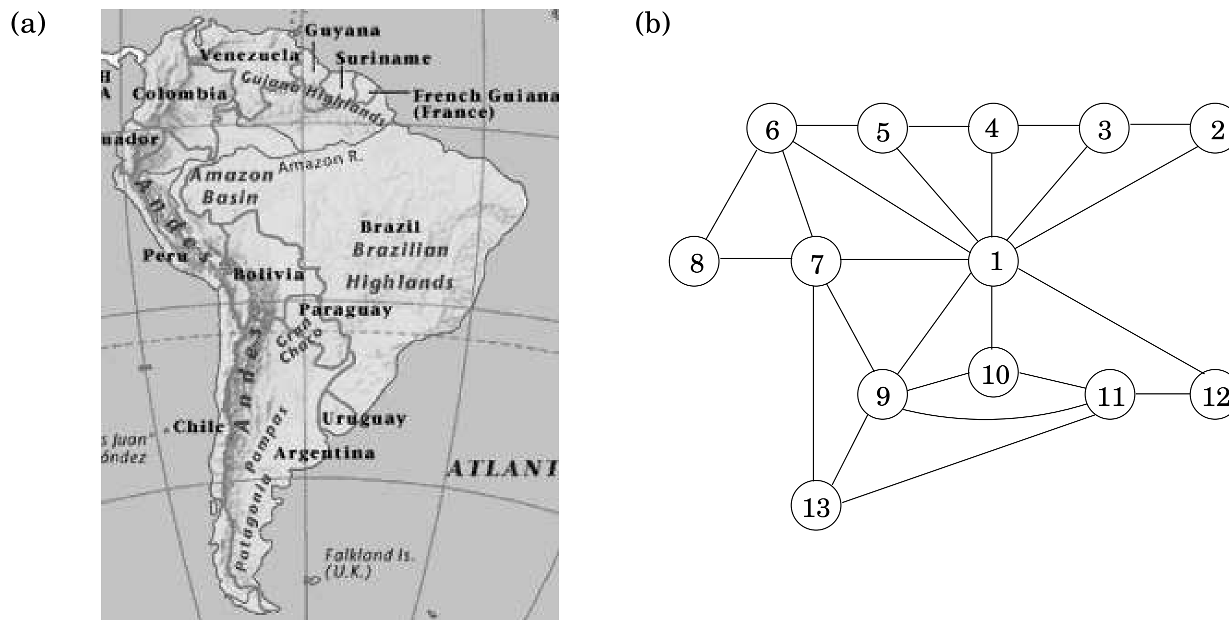
Figure 1.1 A map and its graph.



Decompositions of graphs[1]

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 1.1(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 1.1(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.

This suitably abstracted graph coloring problem turns out to have many other applications. For example, 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

[1]Copyright ©2004 S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani.

devoted to some of the most fundamental of these algorithms – those which 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 $\{1, 2\}, \{9, 11\}$, 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*, and are part of an *undirected graph*.

Sometimes graphs depict relations which 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 URL, and a directed edge (u, v) whenever URL u has a link to URL v: in total, billions of nodes and edges! Understanding even the most basic connectivity properties of the web is of great economic and sociological 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.

2 How is a graph represented?

A common computer representation of a graph is as an *adjacency matrix*. If there are n vertices $v_1, \ldots, v_n$, this is an $n \times n$ matrix 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: 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.

A more economical representation, with size proportional to the number of edges, is the *adjacency list*. It is also very simple: for each vertex, there is a list of outgoing edges. 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, and this turns out to be a very useful operation in graph algorithms.

3 Depth-first search in undirected graphs

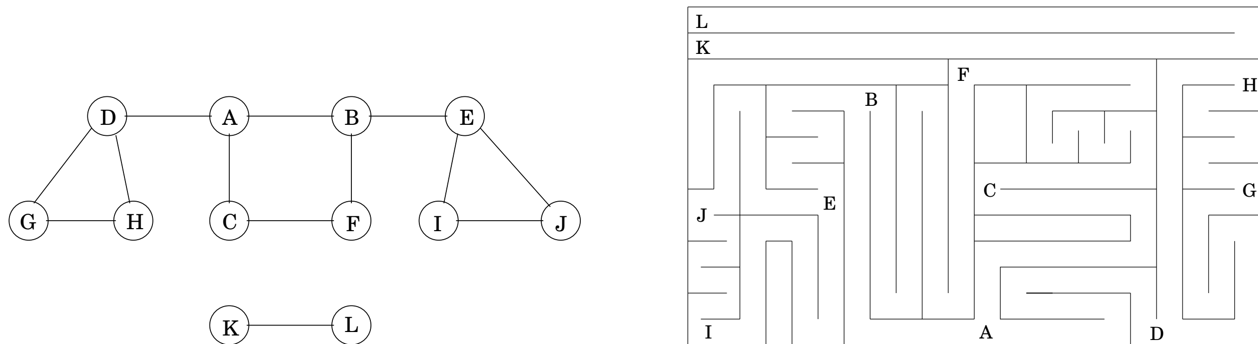
3.1 Exploring mazes

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

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

To understand this task, try putting yourself in the position of a computer which has just been given a new graph, say in the form of an adjacency list. This representation offers just one

Figure 3.1 Exploring a graph is rather like navigating a maze.



Algorithm 3.1 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) then explore(u)
postvisit(v)
```

basic operation: finding the neighbors of a vertex. With only this primitive, the reachability problem is rather like exploring a labyrinth (Figure 3.1). 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, and it is well known that you can do it with only a ball of string and a piece of chalk.[2] The chalk prevents looping, by marking the junctions you have already visited. The string always leads back to the starting place, enabling you to return to passages which you previously saw but did not yet investigate.

How can these be simulated on a computer? The chalk marks are easy to handle: 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 into 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 result is Algorithm 3.1. The `previsit` and `postvisit` procedures are optional, meant for performing operations on a vertex when

[2]Actually the Greek hero Theseus did it with just the string, but we won't be quite so macho.

Algorithm 3.2 Depth-first search.

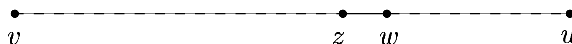
procedure dfs(G)

```
for all v ∈ V:
   visited(v) = false

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

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 work 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 which 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.

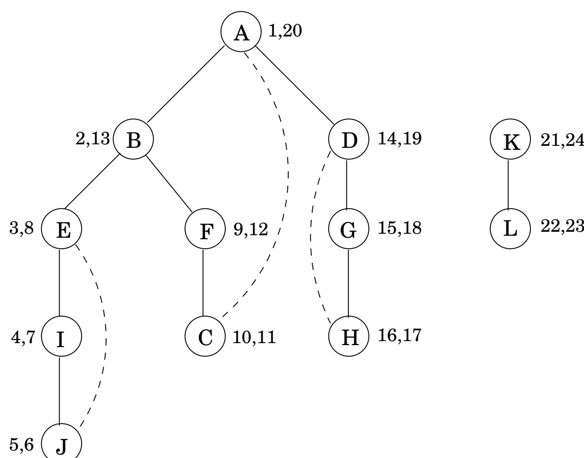
3.2 Depth-first search

An undirected graph is *connected* if there is a path between any pair of vertices. Figure 3.1 is *not* connected because, for instance, there is no path from A to K. However, it does have two disjoint connected regions. These regions are called *connected components*: each of them is a subgraph which is internally connected but has no edges to the remaining vertices. A call to `explore` starting at vertex A will visit the connected component containing A, since this is exactly the set of vertices reachable from A. To move on to the rest of the graph, we need to restart the exploration procedure elsewhere, at some vertex which has not yet been visited. Algorithm 3.2 does this repeatedly, until the entire graph has been traversed.

Depth-first search is highly efficient. Notice that because of the `visited` array (the chalk marks), `explore` gets called only once for each vertex. Assuming that `pre-` and `postvisit` are each $O(1)$, the total time needed by everything other than the inner loop of `explore` is $O(|V|)$. In this inner loop, each edge (x, y) is checked twice, once during `explore(`x`)` and once during `explore(`y`)`. The overall time spent in this loop is therefore $O(|E|)$ and so depth-first search has a running time of $O(|V| + |E|)$, linear in the size of its input.

Figure 3.2 shows the outcome of depth-first search on our earlier example graph, considering vertices in lexicographic order. The solid edges are those which 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 collection of *trees* (connected graphs with no cycles) and are therefore called *tree edges*. The dotted ones were

Figure 3.2 The result of DFS on the graph of Figure 3.1.



ignored because they led back to familiar terrain, to vertices previously visited. They are *back edges*. Each DFS tree is *rooted* at the vertex at which exploration began and spans the connected component of the graph containing that vertex. Together these trees constitute a *forest*.[3]

3.3 Previsit and postvisit orderings

In Figure 3.2 two times are noted for each node – the moment of first discovery (corresponding to `previsit`) and that of final departure (`postvisit`) – and these times are arranged chronologically. All in all, the depth-first search of this graph consists of 24 events. The 9th event is the discovery of F. The 18th event consists of leaving G 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
```

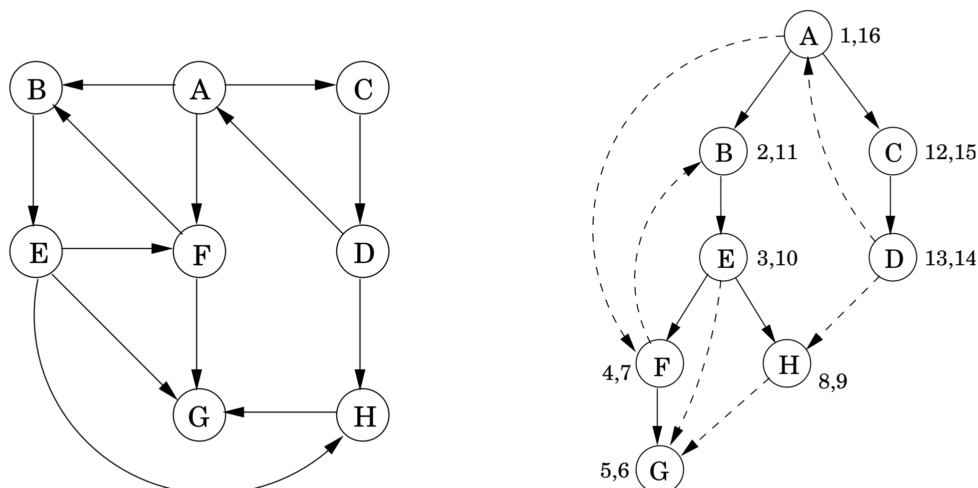
At present this numbering is merely an explanatory device, but later these same arrays will take on larger significance. You might have noticed from Figure 3.2 that

(1) For any u, v, the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are either disjoint or one is contained within the other.

Why? $[\text{pre}(u), \text{post}(u)]$ 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] This unabashedly florid terminology is common in mathematics. Study graph algorithms a bit further and you'll encounter blossoms, branchings, and even sunflowers.

Figure 4.1 DFS on a directed graph.



3.4 Connected components

We have already seen that each time depth-first search calls the `explore` subroutine, exactly one connected component is retrieved. We can therefore very easily tell whether the graph is connected: if `dfs` needs to call `explore` more than once, it isn't. More generally, we can assign to each node u an integer `ccnum`$[u]$ identifying the connected component to which it belongs.

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

Here `cc` is initialized to zero, and is incremented each time the `dfs` procedure calls `explore`.

4 Depth-first search in directed graphs

4.1 Types of edges

Our depth-first search (DFS) algorithm can be run verbatim on directed graphs, taking care to traverse edges only in their prescribed directions. Figure 4.1 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, 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 non-tree 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 *non-child* descendant in the DFS tree.

Back edges lead to an ancestor in the DFS tree.

Figure 4.2 DFS on a directed graph partitions the edges into four categories.

Type of edge	`pre/post` criterion for edge (u, v)	Part of search tree?
Tree	$\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$	Yes
Forward	$\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$	No
Back	$\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$	No
Cross	$\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$	No

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

Figure 4.1 has two forward edges, two back edges, and two cross edges. Can you spot them?

We now come to an interesting use of `pre` and `post` numbers: they can uniquely identify ancestor and descendant relationships. Because of the manner in which the DFS forest is constructed, vertex u is an ancestor of vertex v exactly in those cases where u is discovered first and v is discovered during `explore(`u`)`. In other words,

(2) Node u is an ancestor of node v if and only if $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$.

This also takes care of descendants, since u is a descendant of v if and only if v is an ancestor of u. Therefore we can classify edges based (almost) entirely on `pre/post` information; see Figure 4.2. Do you see that every edge must fall into one of these groups?

4.2 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 4.1 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, just by running depth-first search.

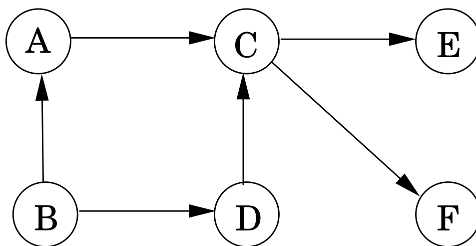
(3) A directed graph has a cycle if and only if its depth-first search reveals a back edge.

One direction is quite easy to see: 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 lowest `pre` number). Suppose it is v_i. All the other v_j on the cycle will be its descendants in the search tree. And (v_{i-1}, v_i) (or (v_k, v_0) if $i = 0$) is 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. In one typical scheduling application, various tasks need to be performed, 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 and undressed before you can take a shower; and so on). The question then is, what is a valid order in which to perform the tasks?

Figure 4.3 A directed acyclic graph with one source, two sinks, and four possible topological orderings.



The 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 to *topologically sort* it, to number the vertices (that is, order the tasks) in such a way that these numbers are increasing along each edge, so that all precedence constraints are respected. In Figure 4.3, for instance, one valid ordering is B, A, D, C, E, F. Can you spot the other three?

Is it always possible to topologically sort a dag? The answer is yes, and once again depth-first search tells us exactly how to do it: simply perform tasks in decreasing order of their `post` numbers. As justification, we know from Figure 4.2 that the only edges $(u, v) \in E$ for which $\texttt{post}(u) < \texttt{post}(v)$ are back edges. And we have seen that a dag cannot have back edges. Therefore,

(4) In a dag, every edge leads to a vertex with lower `post` number.

This immediately gives us a linear-time algorithm for topologically sorting a dag.

The same argument implies that in any dag, the vertex with smallest `post` number is a *sink* – that is, it has no outgoing edges. Any such edge would necessarily lead to a vertex with higher `post` number, which have seen is impossible. Symmetrically, every dag must also have a *source*, a node with no incoming edges (can you always identify such a node just by looking at `pre` and `post` numbers?). The guaranteed existence of a source suggests another approach to topological sorting.

Find a source, output it, and delete it from the graph.

Repeat until the graph is empty.

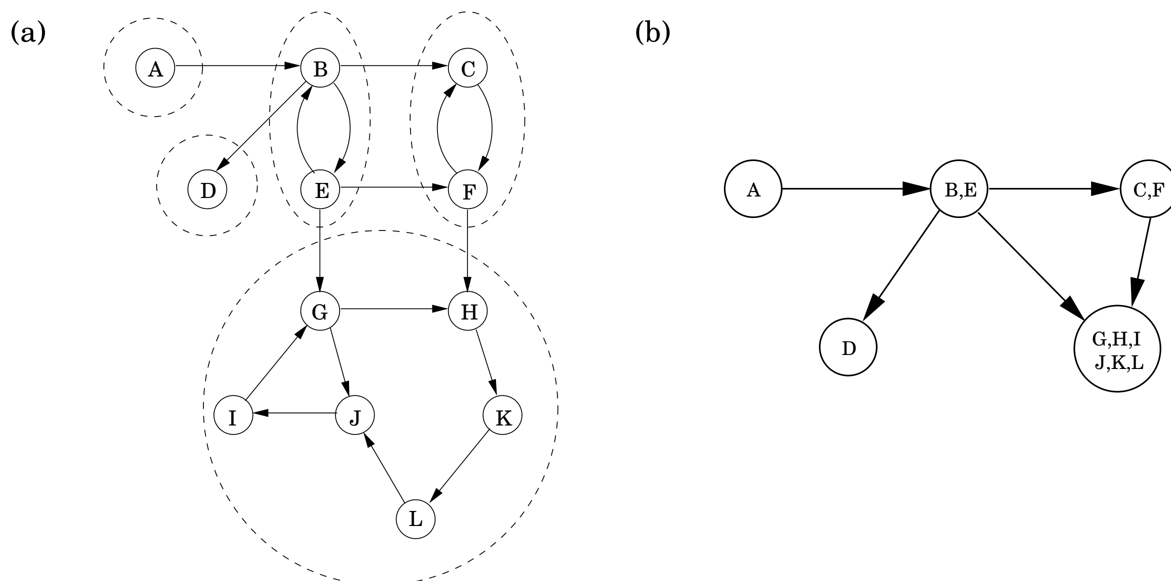
Can you see why this generates a valid topological ordering for any dag? What would happen if the graph had cycles? How could this algorithm be implemented in linear time?

5 Strongly connected components

5.1 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 5.1 (a) A directed graph and its strongly connected components. (b) The meta-graph.



ure 3.1 is a case in point). As we have seen, 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 5.1(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 only meaningful 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 between nodes is reflexive, symmetric, and transitive, so it is an equivalence relation. As such, it partitions V into disjoint sets which we call *strongly connected components*. The graph of Figure 5.1(a) has five of them.

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 5.1(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,

(5) Every directed graph is a dag of its strongly connected components.

Thus 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, we know it must always have a source and a sink. 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.

5.2 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 which 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 can find a node which lies somewhere in a *sink* strongly connected component (a strongly connected component which is a sink in the meta-graph), then calling `explore` on this node will retrieve exactly that component. Figure 5.1 has two sink strongly connected components. Starting `explore` at node K, for instance, will completely traverse the larger of them.

This suggests a way of finding one strongly connected component, but still leaves open two major problems: (A) how do we find a node which 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 which 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 which receives the highest `post` number in 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'.

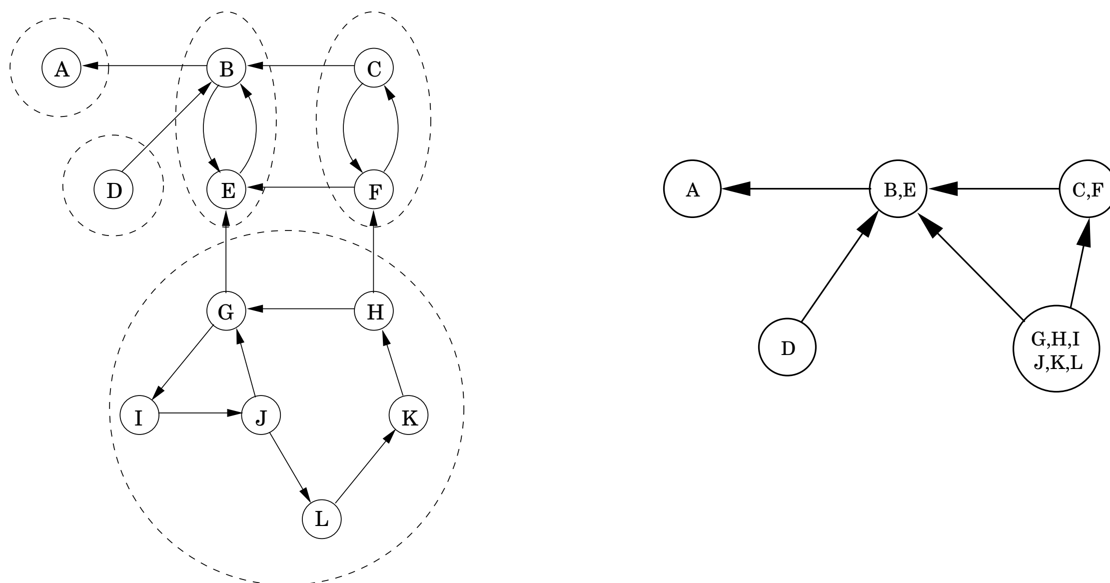
In proving Property 3, there are two cases to consider. If the depth-first search visits component C before component C', then clearly all of C and C' will be traversed before the procedure gets stuck (see Property 1). Therefore the first node visited in C will have a higher `post` number than any node of C'. On the other hand, if C' gets visited first, then 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 topologically sorted by arranging them in decreasing order of their highest* `post` *numbers*. This is a generalization of our earlier algorithm for topologically ordering dags; in a dag, each node is a singleton strongly connected component.

Property 2 gives an indirect solution to Problem (A). Consider the *reverse* graph G^R, the same as G but with all edges reversed (Figure 5.2). 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 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

Figure 5.2 The *reverse* of the graph from Figure 5.1.



component and deleted it from the graph, the node with highest `post` number among those remaining will belong to a sink strongly connected 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 (which we developed earlier in this chapter) 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 5.1. 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: $\{G, H, I, J, K, L\}, \{D\}, \{C, F\}, \{B, E\}, \{A\}$.