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. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once 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 depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.[3]

As in breadth-first search, whenever depth-first search discovers a vertex v during a scan of the adjacency list of an already discovered vertex u, it records this event by setting v's predecessor attribute $v.\pi$ 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 repeat from multiple sources. Therefore, we define the ***predecessor subgraph*** of a depth-first search slightly differently from that of a breadth-first search: we let $G_\pi = (V, E_\pi)$, where

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

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

As in breadth-first search, depth-first search colors vertices 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 $v.d$ records when v is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens v). These timestamps

[3]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 usually serves 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.

provide important information about the structure of the graph 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 attribute $u.d$ and when it finishes vertex u in the attribute $u.f$. 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,

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

Vertex u is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$, 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 ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1              // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]        // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK              // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

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 attributes 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(G, u) is called in line 7, vertex u becomes

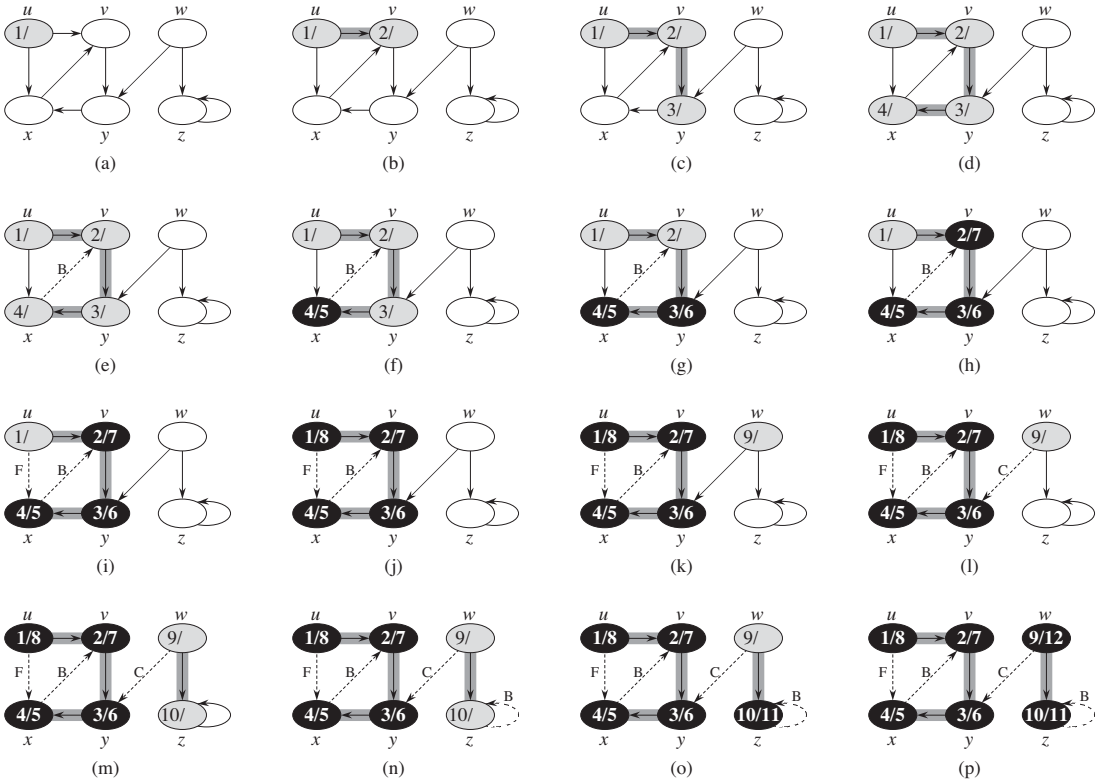

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges.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

the root of a new tree in the depth-first forest. When DFS returns, every vertex u has been assigned a ***discovery time*** $u.d$ and a ***finishing time*** $u.f$.

In each call DFS-VISIT(G, u), vertex u is initially white. Line 1 increments the global variable *time*, line 2 records the new value of *time* as the discovery time $u.d$, and line 3 paints u gray. 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 8–10 paint u black, increment *time*, and record the finishing time in $u.f$.

Note that the results of depth-first search may depend upon the order in which line 5 of DFS examines the vertices and upon the order in which line 4 of DFS-VISIT visits the neighbors of a vertex. These different visitation orders tend not

to cause problems in practice, as we can usually use *any* depth-first search result 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 the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT(G, v), the loop on lines 4–7 executes $|Adj[v]|$ 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, $u = v.\pi$ if and only if DFS-VISIT(G, 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). The following theorem provides another way to characterize the parenthesis structure.

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

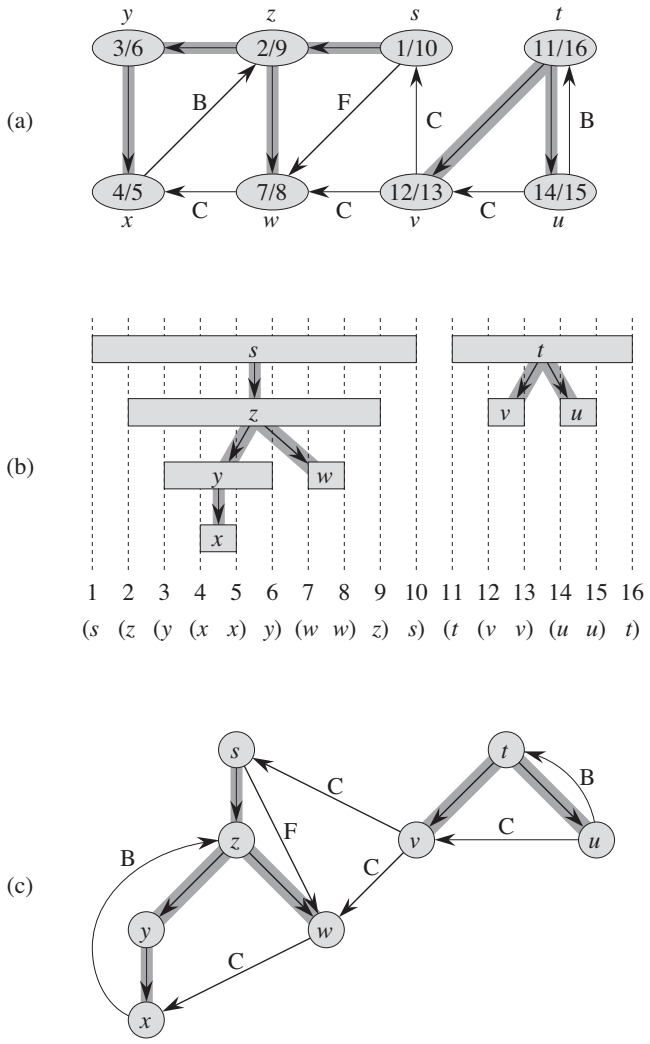

Figure 22.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. **(b)** Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex. Only 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 $u.d < v.d$. We consider two subcases, according to whether $v.d < u.f$ or not. The first subcase occurs when $v.d < u.f$, so v was discovered while u was still gray, which 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 $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$. In the other subcase, $u.f < v.d$, and by inequality (22.2), $u.d < u.f < v.d < v.f$; thus the intervals $[u.d, u.f]$ and $[v.d, v.f]$ 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 $v.d < u.d$ 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 $u.d < v.d < v.f < u.f$.

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 $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

Proof $\Rightarrow$: If $v = u$, then the path from u to v contains just vertex u, which is still white when we set the value of $u.d$. Now, suppose that v is a proper descendant of u in the depth-first forest. By Corollary 22.8, $u.d < v.d$, and so v is white at time $u.d$. Since v can be any descendant of u, all vertices on the unique simple path from u to v in the depth-first forest are white at time $u.d$.

$\Leftarrow$: Suppose that there is a path of white vertices from u to v at time $u.d$, but v does not become a descendant of u in the depth-first tree. Without loss of generality, assume that every vertex other than v 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). By Corollary 22.8, $w.f \leq u.f$. Because v must be discovered after u is discovered, but before w is finished, we have $u.d < v.d < w.f \leq u.f$. Theorem 22.7 then implies that the interval $[v.d, v.f]$

is contained entirely within the interval $[u.d, u.f]$. 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 $G = (V, E)$. The type of each edge can provide 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. We consider self-loops, which may occur in directed graphs, 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.

In Figures 22.4 and 22.5, edge labels indicate edge types. Figure 22.5(c) also shows how to redraw the graph of Figure 22.5(a) so that all tree and forward edges head downward in a depth-first tree and all back edges go up. We can redraw any graph in this fashion.

The DFS algorithm has enough information to classify some edges as it encounters them. The key idea is that when we first explore an edge (u, v), the color of vertex v tells us something about the edge:

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 has reached an ancestor. The third case handles the remaining possibility; Exercise 22.3-5 asks you to show that such an edge (u, v) is a forward edge if $u.d < v.d$ and a cross edge if $u.d > v.d$.

An undirected graph may entail some ambiguity in how we classify edges, since (u, v) and (v, u) are really the same edge. In such a case, we classify the edge as the *first* type in the classification list that applies. Equivalently (see Exercise 22.3-6), we classify the edge according to whichever of (u, v) or (v, u) the search encounters first.

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 $u.d < v.d$. Then the search must discover and finish v before it finishes u (while u is gray), since v is on u's adjacency list. If the first time that the search explores edge (u, v), it is in the direction from u to v, then v is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from v to u. Thus, (u, v) becomes a tree edge. If the search explores (u, v) 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.

Exercises

22.3-1
Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i, j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j. For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

22.3-2
Show how depth-first search works on the graph of Figure 22.6. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.

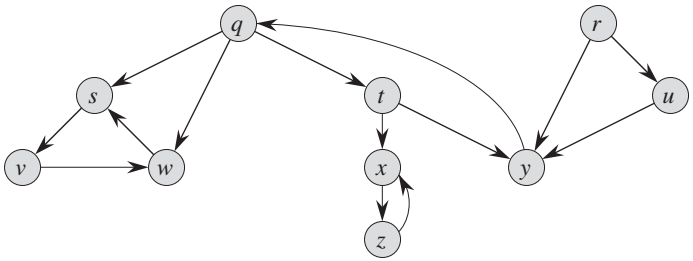

Figure 22.6 A directed graph for use in Exercises 22.3-2 and 22.5-2.

22.3-3
Show the parenthesis structure of the depth-first search of Figure 22.4.

22.3-4
Show that using a single bit to store each vertex color suffices by arguing that the DFS procedure would produce the same result if line 3 of DFS-VISIT was removed.

22.3-5
Show that edge (u, v) is

a. a tree edge or forward edge if and only if $u.d < v.d < v.f < u.f$,

b. a back edge if and only if $v.d \leq u.d < u.f \leq v.f$, and

c. a cross edge if and only if $v.d < v.f < u.d < u.f$.

22.3-6
Show that in an undirected graph, classifying an edge (u, v) as a tree edge or a back edge according to whether (u, v) or (v, u) is encountered first during the depth-first search is equivalent to classifying it according to the ordering of the four types in the classification scheme.

22.3-7
Rewrite the procedure DFS, using a stack to eliminate recursion.

22.3-8
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, and if $u.d < v.d$ in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.

22.3-9
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.d \leq u.f$.

22.3-10
Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G, together with its type. Show what modifications, if any, you need to make if G is undirected.

22.3-11
Explain how a vertex u of a directed graph can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G.

22.3-12
Show that we can use a depth-first search of an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v are in the same connected component.

22.3-13 ⋆
A directed graph $G = (V, E)$ is ***singly connected*** if $u \rightsquigarrow v$ implies that G contains at most one simple path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether or not a directed graph is singly connected.

22.4 Topological sort

This section shows how we can use depth-first search 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 contains a cycle, then no linear ordering is possible.) We can view a topological sort of a graph 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.

Many applications use directed acyclic graphs 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

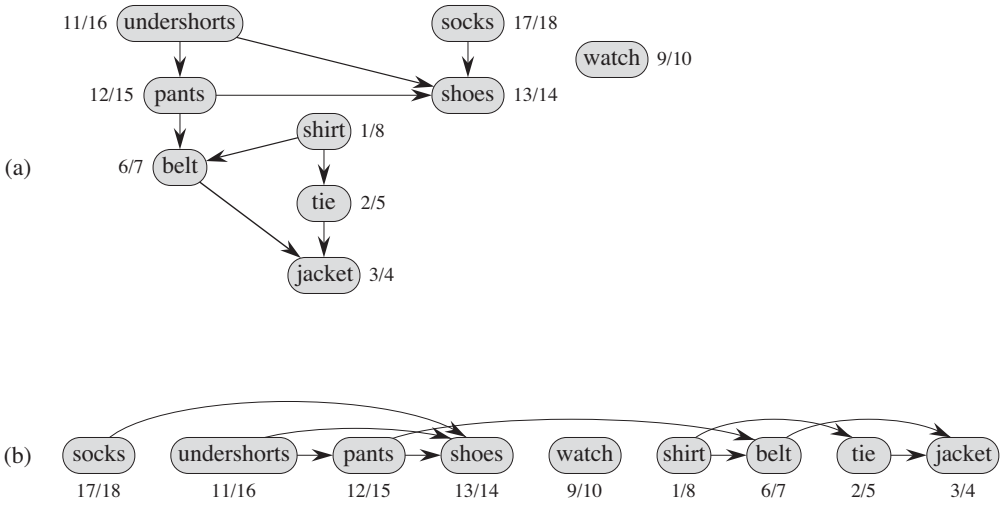

Figure 22.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically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

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:

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ 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 a depth-first search produces a back edge (u, v). Then vertex v is an ancestor of vertex u in the depth-first forest. Thus, G contains a path from v to u, and the back edge (u, v) completes a cycle.

$\Leftarrow$: 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 $v.d$, 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 produces a topological sort of the directed acyclic graph provided as its input.

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 G contains an edge from u to v, then $v.f < u.f$. 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 $v.f < u.f$. If v is black, it has already been finished, so that $v.f$ has already been set. Because we are still exploring from u, we have yet to assign a timestamp to $u.f$, and so once we do, we will have $v.f < u.f$ as well. Thus, for any edge (u, v) in the dag, we have $v.f < u.f$, proving the theorem. ■

Exercises

22.4-1
Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 22.8, under the assumption of Exercise 22.3-2.

22.4-2
Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t, and returns the number of simple paths from s to t in G. For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex p to vertex v: pov, $poryv$, $posryv$, and $psryv$. (Your algorithm needs only to count the simple paths, not list them.)

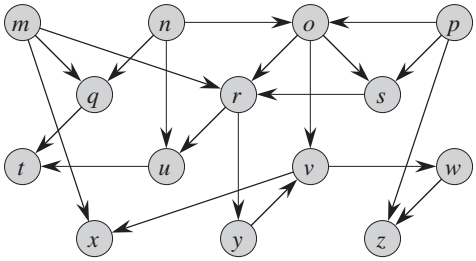

Figure 22.8 A dag for topological sorting.

22.4-3
Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

22.4-4
Prove or disprove: If a directed graph G contains cycles, then TOPOLOGICAL-SORT(G) produces a vertex ordering that minimizes the number of "bad" edges that are inconsistent with the ordering produced.

22.4-5
Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles?

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 so using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition. After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among 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.

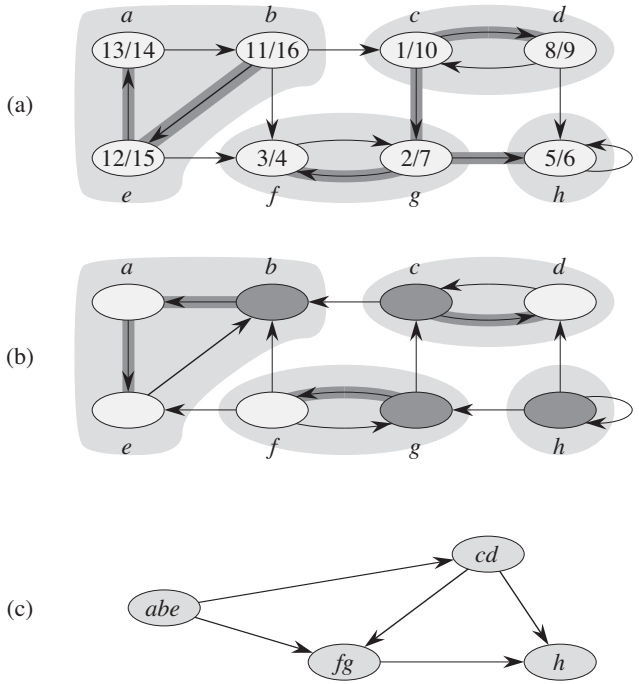

Figure 22.9 **(a)** A directed graph G. Each shaded region is a strongly connected component of G. Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded. **(b)** The graph G^{T}, the transpose of G, with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and 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.

Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of G, which we 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 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 $u.f$ 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 $u.f$ (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 G contains a path $u \rightsquigarrow u'$. Then G cannot also contain a path $v' \rightsquigarrow v$.

Proof If G contains a path $v' \rightsquigarrow v$, then it contains paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$. 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 the STRONGLY-CONNECTED-COMPONENTS procedure performs two depth-first searches, there is the potential for ambiguity when we discuss $u.d$ or $u.f$. 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 $d(U) = \min_{u \in U} \{u.d\}$ and $f(U) = \max_{u \in U} \{u.f\}$. 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 We consider 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 $x.d$, all vertices in C and C' are white. At that time, G contains a path 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 in G at time $x.d$ from x to w 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, x has the latest finishing time of any of its descendants, and so $x.f = f(C) > f(C')$.

If instead we have $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $y.d$, all vertices in C' are white and G contains a path 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, $y.f = f(C')$. At time $y.d$, 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 $y.f$, therefore, all vertices in C are still white. Thus, for any vertex $w \in C$, we have $w.f > y.f$, 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 $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof Since $(u, v) \in E^{\mathrm{T}}$, we have $(v, u) \in E$. Because 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 algorithm 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, G^{T} contains no edges 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 the search already visited. Each depth-first tree, therefore, will be exactly one strongly connected component. The following theorem formalizes this argument.

Theorem 22.16
The STRONGLY-CONNECTED-COMPONENTS procedure correctly computes the strongly connected components of the directed graph G provided as its input.

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, $u.f = f(C) > f(C')$ 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 $(G^{\mathrm{T}})^{\mathrm{SCC}}$ of G^{T}. If we map each strongly connected component visited in the second depth-first search to a vertex of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, the second depth-first search visits vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in the reverse of a topologically sorted order. If we reverse 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.

Exercises

22.5-1
How can the number of strongly connected components of a graph change if a new edge is added?

22.5-2
Show how the procedure STRONGLY-CONNECTED-COMPONENTS works on the graph of Figure 22.6. Specifically, show the finishing times computed in line 1 and the forest produced in line 3. Assume that the loop of lines 5–7 of DFS considers vertices in alphabetical order and that the adjacency lists are in alphabetical order.

22.5-3
Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of *increasing* finishing times. Does this simpler algorithm always produce correct results?

22.5-4
Prove that for any directed graph G, we have $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$. That is, the transpose of the component graph of G^{T} is the same as the component graph of G.

22.5-5
Give an $O(V + E)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

22.5-6
Given a directed graph $G = (V, E)$, explain how to create another graph $G' = (V, E')$ such that (a) G' has the same strongly connected components as G, (b) G' has the same component graph as G, and (c) E' is as small as possible. Describe a fast algorithm to compute G'.

22.5-7
A directed graph $G = (V, E)$ is ***semiconnected*** if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ or $v \rightsquigarrow u$. Give an efficient algorithm to determine whether or not G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

Problems

22-1 Classifying edges by breadth-first search
A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.

a. Prove that in a breadth-first search of an undirected graph, the following properties hold:

1. There are no back edges and no forward edges.
2. For each tree edge (u, v), we have $v.d = u.d + 1$.
3. For each cross edge (u, v), we have $v.d = u.d$ or $v.d = u.d + 1$.

b. Prove that in a breadth-first search of a directed graph, the following properties hold:

1. There are no forward edges.
2. For each tree edge (u, v), we have $v.d = u.d + 1$.
3. For each cross edge (u, v), we have $v.d \leq u.d + 1$.
4. For each back edge (u, v), we have $0 \leq v.d \leq u.d$.

22-2 Articulation points, bridges, and biconnected components
Let $G = (V, E)$ be a connected, undirected graph. An ***articulation point*** of G is a vertex whose removal disconnects G. A ***bridge*** of G is an edge whose removal disconnects G. A ***biconnected component*** of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. Figure 22.10 illustrates