ESO207: Data Structures and Algorithms *Time : 90min.*

Quiz: 2 *Max. Marks: 100*

Instructions.

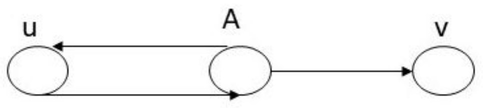
a. The exam is closed-book and closed-notes. You may not have your cell phone on your person.

b. You may use algorithms done in the class as subroutines and cite their properties.

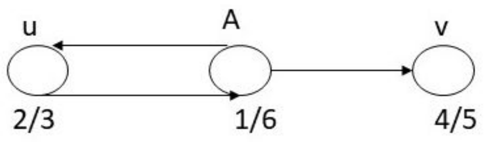
c. Describe your algorithms completely and precisely in English, preferably using pseudo-code. Grading will be based also on the justification given and on the clarity of your arguments. Always argue correctness of your algorithms and analyze them.

Problem 1.

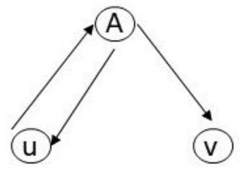
a. Give an example of a graph $G = (V, E)$ with two named vertices u and v (there may be other vertices also) such that (1) there is a path from u to v in G, and, (2) A DFS is run on G that gives $u.d < v.d$ but v is not a descendant of u in the DFS-tree. (In other words, the pre-conditions of the white path theorem do not fully hold and hence the statement of the theorem fails). Show the trace of your DFS run. (7)



Graph G



DFS start and finish times



DFS tree starting from root A

b. Given an undirected graph $G = (V, E)$, let *twodegree*$[u]$ denote the sum of the degrees of the neighbors of u. Given an $O(|V| + |E|)$ time algorithm to compute the entire array *twodegree*$[\cdot]$, given that the graph is given in adjacency list format. (7)

Soln.

```
Twodegree(G)
1.  // Assume $V$ is the set $\{1, 2, \ldots, n\}$. All vertices are numbered from 1 to $|V|$
2.  Create two new arrays $deg[1 \ldots |V|]$ and $twodeg[1 \ldots |V|]$
3.  Initialize entries of both these arrays to all zeros
4.  // First calculate degree($u$) for every vertex $u \in V$
5.  for every $u \in V$
6.      $p = Adj[u]$
7.      $deg[u] = 0$
8.      while $p \neq$ NIL
9.          $deg[u] = deg[u] + 1$
10.         $p = p.next$
11. // Now calculate $twodegree(u)$ for every vertex $u \in V$
12. for every $u \in V$
13.     $q = Adj[u]$
14.     while $q \neq$ NIL
15.         $v = q.vertex$
16.         $twodeg[u] = twodeg[u] + deg[v]$
17.         $q = q.next$
```

The algorithm runs in time $O(|V| + |E|)$, and makes two passes over the adjacency list of G.

c. The mayor of the city Computopia has made all streets one-way. The mayor claims that the there is a way to drive legally from any intersection of the city to any other intersection. Formulate this problem graph-theoretically and show how it can be solved in linear time. (6)

Soln. Model the graph by keeping intersections as vertices and streets between intersections as edges. Now check if the graph is a single strong component. If so, then we can drive from any intersection to any other intersection, and otherwise, there are two or more strong components, implying that there are pairs of intersections such that from one it is not possible to drive to the other. Find the decomposition of vertices into strongly connected components takes time $O(|V| + |E|)$ (essentially two calls to DFS).

Problem 2. You are given a tree $T = (V, E)$ in adjacency list format along with a designated root node r. Design an $O(|V|)$ time pre-processing algorithm for T so that queries of the form "is u an ancestor of v" can be answered in $O(1)$ time. (20)

Soln. Do a DFS on T starting at the root node r. For every node $u \in V$, keep the interval $[u.s, u.f]$ as the starting and finish times when the node was accessed by DFS. The DFS tree obtained by running DFS from the root is the tree T itself with the ancestor descendant relationship being identical. Then, by the parenthesis theorem, u is an ancestor of v iff the interval $[u.s, u.f]$ contains the interval $[v.s, v.f]$. The DFS takes time $O(|E|) = O(|V|) = O(|V| + |E|)$. The check whether the interval for v is contained in the interval for u is done in $O(1)$ time from the intervals $[u.s, u.f]$ and $[v.s, v.f]$ (i.e., $u.s \leq v.s$ and $v.f \leq u.f$).

Problem 3. Let $G = (V, E)$ be a directed acyclic graph where the nodes may be thought of as tasks. Each task requires unit time. A node s that does not have any incoming edge in G may start at the earliest time 0 and terminates at time 1. In general, a task u may not begin until all the tasks it depends upon, namely $\{w \mid (w, u) \in E\}$ have terminated. Give an algorithm that runs in $O(|V| + |E|)$ to compute the earliest time when all the tasks corresponding to the graph nodes have terminated. (20)

Soln. Let v be a vertex and (u, v) be an edge. Let $t(u)$ and $t(v)$ be the earliest time that u and v can finish. Hence, $t(v)$ cannot be smaller than $t(u) + 1$, since, v can start after u finishes and takes unit time. For the smallest completion time of v, we have,

$$t(v) = 1 + \min_{(u,v)\in E} t(u) \ .$$

Note that for any v, all vertices u such that there is an edge $(u, v) \in E$ appear earlier than v in the linearized order (topological order) of G. To initialize, set $t(w) = 0$ for all vertices w. Now to find the latest among all the completion times of the vertices, make a pass over all the vertices and return the largest completion time among vertices. In this algorithm, for every vertex v, we go over the adjacency list of v in $Adj^R[v]$, which is also denoted as $Pred(v)$.

A summary description of the algorithm is given below.

```
1.  for each $u \in V$ such that $Pred[u]$ is empty { $u.t = 1$ }
2.  Linearize the vertices in $G$
3.  for $v \in V$ considered in this order
4.      $v.t = 1 + \max_{(u,v)\in E} u.t$
```

The following algorithm is more detailed. Note that the adjacency list of G^R can be calculated in time $O(|V| + |E|)$.

```
1.  for each $u \in V$ such that $Pred[u]$ is empty { $u.t = 1$ }
2.  Linearize the vertices of $G$ (Topological Sort)
3.  for each $v \in V$ in this linearized order
4.      for each $u \in Pred[v]$
5.          if $u.t < v.t + 1$
6.              $u.t = v.t + 1$
7.  return $\max_{u\in V} u.t$
```

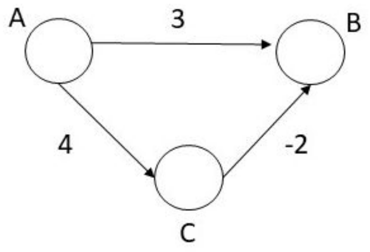
Problem 4.

a. Consider the following algorithm for finding the shortest path from node s to node t in a directed graph with some negative edges. Add a large enough constant to each edge-weight so that all weights become positive, then run Dijkstra's algorithm starting at node s and return the shortest path found to node t. Is this a valid method? Either prove that it works correctly, or give a counterexample. (10)

Soln. The following diagram shows an original graph and its weights, and the second graph which is the same as first except that all weights are increased by 3 to make them positive. The shortest path between A and B returned in the second graph is $A \to B$ and is not the same as the correct shortest path in the first graph which is $A \to C \to B$.

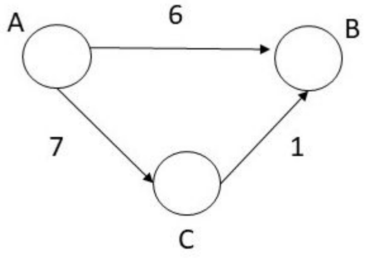
b. Does Kruskal's algorithm work correctly for finding a minimum spanning tree for an undirected graph with negative edge weights. Either prove that it works correctly, or give a counterexample. (10)

Soln. Kruskal's and Prim's algorithms both work correctly for finding an MST with possible negative edge weights. There are two ways to see this. First, the correctness of the algorithms are based on the cut-property which no where uses the assumption that edge weights are non-negative.



Running Dijkstra's algorithm.
Shortest path from A to B is incorrectly calculated as A->B of weight 3.
Actual Shortest path from A to B is via C of weight 4-2 = 2.

Adding 3 to all weights



Running Dijkstra's algorithm.
Shortest path from A to B is correctly calculated as A->B of weight 6.

So the cut property holds if edge weights are positive or negative. Hence the correctness of both Kruskal and Prim's algorithm follow.

Alternatively, let G be the undirected graph where w_e is the weight of edge $e \in E$ and let m be the lowest negative weight $m = \min_e w_e$. Consider the altered weight graph G' such that we add m to all the weights, so that $w'_e = -m + w_e$. Let T be a spanning tree for G, which will also be a spanning tree for G'. Then, $w(T) = \sum_{e \in T} w_e$ and

$$w'(T) = \sum_{e \in T} w'_e = (n-1)(-m) + \sum_{e \in T} w_e = w(T) + (n-1)(-m)$$

Thus, the weight of the spanning tree under the translated edge weight by $-m$ itself translates the whole weight of the tree by $(n-1)(-m)$, since there are $n-1$ edges in any spanning tree and each of them is increased by $-m$. Thus, the optimal tree(s) is (are) unchanged. We can find the optimal tree for the shifted weights and then decrease its cost by $(n-1)(-m)$ to get the optimal tree cost for the original weights.

Problem 5. Either prove the following statements or give a counterexample in each case.

1. Let e be any edge of minimum weight in G. Then e must be part of some MST. (10)

   Soln. Use the cut-property. Let $X = \phi$ be the set of edges that is part of some MST. Let $e = \{u, v\}$ be an edge of minimum weight in G. Now consider the cut $C = (\{u\}, V - \{u\})$. Then $\{u, v\}$ is a cut-edge and is the globally minimum weight edge, so it is a minimum weight cut-edge of C. By the cut-property $X \cup \{e\} = \{e\}$ is a part of some MST.

2. If the lightest edge in a graph is unique, then it must be part of every MST. (10)

   Soln. Let $e = \{u, v\}$ be the lightest edge and let T be an MST not containing e. Consider the cut $C = (\{u\}, V - \{u\})$. Then, $T \cup \{e\}$ has a cycle D containing e, since in T there is a different path from u to v. Consider the edges in the cycle D, there is at least one edge e' that crosses the cut, say it is $e' = \{u, w\}$. Now let $T' = (T \cup \{e\}) - \{e'\}$. T' is connected and has $n-1$ edges and so is a tree. Now,

$$w(T') = w(T) + w(e) - w(e')$$

but $w(e) < w(e')$, since e is given to be the unique lightest edge. So $w(T') < w(T)$, which means that T is not an MST–contradiction. Hence the lightest edge being unique, is part of every MST.