Ts'ui Pe must have said once: I am withdrawing to write a book.
And another time: I am withdrawing to construct a labyrinth.
Every one imagined two works;
to no one did it occur that the book and the maze were one and the same thing.

— Jorge Luis Borges, "El jardín de senderos que se bifurcan" (1942)
English translation ("The Garden of Forking Paths") by Donald A. Yates (1958)

"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)

At some point, the learning stops and the pain begins.

— Rao Kosaraju

19 Depth-First Search

Recall from the previous lecture the recursive formulation of depth-first search in undirected graphs.

```
DFS(v):
  if v is unmarked
      mark v
      for each edge vw
          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 define parent pointers and other useful information about the vertices. This additional information is computed by 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)
```

We can search any *connected* graph by unmarking all vertices and then calling DFS(s) for an arbitrary start vertex s. As we argued in the previous lecture, the subgraph of all parent edges $v \to parent(v)$ defines a spanning tree of the graph, which we consider to be rooted at the start vertex s.

Lemma 1. *Let T be a depth-first spanning tree of a connected undirected graph G, computed by calling DFS(s). For any node v, the vertices that are marked during the execution of DFS(v) are the proper descendants of v in T.*

Proof: T is also the recursion tree for DFS(s). □

Lemma 2. *Let T be a depth-first spanning tree of a connected undirected graph G. For every edge vw in G, either v is an ancestor of w in T, or v is a descendant of w in T.*

Proof: Assume without loss of generality that v is marked before w. Then w is unmarked when DFS(v) is invoked, but marked when DFS(v) returns, so the previous lemma implies that w is a proper descendant of v in T. □

Lemma 2 implies that any depth-first spanning tree T divides the edges of G into two classes: *tree* edges, which appear in T, and *back* edges, which connect some node in T to one of its ancestors.

19.1 Counting and Labeling Components

For graphs that might be disconnected, we can compute a depth-first spanning *forest* by calling the following wrapper function; again, we introduce a generic black-box subroutine PREPROCESS to perform any necessary preprocessing for the POSTVISIT and POSTVISIT functions.

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

With very little additional effort, we can count the components of a graph; we simply increment a counter inside the wrapper function. Moreover, we can also record which component contains each vertex in the graph by passing this counter to DFS. The single line $comp(v) \leftarrow count$ is a trivial example of PREVISIT. (And the absence of code after the for loop is a vacuous example of POSTVISIT.)

```
COUNTANDLABEL(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          LABELCOMPONENT(v, count)
  return count
```

```
LABELCOMPONENT(v, count):
  mark v
  comp(v) ← count
  for each edge vw
      if w is unmarked
          LABELCOMPONENT(w, count)
```

It should be emphasized that depth-first search is not specifically required here; any other instantiation of our earlier generic traversal algorithm ("whatever-first search") can be used to count components in the same asymptotic running time. However, most of the other algorithms we consider in this note *do* specifically require *depth*-first search.

19.2 Preorder and Postorder Labeling

You should already be familiar with preorder and postorder traversals of rooted trees, both of which can be computed using from depth-first search. Similar traversal orders can be defined for arbitrary graphs by passing around a counter as follows:

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

```
LABELCOMPONENT(v, clock):
  mark v
  pre(v) ← clock
  clock ← clock + 1
  for each edge vw
      if w is unmarked
          clock ← LABELCOMPONENT(w, clock)
  post(v) ← clock
  clock ← clock + 1
  return clock
```

Equivalently, if we're willing to use (shudder) global variables, we can use our generic depth-first-search algorithm with the following subroutines PREPROCESS, PREVISIT, and POSTVISIT.

```
PREPROCESS(G):
  clock ← 0
```

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

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

Consider two vertices u and v, where u is marked after v. Then we must have $pre(u) < pre(v)$. Moreover, Lemma 1 implies that if v is a descendant of u, then $post(u) > post(v)$, and otherwise, $pre(v) > post(u)$. Thus, for any two vertices u and v, the intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or nested; in particular, if uv is an edge, Lemma 2 implies that the intervals must be nested.

19.3 Directed Graphs and Reachability

The recursive algorithm requires only one minor change to handle directed graphs:

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

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

However, we can no longer use this modified algorithm to count components. Suppose G is a single directed path. Depending on the order that we choose to visit the nodes in DFSALL, we may discover any number of "components" between 1 and n. All that we can guarantee is that the "component" numbers computed by DFSALL do not increase as we traverse the path. In fact, the real problem is that the *definition* of "component" is only suitable for *undirected* graphs.

Instead, for directed graphs we rely on a more subtle notion of ***reachability***. We say that a node v is *reachable* from another node u in a directed graph G—or more simply, that u can reach v—if and only if there is a directed path in G from u to v. Let ***Reach(u)*** denote the set of vertices that are reachable from u (including u itself). A simple inductive argument proves that $Reach(u)$ is precisely the subset of nodes that are marked by calling DFS(u).

19.4 Directed Acyclic Graphs

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***. Every dag has at least one source and one sink (Do you see why?), 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.

We can check whether a given directed graph G is a dag in $O(V + E)$ time as follows. First, to simplify the algorithm, we add a single artificial source s, with edges from s to every other vertex. Let $G + s$ denote the resulting augmented graph. Because s has no outgoing edges, no directed cycle in $G + s$ goes through s, which implies that $G + s$ is a dag if and only if G is a dag. Then we preform a depth-first search of $G + s$ starting at the new source vertex s; by construction every other vertex is reachable from s, so this search visits every node in the graph.

Instead of vertices being merely marked or unmarked, each vertex has one of three statuses—NEW, ACTIVE, or DONE—which depend on whether we have started or finished the recursive depth-first search at that vertex. (Since this algorithm never uses parent pointers, I've removed the line "$parent(w) \leftarrow v$".)

```
IsAcyclic(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← NEW
  return IsAcyclicDFS(s)
```

```
IsAcyclicDFS(v):
  status(v) ← ACTIVE
  for each edge v→w
      if status(w) = ACTIVE
          return FALSE
      else if status(w) = NEW
          if IsAcyclicDFS(w) = FALSE
              return FALSE
  status(v) ← DONE
  return TRUE
```

Suppose the algorithm returns FALSE. Then the algorithm must discover an edge $v \to w$ such that $status(w) = \text{ACTIVE}$. The active vertices are precisely the vertices currently on the recursion stack, which are all ancestors of the current vertex v. Thus, there is a directed path from w to v, and so the graph has a directed cycle.

On the other hand, suppose G has a directed cycle. Let w be the first vertex in this cycle that we visit, and let $v \to w$ be the edge leading into v in the same cycle. Because there is a directed path from w to v, we must call IsAcyclicDFS(v) during the execution of IsAcyclicDFS(w), unless we discover some other cycle first. During the execution of IsAcyclicDFS(v), we consider the edge $v \to w$, discover that $status(w) = \text{ACTIVE}$. The return value FALSE bubbles up through all the recursive calls to the top level.

We conclude that IsAcyclic(G) returns TRUE if and only if G is a dag.

19.5 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, every dag has a topological order, which can be computed by either of the following algorithms.

```
TopologicalSort(G):
  n ← |V|
  for i ← 1 to n
      v ← any source in G
      S[i] ← v
      delete v and all edges leaving v
  return S[1..n]
```

```
TopologicalSort(G):
  n ← |V|
  for i ← n down to 1
      v ← any sink in G
      S[1] ← v
      delete v and all edges entering v
  return S[1..n]
```

The correctness of these algorithms follow inductively from the observation that *deleting* a vertex cannot *create* a cycle.

This simple algorithm has two major disadvantages. First, the algorithm actually destroys the input graph. This destruction can be avoided by simply marking the "deleted" vertices, instead of actually deleting them, and defining a vertex to be a source (sink) if none of its incoming (outgoing) edges come from (lead to) an unmarked vertex. The more serious problem is that finding a source vertex seems to require $\Theta(V)$ time in the worst case, which makes the running time of this algorithm $\Theta(V^2)$. In fact, a careful implementation of this algorithm computes a topological ordering in $O(V + E)$ time without removing any edges.

But there is a simpler linear-time algorithm based on our earlier algorithm for deciding whether a directed graph is acyclic. The new algorithm is based on the following observation:

Lemma 3. *For any directed acyclic graph G, the first vertex marked* Done *by* IsAcyclic(G) *must be a sink.*

Proof: Let v be the first vertex marked Done during an execution of IsAcyclic. For the sake of argument, suppose v has an outgoing edge $v \to w$. When IsAcyclicDFS first considers the edge $v \to w$, there are three cases to consider.

- If $status(w) =$ Done, then w is marked Done before v, which contradicts the definition of v.
- If $status(w) =$ New, the algorithm calls TopoSortDFS(w), which (among other computation) marks w Done. Thus, w is marked Done before v, which contradicts the definition of v.
- If $status(w) =$ Active, then G has a directed cycle, contradicting our assumption that G is acyclic.

In all three cases, we have a contradiction, so v must be a sink. □

Thus, to topologically sort a dag G, it suffice to list the vertices in the *reverse* order of being marked Done. For example, we could push each vertex onto a stack when we mark it Done, and then pop every vertex off the stack.

```
TopologicalSort(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← New
  TopoSortDFS(s)
  for i ← 1 to V
      S[i] ← Pop
  return S[1..V]
```

```
TopoSortDFS(v):
  status(v) ← Active
  for each edge v→w
      if status(w) = New
          ProcessBackwardDFS(w)
      else if status(w) = Active
          fail gracefully
  status(v) ← Done
  Push(v)
  return True
```

But maintaining a separate data structure is actually overkill. In most applications of topological sort, an explicit sorted list of the vertices is not our actual goal; instead, we want to performing some fixed computation at each vertex of the graph, either in topological order or in reverse topological order. In this case, it is not necessary to *record* the topological order. To process the graph in *reverse* topological order, we can just process each vertex at the end of its recursive depth-first search.

```
ProcessBackward(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← New
  ProcessBackwardDFS(s)
```

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

If we already *know* that the input graph is acyclic, we can simplify the algorithm by simply marking vertices instead of labeling them Active or Done.

```
ProcessDagBackward(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      unmark v
  ProcessDagBackwardDFS(s)
```

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

Except for the addition of the artificial source vertex s, which we need to ensure that every vertex is visited, this is just the standard depth-first search algorithm, with PostVisit renamed to Process!

The simplest way to process a dag in *forward* topological order is to construct the ***reversal*** of the input graph, which is 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 becomes a sink in the reversal of G and vice versa; it follows inductively that every topological ordering for the reversal of G is the reversal of a topological ordering of G. The reversal of any directed graph can be computed in $O(V + E)$ time; the details of this construction are left as an easy exercise.

19.6 Every Dynamic Programming Algorithm?

Our topological sort algorithm is arguably the model fora 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, and the black-box subroutine Process is a placeholder for the actual value computation.

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 given as part of the input. 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, the dependency graph for edit distance 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 lets us hard-wire a topological order directly into the algorithm, so we don't have to compute it at run time.

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 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. For any node s, let $LLP(s,t)$ denote the Length of the Longest Path in G from s to t. If G is a dag, this function satisfies the recurrence

$$LLP(s,t) = \begin{cases} 0 & \text{if } s = t, \\ \max_{s \to v} (\ell(s \to v) + LLP(v,t)) & \text{otherwise,} \end{cases}$$

where $\ell(v \to w)$ is the given weight ("length") of edge $v \to w$. In particular, if s is a *sink* but not equal to t, then $LLP(s,t) = \infty$. The dependency graph for this recurrence is the input graph G itself: subproblem $LLP(u,t)$ depends on subproblem $LLP(v,t)$ if and only if $u \to v$ 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.

```
LongestPath(s, t):
  if s = t
      return 0

  if LLP(s) is undefined
      LLP(s) ← ∞
      for each edge s→v
          LLP(s) ← max{LLP(v), ℓ(s→v) + LongestPath(v, t)}
  return LLP(s)
```

A surprisingly large number of dynamic programming problems (but *not* all) can be recast as optimal path problems in the associated dependency graph.

19.7 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 graph G if there is a directed path in G from u to v, and that $Reach(u)$ 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. Tedious definition-chasing implies that strong connectivity is an equivalence relation over the set of vertices of any directed graph, just as connectivity is for undirected graphs. The equivalence classes of this relation are called the ***strongly connected components*** (or more simply, the ***strong components***) of G. If G has a single strong component, we call it ***strongly connected***. G is a directed acyclic graph if and only if every strong component of G is a single vertex.

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

We can compute *all* the strong components in a directed graph by wrapping the single-strong-component algorithm in a wrapper function, just as we did for depth-first search in undirected graphs. However, the resulting algorithm runs in $O(VE)$ time; there are at most V strong components, and each requires $O(E)$ time to discover. Surely we can do better! After all, we only need $O(V+E)$ time to decide whether every strong component is a single vertex.

19.8 Strong Components in Linear Time

For any directed graph G, the ***strong component graph scc(G)*** is another directed graph obtained by contracting each strong component of G to a single (meta-)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 $\mathit{scc}(G)$ is always a dag. Thus, in principle, it is possible to topologically order the strong components of G; that is, the vertices can be ordered so that every *backward* edge joins two edges in the same strong component.

Let C be any strong component of G that is a sink in $\mathit{scc}(G)$; we call C a *sink component*. Every vertex in C can reach every other vertex in C, so a depth-first search from any vertex in C visits every vertex in C. On the other hand, because C is a sink component, there is no edge from C to any other strong component, so a depth-first search starting in C visits *only* vertices in C. So if we can compute all the strong components as follows:

```
StrongComponents(G):
  count ← 0
  while G is non-empty
      count ← count + 1
      v ← any vertex in a sink component of G
      C ← OneComponent(v, count)
      remove C and incoming edges from G
```

At first glance, finding a vertex in a sink component *quickly* seems quite hard. However, we can quickly find a vertex in a *source* component using the standard depth-first search. A source component is a strong component of G that corresponds to a source in $\mathit{scc}(G)$. Specifically, we compute *finishing times* (otherwise known as post-order labeling) for the vertices of G as follows.

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

```
DFS(v, clock):
  mark v
  for each edge v→w
      if w is unmarked
          clock ← DFS(w, clock)
  clock ← clock + 1
  finish(v) ← clock
  return clock
```

Lemma 4. *The vertex with largest finishing time lies in a source component of G.*

Proof: Let v be the vertex with largest finishing time. Then $\mathrm{DFS}(v, \mathit{clock})$ must be the last direct call to DFS made by the wrapper algorithm DFSAll.

Let C be the strong component of G that contains v. For the sake of argument, suppose there is an edge $x \to y$ such that $x \notin C$ and $y \in C$. Because v and y are strongly connected, y can reach v, and therefore x can reach v. There are two cases to consider.

- If x is already marked when DFS(v) begins, then v must have been marked during the execution of DFS(x), because x can reach v. But then v was already marked when DFS(v) was called, which is impossible.
- If x is not marked when DFS(v) begins, then x must be marked during the execution of DFS(v), which implies that v can reach x. Since x can also reach v, we must have $x \in C$, contradicting the definition of x.

We conclude that C is a source component of G. □

Essentially the same argument implies the following more general result.

Lemma 5. *For any edge $v \to w$ in G, if* finish(v) < finish(w)*, then v and w are strongly connected in G.*

Proof: Let $v \to w$ be an arbitrary edge of G. There are three cases to consider.If w is unmarked when DFS(v) begins, then the recursive call to DFS(w) finishes w, which implies that $finish(w) < finish(v)$. If w is still active when DFS(v) begins, there must be a path from w to v, which implies that v and w are strongly connected. Finally, if w is finished when DFS(v) begins, then clearly $finish(w) < finish(v)$. □

This observation is consistent with our earlier topological sorting algorithm; for *every* edge $v \to w$ in a directed acyclic graph, we have $finish(v) > finish(w)$.

It is easy to check (hint, hint) that any directed G has exactly the same strong components as its reversal $rev(G)$; in fact, we have $rev(scc(G)) = scc(rev(G))$. Thus, if we order the vertices of G by their finishing times in DFSALL($rev(G)$), the *last* vertex in this order lies in a sink component of G. Thus, if we run DFSALL(G), visiting vertices in reverse order of their finishing times in DFSALL($rev(G)$), then each call to DFS visits exactly one strong component of G.

Putting everything together, we obtain the following algorithm to count and label the strong components of a directed graph in $O(V + E)$ time, first discovered (but never published) by Rao Kosaraju in 1978, and then independently rediscovered by Micha Sharir in 1981. The Kosaraju-Sharir algorithm has two phases. The first phase performs a depth-first search of the reversal of G, pushing each vertex onto a stack when it is finished. In the second phase, we perform another depth-first search of the original graph G, considering vertices in the order they appear on the stack.

```
KosarajuSharir(G):
  ⟨⟨Phase 1: Push in finishing order⟩⟩
  unmark all vertices
  for all vertices v
      if v is unmarked
          clock ← RevPushDFS(v)
  ⟨⟨Phase 2: DFS in stack order⟩⟩
  unmark all vertices
  count ← 0
  while the stack is non-empty
      v ← Pop
      if v is unmarked
          count ← count + 1
          LabelOneDFS(v, count)
```

```
RevPushDFS(v):
  mark v
  for each edge v→u in rev(G)
      if u is unmarked
          RevPushDFS(u)
  Push(v)
```

```
LabelOneDFS(v, count):
  mark v
  label(v) ← count
  for each edge v→w in G
      if w is unmarked
          LabelOneDFS(w, count)
```

With further minor modifications, we can also compute the strongly connected component graph $scc(G)$ in $O(V+E)$ time.

Exercises

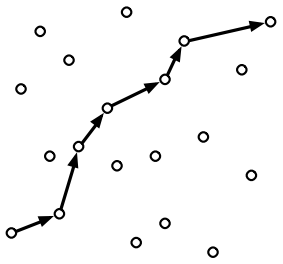
0. (a) Describe an algorithm to compute the reversal $rev(G)$ of a directed graph in $O(V+E)$ time.

   (b) Prove that for any directed graph G, the strong component graph $scc(G)$ is a dag.

   (c) Prove that for any directed graph G, we have $scc(rev(G)) = rev(scc(G))$.

   (d) Suppose S and T are two strongly connected components in a directed graph G. Prove that $finish(u) < finish(v)$ for all vertices $u \in S$ and $v \in T$.

1. 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.



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').

2. Let $G = (V, E)$ be a given directed graph.

   (a) The *transitive closure* G^T 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. Describe an efficient algorithm to compute the transitive closure of G.

   (b) A *transitive reduction* G^{TR} is a graph with the smallest possible number of edges whose transitive closure is G^T. (The same graph may have several transitive reductions.) Describe an efficient algorithm to compute the transitive reduction of G.

3. One of the oldest[1] algorithms for exploring graphs was proposed by Gaston Tarry in 1895. The input to Tarry's algorithm is a directed graph G that contains both directions of every edge; that is, for every edge $u \to v$ in G, its reversal $v \to u$ is also an edge in G.

```
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                       } ⟨⟨"traverse v→w"⟩⟩
      RECTARRY(w)
  else if there is a green arc v→w
      color v→w red                       } ⟨⟨"traverse v→w"⟩⟩
      RECTARRY(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).

   (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 when TARRY(G) ends, the last visited vertex 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.

4. 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:

[1] 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.

```
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
      pre(v) ← 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
      post(v) ← 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 $pre(v)$ and $post(v)$ 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.

5. For any two nodes u and v in a directed acyclic graph G, the ***interval*** $\boldsymbol{G[u,v]}$ is the union of all directed paths in G from u to v. Equivalently, $G[u,v]$ consists of all vertices x such that $x \in Reach(u)$ and $v \in Reach(x)$, 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. *[Hint: Don't try to be clever.]*

6. 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, such that the path contains at most ℓ edges. (Assume there is at least one such path.)

   (d) Suppose the vertices of G have integer labels, where $label(s) = -\infty$ and $label(t) = \infty$. Describe an algorithm to find the path from s to t with the maximum number of edges, such that the vertex labels define an increasing sequence.

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

7. Let G and H be directed acyclic graphs, 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.

(f) Describe an algorithm to find the longest string that is both a label of a directed path in G and the label of a directed path in H.

(g) Describe an algorithm to find the longest string that is both a *subsequence* of the label of a directed path in G and a *subsequence* of the label of a directed path in H.

(h) Describe an algorithm to find the shortest string that is both a *supersequence* of the label of a directed path in G and a *supersequence* of the label of a directed path in H.

(i) Describe an algorithm to find the longest path in G whose label is a palindrome.

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

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

8. Suppose two players are playing a turn-based game on a directed acyclic graph G with a unique source s. Each vertex v of G is labeled with a real number $\ell(v)$, which could be positive, negative, or zero. The game starts with three tokens at s. In each turn, the current player moves one of the tokens along a directed edge from its current node to another node, and the current player's score is increased by $\ell(u) \cdot \ell(v)$, where u and v are the locations of the two tokens that did *not* move. At most one token is allowed on any node except s at any time. The game ends when the current player is unable to move (for example, when all three tokens lie on sinks); at that point, the player with the higher score is the winner.

   Describe an efficient algorithm to determine who wins this game on a given labeled graph, assuming both players play optimally.

*9. 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 $(\texttt{AR})^*$ and $x = \texttt{ABRACADABRA}$, 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 $(\texttt{ABCDR})^*$ and $x = \texttt{ABRACADABRA}$, your algorithm should output the number 25, which is the length of the supersequence ABCDRABCDRABCDRABCDRABCDR.

10. Not *every* dynamic programming algorithm can be modeled as finding an optimal path through a directed acyclic graph; the most obvious counterexample is the optimal binary search tree problem. But every dynamic programming problem does traverse a dependency graph in reverse topological order, performing some additional computation at every vertex.

    (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) 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 numerical value $p(v \to w)$ between 0 and 1.
    - For each non-sink vertex v, we have $\sum_w p(v \to w) = 1$.

    The values $p(v \to w)$ 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 $p(v \to w)$. Describe and analyze an algorithm to compute the probability that this random walk reaches sink t_i, for every index i. (Assume that any arithmetic operation requires $O(1)$ time.)