# 22 Single-Source Shortest Paths

Suppose that you need to drive from Oceanside, New York, to Oceanside, California, by the shortest possible route. Your GPS contains information about the entire road network of the United States, including the road distance between each pair of adjacent intersections. How can your GPS determine this shortest route?

One possible way is to enumerate all the routes from Oceanside, New York, to Oceanside, California, add up the distances on each route, and select the shortest. But even disallowing routes that contain cycles, your GPS would need to examine an enormous number of possibilities, most of which are simply not worth considering. For example, a route that passes through Miami, Florida, is a poor choice, because Miami is several hundred miles out of the way.

This chapter and Chapter 23 show how to solve such problems efficiently. The input to a ***shortest-paths problem*** is a weighted, directed graph $G = (V, E)$, with a weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights. The ***weight*** $w(p)$ 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*** $\delta(u, v)$ from $u$ to $v$ by

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

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 example of going from Oceanside, New York, to Oceanside, California, your GPS models the road network as a graph: vertices represent intersections, edges represent road segments between intersections, and edge weights represent road distances. The goal is to find a shortest path from a given intersection in

Oceanside, New York (say, Brower Avenue and Skillman Avenue) to a given intersection in Oceanside, California (say, Topeka Street and South Horne Street).

Edge weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that you want to minimize.

The breadth-first-search algorithm from Section 20.2 is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge has unit weight. Because many of the concepts from breadth-first search arise in the study of shortest paths in weighted graphs, you might want to review Section 20.2 before proceeding.

### Variants

This chapter focuses on the ***single-source shortest-paths problem***: given a graph $G = (V, E)$, find a shortest path from a given ***source vertex*** $s \in V$ to every vertex $v \in V$. The algorithm for the single-source problem can solve many other problems, including the following variants.

**Single-destination shortest-paths problem:** Find a shortest path to a given ***destination vertex*** $t$ from each vertex $v$. By reversing the direction of each edge in the graph, you 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 you solve the single-source problem with source vertex $u$, you solve this problem also. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.

**All-pairs shortest-paths problem:** Find a shortest path from $u$ to $v$ for every pair of vertices $u$ and $v$. Although you can solve this problem by running a single-source algorithm once from each vertex, you often can solve it faster. Additionally, its structure is interesting in its own right. Chapter 23 addresses the all-pairs problem in detail.

### Optimal substructure of a shortest path

Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it. (The Edmonds-Karp maximum-flow algorithm in Chapter 24 also relies on this property.) Recall that optimal substructure is one of the key indicators that dynamic programming (Chapter 14) and the greedy method (Chapter 15) might apply. Dijkstra's algorithm, which we shall see in Section 22.3, is a greedy algorithm, and the Floyd-Warshall algorithm, which finds a shortest path between every pair of vertices (see Sec-

tion 23.2), is a dynamic-programming algorithm. The following lemma states the optimal-substructure property of shortest paths more precisely.

***Lemma 22.1 (Subpaths of shortest paths are shortest paths)***
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex $v_0$ to vertex $v_k$ and, for any $i$ and $j$ such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ 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*** Decompose path $p$ into $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, so that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path $p'_{ij}$ from $v_i$ to $v_j$ with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ is a path from $v_0$ to $v_k$ whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that $p$ is a shortest path from $v_0$ to $v_k$. ■

### Negative-weight edges

Some instances of the single-source shortest-paths problem may include 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 the graph contains 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—you can always find a path with lower weight by following the proposed "shortest" path and then traversing 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 22.1 illustrates the effect of negative weights and negative-weight cycles on shortest-path weights. Because there is only one path from $s$ to $a$ (the path $\langle s, a \rangle$), we have $\delta(s, a) = w(s, a) = 3$. 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 $\delta(s, c) = w(s, c) = 5$, and 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. Because 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, you 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 $f$, you can also find paths with arbitrarily large negative weights from $s$ to $g$,

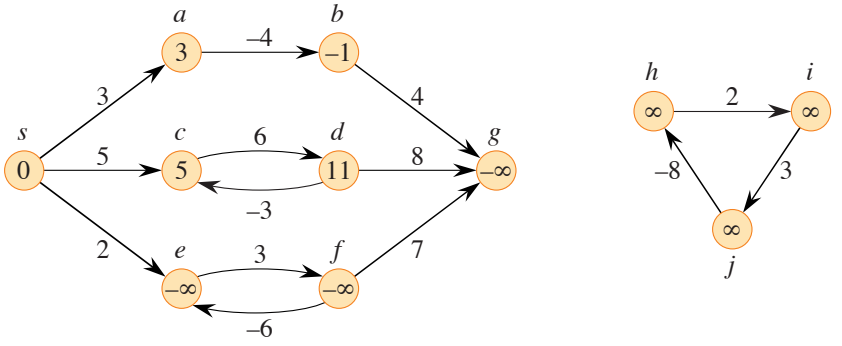


**Figure 22.1** Negative edge weights in a directed graph. The shortest-path weight from source $s$ appears within each vertex. Because vertices $e$ and $f$ form a negative-weight cycle reachable from $s$, they have shortest-path weights of $-\infty$. Because vertex $g$ is reachable from a vertex whose shortest-path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$. Vertices such as $h$, $i$, and $j$ are not reachable from $s$, and so their shortest-path weights are $\infty$, even though they lie on a negative-weight cycle.

and so $\delta(s, g) = -\infty$. Vertices $h, i$, and $j$ also form a negative-weight cycle. They are not reachable from $s$, however, and so $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

Some shortest-paths algorithms, such as Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative, as in a road network. 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.

## Cycles

Can a shortest path contain a cycle? As we have just seen, it cannot contain a negative-weight cycle. Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight. That is, if $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a path and $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a positive-weight cycle on this path (so that $v_i = v_j$ and $w(c) > 0$ ), then the path $p' = \langle v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k \rangle$ has weight $w(p') = w(p) - w(c) < w(p)$, and so $p$ cannot be a shortest path from $v_0$ to $v_k$.

That leaves only 0-weight cycles. You can remove a 0-weight cycle from any path to produce another path whose weight is the same. Thus, if there is a shortest path from a source vertex $s$ to a destination vertex $v$ that contains a 0-weight cycle, then there is another shortest path from $s$ to $v$ without this cycle. As long as a shortest path has 0-weight cycles, you can repeatedly remove these cycles from the path until you have a shortest path that is cycle-free. Therefore, without loss of

generality, assume that shortest paths have no cycles, that is, they are simple paths. Since any acyclic path in a graph $G = (V, E)$ contains at most $|V|$ distinct vertices, it also contains at most $|V| - 1$ edges. Assume, therefore, that any shortest path contains at most $|V| - 1$ edges.

### Representing shortest paths

It is usually not enough to compute only shortest-path weights. Most applications of shortest paths need to know the vertices on shortest paths as well. For example, if your GPS told you the distance to your destination but not how to get there, it would not be terribly useful. We represent shortest paths similarly to how we represented breadth-first trees in Section 20.2. Given a graph $G = (V, E)$, maintain for each vertex $v \in V$ a ***predecessor*** $v.\pi$ that is either another vertex or NIL. The shortest-paths algorithms in this chapter set the $\pi$ attributes so that the chain of predecessors originating at a vertex $v$ runs backward along a shortest path from $s$ to $v$. Thus, given a vertex $v$ for which $v.\pi \neq \text{NIL}$, the procedure PRINT-PATH$(G, s, v)$ from Section 20.2 prints a shortest path from $s$ to $v$.

In the midst of executing a shortest-paths algorithm, however, the $\pi$ values might not indicate shortest paths. The ***predecessor subgraph*** $G_\pi = (V_\pi, E_\pi)$ induced by the $\pi$ values is defined the same for single-source shortest paths as for breadth-first search in equations (20.2) and (20.3) on page 561:

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\} ,$$
$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\} .$$

We'll prove that the $\pi$ 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 the source $s$ to every vertex that is reachable from $s$. A shortest-paths tree is like the breadth-first tree from Section 20.2, 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 $w : E \to \mathbb{R}$, 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 $G$.

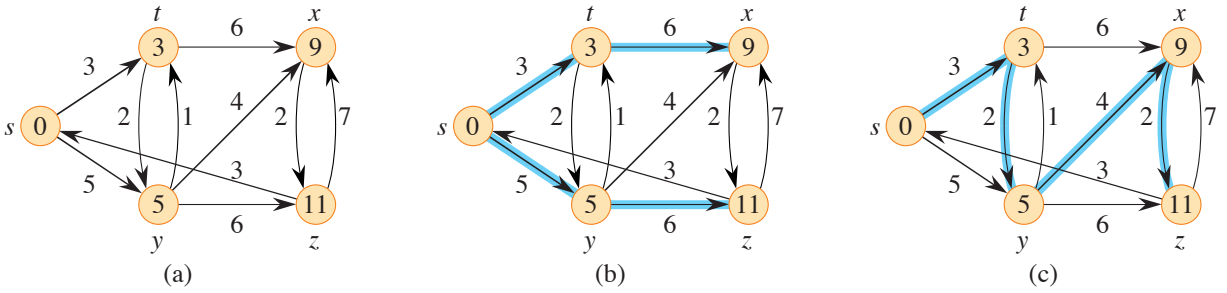


**Figure 22.2** **(a)** A weighted, directed graph with shortest-path weights from source $s$. **(b)** The blue edges form a shortest-paths tree rooted at the source $s$. **(c)** Another shortest-paths tree with the same root.

Shortest paths are not necessarily unique, and neither are shortest-paths trees. For example, Figure 22.2 shows a weighted, directed graph and two shortest-paths trees with the same root.

### Relaxation

The algorithms in this chapter use the technique of ***relaxation***. For each vertex $v \in V$, the single-source shortest paths algorithms maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source $s$ to $v$. We call $v.d$ a ***shortest-path estimate***. To initialize the shortest-path estimates and predecessors, call the $\Theta(V)$-time procedure INITIALIZE-SINGLE-SOURCE. After initialization, we have $v.\pi = \text{NIL}$ for all $v \in V$, $s.d = 0$ and $v.d = \infty$ for $v \in V - \{s\}$.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

The process of ***relaxing*** an edge $(u, v)$ consists of testing whether going through vertex $u$ improves the shortest path to vertex $v$ found so far and, if so, updating $v.d$ and $v.\pi$. A relaxation step might decrease the value of the shortest-path estimate $v.d$ and update $v$'s predecessor attribute $v.\pi$. The RELAX procedure on the following page performs a relaxation step on edge $(u, v)$ in $O(1)$ time. Figure 22.3 shows two examples of relaxing an edge, one in which a shortest-path estimate decreases and one in which no estimate changes.

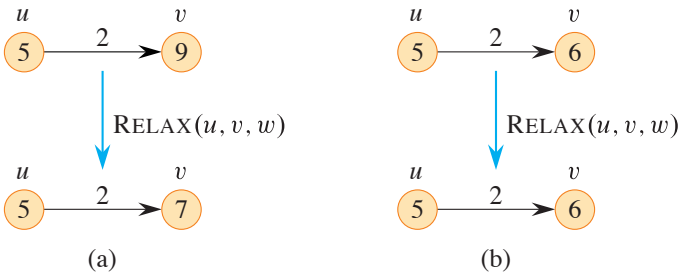


**Figure 22.3** Relaxing an edge $(u, v)$ with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Since we have $v.d \leq u.d + w(u, v)$ before relaxing the edge, the relaxation step leaves $v.d$ unchanged.

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Each algorithm in this chapter calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges.[1] 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. Dijkstra's algorithm and the shortest-paths algorithm for directed acyclic graphs relax each edge exactly once. The Bellman-Ford algorithm relaxes each edge $|V| - 1$ times.

### Properties of shortest paths and relaxation

To prove the algorithms in this chapter correct, we'll appeal to several properties of shortest paths and relaxation. We state these properties here, and Section 22.5 proves them formally. For your reference, each property stated here includes the appropriate lemma or corollary number from Section 22.5. The latter five of these properties, which refer to shortest-path estimates or the predecessor subgraph, im-

[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 $v.d \leq u.d + w(u, v)$, which, by the triangle inequality (Lemma 22.10 on page 633), must be satisfied if $u.d = \delta(s, u)$ and $v.d = \delta(s, v)$. That is, if $v.d \leq u.d + w(u, v)$, there is no "pressure" to satisfy this constraint, so the constraint is "relaxed."

plicitly assume that the graph is initialized with a call to INITIALIZE-SINGLE-SOURCE$(G, s)$ and that the only way that shortest-path estimates and the predecessor subgraph change are by some sequence of relaxation steps.

**Triangle inequality** (Lemma 22.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

**Upper-bound property** (Lemma 22.11)
We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

**No-path property** (Corollary 22.12)
If there is no path from $s$ to $v$, then we always have $v.d = \delta(s, v) = \infty$.

**Convergence property** (Lemma 22.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in $G$ for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge $(u, v)$, then $v.d = \delta(s, v)$ at all times afterward.

**Path-relaxation property** (Lemma 22.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to $v_k$, and the edges of $p$ are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of $p$.

**Predecessor-subgraph property** (Lemma 22.17)
Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at $s$.

### Chapter outline

Section 22.1 presents the Bellman-Ford algorithm, which solves the single-source shortest-paths problem in the general case in which edges can have negative weight. The Bellman-Ford algorithm is remarkably simple, and it has the further benefit of detecting whether a negative-weight cycle is reachable from the source. Section 22.2 gives a linear-time algorithm for computing shortest paths from a single source in a directed acyclic graph. Section 22.3 covers Dijkstra's algorithm, which has a lower running time than the Bellman-Ford algorithm but requires the edge weights to be nonnegative. Section 22.4 shows how to use the Bellman-Ford algorithm to solve a special case of linear programming. Finally, Section 22.5 proves the properties of shortest paths and relaxation stated above.

This chapter does arithmetic with infinities, and so we need some conventions for when $\infty$ or $-\infty$ appears in an arithmetic expression. We 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 assume that for any real number $a \neq \infty$, we have $a + (-\infty) = (-\infty) + a = -\infty$.

## 22.3 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$, but it requires nonnegative weights on all edges: $w(u, v) \geq 0$ 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.

You can think of Dijkstra's algorithm as generalizing breadth-first search to weighted graphs. A wave emanates from the source, and the first time that a wave arrives at a vertex, a new wave emanates from that vertex. Whereas breadth-first search operates as if each wave takes unit time to traverse an edge, in a weighted graph, the time for a wave to traverse an edge is given by the edge's weight. Because a shortest path in a weighted graph might not have the fewest edges, a simple, first-in, first-out queue won't suffice for choosing the next vertex from which to send out a wave.

Instead, 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 $u$ into $S$, and relaxes all edges leaving $u$. The procedure DIJKSTRA replaces the first-in, first-out queue of breadth-first search by a min-priority queue $Q$ of vertices, keyed by their $d$ values.

```
DIJKSTRA(G, w, s)
 1  INITIALIZE-SINGLE-SOURCE(G, s)
 2  S = ∅
 3  Q = ∅
 4  for each vertex u ∈ G.V
 5      INSERT(Q, u)
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      S = S ∪ {u}
 9      for each vertex v in G.Adj[u]
10          RELAX(u, v, w)
11          if the call of RELAX decreased v.d
12              DECREASE-KEY(Q, v, v.d)
```

Dijkstra's algorithm relaxes edges as shown in Figure 22.6. Line 1 initializes the $d$ and $\pi$ 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

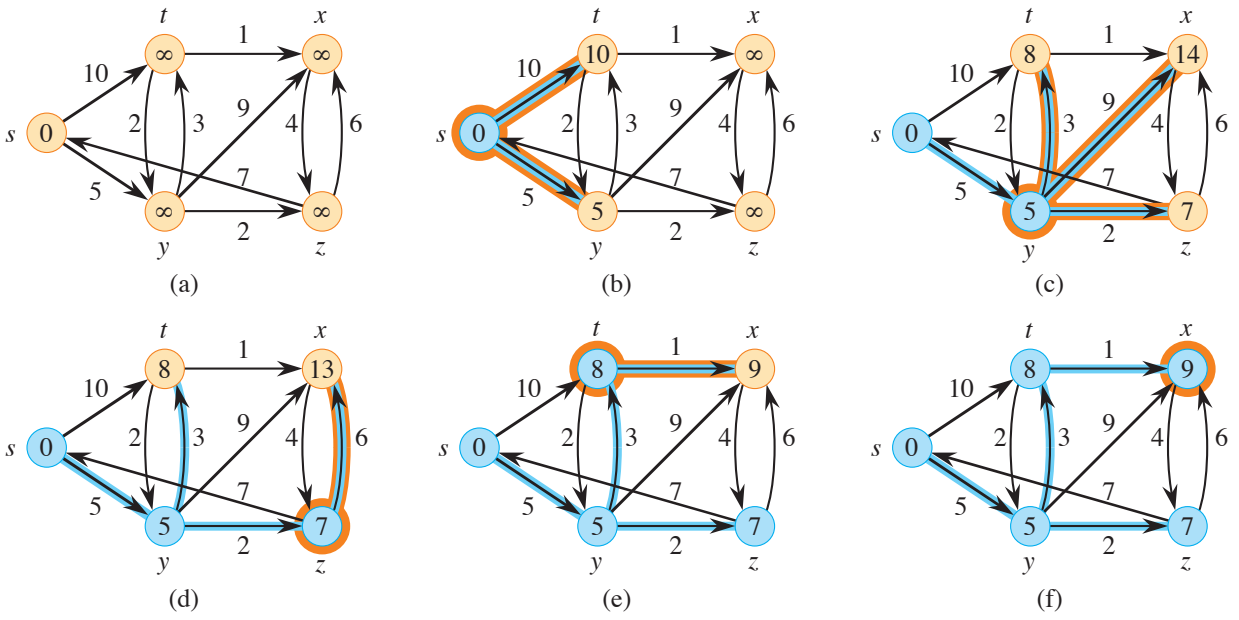


**Figure 22.6** The execution of Dijkstra's algorithm. The source $s$ is the leftmost vertex. The shortest-path estimates appear within the vertices, and blue edges indicate predecessor values. Blue vertices belong to the set $S$, and tan vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 6–12. **(b)–(f)** The situation after each successive iteration of the **while** loop. In each part, the vertex highlighted in orange was chosen as vertex $u$ in line 7, and each edge highlighted in orange caused a $d$ value and a predecessor to change when the edge was relaxed. The $d$ values and predecessors shown in part (f) are the final values.

of the **while** loop of lines 6–12. Lines 3–5 initialize the min-priority queue $Q$ to contain all the vertices in $V$. Since $S = \emptyset$ at that time, the invariant is true upon first reaching line 6. Each time through the **while** loop of lines 6–12, line 7 extracts a vertex $u$ from $Q = V - S$ and line 8 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 9–12 relax each edge $(u, v)$ leaving $u$, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if the shortest path to $v$ found so far improves by going through $u$. Whenever a relaxation step changes the $d$ and $\pi$ values, the call to DECREASE-KEY in line 12 updates the min-priority queue. The algorithm never inserts vertices into $Q$ after the **for** loop of lines 4–5, and each vertex is extracted from $Q$ and added to $S$ exactly once, so that the **while** loop of lines 6–12 iterates exactly $|V|$ times.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set $S$, you can think of it as using a greedy strategy. Chapter 15 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

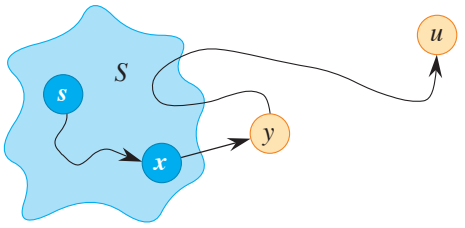


**Figure 22.7** The proof of Theorem 22.6. Vertex $u$ is selected to be added into set $S$ in line 7 of DIJKSTRA. Vertex $y$ is the first vertex on a shortest path from the source $s$ to vertex $u$ that is not in set $S$, and $x \in S$ is $y$'s predecessor on that shortest path. The subpath from $y$ to $u$ may or may not re-enter set $S$.

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 $u.d = \delta(s, u)$ each time it adds a vertex $u$ to set $S$.

***Theorem 22.6 (Correctness of Dijkstra's algorithm)***
Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function $w$ and source vertex $s$, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

***Proof*** We will show that at the start of each iteration of the **while** loop of lines 6–12, we have $v.d = \delta(s, v)$ for all $v \in S$. The algorithm terminates when $S = V$, so that $v.d = \delta(s, v)$ for all $v \in V$.

The proof is by induction on the number of iterations of the **while** loop, which equals $|S|$ at the start of each iteration. There are two bases: for $|S| = 0$, so that $S = \emptyset$ and the claim is trivially true, and for $|S| = 1$, so that $S = \{s\}$ and $s.d = \delta(s, s) = 0$.

For the inductive step, the inductive hypothesis is that $v.d = \delta(s, v)$ for all $v \in S$. The algorithm extracts vertex $u$ from $V - S$. Because the algorithm adds $u$ into $S$, we need to show that $u.d = \delta(s, u)$ at that time. If there is no path from $s$ to $u$, then we are done, by the no-path property. If there is a path from $s$ to $u$, then, as Figure 22.7 shows, let $y$ be the first vertex on a shortest path from $s$ to $u$ that is not in $S$, and let $x \in S$ be the predecessor of $y$ on that shortest path. (We could have $y = u$ or $x = s$.) Because $y$ appears no later than $u$ on the shortest path and all edge weights are nonnegative, we have $\delta(s, y) \leq \delta(s, u)$. Because the call of EXTRACT-MIN in line 7 returned $u$ as having the minimum $d$ value in $V - S$, we also have $u.d \leq y.d$, and the upper-bound property gives $\delta(s, u) \leq u.d$.

Since $x \in S$, the inductive hypothesis implies that $x.d = \delta(s, x)$. During the iteration of the **while** loop that added $x$ into $S$, edge $(x, y)$ was relaxed. By the convergence property, $y.d$ received the value of $\delta(s, y)$ at that time. Thus, we have

$$\delta(s, y) \le \delta(s, u) \le u.d \le y.d \quad \text{and} \quad y.d = \delta(s, y) ,$$

so that

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

Hence, $u.d = \delta(s, u)$, and by the upper-bound property, this value never changes again. ■

***Corollary 22.7***
After Dijkstra's algorithm is run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function $w$ and source vertex $s$, the predecessor subgraph $G_\pi$ is a shortest-paths tree rooted at $s$.

***Proof*** Immediate from Theorem 22.6 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 (in line 5), EXTRACT-MIN (in line 7), and DECREASE-KEY (in line 12). 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 9–12 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.)

Just as in Prim's algorithm, the running time of Dijkstra's algorithm depends on the specific implementation of the min-priority queue $Q$. A simple implementation takes advantage of the vertices being numbered 1 to $|V|$: simply store $v.d$ in the $v$th entry of an array. Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT-MIN operation takes $O(V)$ time (since it has 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, $E = o(V^2/\lg V)$—you can improve the running time by implementing the min-priority queue with a binary min-heap that includes a way to map between vertices and their corresponding heap elements. Each EXTRACT-MIN operation then takes $O(\lg V)$ time. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. (As noted in Section 21.2, you don't even need to call BUILD-MIN-HEAP.) Each DECREASE-KEY operation takes $O(\lg V)$ time, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E) \lg V)$, which is $O(E \lg V)$ in the typical case that $|E| = \Omega(V)$. This running time improves upon the straightforward $O(V^2)$-time implementation if $E = o(V^2/\lg V)$.

By implementing the min-priority queue with a Fibonacci heap (see page 478), you can improve the running time to $O(V \lg V + E)$. 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 20.2) and Prim's algorithm for computing minimum spanning trees (see Section 21.2). 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

### *22.3-1*

Run Dijkstra's algorithm on the directed graph of Figure 22.2, first using vertex $s$ as the source and then using vertex $z$ as the source. In the style of Figure 22.6, show the $d$ and $\pi$ values and the vertices in set $S$ after each iteration of the **while** loop.

### *22.3-2*

Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces an incorrect answer. Why doesn't the proof of Theorem 22.6 go through when negative-weight edges are allowed?

### *22.3-3*

Suppose that you change line 6 of Dijkstra's algorithm to read

6 **while** $|Q| > 1$

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