Figure 3.3.[1] The `previsit` and `postvisit` procedures are optional, meant for performing operations on a vertex when it is first discovered and also when it is being left for the last time. We will soon see some creative uses for them.

More immediately, we need to confirm that `explore` always works correctly. It certainly does not venture too far, because it only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v. But does it find *all* vertices reachable from v? Well, if there is some u that it misses, choose any path from v to u, and look at the last vertex on that path that the procedure actually visited. Call this node z, and let w be the node immediately after it on the same path.

So z was visited but w was not. This is a contradiction: while the `explore` procedure was at node z, it would have noticed w and moved on to it.

Incidentally, this pattern of reasoning arises often in the study of graphs and is in essence a streamlined induction. A more formal inductive proof would start by framing a hypothesis, such as "for any $k \geq 0$, all nodes within k hops from v get visited." The base case is as usual trivial, since v is certainly visited. And the general case—showing that if all nodes k hops away are visited, then so are all nodes $k+1$ hops away—is precisely the same point we just argued.

Figure 3.4 shows the result of running `explore` on our earlier example graph, starting at node A, and breaking ties in alphabetical order whenever there is a choice of nodes to visit. The solid edges are those that were actually traversed, each of which was elicited by a call to `explore` and led to the discovery of a new vertex. For instance, while B was being visited, the edge $B - E$ was noticed and, since E was as yet unknown, was traversed via a call to `explore`(E). These solid edges form a tree (a connected graph with no cycles) and are therefore called *tree edges*. The dotted edges were ignored because they led back to familiar terrain, to vertices previously visited. They are called *back edges*.

3.2.2 Depth-first search

The `explore` procedure visits only the portion of the graph reachable from its starting point. To examine the rest of the graph, we need to restart the procedure elsewhere, at some vertex that has not yet been visited. The algorithm of Figure 3.5, called *depth-first search* (DFS), does this repeatedly until the entire graph has been traversed.

The first step in analyzing the running time of DFS is to observe that each vertex is `explore`'d just once, thanks to the `visited` array (the chalk marks). During the exploration of a vertex, there are the following steps:

1. Some fixed amount of work—marking the spot as visited, and the `pre`/`postvisit`.

2. A loop in which adjacent edges are scanned, to see if they lead somewhere new.

[1] As with many of our graph algorithms, this one applies to both undirected and directed graphs. In such cases, we adopt the *directed* notation for edges, (x, y). If the graph is undirected, then each of its edges should be thought of as existing in both directions: (x, y) and (y, x).

Figure 3.4 The result of `explore`(A) on the graph of Figure 3.2.

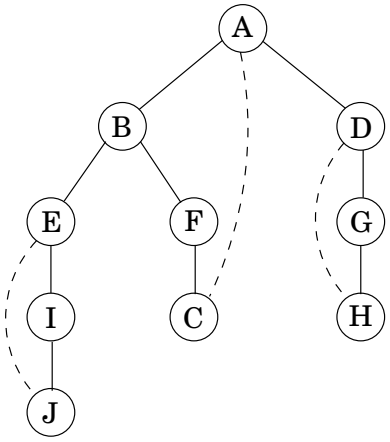


Figure 3.5 Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

This loop takes a different amount of time for each vertex, so let's consider all vertices together. The total work done in step 1 is then $O(|V|)$. In step 2, over the course of the entire DFS, each edge $\{x, y\} \in E$ is examined exactly *twice*, once during `explore`(x) and once during `explore`(y). The overall time for step 2 is therefore $O(|E|)$ and so the depth-first search has a running time of $O(|V| + |E|)$, linear in the size of its input. This is as efficient as we could possibly hope for, since it takes this long even just to read the adjacency list.

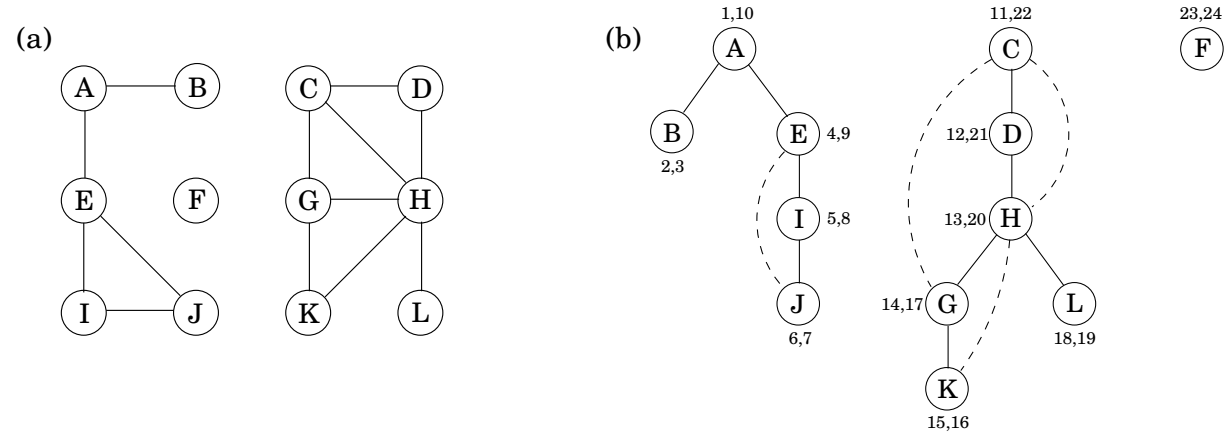
Figure 3.6 shows the outcome of depth-first search on a 12-node graph, once again breaking ties alphabetically (ignore the pairs of numbers for the time being). The outer loop of DFS calls `explore` three times, on A, C, and finally F. As a result, there are three trees, each rooted at one of these starting points. Together they constitute a *forest*.

3.2.3 Connectivity in undirected graphs

An undirected graph is *connected* if there is a path between any pair of vertices. The graph of Figure 3.6 is *not* connected because, for instance, there is no path from A to K. However, it does have three disjoint connected regions, corresponding to the following sets of vertices:

$$\{A, B, E, I, J\} \qquad \{C, D, G, H, K, L\} \qquad \{F\}$$

Figure 3.6 (a) A 12-node graph. (b) DFS search forest.



These regions are called *connected components*: each of them is a subgraph that is internally connected but has no edges to the remaining vertices. When `explore` is started at a particular vertex, it identifies precisely the connected component containing that vertex. And each time the DFS outer loop calls `explore`, a new connected component is picked out.

Thus depth-first search is trivially adapted to check if a graph is connected and, more generally, to assign each node v an integer `ccnum`$[v]$ identifying the connected component to which it belongs. All it takes is

```
procedure previsit(v)
ccnum[v] = cc
```

where `cc` needs to be initialized to zero and to be incremented each time the DFS procedure calls `explore`.

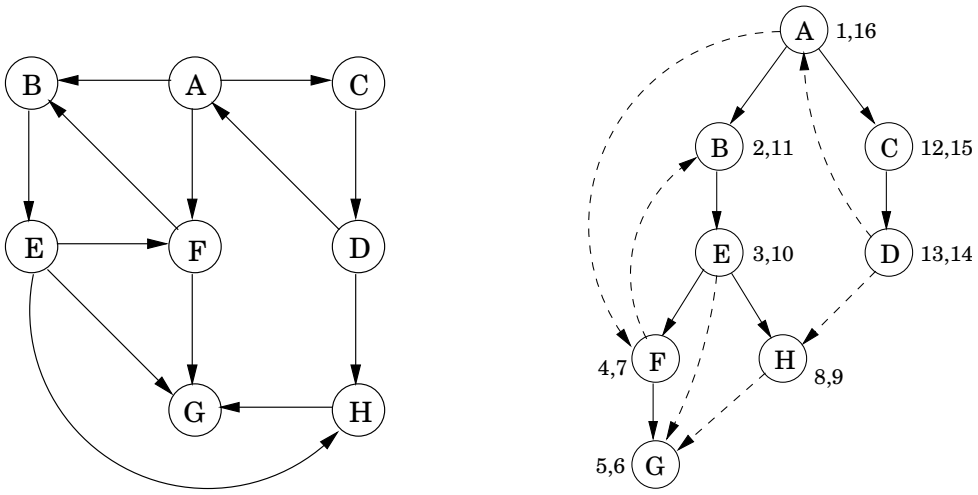
3.2.4 Previsit and postvisit orderings

We have seen how depth-first search—a few unassuming lines of code—is able to uncover the connectivity structure of an undirected graph in just linear time. But it is far more versatile than this. In order to stretch it further, we will collect a little more information during the exploration process: for each node, we will note down the times of two important events, the moment of first discovery (corresponding to `previsit`) and that of final departure (`postvisit`). Figure 3.6 shows these numbers for our earlier example, in which there are 24 events. The fifth event is the discovery of I. The 21st event consists of leaving D behind for good.

One way to generate arrays `pre` and `post` with these numbers is to define a simple counter `clock`, initially set to 1, which gets updated as follows.

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1
```

Figure 3.7 DFS on a directed graph.




```
procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

These timings will soon take on larger significance. Meanwhile, you might have noticed from Figure 3.4 that:

Property *For any nodes u and v, the two intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained within the other.*

Why? Because $[\text{pre}(u), \text{post}(u)]$ is essentially the time during which vertex u was on the stack. The last-in, first-out behavior of a stack explains the rest.

3.3 Depth-first search in directed graphs

3.3.1 Types of edges

Our depth-first search algorithm can be run verbatim on directed graphs, taking care to traverse edges only in their prescribed directions. Figure 3.7 shows an example and the search tree that results when vertices are considered in lexicographic order.

In further analyzing the directed case, it helps to have terminology for important relationships between nodes of a tree. A is the *root* of the search tree; everything else is its *descendant*. Similarly, E has descendants F, G, and H, and conversely, is an *ancestor* of these three nodes. The family analogy is carried further: C is the *parent* of D, which is its *child*.

For undirected graphs we distinguished between tree edges and nontree edges. In the directed case, there is a slightly more elaborate taxonomy:

Tree edges are actually part of the DFS forest.

Forward edges lead from a node to a *nonchild* descendant in the DFS tree.

Back edges lead to an ancestor in the DFS tree.

Cross edges lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already postvisited).

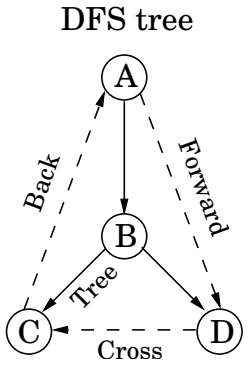


Figure 3.7 has two forward edges, two back edges, and two cross edges. Can you spot them?

Ancestor and descendant relationships, as well as edge types, can be read off directly from `pre` and `post` numbers. Because of the depth-first exploration strategy, vertex u is an ancestor of vertex v exactly in those cases where u is discovered first and v is discovered during `explore(`u`)`. This is to say $\texttt{pre}(u) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(u)$, which we can depict pictorially as two nested intervals:

$$\underset{u}{[} \quad \underset{v}{[} \quad \underset{v}{]} \quad \underset{u}{]}$$

The case of descendants is symmetric, since u is a descendant of v if and only if v is an ancestor of u. And since edge categories are based entirely on ancestor-descendant relationships, it follows that they, too, can be read off from `pre` and `post` numbers. Here is a summary of the various possibilities for an edge (u, v):

`pre`/`post` *ordering for* (u, v)	*Edge type*
$\underset{u}{[} \; \underset{v}{[} \; \underset{v}{]} \; \underset{u}{]}$	Tree/forward
$\underset{v}{[} \; \underset{u}{[} \; \underset{u}{]} \; \underset{v}{]}$	Back
$\underset{v}{[} \; \underset{v}{]} \; \underset{u}{[} \; \underset{u}{]}$	Cross

You can confirm each of these characterizations by consulting the diagram of edge types. Do you see why no other orderings are possible?

3.3.2 Directed acyclic graphs

A *cycle* in a directed graph is a circular path $v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0$. Figure 3.7 has quite a few of them, for example, $B \to E \to F \to B$. A graph without cycles is *acyclic*. It turns out we can test for acyclicity in linear time, with a single depth-first search.