most $\sum_u n_v = 2m$. This proves the desired $O(m+n)$ bound on the running time of DFS.

(3.13) *The above implementation of the DFS algorithm runs in time $O(m+n)$ (i.e., linear in the input size), if the graph is given by the adjacency list representation.*

Finding the Set of All Connected Components

In the previous section we talked about how one can use BFS (or DFS) to find all connected components of a graph. We start with an arbitrary node s, and we use BFS (or DFS) to generate its connected component. We then find a node v (if any) that was not visited by the search from s and iterate, using BFS (or DFS) starting from v to generate its connected component—which, by (3.8), will be disjoint from the component of s. We continue in this way until all nodes have been visited.

Although we earlier expressed the running time of BFS and DFS as $O(m+n)$, where m and n are the total number of edges and nodes in the graph, both BFS and DFS in fact spend work only on edges and nodes in the connected component containing the starting node. (They never see any of the other nodes or edges.) Thus the above algorithm, although it may run BFS or DFS a number of times, only spends a constant amount of work on a given edge or node in the iteration when the connected component it belongs to is under consideration. Hence the overall running time of this algorithm is still $O(m+n)$.

3.4 Testing Bipartiteness: An Application of Breadth-First Search

Recall the definition of a bipartite graph: it is one where the node set V can be partitioned into sets X and Y in such a way that every edge has one end in X and the other end in Y. To make the discussion a little smoother, we can imagine that the nodes in the set X are colored red, and the nodes in the set Y are colored blue. With this imagery, we can say a graph is bipartite if it is possible to color its nodes red and blue so that every edge has one red end and one blue end.

The Problem

In the earlier chapters, we saw examples of bipartite graphs. Here we start by asking: What are some natural examples of a nonbipartite graph, one where no such partition of V is possible?

Clearly a triangle is not bipartite, since we can color one node red, another one blue, and then we can't do anything with the third node. More generally, consider a cycle C of odd length, with nodes numbered $1, 2, 3, \ldots, 2k, 2k+1$. If we color node 1 red, then we must color node 2 blue, and then we must color node 3 red, and so on—coloring odd-numbered nodes red and even-numbered nodes blue. But then we must color node $2k+1$ red, and it has an edge to node 1, which is also red. This demonstrates that there's no way to partition C into red and blue nodes as required. More generally, if a graph G simply *contains* an odd cycle, then we can apply the same argument; thus we have established the following.

(3.14) *If a graph G is bipartite, then it cannot contain an odd cycle.*

It is easy to recognize that a graph is bipartite when appropriate sets X and Y (i.e., red and blue nodes) have actually been identified for us; and in many settings where bipartite graphs arise, this is natural. But suppose we encounter a graph G with no annotation provided for us, and we'd like to determine for ourselves whether it is bipartite—that is, whether there exists a partition into red and blue nodes, as required. How difficult is this? We see from (3.14) that an odd cycle is one simple "obstacle" to a graph's being bipartite. Are there other, more complex obstacles to bipartitness?

Designing the Algorithm

In fact, there is a very simple procedure to test for bipartiteness, and its analysis can be used to show that odd cycles are the *only* obstacle. First we assume the graph G is connected, since otherwise we can first compute its connected components and analyze each of them separately. Next we pick any node $s \in V$ and color it red; there is no loss in doing this, since s must receive some color. It follows that all the neighbors of s must be colored blue, so we do this. It then follows that all the neighbors of *these* nodes must be colored red, their neighbors must be colored blue, and so on, until the whole graph is colored. At this point, either we have a valid red/blue coloring of G, in which every edge has ends of opposite colors, or there is some edge with ends of the same color. In this latter case, it seems clear that there's nothing we could have done: G simply is not bipartite. We now want to argue this point precisely and also work out an efficient way to perform the coloring.

The first thing to notice is that the coloring procedure we have just described is essentially identical to the description of BFS: we move outward from s, coloring nodes as soon as we first encounter them. Indeed, another way to describe the coloring algorithm is as follows: we perform BFS, coloring

s red, all of layer L_1 blue, all of layer L_2 red, and so on, coloring odd-numbered layers blue and even-numbered layers red.

We can implement this on top of BFS, by simply taking the implementation of BFS and adding an extra array `Color` over the nodes. Whenever we get to a step in BFS where we are adding a node v to a list $L[i+1]$, we assign `Color`$[v]$ = `red` if $i+1$ is an even number, and `Color`$[v]$ = `blue` if $i+1$ is an odd number. At the end of this procedure, we simply scan all the edges and determine whether there is any edge for which both ends received the same color. Thus, the total running time for the coloring algorithm is $O(m+n)$, just as it is for BFS.

Analyzing the Algorithm

We now prove a claim that shows this algorithm correctly determines whether G is bipartite, and it also shows that we can find an odd cycle in G whenever it is not bipartite.

(3.15) *Let G be a connected graph, and let $L_1, L_2, \ldots$ be the layers produced by BFS starting at node s. Then exactly one of the following two things must hold.*

(i) There is no edge of G joining two nodes of the same layer. In this case G is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue.

(ii) There is an edge of G joining two nodes of the same layer. In this case, G contains an odd-length cycle, and so it cannot be bipartite.

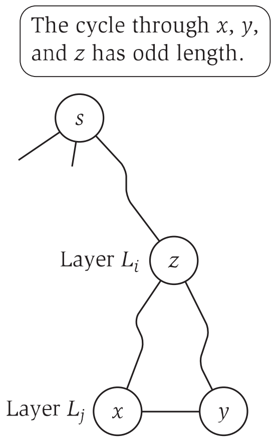


Figure 3.6 If two nodes x and y in the same layer are joined by an edge, then the cycle through x, y, and their lowest common ancestor z has odd length, demonstrating that the graph cannot be bipartite.

Proof. First consider case (i), where we suppose that there is no edge joining two nodes of the same layer. By (3.4), we know that every edge of G joins nodes either in the same layer or in adjacent layers. Our assumption for case (i) is precisely that the first of these two alternatives never happens, so this means that *every* edge joins two nodes in adjacent layers. But our coloring procedure gives nodes in adjacent layers the opposite colors, and so every edge has ends with opposite colors. Thus this coloring establishes that G is bipartite.

Now suppose we are in case (ii); why must G contain an odd cycle? We are told that G contains an edge joining two nodes of the same layer. Suppose this is the edge $e = (x, y)$, with $x, y \in L_j$. Also, for notational reasons, recall that L_0 ("layer 0") is the set consisting of just s. Now consider the BFS tree T produced by our algorithm, and let z be the node whose layer number is as large as possible, subject to the condition that z is an ancestor of both x and y in T; for obvious reasons, we can call z the *lowest common ancestor* of x and y. Suppose $z \in L_i$, where $i < j$. We now have the situation pictured in Figure 3.6. We consider the cycle C defined by following the z-x path in T, then the edge e, and then the y-z path in T. The length of this cycle is $(j - i) + 1 + (j - i)$, adding the length of its three parts separately; this is equal to $2(j - i) + 1$, which is an odd number. ■

3.5 Connectivity in Directed Graphs

Thus far, we have been looking at problems on undirected graphs; we now consider the extent to which these ideas carry over to the case of directed graphs.

Recall that in a directed graph, the edge (u, v) has a direction: it goes from u to v. In this way, the relationship between u and v is asymmetric, and this has qualitative effects on the structure of the resulting graph. In Section 3.1, for example, we discussed the World Wide Web as an instance of a large, complex directed graph whose nodes are pages and whose edges are hyperlinks. The act of browsing the Web is based on following a sequence of edges in this directed graph; and the directionality is crucial, since it's not generally possible to browse "backwards" by following hyperlinks in the reverse direction.

At the same time, a number of basic definitions and algorithms have natural analogues in the directed case. This includes the adjacency list representation and graph search algorithms such as BFS and DFS. We now discuss these in turn.

Representing Directed Graphs

In order to represent a directed graph for purposes of designing algorithms, we use a version of the adjacency list representation that we employed for undirected graphs. Now, instead of each node having a single list of neighbors, each node has two lists associated with it: one list consists of nodes *to which* it has edges, and a second list consists of nodes *from which* it has edges. Thus an algorithm that is currently looking at a node u can read off the nodes reachable by going one step forward on a directed edge, as well as the nodes that would be reachable if one went one step in the reverse direction on an edge from u.

The Graph Search Algorithms

Breadth-first search and depth-first search are almost the same in directed graphs as they are in undirected graphs. We will focus here on BFS. We start at a node s, define a first layer of nodes to consist of all those to which s has an edge, define a second layer to consist of all additional nodes to which these first-layer nodes have an edge, and so forth. In this way, we discover nodes layer by layer as they are reached in this outward search from s, and the nodes in layer j are precisely those for which the shortest path *from* s has exactly j edges. As in the undirected case, this algorithm performs at most constant work for each node and edge, resulting in a running time of $O(m + n)$.

It is important to understand what this directed version of BFS is computing. In directed graphs, it is possible for a node s to have a path to a node t even though t has no path to s; and what directed BFS is computing is the set of all nodes t with the property that s has a path to t. Such nodes may or may not have paths back to s.

There is a natural analogue of depth-first search as well, which also runs in linear time and computes the same set of nodes. It is again a recursive procedure that tries to explore as deeply as possible, in this case only following edges according to their inherent direction. Thus, when DFS is at a node u, it recursively launches a depth-first search, in order, for each node to which u has an edge.

Suppose that, for a given node s, we wanted the set of nodes with paths to s, rather than the set of nodes to which s has paths. An easy way to do this would be to define a new directed graph, G^{rev}, that we obtain from G simply by reversing the direction of every edge. We could then run BFS or DFS in G^{rev}; a node has a path *from* s in G^{rev} if and only if it has a path *to* s in G.

Strong Connectivity

Recall that a directed graph is *strongly connected* if, for every two nodes u and v, there is a path from u to v and a path from v to u. It's worth also formulating some terminology for the property at the heart of this definition; let's say that two nodes u and v in a directed graph are *mutually reachable* if there is a path from u to v and also a path from v to u. (So a graph is strongly connected if every pair of nodes is mutually reachable.)

Mutual reachability has a number of nice properties, many of them stemming from the following simple fact.

(3.16) *If u and v are mutually reachable, and v and w are mutually reachable, then u and w are mutually reachable.*

Proof. To construct a path from u to w, we first go from u to v (along the path guaranteed by the mutual reachability of u and v), and then on from v to w (along the path guaranteed by the mutual reachability of v and w). To construct a path from w to u, we just reverse this reasoning: we first go from w to v (along the path guaranteed by the mutual reachability of v and w), and then on from v to u (along the path guaranteed by the mutual reachability of u and v). ■

There is a simple linear-time algorithm to test if a directed graph is strongly connected, implicitly based on (3.16). We pick any node s and run BFS in G starting from s. We then also run BFS starting from s in G^{rev}. Now, if one of these two searches fails to reach every node, then clearly G is not strongly connected. But suppose we find that s has a path to every node, and that every node has a path to s. Then s and v are mutually reachable for every v, and so it follows that *every* two nodes u and v are mutually reachable: s and u are mutually reachable, and s and v are mutually reachable, so by (3.16) we also have that u and v are mutually reachable.

By analogy with connected components in an undirected graph, we can define the *strong component* containing a node s in a directed graph to be the set of all v such that s and v are mutually reachable. If one thinks about it, the algorithm in the previous paragraph is really computing the strong component containing s: we run BFS starting from s both in G and in G^{rev}; the set of nodes reached by *both* searches is the set of nodes with paths to *and* from s, and hence this set is the strong component containing s.

There are further similarities between the notion of connected components in undirected graphs and strong components in directed graphs. Recall that connected components naturally partitioned the graph, since any two were either identical or disjoint. Strong components have this property as well, and for essentially the same reason, based on (3.16).

(3.17) *For any two nodes s and t in a directed graph, their strong components are either identical or disjoint.*

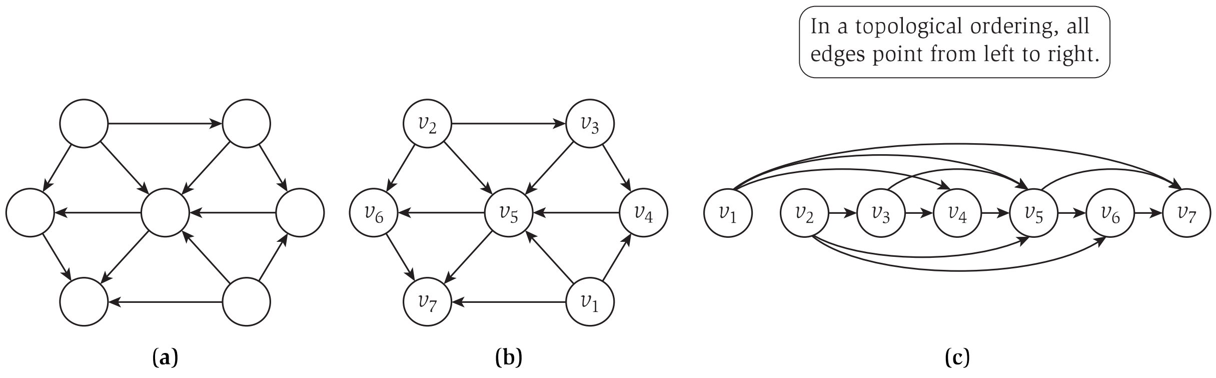
Proof. Consider any two nodes s and t that are mutually reachable; we claim that the strong components containing s and t are identical. Indeed, for any node v, if s and v are mutually reachable, then by (3.16), t and v are mutually reachable as well. Similarly, if t and v are mutually reachable, then again by (3.16), s and v are mutually reachable.

On the other hand, if s and t are not mutually reachable, then there cannot be a node v that is in the strong component of each. For if there were such a node v, then s and v would be mutually reachable, and v and t would be mutually reachable, so from (3.16) it would follow that s and t were mutually reachable. ■

In fact, although we will not discuss the details of this here, with more work it is possible to compute the strong components for all nodes in a total time of $O(m + n)$.

3.6 Directed Acyclic Graphs and Topological Ordering

If an undirected graph has no cycles, then it has an extremely simple structure: each of its connected components is a tree. But it is possible for a directed graph to have no (directed) cycles and still have a very rich structure. For example, such graphs can have a large number of edges: if we start with the node



In a topological ordering, all edges point from left to right.
v_1
v_2
v_3
v_4
v_5
v_6
v_7
(a)
(b)
(c)

Figure 3.7 (a) A directed acyclic graph. (b) The same DAG with a topological ordering, specified by the labels on each node. (c) A different drawing of the same DAG, arranged so as to emphasize the topological ordering.

set $\{1, 2, \ldots, n\}$ and include an edge (i, j) whenever $i < j$, then the resulting directed graph has $\binom{n}{2}$ edges but no cycles.

If a directed graph has no cycles, we call it—naturally enough—a *directed acyclic graph*, or a *DAG* for short. (The term *DAG* is typically pronounced as a word, not spelled out as an acronym.) In Figure 3.7(a) we see an example of a DAG, although it may take some checking to convince oneself that it really has no directed cycles.

The Problem

DAGs are a very common structure in computer science, because many kinds of dependency networks of the type we discussed in Section 3.1 are acyclic. Thus DAGs can be used to encode *precedence relations* or *dependencies* in a natural way. Suppose we have a set of tasks labeled $\{1, 2, \ldots, n\}$ that need to be performed, and there are dependencies among them stipulating, for certain pairs i and j, that i must be performed before j. For example, the tasks may be courses, with prerequisite requirements stating that certain courses must be taken before others. Or the tasks may correspond to a pipeline of computing jobs, with assertions that the output of job i is used in determining the input to job j, and hence job i must be done before job j.

We can represent such an interdependent set of tasks by introducing a node for each task, and a directed edge (i, j) whenever i must be done before j. If the precedence relation is to be at all meaningful, the resulting graph G must be a DAG. Indeed, if it contained a cycle C, there would be no way to do any of the tasks in C: since each task in C cannot begin until some other one completes, no task in C could ever be done, since none could be done first.

Let's continue a little further with this picture of DAGs as precedence relations. Given a set of tasks with dependencies, it would be natural to seek a valid order in which the tasks could be performed, so that all dependencies are respected. Specifically, for a directed graph G, we say that a *topological ordering* of G is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j), we have $i < j$. In other words, all edges point "forward" in the ordering. A topological ordering on tasks provides an order in which they can be safely performed; when we come to the task v_j, all the tasks that are required to precede it have already been done. In Figure 3.7(b) we've labeled the nodes of the DAG from part (a) with a topological ordering; note that each edge indeed goes from a lower-indexed node to a higher-indexed node.

In fact, we can view a topological ordering of G as providing an immediate "proof" that G has no cycles, via the following.

(3.18) *If G has a topological ordering, then G is a DAG.*

Proof. Suppose, by way of contradiction, that G has a topological ordering $v_1, v_2, \ldots, v_n$, and also has a cycle C. Let v_i be the lowest-indexed node on C, and let v_j be the node on C just before v_i—thus (v_j, v_i) is an edge. But by our choice of i, we have $j > i$, which contradicts the assumption that $v_1, v_2, \ldots, v_n$ was a topological ordering. ∎

The proof of acyclicity that a topological ordering provides can be very useful, even visually. In Figure 3.7(c), we have drawn the same graph as in (a) and (b), but with the nodes laid out in the topological ordering. It is immediately clear that the graph in (c) is a DAG since each edge goes from left to right.

Computing a Topological Ordering The main question we consider here is the converse of (3.18): Does every DAG have a topological ordering, and if so, how do we find one efficiently? A method to do this for every DAG would be very useful: it would show that for any precedence relation on a set of tasks without cycles, there is an efficiently computable order in which to perform the tasks.

Designing and Analyzing the Algorithm

In fact, the converse of (3.18) does hold, and we establish this via an efficient algorithm to compute a topological ordering. The key to this lies in finding a way to get started: which node do we put at the beginning of the topological ordering? Such a node v_1 would need to have no incoming edges, since any such incoming edge would violate the defining property of the topological ordering, that all edges point forward. Thus, we need to prove the following fact.

(3.19) *In every DAG G, there is a node v with no incoming edges.*

Proof. Let G be a directed graph in which every node has at least one incoming edge. We show how to find a cycle in G; this will prove the claim. We pick any node v, and begin following edges backward from v: since v has at least one incoming edge (u, v), we can walk backward to u; then, since u has at least one incoming edge (x, u), we can walk backward to x; and so on. We can continue this process indefinitely, since every node we encounter has an incoming edge. But after $n + 1$ steps, we will have visited some node w twice. If we let C denote the sequence of nodes encountered between successive visits to w, then clearly C forms a cycle. ■

In fact, the existence of such a node v is all we need to produce a topological ordering of G by induction. Specifically, let us claim by induction that every DAG has a topological ordering. This is clearly true for DAGs on one or two nodes. Now suppose it is true for DAGs with up to some number of nodes n. Then, given a DAG G on $n + 1$ nodes, we find a node v with no incoming edges, as guaranteed by (3.19). We place v first in the topological ordering; this is safe, since all edges out of v will point forward. Now $G-\{v\}$ is a DAG, since deleting v cannot create any cycles that weren't there previously. Also, $G-\{v\}$ has n nodes, so we can apply the induction hypothesis to obtain a topological ordering of $G-\{v\}$. We append the nodes of $G-\{v\}$ in this order after v; this is an ordering of G in which all edges point forward, and hence it is a topological ordering.

Thus we have proved the desired converse of (3.18).

(3.20) *If G is a DAG, then G has a topological ordering.*

The inductive proof contains the following algorithm to compute a topological ordering of G.

```
To compute a topological ordering of $G$:
Find a node $v$ with no incoming edges and order it first
Delete $v$ from $G$
Recursively compute a topological ordering of $G-\{v\}$
  and append this order after $v$
```

In Figure 3.8 we show the sequence of node deletions that occurs when this algorithm is applied to the graph in Figure 3.7. The shaded nodes in each iteration are those with no incoming edges; the crucial point, which is what

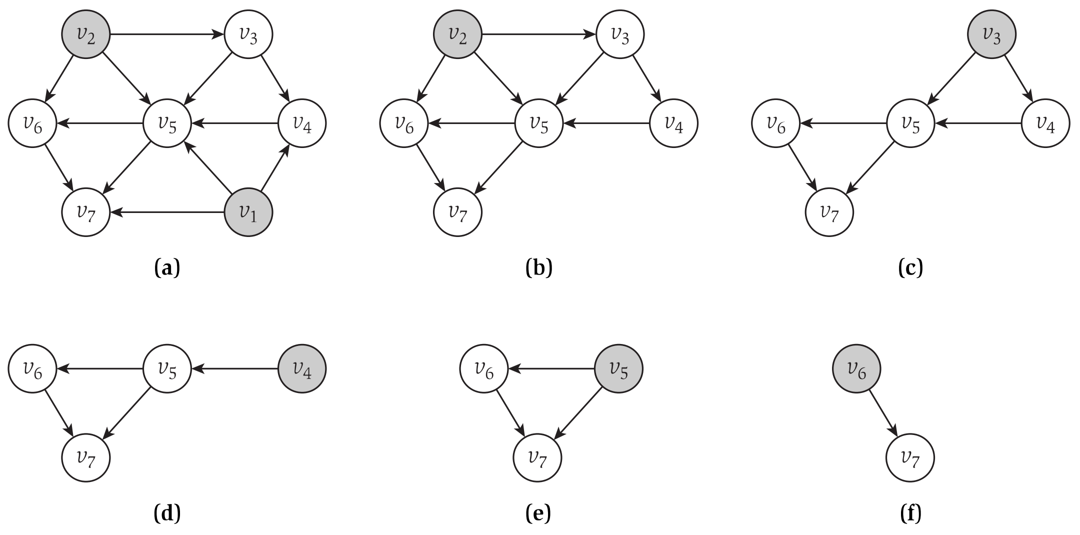


Figure 3.8 Starting from the graph in Figure 3.7, nodes are deleted one by one so as to be added to a topological ordering. The shaded nodes are those with no incoming edges; note that there is always at least one such edge at every stage of the algorithm's execution.

(3.19) guarantees, is that when we apply this algorithm to a DAG, there will always be at least one such node available to delete.

To bound the running time of this algorithm, we note that identifying a node v with no incoming edges, and deleting it from G, can be done in $O(n)$ time. Since the algorithm runs for n iterations, the total running time is $O(n^2)$.

This is not a bad running time; and if G is very dense, containing $\Theta(n^2)$ edges, then it is linear in the size of the input. But we may well want something better when the number of edges m is much less than n^2. In such a case, a running time of $O(m + n)$ could be a significant improvement over $\Theta(n^2)$.

In fact, we can achieve a running time of $O(m + n)$ using the same high-level algorithm—iteratively deleting nodes with no incoming edges. We simply have to be more efficient in finding these nodes, and we do this as follows.

We declare a node to be "active" if it has not yet been deleted by the algorithm, and we explicitly maintain two things:

(a) for each node w, the number of incoming edges that w has from active nodes; and

(b) the set S of all active nodes in G that have no incoming edges from other active nodes.

At the start, all nodes are active, so we can initialize (a) and (b) with a single pass through the nodes and edges. Then, each iteration consists of selecting a node v from the set S and deleting it. After deleting v, we go through all nodes w to which v had an edge, and subtract one from the number of active incoming edges that we are maintaining for w. If this causes the number of active incoming edges to w to drop to zero, then we add w to the set S. Proceeding in this way, we keep track of nodes that are eligible for deletion at all times, while spending constant work per edge over the course of the whole algorithm.

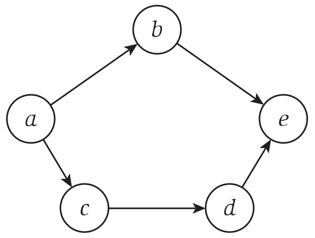


Figure 3.9 How many topological orderings does this graph have?

Solved Exercises

Solved Exercise 1

Consider the directed acyclic graph G in Figure 3.9. How many topological orderings does it have?

Solution Recall that a topological ordering of G is an ordering of the nodes as $v_1, v_2, \ldots, v_n$ so that all edges point "forward": for every edge (v_i, v_j), we have $i < j$.

So one way to answer this question would be to write down all $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ possible orderings and check whether each is a topological ordering. But this would take a while.

Instead, we think about this as follows. As we saw in the text (or reasoning directly from the definition), the first node in a topological ordering must be one that has no edge coming into it. Analogously, the last node must be one that has no edge leaving it. Thus, in every topological ordering of G, the node a must come first and the node e must come last.

Now we have to figure how the nodes b, c, and d can be arranged in the middle of the ordering. The edge (c, d) enforces the requirement that c must come before d; but b can be placed anywhere relative to these two: before both, between c and d, or after both. This exhausts all the possibilities, and so we conclude that there are three possible topological orderings:

$$a, b, c, d, e$$

$$a, c, b, d, e$$

$$a, c, d, b, e$$

Solved Exercise 2

Some friends of yours are working on techniques for coordinating groups of mobile robots. Each robot has a radio transmitter that it uses to communicate