relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

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

Chapter outline

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

This chapter does arithmetic with infinities, and so we need some conventions for when ∞ or $-\infty$ appears in an arithmetic expression. We assume that for any real number $a \neq -\infty$, we have $a + \infty = \infty + a = \infty$. Also, to make our proofs hold in the presence of negative-weight cycles, we assume that for any real number $a \neq \infty$, we have $a + (-\infty) = (-\infty) + a = -\infty$.

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 each algorithm traverses an adjacency list, it can find edge weights in $O(1)$ time per edge.

22.1 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 vertex s and weight function $w : E \rightarrow \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether 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 procedure BELLMAN-FORD relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

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

Figure 22.4 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 22.4(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.)

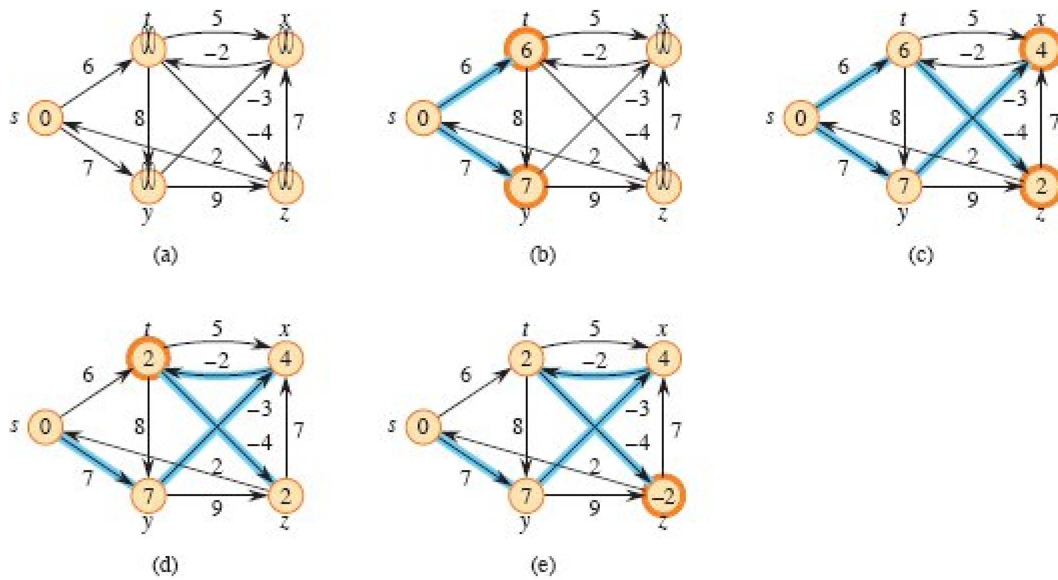

Figure 22.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and blue edges indicate predecessor values: if edge (u, v) is blue, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y). **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. Vertices whose shortest-path estimates and predecessors have changed due to a pass are highlighted in orange. 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 $O(V^2 + VE)$ time when the graph is represented by adjacency lists, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(V + E)$ time (examining $|V|$ adjacency lists to find the $|E|$ edges), and the **for** loop of lines 5–7 takes $O(V + E)$ time. Fewer than $|V| - 1$ passes over the edges sometimes suffice (see Exercise 22.1-3), which is why we say $O(V^2+VE)$ time, rather than $\Theta(V^2+VE)$ time. In the frequent case where $|E| = \Omega(V)$, we can express this running time as $O(VE)$. Exercise 22.1-5 asks you to make the Bellman-Ford algorithm run in $O(VE)$ time even when $|E| = o(V)$.

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 22.2
Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the **for** loop of lines 2–4 of BELLMAN-FORD, $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

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

■

Corollary 22.3
Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \to \mathbb{R}$. Then, for each vertex $v \in V$, there is a path from s to v if and only if BELLMAN-FORD terminates with $v.d < \infty$ when it is run on G.

Proof The proof is left as Exercise 22.1-2.

■

Theorem 22.4 (Correctness of the Bellman-Ford algorithm)
Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source vertex s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof Suppose that graph G contains no negative-weight cycles that are reachable from the source s. We first prove the claim that at termination, $v.d = \delta(s, v)$ for all vertices $v \in V$. If vertex v is reachable from s, then Lemma 22.2 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, for all edges $(u, v) \in E$ we have

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

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

Now, suppose that graph G contains a negative-weight cycle reachable from the source s. Let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$, in which case we have

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 . \qquad (22.1)$$

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

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

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

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

Moreover, by Corollary 22.3, $v_i.d$ is finite for $i = 1, 2, \ldots, k$. Thus,

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

which contradicts inequality (22.1). 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

22.1-1

Run the Bellman-Ford algorithm on the directed graph of Figure 22.4, 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.

22.1-2

Prove Corollary 22.3.

22.1-3

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

22.1-4

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

22.1-5

Suppose that the graph given as input to the Bellman-Ford algorithm is represented with a list of $|E|$ edges, where each edge indicates the

vertices it leaves and enters, along with its weight. Argue that the Bellman-Ford algorithm runs in $O(VE)$ time without the constraint that $|E| = \Omega(V)$. Modify the Bellman-Ford algorithm so that it runs in $O(VE)$ time in all cases when the input graph is represented with adjacency lists.

22.1-6

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

22.1-7

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

22.2 Single-source shortest paths in directed acyclic graphs

In this section, we introduce one further restriction on weighted, directed graphs: they are acyclic. That is, we are concerned with weighted dags. Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist. We'll see that if the edges of a weighted dag $G = (V, E)$ are relaxed according to a topological sort of its vertices, it takes only $\Theta(V + E)$ time to compute shortest paths from a single source.

The algorithm starts by topologically sorting the dag (see Section 20.4) to impose a linear ordering on the vertices. If the dag contains a path from vertex u to vertex v, then u precedes v in the topological sort. The DAG-SHORTEST-PATHS procedure makes just one pass over the vertices in the topologically sorted order. As it processes each vertex, it relaxes each edge that leaves the vertex. Figure 22.5 shows the execution of this algorithm.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
```

```
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 for each vertex u ∈ G.V, taken in topologically sorted order
4     for each vertex v in G.Adj[u]
5         RELAX(u, v, w)
```

Let's analyze the running time of this algorithm. As shown in Section 20.4, the topological sort of line 1 takes $\Theta(V + E)$ time. The call of INITIALIZE-SINGLE-SOURCE in line 2 takes $\Theta(V)$ time. The **for** loop of lines 3–5 makes one iteration per vertex. Altogether, the **for** loop of lines 4–5 relaxes each edge exactly once. (We have used an aggregate analysis here.) Because each iteration of the inner **for** loop takes $\Theta(1)$ time, the total running time is $\Theta(V + E)$, which is linear in the size of an adjacency-list representation of the graph.

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

Theorem 22.5

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

Proof We first show that $v.d = \delta(s, v)$ for all vertices $v \in V$ at termination. If v is not reachable from s, then $v.d = \delta(s, v) = 1$ 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 DAG-SHORTEST-PATHS processes the vertices in topologically sorted order, it relaxes the edges on p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. The path-relaxation property implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 0, 1, \ldots, k$. Finally, by the predecessor-subgraph property, G_π is a shortest-paths tree.

■

A useful application of this algorithm arises in determining critical paths in ***PERT chart***[2] analysis. *A* job consists of several tasks. Each task

takes a certain amount of time, and some tasks must be completed before others can be started. For example, if the job is to build a house, then the foundation must be completed before starting to frame the exterior walls, which must be completed before starting on the roof. Some tasks require more than one other task to be completed before they can be started: before the drywall can be installed over the wall framing, both the electrical system and plumbing must be installed. A dag models the tasks and dependencies. Edges represent tasks, with the weight of an edge indicating the time required to perform the task. Vertices represent "milestones," which are achieved when all the tasks represented by the edges entering the vertex have been completed. If edge (u, v) enters vertex v and edge (v, x) leaves v, then task (u, v) must be completed before task (v, x) is started. A path through this dag represents a sequence of tasks that must be performed in a particular order. A ***critical path*** is a *longest* path through the dag, corresponding to the longest time to perform any sequence of tasks. Thus, the weight of a critical path provides a lower bound on the total time to perform all the tasks, even if as many tasks as possible are performed simultaneously. You can find a critical path by either

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

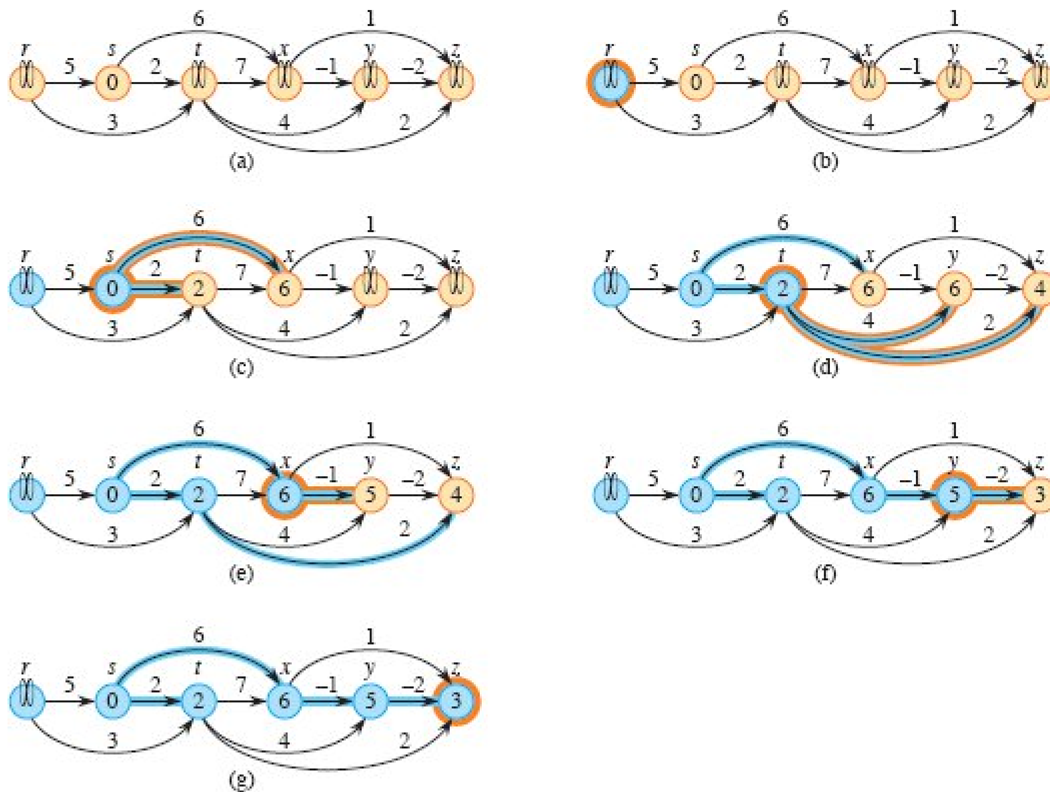

Figure 22.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values appear within the vertices, and blue edges indicate the π values. **(a)** The situation before the first iteration of the **for** loop of lines 3–5. **(b)–(g)** The situation after each iteration of the **for** loop of lines 3–5. Blue vertices have had their outgoing edges relaxed. The vertex highlighted in orange was used as u in that iteration. Each edge highlighted in orange caused a d value to change when it was relaxed in that iteration. The values shown in part (g) are the final values.

Exercises

22.2-1

Show the result of running DAG-SHORTEST-PATHS on the directed acyclic graph of Figure 22.5, using vertex r as the source.

22.2-2

Suppose that you 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 remains correct.

22.2-3

An alternative way to represent a PERT chart looks more like the dag of Figure 20.7 on page 574. Vertices represent tasks and edges represent sequencing constraints, that is, edge (u, v) indicates that task u must be performed before task v. Vertices, not edges, have weights. Modify the DAG-SHORTEST-PATHS procedure so that it finds a longest path in a directed acyclic graph with weighted vertices in linear time.

★ ***22.2-4***

Give an efficient algorithm to count the total number of paths in a directed acyclic graph. The count should include all paths between all pairs of vertices and all paths with 0 edges. Analyze your algorithm.

22.3 Dijkstra's algorithm

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

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

Instead, Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u into S, and relaxes all edges leaving u. The procedure DIJKSTRA replaces the first-in, first-out queue of