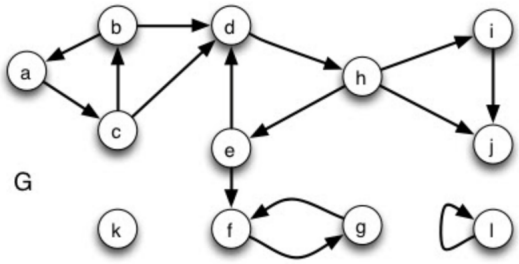
1. SCC (10 pts)

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
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  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. Run DFS on this graph. To make grading and comparison of solutions easier, visit vertices in alphabetical order (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).

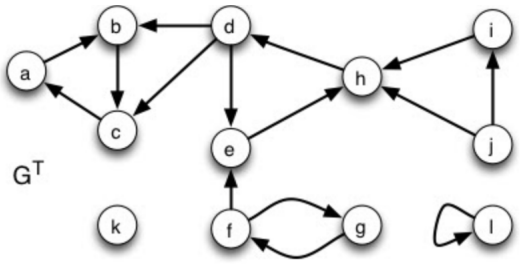
	a	b	c	d	e	f	g	h	i	j	k	l
d	1	3	2	4	8	7	8	5	12	13	21	23
f	20	18	19	17	11	10	9	16	15	14	22	24
π	NIL	c	a	b	h	e	f	d	h	i	NIL	NIL

b. Write the vertices in order from largest to smallest finish time:

Order from smallest to largest goes from left to right:

LARGEST l >k >a > c >b >d >h >i >j >e >f >g **SMALLEST**

2. Now run DFS on the transpose graph, visiting vertices in order of largest to smallest finish time 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	13	3	1
f	10	9	8	16	15	24	23	14	18	14	4	2
π	NIL	a	b	NIL	e	NIL	f	e	NIL	NIL	NIL	NIL

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

SCC 1: Tree Edges: ; Vertices: l
SCC 2: Tree Edges: 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 $theta(V + E)$ 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

Algorithm 1: Longest-Path-Bottom-Up(G,s,t,dist,prev)

```
1  let dist[1...n] be a new Array;
2  let prev[1...n] be a new Array;
3  Topological-Sort(G) ;
4  for i = 1 to G.V do
5  |  dist[i] = - ∞ ;
6  end
7  dist[s] == 0 ;
8  for each element G.V in topological order, starting from S do
9  |  for edge (u,v) within G.adj[u] do
10 |  |  if dist[u] + w(u,v) is an element of G.adj[u] then
11 |  |  |  dist[v] = dist[u] + w(u,v);
12 |  |  |  prev[v] = u ;
13 |  |  end
14 |  end
15 end
16 return (dist,next) ;
```

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

This algorithm works because it sorts the given path to choose the next adjacent vertex. By sorting the subsections of the graph into smaller ordered portions, once the vertex at u is discovered, all other given vertices have already been processed in topological order. Given this information, all of the directed edges of u are taken into account

(c) Analyze its asymptotic run time.

The given asymptotic run time would be $\theta(V + E)$. Since this is based off of Topological Sort, the given asymptotic run time of that particular algorithm is $\theta(V + E)$. Starting the algorithm, the first prominent feature is on line 4 to 5; since $i = 1$ must traverse every vertice, the average of this gives the run time of $\theta(V)$. Given line 7 of this algorithm will run |V| times, it is known that the number of vertices traversed will also traverse the number of edges.
*Got help from Matthew Kirts in Section 01 on Question 02. *

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 to go both ways) over "links:. We can model such a network using a directed graph G = (V,E): Nodes within a strongly connected 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.

Algorithm 2: Discovering-Bad-Networks

```
for each node u within G.V do
    Run DFS(Depth-First-Search) beginning from u ;
    if vertex v (a black node) is visited then
        return FALSE // This is not a bad network ;
    end
end
return TRUE // This is a bad network;
```

(b) Explain why it works.

This algorithm works because it considers the paths to given nodes and executes these conditionals if they are true to detect the bad networks. By maintaining the properties of Depth-First-Search(DFS), it checks to see the children of the graph and follows the forward edge of (u,v), along with the cross edge (w,v). The detection of the forward and cross edges are checked by traversing using the method of Depth-First-Search(DFS).

(c) Analyze its asymptotic run time.

The run time of this given algorithm is $O(V(V + E))$, since it must run through each given iteration of Depth-First-Search, which is known to run in $O(V + E)$ run time.

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 returned 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 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.

(a) Show the pseudocode.

Algorithm 3: Counting-Simple-Paths-In-Dag(G,s,t)

```
1  for each v in G.V do
2  |  v.pi = 0 ;
3  end
4  t.p = 1 ;
5  Find the Transpose of Graph G as G^T = (V, E^T) ;
6  Find the topological order of Transpose Graph G G^T ;
7  for each v within the topological order of transpose G^T do
8  |  if v == s then
9  |  |  return v.pi ;
10 |  end
11 |  for each given u in Transpose G^T.adj[v] do
12 |  |  v.pi = v.pi + u.pi ;
13 |  end
14 end
```

(b) Explain why it works.

This algorithm works because it takes into consideration the given number of paths. The path from a given vertex v to t will be the sum of the number of paths that has a given edge to v. The given values of the neighbors of these paths are sorted within the topological order of the graph, so t would have already been within the path visited.

(c) Analyze its asymptotic run time.
The given asymptotic run time is $O(V + E)$. Analyzing the algorithm by lines, lines 1-3 will run in $O(V)$, lines 5-6 is $O(V + E)$, which depends on the transpose of the graph processing G(V,E). The loop that runs through the transpose of the graph is also $O(V + E)$. Therefore, the final run time of this algorithm is $O(V + E)$.
*Got help from Matthew Kirts in Section 01 on Question 04. *