And, for the hous is crinkled to and fro,
And hath so queinte weyes for to go—
For hit is shapen as the mase is wroght—
Therto have I a remedie in my thoght,
That, by a clewe of twyne, as he hath goon,
The same wey he may returne anoon,
Folwing alwey the threed, as he hath come.

— Geoffrey Chaucer, *The Legend of Good Women* (c. 1385)

"Com'è bello il mondo e come sono brutti i labirinti!" dissi sollevato.
"Come sarebbe bello il mondo se ci fosse una regola per girare nei labirinti,"
rispose il mio maestro.
["How beautiful the world is, and how ugly labyrinths are," I said, relieved.
"How beautiful the world would be if there were a procedure for moving through
labyrinths," my master replied.]

— Umberto Eco, *Il nome della rosa* (1980)
English translation (*The Name of the Rose*) by William Weaver (1983)

6

Depth-First Search

In the previous chapter, we considered a generic algorithm—whatever-first search—for traversing arbitrary graphs, both undirected and directed. In this chapter, we focus on a particular instantiation of this algorithm called *depth-first search*, and primarily on the behavior of this algorithm in directed graphs. Rather than using an explicit stack, depth-first search is normally implemented recursively as follows:

```
DFS(v):
  if v is unmarked
      mark v
      for each edge v→w
          DFS(w)
```

We can make this algorithm slightly faster (in practice) by checking whether a node is marked *before* we recursively explore it. This modification ensures

that we call DFS(v) only once for each vertex v. We can further modify the algorithm to compute other useful information about the vertices and edges, by introducing two black-box subroutines, PREVISIT and POSTVISIT, which we leave unspecified for now.

```
DFS(v):
  mark v
  PREVISIT(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  POSTVISIT(v)
```

Recall that a node w is *reachable* from another node v in a directed graph G—or more simply, v can reach w—if and only if G contains a directed path from v to w. Let ***reach(v)*** denote the set of vertices reachable from v (including v itself). If we unmark all vertices in G, and then call DFS(v), the set of marked vertices is precisely *reach*(v).

Reachability in undirected graphs is symmetric: v can reach w if and only if w can reach v. As a result, after unmarking all vertices of an undirected graph G, calling DFS(v) traverses the entire component of v, and the parent pointers define a spanning tree of that component.

The situation is more subtle with directed graphs, as shown in the figure below. Even though the graph is "connected", different vertices can reach different, and potentially overlapping, portions of the graph. The parent pointers assigned by DFS(v) define a tree rooted at v whose vertices are precisely *reach*(v), but this is not necessarily a spanning tree of the graph.

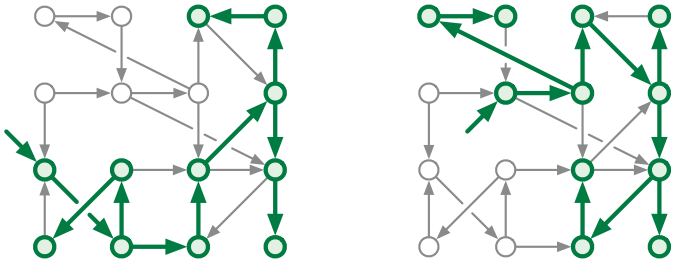

Figure 6.1. Depth-first trees rooted at different vertices in the same directed graph.

As usual, we can extend our reachability algorithm to traverse the *entire* input graph, even if it is disconnected, using the standard wrapper function shown on the left in Figure 6.2. Here we add a generic black-box subroutine PREPROCESS to perform any necessary preprocessing for the PREVISIT and POSTVISIT functions.

```
DFSALL(G):
  PREPROCESS(G)
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

```
DFSALL(G):
  PREPROCESS(G)
  add vertex s
  for all vertices v
      add edge s→v
      unmark v
  DFS(s)
```

Figure 6.2. Two formulations of the standard wrapper algorithm for depth-first search

Alternatively, if we are allowed to modify the graph, we can add a new *source* vertex s, with edges to every other vertex in G, and then make a single call to DFS(s), as shown on the right of Figure 6.2. Now the resulting parent pointers always define a spanning tree of the *augmented* input graph, but not of the *original* input graph. Otherwise, the two wrapper functions have essentially identical behavior; choosing one or the other is entirely a matter of convenience.[1]

Again, this algorithm behaves slightly differently for undirected and directed graphs. In undirected graphs, as we saw in the previous chapter, it is easy to adapt DFSALL to count the components of a graph; in particular, the parent pointers computed by DFSALL define a spanning forest of the input graph, containing a spanning tree for each component. When the graph is directed, however, DFSALL may discover any number of "components" between 1 and V, even when the graph is "connected", depending on the precise structure of the graph and the order in which the wrapper algorithm visits the vertices.

6.1 Preorder and Postorder

Hopefully you are already familiar with preorder and postorder traversals of rooted *trees*, both of which can be computed using depth-first search. Similar traversal orders can be defined for arbitrary directed graphs—even if they are disconnected—by passing around a counter as follows:

```
DFSALL(G):
  clock ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          clock ← DFS(v, clock)
```

```
DFS(v, clock):
  mark v
  clock ← clock + 1;  v.pre ← clock
  for each edge v→w
      if w is unmarked
          w.parent ← v
          clock ← DFS(w, clock)
  clock ← clock + 1;  v.post ← clock
  return clock
```

[1]The equivalence of these two wrapper functions is a specific feature of ***depth***-first search. In particular, wrapping *breadth*-first search in a for-loop to visit every vertex does *not* yield the same traversal order as adding a source vertex and invoking breadth-first search at s.

Equivalently, we can use our generic depth-first-search algorithm with the following subroutines PREPROCESS, PREVISIT, and POSTVISIT.

```
PREPROCESS(G):
  clock ← 0
```

```
PREVISIT(v):
  clock ← clock + 1
  v.pre ← clock
```

```
POSTVISIT(v):
  clock ← clock + 1
  v.post ← clock
```

This algorithm assigns $v.pre$ (and advance the clock) just after it pushes v onto the recursion stack, and it assigns $v.post$ (and advance the clock) just before it pops v off the recursion stack. It follows that for any two vertices u and v, the intervals $[u.pre, u.post]$ and $[v.pre, v.post]$ are either disjoint or nested. Moreover, $[u.pre, u.post]$ contains $[v.pre, v.post]$ if and only if DFS(v) is called during the execution of DFS(u), or equivalently, if and only if u is an ancestor of v in the final forest of parent pointers.

After DFSALL labels every node in the graph, the labels $v.pre$ define a ***preordering*** of the vertices, and the labels $v.post$ define a ***postordering*** of the vertices.[2] With a few trivial exceptions, every graph has several different pre- and postorderings, depending on the order that DFS considers edges leaving each vertex, and the order that DFSALL considers vertices.

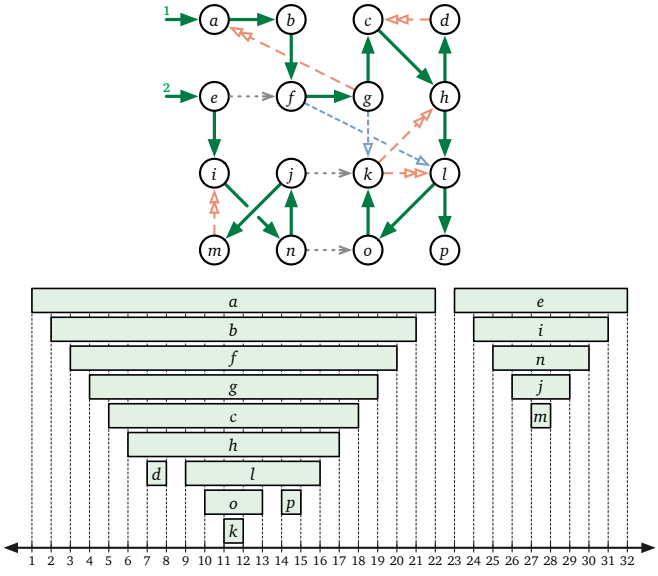

Figure 6.3. A depth-first forest of a directed graph, and the corresponding active intervals of its vertices, defining the preordering *abfgchdlokpeinjm* and the postordering *dkoplhcgfbamjnie*. Forest edges are solid; dashed edges are explained in Figure 6.4.

For the rest of this chapter, we refer to $v.pre$ as the ***starting time*** of v (or less formally, "when v starts"), $v.post$ as the ***finishing time*** of v (or less formally,

[2] Confusingly, *both* of these orders are sometimes called "depth-first ordering". Please don't do that.

"when v finishes"), and the interval between the starting and finishing times as the ***active interval*** of v (or less formally, "while v is active").

Classifying Vertices and Edges

During the execution of DFSALL, each vertex v of the input graph has one of three states:

- ***new*** if DFS(v) has not been called, that is, if $clock < v.pre$;
- ***active*** if DFS(v) has been called but has not returned, that is, if $v.pre \le clock < v.post$;
- ***finished*** if DFS(v) has returned, that is, if $v.post \le clock$.

Because starting and finishing times correspond to pushes and pops on the recursion stack, a vertex is active if and only if it is on the recursion stack. It follows that the active nodes always comprise a directed path in G.

The edges of the input graph fall into four different classes, depending on how their active intervals intersect. Fix your favorite edge $u \to v$.

- If v is new when DFS(u) begins, then DFS(v) must be called during the execution of DFS(u), either directly or through some intermediate recursive calls. In either case, u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.
  - If DFS(u) calls DFS(v) directly, then $u = v.parent$ and $u \to v$ is called a ***tree edge***.
  - Otherwise, $u \to v$ is called a ***forward edge***.
- If v is active when DFS(u) begins, then v is already on the recursion stack, which implies the opposite nesting order $v.pre < u.pre < u.post < v.post$. Moreover, G must contain a directed path from v to u. Edges satisfying this condition are called ***back edges***.
- If v is finished when DFS(u) begins, we immediately have $v.post < u.pre$. Edges satisfying this condition are called ***cross edges***.
- Finally, the fourth ordering $u.post < v.pre$ is impossible.

These edge classes are illustrated in Figure 6.4. Again, the actual classification of edges depends on the order in which DFSALL considers vertices and the order in which DFS considers the edges leaving each vertex.

Finally, the following key lemma characterizes ancestors and descendants in any depth-first forest according to vertex states during the traversal.

Lemma 6.1. *Fix an arbitrary depth-first traversal of any directed graph G. The following statements are equivalent for all vertices u and v of G.*

(a) u is an ancestor of v in the depth-first forest.

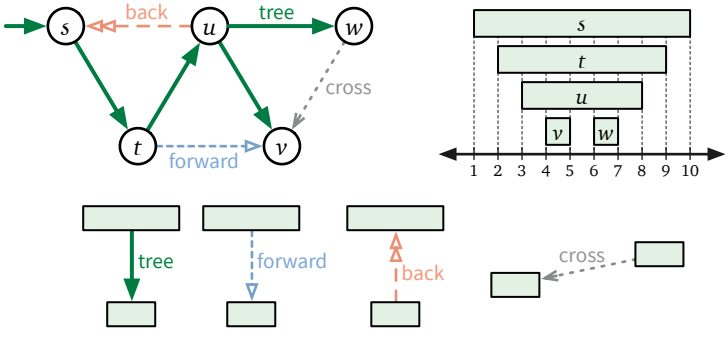

Figure 6.4. Classification of edges by depth-first search.

(b) $u.pre \le v.pre < v.post \le u.post$.
(c) Just after DFS(v) is called, u is active.
(d) Just before DFS(u) is called, there is a path from u to v in which every vertex (including u and v) is new.

Proof: First, suppose u is an ancestor of v in the depth-first forest. Then by definition there is a path P of tree edges u to v. By induction on the path length, we have $u.pre \le w.pre < w.post \le u.post$ for every vertex w in P, and thus every vertex in P is new before DFS(u) is called. In particular, we have $u.pre \le v.pre < v.post \le u.post$, which implies that u is active while DFS(v) is executing.

Because parent pointers correspond to recursive calls, $u.pre \le v.pre < v.post \le u.post$ implies that u is an ancestor of v.

Suppose u is active just after DFS(v) is called. Then $u.pre \le v.pre < v.post \le u.post$, which implies that there is a path of (zero or more) tree edges from u, through the intermediate nodes on the recursion stack (if any), to v.

Finally, suppose u is not an ancestor of v. Fix an arbitrary path P from u to v, let x be the first vertex in P that is not a descendant of u, and let w be the predecessor of x in P. The edge $w \to x$ guarantees that $x.pre < w.post$, and $w.post < u.post$ because w is a descendant of u, so $x.pre < u.post$. It follows that $x.pre < u.pre$, because otherwise x would be a descendant of u. Because active intervals are properly nested, there are only two possibilities:

- If $u.post < x.post$, then x is active when DFS(u) is called.
- If $x.post < u.pre$, then x is already finished when DFS(u) is called.

We conclude that *every* path from u to v contains a vertex that is not new when DFS(u) is called. □

6.2 Detecting Cycles

A ***directed acyclic graph*** or ***dag*** is a directed graph with no directed cycles. Any vertex in a dag that has no incoming vertices is called a ***source***; any vertex with no outgoing edges is called a ***sink***. An isolated vertex with no incident edges at all is both a source and a sink. Every dag has at least one source and one sink, but may have more than one of each. For example, in the graph with n vertices but no edges, every vertex is a source and every vertex is a sink.

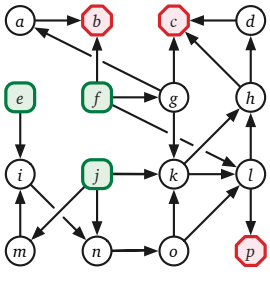

Figure 6.5. A directed acyclic graph. Vertices e, f, and j are sources; vertices b, c, and p are sinks.

Recall from our earlier case analysis that if $u.post < v.post$ for any edge $u \to v$, the graph contains a directed path from v to u, and therefore contains a directed *cycle* through the edge $u \to v$. Thus, we can determine whether a given directed graph G is a dag in $O(V + E)$ time by computing a postordering of the vertices and then checking each edge by brute force.

Alternatively, instead of numbering the vertices, we can explicitly maintain the status of each vertex and immediately return FALSE if we ever discover an edge to an active vertex. This algorithm also runs in $O(V + E)$ time; see Figure 6.6.

```
IsAcyclic(G):
  for all vertices v
      v.status ← New
  for all vertices v
      if v.status = New
          if IsAcyclicDFS(v) = False
              return False
  return True
```

```
IsAcyclicDFS(v):
  v.status ← Active
  for each edge v→w
      if w.status = Active
          return False
      else if w.status = New
          if IsAcyclicDFS(w) = False
              return False
  v.status ← Finished
  return True
```

Figure 6.6. A linear-time algorithm to determine if a graph is acyclic.

6.3 Topological Sort

A ***topological ordering*** of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \to v$. Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right. A topological ordering is clearly impossible if the graph G has a directed cycle—the rightmost vertex of the cycle would have an edge pointing to the left!

On the other hand, consider an arbitrary postordering of an arbitrary directed graph G. Our earlier analysis implies that $u.post < v.post$ for any edge $u \to v$, then G contains a directed path from v to u, and therefore contains a directed cycle through $u \to v$. Equivalently, if G is acyclic, then $u.post > v.post$ for every edge $u \to v$. It follows that every directed acyclic graph G has a topological ordering; in particular, the reversal of any postordering of G is a topological ordering of G.

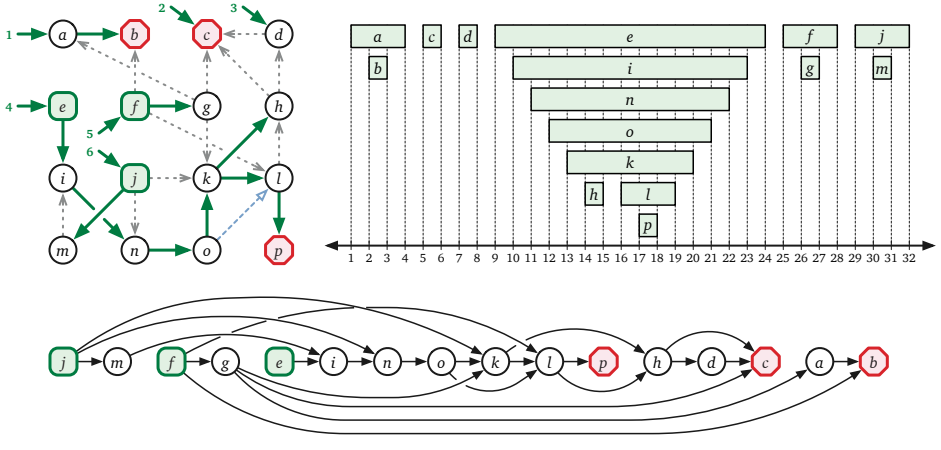

Figure 6.7. Reversed postordering of the dag from Figure 6.5.

If we require the topological ordering in a separate data structure, we can simply write the vertices into an array in reverse postorder, in $O(V + E)$ time, as shown in Figure 6.8.

Implicit Topological Sort

But recording the topological order into a separate data structure is usually overkill. In most applications of topological sort, the ordered list of the vertices is not our actual goal; rather, we want to perform some fixed computation at each vertex of the graph, either in topological order or in reverse topological order. For these applications, it is not necessary to *record* the topological order at all!

```
TopologicalSort(G):
  for all vertices v
      v.status ← New
  clock ← V
  for all vertices v
      if v.status = New
          clock ← TopSortDFS(v, clock)
  return S[1..V]
```

```
TopSortDFS(v, clock):
  v.status ← Active
  for each edge v→w
      if w.status = New
          clock ← TopSortDFS(v, clock)
      else if w.status = Active
          fail gracefully
  v.status ← Finished
  S[clock] ← v
  clock ← clock − 1
  return clock
```

Figure 6.8. Explicit topological sort

If we want to process a directed acyclic graph in *reverse* topological order, it suffices to process each vertex at the end of its recursive depth-first search. After all, topological order is the same as reversed postorder!

```
PostProcess(G):
  for all vertices v
      v.status ← New
  for all vertices v
      if v is unmarked
          PostProcessDFS(v)
```

```
PostProcessDFS(v):
  v.status ← Active
  for each edge v→w
      if w.status = New
          PostProcessDFS(w)
      else if w.status = Active
          fail gracefully
  v.status ← Finished
  Process(v)
```

If we already *know* that the input graph is acyclic, we can further simplify the algorithm by simply marking vertices instead of recording their search status.

```
PostProcessDag(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          PostProcessDagDFS(s)
```

```
PostProcessDagDFS(v):
  mark v
  for each edge v→w
      if w is unmarked
          PostProcessDagDFS(w)
  Process(v)
```

This is just the standard depth-first search algorithm, with PostVisit renamed to Process!

Because it is such a common operation on directed acyclic graphs, I sometimes express postorder processing of a dag idiomatically as follows:

```
PostProcessDag(G):
  for all vertices v in postorder
      Process(v)
```

For example, our earlier explicit topological sort algorithm can be written as follows:

```
TopologicalSort(G):
  clock ← V
  for all vertices v in postorder
      S[clock] ← v
      clock ← clock − 1
  return S[1..V]
```

To process a dag in *forward* topological order, we can record a topological ordering of the vertices into an array and then run a simple for-loop. Alternatively, we can apply depth-first search to the ***reversal*** of G, denoted $\boldsymbol{rev(G)}$, obtained by replacing each each $v \to w$ with its reversal $w \to v$. Reversing a directed cycle gives us another directed cycle with the opposite orientation, so the reversal of a dag is another dag. Every source in G is a sink in $rev(G)$ and vice versa; it follows inductively that every topological ordering of $rev(G)$ is the reversal of a topological ordering of G.[3] The reversal of any directed graph (represented in a standard adjacency list) can be computed in $O(V + E)$ time; the details of this construction are left as an easy exercise.

6.4 Memoization and Dynamic Programming

Our topological sort algorithm is arguably the model for a wide class of dynamic programming algorithms. Recall that the ***dependency graph*** of a recurrence has a vertex for every recursive subproblem and an edge from one subproblem to another if evaluating the first subproblem requires a recursive evaluation of the second. The dependency graph must be acyclic, or the naïve recursive algorithm would never halt.

Evaluating any recurrence with memoization is *exactly* the same as performing a depth-first search of the dependency graph. In particular, a vertex of the dependency graph is "marked" if the value of the corresponding subproblem has already been computed. The black-box subroutines PreVisit and PostVisit are proxies for the actual value computation. See Figure 6.9.

Carrying this analogy further, evaluating a recurrence *using dynamic programming* is the same as evaluating all subproblems in the dependency graph of the recurrence in reverse topological order—every subproblem is considered *after* the subproblems it depends on. Thus, *every* dynamic programming algorithm is equivalent to a postorder traversal of the dependency graph of its underlying recurrence!

[3] A postordering of the reversal of G is not necessarily the reversal of a postordering of G, even though both are topological orderings of G.

```
MEMOIZE(x):
  if value[x] is undefined
    initialize value[x]

    for all subproblems y of x
      MEMOIZE(y)
      update value[x] based on value[y]
    finalize value[x]
```

```
DFS(v):
  if v is unmarked
    mark v
    PREVISIT(x)
    for all edges v→w
      DFS(w)

    POSTVISIT(x)
```

Figure 6.9. Memoized recursion is depth-first search. Depth-first search is memoized recursion.

```
DYNAMICPROGRAMMING(G):
  for all subproblems x in postorder
    initialize value[x]
    for all subproblems y of x
      update value[x] based on value[y]
    finalize value[x]
```

Figure 6.10. Dynamic programming is postorder traversal.

However, there are some minor differences between most dynamic programming algorithms and topological sort. First, in most dynamic programming algorithms, the dependency graph is *implicit*—the nodes and edges are not explicitly stored in memory, but rather are encoded by the underlying recurrence. But this difference really is minor; as long as we can enumerate recursive subproblems in constant time each, we can traverse the dependency graph exactly *as if* it were explicitly stored in an adjacency list.

More significantly, most dynamic programming recurrences have highly structured dependency graphs. For example, as we discussed in Chapter **??**, the dependency graph for the edit distance recurrence is a regular grid with diagonals, and the dependency graph for optimal binary search trees is an upper triangular grid with all possible rightward and upward edges. This regular structure allows us to hard-wire a (reverse) topological order directly into the algorithm, typically as a collection of nested loops, so there is no need to topologically sort the dependency graph at run time. We previously called the reverse topological order an *evaluation order*.

Dynamic Programming in Dags

Conversely, we can use depth-first search to build dynamic programming algorithms for problems with less structured dependency graphs. For example, consider the ***longest path*** problem, which asks for the path of *maximum* total weight from one node s to another node t in a directed graph G with weighted edges. The longest path problem is NP-hard in general directed graphs, by an

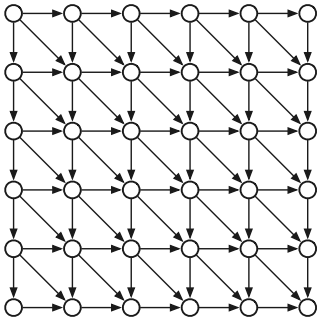

Figure 6.11. The dependency **dag** of the edit distance recurrence.

easy reduction from the traveling salesman problem, but it is easy to solve in linear time if the input graph G is acyclic, as follows.

Fix the target vertex t, and for any node v, let $\boldsymbol{LLP(v)}$ denote the Length of the Longest Path in G from v to t. If G is a dag, this function satisfies the recurrence

$$LLP(v) = \begin{cases} 0 & \text{if } v = t, \\ \max\left\{ \ell(v \to w) + LLP(w) \;\middle|\; v \to w \in E \right\} & \text{otherwise,} \end{cases}$$

where $\ell(v \to w)$ denotes the given weight ("length") of edge $v \to w$, and $\max \varnothing = -\infty$. In particular, if v is a *sink* but not equal to t, then $LLP(v) = -\infty$.

The dependency graph for this recurrence is the input graph G itself: subproblem $LLP(v)$ depends on subproblem $LLP(w)$ if and only if $v \to w$ is an edge in G. Thus, we can evaluate this recursive function in $O(V + E)$ time by performing a depth-first search of G, starting at s. The algorithm memoizes each length $LLP(v)$ into an extra field in the corresponding node v.

```
LongestPath(v, t):
  if v = t
      return 0
  if v.LLP is undefined
      v.LLP ← −∞
      for each edge v→w
          v.LLP ← max{v.LLP, ℓ(v→w) + LongestPath(w, t)}
      return v.LLP
```

In principle, we can transform this memoized recursive algorithm into a dynamic programming algorithm via topological sort:

```
LongestPath(s, t):
  for each node v in postorder
      if v = t
          v.LLP ← 0
      else
          v.LLP ← −∞
          for each edge v→w
              v.LLP ← max{v.LLP, ℓ(v→w) + w.LLP}
  return s.LLP
```

These two algorithms are arguably identical—the recursion in the first algorithm and the for-loop in the second algorithm represent the "same" depth-first search! Choosing one of these formulations over the other is entirely a matter of convenience.

Almost any dynamic programming problem that asks for an optimal *sequence* of decisions can be recast as finding an optimal *path* in some associated dag. For example, the text segmentation, subset sum, longest increasing subsequence, and edit distance problems we considered in Chapters **??** and **??** can all be reformulated as finding either a longest path or a shortest path in a dag, possibly with weighted vertices or edges. On the other hand, "tree-shaped" dynamic programming problems, like finding optimal binary search trees or maximum independent sets in trees, cannot be recast as finding an optimal *path* in a dag.

6.5 Strong Connectivity

Let's go back to the proper definition of connectivity in directed graphs. Recall that one vertex u can *reach* another vertex v in a directed graph G if G contains a directed path from u to v, and that $\mathit{reach}(u)$ denotes the set of all vertices that u can reach. Two vertices u and v are ***strongly connected*** if u can reach v and v can reach u. A directed graph is strongly connected if and only if every pair of vertices is strongly connected.

Tedious definition-chasing implies that strong connectivity is an equivalence relation over the set of vertices of any directed graph, just like connectivity in undirected graphs. The equivalence classes of this relation are called the ***strongly connected components***—or more simply, the ***strong components***—of G. Equivalently, a strong component of G is a maximal strongly connected subgraph of G. A directed graph G is strongly connected if and only if G has exactly one strong component; at the other extreme, G is a dag if and only if every strong component of G consists of a single vertex.

The ***strong component graph scc(G)*** is another directed graph obtained from G by contracting each strong component to a single vertex and collapsing parallel edges. (The strong component graph is sometimes also called the

meta-graph or *condensation* of G.) It's not hard to prove (hint, hint) that $scc(G)$ is always a dag. Thus, at least in principle, it is possible to topologically order the strong components of G; that is, the vertices can be ordered so that every *back* edge joins two edges in the same strong component.

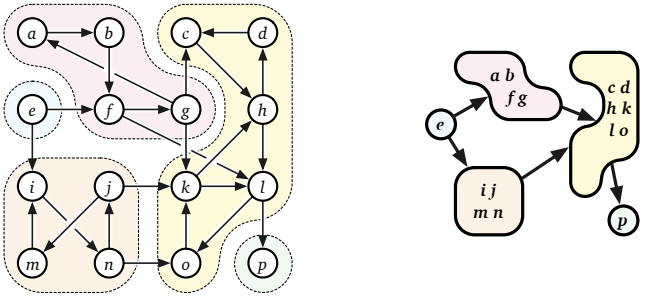

Figure 6.12. The strong components of a graph G and the strong component graph $scc(G)$.

It is straightforward to compute the strong component of a single vertex v in $O(V + E)$ time. First we compute $reach(v)$ via whatever-first search. Then we compute $reach^{-1}(v) = \{u \mid v \in reach(u)\}$ by searching the reversal of G. Finally, the strong component of v is the intersection $reach(v) \cap reach^{-1}(v)$. In particular, we can determine whether the entire graph is strongly connected in $O(V + E)$ time.

Similarly, we can compute *all* the strong components in a directed graph by combining the previous algorithm with our standard wrapper function. However, the resulting algorithm runs in $O(VE)$ time; there are at most V strong components, and each requires $O(E)$ time to discover, even when the graph is a dag. Surely we can do better! After all, we only need $O(V + E)$ time to decide whether every strong component is a single vertex.

6.6 Strong Components in Linear Time

In fact, there are several algorithms to compute strong components in $O(V + E)$ time, all of which rely on the following observation.

Lemma 6.2. *Fix a depth-first traversal of any directed graph G. Each strong component C of G contains exactly one node that does not have a parent in C. (Either this node has a parent in another strong component, or it has no parent.)*

Proof: Let C be an arbitrary strong component of G. Consider any path from one vertex $v \in C$ to another vertex $w \in C$. Every vertex on this path can reach w, and thus can reach every vertex in C; symmetrically, every node on this path can be reached by v, and thus can be reached by every vertex in C. We conclude that every vertex on this path is also in C.

Let v be the vertex in C with the earliest starting time. If v has a parent, then $parent(v)$ starts before v and thus cannot be in C.

Now let w be another vertex in C. Just before DFS(v) is called, every vertex in C is new, so there is a path of new vertices from v to w. Lemma 1 now implies that w is a descendant of v in the depth-first forest. Every vertex on the path of tree edges v to w lies in C; in particular, $parent(w) \in C$. □

The previous lemma implies that each strong component of a directed graph G defines a connected subtree of any depth-first forest of G. In particular, for any strong component C, the vertex in C with the earliest starting time is the lowest common ancestor of all vertices in C; we call this vertex the ***root*** of C.

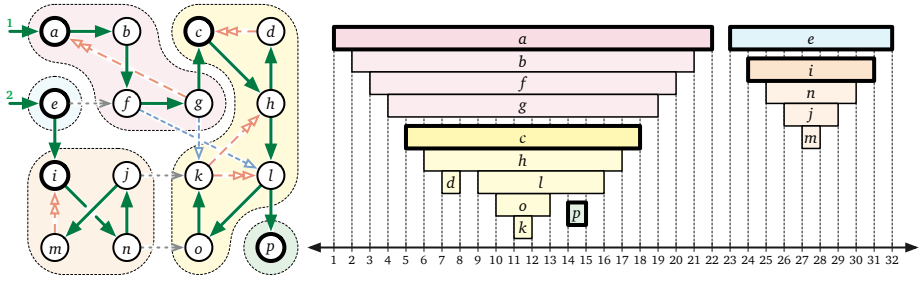

Figure 6.13. Strong components are contiguous in the depth-first forest.

I'll present two algorithms, both of which follow the same intuitive outline. Let C be any strong component of G that is a sink in $scc(G)$; we call C a ***sink component***. Equivalently, C is a sink component if the reach of any vertex in C is precisely C. We can find all the strong components in G by repeatedly finding a vertex v in some sink component (somehow), finding the vertices reachable from v, and removing that sink component from the input graph, until no vertices remain. This isn't quote an algorithm yet, because it's not clear how to find a vertex in a sink component!

```
StrongComponents(G):
  count ← 0
  while G is non-empty
      C ← ∅
      count ← count + 1
      v ← any vertex in a sink component of G     ⟨⟨Magic!⟩⟩
      for all vertices w in reach(v)
          w.label ← count
          add w to C
      remove C and its incoming edges from G
```

Figure 6.14. Almost an algorithm to compute strong components.

Kosaraju and Sharir's Algorithm

At first glance, finding a vertex in a sink component *quickly* seems quite difficult. However, it's actually quite easy to find a vertex in a *source* component—a strong component of G that corresponds to a *source* in $scc(G)$—using depth-first search.

Lemma 6.3. *The last vertex in any postordering of G lies in a source component of G.*

Proof: Fix a depth-first traversal of G, and let v be the last vertex in the resulting postordering. Then DFS(v) must be the last direct call to DFS made by the wrapper algorithm DFSALL. Moreover, v is the root of one of the trees in the depth-first forest, so any node x with $x.post > v.pre$ is a descendant of v. Finally, v is the root of its strong component C.

For the sake of argument, suppose there is an edge $x \to y$ such that $x \notin C$ and $y \in C$. Then x can reach y, and y can reach v, so x can reach v. Because v is the root of C, vertex y is a descendant of v, and thus $v.pre < y.pre$. The edge $x \to y$ guarantees that $y.pre < x.post$ and therefore $v.pre < x.post$. It follows that x is a descendant of v. But then v can reach x (through tree edges), contradicting our assumption that $x \notin C$. □

It is easy to check (hint, hint) that $rev(scc(G)) = scc(rev(G))$ for any directed graph G. Thus, the *last* vertex in a postordering of $rev(G)$ lies in a *sink* component of the original graph G. Thus, if we traverse the graph a second time, where the wrapper function follows a reverse postordering of $rev(G)$, then each call to DFS visits exactly one strong component of G.[4]

Putting everything together, we obtain the algorithm shown in Figure 6.15, which counts and labels the strong components of any directed graph in $O(V+E)$ time. This algorithm was discovered (but never published) by Rao Kosaraju in 1978, and later independently rediscovered by Micha Sharir in 1981.[5] The Kosaraju-Sharir algorithm has two phases. The first phase performs a depth-first search of $rev(G)$, pushing each vertex onto a stack when it is finished. In the second phase, we perform a *whatever*-first traversal of the original graph G, considering vertices in the order they appear on the stack. The algorithm labels each vertex with the root of its strong component (with respect to the second depth-first traversal).

Figure 6.16 shows the Kosaraju-Sharir algorithm running on our example graph. With only minor modifications to the algorithm, we can also compute the strong component graph $scc(G)$ in $O(V+E)$ time.

[4]Again: A reverse postordering of $rev(G)$ is not the same as a postordering of G.

[5]There are rumors that the same algorithm appears in the Russian literature even before Kosaraju, but I haven't found a reliable source for that rumor yet.

```
KosarajuSharir(G):
  S ← new empty stack
  for all vertices v
      unmark v
      v.root ← None
  ⟨⟨Phase 1: Push in postorder in rev(G)⟩⟩
  for all vertices v
      if v is unmarked
          PushPostRevDFS(v, S)
  ⟨⟨Phase 2: DFS again in stack order⟩⟩
  while S is non-empty
      v ← Pop(S)
      if v.root = None
          LabelOneDFS(v, v)
```

```
PushPostRevDFS(v, S):
  mark v
  for each edge u→v   ⟨⟨Reversed!⟩⟩
      if u is unmarked
          PushPostRevDFS(u, S)
  Push(v, S)
```

```
LabelOneDFS(v, r):
  v.root ← r
  for each edge v→w
      if w.root = None
          LabelOneDFS(w, r)
```

Figure 6.15. The Kosaraju-Sharir strong components algorithm

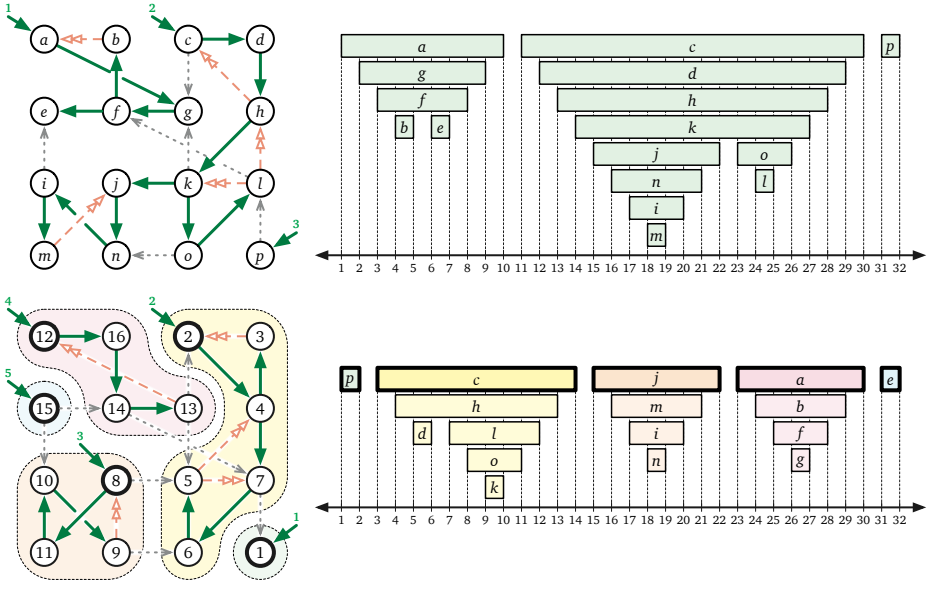

Figure 6.16. The Kosaraju-Sharir algorithm in action. Top: Depth-first traversal of the reversed graph. Bottom: Depth-first traversal of the original graph, visiting root vertices in reversed postorder from the first traversal.

♥Tarjan's Algorithm

An earlier but considerably more subtle linear-time algorithm to compute strong components was published by Bob Tarjan in 1972.[6] Intuitively, Tarjan's algorithm identifies a *source* component of G, "deletes" it, and then "recursively" finds the remaining strong components; however, the entire computation happens during a single depth-first search.

Fix an arbitrary depth-first traversal of some directed graph G. For each vertex v, let $\boldsymbol{low(v)}$ denote the smallest *starting time* among all vertices reachable from v by a path of tree edges followed by *at most one* non-tree edge. Trivially, $low(v) \le v.pre$, because v can reach itself through zero tree edges followed by zero non-tree edges. Tarjan observed that sink components can be characterized in terms of this *low* function.

Lemma 6.4. *A vertex v is the root of a sink component of G if and only if $low(v) = v.pre$ and $low(w) < w.pre$ for every proper descendant w of v.*

Proof: First, let v be a vertex such that $low(v) = v.pre$. Then there is no edge $w \to x$ where w is a descendant of v and $x.pre < v.pre$. On the other hand, v cannot reach any vertex y such that $y.pre > v.post$. It follows that v can reach only its descendants, and therefore any descendant of v can reach only descendants of v. In particular, v cannot reach its parent (if it has one), so v is the root of its strong component.

Now suppose in addition that $low(w) < w.pre$ for every descendant w of v. Then each descendant w can reach another vertex x (which must be another descendant of v) such that $x.pre < w.pre$. Thus, by induction, every descendant of v can reach v. It follows that the descendants of v comprise the strong component C whose root is v. Moreover, C must be a sink component, because v cannot reach any vertex outside of C.

On the other hand, suppose v is the root of a sink component C. Then v can reach another vertex w if and only if $w \in C$. But v can reach all of its descendants, and every vertex in C is a descendant of v, so v's descendants comprise C. If $low(w) = w.pre$ for any other node $w \in C$, then w would be another root of C, which is impossible. □

Computing $low(v)$ for every vertex v via depth-first search is straightforward; see Figure 6.17.

Lemma 6.4 implies that after running FINDLOW, we can identify the root of *every* sink component in $O(V + E)$ time (by a global whatever-first search),

[6] According to legend, Kosaraju apparently discovered his algorithm *during* an algorithms lecture; he was supposed to present Tarjan's algorithm, but he forgot his notes, so he had to make up something else on the fly. The only thing I find surprising about this story is that nobody tells it about Sharir or Tarjan.

```
FindLow(G):
  clock ← 0
  for all vertices v
      unmark v
      for all vertices v
          if v is unmarked
              FindLowDFS(v)
```

```
FindLowDFS(v):
  mark v
  clock ← clock + 1
  v.pre ← clock
  v.low ← v.pre
  for each edge v→w
      if w is unmarked
          FindLowDFS(w)
          v.low ← min{v.low, w.low}
      else
          v.low ← min{v.low, w.pre}
```

Figure 6.17. Computing *low*(v) for every vertex v.

and then mark and delete those sink components in $O(V + E)$ additional time (by calling whatever-first search at each root), and then recurse. Unfortunately, the resulting algorithm might require V iterations, each removing only a single vertex, naively giving us a total running time of $O(VE)$.

To speed up this strategy, Tarjan's algorithm maintains a stack of vertices (separate from the recursion stack). Whenever we start a new vertex v, we push it onto the stack. Whenever we finish a vertex v, we compare $v.low$ with $v.pre$. Then the *first* time we discover that $v.low = v.pre$, we know three things:

- Vertex v is the root of a sink component C.
- All vertices in C appear consecutively at the top of the stack.
- The *deepest* vertex in C on the stack is v.

At this point, we can identify the vertices in C by popping them off the stack one by one, stopping when we pop v.

We could delete the vertices in C and recursively compute the strong components of the remaining graph, but that would be wasteful, because we would repeat *verbatim* all computation done before visiting v. Instead, we *label* each vertex in C, identifying v as the root of its strong component, and then ignore labeled vertices for the rest of the depth-first search. Formally, this modification changes the definition of *low*(v) to the smallest starting time among all vertices ***in the same strong component as v*** that v can reach by a path of tree edges followed by at most one non-tree edge. But to prove correctness, it's easier to observe that ignoring labeled vertices leads the algorithm to exactly the same behavior as actually deleting them.

Finally, Tarjan's algorithm is shown in Figure 6.18, with the modifications from FindLow (Figure 6.17) indicated in bold red. The running time of the algorithm can be split into two parts. Each vertex is pushed onto S once and popped off S once, so the total time spent maintaining the stack (the red stuff) is $O(V)$. If we ignore the stack maintenance, the rest of the algorithm is just a

standard depth-first search. We conclude that the algorithm runs in ***O(V + E) time***.

```
TARJAN(G):
  clock ← 0
  S ← new empty stack
  for all vertices v
      unmark v
      v.root ← NONE
  for all vertices v
      if v is unmarked
          TARJANDFS(v)
```

```
TARJANDFS(v):
  mark v
  clock ← clock + 1
  v.pre ← clock
  v.low ← v.pre
  PUSH(S, v)
  for each edge v→w
      if w is unmarked
          TARJANDFS(w)
          v.low ← min{v.low, w.low}
      else if w.root = NONE
          v.low ← min{v.low, w.pre}
  if v.low = v.pre
      repeat
          w ← POP(S)
          w.root ← v
      until w = v
```

Figure 6.18. Tarjan's strong components algorithm.

Exercises

Depth-first search, topological sort, and strong components

o. (a) Describe an algorithm to compute the reversal $rev(G)$ of a directed graph in $O(V + E)$ time.

(b) Prove that for every directed graph G, the strong component graph $scc(G)$ is acyclic.

(c) Prove that $scc(rev(G)) = rev(scc(G))$ for every directed graph G.

(d) Fix an arbitrary directed graph G. For any vertex v of G, let $S(v)$ denote the strong component of G that contains v. For all vertices u and v of G, prove that v is reachable from u in G if and only if $S(v)$ is reachable from $S(u)$ in $scc(G)$.

(e) Suppose S and T are two strong components in a directed graph G. Prove that either $finish(u) < finish(v)$ for all vertices $u \in S$ and $v \in T$, or $finish(u) > finish(v)$ for all vertices $u \in S$ and $v \in T$.

1. A directed graph G is *semi-connected* if, for every pair of vertices u and v, either u is reachable from v or v is reachable from u (or both).

(a) Give an example of a dag with a unique source that is *not* semi-connected.

(b) Describe and analyze an algorithm to determine whether a given directed *acyclic* graph is semi-connected.

(c) Describe and analyze an algorithm to determine whether an arbitrary directed graph is semi-connected.

2. The police department in the city of Shampoo-Banana has made every street in the city one-way. Despite widespread complaints from confused motorists, the mayor claims that it is possible to legally drive from any intersection in Shampoo-Banana to any other intersection.

   (a) The city needs to either verify or refute the mayor's claim. Formalize this problem in terms of graphs, and then describe and analyze an algorithm to solve it.

   (b) After running your algorithm from part (a), the mayor reluctantly admits that she was ~~lying~~ misinformed. Call an intersection x *good* if, for any intersection y that one can legally reach from x, it is possible to legally drive from y back to x. Now the mayor claims that over 95% of the intersections in Shampoo-Banana are good. Describe and analyze an efficient algorithm to verify or refute her claim.

   For full credit, both algorithms should run in linear time.

3. A vertex v in a connected undirected graph G is called a ***cut vertex*** if the subgraph $G - v$ (obtained by removing v from G) is disconnected.

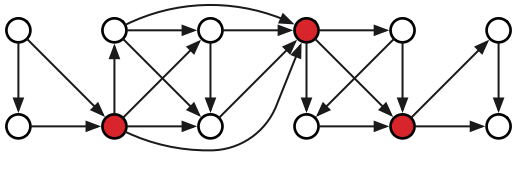

Figure 6.19. A dag with three cut vertices.

   (a) Describe a linear-time algorithm that determines, given a graph G and a vertex v, whether v is a cut vertex in G. What is the running time to find all cut vertices by trying your algorithm for each vertex?

   (b) Describe and analyze an algorithm that correctly determines whether a given directed *acyclic* graph contains a cut vertex. *[Hint: This is a warmup.]*

   (c) Let T be an arbitrary spanning tree of G, rooted at an arbitrary vertex r. For each vertex v, let T_v denote the subtree of T rooted at v; for example, $T_r = T$. Prove that v is a cut vertex of G if and only if G does not have an edge with exactly one endpoint in T_v.

(d) Describe a linear-time algorithm to identify every cut vertex in G. *[Hint: Let T be a depth-first spanning tree of G.]*

4. An edge e in a connected undirected graph G is called a ***cut edge*** if the subgraph $G - e$ (obtained by removing e from G) is disconnected.

   (a) Given G and edge e describe a linear-time algorithm that determines whether e is a cut edge or not. What is the running time to find all cut edges by trying your algorithm for each edge?

   (b) Let T be an arbitrary spanning tree of G. Prove that *every* cut edge of G is also an edge in T. This claim implies that G has at most $V - 1$ cut edges. How does this information improve your algorithm from part (a) to find all cut-edges?

   (c) Now suppose we root T at an arbitrary vertex r. For any vertex v, let T_v denote the subtree of T rooted at v; for example, $T_r = T$. Let uv be an arbitrary edge of T, where u is the parent of v. Prove that uv is a cut edge of G if and only if uv is the only edge in G with exactly one endpoint in T_v.

   (d) Describe a linear-time algorithm to identify every cut edge in G. *[Hint: Let T be a depth-first spanning tree of G.]*

5. The ***transitive closure G^T*** of a directed graph G is a directed graph with the same vertices as G, that contains any edge $u \to v$ if and only if there is a directed path from u to v in G. A ***transitive reduction*** of G is a graph with the smallest possible number of edges whose transitive closure is G^T. The same graph may have several transitive reductions.

   (a) Describe an efficient algorithm to compute the transitive closure of a given directed graph.

   (b) Prove that a directed graph G has a *unique* transitive reduction if and only if G is acyclic.

   (c) Describe an efficient algorithm to compute a transitive reduction of a given directed graph.

6. One of the oldest algorithms for exploring arbitrary connected graphs was proposed by Gaston Tarry in 1895, as a procedure for solving mazes.[7] The input to Tarry's algorithm is an undirected graph G; however, for ease of presentation, we formally split each undirected edge uv into two directed edges $u \to v$ and $v \to u$. (In an actual implementation, this split is trivial;

[7] Even older graph-traversal algorithms were described by Charles Trémaux in 1882, by Christian Wiener in 1873, and (implicitly) by Leonhard Euler in 1736. Wiener's algorithm is equivalent to depth-first search in a connected undirected graph.

the algorithm simply uses the given adjacency list for G *as though* G were directed.)

```
TARRY(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s)
```

```
RECTARRY(v):
  mark v                                  ⟨⟨"visit v"⟩⟩
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red                       }
      RECTARRY(w)                         } ⟨⟨"traverse v→w"⟩⟩
  else if there is a green arc v→w
      color v→w red                       }
      RECTARRY(w)                         } ⟨⟨"traverse v→w"⟩⟩
```

We informally say that Tarry's algorithm "visits" vertex v every time it marks v, and it "traverses" edge $v \to w$ when it colors that edge red and recursively calls RECTARRY(w). Unlike our earlier graph traversal algorithm, Tarry's algorithm can mark same vertex multiple times.

(a) Describe how to implement Tarry's algorithm so that it runs in $O(V + E)$ time.

(b) Prove that no directed edge in G is traversed more than once.

(c) When the algorithm visits a vertex v for the kth time, exactly how many edges into v are red, and exactly how many edges out of v are red? *[Hint: Consider the starting vertex s separately from the other vertices.]*

(d) Prove each vertex v is visited at most $\deg(v)$ times, except the starting vertex s, which is visited at most $\deg(s)+1$ times. This claim immediately implies that TARRY(G) terminates.

(e) Prove that the last vertex visited by TARRY(G) is the starting vertex s.

(f) For every vertex v that TARRY(G) visits, prove that all edges into v and out of v are red when TARRY(G) halts. *[Hint: Consider the vertices in the order that they are marked for the first time, starting with s, and prove the claim by induction.]*

(g) Prove that TARRY(G) visits every vertex of G. This claim and the previous claim imply that TARRY(G) traverses every edge of G exactly once.

7. Consider the following variant of Tarry's graph-traversal algorithm; this variant traverses green edges without recoloring them red and assigns two numerical labels to every vertex:

```
TARRY2(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s, 1)
```

```
RECTARRY2(v, clock):
  if v is unmarked
      v.pre ← clock; clock ← clock + 1
      mark v
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red
      RECTARRY2(w, clock)
  else if there is a green arc v→w
      v.post ← clock; clock ← clock + 1
      RECTARRY2(w, clock)
```

Prove or disprove the following claim: When TARRY2(G) halts, the green edges define a spanning tree and the labels $v.pre$ and $v.post$ define a preorder and postorder labeling that are all consistent with a single depth-first search of G. In other words, prove or disprove that TARRY2 produces the same *output* as depth-first search, even though it visits the edges in a completely different order.

8. You have a collection of n lock-boxes and m gold keys. Each key unlocks *at most* one box. However, each box might be unlocked by one key, by multiple keys, or by no keys at all. There are only two ways to open each box once it is locked: Unlock it properly (which requires having one matching key in your hand), or smash it to bits with a hammer.

   Your baby brother, who loves playing with shiny objects, has somehow managed to lock all your keys inside the boxes! Luckily, your home security system recorded everything, so you know exactly which keys (if any) are inside each box. You need to get all the keys back out of the boxes, because they are made of gold. Clearly you have to smash at least one box.

   (a) Your baby brother has found the hammer and is eagerly eyeing one of the boxes. Describe and analyze an algorithm to determine if it is possible to retrieve all the keys without smashing any box except the one your brother has chosen.

   (b) Describe and analyze an algorithm to compute the minimum number of boxes that must be smashed to retrieve all the keys.

9. Suppose you are teaching an algorithms course. In your second midterm, you give your students a drawing of a graph and ask then to indicate a breadth-first search tree and a depth-first search tree rooted at a particular vertex. Unfortunately, once you start grading the exam, you realize that the graph you gave the students has several such spanning trees—far too many

to list. Instead, you need a way to tell whether each student's submission is correct!

In each of the following problems, suppose you are given a connected graph G, a start vertex s, and a spanning tree T of G.

(a) Suppose G is *undirected*. Describe and analyze an algorithm to decide whether T is a *depth*-first spanning tree rooted at s.

(b) Suppose G is *undirected*. Describe and analyze an algorithm to decide whether T is a *breadth*-first spanning tree rooted at s. *[Hint: It's not enough for T to be an unweighted shortest-path tree. Yes, this is the right chapter for this problem!]*

(c) Suppose G is *directed*. Describe and analyze an algorithm to decide whether T is a *breadth*-first spanning tree rooted at s. *[Hint: Solve part (b) first.]*

(d) Suppose G is *directed*. Describe and analyze an algorithm to decide whether T is a *depth*-first spanning tree rooted at s.

10. Several modern programming languages, including JavaScript, Python, Perl, and Ruby, include a feature called ***parallel assignment***, which allows multiple assignment operations to be encoded in a single line of code. For example, the Python code `x,y = 0,1` simultaneously sets `x` to 0 and `y` to 1. The values of the right-hand side of the assignment are all determined by the *old* values of the variables. Thus, the Python code `a,b = b,a` swaps the values of `a` and `b`, and the following Python code computes the nth Fibonacci number:

```
def fib(n):
    prev, curr = 1, 0
    while n > 0:
        prev, curr, n = curr, prev+curr, n-1
    return curr
```

Suppose the interpreter you are writing needs to convert every parallel assignment into an equivalent sequence of individual assignments. For example, the parallel assignment `a,b = 0,1` can be serialized in either order—either `a=0; b=1` or `a=0; b=1`—but the parallel assignment `x,y = x+1,x+y` can only be serialized as `y=x+y; x=x+1`. Serialization may require one or more additional temporary variables; for example, serializing `a,b = b,a` requires one temporary variable, and serializing `x,y = x+y,x-y` requires two temporary variables.

(a) Describe an algorithm to determine whether a given parallel assignment can be serialized without additional temporary variables.

(b) Describe an algorithm to determine whether a given parallel assignment can be serialized with *exactly one* additional temporary variable.

Assume that the given parallel assignment involves only simple integer variables (no indirection via pointers or arrays); no variable appears on the left side more than once; and expressions on the right side have no side effects. Don't worry about the details of parsing the assignment statement; just assume (but describe!) an appropriate graph representation.

Dynamic Programming

11. Suppose we are given a directed acyclic graph G whose nodes represent jobs and whose edges represent precedence constraints; that is. each edge $u \to v$ indicates the job u must be completed before job v begins. Each node v also has a weight $T(v)$ indicating the time required to execute job v.

    (a) Describe an algorithm to determine the shortest interval of time in which all jobs in G can be executed.

    (b) Suppose the first job starts at time 0. Describe an algorithm to determine, for each vertex v, the earliest time when job v can begin.

    (c) Now describe an algorithm to determine, for each vertex v, the *latest* time when job v can begin without violating the precedence constraints or increasing the overall completion time (computed in part (a)), assuming that every job except v starts at its earliest start time (computed in part (b)).

12. Let G be a directed acyclic graph with a unique source s and a unique sink t.

    (a) A *Hamiltonian path* in G is a directed path in G that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

    (b) Suppose the *vertices* of G have weights. Describe an efficient algorithm to find the path from s to t with maximum total weight.

    (c) Suppose we are also given an integer ℓ. Describe an efficient algorithm to find the maximum-weight path from s to t that contains at most ℓ edges. (Assume there is at least one such path.)

    (d) Suppose some of the vertices of G are marked as *important*, and we are also given an integer k. Describe an efficient algorithm to find the maximum-weight path from s to t that visits at least k important vertices. (Assume there is at least one such path.)

    (e) Describe an algorithm to compute the number of paths from s to t in G. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

13. Let G be a directed acyclic graph whose vertices have labels from some fixed alphabet, and let $A[1..\ell]$ be a string over the same alphabet. Any directed

path in G has a label, which is a string obtained by concatenating the labels of its vertices.

(a) Describe an algorithm that either finds a path in G whose label is A or correctly reports that there is no such path.

(b) Describe an algorithm to find the *number* of paths in G whose label is A. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

(c) Describe an algorithm to find the longest path in G whose label is a subsequence of A.

(d) Describe an algorithm to find the *shortest* path in G whose label is a *supersequence* of A.

(e) Describe an algorithm to find a path in G whose label has minimum edit distance from A.

14. A ***polygonal path*** is a sequence of line segments joined end-to-end; the endpoints of these line segments are called the ***vertices*** of the path. The ***length*** of a polygonal path is the sum of the lengths of its segments. A polygonal path with vertices $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ is ***monotonically increasing*** if $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for every index i—informally, each vertex of the path is above and to the right of its predecessor.

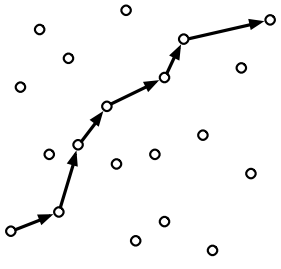

Figure 6.20. A monotonically increasing polygonal path with seven vertices through a set of points

Suppose you are given a set S of n points in the plane, represented as two arrays $X[1..n]$ and $Y[1..n]$. Describe and analyze an algorithm to compute the length of the maximum-length monotonically increasing path with vertices in S. Assume you have a subroutine $\textsc{Length}(x, y, x', y')$ that returns the length of the segment from (x, y) to (x', y').

15. For any two nodes u and v in a directed acyclic graph G, the ***interval*** $G[u, v]$ is the union of all directed paths in G from u to v. Equivalently, $G[u, v]$ consists of all vertices x such that $x \in \mathit{reach}(u)$ and $v \in \mathit{reach}(x)$, together with all the edges in G connecting those vertices.

Suppose we are given a directed acyclic graph G, in which every edge has a numerical weight, which may be positive, negative, or zero. Describe

an efficient algorithm to find the maximum-weight interval in G, where the weight of any interval is the sum of the weights of its vertices.

16. Let G be a directed acyclic graph whose vertices have labels from some fixed alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices. Recall that a *palindrome* is a string that is equal to its reversal.

    (a) Describe and analyze an algorithm to find the length of the longest palindrome that is the label of a path in G. For example, given the graph in Figure 6.21, your algorithm should return the integer 6, which is the length of the palindrome HANNAH.

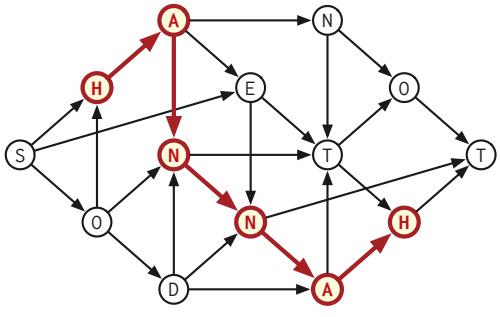

Figure 6.21. A dag whose longest palindrome path label has length 6.

    (b) Describe an algorithm to find the longest palindrome that is a subsequence of the label of a path in G.

    (c) Describe an algorithm to find the shortest palindrome that is a supersequence of the label of a path in G.

17. Suppose you are given two directed acyclic graphs G and H in which every node has a *label* from some finite alphabet; different nodes may have the same label. The label of a *path* in either dag is the string obtained by concatenating the labels of its vertices.

    (a) Describe and analyze an algorithm to compute the length of the longest string that is both the label of a path in G and the label of a path in H.

    (b) Describe and analyze an algorithm to compute the length of the longest string that is both a subsequence of the label of a path in G both a subsequence of the label of a path in H.

    (c) Describe and analyze an algorithm to compute the length of the shortest string that is both a supersequence of the label of a path in G both a supersequence of the label of a path in H.

18. Let G be an arbitrary (*not* necessarily acyclic) directed graph in which every vertex v has an integer weight $w(v)$.

    (a) Describe an algorithm to find the longest directed path in G whose vertex weights define an increasing sequence.

    (b) Describe and analyze an algorithm to determine the maximum-weight vertex reachable from each vertex in G. That is, for each vertex v, your algorithm needs to compute $\max\{w(x) \mid x \in \mathit{reach}(v)\}$.

19. Suppose you are given a directed graph G in which ***every edge has negative weight***, and a source vertex s. Describe and analyze an efficient algorithm that computes the shortest-path distances from s to every other vertex in G. Specifically, for every vertex t:

    - If t is not reachable from s, your algorithm should report $\mathit{dist}(t) = \infty$.
    - If G has a cycle that is reachable from s, and t is reachable from that cycle, then the shortest-path distance from s to t is not well-defined, because there are paths (formally, walks) from s to t of arbitrarily large negative length. In this case, your algorithm should report $\mathit{dist}(t) = -\infty$.
    - If neither of the two previous conditions applies, your algorithm should report the correct shortest-path distance from s to t.

    [Hint: This problem may be easier after you've read about shortest paths in Chapter **??***. First think about graphs where the first two conditions never happen.]*

20. (a) Suppose you are given a directed acyclic graph G with n vertices and an integer $k \leq n$. Describe an efficient algorithm to find a set of at most k vertex-disjoint paths that visit every vertex in G.

    (b) Now suppose the edges of the input dag G have weights, which may be positive, negative, or zero. Describe an efficient algorithm to find a set of at most k vertex-disjoint paths *with minimum total weight* that visit every vertex in G.

    Your algorithms should run in $O(n^{k+c})$ time for some small constant c. A single vertex is a path with weight zero. (We will see a more efficient algorithm for part (a) in Chapter **??**.)

21. Kris is a professional rock climber who is competing in the U.S. climbing nationals. The competition requires Kris to use as many holds on the climbing wall as possible, using only transitions that have been explicitly allowed by the route-setter.

    The climbing wall has n holds. Kris is given a list of m pairs (x, y) of holds, each indicating that moving directly from hold x to hold y is allowed;

however, moving directly from y to x is not allowed unless the list also includes the pair (y, x). Kris needs to figure out a sequence of allowed transitions that uses as many holds as possible, since each new hold increases his score by one point. The rules allow Kris to choose the first and last hold in his climbing route. The rules also allow him to use each hold as many times as he likes; however, only the first use of each hold increases Kris's score.

(a) Define the natural graph representing the input. Describe and analyze an algorithm to solve Kris's climbing problem if you are guaranteed that the input graph is a dag.

(b) Describe and analyze an algorithm to solve Kris's climbing problem with no restrictions on the input graph.

Both of your algorithms should output the maximum possible score that Kris can earn.

22. There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. However, the company that runs the teleport-ways has established an extremely lucrative cost structure: Anyone can teleport *further* from their home galaxy at no cost whatsoever, but teleporting *toward* their home galaxy is prohibitively expensive.

Judy has decided to take a sabbatical tour of the universe by visiting as many galaxies as possible, starting at her home galaxy. To save on travel expenses, she wants to teleport away from her home galaxy at every step, except for the very last teleport home.

(a) Describe and analyze an algorithm to compute the maximum number of galaxies that Judy can visit. Your input consists of an undirected graph G with n vertices and m edges describing the teleport-way network, an integer $1 \le s \le n$ identifying Judy's home galaxy, and an array $D[1..n]$ containing the distances of each galaxy from s.

♥(b) Just before embarking on her universal tour, Judy wins the space lottery, giving her just enough money to afford *two* teleports toward her home galaxy. Describe a new algorithm to compute the maximum number of distinct galaxies Judy can visit. She can visit the same galaxy more than once, but crucially, only the first visit counts toward her total.

23. The Doctor and River Song decide to play a game on a directed acyclic graph G, which has one source s and one sink t.[8]

[8]The labels s and t are abbreviations for the Untempered **S**chism and the **T**ime Vortex, or the Shining World of the Seven Systems (also known as Gallifrey) and Trenzalore, or Skaro and Telos, or Something else Timey-wimey. It's all very complicated, never mind.

Each player has a token on one of the vertices of G. At the start of the game, The Doctor's token is on the source vertex s, and River's token is on the sink vertex t. The players alternate turns, with The Doctor moving first. On each of his turns, the Doctor moves his token forward along a directed edge; on each of her turns, River moves her token *backward* along a directed edge.

If the two tokens ever meet on the same vertex, River wins the game. ("Hello, Sweetie!") If the Doctor's token reaches t or River's token reaches s before the two tokens meet, then the Doctor wins the game.

Describe and analyze an algorithm to determine who wins this game, assuming both players play perfectly. That is, if the Doctor can win *no matter how River moves*, then your algorithm should output "Doctor", and if River can win *no matter how the Doctor moves*, your algorithm should output "River". (Why are these the only two possibilities?) The input to your algorithm is the graph G.

♣♥24. Let $x = x_1 x_2 \ldots x_n$ be a given n-character string over some finite alphabet Σ, and let A be a deterministic finite-state machine with m states over the same alphabet.

(a) Describe and analyze an algorithm to compute the length of the longest subsequence of x that is accepted by A. For example, if A accepts the language $(\mathtt{AR})^*$ and $x = \mathtt{ABRACADABRA}$, your algorithm should output the number 4, which is the length of the subsequence ARAR.

(b) Describe and analyze an algorithm to compute the length of the shortest supersequence of x that is accepted by A. For example, if A accepts the language $(\mathtt{ABCDR})^*$ and $x = \mathtt{ABRACADABRA}$, your algorithm should output the number 25, which is the length of the supersequence ABCDRABCDRABCDRABCDRABCDR.

Analyze your algorithms in terms of the length n of the input string, the number m of states in the finite-state machine, and the size of the alphabet Σ.

25. Not *every* dynamic programming algorithm can be modeled as finding an optimal path through a directed acyclic graph, but every dynamic programming algorithm does process some underlying dependency graph in postorder.

(a) Suppose we are given a directed acyclic graph G where every node stores a numerical search key. Describe and analyze an algorithm to find the largest binary search tree that is a subgraph of G.

(b) Suppose we are given a directed acyclic graph G and two vertices s and t. Describe an algorithm to compute the number of directed paths in G from s to t. (Assume that any arithmetic operation requires $O(1)$ time.)

(c) Let G be a directed acyclic graph with the following features:

- G has a single source s and several sinks $t_1, t_2, \ldots, t_k$.
- Each edge $v \to w$ has an associated weight $p(v \to w)$ between 0 and 1.
- For each non-sink vertex v, the total weight of all edges leaving v is 1; that is, $\sum_w p(v \to w) = 1$.

The weights $p(v \to w)$ define a random walk in G from the source s to some sink t_i; after reaching any non-sink vertex v, the walk follows edge $v \to w$ with probability $p(v \to w)$. All probabilities are mutually independent. Describe and analyze an algorithm to compute the probability that this random walk reaches sink t_i, for every index i. (Assume that each arithmetic operation takes only $O(1)$ time.)