Lecture 4

We now discuss some algorithm design strategies. These are heuristic ways to think about designing algorithms for problems. Underlying each approach is a design mindset, which we discuss. In this lecture, we discuss:

- The incremental strategy for algorithm design.

In the incremental strategy, we seek to achieve a goal by progressing a constant number of steps towards the goal at a time. (Note: "step" here is different from what we mean by "step" in our definition of an algorithm.) Typically, a recurrence that characterizes the running-time of an algorithm based on the incremental strategy looks like the following:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c_1 \\ \Theta(n^{c_2}) + T(n - c_3) & \text{otherwise} \end{cases}$$

where c_1, c_2 and c_3 are constants. The term "$T(n - c_3)$" can be seen as that of a subproblem we are left to solve, given that the original input was of size n. Thus, we have shrunk the problem-size down by a constant, c_3.

All three versions of insertion sort from the previous lectures fall in to the incremental strategy. In insertion sort, $c_1 = c_2 = c_3 = 1$; that is, the problem-size decreases by 1 with each sub-problem. Another approach for sorting which adopts the incremental strategy is selection sort, whose pseudo-code is below.

```
SelectionSort(A[1, ..., n])
1 foreach i from 1 to n do
2     m ← IndexOfMin(A[i, ..., n])
3     if i ≠ m then swap A[i] ↔ A[m]
```

where the subroutine IndexOfMin find an index of the minimum value in a (sub-)array $A[i, \ldots, n]$. Thus, similar to insertion sort, at the start of iteration i of the **foreach** loop, $A[1, \ldots, i-1]$ is sorted, but in addition, unlike in insertion sort, each of $A[1, \ldots, i-1]$ contains exactly what A should once it is sorted.

Similar to insertion sort, a recurrence for the worst-case running time looks like $T(n) = \Theta(n) + T(n-1)$, whose solution is $T(n) = \Theta(n^2)$.

The elementary graph algorithms, Breadth First Search (BFS) and Depth First Search (DFS) can also be seen as incremental, because we process one vertex at a time.

22 Elementary Graph Algorithms

This chapter presents methods for representing a graph and for searching a graph. Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph. A graph-searching algorithm can discover much about the structure of a graph. Many algorithms begin by searching their input graph to obtain this structural information. Other graph algorithms are organized as simple elaborations of basic graph-searching algorithms. Techniques for searching a graph are at the heart of the field of graph algorithms.

Section 22.1 discusses the two most common computational representations of graphs: as adjacency lists and as adjacency matrices. Section 22.2 presents a simple graph-searching algorithm called breadth-first search and shows how to create a breadth-first tree. Section 22.3 presents depth-first search and proves some standard results about the order in which depth-first search visits vertices. Section 22.4 provides our first real application of depth-first search: topologically sorting a directed acyclic graph. A second application of depth-first search, finding the strongly connected components of a directed graph, is given in Section 22.5.

22.1 Representations of graphs

There are two standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. Either way is applicable to both directed and undirected graphs. The adjacency-list representation is usually preferred, because it provides a compact way to represent ***sparse*** graphs—those for which $|E|$ is much less than $|V|^2$. Most of the graph algorithms presented in this book assume that an input graph is represented in adjacency-list form. An adjacency-matrix representation may be preferred, however, when the graph is ***dense***—$|E|$ is close to $|V|^2$—or when we need to be able to tell quickly if there is an edge connecting two given vertices. For example, two of the all-pairs shortest-paths algorithms pre-

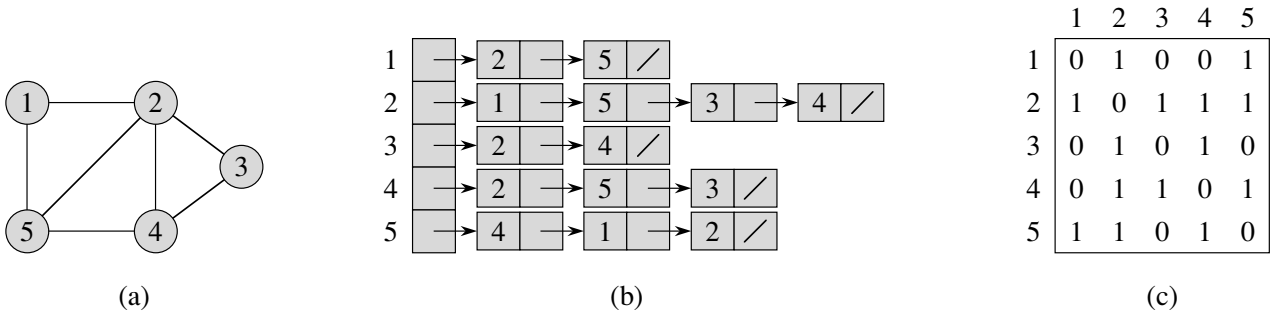

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

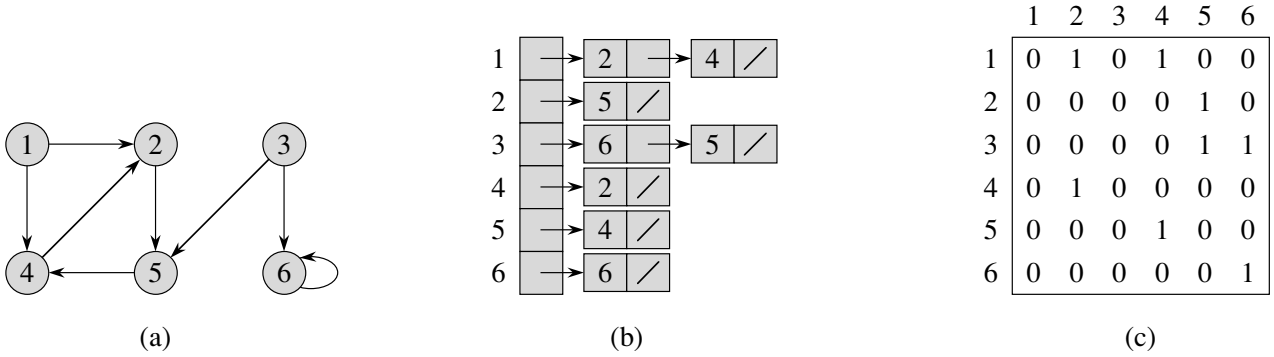

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

sented in Chapter 25 assume that their input graphs are represented by adjacency matrices.

The ***adjacency-list representation*** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G. (Alternatively, it may contain pointers to these vertices.) The vertices in each adjacency list are typically stored in an arbitrary order. Figure 22.1(b) is an adjacency-list representation of the undirected graph in Figure 22.1(a). Similarly, Figure 22.2(b) is an adjacency-list representation of the directed graph in Figure 22.2(a).

If G is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form (u, v) is represented by having v appear in $Adj[u]$. If G is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa.

For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$.

Adjacency lists can readily be adapted to represent ***weighted graphs***, that is, graphs for which each edge has an associated ***weight***, typically given by a ***weight function*** $w : E \to \mathbf{R}$. For example, let $G = (V, E)$ be a weighted graph with weight function w. The weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored with vertex v in u's adjacency list. The adjacency-list representation is quite robust in that it can be modified to support many other graph variants.

A potential disadvantage of the adjacency-list representation is that there is no quicker way to determine if a given edge (u, v) is present in the graph than to search for v in the adjacency list $Adj[u]$. This disadvantage can be remedied by an adjacency-matrix representation of the graph, at the cost of using asymptotically more memory. (See Exercise 22.1-8 for suggestions of variations on adjacency lists that permit faster edge lookup.)

For the ***adjacency-matrix representation*** of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Figures 22.1(c) and 22.2(c) are the adjacency matrices of the undirected and directed graphs in Figures 22.1(a) and 22.2(a), respectively. The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.

Observe the symmetry along the main diagonal of the adjacency matrix in Figure 22.1(c). We define the ***transpose*** of a matrix $A = (a_{ij})$ to be the matrix $A^{\mathrm{T}} = (a_{ij}^{\mathrm{T}})$ given by $a_{ij}^{\mathrm{T}} = a_{ji}$. Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose: $A = A^{\mathrm{T}}$. In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.

Like the adjacency-list representation of a graph, the adjacency-matrix representation can be used for weighted graphs. For example, if $G = (V, E)$ is a weighted graph with edge-weight function w, the weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored as the entry in row u and column v of the adjacency matrix. If an edge does not exist, a NIL value can be stored as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞.

Although the adjacency-list representation is asymptotically at least as efficient as the adjacency-matrix representation, the simplicity of an adjacency matrix may make it preferable when graphs are reasonably small. Moreover, if the graph is unweighted, there is an additional advantage in storage for the adjacency-matrix

representation. Rather than using one word of computer memory for each matrix entry, the adjacency matrix uses only one bit per entry.

22.2 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim's minimum-spanning-tree algorithm (Section 23.2) and Dijkstra's single-source shortest-paths algorithm (Section 24.3) use ideas similar to those in breadth-first search.

Given a graph $G = (V, E)$ and a distinguished ***source*** vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is ***discovered*** the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but

breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the ***predecessor*** or ***parent*** of v in the breadth-first tree. Since a vertex is discovered at most once, it has at most one parent. Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on a path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.

The breadth-first-search procedure BFS below assumes that the input graph $G = (V, E)$ is represented using adjacency lists. It maintains several additional data structures with each vertex in the graph. The color of each vertex $u \in V$ is stored in the variable $color[u]$, and the predecessor of u is stored in the variable $\pi[u]$. If u has no predecessor (for example, if $u = s$ or u has not been discovered), then $\pi[u] = \text{NIL}$. The distance from the source s to vertex u computed by the algorithm is stored in $d[u]$. The algorithm also uses a first-in, first-out queue Q (see Section 10.1) to manage the set of gray vertices.

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                     then color[v] ← GRAY
15                          d[v] ← d[u] + 1
16                          π[v] ← u
17                          ENQUEUE(Q, v)
18          color[u] ← BLACK
```

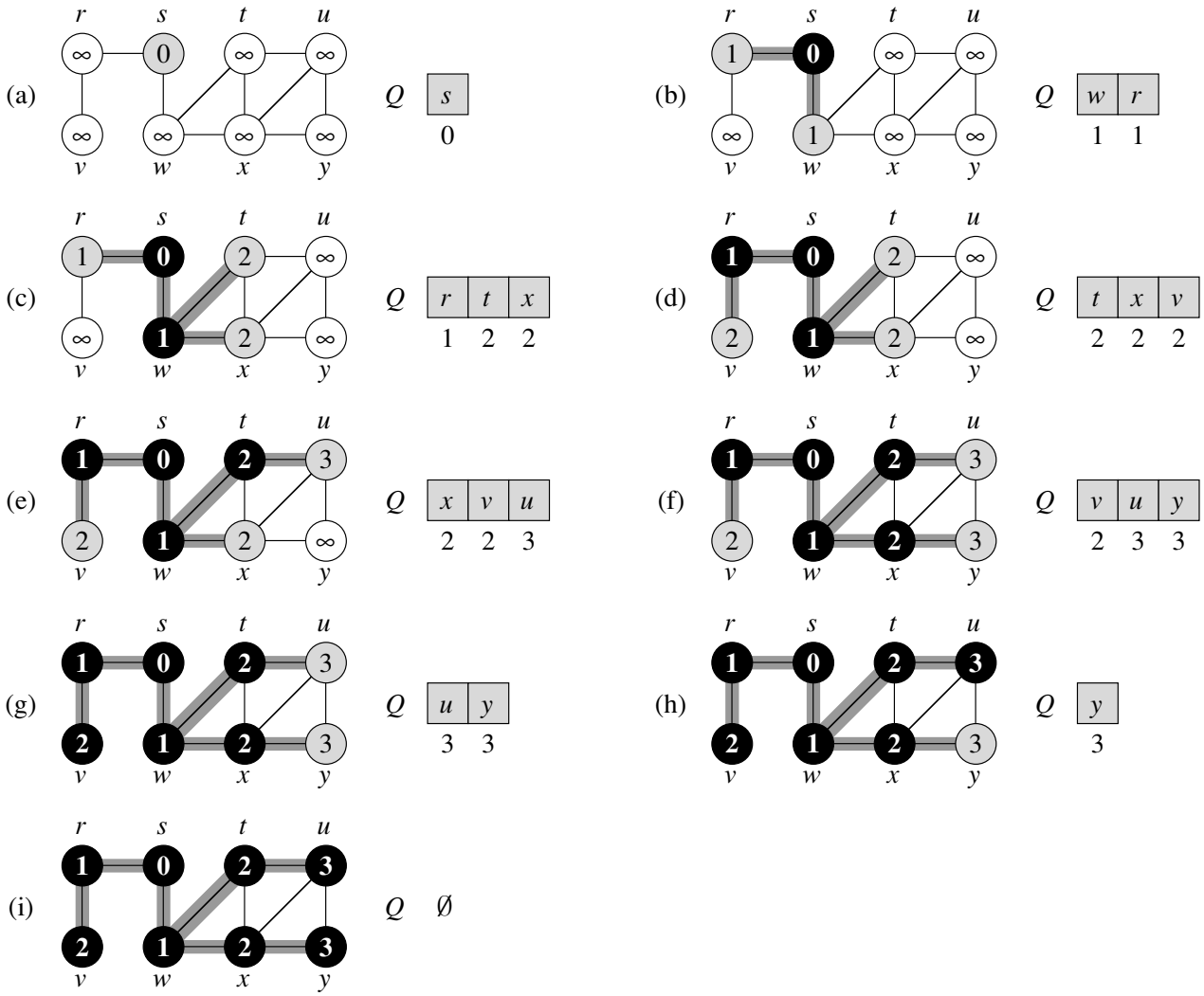

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue.

Figure 22.3 illustrates the progress of BFS on a sample graph.

The procedure BFS works as follows. Lines 1–4 paint every vertex white, set $d[u]$ to be infinity for each vertex u, and set the parent of every vertex to be NIL. Line 5 paints the source vertex s gray, since it is considered to be discovered when the procedure begins. Line 6 initializes $d[s]$ to 0, and line 7 sets the predecessor of the source to be NIL. Lines 8–9 initialize Q to the queue containing just the vertex s.

The **while** loop of lines 10–18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined. This **while** loop maintains the following invariant:

At the test in line 10, the queue Q consists of the set of gray vertices.

Although we won't use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The **for** loop of lines 12–17 considers each vertex v in the adjacency list of u. If v is white, then it has not yet been discovered, and the algorithm discovers it by executing lines 14–17. It is first grayed, and its distance $d[v]$ is set to $d[u] + 1$. Then, u is recorded as its parent. Finally, it is placed at the tail of the queue Q. When all the vertices on u's adjacency list have been examined, u is blackened in line 18. The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).

The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12: the breadth-first tree may vary, but the distances d computed by the algorithm will not. (See Exercise 22.2-4.)

Analysis

Before proving the various properties of breadth-first search, we take on the somewhat easier job of analyzing its running time on an input graph $G = (V, E)$. We use aggregate analysis, as we saw in Section 17.1. After initialization, no vertex is ever whitened, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, so the total time devoted to queue operations is $O(V)$. Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of BFS is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Shortest paths

At the beginning of this section, we claimed that breadth-first search finds the distance to each reachable vertex in a graph $G = (V, E)$ from a given source vertex $s \in V$. Define the ***shortest-path distance*** $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v; if there is no path from s to v,

then $\delta(s, v) = \infty$. A path of length $\delta(s, v)$ from s to v is said to be a ***shortest path***[1] from s to v. Before showing that breadth-first search actually computes shortest-path distances, we investigate an important property of shortest-path distances.

Lemma 22.1
Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + 1 .$$

Proof If u is reachable from s, then so is v. In this case, the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u, v), and thus the inequality holds. If u is not reachable from s, then $\delta(s, u) = \infty$, and the inequality holds. ■

We want to show that BFS properly computes $d[v] = \delta(s, v)$ for each vertex $v \in V$. We first show that $d[v]$ bounds $\delta(s, v)$ from above.

Lemma 22.2
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $d[v]$ computed by BFS satisfies $d[v] \geq \delta(s, v)$.

Proof We use induction on the number of ENQUEUE operations. Our inductive hypothesis is that $d[v] \geq \delta(s, v)$ for all $v \in V$.

The basis of the induction is the situation immediately after s is enqueued in line 9 of BFS. The inductive hypothesis holds here, because $d[s] = 0 = \delta(s, s)$ and $d[v] = \infty \geq \delta(s, v)$ for all $v \in V - \{s\}$.

For the inductive step, consider a white vertex v that is discovered during the search from a vertex u. The inductive hypothesis implies that $d[u] \geq \delta(s, u)$. From the assignment performed by line 15 and from Lemma 22.1, we obtain

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) . \end{aligned}$$

---

[1] In Chapters 24 and 25, we shall generalize our study of shortest paths to weighted graphs, in which every edge has a real-valued weight and the weight of a path is the sum of the weights of its constituent edges. The graphs considered in the present chapter are unweighted or, equivalently, all edges have unit weight.

Vertex v is then enqueued, and it is never enqueued again because it is also grayed and the **then** clause of lines 14–17 is executed only for white vertices. Thus, the value of $d[v]$ never changes again, and the inductive hypothesis is maintained. ■

To prove that $d[v] = \delta(s, v)$, we must first show more precisely how the queue Q operates during the course of BFS. The next lemma shows that at all times, there are at most two distinct d values in the queue.

Lemma 22.3
Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_{i+1}]$ for $i = 1, 2, \ldots, r - 1$.

Proof The proof is by induction on the number of queue operations. Initially, when the queue contains only s, the lemma certainly holds.

For the inductive step, we must prove that the lemma holds after both dequeuing and enqueuing a vertex. If the head v_1 of the queue is dequeued, v_2 becomes the new head. (If the queue becomes empty, then the lemma holds vacuously.) By the inductive hypothesis, $d[v_1] \leq d[v_2]$. But then we have $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, and the remaining inequalities are unaffected. Thus, the lemma follows with v_2 as the head.

Enqueuing a vertex requires closer examination of the code. When we enqueue a vertex v in line 17 of BFS, it becomes v_{r+1}. At that time, we have already removed vertex u, whose adjacency list is currently being scanned, from the queue Q, and by the inductive hypothesis, the new head v_1 has $d[v_1] \geq d[u]$. Thus, $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. From the inductive hypothesis, we also have $d[v_r] \leq d[u] + 1$, and so $d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$, and the remaining inequalities are unaffected. Thus, the lemma follows when v is enqueued. ■

The following corollary shows that the d values at the time that vertices are enqueued are monotonically increasing over time.

Corollary 22.4
Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $d[v_i] \leq d[v_j]$ at the time that v_j is enqueued.

Proof Immediate from Lemma 22.3 and the property that each vertex receives a finite d value at most once during the course of BFS. ■

We can now prove that breadth-first search correctly finds shortest-path distances.

Theorem 22.5 (Correctness of breadth-first search)
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $d[v] = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by the edge $(\pi[v], v)$.

Proof Assume, for the purpose of contradiction, that some vertex receives a d value not equal to its shortest path distance. Let v be the vertex with minimum $\delta(s, v)$ that receives such an incorrect d value; clearly $v \neq s$. By Lemma 22.2, $d[v] \geq \delta(s, v)$, and thus we have that $d[v] > \delta(s, v)$. Vertex v must be reachable from s, for if it is not, then $\delta(s, v) = \infty \geq d[v]$. Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, u) < \delta(s, v)$, and because of how we chose v, we have $d[u] = \delta(s, u)$. Putting these properties together, we have

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 \ . \tag{22.1}$$

Now consider the time when BFS chooses to dequeue vertex u from Q in line 11. At this time, vertex v is either white, gray, or black. We shall show that in each of these cases, we derive a contradiction to inequality (22.1). If v is white, then line 15 sets $d[v] = d[u] + 1$, contradicting inequality (22.1). If v is black, then it was already removed from the queue and, by Corollary 22.4, we have $d[v] \leq d[u]$, again contradicting inequality (22.1). If v is gray, then it was painted gray upon dequeuing some vertex w, which was removed from Q earlier than u and for which $d[v] = d[w] + 1$. By Corollary 22.4, however, $d[w] \leq d[u]$, and so we have $d[v] \leq d[u] + 1$, once again contradicting inequality (22.1).

Thus we conclude that $d[v] = \delta(s, v)$ for all $v \in V$. All vertices reachable from s must be discovered, for if they were not, they would have infinite d values. To conclude the proof of the theorem, observe that if $\pi[v] = u$, then $d[v] = d[u] + 1$. Thus, we can obtain a shortest path from s to v by taking a shortest path from s to $\pi[v]$ and then traversing the edge $(\pi[v], v)$. ■

Breadth-first trees

The procedure BFS builds a breadth-first tree as it searches the graph, as illustrated in Figure 22.3. The tree is represented by the π field in each vertex. More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where

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

and

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

The predecessor subgraph G_π is a ***breadth-first tree*** if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G. A breadth-first tree is in fact a tree, since it is connected and $|E_\pi| = |V_\pi| - 1$ (see Theorem B.2). The edges in E_π are called ***tree edges***.

After BFS has been run from a source s on a graph G, the following lemma shows that the predecessor subgraph is a breadth-first tree.

Lemma 22.6
When applied to a directed or undirected graph $G = (V, E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

Proof Line 16 of BFS sets $\pi[v] = u$ if and only if $(u, v) \in E$ and $\delta(s, v) < \infty$—that is, if v is reachable from s—and thus V_π consists of the vertices in V reachable from s. Since G_π forms a tree, by Theorem B.2, it contains a unique path from s to each vertex in V_π. By applying Theorem 22.5 inductively, we conclude that every such path is a shortest path. ■

The following procedure prints out the vertices on a shortest path from s to v, assuming that BFS has already been run to compute the shortest-path tree.

```
PRINT-PATH(G, s, v)
1  if v = s
2     then print s
3     else if π[v] = NIL
4             then print "no path from" s "to" v "exists"
5             else PRINT-PATH(G, s, π[v])
6                  print v
```

This procedure runs in time linear in the number of vertices in the path printed, since each recursive call is for a path one vertex shorter.

22.3 Depth-first search

The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered.

As in breadth-first search, whenever a vertex v is discovered during a scan of the adjacency list of an already discovered vertex u, depth-first search records this event by setting v's predecessor field $\pi[v]$ to u. Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may be repeated from multiple sources.[2] The ***predecessor subgraph*** of a depth-first search is therefore defined slightly differently from that of a breadth-first search: we let $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\} \ .$$

The predecessor subgraph of a depth-first search forms a ***depth-first forest*** composed of several ***depth-first trees***. The edges in E_π are called ***tree edges***.

As in breadth-first search, vertices are colored during the search to indicate their state. Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, depth-first search also ***timestamps*** each vertex. Each vertex v has two timestamps: the first timestamp $d[v]$ records when v is first discovered (and grayed), and the second timestamp $f[v]$ records when the search finishes examining v's adjacency list (and blackens v). These timestamps

[2] It may seem arbitrary that breadth-first search is limited to only one source whereas depth-first search may search from multiple sources. Although conceptually, breadth-first search could proceed from multiple sources and depth-first search could be limited to one source, our approach reflects how the results of these searches are typically used. Breadth-first search is usually employed to find shortest-path distances (and the associated predecessor subgraph) from a given source. Depth-first search is often a subroutine in another algorithm, as we shall see later in this chapter.

are used in many graph algorithms and are generally helpful in reasoning about the behavior of depth-first search.

The procedure DFS below records when it discovers vertex u in the variable $d[u]$ and when it finishes vertex u in the variable $f[u]$. These timestamps are integers between 1 and $2|V|$, since there is one discovery event and one finishing event for each of the $|V|$ vertices. For every vertex u,

$$d[u] < f[u] . \tag{22.2}$$

Vertex u is WHITE before time $d[u]$, GRAY between time $d[u]$ and time $f[u]$, and BLACK thereafter.

The following pseudocode is the basic depth-first-search algorithm. The input graph G may be undirected or directed. The variable *time* is a global variable that we use for timestamping.

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)

DFS-VISIT(u)
1  color[u] ← GRAY          ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]      ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK         ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 illustrates the progress of DFS on the graph shown in Figure 22.2.

Procedure DFS works as follows. Lines 1–3 paint all vertices white and initialize their π fields to NIL. Line 4 resets the global time counter. Lines 5–7 check each vertex in V in turn and, when a white vertex is found, visit it using DFS-VISIT. Every time DFS-VISIT(u) is called in line 7, vertex u becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex u has been assigned a ***discovery time*** $d[u]$ and a ***finishing time*** $f[u]$.

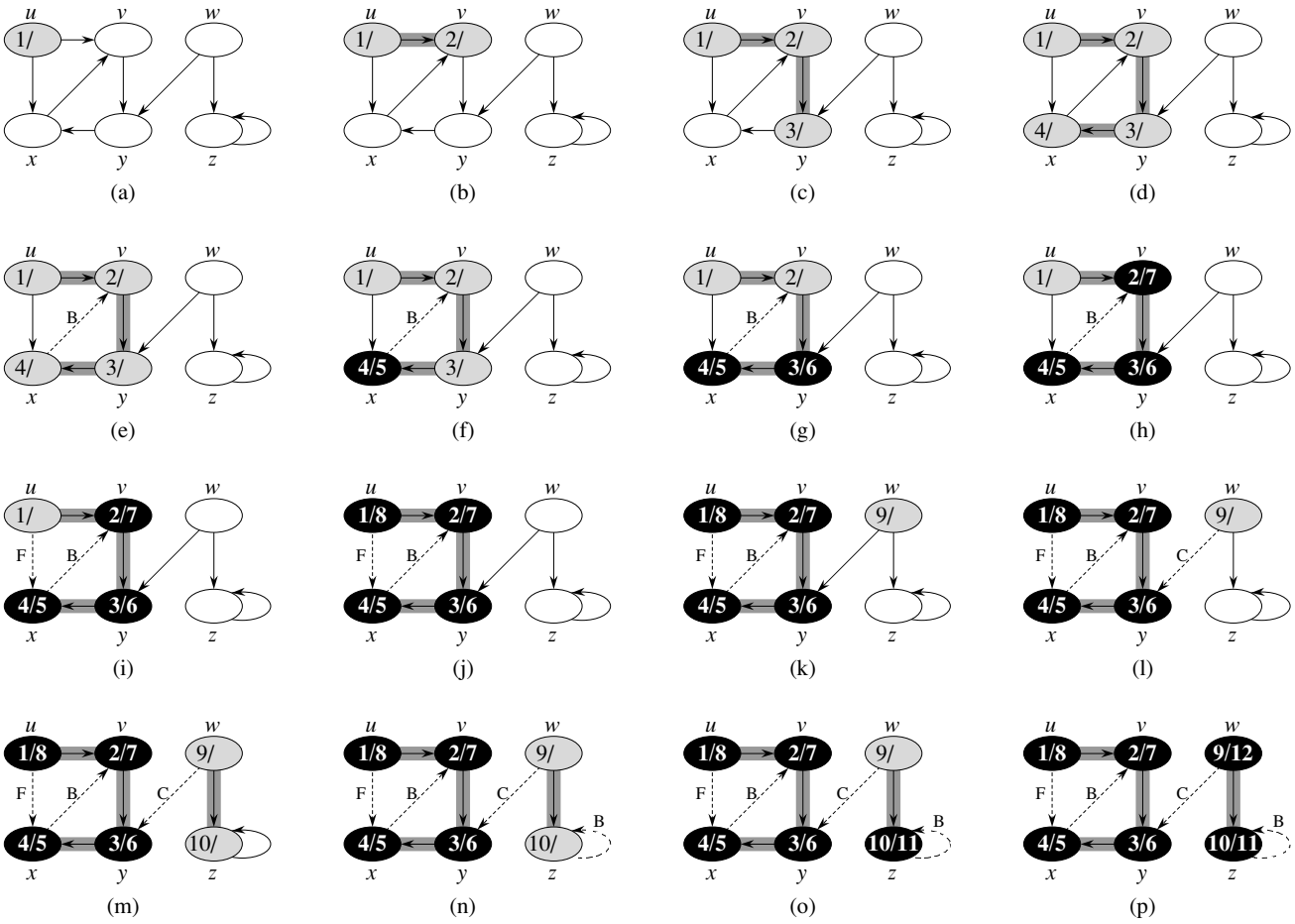

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

In each call DFS-VISIT(u), vertex u is initially white. Line 1 paints u gray, line 2 increments the global variable *time*, and line 3 records the new value of *time* as the discovery time $d[u]$. Lines 4–7 examine each vertex v adjacent to u and recursively visit v if it is white. As each vertex $v \in Adj[u]$ is considered in line 4, we say that edge (u, v) is ***explored*** by the depth-first search. Finally, after every edge leaving u has been explored, lines 8–9 paint u black and record the finishing time in $f[u]$.

Note that the results of depth-first search may depend upon the order in which the vertices are examined in line 5 of DFS, and upon the order in which the neighbors of a vertex are visited in line 4 of DFS-VISIT. These different visitation orders tend not to cause problems in practice, as *any* depth-first search result can usually be used effectively, with essentially equivalent results.

What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since DFS-VISIT is invoked only on white vertices and the first thing it does is paint the vertex gray. During an execution of DFS-VISIT(v), the loop on lines 4–7 is executed $|Adj[v]|$ times. Since

$$\sum_{v \in V} |Adj[v]| = \Theta(E) ,$$

the total cost of executing lines 4–7 of DFS-VISIT is $\Theta(E)$. The running time of DFS is therefore $\Theta(V + E)$.

Properties of depth-first search

Depth-first search yields valuable information about the structure of a graph. Perhaps the most basic property of depth-first search is that the predecessor subgraph G_π does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT. That is, $u = \pi[v]$ if and only if DFS-VISIT(v) was called during a search of u's adjacency list. Additionally, vertex v is a descendant of vertex u in the depth-first forest if and only if v is discovered during the time in which u is gray.

Another important property of depth-first search is that discovery and finishing times have ***parenthesis structure***. If we represent the discovery of vertex u with a left parenthesis "(u" and represent its finishing by a right parenthesis "u)", then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested. For example, the depth-first search of Figure 22.5(a) corresponds to the parenthesization shown in Figure 22.5(b). Another way of stating the condition of parenthesis structure is given in the following theorem.

Theorem 22.7 (Parenthesis theorem)
In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in a depth-first tree, or
- the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in a depth-first tree.

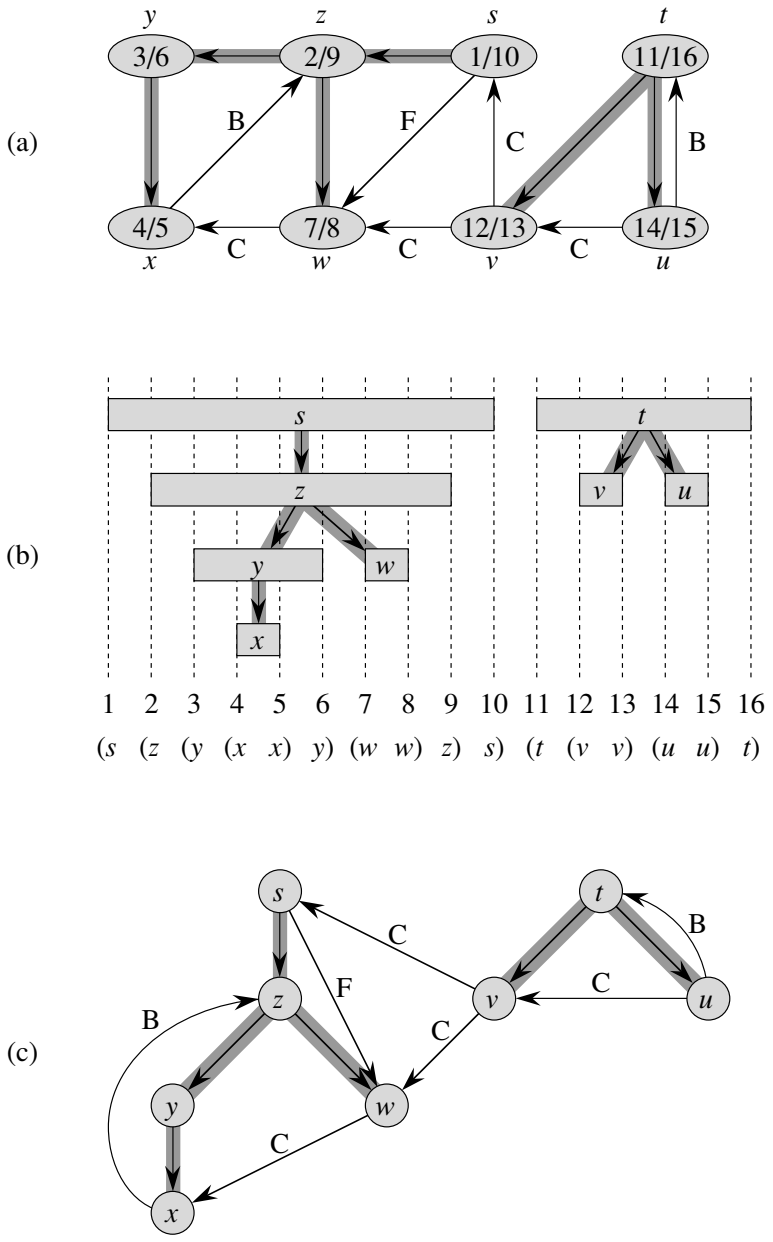

Figure 22.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. **(b)** Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex. Tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. **(c)** The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

Proof We begin with the case in which $d[u] < d[v]$. There are two subcases to consider, according to whether $d[v] < f[u]$ or not. The first subcase occurs when $d[v] < f[u]$, so v was discovered while u was still gray. This implies that v is a descendant of u. Moreover, since v was discovered more recently than u, all of its outgoing edges are explored, and v is finished, before the search returns to and finishes u. In this case, therefore, the interval $[d[v], f[v]]$ is entirely contained within the interval $[d[u], f[u]]$. In the other subcase, $f[u] < d[v]$, and inequality (22.2) implies that the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

The case in which $d[v] < d[u]$ is similar, with the roles of u and v reversed in the above argument. ■

Corollary 22.8 (Nesting of descendants' intervals)
Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $d[u] < d[v] < f[v] < f[u]$.

Proof Immediate from Theorem 22.7. ■

The next theorem gives another important characterization of when one vertex is a descendant of another in the depth-first forest.

Theorem 22.9 (White-path theorem)
In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $d[u]$ that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

Proof $\Rightarrow$: Assume that v is a descendant of u. Let w be any vertex on the path between u and v in the depth-first tree, so that w is a descendant of u. By Corollary 22.8, $d[u] < d[w]$, and so w is white at time $d[u]$.

$\Leftarrow$: Suppose that vertex v is reachable from u along a path of white vertices at time $d[u]$, but v does not become a descendant of u in the depth-first tree. Without loss of generality, assume that every other vertex along the path becomes a descendant of u. (Otherwise, let v be the closest vertex to u along the path that doesn't become a descendant of u.) Let w be the predecessor of v in the path, so that w is a descendant of u (w and u may in fact be the same vertex) and, by Corollary 22.8, $f[w] \leq f[u]$. Note that v must be discovered after u is discovered, but before w is finished. Therefore, $d[u] < d[v] < f[w] \leq f[u]$. Theorem 22.7 then implies that the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$. By Corollary 22.8, v must after all be a descendant of u. ■

Classification of edges

Another interesting property of depth-first search is that the search can be used to classify the edges of the input graph $G = (V, E)$. This edge classification can be used to glean important information about a graph. For example, in the next section, we shall see that a directed graph is acyclic if and only if a depth-first search yields no "back" edges (Lemma 22.11).

We can define four edge types in terms of the depth-first forest G_π produced by a depth-first search on G.

1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees. For example, siblings, or one that connects 2 trees.

In Figures 22.4 and 22.5, edges are labeled to indicate their type. Figure 22.5(c) also shows how the graph of Figure 22.5(a) can be redrawn so that all tree and forward edges head downward in a depth-first tree and all back edges go up. Any graph can be redrawn in this fashion.

The DFS algorithm can be modified to classify edges as it encounters them. The key idea is that each edge (u, v) can be classified by the color of the vertex v that is reached when the edge is first explored (except that forward and cross edges are not distinguished):

1. WHITE indicates a tree edge,
2. GRAY indicates a back edge, and
3. BLACK indicates a forward or cross edge.

The first case is immediate from the specification of the algorithm. For the second case, observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active DFS-VISIT invocations; the number of gray vertices is one more than the depth in the depth-first forest of the vertex most recently discovered. Exploration always proceeds from the deepest gray vertex, so an edge that reaches another gray vertex reaches an ancestor. The third case handles the remaining possibility; it can be shown that such an edge (u, v) is a forward edge if $d[u] < d[v]$ and a cross edge if $d[u] > d[v]$. (See Exercise 22.3-4.)

In an undirected graph, there may be some ambiguity in the type classification, since (u, v) and (v, u) are really the same edge. In such a case, the edge is classified as the *first* type in the classification list that applies. Equivalently (see Exercise 22.3-5), the edge is classified according to whichever of (u, v) or (v, u) is encountered first during the execution of the algorithm.

We now show that forward and cross edges never occur in a depth-first search of an undirected graph.

Theorem 22.10
In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

Proof Let (u, v) be an arbitrary edge of G, and suppose without loss of generality that $d[u] < d[v]$. Then, v must be discovered and finished before we finish u (while u is gray), since v is on u's adjacency list. If the edge (u, v) is explored first in the direction from u to v, then v is undiscovered (white) until that time, for otherwise we would have explored this edge already in the direction from v to u. Thus, (u, v) becomes a tree edge. If (u, v) is explored first in the direction from v to u, then (u, v) is a back edge, since u is still gray at the time the edge is first explored. ■

We shall see several applications of these theorems in the following sections.

22.4 Topological sort

This section shows how depth-first search can be used to perform a topological sort of a directed acyclic graph, or a "dag" as it is sometimes called. A ***topological sort*** of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. (If the graph is not acyclic, then no linear ordering is possible.) A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of "sorting" studied in Part II.

Directed acyclic graphs are used in many applications to indicate precedences among events. Figure 22.7 gives an example that arises when Professor Bumstead gets dressed in the morning. The professor must don certain garments before others (e.g., socks before shoes). Other items may be put on in any order (e.g., socks and pants). A directed edge (u, v) in the dag of Figure 22.7(a) indicates that garment u must be donned before garment v. A topological sort of this dag therefore gives an order for getting dressed. Figure 22.7(b) shows the topologically sorted dag as an ordering of vertices along a horizontal line such that all directed edges go from left to right.

The following simple algorithm topologically sorts a dag.

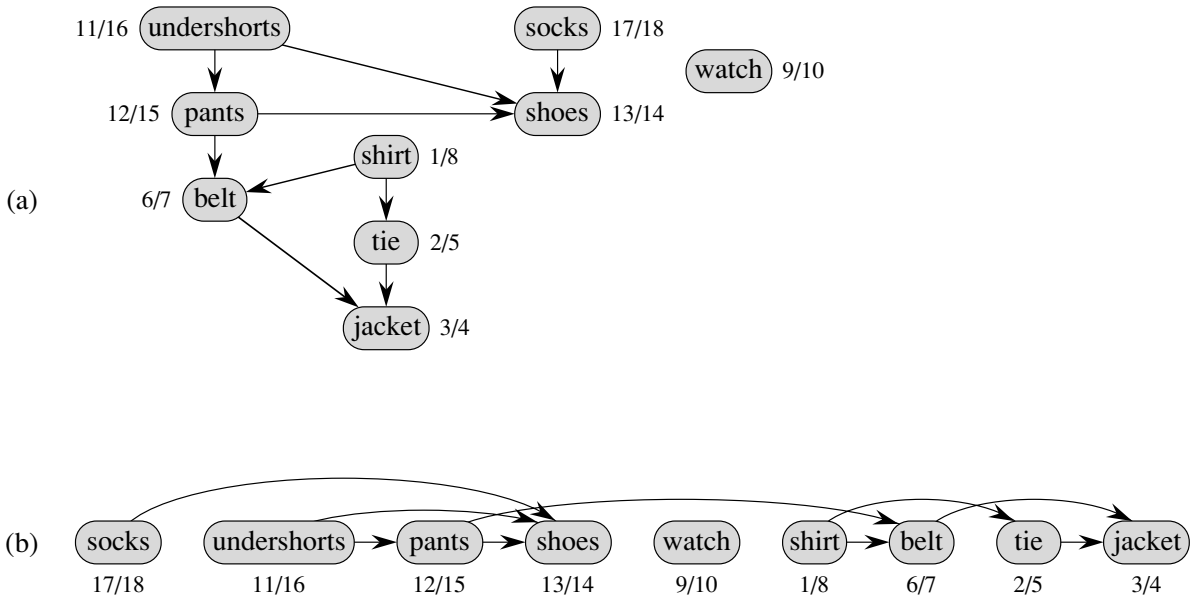

Figure 22.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted. Its vertices are arranged from left to right in order of decreasing finishing time. Note that all directed edges go from left to right.

TOPOLOGICAL-SORT(G)
1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Figure 22.7(b) shows how the topologically sorted vertices appear in reverse order of their finishing times.

We can perform a topological sort in time $\Theta(V + E)$, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

We prove the correctness of this algorithm using the following key lemma characterizing directed acyclic graphs.

Lemma 22.11
A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

Proof $\Rightarrow$: Suppose that there is a back edge (u, v). Then, vertex v is an ancestor of vertex u in the depth-first forest. There is thus a path from v to u in G, and the back edge (u, v) completes a cycle.

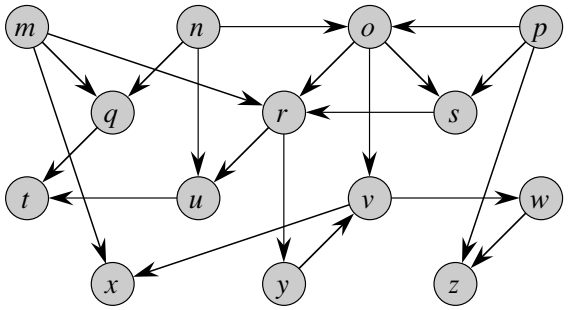

Figure 22.8 A dag for topological sorting.

$\Leftarrow$: Suppose that G contains a cycle c. We show that a depth-first search of G yields a back edge. Let v be the first vertex to be discovered in c, and let (u, v) be the preceding edge in c. At time $d[v]$, the vertices of c form a path of white vertices from v to u. By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest. Therefore, (u, v) is a back edge. ■

Theorem 22.12

TOPOLOGICAL-SORT(G) produces a topological sort of a directed acyclic graph G.

Proof Suppose that DFS is run on a given dag $G = (V, E)$ to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices $u, v \in V$, if there is an edge in G from u to v, then $f[v] < f[u]$. Consider any edge (u, v) explored by DFS(G). When this edge is explored, v cannot be gray, since then v would be an ancestor of u and (u, v) would be a back edge, contradicting Lemma 22.11. Therefore, v must be either white or black. If v is white, it becomes a descendant of u, and so $f[v] < f[u]$. If v is black, it has already been finished, so that $f[v]$ has already been set. Because we are still exploring from u, we have yet to assign a timestamp to $f[u]$, and so once we do, we will have $f[v] < f[u]$ as well. Thus, for any edge (u, v) in the dag, we have $f[v] < f[u]$, proving the theorem. ■

22.5 Strongly connected components

We now consider a classic application of depth-first search: decomposing a directed graph into its strongly connected components. This section shows how to do this decomposition using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition. After decomposition, the algorithm is run separately on each strongly connected component. The solutions are then combined according to the structure of connections between components.

Recall from Appendix B that a strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other. Figure 22.9 shows an example.

Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of G, which is defined in Exercise 22.1-3 to be the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$. That is, E^{T} consists of the edges of G with their directions reversed. Given an adjacency-list representation of G, the time to create G^{T} is $O(V + E)$. It is interesting to observe that G and G^{T} have

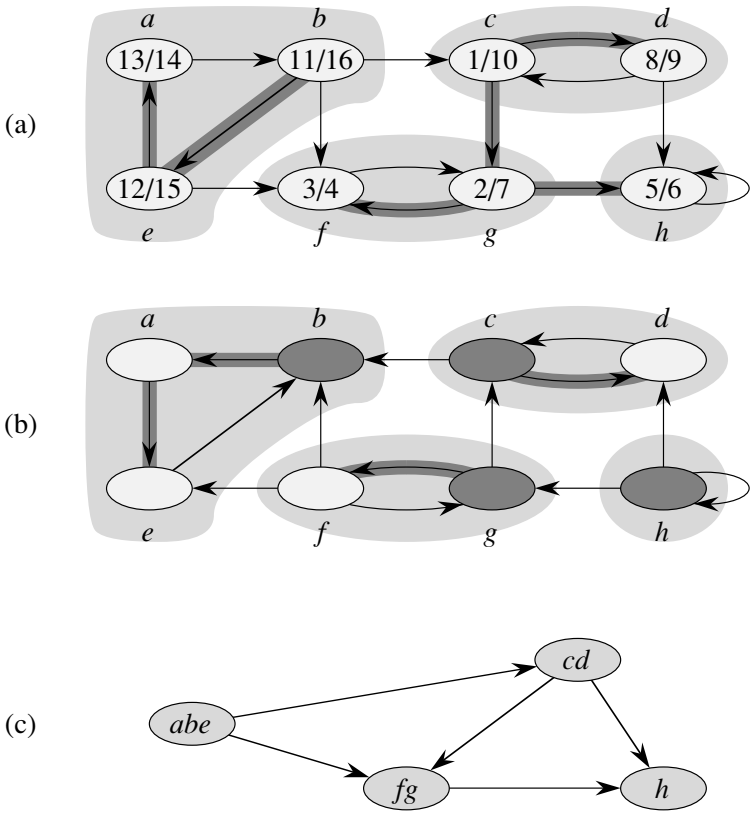

Figure 22.9 **(a)** A directed graph G. The strongly connected components of G are shown as shaded regions. Each vertex is labeled with its discovery and finishing times. Tree edges are shaded. **(b)** The graph G^{T}, the transpose of G. The depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS is shown, with tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^{T}. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

exactly the same strongly connected components: u and v are reachable from each other in G if and only if they are reachable from each other in G^{T}. Figure 22.9(b) shows the transpose of the graph in Figure 22.9(a), with the strongly connected components shaded.

The following linear-time (i.e., $\Theta(V+E)$-time) algorithm computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^{T}.

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $f[u]$ for each vertex u
2 compute G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

The idea behind this algorithm comes from a key property of the ***component graph*** $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$, which we define as follows. Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set V^{SCC} is $\{v_1, v_2, \ldots, v_k\}$, and it contains a vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E^{\mathrm{SCC}}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$. Looked at another way, by contracting all edges whose incident vertices are within the same strongly connected component of G, the resulting graph is G^{SCC}. Figure 22.9(c) shows the component graph of the graph in Figure 22.9(a).

The key property is that the component graph is a dag, which the following lemma implies.

Lemma 22.13

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

Proof If there is a path $v' \rightsquigarrow v$ in G, then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G. Thus, u and v' are reachable from each other, thereby contradicting the assumption that C and C' are distinct strongly connected components. ■

We shall see that by considering vertices in the second depth-first search in decreasing order of the finishing times that were computed in the first depth-first search, we are, in essence, visiting the vertices of the component graph (each of which corresponds to a strongly connected component of G) in topologically sorted order.

Because STRONGLY-CONNECTED-COMPONENTS performs two depth-first searches, there is the potential for ambiguity when we discuss $d[u]$ or $f[u]$. In this section, these values always refer to the discovery and finishing times as computed by the first call of DFS, in line 1.

We extend the notation for discovery and finishing times to sets of vertices. If $U \subseteq V$, then we define $d(U) = \min_{u \in U} \{d[u]\}$ and $f(U) = \max_{u \in U} \{f[u]\}$. That is, $d(U)$ and $f(U)$ are the earliest discovery time and latest finishing time, respectively, of any vertex in U.

The following lemma and its corollary give a key property relating strongly connected components and finishing times in the first depth-first search.

Lemma 22.14
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Proof There are two cases, depending on which strongly connected component, C or C', had the first discovered vertex during the depth-first search.

If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $d[x]$, all vertices in C and C' are white. There is a path in G from x to each vertex in C consisting only of white vertices. Because $(u, v) \in E$, for any vertex $w \in C'$, there is also a path at time $d[x]$ from x to w in G consisting only of white vertices: $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. By the white-path theorem, all vertices in C and C' become descendants of x in the depth-first tree. By Corollary 22.8, $f[x] = f(C) > f(C')$.

If instead we have $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $d[y]$, all vertices in C' are white and there is a path in G from y to each vertex in C' consisting only of white vertices. By the white-path theorem, all vertices in C' become descendants of y in the depth-first tree, and by Corollary 22.8, $f[y] = f(C')$. At time $d[y]$, all vertices in C are white. Since there is an edge (u, v) from C to C', Lemma 22.13 implies that there cannot be a path from C' to C. Hence, no vertex in C is reachable from y. At time $f[y]$, therefore, all vertices in C are still white. Thus, for any vertex $w \in C$, we have $f[w] > f[y]$, which implies that $f(C) > f(C')$. ■

The following corollary tells us that each edge in G^{T} that goes between different strongly connected components goes from a component with an earlier finishing time (in the first depth-first search) to a component with a later finishing time.

Corollary 22.15
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^{\mathrm{T}}$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof Since $(u, v) \in E^{\mathrm{T}}$, we have $(v, u) \in E$. Since the strongly connected components of G and G^{T} are the same, Lemma 22.14 implies that $f(C) < f(C')$. ■

Corollary 22.15 provides the key to understanding why the STRONGLY-CONNECTED-COMPONENTS procedure works. Let us examine what happens when we perform the second depth-first search, which is on G^{T}. We start with the strongly connected component C whose finishing time $f(C)$ is maximum. The

search starts from some vertex $x \in C$, and it visits all vertices in C. By Corollary 22.15, there are no edges in G^T from C to any other strongly connected component, and so the search from x will not visit vertices in any other component. Thus, the tree rooted at x contains exactly the vertices of C. Having completed visiting all vertices in C, the search in line 3 selects as a root a vertex from some other strongly connected component C' whose finishing time $f(C')$ is maximum over all components other than C. Again, the search will visit all vertices in C', but by Corollary 22.15, the only edges in G^T from C' to any other component must be to C, which we have already visited. In general, when the depth-first search of G^T in line 3 visits any strongly connected component, any edges out of that component must be to components that were already visited. Each depth-first tree, therefore, will be exactly one strongly connected component. The following theorem formalizes this argument.

Theorem 22.16

STRONGLY-CONNECTED-COMPONENTS(G) correctly computes the strongly connected components of a directed graph G.

Proof We argue by induction on the number of depth-first trees found in the depth-first search of G^T in line 3 that the vertices of each tree form a strongly connected component. The inductive hypothesis is that the first k trees produced in line 3 are strongly connected components. The basis for the induction, when $k = 0$, is trivial.

In the inductive step, we assume that each of the first k depth-first trees produced in line 3 is a strongly connected component, and we consider the $(k+1)$st tree produced. Let the root of this tree be vertex u, and let u be in strongly connected component C. Because of how we choose roots in the depth-first search in line 3, $f[u] = f(C) > f(C')$ for any strongly connected component C' other than C that has yet to be visited. By the inductive hypothesis, at the time that the search visits u, all other vertices of C are white. By the white-path theorem, therefore, all other vertices of C are descendants of u in its depth-first tree. Moreover, by the inductive hypothesis and by Corollary 22.15, any edges in G^T that leave C must be to strongly connected components that have already been visited. Thus, no vertex in any strongly connected component other than C will be a descendant of u during the depth-first search of G^T. Thus, the vertices of the depth-first tree in G^T that is rooted at u form exactly one strongly connected component, which completes the inductive step and the proof. ■

Here is another way to look at how the second depth-first search operates. Consider the component graph $(G^T)^{SCC}$ of G^T. If we map each strongly connected component visited in the second depth-first search to a vertex of $(G^T)^{SCC}$, the vertices of $(G^T)^{SCC}$ are visited in the reverse of a topologically sorted order. If we re-

verse the edges of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, we get the graph $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}}$. Because $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$ (see Exercise 22.5-4), the second depth-first search visits the vertices of G^{SCC} in topologically sorted order.

The above discussions from CLRS on BFS include those on shortest-paths. BFS works as a shortest-paths algorithm for single-source, all destinations, for unweighted graphs. For graphs with weighted edges, it is easy to establish that BFS does not work as a shortest-paths algorithm.

A candidate algorithm is Bellman-Ford, which can be perceived as incremental, because after i iterations, it guarantees that a shortest-path to any vertex from the source which comprises at most i edges has been discovered. We now reproduce, from CLRS, background on the generalized shortest-paths problem, and the Bellman-Ford algorithm. We reproduce also the situation that we have a Directed Acyclic Graph (DAG), for which it is easy to determine shortest-paths and distances.

24 Single-Source Shortest Paths

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 $w : E \to \mathbf{R}$ 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 intersection in Boston (say, Brookline Ave. and Yawkey Way).

Edge weights can be interpreted as metrics other than distances. They are often used to represent time, cost, penalties, 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 $w : E \to \mathbf{R}$, let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from vertex v_1 to vertex v_k and, for any i and j such that $1 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ 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 $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

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. Oth-

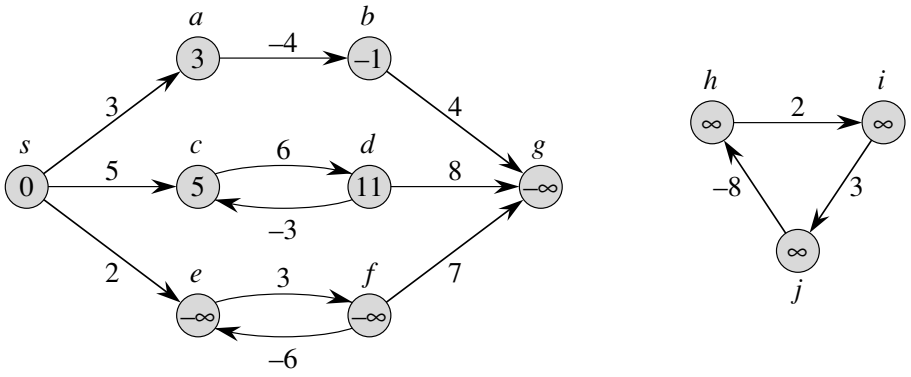

Figure 24.1 Negative edge weights in a directed graph. Shown within each vertex is its shortest-path weight from source s. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path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

ers, 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 $\pi[v] \neq \text{NIL}$, 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 $w : E \to \mathbf{R}$, 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.

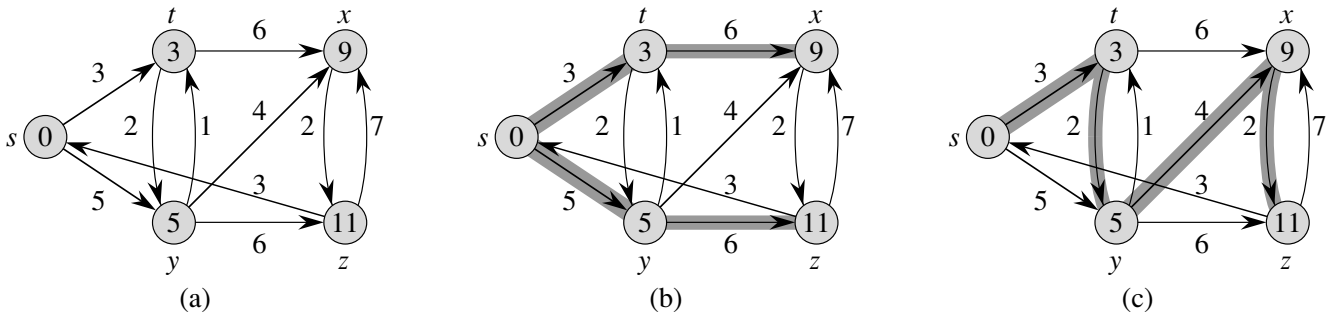

Figure 24.2 **(a)** A weighted, directed graph with shortest-path weights from source s. **(b)** The shaded edges form a shortest-paths tree rooted at the source s. **(c)** Another shortest-paths tree with the same root.

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 estimate $d[v]$ and update v's predecessor field $\pi[v]$. The following code performs a relaxation step on edge (u, v).

[1] It may seem strange that the term "relaxation" is used for an operation that tightens an upper bound. The use of the term is historical. The outcome of a relaxation step can be viewed as a relaxation of the constraint $d[v] \le d[u] + w(u, v)$, which, by the triangle inequality (Lemma 24.10), must be satisfied if $d[u] = \delta(s, u)$ and $d[v] = \delta(s, v)$. That is, if $d[v] \le d[u] + w(u, v)$, there is no "pressure" to satisfy this constraint, so the constraint is "relaxed."

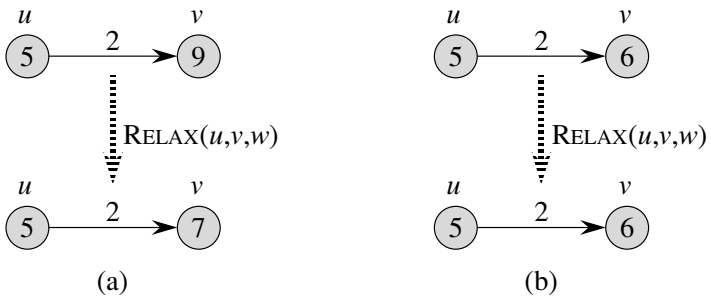

Figure 24.3 Relaxation of an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex is shown within the vertex. **(a)** Because $d[v] > d[u] + w(u, v)$ prior to relaxation, the value of $d[v]$ decreases. **(b)** Here, $d[v] \leq d[u] + w(u, v)$ before the relaxation step, and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

```
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.

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

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.

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.

24.5 Proofs of shortest-paths properties

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 $w : E \to \mathbf{R}$ 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 $w : E \to \mathbf{R}$. 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

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

and so the invariant is maintained.

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 $w : E \to \mathbf{R}$, 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 $w : E \to \mathbf{R}$, 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 $w : E \to \mathbf{R}$, let $s \in V$ be a source vertex, and let $s \rightsquigarrow u \to v$ 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

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

By the upper-bound property, $d[v] \geq \delta(s, v)$, 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 $w : E \to \mathbf{R}$, 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 $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, 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 $w : E \to \mathbf{R}$, 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

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

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

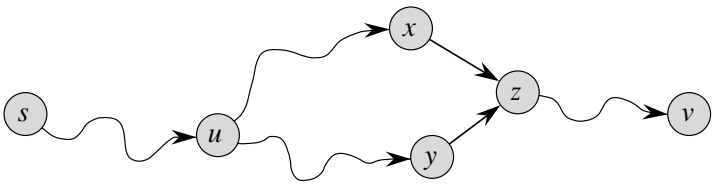

Figure 24.9 Showing that a path in G_π from source s to vertex v is unique. If there are two paths p_1 ($s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$) and p_2 ($s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$), where $x \neq y$, then $\pi[z] = x$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

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. ■

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

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

Thus, $w(p) \le \delta(s, v_k)$. Since $\delta(s, v_k)$ is a lower bound on the weight of any path from s to v_k, we conclude that $w(p) = \delta(s, v_k)$, and thus p is a shortest path from s to $v = v_k$. ■

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 $w : E \to \mathbf{R}$, 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.)

The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(E)$ time, and the **for** loop of lines 5–7 takes $O(E)$ time.

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

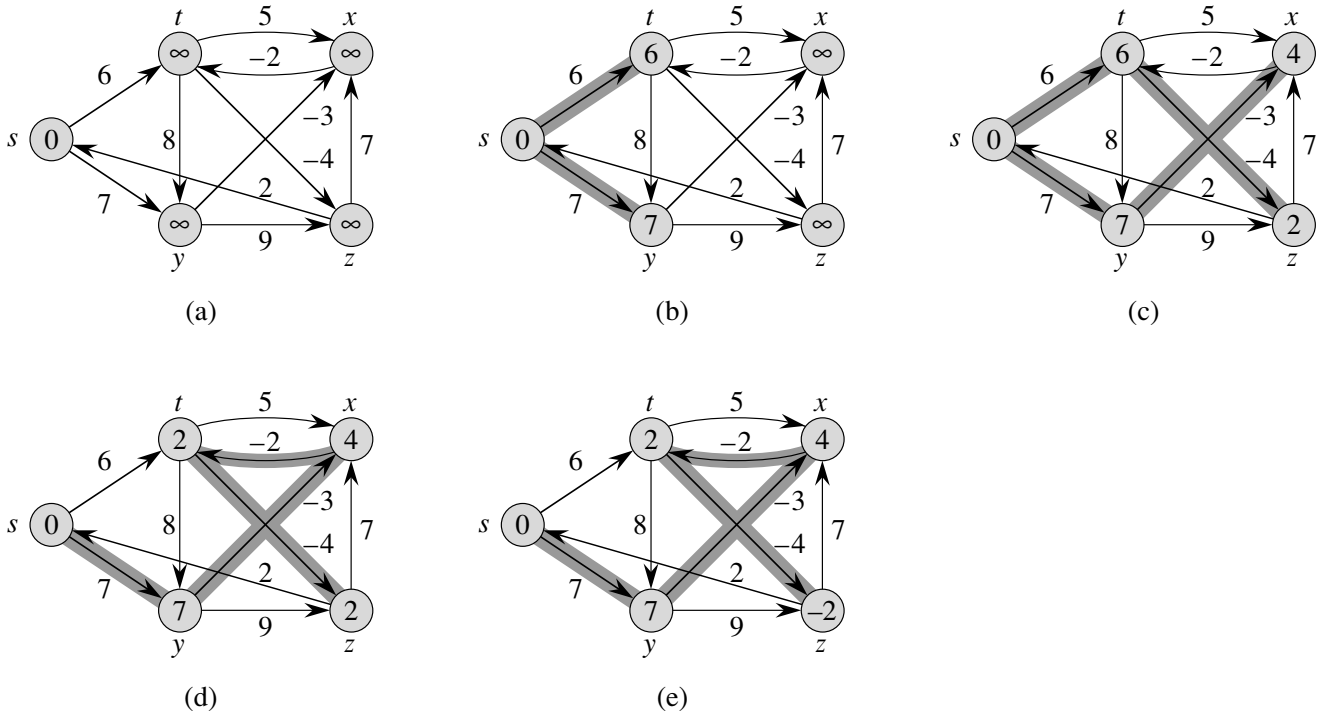

Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order (t, x), (t, y),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

Lemma 24.2

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \to \mathbf{R}$, 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 $w : E \to \mathbf{R}$. 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.

Theorem 24.4 (Correctness of the Bellman-Ford algorithm)
Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbf{R}$. 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 $(u, v) \in E$,

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

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

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

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

Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE. Thus, $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ 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. ■

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.

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.

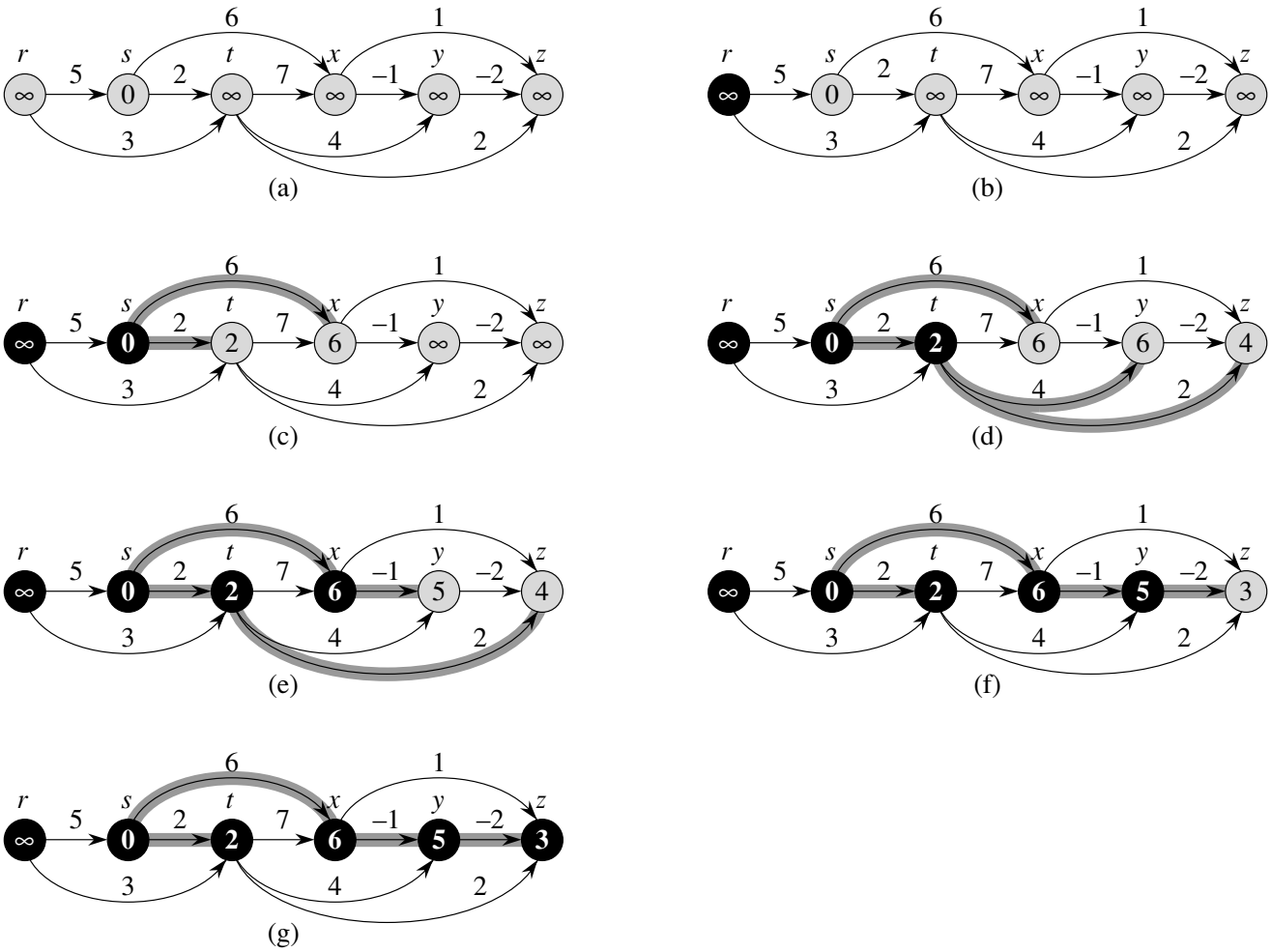

Figure 24.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values are shown within the vertices, and shaded edges indicate the π values. **(a)** The situation before the first iteration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

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