize a linear function subject to a set of linear inequalities. In this section, we investigate a special case of linear programming that can be reduced to finding shortest paths from a single source. The single-source shortest-paths problem that results can then be solved using the Bellman-Ford algorithm, thereby also solving the linear-programming problem.

Linear programming

In the general ***linear-programming problem***, we are given an $m \times n$ matrix A, an m-vector b, and an n-vector c. We wish to find a vector x of n elements that maximizes the ***objective function*** $\sum_{i=1}^{n} c_i x_i$ subject to the m constraints given by $Ax \le b$.

Many problems can be expressed as linear programs, and for this reason much work has gone into algorithms for linear programming. The ***simplex algorithm***[3] solves general linear programs very quickly in practice. With some carefully contrived inputs, however, the simplex method can require exponential time. General linear programs can be solved in polynomial time by either the ***ellipsoid algorithm,*** which runs slowly in practice, or ***Karmarkar's algorithm***, which in practice is often competitive with the simplex method.

[3] The simplex algorithm finds an optimal solution to a linear programming problem by examining a sequence of points in the feasible region--the region in n-space that satisfies $Ax \le b$. The algorithm is based on the fact that a solution that maximizes the objective function over the feasible region occurs at some "extreme point," or "corner," of the feasible region. The simplex algorithm proceeds from corner to corner of the feasible region until no further improvement of the objective function is possible. A "simplex" is the convex hull (see Section 35.3) of $d + 1$ points in d-dimensional space (such as a triangle in the plane, or a tetrahedron in 3-space). According to Dantzig [53], it is possible to view the operation of moving from one corner to another as an operation on a simplex derived from a "dual" interpretation of the linear programming problem--hence the name "simplex method."

Due to the mathematical investment needed to understand and analyze them, this text does not cover general linear-programming algorithms. For several reasons, though, it is important to understand the setup of linear-programming problems. First, knowing that a given problem can be cast as a polynomial-sized linear-programming problem immediately means that there is a polynomial-time algorithm for the problem. Second, there are many special cases of linear programming for which faster algorithms exist. For example, as shown in this section, the single-source shortest-paths problem is a special case of linear programming. Other problems that can be cast as linear programming include the single-pair shortest-path problem (Exercise 25.5-4) and the maximum-flow problem (Exercise 27.1-8).

Sometimes we don't really care about the objective function; we just wish to find any ***feasible solution***, that is, any vector x that satisfies $Ax \le b$, or to determine that no feasible solution exists. We shall focus on one such ***feasibility problem.***

Systems of difference constraints

In a ***system of difference constraints***, each row of the linear-programming matrix A contains one 1 and one - 1, and all other entries of A are 0. Thus, the constraints given by $Ax \le b$ are a set of m ***difference constraints*** involving n unknowns, in which each constraint is a simple linear inequality of the form

$x_j - x_i \le b_k$,

where $1 \le i, j \le n$ and $1 \le k \le m$.

For example, consider problem of finding the 5-vector $x = (x_i)$ that satisfies

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \le \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix} .$$

This problem is equivalent to finding the unknowns x_i, for $i = 1, 2, \ldots, 5$, such that the following 8 difference constraints are satisfied:

$x_1 - x_2 \le 0$,

$x_1 - x_5 \le -1$,

$x_2 - x_5 \le 1$,

$x_3 - x_1 \le 5$,

$x_4 - x_1 \le 4$,

$x_4 - x_3 \le -1$,

$x_5 - x_3 \le -3$,

$x_5 - x_4 \le -3$

(25.4)

One solution to this problem is $x = (-5, -3, 0, -1, -4)$, as can be verified directly by checking each inequality. In fact, there is more than one solution to this problem. Another is $x' = (0, 2, 5, 4, 1)$. These two solutions are related: each component of x' is 5 larger than the corresponding component of x. This fact is not mere coincidence.

Lemma 25.16

Let $x = (x_1, x_2, \ldots, x_n)$ be a solution to a system $Ax \le b$ of difference constraints, and let d be any constant. Then $x + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$ is a solution to $Ax \le b$ as well.

Proof For each x_i and x_j, we have $(x_j + d) - (x_i + d) = x_j - x_i$. Thus, if x satisfies $Ax \le b$, so does $x + d$

Systems of difference constraints occur in many different applications. For example, the unknowns x_i may be times at which events are to occur. Each constraint can be viewed as stating that one event cannot occur too much later than another event. Perhaps the events are jobs to be performed during the construction of a house. If the digging of the foundation begins at time x_1 and takes 3 days and the pouring of the concrete for the foundation begins at time x_2, we may well desire that $x_2 \le x_1 + 3$ or, equivalently, that $x_1 - x_2 \le -3$. Thus, the relative timing constraint can be expressed as a difference constraint.

Constraint graphs

It is beneficial to interpret systems of difference constraints from a graph-theoretic point of view. The idea is that in a system $Ax \le b$ of difference constraints, the $n \times m$ linear-programming matrix A can be viewed as an incidence matrix (see Exercise 23.1-7) for a graph with n vertices and m edges. Each vertex v_i in the graph, for $i = 1, 2, \ldots, n$, corresponds to one of the n unknown variables x_i. Each directed edge in the graph corresponds to one of the m inequalities involving two unknowns.

More formally, given a system $Ax \le b$ of difference constraints, the corresponding ***constraint graph*** is a weighted, directed graph $G= (V, E)$, where

$$V = \{v_0, v_1, \ldots, v_n\}$$

and

$$E = \{(v_i, v_j) : x_j - x_i \le b_k \text{ is a constraint}\}$$

$$\cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_n)\} .$$

The additional vertex v_0 is incorporated, as we shall see shortly, to guarantee that every other vertex is reachable from it. Thus, the vertex set V consists of a vertex v_i for each unknown x_i, plus an additional vertex v_0. The edge set E contains an edge for each difference constraint, plus an edge (v_0, v_i) for each unknown x_i. If $x_j - x_i \le b_k$ is a difference constraint, then the weight of edge (v_i, v_j) is $w(v_i, v_j) = b_k$. The weight of each edge leaving v_0 is 0. Figure 25.9 shows the constraint graph for the system (25.4) of difference constraints.

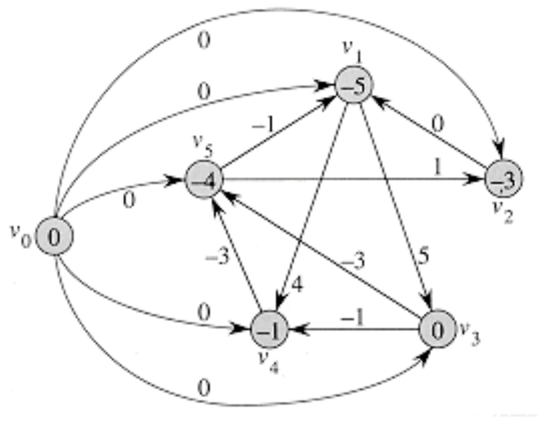

Figure 25.9 The constraint graph corresponding to the system (25.4) of difference constraints. The value of $\delta(v_0, v_j)$ is shown in each vertex v_i. A feasible solution to the

system is x = (-5, -3, 0, -1, -4).

The following theorem shows that a solution to a system of difference constraints can be obtained by finding shortest-path weights in the corresponding constraint graph.

Theorem 25.17

Given a system $Ax \leq b$ of difference constraints, let $G = (V, E)$ be the corresponding constraint graph. If G contains no negative-weight cycles, then

$$x = (\delta(v_0,v_1), \delta(v_0,v_2), \delta(v_0,v_3), \ldots, \delta(v_0,v_n))$$

(25.5)

is a feasible solution for the system. If G contains a negative-weight cycle, then there is no feasible solution for the system.

Proof We first show that if the constraint graph contains no negative-weight cycles, then equation (25.5) gives a feasible solution. Consider any edge $(v_i, v_j) \in E$. By Lemma 25.3, $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$ or, equivalently, $\delta(v_0, v_j) - \delta(v_0, v_j) \leq w(v_i, v_j)$. Thus, letting $x_i = \delta(v_0, v_i)$ and $x_j = \delta(v_0, v_j)$ satisfies the difference constraint $x_j - x_i \leq w(v_i, v_j)$ that corresponds to edge (v_i, v_j).

Now we show that if the constraint graph contains a negative-weight cycle, then the system of difference constraints has no feasible solution. Without loss of generality, let the negative-weight cycle be $c = \langle v_1, v_2, \ldots, v_k \rangle$, where $v_1 = v_k$. (The vertex v_0 cannot be on cycle c, because it has no entering edges.) Cycle c corresponds to the following difference constraints:

$$x_2 - x_1 \leq w(v_1, v_2),$$

$$x_3 - x_2 \leq w(v_2, v_3),$$

$$\vdots$$

$$x_k - x_{k-1} \leq w(v_{k-1}, v_k),$$

$$x_1 - x_k \leq w(v_k, v_1) .$$

Since any solution for x must satisfy each of these k inequalities, any solution must also satisfy the inequality that results when we sum them together. If we sum the left-hand sides, each unknown x_i is added in once and subtracted out once, so that the left-hand side of the sum is 0. The right-hand side sums to $w(c)$, and thus we obtain $0 \leq w(c)$. But since c is a negative-weight cycle, $w(c) \leq 0$, and hence any solution for the x must satisfy $0 \leq w(c) \leq 0$, which is impossible.

Solving systems of difference constraints

Theorem 25.17 tells us that we can use the Bellman-Ford algorithm to solve a system of difference constraints. Because there are edges from the source vertex v_0 to all other vertices in the constraint graph, any negative-weight cycle in the constraint graph is reachable from v_0. If the Bellman-Ford algorithm returns TRUE, then the shortest-path weights give a feasible solution to the system. In Figure 25.9, for example, the shortest-path weights provide the feasible solution $x = (-5, -3, 0, -1, -4)$, and by Lemma 25.16, $x = (d - 5, d - 3, d, d - 1, d - 4)$ is also a feasible solution for any constant d. If the Bellman-Ford algorithm returns FALSE, there is no feasible solution to the system of difference constraints.

A system of difference constraints with m constraints on n unknowns produces a graph with $n + 1$ vertices and $n + m$ edges. Thus, using the Bellman-Ford algorithm, we can solve the system in $O((n + 1)(n + m)) = O(n^2+nm)$ time. Exercise 25.5-5 asks you to show that the algorithm actually runs in $O(nm)$ time, even if m is much less than n.

Exercises

25.5-1

Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:

$x_1 - x_2 \leq 1,$

$x_1 - x_4 \leq -4,$

$x_2 - x_3 \leq 2,$

$x_2 - x_5 \leq 7,$

$x_2 - x_6 \leq 5,$

$x_3 - x_6 \leq 10,$

$x_4 - x_2 \leq 2,$

$x_5 - x_1 \leq -1,$

$x_5 - x_4 \leq 3,$

$x_6 - x_3 \leq -8.$

25.5-2

Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:

$x1 - x2 \leq 4,$

$x1 - x5 \leq 5,$

$x2 - x4 \leq -6,$

$x3 - x2 \leq 1,$

```
x4 - x1  ≤   3,

x4 - x3  ≤   5,

x4 - x5  ≤  10,

x5 - x3  ≤  -4,

x5 - x4  ≤  -8.
```

25.5-3

Can any shortest-path weight from the new vertex v_0 in a constraint graph be positive? Explain.

25.5-4

Express the single-pair shortest-path problem as a linear program.

25.5-5

Show how to modify the Bellman-Ford algorithm slightly so that when it is used to solve a system of difference constraints with *m* inequalities on *n* unknowns, the running time is $O(nm)$.

25.5-6

Show how a system of difference constraints can be solved by a Bellman-Ford-like algorithm that runs on a constraint graph without the extra vertex v_0.

25.5-7

Let $Ax \leq b$ be a system of *m* difference constraints in *n* unknowns. Show that the Bellman-Ford algorithm, when run on the corresponding constraint graph, maximizes $\sum_{i=1}^{n} x_i$ subject to $Ax \leq b$ and $x_i \leq 0$ for all x_i.

25.5-8

Show that the Bellman-Ford algorithm, when run on the constraint graph for a system $Ax \leq b$ of difference constraints, minimizes the quantity (max $\{x_i\}$ - min $\{x_i\}$) subject to $Ax \leq b$. Explain how this fact might come in handy if the algorithm is used to schedule construction jobs.

25.5-9

Suppose that every row in the matrix *A* of a linear program $Ax \leq b$ corresponds to a difference constraint, a single-variable constraint of the form $x_i \leq b_k$, or a single-variable constraint of the form $-x_i \leq b_k$. Show how the Bellman-Ford algorithm can be adapted to solve this variety of constraint system.

25.5-10

Suppose that in addition to a system of difference constraints, we want to handle equality constraints of the form $x_i = x_j + b_k$. Show how the Bellman-Ford algorithm can be adapted to solve this variety of constraint system.

25.5-11

Give an efficient algorithm to solve a system $Ax \le b$ of difference constraints when all of the elements of b are real-valued and all of the unknowns x_i must be integers.

25.5-12

Give an efficient algorithm to solve a system $Ax \le b$ of difference constraints when all of the elements of b are real-valued and some, but not necessarily all, of the unknowns x_i must be integers.

Problems

25-1 Yen's improvement to Bellman-Ford

Suppose that we order the edge relaxations in each pass of the Bellman-Ford algorithm as follows. Before the first pass, we assign an arbitrary linear order $v_1, v_2, \ldots, v_{|V|}$, to the vertices of the input graph $G = (V, E)$. Then, we partition the edge set E into $E_f \cup E_b$, where $E_f = \{(v_i, v_j) \in E: i < j\}$ and $E_b = \{(v_i, v_j) \in E: i > j\}$. Define $G_f = (V, E_f)$ and $G_b = (V, E_b)$.

a. Prove that G_f is acyclic with topological sort $\langle v_1, v_2, \ldots, v_{|V|}\rangle$ and that G_b is acyclic with topological sort $(v_{V|}, v_{|V|-1}, \ldots v1)$.

Suppose that we implement each pass of the Bellman-Ford algorithm in the following way. We visit each vertex in the order $v_1, v_2, \ldots, v_{|V|}$ relaxing edges of E_f that leave the vertex. We then visit each vertex in the order $v_{|V|}, v_{|V|-1}, \ldots v_1$, relaxing edges of E_b that leave the vertex.

b. Prove that with this scheme, if G contains no negative-weight cycles that are reachable from the source vertex s, then after only $\lceil |V|/2 \rceil$ passes over the edges, $d[v] = \delta(s, v)$ for all vertices $v \in V$.

c. How does this scheme affect the running time of the Bellman-Ford algorithm?

25-2 Nesting boxes

A d-dimensional box with dimensions $(x_1, x_2, \ldots, x_d)$ ***nests*** within another box with dimensions $(y_1, y_2, \ldots, y_d)$ if there exists a permutation π on $\{1, 2, \ldots, d\}$ such that $x_{\pi(1)} < y_1, x_{\pi(d)} < y_d$.

a. Argue that the nesting relation is transitive.

b. Describe an efficient method to determine whether or not one d-dimensional box nests inside another.

c. Suppose that you are given a set of n d-dimensional boxes $\{B_1, B_2, \ldots, B_n\}$. Describe an efficient algorithm to determine the longest sequence $\langle B_i1, B_i2, \ldots, B_{ik}\rangle$ of boxes such that B_{ij} nests within $B_{ij}+1$ for $j = 1, 2, \ldots, k - 1$. Express the running time of your algorithm in terms of n and d.

25-3 Arbitrage

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 0.7 British pound, 1 British pound buys 9.5 French francs, and 1 French franc buys 0.16 U.S. dollar. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $0.7 \times 9.5 \times 0.16 = 1.064$ U.S. dollars, thus turning a profit of 6.4 percent.

Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j.

a. Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_i1, c_i2, \ldots, c_{ik}\rangle$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Analyze the running time of your algorithm.

b. Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

25-4 Gabow's scaling algorithm for single-source shortest paths

A ***scaling*** algorithm solves a problem by initially considering only the highest-order bit of each relevant input value (such as an edge weight). It then refines the initial solution by looking at the two highest-order bits. It progressively looks at more and more high-order bits, refining the solution each time, until all bits have been considered and the correct solution has been computed.

In this problem, we examine an algorithm for computing the shortest paths from a single source by scaling edge weights. We are given a directed graph $G = (V, E)$ with nonnegative integer edge weights w. Let $W = \max_{(u,v) \in E} \{w(u, v)\}$. Our goal is to develop an algorithm that runs in $O(E \lg W)$ time.

The algorithm uncovers the bits in the binary representation of the edge weights one at a time, from the most significant bit to the least significant bit. Specifically, let $k = \lceil \lg (W + 1) \rceil$ be the number of bits in the binary representation of W, and for $i = 1, 2, \ldots, k$, let $w_i(u, v) = \lfloor w(u, v)/2^{k-i} \rfloor$. That is, $w_i(u, v)$ is the "scaled-down" version of $w(u, v)$ given by the i most significant bits of $w(u, v)$. (Thus, $w_k(u, v) = w(u, v)$ for all $(u, v) \in E$.) For example, if $k = 5$ and $w(u, v) = 25$, which has the binary representation $\langle 11001\rangle$, then $w_3(u, v) = \langle 110\rangle =$

6. As another example with $k = 5$, if $w(u, v) = \langle 00100 \rangle = 4$, then $w_3(u, v) = \langle 001 \rangle = 1$. Let us define $\delta_i(u, v)$ as the shortest-path weight from vertex u to vertex v using weight function w_i. Thus, $\delta_k(u, v) = \delta(u, v)$ for all $u, v \in V$. For a given source vertex s, the scaling algorithm first computes the shortest-path weights $\delta_1(s, v)$ for all $v \in V$, then computes $\delta_2(s, v)$ for all $v \in V$, and so on, until it computes $\delta_k(s, v)$ for all $v \in V$. We assume throughout that $|E| \ge |V| - 1$, and we shall see that computing δ_i from δ_{i-1}takes $O(E)$ time, so that the entire algorithm takes $O(kE) = O(E \lg W)$ time.

a. uppose that for all vertices $v \in V$, we have $\delta(s, v) \le |E|$. Show that we can compute $\delta(s, v)$ for all $v \in V$ in $O(E)$ time.

b. Show that we can compute $\delta_1(s, v)$ for all $v \in V$ in $O(E)$ time.

Let us now concentrate on computing δ_i from δ_{i-1}.

c. Prove that for $i = 2, 3, \ldots, k$, either $w_i(u, v) = 2w_{i-1}(u, v)$ or $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Then, prove that

$$2\delta_{i-1}(s,v) \le \delta_i(s,v) \le 2\delta_{i-1}(s,v) + |V| - 1$$

for all $v \in V$.

d. Define for $i = 2, 3, \ldots, k$ and all $(u, v) \in E$,

$$\widehat{w}_i(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v) .$$

Prove that for $i = 2, 3, \ldots, k$ and all $u, v \in V$, the "reweighted" value $\widehat{w}_i(u,v)$ of edge (u, v) is a nonnegative integer.

e. Now, define $\widehat{\delta}_i(s,v)$ as the shortest-path weight from s to v using the weight function $\widehat{w}_i$. Prove that for $i = 2, 3, \ldots, k$ and all $v \in V$,

$$\delta_i(s,v) = \widehat{\delta}_i(s,v) + 2\delta_{i-1}(s,v)$$

and that $\widehat{\delta}_i(s,v) \le |E|$.

f. Show how to compute $\delta_i(s, v)$ from $\delta_{i-1}(s, v)$ for all $v \in V$ in $O(E)$ time, and conclude that $\delta(s, v)$ can be computed for all $v \in V$ in $O(E \lg W)$ time.

25-5 Karp's minimum mean-weight cycle algorithm

Let $G = (V, E)$ be a directed graph with weight function $w: E \rightarrow \mathbf{R}$, and let $n = |V|$. We define the ***mean weight*** of a cycle $c = \langle e_1, e_2, \ldots, e_k \rangle$ of edges in E to be

$$\mu(c) = \frac{1}{k}\sum_{i=1}^{k} w(e_i) .$$

Let $\mu^* = \min_c \mu(c)$, where c ranges over all directed cycles in G. A cycle c for which $\mu(c) = \mu^*$

is called a ***minimum mean-weight cycle.*** This problem investigates an efficient algorithm for computing μ^*.

Assume without loss of generality that every vertex $v \in V$ is reachable from a source vertex $s \in V$. Let $\delta(s, v)$ be the weight of a shortest path from s to v, and let $\delta_k(s, v)$ be the weight of a shortest path from s to v consisting of *exactly* k edges. If there is no path from s to v with exactly k edges, then $\delta_k(s, v) = \infty$.

a. Show that if $\mu^* = 0$, then G contains no negative-weight cycles and $\delta(s, v) = \min_{0 \le k \le n-1} \delta_k(s, v)$ *for all vertices* $v \in V$.

b. Show that if $\mu^* = 0$, then

$$\max_{0\le k\le n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} \ge 0$$

for all vertices $v \in V$. (*Hint:* Use both properties from part (a).)

c. Let c be a 0-weight cycle, and let u and v be any two vertices on c. Suppose that the weight of the path from u to v along the cycle is x. Prove that $\delta(s, v) = \delta(s, u) + x$. (*Hint*: The weight of the path from v to u along the cycle is $-x$.)

d. Show that if $\mu^* = 0$, then there exists a vertex v on the minimum mean-weight cycle such that

$$\max_{0\le k\le n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} = 0 .$$

(*Hint*: Show that a shortest path to any vertex on the minimum mean-weight cycle can be extended along the cycle to make a shortest path to the next vertex on the cycle.)

e. Show that if $\mu^* = 0$, then

$$\min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} = 0 .$$

f. Show that if we add a constant t to the weight of each edge of G, then μ^* is increased by t. Use this to show that

$$\mu^* = \min_{v\in V} \max_{0\le k\le n-1} \frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} .$$

g. Give an $O(V E)$-time algorithm to compute μ^*.

Chapter notes

Dijkstra's algorithm [55] appeared in 1959, but it contained no mention of a priority queue. The Bellman-Ford algorithm is based on separate algorithms by Bellman [22] and

Ford [71]. Bellman describes the relation of shortest paths to difference constraints. Lawler [132] describes the linear-time algorithm for shortest paths in a dag, which he considers part of the folklore.

When edge weights are relatively small integers, more efficient algorithms can be used to solve the single-source shortest-paths problem. Ahuja, Mehlhorn, Orlin, and Tarjan [6] give an algorithm that runs in $O(E + V\sqrt{\lg W})$ time on graphs with nonnegative edge weights, where *W* is the largest weight of any edge in the graph. They also give an easily programmed algorithm that runs in $O(E + V \lg W)$ time. For graphs with negative edge weights, the algorithm due to Gabow and Tarjan [77] runs in $O(\sqrt{V}E\lg(VW))$ time, where the magnitude of the largest-magnitude weight of any edge in the graph.

Papadimitriou and Steiglitz [154] have a good discussion of the simplex method and the ellipsoid algorithm as well as other algorithms related to linear programming. The simplex algorithm for linear programming was invented by G. Danzig in 1947. Variants of simplex remain the most popular method for solving linear-programming problems. The ellipsoid algorithm is due to L. G. Khachian in 1979, based on earlier work by N. Z. Shor, D. B. Judin, and A. S. Nemirovskii. Karmarkar describes his algorithm in [115].

Go to Chapter 26   Back to Table of Contents