Tarjan [98] give an algorithm that runs in $O(E \lg^* V)$ time. Gabow, Galil, Spencer, and Tarjan [102] improved this algorithm to run in $O(E \lg \lg^* V)$ time. Chazelle [53] gives an algorithm that runs in $O(E\,\hat{\alpha}(E, V))$ time, where $\hat{\alpha}(E, V)$ is the functional inverse of Ackermann's function. (See the chapter notes for Chapter 21 for a brief discussion of Ackermann's function and its inverse.) Unlike previous minimum-spanning-tree algorithms, Chazelle's algorithm does not follow the greedy method.

A related problem is ***spanning tree verification***, in which we are given a graph $G = (V, E)$ and a tree $T \subseteq E$, and we wish to determine whether T is a minimum spanning tree of G. King [177] gives a linear-time algorithm for spanning tree verification, building on earlier work of Komlós [188] and Dixon, Rauch, and Tarjan [77].

The above algorithms are all deterministic and fall into the comparison-based model described in Chapter 8. Karger, Klein, and Tarjan [169] give a randomized minimum-spanning-tree algorithm that runs in $O(V + E)$ expected time. This algorithm uses recursion in a manner similar to the linear-time selection algorithm in Section 9.3: a recursive call on an auxiliary problem identifies a subset of the edges E' that cannot be in any minimum spanning tree. Another recursive call on $E - E'$ then finds the minimum spanning tree. The algorithm also uses ideas from Boruvka's algorithm and King's algorithm for spanning tree verification.

Fredman and Willard [100] showed how to find a minimum spanning tree in $O(V + E)$ time using a deterministic algorithm that is not comparison based. Their algorithm assumes that the data are b-bit integers and that the computer memory consists of addressable b-bit words.

Chapter 24: Single-Source Shortest Paths

Overview

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 25, 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

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

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 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 inter-section 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, loss, or any other quantity that accumulates linearly along a path and that one wishes to minimize.

The breadth-first-search algorithm from Section 22.2 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 22.2 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 each 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 each 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. Although this problem can be solved by running a single-source algorithm once from each vertex, it can usually be solved faster. Additionally, its structure is of interest in its own right. Chapter 25 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 26 also relies on this property.) This optimal-substructure property is a hallmark of the applicability of both dynamic programming (Chapter 15) and the greedy method (Chapter 16). Dijkstra's algorithm, which we shall see in Section 24.3, is a greedy algorithm, and the Floyd-Warshall algorithm, which finds shortest paths between all pairs of vertices (see

Chapter 25), is a dynamic-programming algorithm. The following lemma states the optimal-substructure property of shortest paths more precisely.

Lemma 24.1: (Subpaths of shortest paths are shortest paths)

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 $1 \leq i \leq j \leq k$, 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 If we decompose path p into $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, then we have that $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ 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 assumption that p is a shortest path from v_1 to v_k.

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 24.1 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 $\langle s, a \rangle$), $\delta(s, a) = w(s, a) = 3$. 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 $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

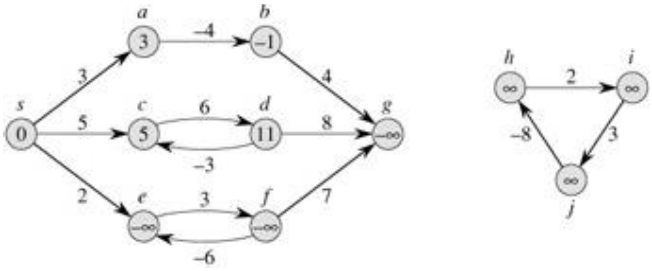

Figure 24.1: Negative edge weights in a directed graph. Shown within each vertex is its shortest-path weight from source s. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path 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 ∞, 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.

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. We 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, we can repeatedly remove these cycles from the path until we have a shortest path that is cycle-free. Therefore, without loss of generality we can assume that when we are finding shortest paths, they have no cycles. Since any acyclic path in a graph $G = (V, E)$ contains at most $|V|$ distinct vertices, it also contains at most $|V| - 1$ edges. Thus, we can restrict our attention to shortest paths of at most $|V| - 1$ edges.

Representing shortest paths

We often wish to compute not only shortest-path weights, but the vertices on shortest paths as well. The representation we use for shortest paths is similar to the one we used for breadth-first trees in Section 22.2. 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 22.2 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_π 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_π is the set of edges induced by the π values for vertices in V_π:

$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$.

We shall prove that the π values produced by the algorithms in this chapter have the property that at termination G_π 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 22.2, 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 24.2 shows a weighted, directed graph and two shortest-paths trees with the same root.

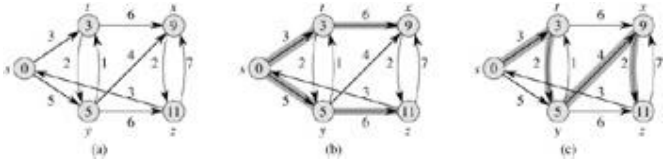

Figure 24.2: (a) A weighted, directed graph with shortest-path weights from source s. (b) The shaded edges form a shortest-paths tree rooted at the source s. (c) Another shortest-paths tree with the same root.

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 $\Theta(V)$-time procedure.

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

After initialization, $\pi[v] = \text{NIL}$ for all $v \in V$, $d[s] = 0$, 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 esti-mate $d[v]$ and update v's predecessor field $\pi[v]$. The following code performs a relaxation step on edge (u, v).

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

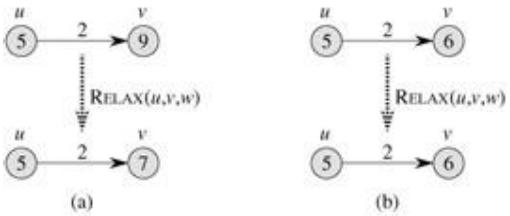

Figure 24.3: Relaxation of an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex is shown within the vertex. (a) Because $d[v] > d[u] + w(u, v)$ prior to relaxation, the value of $d[v]$ decreases. (b) Here, $d[v] \leq d[u] + w(u, v)$ before the relaxation step, and so $d[v]$ 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 shortestpath 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 many times.

[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 $d[v] \leq d[u] + w(u, v)$, which, by the triangle inequality (Lemma 24.10), must be satisfied if $d[u] = \delta(s, u)$ and $d[v] = \delta(s, v)$. That is, if $d[v] \leq d[u] + w(u, v)$, there is no "pressure" to satisfy this constraint, so the constraint is "relaxed."

Properties of shortest paths and relaxation

To prove the algorithms in this chapter correct, we shall appeal to several properties of shortest paths and relaxation. We state these properties here, and Section 24.5 proves them formally. For your reference, each property stated here includes the appropriate lemma or corollary number from Section 24.5. The latter five of these properties, which refer to shortest-path estimates or the predecessor subgraph, implicitly 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 24.10)

- For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 24.11)

- We always have $d[v] \geq \delta(s, v)$ for all vertices $v \in V$, and once $d[v]$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)

- If there is no path from s to v, then we always have $d[v] = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14)

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

Path-relaxation property (Lemma 24.15)

- 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 $d[v_k] = \delta(s, v_k)$. 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 24.17)

- Once $d[v] = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Chapter outline

Section 24.1 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 remarkable in its simplicity, and it has the further benefit of detecting whether a negative-weight cycle is reachable from the source. Section 24.2 gives a linear-time algorithm for computing shortest paths from a single source in a directed acyclic graph. Section 24.3 covers Dijkstra's algorithm, which has a lower running time than the Bellman-Ford algorithm but requires the edge weights to be nonnegative. Section 24.4 shows how the Bellman-Ford algorithm can be used to solve a special case of "linear programming." Finally, Section 24.5 proves the properties of shortest paths and relaxation stated above.

We 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 $a + (-\infty) = (-\infty) + a = -\infty$.

All algorithms in this chapter assume that the directed graph G is stored in the adjacency-list representation. Additionally, stored with each edge is its weight, so that as we traverse each adjacency list, we can determine the edge weights in $O(1)$ time per edge.

24.1 The Bellman-Ford algorithm

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

The algorithm uses 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 ← 1 to |V[G]| - 1
3       do for each edge (u, v) ∈ E[G]
4             do RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

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

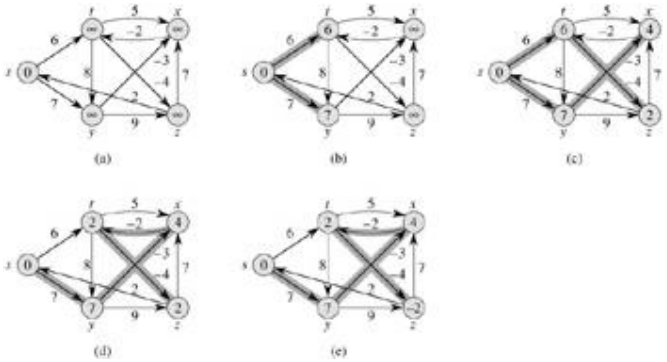

Figure 24.4: The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order (t, x), (t, y),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

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 $\Theta(E)$ time, and the **for** loop of lines 5-7 takes $O(E)$ time.

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

Lemma 24.2

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, after the $|V| - 1$ iterations of the **for** loop of lines 2-4 of BELLMAN-FORD, we have $d[v] = \delta(s, v)$ for all vertices v that are reachable from s.

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

Corollary 24.3

Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \rightarrow \mathbf{R}$. 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 left as Exercise 24.1-2.

Theorem 24.4: (Correctness of the Bellman-Ford algorithm)

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow \mathbf{R}$. 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 24.2 proves this claim. If v is not reachable from s, then the claim follows from the no-path property. Thus, the claim is proven. The predecessor-subgraph property, along with the claim, implies that G_π is a shortest-paths tree. Now we use the claim to show that BELLMAN-FORD returns TRUE. At termination, we have for all edges $(u, v) \in E$, and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. It therefore returns TRUE.

$d[v] = \delta(s, v)$
$\leq \delta(s, u) + w(u, v)$ (by the triangle inequality)
$= d[u] + w(u, v)$,

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

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

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

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

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

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

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

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

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

Exercises 24.1-1

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

Exercise 24.1-2

Prove Corollary 24.3.

Exercise 24.1-3

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.

Exercise 24.1-4

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.

Exercise 24.1-5: ⋆

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

Exercise 24.1-6: ⋆

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

24.2 Single-source shortest paths in directed acyclic graphs

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

The algorithm starts by topologically sorting the dag (see Section 22.4) to impose a linear ordering on the vertices. If 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 we process each vertex, we relax each edge that leaves the vertex.

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

Figure 24.5 shows the execution of this algorithm.

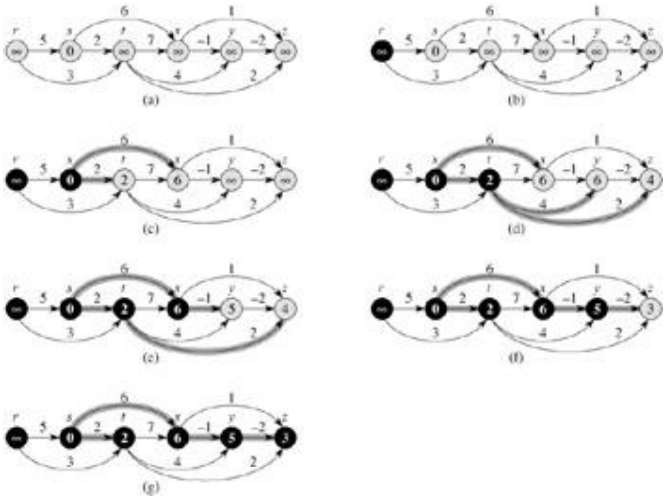

Figure 24.5: The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values are shown within the vertices, and shaded edges indicate the π values. (a) The situation before the first iteration 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 u in that iteration. The values shown in part (g) are the final values.

The running time of this algorithm is easy to analyze. As shown in Section 22.4, the topological sort of line 1 can be performed in $\Theta(V + E)$ time. The call of INITIALIZE-SINGLE-SOURCE in line 2 takes $\Theta(V)$ time. There is one iteration per vertex in the **for** loop of lines 3-5. For each vertex, the edges that leave the vertex are each examined exactly once. Thus, there are a total of $|E|$ iterations of the inner **for** loop of lines 4-5. (We have used an aggregate analysis here.) Because each iteration of the inner **for** loop takes $\Theta(1)$ time, the total running time is $\Theta(V + E)$, which is linear in the size of an adjacency-list representation of the graph.

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

Theorem 24.5

If a weighted, directed graph $G = (V, E)$ has source vertex s and no cycles, then at the termination of the DAG-SHORTEST-PATHS procedure, $d[v] = \delta(s, v)$ 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 the no-path property. Now, suppose that v is

reachable from s, so that there is a shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$. Because we process the vertices in topologically sorted order, the edges on p are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. The path-relaxation property implies that $d[v_i] = \delta(s, v_i)$ at termination for $i = 0, 1, \ldots, k$. Finally, by the predecessor-subgraph property, G_π is a shortest-paths tree.

An interesting application of this algorithm arises in determining critical paths in ***PERT chart***[2] analysis. Edges represent jobs to be performed, and edge weights represent the times required to perform particular jobs. If edge (u, v) enters vertex v and edge (v, x) leaves v, then job (u, v) must be performed 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, with the modification that we replace "∞" by "$-\infty$" in line 2 of INITIALIZE-SINGLE-SOURCE and ">" by "<" in the RELAX procedure.

Exercises 24.2-1

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

Exercise 24.2-2

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

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

Show that the procedure would remain correct.

Exercise 24.2-3

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.

Exercise 24.2-4

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

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

24.3 Dijkstra's algorithm

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

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

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

Dijkstra's algorithm relaxes edges as shown in Figure 24.6. Line 1 performs the usual initialization of d and π values, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 4-8. Line 3 initializes the min-priority queue Q to contain all the vertices in V; since $S = \emptyset$ at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4-8, a vertex u is extracted from $Q = V - S$ and added to set S, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 7-8 relax each edge (u, v) leaving u, thus updating the estimate $d[v]$ and the predecessor $\pi[v]$ 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 added to S exactly once, so that the **while** loop of lines 4-8 iterates exactly $|V|$ times.

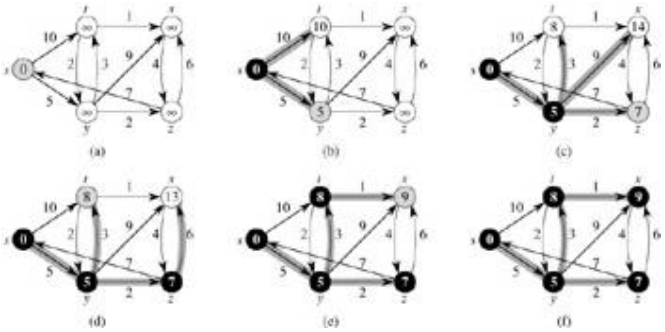

Figure 24.6: The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. (a) The situation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

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy. Greedy strategies are presented in detail in Chapter 16, 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 added to set S, we have $d[u] = \delta(s, u)$.

Theorem 24.6: (Correctness of Dijkstra's algorithm)

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

Proof We use the following loop invariant:

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

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

- **Initialization:** Initially, $S = \emptyset$, and so the invariant is trivially true.
- **Maintenance:** We wish to show that in each iteration, $d[u] = \delta(s, u)$ for the vertex added to set S. For the purpose of contradiction, let u be the first vertex for which $d[u] \neq \delta(s, u)$ when it is added to set S. We shall focus our attention on the situation at the beginning of the iteration of the **while** loop in which u is added to S and derive the contradiction that $d[u] = \delta(s, u)$ at that time by examining a shortest path from s to u. We must have $u \neq s$ because s is the first vertex added to set S and $d[s] = \delta(s, s) = 0$ at that time. Because $u \neq s$, we also have that $S \neq \emptyset$ just before u is added to S. There must be some path from s to u, for otherwise $d[u] = \delta(s, u) = \infty$ by the no-path property, which would violate our assumption that $d[u] \neq \delta(s, u)$. Because there is at least one path, there is a shortest path p from s to u. Prior to adding u to S, path p

connects a vertex in S, namely s, to a vertex in $V - S$, namely u. Let us consider the first vertex y along p such that $y \in V - S$, and let $x \in S$ be y's predecessor. Thus, as shown in Figure 24.7, path p can be decomposed as $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$. (Either of paths p_1 or p_2 may have no edges.)

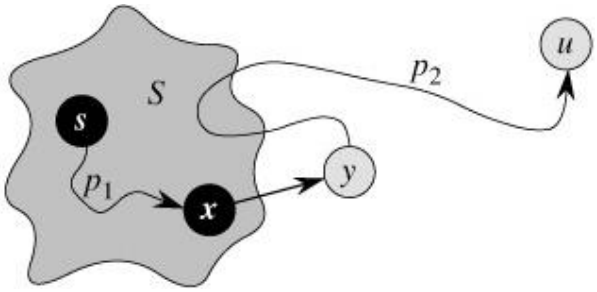

Figure 24.7: The proof of Theorem 24.6. Set S is nonempty just before vertex u is added to it. A shortest path p from source s to vertex u can be decomposed into $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, where y is the first vertex on the path that is not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

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

We can now obtain a contradiction to prove that $d[u] = \delta(s, u)$. 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 $\delta(s, y) \leq \delta(s, u)$, and thus

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

But because both vertices u and y were in $V - S$ when u was chosen in line 5, we have $d[u] \leq d[y]$. Thus, the two inequalities in (24.2) 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 $d[u] = \delta(s, u)$ when u is added to S, and that this equality is maintained at all times thereafter.

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

Corollary 24.7

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

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

Analysis

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

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

If the graph is sufficiently sparse-in particular, $E = o(V^2/\lg V)$-it is practical to implement the min-priority queue with a binary min-heap. (As discussed in Section 6.5, an important implementation detail is that vertices and corresponding heap elements must maintain handles to each other.) Each EXTRACT-MIN operation then takes time $O(\lg V)$. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. Each DECREASE-KEY operation takes time $O(\lg V)$, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E) \lg V)$, which is $O(E \lg V)$ if all vertices are reachable from the source. This running time is an improvement over the straightforward $O(V^2)$-time implementation if $E = o(V^2/\lg V)$.

We can in fact achieve a running time of $O(V \lg V + E)$ by implementing the min-priority queue with a Fibonacci heap (see Chapter 20). 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 in Dijkstra's algorithm there are typically 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 than with binary heaps.

Dijkstra's algorithm bears some similarity to both breadth-first search (see Section 22.2) and Prim's algorithm for computing minimum spanning trees (see Section 23.2). It is like breadth-first search in that set S corresponds to the set of black vertices in a breadth-first search; just as vertices in S have their final shortest-path weights, so do black vertices in a breadth-first

search have their correct breadth-first distances. Dijkstra's algorithm is like Prim's algorithm in that both algorithms use a min-priority queue to find the "lightest" vertex outside a given set (the set S in Dijkstra's algorithm and the tree being grown in Prim's algorithm), add this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

Exercises 24.3-1

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

Exercises 24.3-2

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

Exercises 24.3-3

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

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

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

Exercises 24.3-4

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.

Exercises 24.3-5

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \{1, 2, ..., W\}$ for some positive integer W, and assume that no two vertices have the same shortest-path weights from source vertex s. Now suppose that we define an unweighted, directed graph $G' = (V \cup V', E')$ by replacing each edge $(u, v) \in E$ with $w(u, v)$ unit-weight edges in series. How many vertices does G' have? Now suppose that we run a breadth-first search on G'. Show that the order in which vertices in V are colored black in the breadth-first search of G' is the same as the order in which the vertices of V are extracted from the priority queue in line 5 of DIJKSTRA when run on G.

Exercises 24.3-6

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

Exercises 24.3-7

Modify your algorithm from Exercise 24.3-6 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?)

Exercises 24.3-8

Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

24.4 Difference constraints and shortest paths

Chapter 29 studies the general linear-programming problem, in which 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 $Ax \leq b$.

Although the simplex algorithm, which is the focus of Chapter 29, does not always run in time polynomial in the size of its input, there are other linear-programming algorithms that do run in polynomial time. There are several reasons that 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 24.4-4) and the maximum-flow problem (Exercise 26.1-8).

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 $Ax \leq b$, 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 $Ax \leq b$ are a set of m ***difference constraints*** involving n unknowns, in which each constraint is a simple linear inequality of the form

$x_j - x_i \leq b_k$,

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

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

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

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

$$\begin{aligned}
&(24.3)\ x_1 - x_2 \leq 0, \\
&(24.4)\ x_1 - x_5 \leq -1, \\
&(24.5)\ x_2 - x_5 \leq 1, \\
&(24.6)\ x_3 - x_1 \leq 5, \\
&(24.7)\ x_4 - x_1 \leq 4, \\
&(24.8)\ x_4 - x_3 \leq -1, \\
&(24.9)\ x_5 - x_3 \leq -3, \\
&(24.10)\ x_5 - x_4 \leq -3.
\end{aligned}$$

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 24.8

Let $x = (x_1, x_2, ..., x_n)$ be a solution to a system $Ax \leq b$ of difference constraints, and let d be any constant. Then $x + d = (x_1 + d, x_2 + d,..., x_n + d)$ is a solution to $Ax \leq b$ 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 $Ax \leq b$, 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 there must be at least a certain amount of time, or at most a certain amount of time, between two events. Perhaps the events are jobs to be performed during the assembly of a product. If we apply an adhesive that takes 2 hours to set at time x_1 and we have to wait until it sets to install a part at time x_2, then we have the constraint that $x_2 \geq x_1 + 2$ or, equivalently, that $x_1 - x_2 \leq -2$. Alternatively, we might require that the part be installed after the adhesive has been applied but no later than the time that the adhesive has set halfway. In this case, we get the pair of constraints $x_2 \geq x_1$ and $x_2 \leq x_1 + 1$ or, equivalently, $x_1 - x_2 \leq 0$ and $x_2 - x_1 \leq 1$.

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 $Ax \leq b$ of difference constraints, the $m \times n$ linear-programming matrix A can be viewed as the transpose of an incidence matrix (see Exercise 22.1-7) for a graph with n vertices and m edges. Each vertex v_i in the graph, for $i = 1, 2,..., n$, 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 $Ax \leq b$ of difference constraints, the corresponding ***constraint graph*** is a weighted, directed graph $G = (V, E)$, where

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

and

$E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ is a constraint}\} \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3),..., (v_0, v_n)\}$.

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 $x_j - x_i \leq b_k$ 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 24.8 shows the constraint graph for the system (24.3)-(24.10) of difference constraints.

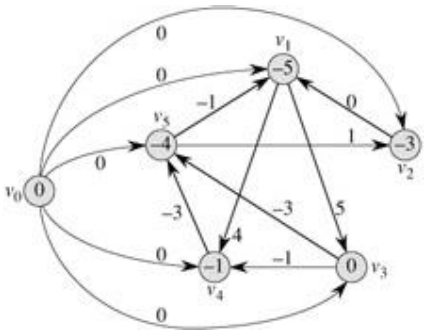

Figure 24.8: The constraint graph corresponding to the system (24.3)-(24.10) of difference constraints. The value of $\delta(v_0, v_i)$ is shown in each vertex v_i. A feasible solution to the system is $x = (-5, -3, 0, -1, -4)$.

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

Theorem 24.9

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

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

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 (24.11) gives a feasible solution. Consider any edge $(v_i, v_j) \in E$. By the triangle inequality, $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$ or, equivalently, $\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$. 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:

$$\begin{aligned}
x_2 - x_1 &\leq w(v_1, v_2), \\
x_3 - x_2 &\leq w(v_2, v_3), \\
&\vdots \\
x_k - x_{k-1} &\leq w(v_{k-1}, v_k), \\
x_1 - x_k &\leq w(v_k, v_1).
\end{aligned}$$

Suppose that there is a solution for x satisfying each of these k inequalities. This solution must also satisfy the inequality that results when we sum the k inequalities 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 $0 \le w(c)$. But since c is a negative-weight cycle, $w(c) < 0$, and we obtain the contradiction that $0 \le w(c) < 0$.

Solving systems of difference constraints

Theorem 24.9 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 24.8, for example, the shortest-path weights provide the feasible solution $x = (-5, -3, 0, -1, -4)$, and by Lemma 24.8, $x = (d - 5, d - 3, d, d - 1, d - 4)$ 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 24.4-5 asks you to modify the algorithm to run in $O(nm)$ time, even if m is much less than n.

Exercises 24.4-1

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

$x_1 - x_2 \le 1$,
$x_1 - x_4 \le -4$,
$x_2 - x_3 \le 2$,
$x_2 - x_5 \le 7$,
$x_2 - x_6 \le 5$,
$x_3 - x_6 \le 10$
,
$x_4 - x_2 \le 2$,
$x_5 - x_1 \le -1$
$x_5 - x_4 \le 3$,
$x_6 - x_3 \le -8$.

Exercises 24.4-2

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

$x_1 - x_2 \le 4$,
$x_1 - x_5 \le 5$,
$x_2 - x_4 \le -6$,
$x_3 - x_2 \le 1$,
$x_4 - x_1 \le 3$,
$x_4 - x_3 \le 5$,
$x_4 - x_5 \le 10$
,
$x_5 - x_3 \le -4$,
$x_5 - x_4 \le -8$.

Exercises 24.4-3

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

Exercises 24.4-4

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

Exercises 24.4-5

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 $O(nm)$.

Exercises 24.4-6

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.

Exercises 24.4-7

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.

Exercises 24.4-8: ⋆

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.

Exercises 24.4-9: ⋆

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.

Exercises 24.4-10

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 to adapt the Bellman-Ford algorithm to solve this variety of constraint system.

Exercises 24.4-11

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

Exercises 24.4-12: ⋆

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

24.5 Proofs of shortest-paths properties

Throughout this chapter, our correctness arguments have relied on the triangle inequality, upper-bound property, no-path property, convergence property, path-relaxation property, and predecessor-subgraph property. We stated these properties without proof at the beginning of this chapter. In this section, we prove them.

The triangle inequality

In studying breadth-first search (Section 22.2), we proved as Lemma 22.1 a simple property of shortest distances in unweighted graphs. The triangle inequality generalizes the property to weighted graphs.

Lemma 24.10: (Triangle inequality)

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

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

Proof Suppose that there is a shortest path p from source s to vertex v. Then p 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).

Exercise 24.5-3 asks you to handle the case in which there is no shortest path from s to v.

Effects of relaxation on shortest-path estimates

The next group of lemmas describes how shortest-path estimates are affected when we execute a sequence of relaxation steps on the edges of a weighted, directed graph that has been initialized by INITIALIZE-SINGLE-SOURCE.

Lemma 24.11: (Upper-bound property)

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, $d[v] \geq \delta(s, v)$ 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 We prove the invariant $d[v] \geq \delta(s, v)$ for all vertices $v \in V$ by induction over the number of relaxation steps.

For the basis, $d[v] \geq \delta(s, v)$ is certainly true after initialization, since $d[s] = 0 \geq \delta(s, s)$ (note that $\delta(s, s)$ is $-\infty$ if s is on a negative-weight cycle and 0 otherwise) and $d[v] = \infty$ implies $d[v] \geq \delta(s, v)$ for all $v \in V - \{s\}$.

For the inductive step, consider the relaxation of an edge (u, v). By the inductive hypothesis, $d[x] \geq \delta(s, x)$ for all $x \in V$ prior to the relaxation. The only d value that may change is $d[v]$. If it changes, we have and so the invariant is maintained.

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

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 $d[v] \geq \delta(s, v)$, and it cannot increase because relaxation steps do not increase d values.

Corollary 24.12: (No-path property)

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 $d[v] = \delta(s, v) = \infty$, and this equality is maintained as an invariant over any sequence of relaxation steps on the edges of G.

Proof By the upper-bound property, we always have $\infty = \delta(s, v) \leq d[v]$, and thus $d[v] = \infty = \delta(s, v)$.

Lemma 24.13

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, then neither $d[u]$ nor $d[v]$ changes, and so $d[v] \leq d[u] + w(u, v)$ afterward.

Lemma 24.14: (Convergence property)

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 $s \rightsquigarrow u \rightarrow v$ 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 the upper-bound property, 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

$$\begin{aligned} d[v] &\le d[u] + w(u, v) \quad \text{(by Lemma 24.13)} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) \quad \text{(by Lemma 24.1)}. \end{aligned}$$

By the upper-bound property, $d[v] \ge \delta(s, v)$, from which we conclude that $d[v] = \delta(s, v)$, and this equality is maintained thereafter.

Lemma 24.15: (Path-relaxation property)

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$, and let $s \in V$ be a source vertex. Consider any shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$ from $s = v_0$ to v_k. If G is initialized by INITIALIZE-SINGLE-SOURCE(G, s) and then a sequence of relaxation steps occurs that includes, in order, relaxations of edges (v_0, v_1), (v_1, v_2),..., (v_{k-1}, v_k), then $d[v_k] = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of the edges of p.

Proof We show by induction that after the ith edge of path p is relaxed, we have $d[v_i] = \delta(s, v_i)$. For the basis, $i = 0$, and before any edges of p have been relaxed, we have from the initialization that $d[v_0] = d[s] = 0 = \delta(s, s)$. By the upper-bound property, the value of $d[s]$ never changes after initialization.

For the inductive step, we assume that $d[v_{i-1}] = \delta(s, v_{i-1})$, and we examine the relaxation of edge (v_{i-1}, v_i). By the convergence property, after this relaxation, we have $d[v_i] = \delta(s, v_i)$, and this equality is maintained at all times thereafter.

Relaxation and shortest-paths trees

We now show that once a sequence of relaxations has caused the shortest-path estimates to converge to shortest-path weights, the predecessor subgraph G_π induced by the resulting π values is a shortest-paths tree for G. We start with the following lemma, which shows that the predecessor subgraph always forms a rooted tree whose root is the source.

Lemma 24.16

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 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 the upper-bound property, 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 $d[v_i] \leftarrow d[v_{i-1}] + w(v_{i-1}, v_i)$. If $d[v_{i-1}]$ changed since then, it decreased. Therefore, just before the call RELAX(v_{k-1}, v_k, w), we have

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

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 (24.12), 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 equality implies

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

Thus, the sum of weights around the cycle c is negative, which provides the desired contradiction.

We have now proven that G_π is a directed, acyclic graph. To show that it forms a rooted tree with root s, it suffices (see Exercise B.5-2) 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 24.5-6.

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 $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$ and p_2, which can be decomposed into $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$, where $x \neq y$. (See Figure 24.9.) 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.

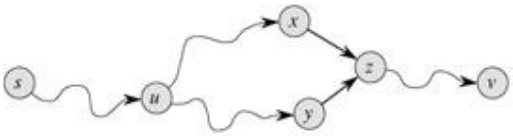

Figure 24.9: Showing that a path in G_π from source s to vertex v is unique. If there are two paths p_1 ($s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$) and p_2 ($s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$), where $x \neq y$, then $\pi[z] = x$ and $\pi[z] = y$, 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 24.17: (Predecessor-subgraph property)

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 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 given on page 584 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 24.16.

It remains, therefore, to prove the last property of shortest-paths trees: for each vertex $v \in V_\pi$, the unique simple path $s \overset{p}{\rightsquigarrow} v$ 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

$$\begin{aligned} w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) && \text{(because the sum telescopes)} \\ &= \delta(s, v_k) && \text{(because } \delta(s, v_0) = \delta(s, s) = 0\text{)} . \end{aligned}$$

Thus, $w(p) \leq \delta(s, v_k)$. 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 24.5-1

Give two shortest-paths trees for the directed graph of Figure 24.2 other than the two shown.

Exercises 24.5-2

Give an example of a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbf{R}$ 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).

Exercises 24.5-3

Embellish the proof of Lemma 24.10 to handle cases in which shortest-path weights are ∞ or $-\infty$.

Exercises 24.5-4

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.

Exercises 24.5-5

Let $G = (V, E)$ be a weighted, directed graph with no negative-weight edges. Let $s \in V$ be the source vertex, and suppose that we allow $\pi[v]$ to be the predecessor of v on *any* 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 24.16, such an assignment cannot be produced by a sequence of relaxation steps.)

Exercises 24.5-6

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.

Exercises 24.5-7

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 exists a sequence of $|V| - 1$ relaxation steps that produces $d[v] = \delta(s, v)$ for all $v \in V$.

Exercises 24.5-8

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.

Problems 24-1: 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 $E_b = \{(v_i, v_j) \in E : i > j\}$. (Assume that G contains no self-loops, so that every edge is in either E_f or E_b.) 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 $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1 \rangle$.

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. Does this scheme improve the asymptotic running time of the Bellman-Ford algorithm?

Problems 24-2: 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 $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \ldots, x_{\pi(d)} < y_d$.

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 $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k} \rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1, 2, \ldots, k - 1$. Express the running time of your algorithm in terms of n and d.

Problems 24-3: 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 46.4 Indian rupees, 1 Indian rupee buys 2.5 Japanese yen, and 1 Japanese yen buys 0.0091 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $46.4 \times 2.5 \times 0.0091 = 1.0556$ U.S. dollars, thus turning a profit of 5.56 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 $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that

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

   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.

Problems 24-4: 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. We assume that all vertices are reachable from the source.

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 $|E| \geq |V| - 1$, and we shall see that computing δ_i from δ_{i-1} takes $O(E)$ time, so that the entire algorithm takes $O(k E) = O(E \lg W)$ time.

a. Suppose that for all vertices $v \in V$, we have $\delta(s, v) \leq |E|$. 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 focus on computing δ_i from δ_{i-1}.

c. Prove that for $i = 2, 3, \ldots, k$, we have 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 $(u, v) \in E$,

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

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

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

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

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

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.

Problems 24-5: Karp's minimum mean-weight cycle algorithm

Let $G = (V, E)$ be a directed graph with weight function $w : E \to \mathbf{R}$, 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 $\mu^* = 0$ and 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 on each minimum mean-weight cycle there exists a vertex v 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 a 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 fact 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 $O(VE)$-time algorithm to compute μ^*.

Problems 24-6: Bitonic shortest paths

A sequence is ***bitonic*** if it monotonically increases and then monotonically decreases, or if it can be circularly shifted to monotonically increase and then monotonically decrease. For example the sequences □1, 4, 6, 8, 3, -2□, □9, 2, -4, -10, -5□, and □1, 2, 3, 4□ are bitonic, but □1, 3, 12, 4, 2, 10□ is not bitonic. (See Chapter 27 for a discussion of bitonic sorters, and see Problem 15-1 for the bitonic euclidean traveling-salesman problem.)

Suppose that we are given a directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbf{R}$, and we wish to find single-source shortest paths from a source vertex s. We are given one additional piece of information: for each vertex $v \in V$, the weights of the edges along any shortest path from s to v form a bitonic sequence.

Give the most efficient algorithm you can to solve this problem, and analyze its running time.

Chapter notes

Dijkstra's algorithm [75] appeared in 1959, but it contained no mention of a priority queue. The Bellman-Ford algorithm is based on separate algorithms by Bellman [35] and Ford [93]. Bellman describes the relation of shortest paths to difference constraints. Lawler [196] describes the linear-time algorithm for shortest paths in a dag, which he considers part of the folklore.

When edge weights are relatively small nonnegative integers, more efficient algorithms can be used to solve the single-source shortest-paths problem. The sequence of values returned by the EXTRACT-MIN calls in Dijkstra's algorithm is monotonically increasing over time. As discussed in the chapter notes for Chapter 6, in this case there are several data structures that can implement the various priority-queue operations more efficiently than a binary heap or a Fibonacci heap. Ahuja, Mehlhorn, Orlin, and Tarjan [8] 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. The best bounds are by Thorup [299], who gives an algorithm that runs in $O(E \lg \lg V)$ time, and by Raman, who gives an algorithm that runs in $O(E + V \min\{(\lg V)^{1/3+\epsilon}, (\lg W)^{1/4+\epsilon}\})$ time. These two algorithms use an amount of space that depends on the word size of the underlying machine. Although the amount of space used can be unbounded in the size of the input, it can be reduced to be linear in the size of the input using randomized hashing.

For undirected graphs with integer weights, Thorup [298] gives an $O(V + E)$-time algorithm for single-source shortest paths. In contrast to the algorithms mentioned in the previous paragraph, this algorithm is not an implementation of Dijkstra's algorithm, since the sequence of values returned by EXTRACT-MIN calls is not monotonically increasing over time.

For graphs with negative edge weights, an algorithm due to Gabow and Tarjan [104] runs in $O(\sqrt{V}E \lg(VW))$ time, and one by Goldberg [118] runs in $O(\sqrt{V}E \lg W)$ time, where $W = \max_{(u,v)\in E}\{|w(u, v)|\}$.

Cherkassky, Goldberg, and Radzik [57] conducted extensive experiments comparing various shortest-path algorithms.

Chapter 25: All-Pairs Shortest Paths

Overview

In this chapter, we consider the problem of finding shortest paths between all pairs of vertices in a graph. This problem might arise in making a table of distances between all pairs of cities for a road atlas. As in Chapter 24, we are given a weighted, directed graph $G = (V, E)$ with a weight function $w : E \rightarrow \mathbf{R}$ that maps edges to real-valued weights. We wish to find, for every pair of vertices $u, v \in V$, a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges. We typically want the output in tabular form: the entry in u's row and v's column should be the weight of a shortest path from u to v.

We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm $|V|$ times, once for each vertex as the source. If all edge weights are nonnegative, we can use Dijkstra's algorithm. If we use the linear-array implementation of the min-priority queue, the running time is $O(V^3 + VE) = O(V^3)$. The binary min-heap implementation of the min-priority queue yields a running time of $O(VE \lg V)$, which is an improvement if the graph is sparse. Alternatively, we can implement the min-priority queue with a Fibonacci heap, yielding a running time of $O(V^2 \lg V + VE)$.

If negative-weight edges are allowed, Dijkstra's algorithm can no longer be used. Instead, we must run the slower Bellman-Ford algorithm once from each vertex. The resulting running time is $O(V^2E)$, which on a dense graph is $O(V^4)$. In this chapter we shall see how to do better.