Solutions: ICS 311, Fall 2020, Problem Set 8, Topic 14

This is a 40 point homework. The extra 10 points are extra credit opportunities, or you can leave out a problem that you don't have time to solve.

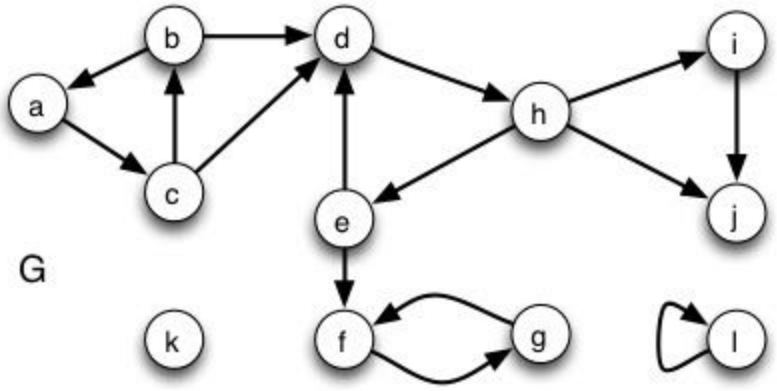
#1 (10 pts). SCC

In the following, use the SCC and DFS algorithms of the CLRS textbook:

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

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
       u.π = NIL
   time = 0
   for each vertex u ∈ G.V
       if u.color == WHITE
           DFS-VISIT(G, u)
S-VISIT(G, u)
   time = time + 1
   u.d = time
   u.color = GRAY
   for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

1. <u>Run DFS on this graph</u>. To make grading and comparison of solutions easier, **<u>visit vertices in alphabetical order</u>** (both in the main loop of DFS and the adjacency list loop of DFS-Visit).



a. For each vertex, show values d (discovery), f (finish), and π (parent).

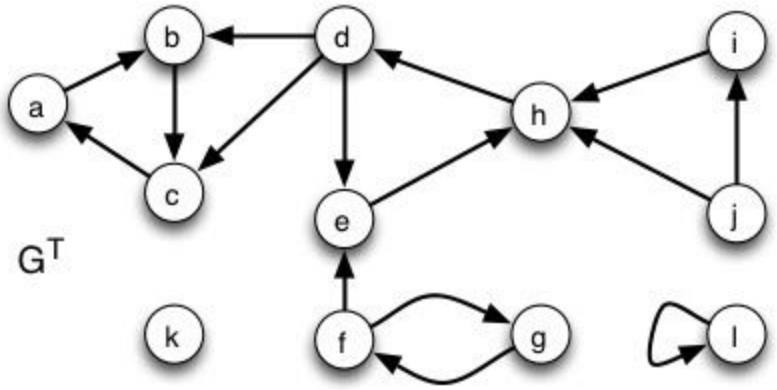
	a	b	c	d	e	f	g	h	i	j	k	l
d	1	3	2	4	6	7	8	5	12	13	21	23

f	20	18	19	17	11	10	9	16	15	14	22	24
π	NIL	c	a	b	h	e	f	d	h	i	NIL	NIL

b. Write the vertices in order <u>from largest to smallest</u> finish time:

l, k, a, c, b, d, h, i, j, e, f, g

2. Now <u>run DFS on the transpose graph, visiting vertices in order of largest to smallest finish time</u> from the DFS of step 1 (as required by the SCC algorithm). Again, **show values d (discovery), f (finish), and π (parent)**.



	a	b	c	d	e	f	g	h	i	j	k	l
d	5	6	7	11	12	21	22	13	17	19	3	1
f	10	9	8	16	15	24	23	14	18	20	4	2
π	NIL	a	b	NIL	d	NIL	f	e	NIL	NIL	NIL	NIL

3. List the strongly connected components you found by first <u>listing the tree edges</u> in the transpose graph that define the SCC, and then <u>listing the vertices</u> in the SCC (the first SCC containing a single vertex is shown):

SCC 1: Tree edges: { }; Vertices: {l}
SCC 2: Tree edges: { }; Vertices: {k}
SCC 3: Tree edges: {(a,b) (b,c) }; Vertices: {a, b, c}
SCC 4: Tree edges: {(d, e) (e, h) }; Vertices: {d, e, h}
SCC 5: Tree edges: { }; Vertices: {i}
SCC 6: Tree edges: { }; Vertices: {j}

SCC 7: Tree edges: {(f,g)}; Vertices: {f, g}

#2 (10 pts) Bottom-Up Longest-Paths

In class for Topic 12 Dynamic Programming, you (1) characterized the structure of an optimal solution for Longest Paths; (2) recursively defined the value of an optimal solution; (3) recursively computed the value of an optimal solution; and (4) memoize this recursive solution. Then in problem Set 7, you wrote additional code to actually recover the path.

Perhaps you thought you were done, but here, because we know how much *you **love** dynamic programming problems*, you will solve the same problem in Θ(V+E) using a **bottom-up** dynamic programming approach. At least, all of the preliminaries have been done! Be sure to include `dist` and `next`.

Hint: We need to arrange to solve smaller problems before larger ones: use topological sort (which you may assume has already been written). You may want to write the explanation for (b) before writing the pseudocode to guide your coding, but then revise it to reference the lines of code.

(a) Show your pseudocode for `Longest-Path-Bottom-Up`.

Solution:

```
// s is start, t is target.
// new arrays will be assigned to dist and next
// or one can assume that empty arrays are passed.
Longest-Path-Bottom-Up (G, s, t, dist, next)
1  let dis[1..n] and next[1..n] be new arrays
2  topologically sort the vertices of G
3  for i = 1 to |G.V|
4      dist[i] = -∞ // or set to 0 is OK
5  dist[s] = 0
6  for each u ∈ G.V in topological order starting from s
7      for each edge (u,v)∈ G.Adj[u]
8          if dist[u] + w(u,v) > dist[v]
9              dist[v] = dist[u] + w(u,v)
10             next[u] = v
11 return (dist,next) // or similar
```

One can optionally add lines to print dist[t] and the path from s to to in next, but these can be looked up later.

(b) Explain why it works; in particular, why topological sort is useful.

Solution: The topological sort ensures that we solve smaller problems before larger ones by ensuring that by the time we get to any vertex we will have already processed all vertices that are in-incident on it (that is, all of the vertices by which we can reach the present vertex already have their longest path solution computed).

Then the inner loop works as before: if we find a longer way to get to v, then we update dist[v] and assign u to go to v.

(c) Analyze its asymptotic run time.

Solution: Topological sort in line 2 is O(V + E) from prior analysis. Lines 3-4 are O(V) so this is less. The outer loop at line 6 executes |V| times, but we use aggregate analysis for the inner loop at line 7: Across all invocations of this inner loop (each invocation being in a pass of the outer loop), O(E) edges are processed. Processing of each edge is constant time. Thus we get O(V + E) for the nested loops as well, and the overall result is **O(V + E)**.

#3 (10 pts) "Really Bad Networks"

Suppose we have a network in which information or material flows between entities that we will call "nodes", and this flow is directed (need not go both ways) over "links". We can model such a network using a directed graph $G = (V, E)$: Nodes within a strongly connected component can send and receive information to and from any node within the same strongly connected component. Real world networks are usually designed in such a way that there are more than one simple path between every pair of vertices: Then if any link becomes unavailable, information can still be distributed using the remaining links.

In this problem we are interested in detecting **really bad networks**. Given a directed graph $G = (V, E)$ design an algorithm that determines whether there is at most **one** directed path between **every** pair of nodes in G. It will return TRUE if the network is

really bad, and FALSE if not. For full credit, your algorithm should run in O(V(V+E)) time.

(a) Show your pseudocode.

Solution: The following algorithm identifies really bad networks:

```
1 for each node u in G.V
2     Run a modified DFS starting from u, where
          if at any time a black node was accessed
              return FALSE // Not a really bad network
3 return TRUE // A really bad network
```

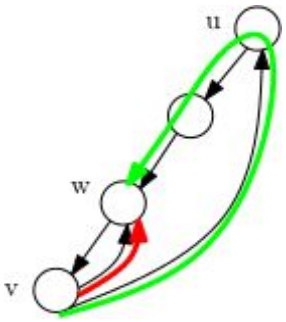
(b) Explain why it works.

Solution: Consider running DFS from some node *u*. Observe that there are two paths from *u* to another node *v* if and only if one of the following conditions is true:

- There is a forward edge (*u,v*): one path is along the tree edges from *u* to *v*, and another path is through the edge (*u,v*);
- There is a cross edge (*w,v*) from some node *w*, which is a descendant of *u*: one path is from *u* to *v* along the tree edges, and another path is $u \rightsquigarrow w$ and (*w,v*), where $u \rightsquigarrow w$ is a path from *u* to *w* along the tree edges.

We can detect forward edges and cross edges whenever we access a black node during the DFS traversal (see CLRS section 22.3, if you don't see why).

Now, if we run DFS from every node of the graph and detect one of the two conditions above, then we know that the graph is definitely NOT a really bad network, because there are at least two paths between some pair of nodes u and v.



However, if we run DFS from every node of the graph and none of the above conditions occurs, how do we know that this is not a really bad network? I.e. that we didn't miss some other paths? For example (see image on the right), if there are two back edges (v,w) and (v,u), where w is descendant of u, and v is a descendant of both u and w, then there are two paths from v to w: 1) along the edge (v,w) (red path) and 2) along the edge (v,u) and then along the path from u to w (green path). However, note that we will detect these paths as two paths from *v* to *w* when we run DFS from node *v*. So we don't have to worry about back edges.

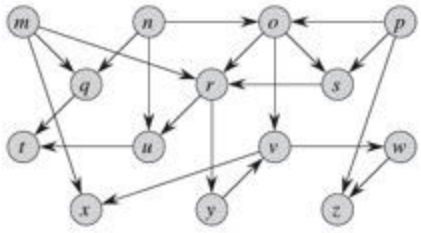
(c) Analyze its asymptotic run time.

Solution: Clearly, the runtime of this algorithm is O(V(V+E)), as it consists of O(V) iterations of DFS, each of which runs in O(V+E) time.

By the way, "really bad networks" are more commonly known as "**singly connected graphs**".

#4 (10 pts) Counting Simple Paths in a DAG

Design an algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t, and returns the number of simple paths from s to t in G. For example, the directed acyclic graph on the right contains exactly four simple paths from vertex p to vertex v: *pov, poryv, posryv*, and *psryv*. Your algorithm should run in time $O(V+E)$. (Your algorithm needs only to count the simple paths, not list them.)



Hints: Solve the more general case of number of paths to all vertices. Use topological sort to solve smaller problems before larger ones. Be sure to consider the boundary cases where $s = t$ and where there are no simple paths between s and t.

Preview of Solution:
Note: when s=t we have one simple path of length 0.
Here is a sketch of the algorithm (and preliminary analysis):

1. Give each vertex v an attribute p that counts the number of known simple paths from v to t. Initialize this to 0 for all vertices, except that t.p = 1. O(V).
2. Perform a topological sort of G. O(V+E)
3. For each v processed in the reverse order of the sort above, set u.p to be the sum of v.p for all outgoing neighbors v (i.e., for every edge (u,v) in the graph). O(V+E) since each vertex and each edge are processed once.
4. When you reach s you are done; return s.p; or continue to compute for all vertices: it is still O(V+E).

Note: one can get the same effect by computing the transpose of G, but this is more expensive than reversing the order of topological sort.

(a) Write the pseudocode.

Solution (students' may differ):

```
SimplePaths(G, s, t)
1  for each v in G.V
2      v.p = 0
3  t.p = 1
4  Compute topological order of G
5  for each u in reversed topological order of G:
6     for each v in G.adj[u]:
7         u.p = u.p + v.p
8  return s.p
```

(b) Explain why it works.

Solution: This works because the number of paths from a vertex v to t is the sum of the number of paths via each of v's downstream neighbors. Vertices to the left of v in the topological sort cannot contribute to any simple path. Thus, by solving the subproblems in the reverse order of the topological sort, we know that the computed number of paths for all vertices to the right of v must be correct when we compute the number of paths for v.

(c) Analyze its asymptotic run time.

Solution: The runtime is O(V+E): lines 1-3 in O(V) time, line 4 takes O(V+E) time. The loop in lines 5-7 takes O(V+E) time (each edge is visited at most once).
