22.3-9
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.d \le u.f$.

22.3-10
Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G, together with its type. Show what modifications, if any, you need to make if G is undirected.

22.3-11
Explain how a vertex u of a directed graph can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G.

22.3-12
Show that we can use a depth-first search of an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v are in the same connected component.

22.3-13 ★
A directed graph $G = (V, E)$ is ***singly connected*** if $u \leadsto v$ implies that G contains at most one simple path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether or not a directed graph is singly connected.

22.4 Topological sort

This section shows how we can use depth-first search to perform a topological sort of a directed acyclic graph, or a "dag" as it is sometimes called. A ***topological sort*** of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. (If the graph contains a cycle, then no linear ordering is possible.) We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of "sorting" studied in Part II.

Many applications use directed acyclic graphs to indicate precedences among events. Figure 22.7 gives an example that arises when Professor Bumstead gets dressed in the morning. The professor must don certain garments before others (e.g., socks before shoes). Other items may be put on in any order (e.g., socks and

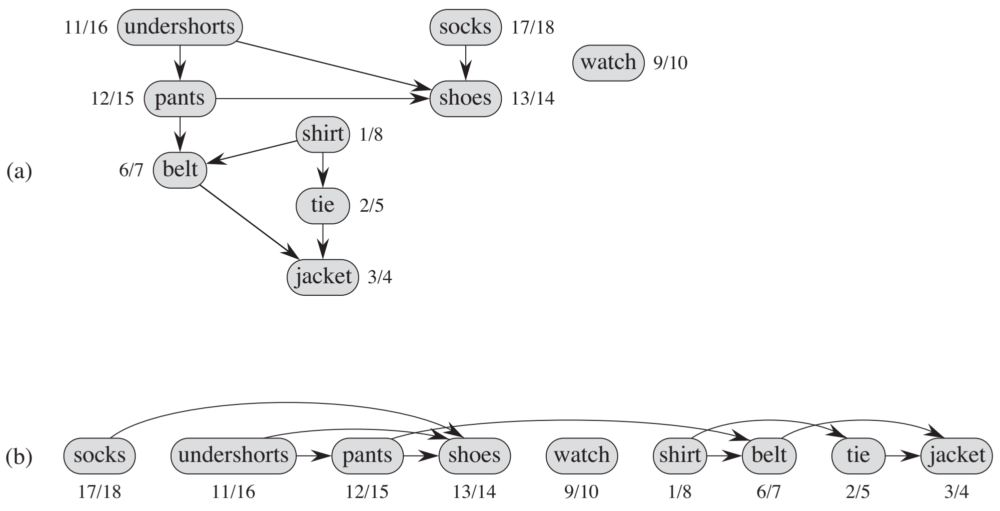

Figure 22.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finishing time. All directed edges go from left to right.

pants). A directed edge (u, v) in the dag of Figure 22.7(a) indicates that garment u must be donned before garment v. A topological sort of this dag therefore gives an order for getting dressed. Figure 22.7(b) shows the topologically sorted dag as an ordering of vertices along a horizontal line such that all directed edges go from left to right.

The following simple algorithm topologically sorts a dag:

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

Figure 22.7(b) shows how the topologically sorted vertices appear in reverse order of their finishing times.

We can perform a topological sort in time $\Theta(V + E)$, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

We prove the correctness of this algorithm using the following key lemma characterizing directed acyclic graphs.

Lemma 22.11
A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

Proof $\Rightarrow$: Suppose that a depth-first search produces a back edge (u, v). Then vertex v is an ancestor of vertex u in the depth-first forest. Thus, G contains a path from v to u, and the back edge (u, v) completes a cycle.

$\Leftarrow$: Suppose that G contains a cycle c. We show that a depth-first search of G yields a back edge. Let v be the first vertex to be discovered in c, and let (u, v) be the preceding edge in c. At time $v.d$, the vertices of c form a path of white vertices from v to u. By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest. Therefore, (u, v) is a back edge. ■

Theorem 22.12
TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input.

Proof Suppose that DFS is run on a given dag $G = (V, E)$ to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices $u, v \in V$, if G contains an edge from u to v, then $v.f < u.f$. Consider any edge (u, v) explored by DFS(G). When this edge is explored, v cannot be gray, since then v would be an ancestor of u and (u, v) would be a back edge, contradicting Lemma 22.11. Therefore, v must be either white or black. If v is white, it becomes a descendant of u, and so $v.f < u.f$. If v is black, it has already been finished, so that $v.f$ has already been set. Because we are still exploring from u, we have yet to assign a timestamp to $u.f$, and so once we do, we will have $v.f < u.f$ as well. Thus, for any edge (u, v) in the dag, we have $v.f < u.f$, proving the theorem. ■

Exercises

22.4-1
Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 22.8, under the assumption of Exercise 22.3-2.

22.4-2
Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t, and returns the number of simple paths from s to t in G. For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex p to vertex v: pov, $poryv$, $posryv$, and $psryv$. (Your algorithm needs only to count the simple paths, not list them.)

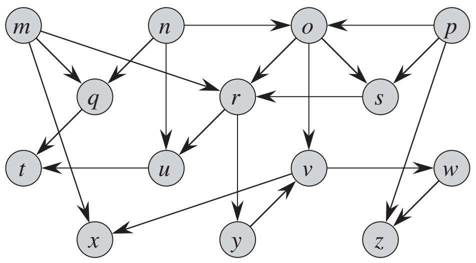

Figure 22.8 A dag for topological sorting.

22.4-3
Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

22.4-4
Prove or disprove: If a directed graph G contains cycles, then TOPOLOGICAL-SORT(G) produces a vertex ordering that minimizes the number of "bad" edges that are inconsistent with the ordering produced.

22.4-5
Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles?

22.5 Strongly connected components

We now consider a classic application of depth-first search: decomposing a directed graph into its strongly connected components. This section shows how to do so using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition. After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among components.

Recall from Appendix B that a strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both $u \leadsto v$ and $v \leadsto u$; that is, vertices u and v are reachable from each other. Figure 22.9 shows an example.

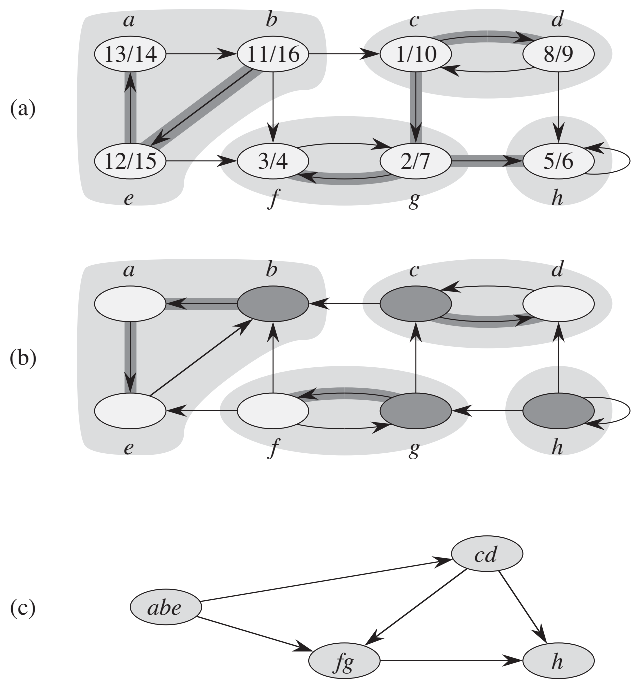

Figure 22.9 **(a)** A directed graph G. Each shaded region is a strongly connected component of G. Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded. **(b)** The graph G^{T}, the transpose of G, with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^{T}. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of G, which we defined in Exercise 22.1-3 to be the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$. That is, E^{T} consists of the edges of G with their directions reversed. Given an adjacency-list representation of G, the time to create G^{T} is $O(V + E)$. It is interesting to observe that G and G^{T} have exactly the same strongly connected components: u and v are reachable from each other in G if and only if they are reachable from each other in G^{T}. Figure 22.9(b) shows the transpose of the graph in Figure 22.9(a), with the strongly connected components shaded.

The following linear-time (i.e., $\Theta(V+E)$-time) algorithm computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^{T}.

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

The idea behind this algorithm comes from a key property of the ***component graph*** $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$, which we define as follows. Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set V^{SCC} is $\{v_1, v_2, \ldots, v_k\}$, and it contains a vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E^{\mathrm{SCC}}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$. Looked at another way, by contracting all edges whose incident vertices are within the same strongly connected component of G, the resulting graph is G^{SCC}. Figure 22.9(c) shows the component graph of the graph in Figure 22.9(a).

The key property is that the component graph is a dag, which the following lemma implies.

Lemma 22.13
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that G contains a path $u \leadsto u'$. Then G cannot also contain a path $v' \leadsto v$.

Proof If G contains a path $v' \leadsto v$, then it contains paths $u \leadsto u' \leadsto v'$ and $v' \leadsto v \leadsto u$. Thus, u and v' are reachable from each other, thereby contradicting the assumption that C and C' are distinct strongly connected components. ■

We shall see that by considering vertices in the second depth-first search in decreasing order of the finishing times that were computed in the first depth-first search, we are, in essence, visiting the vertices of the component graph (each of which corresponds to a strongly connected component of G) in topologically sorted order.

Because the STRONGLY-CONNECTED-COMPONENTS procedure performs two depth-first searches, there is the potential for ambiguity when we discuss $u.d$ or $u.f$. In this section, these values always refer to the discovery and finishing times as computed by the first call of DFS, in line 1.

We extend the notation for discovery and finishing times to sets of vertices. If $U \subseteq V$, then we define $d(U) = \min_{u \in U} \{u.d\}$ and $f(U) = \max_{u \in U} \{u.f\}$. That is, $d(U)$ and $f(U)$ are the earliest discovery time and latest finishing time, respectively, of any vertex in U.

The following lemma and its corollary give a key property relating strongly connected components and finishing times in the first depth-first search.

Lemma 22.14
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Proof We consider two cases, depending on which strongly connected component, C or C', had the first discovered vertex during the depth-first search.

If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $x.d$, all vertices in C and C' are white. At that time, G contains a path from x to each vertex in C consisting only of white vertices. Because $(u, v) \in E$, for any vertex $w \in C'$, there is also a path in G at time $x.d$ from x to w consisting only of white vertices: $x \leadsto u \to v \leadsto w$. By the white-path theorem, all vertices in C and C' become descendants of x in the depth-first tree. By Corollary 22.8, x has the latest finishing time of any of its descendants, and so $x.f = f(C) > f(C')$.

If instead we have $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $y.d$, all vertices in C' are white and G contains a path from y to each vertex in C' consisting only of white vertices. By the white-path theorem, all vertices in C' become descendants of y in the depth-first tree, and by Corollary 22.8, $y.f = f(C')$. At time $y.d$, all vertices in C are white. Since there is an edge (u, v) from C to C', Lemma 22.13 implies that there cannot be a path from C' to C. Hence, no vertex in C is reachable from y. At time $y.f$, therefore, all vertices in C are still white. Thus, for any vertex $w \in C$, we have $w.f > y.f$, which implies that $f(C) > f(C')$. ■

The following corollary tells us that each edge in G^{T} that goes between different strongly connected components goes from a component with an earlier finishing time (in the first depth-first search) to a component with a later finishing time.

Corollary 22.15
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^{\mathrm{T}}$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof Since $(u, v) \in E^{\mathrm{T}}$, we have $(v, u) \in E$. Because the strongly connected components of G and G^{T} are the same, Lemma 22.14 implies that $f(C) < f(C')$. ■

Corollary 22.15 provides the key to understanding why the strongly connected components algorithm works. Let us examine what happens when we perform the second depth-first search, which is on G^{T}. We start with the strongly connected component C whose finishing time $f(C)$ is maximum. The search starts from some vertex $x \in C$, and it visits all vertices in C. By Corollary 22.15, G^{T} contains no edges from C to any other strongly connected component, and so the search from x will not visit vertices in any other component. Thus, the tree rooted at x contains exactly the vertices of C. Having completed visiting all vertices in C, the search in line 3 selects as a root a vertex from some other strongly connected component C' whose finishing time $f(C')$ is maximum over all components other than C. Again, the search will visit all vertices in C', but by Corollary 22.15, the only edges in G^{T} from C' to any other component must be to C, which we have already visited. In general, when the depth-first search of G^{T} in line 3 visits any strongly connected component, any edges out of that component must be to components that the search already visited. Each depth-first tree, therefore, will be exactly one strongly connected component. The following theorem formalizes this argument.

Theorem 22.16
The STRONGLY-CONNECTED-COMPONENTS procedure correctly computes the strongly connected components of the directed graph G provided as its input.

Proof We argue by induction on the number of depth-first trees found in the depth-first search of G^{T} in line 3 that the vertices of each tree form a strongly connected component. The inductive hypothesis is that the first k trees produced in line 3 are strongly connected components. The basis for the induction, when $k = 0$, is trivial.

In the inductive step, we assume that each of the first k depth-first trees produced in line 3 is a strongly connected component, and we consider the $(k + 1)$st tree produced. Let the root of this tree be vertex u, and let u be in strongly connected component C. Because of how we choose roots in the depth-first search in line 3, $u.f = f(C) > f(C')$ for any strongly connected component C' other than C that has yet to be visited. By the inductive hypothesis, at the time that the search visits u, all other vertices of C are white. By the white-path theorem, therefore, all other vertices of C are descendants of u in its depth-first tree. Moreover, by the inductive hypothesis and by Corollary 22.15, any edges in G^{T} that leave C must be to strongly connected components that have already been visited. Thus, no vertex in any strongly connected component other than C will be a descendant of u during the depth-first search of G^{T}. Thus, the vertices of the depth-first tree in G^{T} that is rooted at u form exactly one strongly connected component, which completes the inductive step and the proof. ■

Here is another way to look at how the second depth-first search operates. Consider the component graph $(G^{\mathrm{T}})^{\mathrm{SCC}}$ of G^{T}. If we map each strongly connected component visited in the second depth-first search to a vertex of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, the second depth-first search visits vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in the reverse of a topologically sorted order. If we reverse the edges of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, we get the graph $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}}$. Because $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$ (see Exercise 22.5-4), the second depth-first search visits the vertices of G^{SCC} in topologically sorted order.

Exercises

22.5-1
How can the number of strongly connected components of a graph change if a new edge is added?

22.5-2
Show how the procedure STRONGLY-CONNECTED-COMPONENTS works on the graph of Figure 22.6. Specifically, show the finishing times computed in line 1 and the forest produced in line 3. Assume that the loop of lines 5–7 of DFS considers vertices in alphabetical order and that the adjacency lists are in alphabetical order.

22.5-3
Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of *increasing* finishing times. Does this simpler algorithm always produce correct results?

22.5-4
Prove that for any directed graph G, we have $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$. That is, the transpose of the component graph of G^{T} is the same as the component graph of G.

22.5-5
Give an $O(V + E)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

22.5-6
Given a directed graph $G = (V, E)$, explain how to create another graph $G' = (V, E')$ such that (a) G' has the same strongly connected components as G, (b) G' has the same component graph as G, and (c) E' is as small as possible. Describe a fast algorithm to compute G'.

22.5-7
A directed graph $G = (V, E)$ is ***semiconnected*** if, for all pairs of vertices $u, v \in V$, we have $u \leadsto v$ or $v \leadsto u$. Give an efficient algorithm to determine whether or not G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

Problems

22-1 Classifying edges by breadth-first search
A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.

a. Prove that in a breadth-first search of an undirected graph, the following properties hold:

1. There are no back edges and no forward edges.
2. For each tree edge (u, v), we have $v.d = u.d + 1$.
3. For each cross edge (u, v), we have $v.d = u.d$ or $v.d = u.d + 1$.

b. Prove that in a breadth-first search of a directed graph, the following properties hold:

1. There are no forward edges.
2. For each tree edge (u, v), we have $v.d = u.d + 1$.
3. For each cross edge (u, v), we have $v.d \leq u.d + 1$.
4. For each back edge (u, v), we have $0 \leq v.d \leq u.d$.

22-2 Articulation points, bridges, and biconnected components
Let $G = (V, E)$ be a connected, undirected graph. An ***articulation point*** of G is a vertex whose removal disconnects G. A ***bridge*** of G is an edge whose removal disconnects G. A ***biconnected component*** of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. Figure 22.10 illustrates

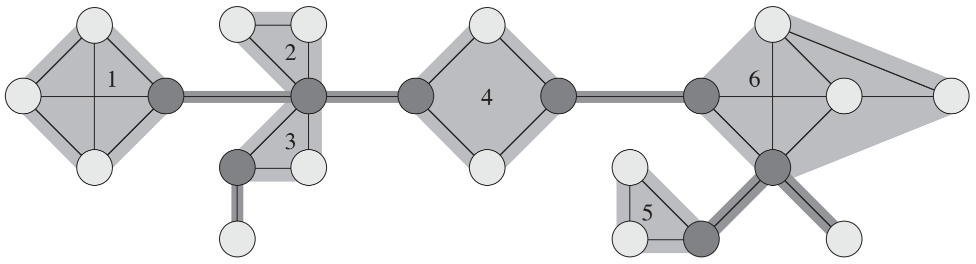

Figure 22.10 The articulation points, bridges, and biconnected components of a connected, undirected graph for use in Problem 22-2. The articulation points are the heavily shaded vertices, the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded regions, with a *bcc* numbering shown.

these definitions. We can determine articulation points, bridges, and biconnected components using depth-first search. Let $G_\pi = (V, E_\pi)$ be a depth-first tree of G.

a. Prove that the root of G_π is an articulation point of G if and only if it has at least two children in G_π.

b. Let v be a nonroot vertex of G_π. Prove that v is an articulation point of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

c. Let

$$v.low = \min \begin{cases} v.d\,, \\ w.d : (u, w) \text{ is a back edge for some descendant } u \text{ of } v\,. \end{cases}$$

Show how to compute $v.low$ for all vertices $v \in V$ in $O(E)$ time.

d. Show how to compute all articulation points in $O(E)$ time.

e. Prove that an edge of G is a bridge if and only if it does not lie on any simple cycle of G.

f. Show how to compute all the bridges of G in $O(E)$ time.

g. Prove that the biconnected components of G partition the nonbridge edges of G.

h. Give an $O(E)$-time algorithm to label each edge e of G with a positive integer $e.bcc$ such that $e.bcc = e'.bcc$ if and only if e and e' are in the same biconnected component.

22-3 Euler tour

An ***Euler tour*** of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

a. Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex $v \in V$.

b. Describe an $O(E)$-time algorithm to find an Euler tour of G if one exists. (*Hint:* Merge edge-disjoint cycles.)

22-4 Reachability

Let $G = (V, E)$ be a directed graph in which each vertex $u \in V$ is labeled with a unique integer $L(u)$ from the set $\{1, 2, \ldots, |V|\}$. For each vertex $u \in V$, let $R(u) = \{v \in V : u \leadsto v\}$ be the set of vertices that are reachable from u. Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum, i.e., $\min(u)$ is the vertex v such that $L(v) = \min\{L(w) : w \in R(u)\}$. Give an $O(V + E)$-time algorithm that computes $\min(u)$ for all vertices $u \in V$.

Chapter notes

Even [103] and Tarjan [330] are excellent references for graph algorithms.

Breadth-first search was discovered by Moore [260] in the context of finding paths through mazes. Lee [226] independently discovered the same algorithm in the context of routing wires on circuit boards.

Hopcroft and Tarjan [178] advocated the use of the adjacency-list representation over the adjacency-matrix representation for sparse graphs and were the first to recognize the algorithmic importance of depth-first search. Depth-first search has been widely used since the late 1950s, especially in artificial intelligence programs.

Tarjan [327] gave a linear-time algorithm for finding strongly connected components. The algorithm for strongly connected components in Section 22.5 is adapted from Aho, Hopcroft, and Ullman [6], who credit it to S. R. Kosaraju (unpublished) and M. Sharir [314]. Gabow [119] also developed an algorithm for strongly connected components that is based on contracting cycles and uses two stacks to make it run in linear time. Knuth [209] was the first to give a linear-time algorithm for topological sorting.