ICS 311, Spring 2016, Problem Set 08, Topic 14

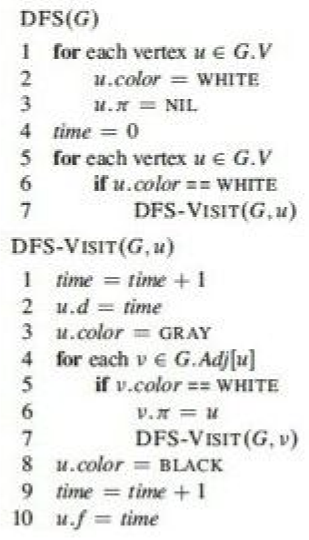
This is a 32 point homework. The extra 2 points are extra credit opportunities.

#1. Peer Credit Assignment

1 Point Extra Credit for replying

You should have named your group partners and allocated 6 points total across them for class the previous week. If you did not, please email the TA with your points assignment.

#2 (8 pts) DFS and Cycles



```
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
```

DFS classifies edges as tree edges, back edges, forward edges, and cross edges (see p. 609).

(a) (2 pts) How can you modify DFS to detect back-edges? Say how you would modify the DFS and/or DFS-Visit procedures to print out all back edges found (reference CLRS line numbers to state your solution clearly), and then justify why your solution leads to the correct result.

Solution: A back-edge is an edge that goes back to an ancestor in a tree. According to the algorithm, ancestors will be colored GRAY while their descendents are being actively explored. Thus, we need to look for v.color == GREY. We can insert a block between lines 4 and 5 that says

```
4.1    if v.color == GREY
4.2        print "(", u, ",", v, ")"    // or similar: ok to just say "print u and v"
```

(b) (2 pts) How can you modify DFS to determine whether a graph has a cycle? Say how you would modify the DFS and/or DFS-Visit procedures to print something and return when a cycle is found, and then justify why your solution leads to the correct result.

Solution: If a graph has a cycle, then the DFS search that starts from one vertex of the cycle will eventually follow a back-edge to that vertex. Conversely, if a DFS search finds a back-edge, there must be a cycle involving the vertex reached by that back-edge. Therefore all we need to do is change the line 4.2 from the above solution to report the cycle (e.g., return immediately if that is all we care about, or set a flag if we wish the search to continue).

(c) (2 pts) What is the asymptotic run time of this modified cycle-detection algorithm on directed graphs? Assume that it exits the DFS as soon as a cycle is found. Justify your answer.

Solution: O(*V* + *E*). This is the same as for DFS itself. We need to check each vertex at least once in line 5 of DFS, and additionally may process each edge in line 4 of DFS-VISIT (in a DAG, every edge will be processed, as there is no cycle to cause exit).

(d) (2 pts) What is the asymptotic run time of this modified cycle-detection algorithm on **un**directed graphs? Assume that it exits the DFS as soon as a cycle is found, and think carefully about how many edges you can have in an undirected graph without cycles. Justify your answer.

Solution: One might think that this is also O(*V* + *E*), but we don't need this much time.

In an undirected graph with |*V*| vertices you cannot have more than |*V*| − 1 edges without having a cycle (since a tree has |*E*| = |*V*| − 1 edges). So, it is impossible to see |V| edges without having detected a cycle along the way: it is O(*V*).

The reason that this argument does not apply to directed graphs is it is possible to have a DAG with more than |*V*| − 1 edges, for example: *G* = ({a, b, c, d}, {(a, b), (a, c), (a, d), (b, d), (c, d)}} (and one can construct examples where the edge set grows faster than |*V*|).

#3 (8 pts) Bottom-Up Longest-Paths

In Problem Set 7 you wrote `Longest-Path-Memoized`, a recursive (top-down) dynamic programming solution to the longest paths problem. Here you will solve the same problem in Θ(V+E) using a bottom-up dynamic programming approach. *Hint:* We need to arrange to solve smaller problems before larger ones: use topological sort (which you may assume has already been written).

(a) Write the pseudocode for `Longest-Path-Bottom-Up`.

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

(c) Analyze its asymptotic run time.

Solution:

(a) (4 pts)

```
// 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)
let dis[1..n] and next[1..n] be new arrays
topologically sort the vertices of G
for i - 1 to |G.V|
    dist[i] == -∞
dist[s] == 0
for each u ∈ G.V in topological order starting from s
    for each edge (u,v)∈ G.Adj[u]
        if dist[u] + w(u,v) > dist[v]
            dist[v] = dist[u] + w(u,v)
            next[u] = v
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.`

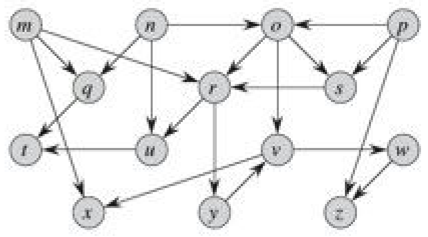
(b) (2 pts) The topological sort ensures that we solve smaller problems before larger ones by ensuring that by the time we get to any vertex we will have already processed all vertices that are in-incident on it (that is, all of the vertices by which we can reach the present vertex already have their 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) (2 pts) 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)**.

#4 (8 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*.

(a) Write the pseudocode.

(b) Explain why it works.

(c) Analyze its asymptotic run time.

Solution:
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 known for all vertices, except that t.p = 1. O(V).

2. Perform a topological sort of G but use the reverse order, which will order t before s. (This is equivalent to performing a topological sort of G^T.) O(V+E)
3. For each v processed in the order of the sort above, set v.p to be the sum of u.p for all incoming neighbors u of v (i.e., for every edge (u,v) in the original graph or every edge (v,u) in the transpose graph). O(V+E) since each vertex and each edge 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).

(a) (4 pts) The pseudocode looks as follows:

```
SimplePaths(G, s, t)
1  for each v in G.V
2      v.p = 0
3  t.p = 1
4  Compute $G^T$=(V, $E^T$), the transpose graph of G
5  Compute topological order of $G^T$
6  for each v in the topological order of $G^T$
7     if v == s
8         return v.p
9     for each u in $G^T$.adj[v]
10       v.p = v.p + u.p
```

(b) (2 pts) This works because the number of paths from a vertex v to t is the sum of the number of paths via each of the neighbors that has an edge to v. We know that when we do the sum the neighbor values are correct because they are topologically closer to t so have already been processed.

(c) (2 pts) The runtime is O(V+E): lines 1-3 in O(V) time, lines 4 and 5 take O(V+E) time each. The loop in lines 6-10 take O(V+E) time (each edge is visited at most once).

#5 (8 pts) Bad Networks

An Internet network can be modeled using a directed graph *G* = (*V, E*): The set of nodes *V* consists of one node for every server in the network and there is an edge (*u,v*) ∈ *E* if and only if there is a direct link from server *u* to server *v*. Any server within a strongly connected component can send and receive packets to and from any server within the same strongly connected component. To ensure uninterrupted connectivity on the network, the 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 goes down, the packets can still be routed 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 if there is at most **one** path between **every** pair of nodes in G. For full credit, your algorithm should run in O(V(V+E)) time.

(a) Write the pseudocode.

(b) Explain why it works.

(c) Analyze its asymptotic run time.

Solution: (in different order: also note that the points are allocated differently than the last problems, as here the explanation is more involved)

(a) (2 pts) The following algorithm identifies really bad networks:

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

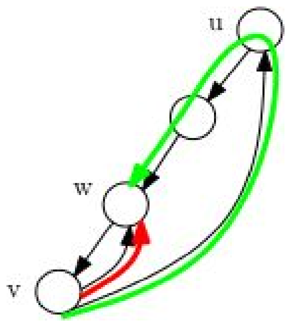
(c) (2 pts) Clearly, the runtime of this algorithm is O(V(V+E)), as it consists of O(V) iterations of DFS, each of which runs in O(V+E) time.

(b) (4 pts) Why the algorithm works: 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 is 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 when we run DFS from node *v*. So we don't have to worry about back edges.

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