And, for the hous is crinkled to and fro,
And hath so queinte weyes for to go—
For hit is shapen as the mase is wroght—
Therto have I a remedie in my thoght,
That, by a clewe of twyne, as he hath goon,
The same wey he may returne anoon,
Folwing alwey the threed, as he hath come.

— Geoffrey Chaucer, *The Legend of Good Women* (c. 1385)

"Com'è bello il mondo e come sono brutti i labirinti!" dissi sollevato.
"Come sarebbe bello il mondo se ci fosse una regola per girare nei labirinti,"
rispose il mio maestro.
["How beautiful the world is, and how ugly labyrinths are," I said, relieved.
"How beautiful the world would be if there were a procedure for moving through
labyrinths," my master replied.]

— Umberto Eco, *Il nome della rosa* (1980)
English translation (*The Name of the Rose*) by William Weaver (1983)

6

Depth-First Search

In the previous chapter, we considered a generic algorithm—whatever-first search—for traversing arbitrary graphs, both undirected and directed. In this chapter, we focus on a particular instantiation of this algorithm called *depth-first search*, and primarily on the behavior of this algorithm in directed graphs.

Although depth-first search can be accurately described as "whatever-first search with a stack", the algorithm is normally implemented recursively, rather than using an explicit stack:

```
DFS(v):
  if v is unmarked
      mark v
      for each edge v→w
          DFS(w)
```

We can make this algorithm slightly faster (in practice) by checking whether a node is marked *before* we recursively explore it. This modification ensures that we call DFS(v) only once for each vertex v. We can further modify the algorithm to compute other useful information about the vertices and edges, by introducing two black-box subroutines, PREVISIT and POSTVISIT, which we leave unspecified for now.

```
DFS(v):
  mark v
  PREVISIT(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  POSTVISIT(v)
```

Recall that a node w is *reachable* from another node v in a directed graph G—or more simply, v can reach w—if and only if G contains a directed path from v to w. Let ***reach(v)*** denote the set of vertices reachable from v (including v itself). If we unmark all vertices in G, and then call DFS(v), the set of marked vertices is precisely *reach*(v).

Reachability in undirected graphs is symmetric: v can reach w if and only if w can reach v. As a result, after unmarking all vertices of an undirected graph G, calling DFS(v) traverses the entire component of v, and the parent pointers define a spanning tree of that component.

The situation is more subtle with directed graphs, as shown in the figure below. Even though the graph is "connected", different vertices can reach different, and potentially overlapping, portions of the graph. The parent pointers assigned by DFS(v) define a tree rooted at v whose vertices are precisely *reach*(v), but this is not necessarily a spanning tree of the graph.

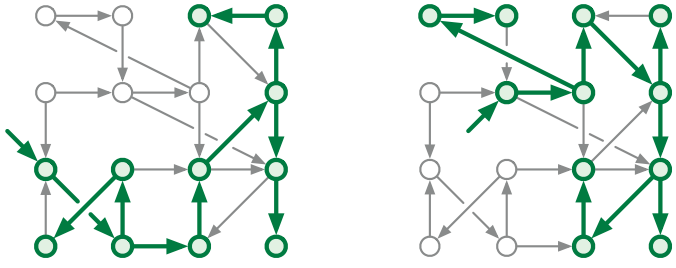

Figure 6.1. Depth-first trees rooted at different vertices in the same directed graph.

As usual, we can extend our reachability algorithm to traverse the *entire* input graph, even if it is disconnected, using the standard wrapper function shown on the left in Figure 6.2. Here we add a generic black-box subroutine

PREPROCESS to perform any necessary preprocessing for the PREVISIT and POSTVISIT functions.

```
DFSALL(G):
  PREPROCESS(G)
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

```
DFSALL(G):
  PREPROCESS(G)
  add vertex s
  for all vertices v
      add edge s→v
      unmark v
  DFS(s)
```

Figure 6.2. Two formulations of the standard wrapper algorithm for depth-first search

Alternatively, if we are allowed to modify the graph, we can add a new *source* vertex s, with edges to every other vertex in G, and then make a single call to DFS(s), as shown on the right of Figure 6.2. Now the resulting parent pointers always define a spanning tree of the *augmented* input graph, but not of the *original* input graph. Otherwise, the two wrapper functions have essentially identical behavior; choosing one or the other is entirely a matter of convenience.[1]

Again, this algorithm behaves slightly differently for undirected and directed graphs. In undirected graphs, as we saw in the previous chapter, it is easy to adapt DFSALL to count the components of a graph; in particular, the parent pointers computed by DFSALL define a spanning forest of the input graph, containing a spanning tree for each component. When the graph is directed, however, DFSALL may discover any number of "components" between 1 and V, even when the graph is "connected", depending on the precise structure of the graph and the order in which the wrapper algorithm visits the vertices.

6.1 Preorder and Postorder

Hopefully you are already familiar with preorder and postorder traversals of rooted *trees*, both of which can be computed using depth-first search. Similar traversal orders can be defined for arbitrary directed graphs—even if they are disconnected—by passing around a counter, as shown in Figure 6.3. Equivalently, we can use our generic depth-first-search algorithm with the following subroutines PREPROCESS, PREVISIT, and POSTVISIT.

```
PREPROCESS(G):
  clock ← 0
```

```
PREVISIT(v):
  clock ← clock + 1
  v.pre ← clock
```

```
POSTVISIT(v):
  clock ← clock + 1
  v.post ← clock
```

[1]The equivalence of these two wrapper functions is a specific feature of ***depth***-first search. In particular, wrapping *breadth*-first search in a for-loop to visit every vertex does *not* yield the same traversal order as adding a source vertex and invoking breadth-first search at s.

```
DFSALL(G):
  clock ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          clock ← DFS(v, clock)
```

```
DFS(v, clock):
  mark v
  clock ← clock + 1;  v.pre ← clock
  for each edge v→w
      if w is unmarked
          w.parent ← v
          clock ← DFS(w, clock)
  clock ← clock + 1;  v.post ← clock
  return clock
```

Figure 6.3. Defining preorder and postorder via depth-first search.

In either formulation, this algorithm assigns assigns $v.pre$ (and advances the clock) just after pushing v onto the recursion stack, and it assigns $v.post$ (and advances the clock) just before popping v off the recursion stack. It follows that for any two vertices u and v, the intervals $[u.pre, u.post]$ and $[v.pre, v.post]$ are either disjoint or nested. Moreover, $[u.pre, u.post]$ contains $[v.pre, v.post]$ if and only if DFS(v) is called during the execution of DFS(u), or equivalently, if and only if u is an ancestor of v in the final forest of parent pointers.

After DFSALL labels every node in the graph, the labels $v.pre$ define a ***preordering*** of the vertices, and the labels $v.post$ define a ***postordering*** of the vertices.[2] With a few trivial exceptions, every graph has several different pre- and postorderings, depending on the order that DFS considers edges leaving each vertex, and the order that DFSALL considers vertices.

For the rest of this chapter, we refer to $v.pre$ as the *starting time* of v (or less formally, "when v starts"), $v.post$ as the *finishing time* of v (or less formally, "when v finishes"), and the interval between the starting and finishing times as the *active interval* of v (or less formally, "while v is active").

Classifying Vertices and Edges

During the execution of DFSALL, each vertex v of the input graph has one of three states:

- ***new*** if DFS(v) has not been called, that is, if $clock < v.pre$;
- ***active*** if DFS(v) has been called but has not returned, that is, if $v.pre \le clock < v.post$;
- ***finished*** if DFS(v) has returned, that is, if $v.post \le clock$.

Because starting and finishing times correspond to pushes and pops on the recursion stack, a vertex is active if and only if it is on the recursion stack. It follows that the active nodes always comprise a directed path in G.

[2] Confusingly, *both* of these orders are sometimes called "depth-first ordering". Please don't do that.

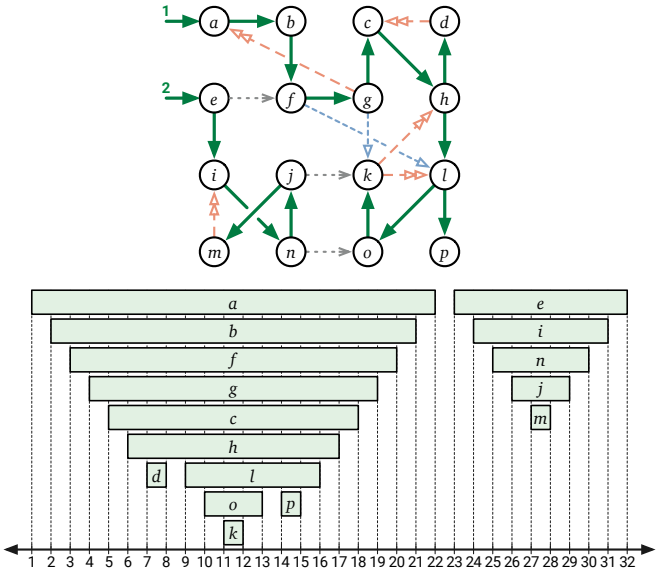

Figure 6.4. A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeinjm* and the postordering *dkoplhcgfbamjnie*. Forest edges are solid; dashed edges are explained in Figure 6.5.

The edges of the input graph fall into four different classes, depending on how their active intervals intersect. Fix your favorite edge $u \to v$.

- If v is new when DFS(u) begins, then DFS(v) must be called during the execution of DFS(u), either directly or through some intermediate recursive calls. In either case, u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.
  - If DFS(u) calls DFS(v) directly, then $u = v.parent$ and $u \to v$ is called a ***tree edge***.
  - Otherwise, $u \to v$ is called a ***forward edge***.
- If v is active when DFS(u) begins, then v is already on the recursion stack, which implies the opposite nesting order $v.pre < u.pre < u.post < v.post$. Moreover, G must contain a directed path from v to u. Edges satisfying this condition are called ***back edges***.
- If v is finished when DFS(u) begins, we immediately have $v.post < u.pre$. Edges satisfying this condition are called ***cross edges***.
- Finally, the fourth ordering $u.post < v.pre$ is impossible.

These edge classes are illustrated in Figure 6.5. Again, the actual classification of edges depends on the order in which DFSALL considers vertices and the order in which DFS considers the edges leaving each vertex.

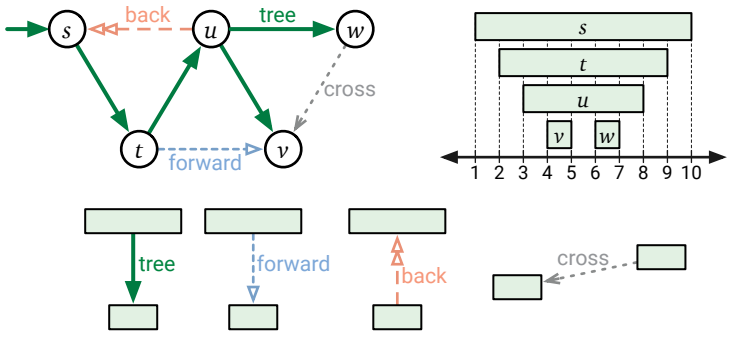

Figure 6.5. Classification of edges by depth-first search.

Finally, the following key lemma characterizes ancestors and descendants in any depth-first forest according to vertex states during the traversal.

Lemma 6.1. *Fix an arbitrary depth-first traversal of any directed graph G. The following statements are equivalent for all vertices u and v of G.*

(a) u is an ancestor of v in the depth-first forest.
(b) $u.pre \le v.pre < v.post \le u.post$.
(c) Just after DFS(v) is called, u is active.
(d) Just before DFS(u) is called, there is a path from u to v in which every vertex (including u and v) is new.

Proof: First, suppose u is an ancestor of v in the depth-first forest. Then by definition there is a path P of tree edges u to v. By induction on the path length, we have $u.pre \le w.pre < w.post \le u.post$ for every vertex w in P, and thus every vertex in P is new before DFS(u) is called. In particular, we have $u.pre \le v.pre < v.post \le u.post$, which implies that u is active while DFS(v) is executing.

Because parent pointers correspond to recursive calls, $u.pre \le v.pre < v.post \le u.post$ implies that u is an ancestor of v.

Suppose u is active just after DFS(v) is called. Then $u.pre \le v.pre < v.post \le u.post$, which implies that there is a path of (zero or more) tree edges from u, through the intermediate nodes on the recursion stack (if any), to v.

Finally, suppose u is not an ancestor of v. Fix an arbitrary path P from u to v, let x be the first vertex in P that is not a descendant of u, and let w be the predecessor of x in P. The edge $w \to x$ guarantees that $x.pre < w.post$, and $w.post < u.post$ because w is a descendant of u, so $x.pre < u.post$. It follows that $x.pre < u.pre$, because otherwise x would be a descendant of u. Because active intervals are properly nested, there are only two possibilities:

- If $u.post < x.post$, then x is active when DFS(u) is called.
- If $x.post < u.pre$, then x is already finished when DFS(u) is called.

We conclude that *every* path from u to v contains a vertex that is not new when DFS(u) is called. □

6.2 Detecting Cycles

A ***directed acyclic graph*** or ***dag*** is a directed graph with no directed cycles. Any vertex in a dag that has no incoming vertices is called a ***source***; any vertex with no outgoing edges is called a ***sink***. An isolated vertex with no incident edges at all is both a source and a sink. Every dag has at least one source and one sink, but may have more than one of each. For example, in the graph with n vertices but no edges, every vertex is a source and every vertex is a sink.

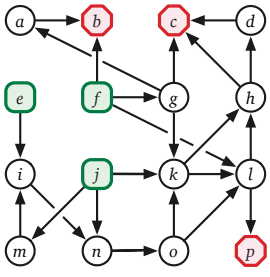

Figure 6.6. A directed acyclic graph. Vertices e, f, and j are sources; vertices b, c, and p are sinks.

Recall from our earlier case analysis that if $u.post < v.post$ for any edge $u \to v$, the graph contains a directed path from v to u, and therefore contains a directed *cycle* through the edge $u \to v$. Thus, we can determine whether a given directed graph G is a dag in $O(V + E)$ time by computing a postordering of the vertices and then checking each edge by brute force.

Alternatively, instead of numbering the vertices, we can explicitly maintain the status of each vertex and immediately return FALSE if we ever discover an edge to an active vertex. This algorithm also runs in $O(V + E)$ time; see Figure 6.7.

```
IsAcyclic(G):
  for all vertices v
      v.status ← NEW
  for all vertices v
      if v.status = NEW
          if IsAcyclicDFS(v) = FALSE
              return FALSE
  return TRUE
```

```
IsAcyclicDFS(v):
  v.status ← ACTIVE
  for each edge v→w
      if w.status = ACTIVE
          return FALSE
      else if w.status = NEW
          if IsAcyclicDFS(w) = FALSE
              return FALSE
  v.status ← FINISHED
  return TRUE
```

Figure 6.7. A linear-time algorithm to determine if a graph is acyclic.