ESO207A: Data Structures and Algorithms	
Homework *4b: Basic Graph Algorithms*	HW Due Date: Oct 31, 2018

Problem 1. *Topological Sorting.* In class, a basic algorithm was presented to compute a topological sorting of a given *DAG* G.

repeat
Find a source vertex and delete it from the graph G.
until the graph is empty.

A source vertex is any vertex of the graph that has no incoming edges. Prove the following.

1. Prove that this algorithm produces a valid linearization of the input DAG.
2. Show that this algorithm can be implemented in linear time.

1. Correctness. Let G be the given DAG and let s be a source vertex. Then, we can make s the first vertex in the topological ordering of G, followed by the topological ordering of $G - \{s\}$. This gives an inductive construction. By induction (on the number of vertices in the graph), suppose we have a topological ordering of the vertices of $G - \{s\}$. Thus, in this ordering, all edges are "forward", from a lower numbered vertex to a higher numbered vertex. Also, there are no edges into s, and so all edges out of s are also forward. Hence the inductive construction works.

2. ***Solution 1 modifying DFS.*** Start from any vertex u. Do a DFS from u on G^R. If we reach a sink vertex of G^R, the algorithm wants us to delete it. Now DFS will color it black. This algorithm works by noting that each time a vertex turns black, it is a sink vertex and can be added to the end of the linear list.

```
LINEARIZE(G)
1.  L is a doubly linked list initialized to empty
2.  Compute G^R
3.  for each vertex u in G u.color = white
4.  for  each vertex u in G
5.       if  ucolor== white
6.            LINEARIZE_VISIT(G, u)
```

```
LINEARIZE_VISIT(G, u)
1.  u.color = gray
2.  for every vertex v adjacent to u in G^R
3.       if  v.color == white
4.            LINEARIZE_VISIT(G, v)
5.  u.color = black
6.  Insert u at end of L
```

Solution 2 using stack. Let us design the algorithm in steps. In the first step, let us try to solve the problem of quickly identifying a new source vertex in $G - \{s\}$ after removing a source vertex s of G.

In the class, we used the following algorithm to find a source vertex. Start from any vertex u, and go to any vertex v such that u is adjacent to v (i.e., (v, u) is an edge). Repeat this; the process results in a new vertex each time, otherwise, we have found a cycle: contradiction with DAG. Since the input graph is finite, the process terminates and the final vertex is a source vertex.

FINDASOURCE(G)
1. Let u be any vertex of V.
2. **while** u has an incoming edge
3. $\quad u :=$ any vertex v such that (u, v) is an edge in G^R.
4. **return** u

This is an $O(|V|)$ worst case algorithm. Why? Because it can be implemented as follows by following adjacency pointers in G^R: v is a predecessor of u, or $(v, u) \in E$ iff v is adjacent to u in E^R. Thus following a predecessor vertex of a vertex in G is the same as following an adjacent vertex in G^R. So a symmetric form of this algorithm is :

FINDSINK(G)
1. Let u be any vertex of V.
2. **while** $Adj[u] \neq$ NIL
3. $\quad u :=$ first vertex in the list $Adj[u]$
4. **return** u

Thus, a source vertex of G is found by finding a sink vertex of G^R. Each iteration yields a new vertex or the loop terminates. Statement 3 runs in time $O(1)$. Thus the loop runs in worst-case time $O(|V|)$.

The next step is to identify and remove the source vertex found above from both G and G^R. Let s be the source vertex found in G. Now it can be easily removed from G by setting $Adj[s]$ to NIL (i.e., deleting the adjacency list of s). However, we have to remove s from G^R as well, since we will find the next sink in $G^R - \{s\}$. Say we keep a *removed* flag for each vertex, initialized to all FALSE, and each time we remove a vertex, we set its removed flag to TRUE.

The main loop is modified as follows. We use a stack, though unnecessary for this routine, is useful in the following linearization routine. It does not disturb the stack, it adds u and all the vertices that it navigates one on top of the other, except for the source vertex.

Global: Set *removed* field to false for all vertices. Stack S is given.
FINDSOURCEANDREMOVE(G, u) // Finds source vertex starting from u in G.
Assumes G^R is given and that $u.removed$ is FALSE.

```
1.  PUSH(S, u)
2.  sourcefound = FALSE
3.  while not sourcefound
4.      repeat
5.          step through the vertices of the list Adj^R[u] in order
6.          Let current vertex be w
7.          if w is not NIL and w.removed == TRUE
8.              delete it from its position in the list Adj^R[u]
9.      until either the list Adj^R[u] is exhausted or w.removed == FALSE
10.     if w == NIL
11.         sourcefound = TRUE // u is source: has no valid predecessors in G
12.     else // w is a predecessor of u
13.         PUSH(S, w)
14.         u = w
15. u.removed = TRUE
16. return u // this is the vertex returned as source vertex in the current G
```

Using the above routine, let us write the linearization routine. We will use an outer loop that runs over all vertices, similar to DFS, to ensure that no vertex is left out. We adapt the procedure FINDSOURCEANDREMOVE(G, u) similar to DFS_Visit.

We can now write the following algorithm for obtaining a linearization order.

```
LINEARIZE(G)
1.  Compute G^R
1.  for all vertices u
2.      u.removed =FALSE
3.      u.exploring = FALSE
4.      u.nremoved = 0
5.  L is the output list (doubly linked) initialized to empty
6.  for all vertices u
7.      if u.exploring == FALSE and u.removed == FALSE
8.          FINDSOURCEANDREMOVE2(G, u, L)    //
```

The procedure FINDSOURCEANDREMOVE2(G, u, L) is similar to FINDSOURCEANDREMOVE except that it places the vertices on a stack and pops them as needed until the stack is empty. Each time a source vertex is found, it is appended to L .

```
FINDSOURCEANDREMOVE2(G, u, L)
// assumes u.removed is FALSE.
1.  PUSH(S, u)
2.  while not ISEMPTY(S)
3.      u = TOP(S)
4.      sourcefound = FALSE
```

```
5.      while not sourcefound
6.          repeat
7.              step through the vertices of the list Adj^R[u] in order
8.              Let current vertex be w
9.              if w is not NIL and is a removed vertex
10.                 delete it from its position in the list Adj^R[u]
11.         until either the list Adj^R[u] is exhausted or w.removed == FALSE
12.         if w == NIL
13.             sourcefound = TRUE
14.         else
15.             PUSH(S, w)
16.             w.exploring == TRUE
17.             u = w
18.     u.removed = TRUE
19.     append u to end of L
20.     POP(S)
```

The code is very similar to DFS. There is an outer loop that runs over all vertices. Every edge $(u, v) \in G^R$ is traversed at most *twice*, once when u is on top of stack, and pushes v on its top, and secondly, when v has been deleted and (u, v) is traversed. at this instant, in $O(1)$ time, the edge (u, v) is deleted. So the total time is linear in the size of the graph.

Problem 2. Design a (simple) linear-time algorithm that given an undirected graph G and a specific edge e, determines whether G has a cycle containing the edge e.

Solution. Run DFS on G and do the following. Consider the first time the edge e is traversed. There are two cases:

1. e is a backedge. In this case e is obviously part of a cycle, and the cycle can be printed.
2. $e = (a, b)$ is a tree edge (direction of traversal is u to v). Now start a second phase of the algorithm. For every back-edge (w, x) found in DFS, check if w is a descendant of v and x is an ancestor of u. If so, we have found a cycle including the edge e.

The modifications to DFS and pseudo-code are given below. The edge $e = a, b$ is given globally.

```
// The edge e = {a, b} is given globally. Checks if e is in some cycle.
// Global variable: Isincycle
ISINCYCLE(G, e)
1.  for all vertices u of G    u.color = white
2.  time = 0
3.  state = NOTFOUND // edge e has not yet been found
4.  Isincycle = FALSE
5.  DFS_VISIT(G, a)
6.  return Isincycle

DFS_VISIT(G, u)
```

```
1.  time = time +1
2.  u.d = time
3.  u.color = gray
4.  for every vertex v adjacent to u in G
5.      if (u, v) is the edge (a, b) or (u, v) is (b, a)
6.          if v.color == gray // back-edge
7.              IsinCycle = TRUE
8.              state = BACKEDGE // the cycle can be printed as well
9.              return
10.         else if v.color == white
11.             v.parent = u
12.             state = TREEEDGE // e is a tree edge
13.             DFS_Visit(G, v)
14.     else // (u, v) is not e
15.         if state == TREEEDGE and a.color == gray and b.color == gray
16.             if v.color == gray // back edge
17.                 if v.d < min(a.d, b.d)
18.                     IsinCycle = TRUE
19.                     // cycle is the tree edges from v to u and (u, v)
20.                     return
21.             else if v.color == white
22.                 v.parent = u
23.                 DFS_Visit(G, v)
24. u.color = black
25. time = time +1
26. u.f = time
27. // terminate as early as possible. Not necessary for correctness though.
28. if u == a or u == b   // e will not be part of any cycle in future
29.     return
```

Problem 3. *Modeling using Graphs.* There is a city *Graphotopia* where all the streets are one-way. The mayor claims that there is a way to drive legally from any intersection in town to any other intersection. The opposition is not convinced and you have to test the veracity of the mayor's claims.

1. Model the problem graph theoretically and show how can it be solved in linear time.
2. Suppose the above claim is false. The mayor now changes her claim. She claims that if one starts at town hall via the one-way streets, then no matter where you reach, you can always drive legally back to the town hall. Model this claim graph theoretically and give an algorithm to solve it in linear time.

Solution. 1. The city can be modeled as a directed graph G whose vertices are the intersections and all one-way streets between consecutive intersections are directed edges. Then the mayor's claim can be modelled as saying that G is strongly connected. We can run the strongly connected components algorithm (which runs in linear time) and check if the graph has only one strong component.

2. Let c denote the town hall vertex. Then the claim says that c is in a sink strong component of G. Suppose c is in a sink strong component and suppose there is a path from c to v. If there is no path from v to c, then, the strong component to which c belongs does not contain v, and so the strong component of c is not a sink component: contradiction. Hence, if c is in a sink strong component then there is a path from c to v and a path from v to c.

Conversely, suppose c is not in a sink strong component. Then, there is a vertex v "downstream" from c (that is there is a vertex v with a directed path in G from c) such that v lies in a different strong component. Thus there is no path from v back to c in G.

Hence the mayor's claim is equivalent to saying that c is part of a sink component. One way to check this is to obtain G^{scc} and check if c is part of a sink component. The details are as follows.

A slightly simpler algorithm is given below. Let G_c be the graph consisting of vertices reachable from c in G, with the edges induced from G on these set of vertices. Let G_c^R be the reverse graph (edges reversed).

1. Find *any* source component C of G_c^R.
2. Choose any vertex v from this source component C.
3. Check if there is a path from v to c in G. If so, then c is in a sink component, else, c is not in a sink component.

Why does this work? Let C be any source component of G_c^R, or a sink component of G. Let v be any vertex of C. If there is a path from v to c in G and we know that there is a path from c to all vertices in G_c, we know that v and c lie in the same strongly connected component. Since v is in a sink component, c is in too. Conversely, if there is no path from v to c, then, v and c lie in different strong components, and so c is not a sink component of G.

G_c can be computed in worst case linear time $O(|V| + |E|)$.

MakeG$_c(G, c)$
1. **for** all vertices u $u.color = white$
2. **for** all vertices u $Adj_c[u]$ =NIL
3. DFS_Visit(G, c)

DFS_Visit(G, u)
4. $u.color = gray$
5. **for** every vertex v adjacent to u in G
6. **if** $v.$ $color == white$
7. Insert v to the adjacency list $Adj_c[u]$
8. DFS_Visit(G, v)
9. $u.color = black$

From G_c, G_c^R can be found in linear time as discussed in class.

Finding some vertex in a source component of G_c^R can be easily found as follows.

1. Do a DFS on G_c^R.

2. Return the vertex with the largest finish time $u.f$.

As argued in the class and text, the vertex with the largest finish time belongs to a source component. For completeness, let us argue this once more. Let u be a vertex with the largest finish time and say it belongs to strong component C. Suppose C is not a source component of G_c^R and there is a strong component C' and an edge $e = (x, y)$ such that $x \in C'$ and $y \in C$, with C' different from C. Now in the DFS we run in Step 1, consider the first vertex of $C \cup C'$ that is visited. If this vertex say z is in C', then, all vertices of C would be descendants of this vertex and hence will finish before z and in particular $z.f > u.f$, which is a contradiction. Otherwise, say some vertex $q \in C$ is visited first. Then, all other vertices of C will be descendants of q and q will have the latest finish time of all vertices in C. However, none of the vertices in C' will be reached and therefore, every vertex in C' will be visited only after all the vertices in C have finished, and they will finish even later. In this case, $v.f > u.f$, for every vertex $v \in C'$–implying again a contradiction.

Hence the vertex with the largest finish time is a member of a source component.

To complete the last step, check if there is a path from c to u, run DFS from c and check if u is reachable from c.

Problem 4. You are given a curriculum of courses for a certain department in a certain , all of which are compulsory. The pre-requisite graph G is a directed graph that has a node for each course and there is an edge from course u to course v if course u is a pre-requisite for course v. Design an algorithm that takes as input G and finds the minimum number of semesters necessary to complete the curriculum (assuming that a student can take any number of courses in a semester).

Solution. The course graph is obviously directed and acyclic. The main idea is to process the vertices in a topological order of the DAG. So we first do a topological sort of the vertices in the graph. Next, we process the vertices in this order. Without loss of generality, let the topological order be $v_1, v_2, \ldots, v_n$. Let t_i be the earliest time at which all the courses corresponding to vertices $v_1, \ldots, v_i$ can be finished. Maintain the invariant that after processing v_i, t_i has been computed.

So having computed $t_1, \ldots, t_i$, how do we compute t_{i+1}. Consider v_{i+1}. The only vertices that can have an edge into v_{i+1} are $v_1, \ldots, v_i$. If (v_j, v_{i+1}) is an edge and v_j can be completed at time t_j, then, certainly, $t_{i+1} \geq t_j + 1$. Hence,

$$t_{j+1} = \max_{(v_j, v_{i+1}) \text{ is an edge}} t_j + 1 \ .$$

If v_j has no predecessor, then t_j is 0. We can write this down.

EarliestFinish(G)
1. Topologically sort the vertices of the course DAG G
2. **for** each vertex u $u.t = 0$
3. **for** every vertex u in increasing topological order
4. **for** every vertex v such that u is adjacent to v in G // $(v, u) \in E$
5. $u.t = \max(u.t, v.t + 1)$
6. **return** $u_n.t$ where u_n is the last vertex in the topological order.