Chapter 3

Decompositions of graphs

3.1 Why graphs?

A wide range of problems can be expressed with clarity and precision in the concise pictorial language of graphs. For instance, consider the task of coloring a political map. What is the minimum number of colors needed, with the obvious restriction that neighboring countries should have different colors? One of the difficulties in attacking this problem is that the map itself, even a stripped-down version like Figure 3.1(a), is usually cluttered with irrelevant information: intricate boundaries, border posts where three or more countries meet, open seas, and meandering rivers. Such distractions are absent from the mathematical object of Figure 3.1(b), a graph with one *vertex* for each country (1 is Brazil, 11 is Argentina) and *edges* between neighbors. It contains exactly the information needed for coloring, and nothing more. The precise goal is now to assign a color to each vertex so that no edge has endpoints of the same color.

Graph coloring is not the exclusive domain of map designers. Suppose a university needs to schedule examinations for all its classes and wants to use the fewest time slots possible. The only constraint is that two exams cannot be scheduled concurrently if some student will be taking both of them. To express this problem as a graph, use one vertex for each exam and put an edge between two vertices if there is a conflict, that is, if there is somebody taking both endpoint exams. Think of each time slot as having its own color. Then, assigning time slots is exactly the same as coloring this graph!

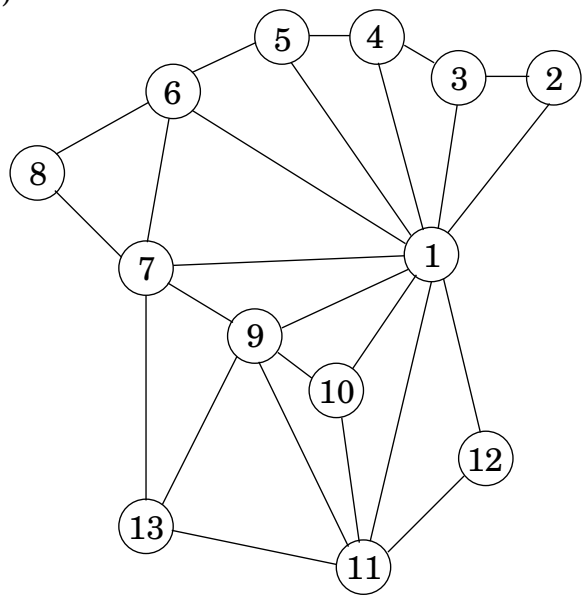
Some basic operations on graphs arise with such frequency, and in such a diversity of contexts, that a lot of effort has gone into finding efficient procedures for them. This chapter is devoted to some of the most fundamental of these algorithms—those that uncover the basic connectivity structure of a graph.

Formally, a graph is specified by a set of vertices (also called *nodes*) V and by edges E between select pairs of vertices. In the map example, $V = \{1, 2, 3, \ldots, 13\}$ and E includes, among many other edges, $\{1, 2\}, \{9, 11\}$, and $\{7, 13\}$. Here an edge between x and y specifically means "x shares a border with y." This is a symmetric relation—it implies also that y shares a border with x—and we denote it using set notation, $e = \{x, y\}$. Such edges are *undirected*

Figure 3.1 (a) A map and (b) its graph.

(a)

(b)



and are part of an *undirected graph*.

Sometimes graphs depict relations that do not have this reciprocity, in which case it is necessary to use edges with directions on them. There can be *directed edges e from x to y* (written $e = (x, y)$), or from y to x (written (y, x)), or both. A particularly enormous example of a *directed graph* is the graph of all links in the World Wide Web. It has a vertex for each site on the Internet, and a directed edge (u, v) whenever site u has a link to site v: in total, billions of nodes and edges! Understanding even the most basic connectivity properties of the Web is of great economic and social interest. Although the size of this problem is daunting, we will soon see that a lot of valuable information about the structure of a graph can, happily, be determined in just linear time.

3.1.1 How is a graph represented?

We can represent a graph by an *adjacency matrix*; if there are $n = |V|$ vertices $v_1, \ldots, v_n$, this is an $n \times n$ array whose (i, j)th entry is

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

For undirected graphs, the matrix is symmetric since an edge $\{u, v\}$ can be taken in either direction.

The biggest convenience of this format is that the presence of a particular edge can be checked in constant time, with just one memory access. On the other hand the matrix takes

up $O(n^2)$ space, which is wasteful if the graph does not have very many edges.

An alternative representation, with size proportional to the number of edges, is the *adjacency list*. It consists of $|V|$ linked lists, one per vertex. The linked list for vertex u holds the names of vertices to which u has an outgoing edge—that is, vertices v for which $(u, v) \in E$. Therefore, each edge appears in exactly one of the linked lists if the graph is directed or two of the lists if the graph is undirected. Either way, the total size of the data structure is $O(|E|)$. Checking for a particular edge (u, v) is no longer constant time, because it requires sifting through u's adjacency list. But it is easy to iterate through all neighbors of a vertex (by running down the corresponding linked list), and, as we shall soon see, this turns out to be a very useful operation in graph algorithms. Again, for undirected graphs, this representation has a symmetry of sorts: v is in u's adjacency list if and only if u is in v's adjacency list.

How big is your graph?

Which of the two representations, adjacency matrix or adjacency list, is better? Well, it depends on the relationship between $|V|$, the number of nodes in the graph, and $|E|$, the number of edges. $|E|$ can be as small as $|V|$ (if it gets much smaller, then the graph degenerates—for example, has isolated vertices), or as large as $|V|^2$ (when all possible edges are present). When $|E|$ is close to the upper limit of this range, we call the graph *dense*. At the other extreme, if $|E|$ is close to $|V|$, the graph is *sparse*. As we shall see in this chapter and the next two chapters, *exactly where $|E|$ lies in this range is usually a crucial factor in selecting the right graph algorithm*.

Or, for that matter, in selecting the graph representation. If it is the World Wide Web graph that we wish to store in computer memory, we should think twice before using an adjacency matrix: at the time of writing, search engines know of about eight billion vertices of this graph, and hence the adjacency matrix would take up *dozens of millions of terabits*. Again at the time we write these lines, it is not clear that there is enough computer memory in the whole world to achieve this. (And waiting a few years until there *is* enough memory is unwise: the Web will grow too and will probably grow faster.)

With adjacency lists, representing the World Wide Web becomes feasible: there are only a few dozen billion hyperlinks in the Web, and each will occupy a few bytes in the adjacency list. You can carry a device that stores the result, a terabyte or two, in your pocket (it may soon fit in your earring, but by that time the Web will have grown too).

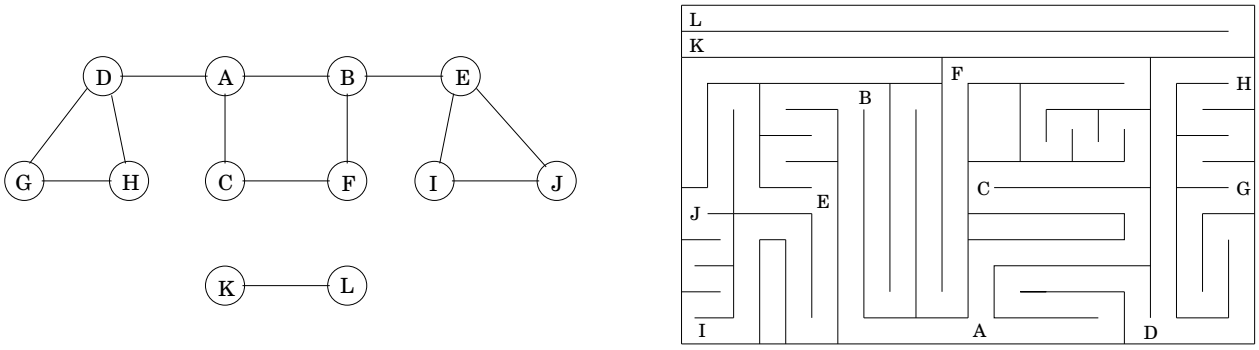
The reason why adjacency lists are so much more effective in the case of the World Wide Web is that the Web is very sparse: the average Web page has hyperlinks to only about half a dozen other pages, out of the billions of possibilities.

3.2 Depth-first search in undirected graphs

3.2.1 Exploring mazes

Depth-first search is a surprisingly versatile linear-time procedure that reveals a wealth of information about a graph. The most basic question it addresses is,

Figure 3.2 Exploring a graph is rather like navigating a maze.



What parts of the graph are reachable from a given vertex?

To understand this task, try putting yourself in the position of a computer that has just been given a new graph, say in the form of an adjacency list. This representation offers just one basic operation: finding the neighbors of a vertex. With only this primitive, the reachability problem is rather like exploring a labyrinth (Figure 3.2). You start walking from a fixed place and whenever you arrive at any junction (vertex) there are a variety of passages (edges) you can follow. A careless choice of passages might lead you around in circles or might cause you to overlook some accessible part of the maze. Clearly, you need to record some intermediate information during exploration.

This classic challenge has amused people for centuries. Everybody knows that all you need to explore a labyrinth is a ball of string and a piece of chalk. The chalk prevents looping, by marking the junctions you have already visited. The string always takes you back to the starting place, enabling you to return to passages that you previously saw but did not yet investigate.

How can we simulate these two primitives, chalk and string, on a computer? The chalk marks are easy: for each vertex, maintain a Boolean variable indicating whether it has been visited already. As for the ball of string, the correct cyberanalog is a *stack*. After all, the exact role of the string is to offer two primitive operations—*unwind* to get to a new junction (the stack equivalent is to *push* the new vertex) and *rewind* to return to the previous junction (*pop* the stack).

Instead of explicitly maintaining a stack, we will do so implicitly via recursion (which is implemented using a stack of activation records). The resulting algorithm is shown in 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.

[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 3.3 Finding all nodes reachable from a particular node.

```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

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*.

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

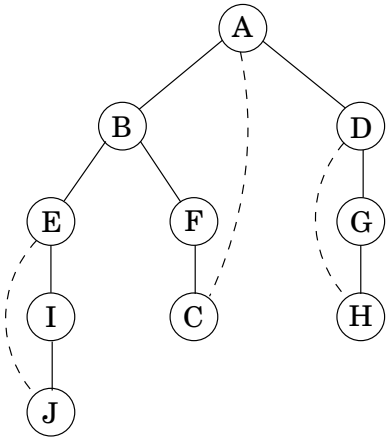


Figure 3.5 Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

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

3.2.2 Depth-first search

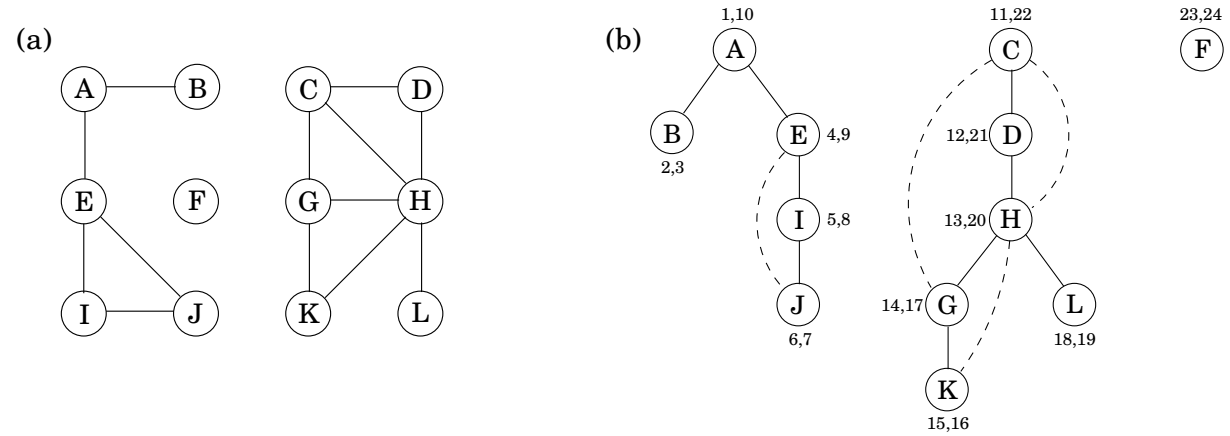
The `explore` procedure visits only the portion of the graph reachable from its starting point. To examine the rest of the graph, we need to restart the procedure elsewhere, at some vertex that has not yet been visited. The algorithm of Figure 3.5, called *depth-first search* (DFS), does this repeatedly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

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

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



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:

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

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

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

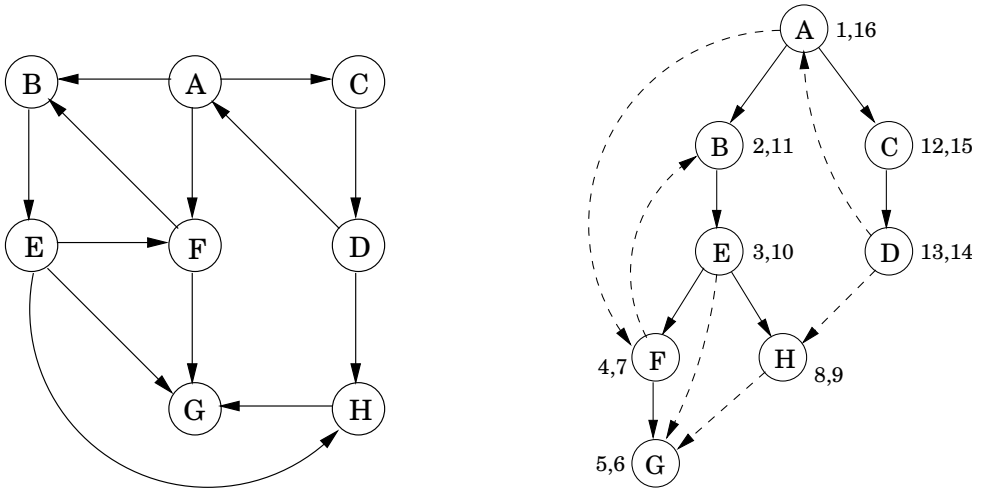
3.3.1 Types of edges

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:

Figure 3.7 DFS on a directed graph.



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).

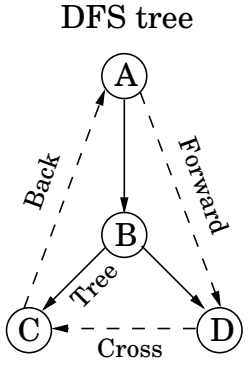


Figure 3.7 has two forward edges, two back edges, and two cross edges. Can you spot them?

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*
$[_u \; [_v \;]_v \;]_u$	Tree/forward
$[_v \; [_u \;]_u \;]_v$	Back
$[_v \;]_v \; [_u \;]_u$	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.

Property *A directed graph has a cycle if and only if its depth-first search reveals a back edge.*

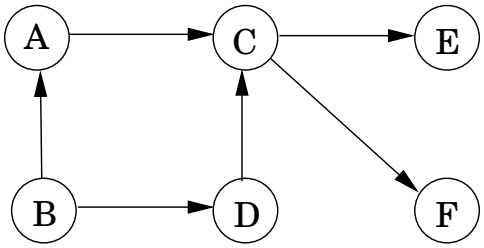
Proof. One direction is quite easy: if (u, v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.

Conversely, if the graph has a cycle $v_0 \to v_1 \to \cdots \to v_k \to v_0$, look at the *first* node on this cycle to be discovered (the node with the lowest `pre` number). Suppose it is v_i. All the other v_j on the cycle are reachable from it and will therefore be its descendants in the search tree. In particular, the edge $v_{i-1} \to v_i$ (or $v_k \to v_0$ if $i = 0$) leads from a node to its ancestor and is thus by definition a back edge. ∎

Directed acyclic graphs, or *dags* for short, come up all the time. They are good for modeling relations like causalities, hierarchies, and temporal dependencies. For example, suppose that you need to perform many tasks, but some of them cannot begin until certain others are completed (you have to wake up before you can get out of bed; you have to be out of bed, but not yet dressed, to take a shower; and so on). The question then is, what is a valid order in which to perform the tasks?

Such constraints are conveniently represented by a directed graph in which each task is a node, and there is an edge from u to v if u is a precondition for v. In other words, before performing a task, all the tasks pointing to it must be completed. If this graph has a cycle, there is no hope: no ordering can possibly work. If on the other hand the graph is a dag, we would like if possible to *linearize* (or *topologically sort*) it, to order the vertices one after the other in such a way that each edge goes from an earlier vertex to a later vertex, so that all precedence constraints are satisfied. In Figure 3.8, for instance, one valid ordering is B, A, D, C, E, F. (Can you spot the other three?)

Figure 3.8 A directed acyclic graph with one source, two sinks, and four possible linearizations.



What types of dags can be linearized? Simple: *All of them*. And once again depth-first search tells us exactly how to do it: simply perform tasks in *decreasing* order of their `post` numbers. After all, the only edges (u, v) in a graph for which `post`(u) <`post`(v) are back edges (recall the table of edge types on page 100)—and we have seen that a dag cannot have back edges. Therefore:

Property *In a dag, every edge leads to a vertex with a lower* `post` *number.*

This gives us a linear-time algorithm for ordering the nodes of a dag. And, together with our earlier observations, it tells us that three rather different-sounding properties—acyclicity, linearizability, and the absence of back edges during a depth-first search—are in fact one and the same thing.

Since a dag is linearized by decreasing `post` numbers, the vertex with the smallest `post` number comes last in this linearization, and it must be a *sink*—no outgoing edges. Symmetrically, the one with the highest `post` is a *source*, a node with no incoming edges.

Property *Every dag has at least one source and at least one sink.*

The guaranteed existence of a source suggests an alternative approach to linearization:

> Find a source, output it, and delete it from the graph.
>
> Repeat until the graph is empty.

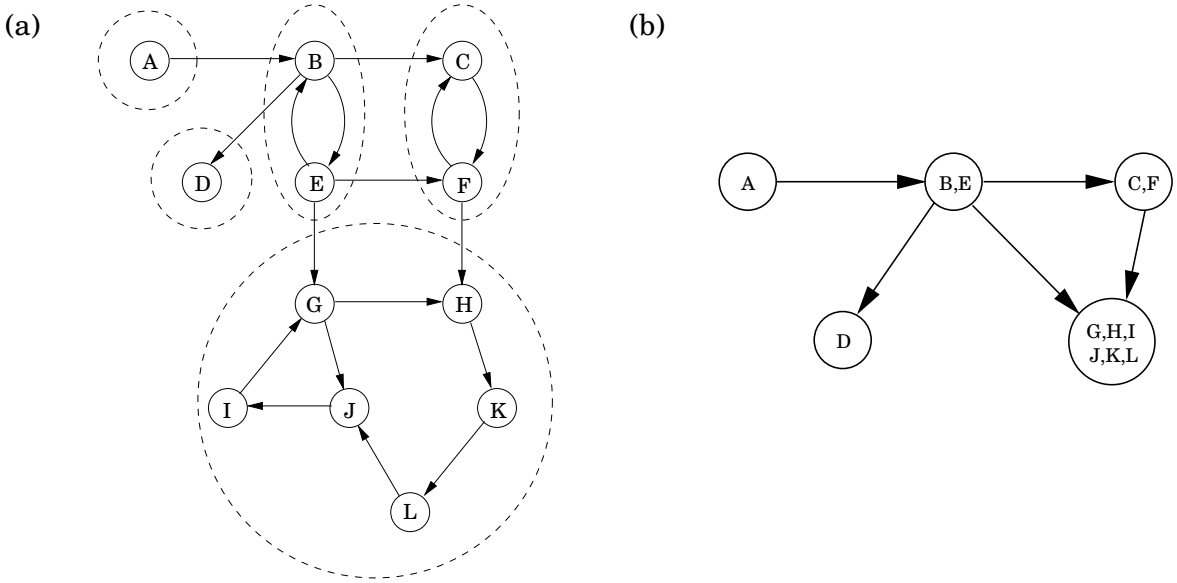
Can you see why this generates a valid linearization for any dag? What happens if the graph has cycles? And, how can this algorithm be implemented in linear time? (Exercise 3.14.)

3.4 Strongly connected components

3.4.1 Defining connectivity for directed graphs

Connectivity in undirected graphs is pretty straightforward: a graph that is not connected can be decomposed in a natural and obvious manner into several connected components (Fig-

Figure 3.9 (a) A directed graph and its strongly connected components. (b) The meta-graph.



ure 3.6 is a case in point). As we saw in Section 3.2.3, depth-first search does this handily, with each restart marking a new connected component.

In directed graphs, connectivity is more subtle. In some primitive sense, the directed graph of Figure 3.9(a) is "connected"—it can't be "pulled apart," so to speak, without breaking edges. But this notion is hardly interesting or informative. The graph cannot be considered connected, because for instance there is no path from G to B or from F to A. The right way to define connectivity for directed graphs is this:

> Two nodes u and v of a directed graph are *connected* if there is a path from u to v *and* a path from v to u.

This relation partitions V into disjoint sets (Exercise 3.30) that we call *strongly connected components*. The graph of Figure 3.9(a) has five of them.

Now shrink each strongly connected component down to a single meta-node, and draw an edge from one meta-node to another if there is an edge (in the same direction) between their respective components (Figure 3.9(b)). The resulting *meta-graph* must be a dag. The reason is simple: a cycle containing several strongly connected components would merge them all into a single, strongly connected component. Restated,

Property *Every directed graph is a dag of its strongly connected components.*

This tells us something important: The connectivity structure of a directed graph is two-tiered. At the top level we have a dag, which is a rather simple structure—for instance, it

can be linearized. If we want finer detail, we can look inside one of the nodes of this dag and examine the full-fledged strongly connected component within.

3.4.2 An efficient algorithm

The decomposition of a directed graph into its strongly connected components is very informative and useful. It turns out, fortunately, that it can be found in linear time by making further use of depth-first search. The algorithm is based on some properties we have already seen but which we will now pinpoint more closely.

Property 1 *If the* `explore` *subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.*

Therefore, if we call `explore` on a node that lies somewhere in a *sink* strongly connected component (a strongly connected component that is a sink in the meta-graph), then we will retrieve exactly that component. Figure 3.9 has two sink strongly connected components. Starting `explore` at node K, for instance, will completely traverse the larger of them and then stop.

This suggests a way of finding one strongly connected component, but still leaves open two major problems: (A) how do we find a node that we know for sure lies in a sink strongly connected component and (B) how do we continue once this first component has been discovered?

Let's start with problem (A). There is not an easy, direct way to pick out a node that is guaranteed to lie in a sink strongly connected component. But there is a way to get a node in a *source* strongly connected component.

Property 2 *The node that receives the highest* `post` *number in a depth-first search must lie in a source strongly connected component.*

This follows from the following more general property.

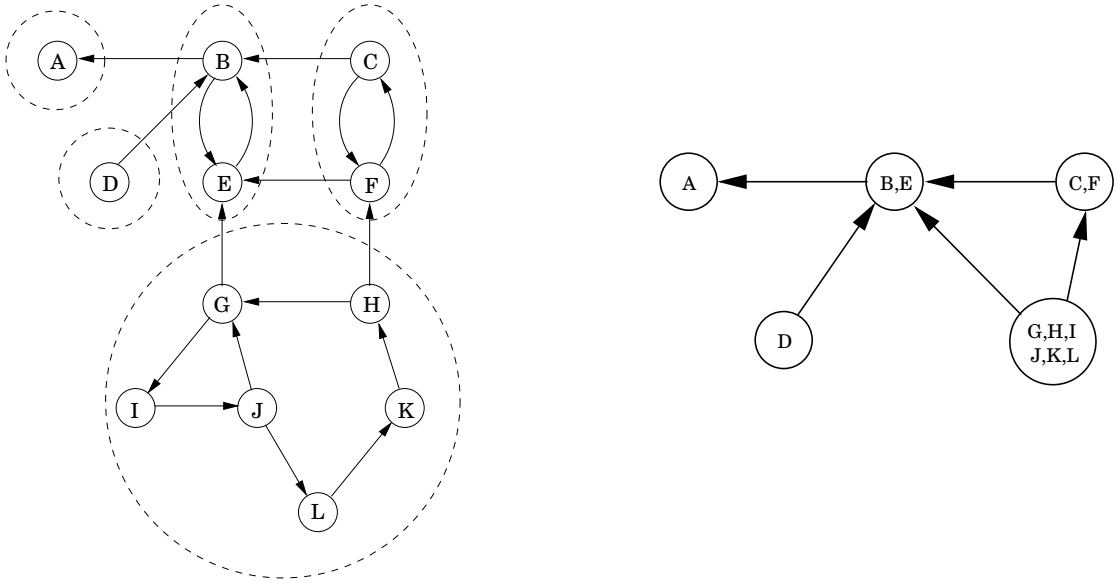
Property 3 *If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest* `post` *number in C is bigger than the highest* `post` *number in C'.*

Proof. In proving Property 3, there are two cases to consider. If the depth-first search visits component C before component C', then clearly all of C and C' will be traversed before the procedure gets stuck (see Property 1). Therefore the first node visited in C will have a higher `post` number than any node of C'. On the other hand, if C' gets visited first, then the depth-first search will get stuck after seeing all of C' but before seeing any of C, in which case the property follows immediately. ∎

Property 3 can be restated as saying that *the strongly connected components can be linearized by arranging them in decreasing order of their highest* `post` *numbers*. This is a generalization of our earlier algorithm for linearizing dags; in a dag, each node is a singleton strongly connected component.

Property 2 helps us find a node in the source strongly connected component of G. However, what we need is a node in the *sink* component. Our means seem to be the opposite of

Figure 3.10 The *reverse* of the graph from Figure 3.9.



our needs! But consider the *reverse* graph G^R, the same as G but with all edges reversed (Figure 3.10). G^R has exactly the same strongly connected components as G (why?). So, if we do a depth-first search of G^R, the node with the highest `post` number will come from a source strongly connected component in G^R, which is to say a sink strongly connected component in G. We have solved problem (A)!

Onward to problem (B). How do we continue after the first sink component is identified? The solution is also provided by Property 3. Once we have found the first strongly connected component and deleted it from the graph, the node with the highest `post` number among those remaining will belong to a sink strongly connected component of whatever remains of G. Therefore we can keep using the `post` numbering from our initial depth-first search on G^R to successively output the second strongly connected component, the third strongly connected component, and so on. The resulting algorithm is this.

1. Run depth-first search on G^R.
2. Run the undirected connected components algorithm (from Section 3.2.3) on G, and during the depth-first search, process the vertices in decreasing order of their `post` numbers from step 1.

This algorithm is linear-time, only the constant in the linear term is about twice that of straight depth-first search. (Question: How does one construct an adjacency list representation of G^R in linear time? And how, in linear time, does one order the vertices of G by decreasing `post` values?)

Let's run this algorithm on the graph of Figure 3.9. If step 1 considers vertices in lexicographic order, then the ordering it sets up for the second step (namely, decreasing `post` numbers in the depth-first search of G^R) is: $G, I, J, L, K, H, D, C, F, B, E, A$. Then step 2 peels off components in the following sequence: $\{G, H, I, J, K, L\}, \{D\}, \{C, F\}, \{B, E\}, \{A\}$.

Crawling fast

All this assumes that the graph is neatly given to us, with vertices numbered 1 to n and edges tucked in adjacency lists. The realities of the World Wide Web are very different. The nodes of the Web graph are not known in advance, and they have to be discovered one by one during the process of search. And, of course, recursion is out of the question.

Still, crawling the Web is done by algorithms very similar to depth-first search. An explicit stack is maintained, containing all nodes that have been discovered (as endpoints of hyperlinks) but not yet explored. In fact, this "stack" is not exactly a last-in, first-out list. It gives highest priority not to the nodes that were inserted most recently (nor the ones that were inserted earliest, that would be a *breadth-first search*, see Chapter 4), but to the ones that look most "interesting"—a heuristic criterion whose purpose is to keep the stack from overflowing and, in the worst case, to leave unexplored only nodes that are very unlikely to lead to vast new expanses.

In fact, crawling is typically done by many computers running `explore` simultaneously: each one takes the next node to be explored from the top of the stack, downloads the http file (the kind of Web files that point to each other), and scans it for hyperlinks. But when a new http document is found at the end of a hyperlink, no recursive calls are made: instead, the new vertex is inserted in the central stack.

But one question remains: When we see a "new" document, how do we know that it is indeed new, that we have not seen it before in our crawl? And how do we give it a *name*, so it can be inserted in the stack and recorded as "already seen"? The answer is *by hashing*.

Incidentally, researchers have run the strongly connected components algorithm on the Web and have discovered some very interesting structure.