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

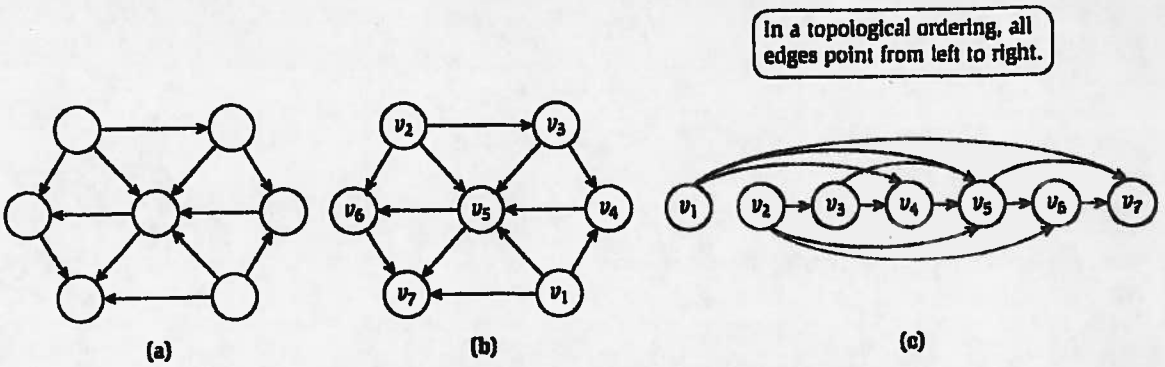


Figure 3.7 (a) A directed acyclic graph. (b) The same DAG with a topological ordering, specified by the labels on each node. (c) A different drawing of the same DAG, arranged so as to emphasize the topological ordering.

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

In fact, we can view a topological ordering of G as providing an immed "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 orde $v_1, v_2, \ldots, v_n$, and also has a cycle C. Let v_i be the lowest-indexed node or and let v_j be the node on C just before v_i—thus (v_j, v_i) is an edge. But by choice of i, we have $j > i$, which contradicts the assumption that $v_1, v_2, \ldots$ was a topological ordering. ■

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

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

Designing and Analyzing the Algorithm

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

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

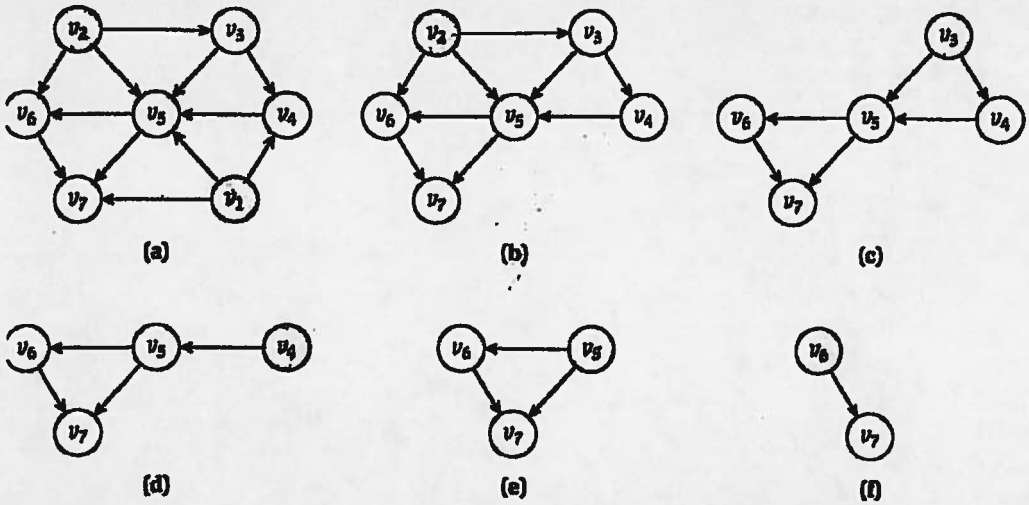


Figure 3.8 Starting from the graph in Figure 3.7, nodes are deleted one by one so as to be added to a topological ordering. The shaded nodes are those with no incoming edges; note that there is always at least one such edge at every stage of the algorithm's execution.

(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 $O(n^2)$.

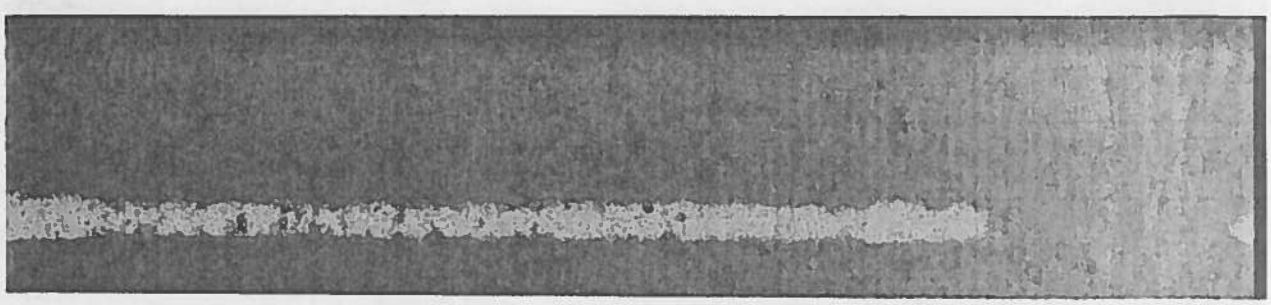
This is not a bad running time; and if G is very dense, containing $\Theta(n^2)$ edges, then it is linear in the size of the input. But we may well want something better when the number of edges m is much less than n^2. In such a case, a running time of $O(m + n)$ could be a significant improvement over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

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.