*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)

*"We'll just go in here, so that you can say you've been, but it's very simple. It's absurd to call it a maze. You keep on taking the first turning to the right. We'll just walk round for ten minutes, and then go and get some lunch."*

— Harris describing the Hampton Court labyrinth
Jerome K. Jerome, *Three Men in a Boat* (1889)

*"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.*

— Umberto Eco, *Il nome della rosa* (1980)

Chapter 6

# Depth-First Search

This is mostly the Spring 2016 revision in the new skin.
In particular, Sections 6.1–6.3 still need significant revision.
Start with directed graphs!

★★★

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 and edges. 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 $v \rightarrow parent(v)$ 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.

## 6.1 Counting and Labeling Components

★★★

Already moved to Chapter 5. (Except for comments about PreVisit and PostVisit.)

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
            LABELONE(v, count)
    return count
```

```
⟨⟨Label one component⟩⟩
LABELONE(v, count):
    mark v
    comp(v) ← count
    for each edge vw
        if w is unmarked
            LABELONE(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.

## 6.2 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 $v$
    $clock \gets 0$
    for all vertices $v$
        if $v$ is unmarked
            $clock \gets$ LabelOne($v, clock$)
```

```
LabelOne(v, clock):
    mark $v$
    $pre(v) \gets clock$
    $clock \gets clock + 1$
    for each edge $vw$
        if $w$ is unmarked
            $clock \gets$ LabelOne($w, clock$)
    $post(v) \gets clock$
    $clock \gets clock + 1$
    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):
    $clock \gets 0$
```

```
PreVisit(v):
    $pre(v) \gets clock$
    $clock \gets clock + 1$
```

```
PostVisit(v):
    $post(v) \gets clock$
    $clock \gets clock + 1$
```

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.

## 6.3 Directed Graphs and Reachability

★★★

Already in Chapter 5

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 $v \to w$
        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$(u)$.

## 6.4 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.

Need an example here.

★★★

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 incoming 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. (This algorithm never uses parent pointers, so 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
```

**Lemma 3.** *IsAcyclic$(G)$ returns False if and only if $G$ contains a directed cycle.*

**Proof:** Suppose IsAcyclic($G$) returns False. Then the algorithm must discover an edge $v \to w$ such that $status(w) = $ Active. When IsAcyclicDFS($v$) is called, the Active vertices are precisely the vertices currently on the recursion stack, all of which are ancestors of 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 $C$. Let $w$ be the first vertex in $C$ that the algorithm marks Active, and let $v \to w$ be the edge in $C$ leading into $v$. Because there is a directed path from $w$ to $v$, we must recursively 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) = $ Active. The return value False bubbles up through all the recursive calls to the top level. □

If modifying the graph is impossible or undesirable, we can use a wrapper function like our earlier WFSAll, which calls the subroutine repeatedly for every New vertex. It should be clear that this is merely a change of notation, not a change in the algorithm; all we've done is replace a loop over the edges $s \to v$ with a loop directly over the vertices $v$.

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

## 6.5 Topological Sort

★★★ Need an example here.

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[i] ← 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 *marking* the 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 the rest of this section, I'll present two topological sorting algorithms that run in $O(V+E)$ time without destroying the graph.

## Whatever-Sink-First Search

Our first fast topological sort algorithm is just a fast implementation of the previous method, first published by Arthur Kahn in 1962.

Like whatever-first search, Khan's algorithm maintains a *bag* of source vertices. (Kahn used a queue, but we don't have to.) Instead of a "mark", each new vertex maintains its *in-degree*, which is the number of incoming edges. At each iteration, we grab a new source $v$ from the bag, "delete" $v$ from the graph, and then add any new "sources" to the bag. But we do not *actually* delete $v$; instead, we merely decrement the in-degrees of every successor of $v$, exactly *as if* $v$ was deleted. Then we put any successor of $v$ with in-degree zero into the bag as a new source and start the next iteration.

```
KhanInitialize(V, E):
  for every vertex $v$
      $v.indeg \gets 0$
  for every edge $u \to v$
      $v.indeg \gets v.indeg + 1$
  for every vertex $v$
      if $v.indeg = 0$
          put $v$ into the bag
```

```
KhanTopoSort(V, E):
  KhanInitialize(V, E)
  $clock \gets 0$
  while the bag is not empty
      take $v$ from the bag
      $clock \gets clock + 1$
      $S[clock] \gets v$
      for all edges $v \to w$
          $w.indeg \gets w.indeg - 1$      $(*)$
          if $w.indeg = 0$
              put $w$ into the bag
  if $clock < V$
      fail gracefully   ⟨⟨There's a cycle!⟩⟩
  else
      return $S[1..V]$
```

(This version of the algorithm adds each vertex $v$ to the output array $S[\,]$ when $v$ is taken out of the bag; we could alternatively add $v$ to the output array when we put $v$ into the bag. You say potato, I say potato.[1])

[1]You say Car-mee-na, I say Car-my-na. You say Buh-ray-na, I say Buh-rah-na. Car-mee-na, Car-my-na, Buh-ray-na, Buh-rah-na. Let's Carl the whole thing Orff!

Suppose each bag operation takes $O(1)$ time; for example, suppose we are using either a stack or a queue. The initialization phase clearly runs in at most $O(V+E)$ time. In the main algorithm, note that line $(*)$ is executed exactly once for each edge, and each vertex is put into the bag at most once and taken out of the bag at most once, so the main algorithm also runs in $O(V+E)$ time. Thus, the overall algorithm runs in $\boldsymbol{O(V+E)}$ ***time***.

Even though Khan's algorithm is universally presented using a queue for the "bag", the algorithm is arguably simpler if we use an implicit stack via recursion.

```
KhanTopoSort(V, E):
  for every vertex $v$
      $v.indeg \gets 0$
  for every edge $u \to v$
      $v.indeg \gets v.indeg + 1$

  $clock \gets 0$
  for every vertex $v$
      if $v.indeg = 0$
          $clock \gets$ KhanDFS($v, clock$)

  if $clock < V$
      fail gracefully    ⟨⟨There's a cycle!⟩⟩
  else
      return $S[1..V]$
```

```
KhanDFS(v, clock):
  $clock \gets clock + 1$
  $S[clock] \gets v$
  for all edges $v \to w$
      $w.indeg \gets w.indeg - 1$    $(*)$
      if $w.indeg = 0$
          $clock \gets$ KhanDFS($w, clock$)
  return $clock$
```

## Depth-First Search

But even the simple bookkeeping in Khan's algorithm is unnecessary; our earlier depth-first search algorithm for deciding if a graph is acyclic actually outputs a topological order as a side-effect. This application of depth-first search was first observed by Robert Tarjan in 1971.

**Lemma 4.** *For any directed graph $G$, the first vertex (if any) marked* Done *by* IsAcyclic$(G)$ *is a sink.*

**Proof:** Let $v$ be the first vertex marked Done during an execution of IsAcyclic$(G)$. 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 $w$ was 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, $w$ was 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. Alternatively: If $status(w) =$ Active, then the algorithm aborts execution before marking $v$ Done

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

It follows by induction that to topologically sort a dag $G$, it suffices to list the vertices in the *reverse* order of being marked DONE.

Need more details here ★★★

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 $v \neq s$
      add edge $s \to v$
      $status(v) \gets$ NEW
  TOPOSORTDFS($s$)
  for $i \gets 1$ to $V$
      $S[i] \gets$ POP
  return $S[1..V]$
```

```
TOPOSORTDFS($v$):
  $status(v) \gets$ ACTIVE
  for each edge $v \to w$
      if $status(w) =$ NEW
          PROCESSBACKWARDDFS($w$)
      else if $status(w) =$ ACTIVE
          fail gracefully
  $status(v) \gets$ DONE
  PUSH($v$)
  return TRUE
```

We don't even need the stack if we are willing let the output array be a global variable.

```
TOPOLOGICALSORT($G$):
  add vertex $s$
  for all vertices $v \neq s$
      add edge $s \to v$
      $status(v) \gets$ NEW
  TOPOSORTDFS($s, V$)
  return $S[1..V]$
```

```
TOPOSORTDFS($v, clock$):
  $status(v) \gets$ ACTIVE
  for each edge $v \to w$
      if $status(w) =$ NEW
          $clock \gets$ TOPOSORTDFS($w, clock$)
      else if $status(w) =$ ACTIVE
          fail gracefully
  $status(v) \gets$ DONE
  $S[clock] \gets v$
  $clock \gets clock - 1$
  return $clock$
```

## Implicit Topological Sort

But maintaining a separate data structure is actually overkill. In most applications of topological sort, our actual goal is not a topologically sorted list of the vertices; instead, 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.

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
    PROCESSPOSTORDERDFS(s)
```

```
PROCESSPOSTORDERDFS(v):
    status(v) ← ACTIVE
    for each edge v→w
        if status(w) = NEW
            PROCESSPOSTORDERDFS(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.

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

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

Similar modification to Khan's algorithm allow us to process any dag in *forward* topological order. Alternatively, we could apply depth-first search to 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.

## 6.6 Memoization

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.

```
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)
```

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 the following algorithm run on the dependency graph of the underlying recurrence:

```
DynamicProgramming(G):
  for all subproblems x in reverse topological order
      initialize value[x]
      for all subproblems y of x
          update value[x] based on value[y]
      finalize value[x]
```

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 5, 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 lets us hard-wire a topological order directly into the algorithm—which we previously called an *evaluation order*—so we don't need to compute it at run time.

## 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

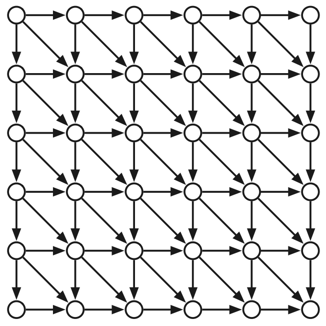


**Figure 6.1.** The dependency **dag** of the edit distance recurrence.

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.

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

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

where $\ell(v\to w)$ is the given weight ("length") of edge $v\to w$. 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
      $v.LLP \leftarrow -\infty$
      for each edge $v\to w$
          $v.LLP \leftarrow \max\{v.LLP,\ \ell(v\to w) + \text{LONGESTPATH}(w, t)\}$
      return $v.LLP$
```

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

```
LongestPath($s, t$):
  for each node $v$ in reverse topological order
      if $v = t$
          $v.LLP \leftarrow 0$
      else
          $v.LLP \leftarrow -\infty$
          for each edge $v \to w$
              $v.LLP \leftarrow \max\{v.LLP,\ \ell(v \to w) + w.LLP\}$
  return $s.LLP$
```

But these two algorithms are arguably identical—the pattern of recursion in the first algorithm and the for-loop in the second algorithm are both just depth-first search!

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 subset sum, longest increasing subsequence, and edit distance problems we considered in Chapters 2 and 3 can all be reformulated as finding either a longest path or a shortest path in a dag, possibly with weighted vertices or edges.

## 6.7 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.

For any directed graph $G$, the ***strong component graph*** $\boldsymbol{scc(G)}$ 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 $scc(G)$ 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.

Need an example here. ★★★

It is straightforward to compute the strong component containing a single vertex $v$ in $O(V + E)$ time. First we compute $reach(v)$ by calling WhateverFirstSearch($v$). 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.

We can compute *all* the strong components in a directed graph by wrapping the single-strong-component algorithm in a 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. Surely we can do better! After all, we only need $O(V + E)$ time to decide whether every strong component is a single vertex.

## 6.8 Strong Components in Linear Time

Let $C$ be any strong component of $G$ that is a sink in $scc(G)$; 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($G$):
  $count \leftarrow 0$
  while $G$ is non-empty
      $count \leftarrow count + 1$
      $v \leftarrow$ any vertex in a sink component of $G$
      $C \leftarrow$ 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 $scc(G)$. 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 $v$
  $clock \leftarrow 0$
  for all vertices $v$
      if $v$ is unmarked
          $clock \leftarrow$ DFS($v, clock$)
```

```
DFS($v, clock$):
  mark $v$
  for each edge $v \rightarrow w$
      if $w$ is unmarked
          $clock \leftarrow$ DFS($w, clock$)
  $clock \leftarrow clock + 1$
  $finish(v) \leftarrow clock$
  return $clock$
```

**Lemma 5.** *The vertex with largest finishing time lies in a source component of $G$.*

**Proof:** Let $v$ be the vertex with largest finishing time. Then DFS($v, clock$) 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 $x \rightarrow y$ 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 6.** *For any edge $v \rightarrow w$ in $G$, if* finish($v$) < finish($w$)*, then $v$ and $w$ are strongly connected in $G$.*

**Proof:** Let $v \rightarrow w$ 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 $\mathit{finish}(w) < \mathit{finish}(v)$. 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 $\mathit{finish}(w) < \mathit{finish}(v)$. □

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

It is easy to check (hint, hint) that any directed $G$ has exactly the same strong components as its reversal $\mathit{rev}(G)$; in fact, we have $\mathit{rev}(\mathit{scc}(G)) = \mathit{scc}(\mathit{rev}(G))$. Thus, if we order the vertices of $G$ by their finishing times in DFSAll($\mathit{rev}(G)$), the *last* vertex in this order lies in a sink component of $G$. Thus, if we run a second *whatever*-first graph traversal WFSAll($G$), where the wrapper function considers vertices in reverse order of their finishing times in DFSAll($\mathit{rev}(G)$), then each call to WFS visits exactly one strong component of $G$. (Both Kosaraju and Sharir used depth-first search in the second phase, but we don't have to.)

Putting everything together, we obtain the following algorithm to count and label the strong components of a directed graph in $O(V + E)$ time, discovered (but never published) by Rao Kosaraju in 1978, and then independently rediscovered by Micha Sharir in 1981. (There are rumors that the same algorithm appears int he Russian literature even before Kosaraju, but I haven't tracked down that source yet.) 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 *whatever*-first traversal of the original graph $G$, considering vertices in the order they appear on the stack.

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

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

```
LabelOneWFS(v, count):
    put $v$ into the bag
    while the bag is not empty
        take $v$ from the bag
        mark $v$
        $label(v) \leftarrow count$
        for each edge $v \to w$ in $G$
            if $w$ is unmarked
                put $w$ into the bag
```

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

## Exercises

★★★ Need more exercises that are *not* solved by dynamic programming!

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 acyclic.

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

   (d) Fix an arbitrary directed graph $G$. For any vertex $v$ of $G$, let $S(v)$ denote the strong component of $G$ that contains $v$. Prove, for all vertices $u$ and $V$ of $G$, 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. 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.

2. 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 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.

3. One of the oldest algorithms for exploring arbitrary connected graphs was proposed by Gaston Tarry in 1895, as a procedure for solving mazes.[2] 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; 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.

[2] 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.

(c) When the algorithm visits a vertex $v$ for the $k$th 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.

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:

```
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, even though it visits the edges in a completely different order.

5. You have a collection of $n$ lockboxes 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.

6. 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 $s$.

7. 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

8. Suppose we are given a directed acyclic graph $G$ whose nodes represent jobs and whose edges represent precedence constraints; that is. each edge $u\rightarrow v$ 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)).

9. 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 $\ell$. Describe an efficient algorithm to find the maximum-weight path from $s$ to $t$ that contains at most $\ell$ 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.)

10. 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.

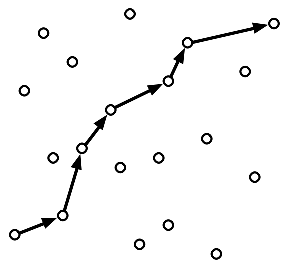
(b) Describe an algorithm to find the *number* of paths in $G$ whose label is $A$. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

(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 $A$.

11. 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 $\text{Length}(x, y, x', y')$ that returns the length of the segment from $(x, y)$ to $(x', y')$.

12. 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 $x \in \mathit{reach}(u)$ 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.

13. 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 6.2, your algorithm should return the integer 6, which is the length of the palindrome HANNAH.

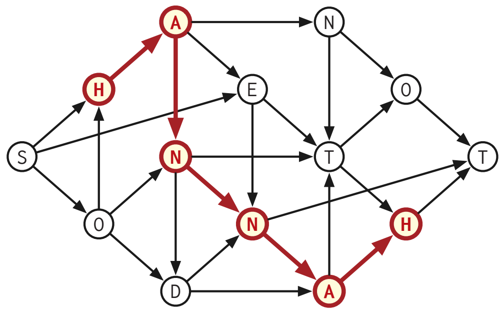


**Figure 6.2.** A dag whose longest palindrome path label has length 6.

    (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 $G$.

14. 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 $H$.

15. 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 descendant of every vertex in $G$. That is, for each vertex $v$, your algorithm needs to compute $\max\{w(x) \mid x \in \mathit{reach}(v)\}$.

16. 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 8. First think about graphs where the first two conditions never happen.]*

17. 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.

18. The Doctor and River Song decide to play a game on a directed acyclic graph $G$, which has one source $s$ and one sink $t$.[3]

    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 $G$.

19. Let $x = x_1 x_2 \dots x_n$ be a given $n$-character string over some finite alphabet $\Sigma$, 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 $x =$ **A**B**R**AC**A**DAB**R**A, 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 $x =$ ABRACADABRA, your algorithm should output the number 25, which is the length of the supersequence **AB**CD**RA**B**C**DR**A**BC**D**R**AB**CD**RA**BCDR.

    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 $\Sigma$.

20. 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

[3]The labels $s$ and $t$ may be abbreviations for the Untempered **S**chism and the **T**ime Vortex, or the Shining World of the Seven Systems (otherwise known as Gallifrey) and Trenzalore, or Skaro and Telos, or Something else Timey-wimey. We'll never know for sure.

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 $v \rightarrow w$ has an associated weight $p(v \rightarrow w)$ between 0 and 1.
- For each non-sink vertex $v$, the total weight of all edges leaving $v$ is 1; that is, $\sum_w p(v \rightarrow w) = 1$.

The weights $p(v \rightarrow w)$ 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 $v \rightarrow w$ with probability $p(v \rightarrow w)$. 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.)