18 Depth-First Search

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

\[
\text{DFS}(v):
\begin{align*}
&\text{if } v \text{ is unmarked} \\
&\quad \text{mark } v \\
&\quad \text{for each edge } vw \\
&\quad \text{DFS}(w)
\end{align*}
\]

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 \texttt{PreVisit} and \texttt{PostVisit}, which we leave unspecified for now.

\[
\text{DFS}(v):
\begin{align*}
&\text{mark } v \\
&\text{\texttt{PreVisit}(v)} \\
&\text{for each edge } vw \\
&\quad \text{if } w \text{ is unmarked} \\
&\quad \quad \text{parent}(w) \leftarrow v \\
&\quad \text{DFS}(w) \\
&\text{\texttt{PostVisit}(v)}
\end{align*}
\]

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 \(v \rightarrow \text{parent}(v)\) 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 $\text{DFS}(v)$ is invoked, but marked when $\text{DFS}(v)$ 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.

18.1 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 $\text{PREPROCESS}$ to perform any necessary preprocessing for the $\text{POSTVISIT}$ and $\text{POSTVISIT}$ functions.

$$\begin{align*}
\text{DFSALL}\left(G\right): & \\
\text{PREPROCESS}\left(G\right) & \\
\text{for all vertices } v & \text{unmark } v \\
\text{for all vertices } v & \text{if } v \text{ is unmarked }
\end{align*}$$

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 $\text{comp}(v) \leftarrow \text{count}$ is a trivial example of $\text{PREVISIT}$. (And the absence of code after the for loop is a vacuous example of $\text{POSTVISIT}$.)

$$\begin{align*}
\text{COUNTANDLABEL}\left(G\right): & \\
\text{count} & \leftarrow 0 \\
\text{for all vertices } v & \text{unmark } v \\
\text{for all vertices } v & \text{if } v \text{ is unmarked }
\end{align*}$$

$$\begin{align*}
\text{LABELCOMPONENT}\left(v, \text{count}\right): & \\
\text{mark } v & \\
\text{comp}(v) & \leftarrow \text{count} \\
\text{for each edge } vw & \text{if } w \text{ is unmarked }
\end{align*}$$

return $\text{count}$

18.2 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:
Algorithms

18.3 Directed Graphs and Reachability

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

\[
\text{DFS}(v): \quad \text{mark } v \\
\text{PREVISIT}(v) \\
\text{for each edge } v \rightarrow w \\
\quad \text{if } w \text{ is unmarked} \\
\quad \text{DFS}(w) \\
\text{POSTVISIT}(v)
\]

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 \( \text{DFSALL} \), we may discover any number of “components” between 1 and \( n \). All that we can guarantee is that the “component” numbers computed by \( \text{DFSALL} \) do not increase as we traverse the path. In fact, the real problem is that the \textit{definition} of “component” is only suitable for \textit{undirected} graphs.

Instead, for directed graphs we rely on a more subtle notion of reachability. We say that a node \( v \) is \textit{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 \( \text{Reach}(u) \) denote the set of vertices that are reachable from \( u \) (including \( u \) itself). A simple inductive argument proves that \( \text{Reach}(u) \) is precisely the subset of nodes that are marked by calling \( \text{DFS}(u) \).

18.4 Directed Acyclic Graphs

A \textit{directed acyclic graph} or \textit{dag} is a directed graph with no directed cycles. Any vertex in a dag that has no incoming vertices is called a \textit{source}; any vertex with no outgoing edges is called a \textit{sink}. Every
A topological ordering of a directed graph $G$ is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \rightarrow v$. 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.

**18.5 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 $u \rightarrow v$. 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.
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 $\text{DONE}$ by $\text{ISACYCLIC}(G)$ must be a sink.

**Proof:** Let $v$ be the first vertex marked $\text{DONE}$ during an execution of $\text{ISACYCLIC}$. For the sake of argument, suppose $v$ has an outgoing edge $v \rightarrow w$. When $\text{ISACYCLICDFS}$ first considers the edge $v \rightarrow w$, there are three cases to consider.

- If $\text{status}(w) = \text{DONE}$, then $w$ is marked $\text{DONE}$ before $v$, which contradicts the definition of $v$.

- If $\text{status}(w) = \text{NEW}$, the algorithm calls $\text{TOPOSORTDFS}(w)$, which (among other computation) marks $w$ $\text{DONE}$. Thus, $w$ is marked $\text{DONE}$ before $v$, which contradicts the definition of $v$.

- If $\text{status}(w) = \text{ACTIVE}$, 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 $\text{DONE}$. For example, we could push each vertex onto a stack when we mark it $\text{DONE}$, and then pop every vertex off the stack.

```
TOPOLOGICALSORT($G$):
add vertex $s$
for all vertices $v \neq s$
    add edge $s \rightarrow v$
    status($v$) ← NEW
TOPOSORTDFS($s$)
for $i ← 1$ to $V$
    $S[i] ← \text{POP}$
return $S[1..V]$
```

```
TOPOSORTDFS($v$):
    status($v$) ← ACTIVE
    for each edge $v \rightarrow w$
        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.
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.

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 $v \rightarrow w$ with its reversal $w \rightarrow v$. 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.

### 18.6 Every Dynamic Programming Algorithm?

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, 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 \( s \).

```python

def LongestPath(s, t):
    if s == t:
        return 0
    if LLP(s) is undefined:
        LLP(s) = \infty
    for each edge s \to v:
        LLP(s) = \max\{LLP(v), \ell(s \to v) + LLP(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.

### 18.7 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 \( \text{Reach}(u) \) 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 \( \text{Reach}(v) \) by calling DFS(\( v \)). Then we compute \( \text{Reach}^{-1}(v) = \{ u \mid v \in \text{Reach}(u) \} \) by searching the reversal of \( G \). Finally, the strong component of \( v \) is the intersection \( \text{Reach}(v) \cap \text{Reach}^{-1}(v) \). 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 \( O(V E) \) 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.
18.8 Strong Components in Linear Time

For any directed graph \( G \), the strong component graph \( \text{scc}(G) \) 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 \textit{meta-graph} or \textit{condensation} of \( G \). It's not hard to prove (hint, hint) that \( \text{scc}(G) \) 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 \textit{backward} edge joins two edges in the same strong component.

Let \( C \) be any strong component of \( G \) that is a sink in \( \text{scc}(G) \); we call \( C \) a \textit{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 \textit{only} vertices in \( C \). So if we can compute all the strong components as follows:

\[
\text{StrongComponents}(G): \\
count \leftarrow 0 \\
\text{while } G \text{ is non-empty} \\
\quad \text{count} \leftarrow \text{count} + 1 \\
\quad v \leftarrow \text{any vertex in a sink component of } G \\
\quad C \leftarrow \text{OneComponent}(v, \text{count}) \\
\quad \text{remove } C \text{ and incoming edges from } G
\]

At first glance, finding a vertex in a sink component \textit{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 \( \text{scc}(G) \). Specifically, we compute \textit{finishing times} (otherwise known as post-order labeling) for the vertices of \( G \) as follows.

\[
\text{DFSAll}(G): \\
\quad \text{for all vertices } v \\
\quad \quad \text{unmark } v \\
\quad \quad clock \leftarrow 0 \\
\quad \text{for all vertices } v \\
\quad \quad \text{if } v \text{ is unmarked} \\
\quad \quad \quad clock \leftarrow \text{DFS}(v, clock)
\]

\[
\text{DFS}(v, clock): \\
\quad \text{mark } v \\
\quad \text{for each edge } v \rightarrow w \\
\quad \quad \text{if } w \text{ is unmarked} \\
\quad \quad \quad clock \leftarrow \text{DFS}(w, clock) \\
\quad \quad clock \leftarrow clock + 1 \\
\quad \text{finish}(v) \leftarrow clock \\
\quad \text{return clock}
\]

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 \( \text{DFS}(v, clock) \) must be the last direct call to DFS made by the wrapper algorithm \( \text{DFSAll} \).

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 \( x \not\in C \) 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 \( \text{DFS}(v) \) begins, then \( v \) must have been marked during the execution of \( \text{DFS}(x) \), because \( x \) can reach \( v \). But then \( v \) was already marked when \( \text{DFS}(v) \) was called, which is impossible.

- If \( x \) is not marked when \( \text{DFS}(v) \) begins, then \( x \) must be marked during the execution of \( \text{DFS}(v) \), 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 \( G \). \( \square \)
Essentially the same argument implies the following more general result.

**Lemma 5.** For any edge \( v \rightarrow w \) in \( G \), if \( \text{finish}(v) < \text{finish}(w) \), 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 \( \text{DFS}(v) \) begins, then the recursive call to \( \text{DFS}(w) \) finishes \( w \), which implies that \( \text{finish}(w) < \text{finish}(v) \). If \( w \) is still active when \( \text{DFS}(v) \) 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 \( \text{DFS}(v) \) begins, then clearly \( \text{finish}(w) < \text{finish}(v) \). □

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

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

```plaintext
KOSARAJU-SHARIR(G):

// Phase 1: Push in finishing order
unmark all vertices
for all vertices v
  if v is unmarked
    clock ← REV-PUSH-DFS(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
    LABEL-ONE-DFS(v, count)
```

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

**Exercises**

0. (a) Describe an algorithm to compute the reversal \( \text{rev}(G) \) of a directed graph in \( O(V + E) \) time.
   (b) Prove that for any directed graph \( G \), the strong component graph \( \text{scc}(G) \) is a dag.
   (c) Prove that for any directed graph \( G \), we have \( \text{scc}(	ext{rev}(G)) = \text{rev(scc}(G)) \).
   (d) Suppose \( S \) and \( T \) are two strongly connected components in a directed graph \( G \). Prove that \( \text{finish}(u) < \text{finish}(v) \) for all vertices \( u \in S \) and \( v \in T \).
1. One of the oldest 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 $u \rightarrow v$ in $G$, its reversal $v \rightarrow u$ is also an edge in $G$.

Tarry($G$):

- unmark all vertices of $G$
- color all edges of $G$ white
- $s \leftarrow$ any vertex in $G$

RecTarry($s$)

RecTarry($v$):

- mark $v$ “(visit $v$)”
- if there is a white arc $v \rightarrow w$
  - if $w$ is unmarked
    - color $w \rightarrow v$ green
    - color $v \rightarrow w$ red
    - RecTarry($w$) “(traverse $v \rightarrow w$)”
- else if there is a green arc $v \rightarrow w$
  - color $v \rightarrow w$ red
  - RecTarry($w$) “(traverse $v \rightarrow w$)”

We informally say that Tarry’s algorithm “visits” vertex $v$ every time it marks $v$, and it “traverses” edge $v \rightarrow w$ 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 $k$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 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.

2. 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:

RecTarry2($v, clock$):

- if $v$ is unmarked
  - $pre(v) \leftarrow clock$; $clock \leftarrow clock + 1$
  - mark $v$
- if there is a white arc $v \rightarrow w$
  - if $w$ is unmarked
    - color $w \rightarrow v$ green
    - color $v \rightarrow w$ red
    - RecTarry2($w, clock$)
- else if there is a green arc $v \rightarrow w$
  - $post(v) \leftarrow clock$; $clock \leftarrow clock + 1$
  - RecTarry2($w, clock$)

Even older graph-traversal algorithms were described by Charles Trémaux in 1882, Christian Wiener in 1873, and (implicitly) Leonhard Euler in 1736. Wiener’s algorithm is equivalent to depth-first search in a connected undirected graph.
Prove that when \textsc{Tarry2}(G) halts, the green edges define a spanning tree and the labels \textit{pre}(v) and \textit{post}(v) define a preorder and postorder labeling that are all consistent with a single depth-first search of G. In other words, prove that \textsc{Tarry2} produces the same output as depth-first search.

3. Let \( G = (V, E) \) be a given directed graph.
   
   (a) The \textit{transitive closure} \( G^T \) is a directed graph with the same vertices as \( G \), that contains any edge \( u \rightarrow v \) 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 \textit{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 \( G \).

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

5. Let \( G \) be a directed acyclic graph with a unique source \( s \) and a unique sink \( t \).
   
   (a) A \textit{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 \textit{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 \( t \), such that the path contains at most \( \ell \) edges. (Assume there is at least one such path.)
   
   (d) Suppose the vertices of \( G \) have integer labels, where \textit{label}(s) = -\infty \) and \textit{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.)

6. 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 \textit{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 $G$.

7. 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.

8. Let $x = x_1x_2\ldots x_n$ 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 longest subsequence of $x$ that is accepted by $A$. For example, if $A$ accepts the language $(ar)^*$ and $x = \text{abracadabra}$, your algorithm should output $\text{arar}$.

(b) Describe and analyze an algorithm to compute the shortest (not necessarily contiguous) supersequence of $x$ that is accepted by $A$. For example, if $A$ accepts the language $(abcdr)^*$ and $x = \text{abracadabra}$, your algorithm should output $\text{abcdrabcdrabcdrabcdrabcd}$.

9. 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 $v \rightarrow w$ has an associated numerical value $p(v \rightarrow w)$ between 0 and 1.
• For each non-sink vertex $v$, we have $\sum_w p(v \rightarrow w) = 1$.

The values $p(v \rightarrow w)$ 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.)