1.5em 0pt
CS 341, Fall 2021 T. Brown, A. Lubiw

ASSIGNMENT 5

DUE: Tuesday November 9, 11:59 PM. DO NOT COPY. ACKNOWLEDGE YOUR SOURCES.

Please read `http://www.student.cs.uwaterloo.ca/~cs341` for general instructions and policies.

1. [10 marks]

   Let $G = (V, E)$ be a **directed** graph with n vertices and m edges. Suppose the vertices are labeled $\{1, 2, ..., n\}$, and edges are provided as an **array** A **of** (s, t) **pairs**, where the edge (s, t) goes from vertex s to vertex t.

   In each of the following parts, unless otherwise stated, you should give (i) a high level description, (ii) pseudocode, (iii) a correctness argument, and (iv) runtime analysis.

   (a) [1.5 marks] Provide pseudocode for a procedure *GetAdjRepr* that takes n, m and the array of edges as input, and returns an adjacency list representation A of the graph. In this part, you do **not** need to give (i) a high level description, or (iii) a correctness argument.

   Assume that link list has addFront method that takes a item and adds it to the front of the list.

```
procedure GETADJREPR($n, m, A$])
    $AJ$ = new Array[{null link list}$_1$, {null link list}$_2$...{null link list}$_n$]
    for $i$ in $A$ do
        $AJ[i.s].addFront(i.t)$
    return $AJ$
```

   To initialize the AJ takes $\theta(n)$ time since you have to create a null list for all n vertices. The one line inside the for loop takes $\theta(1)$ because accessing list at $A[i.s]$ takes $\theta(1)$. Then adding the to the link list also takes $\theta(1)$ thus, everything inside for loop takes $\theta(1)$. The for loop goes through all the vertices and there are m of them, thus, the for loop takes $\theta(m)$. Combining the foor loop and the AJ intalization takes $\theta(n+m)$ time.

   (b) [1.5 marks] Provide pseudocode for a procedure *ReverseAdj* that takes n, m and **the output of (a)** as input, and returns an adjacency list representation R of the *reverse graph* in which the direction of every edge is reversed (so if $(s, t) \in A$ then $(t, s) \in R$). In this part, you do **not** need to give (i) or (iii). **Note: arguments are NOT the same as in (a).**

```
procedure REVERSEADJ($n, m, AJ$])
    $A_{reversed}$ = new Array[{null link list}$_1$, {null link list}$_2$...{null link list}$_n$]
    for $i = 1..n$ do
        $l = A[i]$
        while $l$ is not empty do
            $temp = l.remove()$                ▷ l.remove() removes first item and returns it
            $A_{reversed}[temp].addFront(i)$
    return $A_{reversed}$
```

To initialize the $A_{reversed}$ takes $\theta(n)$ time since you have to create a null list for all n vertices. The remove function takes $\theta(1)$ time in while loop and the line $A_{reversed}[temp].addFront(i)$ inside the while loop takes $\theta(1)$ because accessing list at $A[temp]$ takes $\theta(1)$. Then adding to the link list also takes $\theta(1)$ thus, everything inside for while takes $\theta(1)$. The while loop goes through all the edges starting at i where i is the vertex we are on in the for loop. Thus, since we are going through all the vertices in for loop and then in the while loop we go through all the edges from i, then after the for loop we have visited all the m edges once and the n vetices once and all operations on the vertices and edges took $\theta(1)$ time, thus, the overall time taken in foor loop is $\theta(m+n)$ plus the $\theta(n)$ operation to intalize $A_{reversed}$. Therefore, the time complexity is $\theta(m+n)$

(c) [4 marks] Suppose you are in a maze and you want to escape. The vertices of G are the locations of the maze, and each edge (s, t) describes a movement you can make: specifically from location s to location t. Your goal is to move from where you are, at location 1, to the exit at location n. You have access to a powerful device that can reverse the direction of all edges, allowing you to move backwards along edges. You can use this device whenever you want, but you can only use it once, and its effects are permanent. Once you use it, all edges remain "reversed" forever, and you can only travel along them in their "reversed" direction.

Give an algorithm that determines whether you can escape the maze, returning true if so, and false otherwise.

i)To determine whether you can escape the maze you call DFS with the starting vertex. This will give all the possible destination you can arrive starting at vertex 1. If vertex n is one of the discovered vertices then arriving at vertex n is possible from vertex. If vertex n is not discovered then a DFS should be done from vertex n. If there is a back edge, therefore, connecting the component that has vertex 1, then there you can escape the map. This can be done by going from vertex 1 to vertex i, that was the end point of the backedge. Then from vertex i, you reverse all paths.Since going forward there was a path from n to i, now there is a path from i to n. If there is no backedge these components are disjoint thus, no escape!

ii)

```
procedure canEscape($n, m, AJ$)
    $dfs(1)$                          ▷ calling dfs tells us what vertices are discovered
    if vertex n not discovered then
        edges =$dfs(n)$               ▷ dfs returns all edges in tree
        for edge in edges do
            if edge is backedge then
                return true
    else
        return true
    return false
```

iii) The following algorithm is correct because it determines if the two possible escape paths exit either there is a path from 1 to n where you do not use reverse tool, or you go forward for some distance starting at 1, and then go on reversed edges only, to end at n (using reverse tool). If neither cases yield true then there is no path. Calling DFS(1) discovers all points that can be reached from starting at 1. If the n vertex is one of the discovered points then there is path where reverse tool is not needed. If not, we need to determine if there's a path from 1..n using the reverse tool. Checking this condition is done by calling DFS(n). Calling DFS(n) gives all the points we can arrive starting at n. If theres a backedge then that means there's a path from n to the component containing 1. Thus, there is exists a path where you go forward from 1 to the end point of the backedge, i. Then you use the reverse tool, and you will be able to go from i to n. Thus, this cases check if theres a path from 1 to n utilizing the reverse tool. If both of these cases fail then we know there is no escape so we return false. Therefore, this algorithm check both cases.

iv) The time complexity of this algorithm is $\theta(m+n)$. Calling dfs(1) takes $O(m+n)$, calling dfs(n) takes $O(m+n)$. The if statement in the for loop is $O(1)$, and the for loop iterates maximum m times because it goes through the edges and there are only m edges, so the for loop takes $O(m)$ time. Combining this together, the algorithm takes $O(m+n)$ time.

(d) [3 marks] Suppose we want to *reduce* the problem in part (c) to a *single* call to DFS. Describe how to *construct a graph* G' so that a single call to DFS(G') will solve this problem. Be sure to explain in detail which nodes and edges will be contained in G'. Your algorithm should run in $O(n+m)$ time. For this part, it is **not** necessary to give (ii), (iii) or (iv).

To construct G' you create 2 copies of G, G_1 and G_2.And Let all the vertices in G_1 be renamed from i (where i is between 1 and n) to i' and all the vertices in G_2 be renamed from i (where i is between 1 and n) to i". And in G_2 you reverse all the edges using the ReverseAdj function that we created. Now for all vertices, i', in G_1 connect to i" in G_2. The purpose of this is to allow the person to use the reverse tool, but no edge is made in the other direction since you can use the reverse tool only once. If you start at G' and then go to G" you cannot go back with this new graph. This new graph that is created connecting G_1 and G_2 equals G'.

DFS(1') should be called and if n' or n" is discovered then there is an escape otherwise, there is not. By calling DFS(1') we will get all the points we can reach from 1' travelling in G_1, or all the points we can reach from 1' by travelling in G_1 and then reversing all the edges and travelling in only G_2. If n' is discovered then there is a path from 1 to n without reversing in the original G. And if n" is discovered then there is a path from 1 to some point k, in G, and then all paths are reversed and you can go from k to n. Therefore, using calling DFS(1') considers both cases where you use the reverse tool or not.

2. [10 marks] Let $G = (V, E)$ be an undirected graph with n vertices and m edges, and let $T \subseteq E$ be a spanning tree of G. A *swap* operation on T replaces one edge e of T by an edge f of E such that the result, $T - \{e\} \cup \{f\}$, is again a spanning tree.

   Given two spanning trees T and S of G, the goal of this question is to find a minimum length sequence of swaps that changes T into S.

   To be precise, we want a sequence of pairs of edges $(e_1, f_1), (e_2, f_2), \dots, (e_k, f_k)$ such that

   - $T_0 = T$
   - for $i = 1, \dots, k$, let T_i be $T_{i-1} - \{e_i\} \cup \{f_i\}$. Each T_i **MUST** be a spanning tree.
   - $T_k = S$

   and such that k is minimized.

   (a) [0 marks] [Warmup, don't hand this in.] Show that $|T \setminus S| = |S \setminus T|$.

   (b) [4 marks] Prove that the minimum number of swaps required to change T to S is $|T \setminus S|$.

   This proof will first prove that it is not possible to transform T to S in less than $|T \setminus S|$. Then I will show how it is possible to transform T to S in $|T \setminus S|$ swap. Thus, proving that the minimum number of swaps to transform T to S is $|T \setminus S|$.

   i. Firstly, in order to change T to S, we must add each of the edges in $S \setminus T$ to T, and there are $|S \setminus T|$ edges we will add. Since a swap only replaces 1 edge, in T, we must do at least $|S \setminus T| = |T \setminus S|$ swaps to get all the edges in S also in T.

   ii. Let $k = |S \setminus T|$ and $S \setminus T$ be represented as $S_1, S_2, ...S_k$ where S_i is an edge in S but not in T. If we first add S_1 to T, then we will have 1 cycle including edge S_1 since adding edge to spanning tree creates 1 cycle. Since S did not have a cycle with S_1 then there must be at least one edge that is in cycle and is not in S, lets say e_j. If we remove e_j from the graph we have removed the cycle and the graph is still connected because removing edge from cycle is still connected. We have therefore, performed a valid swap since we replaced $e_j \in T$ with $S_1 \in S \in E$ and the graph is connected. If we perform this procedure for all for all $S \setminus T$ we will have performed $|S \setminus T|$ swaps where every edge in T is an edge in S since we never remove an edge in T that is also in S. And by a) that is equal to $|T \setminus S|$.

   iii. Therefore, since $|T \setminus S|$ swaps are required to transform T to S, and it is possible to transform T to S with $|S \setminus T|$ swaps as mentioned in ii, then, the minimum number of swaps required to change T to S is $|S \setminus T|$.

   (c) [4 marks] Give an $O(n^2)$ time algorithm to find a minimum length sequence of swaps to change T into S.

   i)To transform T to S you should first add each $S \setminus T$ element to T one at a time. To determine what edge should be removed for the swap, dfs(T) should be called in order to get all the type of edges. Then you should go through the edges until you arrive at edge that is a non-tree edge since it creates a cycle. This means that if we recursively call parent on one of the vertices that create the non-tree edge we should arrive at the other vertex of the non-tree edge. Once we figured out which vertex parent should be called to arrive at the vertex that is incident with the non-tree edge, then we should determine if the vertex we are on and its parent are in S. If it is not in S then we would have then we remove this edge from T since it removes the cycle, its still a spanning tree, and we are 1 step closer to S. If it is in S then we continue moving up the tree until we find one. One this completes we have transformed T to S.

ii)

```
procedure SWAP($S, T$)
    $A = |S \setminus T|$
    $swapArr$ = empty
    for edge in A do
        $T = T + edge$
        edges= $dfs(T)$
        for e in edges do
            if $e$ ==non-tree edge then
                if $e \notin S$ then
                    $T = T - e$
                    swapArr.add($\{edge, e\}$)
                    break
                if level(e.u) > level(e.v) then
                    lower-vertex = e.u
                    higher-vertex = e.v
                else
                    lower-vertex = e.v
                    higher-vertex = e.u
                while (level(lower-vertex) $\neq$ level(higher-vertex)) do
                    if (parent(lower-vertex), lower-vertex) $\notin S$ then
                        $T = T-$(parent(lower-vertex), lower-vertex)
                        swapArr.add({edge, (parent(lower-vertex), lower-vertex)})
                        break
                    lower-vertex = parent(lower-vertex)
                break
    return swapArr
```

iii) The following algorithm is correct because it adds edges that make it closer to S, but it removes edges that are in T but not S. Furthermore, as explained in part b) the minimum number of swaps is $|T \setminus S|$, and this algorithm forces every element in $S \setminus R$ to be part of swap once resulting in $|T \setminus S|$ swaps.

The algorithm loops through all the elements in $T \setminus S$ since each edge will be part of the swap (ie forcing $|T \setminus S|$ swaps). Then, the edge, e, the edge we are iterating over in A, is added to T and will be part of swap. Then dfs is called on modified graph T, so we can determine where the cycle is created. We know there is a cycle since adding edge to minimum spanning tree creates a cycle. Furthermore, by class notes we know that non-tree edges are part of cycle and lowest level vertex and its ancestors are part of cycle. In order to keep the it a spanning tree the algorithm gets rid of the cycle, and to make it closer to S, an edge not in S must be deleted. Thus, the first 2nd if statement and or the while statement find an edge part of cycle that is not in S to delete. Once this edge, r, is found r and the edge we are on in A are added to swap array. Therefore, by adding all the edges in $|S \setminus T|$ and removing edges in $|T \setminus S|$, we get S, we get minimum number of swaps equalling $T \setminus S$.

iv) The time complexity of this algorithm is $O(n^2)$. The outer for loops iterates over each edge in $S \setminus T$ since there are maximum m edges, and $m \in O(n)$ since spanning tree, then we make maximum n iterations. Calling dfs takes $O(n+m)$ time and since $m \in O(n)$ it takes $O(2n) \in O(n)$. Finally the inner for loop takes $O(n)$. The inner for loop goes over all the edges in T, so there are n iterations. If the edge, e, is not an non-tree edge then it takes $O(1)$. But if it is a non-tree edge then it does some constant time comparison and goes into a while loop. The while loop travels up vertices, in the DFS trees, and since it only reaches each vertex maximum once, and since there are n vertices then the while loop takes $O(n)$ time. However, the while loop in the inner for loop only runs once since, after the while loop you break out of inner for loop. Thus, the time complexity for inner for loop is $O(1)$ each time, and once it may be $O(n)$. Thus, for n iterations in inner for loop it can take $O(n) + n * O(1)$ which belongs in $O(n)$. So adding all that up, the time complexity for everything inside for outer loop is $O(n)+O(1)+O(n) \in O(n)$. And since there are maximum n iteration in outer for loop then the algorithm takes $O(n^2)$. QED

Note: Part (d) asks for a faster more sophisticated algorithm, but you MUST answer this part with a straight-forward $O(n^2)$ algorithm.

(d) [2 marks] Give an $O(n \log n)$ time algorithm to find a minimum length sequence of swaps to change T into S.

Note: This is very a challenging question and we expect that most of you will not be able to do it in a reasonable length of time, which is why it's worth only 2 marks. We will not give part marks, so if you find this question too challenging, please just put it aside.