CS345: Algorithms - II | Due: 19/10/19

Assignment #3

Name: **Ayush Hitesh Soneria**
Roll-No: **170192**
Email: ayushhs@iitk.ac.in OR ayushhs@cse.iitk.ac.in

Instructor: Prof. Nitin Saxena
TA: Abhimanyu
TA's Email: abhimanu@iitk.ac.in

Max Points: 50

Question 1: Merge Sorted Lists

[7 points]

Doing the below problem taking the sorting to be in non-decreasing order.

Algorithm / Idea

```
   Data: $k$ ← # of lists
   Data: $n$ ← # of total numbers
   Data: $S$ ← output list
1  Assign distinct numerical id's to each of the k lists like [1..k]
2  The augmentation of tree needed should be such that each node contains, pointer to left child, pointer to
     right child, the value of the number as key, the id of the list the number originally belonged to
3  Insert-New-Node(ID) is the function which inserts the first number from the list (whose id=ID) to the tree
     and deletes the entry of the number from that list
4  e.g. if list A = [5,23,41], and if first element is inserted into tree from list A, then list A will become [23,41]
5  Create Augmented AVL tree of k nodes, say T, by inserting the first elements from each of the lists, using
     Insert-New-Node function
6  while ((T is not empty) OR (# of nodes in T == 0)) do
7      X ← node with least key in T;                      /* find min in T takes O(logn) */
8      temp ← X.id
9      Append X.key to list S
10     Delete-Node(X)
11     check ← Insert-New-Node(temp);                     /* on error(list temp empty) returns -1 */
12     if (check == -1) then                              /* to see if list with id=X.id is empty */
13         reduce # of nodes in T by 1
14     end
15 end
16 Return S
```

Algorithm 1: Merge k sorted lists containing n numbers $O(nlogk)$

Proof of Correctness & Analysis

At any point of time in the algorithm we can observe that there are atmost k nodes in the tree. Insertion/deletion operations in this tree take the time complexity of standard tree operation which is O(logk), where k is # of nodes. This is because the augmentation does not affect the rotation operation in any way. Tree creation initially takes O(klogk) time as k insertions take place to make the tree of k nodes. We delete and insert n-k nodes after creation of tree, meaning there are O(n) iterations of the while loop. We also observe that $k \leq n$.
Hence time complexity = O(klogk + nlogk) = O(nlogk)

Question 2: Optimized Scheduling

[6 points]

Claim:

Using the greedy paradigm, we can say that the optimal solution would be to schedule the processes in increasing order of their processing time.

Proof:

Let us try and compare the minimized average completion time of the optimal solution with our solution. Let y be the process with least processing time. Let the optimal solution be A. Meaning A is the process scheduling done in the following order (with first process scheduled as x)

$$x, Q_s, ..., Q_k, y, Q_t, ..., Q_e$$

where all these processes are distinct and are some permutation of $P_1, ..., P_n$ and $t_y \leq t_x$

TAx = The completion time of x = t_x
TAqs = The completion time of Q_s = q_s + t_x
...
TAqk = The completion time of Q_k = $q_s + ... + q_k$ + t_x
TAy = The completion time of y = t_y + $q_s + ... + q_k$ + t_x
...
TAqe = The completion time of Q_e = $q_t + ... + q_e$ + t_y + $q_s + ... + q_k$ + t_x
T1 = Average Completion time = (sum of all completion times) / n

Let B be our process scheduling where we swap x with y in A, so we get y as the first process scheduled, so the order will be

$$y, Q_s, ..., Q_k, x, Q_t, ..., Q_e$$

TBy = The completion time of y = t_y
TBqs = The completion time of Q_s = q_s + t_y
...
TBqk = The completion time of Q_k = $q_s + ... + q_k$ + t_y
TBx = The completion time of x = t_x + $q_s + ... + q_k$ + t_y
...
TBqe = The completion time of Q_e = $q_t + ... + q_e$ + t_x + $q_s + ... + q_k$ + t_y
T2 = Average Completion time = (sum of all completion times) / n

Now one can easily verify that as $t_y \leq t_x$ we can say that TBy ≤ TAx.
Similarly TBqs ≤ TAqs, ..., TBqk ≤ TAqk.
But then onwards TBx = TAy, ..., TBqe = TAqe.
Hence we can say that T2 ≤ T1, as T2 is summmation of all TB's and T1 is summation of all TA's.
Which means that y should be the first task scheduled in the optimal solution as B is a better scheduling than A.
Hence in order to get minimized average scheduling time, schedule the processes in increasing order of their processing times.
Hence Proved.

This scheduling problem's time complexity will be O(nlogn), which is due to the sorting step.

Question 3: Minimum Spanning Tree

[5 + 5 + 5 + 5 points]

Sub - Problem 1

Claim: In G=(V,E), the minimum spanning tree (MST) is unique.

Proof:

We will prove this by contradiction. Let us assume there are 2 MSTs A and B, where A=(V,E_A) and B=(V,E_B), E_A,E_B $\in$ E and E_A-E_B, E_B-E_A both not empty.
Let $x \leftarrow$ minimum-weight edge from E_A-E_B OR E_B-E_A. W.l.o.g let's say $x \in E_A - E_B$, meaning $x \in E_A$.
Also $\exists$ y $\in$ E_B-E_A meaning $y \in E_B$ and y != x.

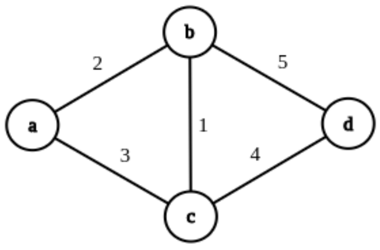
Now we know the concept of cycles in a graph, for a single-cycle connected graph, # of vertices = k and # of edges also = k. Where as for a connected graph with no cycles, e.g. MST, # of vertices = k and # of edges = k-1.

Hence B $\cup$ {x} will be a cycle as # of vertices = k and # of edges also = k-1+1 = k. y will be in the cycle B $\cup$ {x}. Now (B $\cup$ {x})-{y} will be a spanning tree, as cycle not present now. We know that weight(x) < weight(y). And hence total-weight((B $\cup$ {x})-{y}) < total-weight(B). But this is a contradiction as we assumed B to be an MST but we are getting ((B $\cup$ {x})-{y}) as a MST with lesser total-weight. Hence we see that B couldn't have been a MST, and so there couldn't have been 2 MSTs. Therefore MST of G is unique.
Hence Proved.

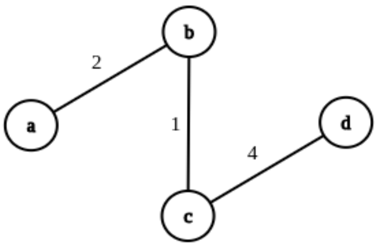
Claim: In G=(V,E), the second best MST need not be unique.

Proof: We will give counter example to an opposite claim, which is that second best MST of G is unique.

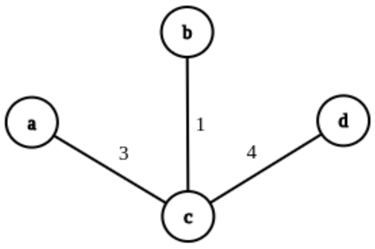
Taking example of a Graph G=({a,b,c,d},{(a,b),(b,d),(a,c),(c,d),(b,c)}



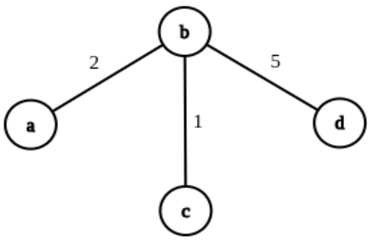
(a) Graph G



(b) unique MST of G



(c) second best MST of G



(d) second best MST of G

Figure 1: Counter-Example to prove non-uniqueness of second best MST of G

Hence claim Proved.

Question 3: Continued

[5 + 5 + 5 + 5 points]

Sub - Problem 2

Claim: There exists edges (u,v) ∈ T and (x,y) ∉ T s.t. T - (u,v) ∪ (x,y) is a second best MST of G.

Proof:

Let T2 be a second best MST and T be MST. Let (u,v) ∈ T - T2. Then, T2 ∪ (u,v) contains a cycle where one of the edges in the cycle is not in T (cycle property). Let this edge be (x,y). Then, we must have weight(x,y) > weight(u,v), for otherwise, we could replace (u,v) in T by (x,y) to get a MST better than T. Now, we note that A = T2 - (x,y) ∪ (u,v) is also a spanning tree since (u,v) and (x,y) are in the same cycle. In addition, total-weight(A) < total-weight(T2). So, A is a best MST. By the uniqueness of MST, we see that A = T.
Hence T = T2 - (x,y) ∪ (u,v), so T2 = T - (u,v) ∪ (x,y).Therefore T and T2 differ with only one edge, i.e. (u,v) replaced by (x,y). Hence Proved.

Sub - Problem 3

Idea / Algorithm & Analysis

As T' is a spanning tree meaning its a connected graph, every node can be reached from every other node. Hence using few heuristics to maintain max in the path between 2 vertices, we can iterate over al possible pair of vertices and find array max, where max[u,v] be an edge of maximum weight on the unique path between u and v in T' for all u,v ∈ V. We want to fill up the table max which has $|V|^2$ entries. For each vertex, the function DFS-NEW takes O(V) time. Hence overall the time complexity is O($|V|^2$).

```
   Data: $V, E \leftarrow$ Vertex and edge sets of $T'$ from graph G respectively
   Data: $W \leftarrow$ weight set of edges
   Result: two-dimensional array max, containing edges
1  Function DFSgraph-NEW(V,E,W):                    /* algorithm as a recursive function */
2      for each vertex $u \in V$ do
3          for each vertex $v \in V$ do
4              max[u,v] = NULL Edge;
5          end
6          DFS-NEW(V,E,W,max,u,u);
7      end
8      Return max;
9  End Function
```

Algorithm 2: finding max array in O($|V|^2$) time

```
1  Function DFS-NEW(V,E,W,max,u,v):                 /* algorithm as a recursive function */
2      for each vertex x, s.t. (v,x) ∈ E do
3          if (max[u,x]==NULL Edge) and (u!=x) then
4              if (v==u) OR (weight(v,x) > weight(max[u,v])) then
5                  max[u,x] ← (v,x);
6              else                                                                  /*  */
7                  max[u,x] ← max[u,v];
8              end
9              DFS-NEW(V,E,W,max,u,x);
10         end
11     end
12 End Function
```

Algorithm 3: Recursive function DFS-NEW(V,E,W,max,u,v)

Question 3: Continued

[5 + 5 + 5 + 5 points]

Sub - Problem 4

Idea:

From Sub-Problem 2, we know that we can replace one edge (u,v) in the MST, say T, by another edge (x,y) to get a second best MST, say T2. Note that if we replace (u,v) by (x,y), then

$$totalweight(T2) = totalweight(T) - weight(u, v) + weight(x, y)$$

If we know which (x,y) to add to T2, then observe that (u,v) must be the max-weight edge in the path from x to y, which can be found by Sub-Problem 3. So, we get the following algorithm:

Algorithm:

```
Data: $V, E \leftarrow$ Vertex and edge sets graph G respectively
1 T ← MST of graph G;
2 Find max-weight edges, 'max' array, as in Sub-Problem 3;
3 Let max[x,y] denote the max-weight edge in the path from x to y in tree T, for all x,y ∈ V;
4 Find an edge (x,y) ∈ E - T that minimizes (weight(x,y) - weight(max[x, y]));
5 Output T2 = (T - max[x,y]) ∪ (x,y);
```

Algorithm 4: Computes Second best MST in $O(|V|^2 log|V|)$ time

Analysis:

Line 1 takes $O(|E|log|V|) = O(|V|^2log|V|)$ time as done in class using AVL tree.
Line 2,3 takes $O(|V|^2)$ as done in Sub-Problem 3.
Line 4 also takes $O(|E|) = O(|V|^2)$ as iteration through all edges takes place.
Hence time complexity of algorithm is $O(|V|^2log|V|)$.

Question 4: Component Graph G^{SCC}

[3 + 3 points]

Sub - Problem 1

Claim: G^{SCC} is a DAG (Directed acyclic graph)

Proof (using contradiction):

First, we observe, by definition that there cannot exist any strongly connected component of G which does not belong to $V^{'}$ — (1)

Now let us assume G^{SCC} to not be a DAG, meaning let's say G^{SCC} has a cycle.
Then there exists strongly connected components $S_1, S_2....S_n \in V^{'}$ which form a cycle in the component graph G^{SCC}.
Then taking all these components together we can get a bigger strongly connected component of G, which does not belong to $V^{'}$. This is contradiction to (1). Hence our assumption that G^{SCC} is not a DAG is wrong. Hence we get that G^{SCC} is indeed a DAG.
Hence proved.

Sub - Problem 2

Claim: $((G^T)^{SCC})^T = G^{SCC}$

Proof:

Let us try and prove this using fundamental basics of graphs.
Property 1: Two graphs are said to be equal if their edge and vertex sets are equal.
Property 2: Vertex sets of G and G^T are equal.
Property 3: Strongly connected component (SCC) of G and G^T are same.

Here first let us try and find relation between vertex sets of $((G^T)^{SCC})^T$ and G^{SCC}.
Using property 3, we can say vertex sets of G^{SCC} and $(G^T)^{SCC}$ are same. Which implies, using property 2, vertex sets of $(G^T)^{SCC}$ and $((G^T)^{SCC})^T$ are same, which further implies that vertex sets of $((G^T)^{SCC})^T$ and G^{SCC} are same.

Now let's find relation between edge sets of $((G^T)^{SCC})^T$ and G^{SCC}.
Let (v_i, v_j) be an edge in $((G^T)^{SCC})^T$. Then (v_j, v_i) will be an edge in $(G^T)^{SCC}$. Now there exists $a \in C_j$ (the SCC denoted by v_j) and $b \in C_i$ (similarly for i), s.t. (a,b) is an edge in G^T. Therefore (b,a) is an edge in G. Since vertex sets haven't been altered in any sense, property 3 holds, and we get that (v_i, v_j) is an edge in G^{SCC}. Hence using this iteratively over the entire set of edges in $((G^T)^{SCC})^T$,we get that all of those edges in $((G^T)^{SCC})^T$ also belong in G^{SCC}. For the other direction, we can similarly prove that all edges in G^{SCC} also belong in $((G^T)^{SCC})^T$. Hence we get the edge sets of $((G^T)^{SCC})^T$ and G^{SCC} equal.

Hence our claim $((G^T)^{SCC})^T = G^{SCC}$ is true using property 1.
Hence proved.

Question 5: Euler Tour

[6 + 5 points]

Sub - Problem 1

Claim: A directed graph G=(V,E) has an Euler Tour iff indeg(v)=outdeg(v) for each v ∈ V.

Proof:

Let us first prove that if G has an Euler Tour then indeg(v)=outdeg(v) for each v ∈ V.
G has an euler tour, T, meaning that T is a cycle. If T is a simple cycle then

$$indeg(v) = outdeg(v) = 1, \forall v \in V$$

so the claim is true.
If T is not a simple cycle then it will have one or more simple cycles. After we remove (delete edges of) one simple cycle from T (and G), then T will still remain Euler tour. Keep removing (deleting edges of) the simple cycles from T (and G) until no edges are left. Now at this point, indeg(v) = outdeg(v) = 0, $\forall v \in V$. Also we observe that when removing a cycle, an in-edge and out-edge of the vertices on the cycle are removed, and hence after a cycle deletion, the in-degree and out-degree of a vertex on the cycle decrease by exactly 1. So the conclusion we come to is that before any deletions took place, indeg(v)=outdeg(v) all vertices v ∈ V.

Here let us claim something which will help us with our next part proof.
Sub-Claim: for any vertex v with indeg(v)=outdeg(v), there must be a path starting from v that comes back to v.
Proof: For any vertex v, there must be a cycle that contains v. Start from v, and chose any outgoing edge of v, say (v, u). Since indeg(u) = outdeg(u) we can pick some outgoing edge of u and continue visiting edges. Each time we pick an edge, we can remove it from further consideration. At each vertex other than v, at the time we visit an entering edge, there must be an outgoing edge left unvisited, since indeg = outdeg for all vertices. The only vertex for which there may not be an unvisited outgoing edge is v, because we started the cycle by visiting one of v's outgoing edges. Since there's always a leaving edge we can visit for any vertex other than v, eventually the cycle must return to v, thus proving the sub-claim.

Now let us prove that if indeg(v)=outdeg(v) for all v ∈ V then G has an Euler tour.
Now we pick a random vertex v, and find a cycle T that comes back to v (as proved above). Delete all the edges of T from G. Each vertex in the updated G has indeg(v)=outdeg(v), so we pick a vertex $v^{'}$ on T that has some edges going out and coming into it. We repeat our process.
Overall we find a cycle T, then another cycle $T^{'}$ that has (at least) a common vertex with T, and so on. We can build a big cycle that goes around T jumps in $T^{'}$ and goes around $T^{'}$ and then comes back to T and finishes in T.

Hence we proved our claim both ways. Hence Proved.

Hence a directed graph G=(V,E) has an Euler Tour iff indeg(v)=outdeg(v) for each v ∈ V.

Question 5: Continued

[6 + 5 points]

Sub - Problem 2

Idea / Algorithm

```
Data: $V, E \leftarrow$ Vertex and edge sets graph G respectively
1 First we can simply check whether indeg(v)=outdeg(v) for all v $\in$ V;
2 if false then return "Euler tour does not exist";
3 else we continue and find the Euler tour;
4 let us define DFSNEW as the modified DFS so that we store, for each edge that we traverse, a pointer to
    the corresponding edge in the adjacency list of G;
5 Pick a vertex v;
6 do DFSNEW on v until we come across a back edge that links back to v;
7 Once we find this cycle, traverse all edges of the cycle, and delete the corresponding edges in the adjacency
    list of G;
8 repeat (from line 5) until no edges remain in G;
```

Algorithm 5: Computes Euler Tour of Graph

Analysis:

lines 1-3 take O($|E|$) time
DFSNEW in line 4 will help the deletion of edges in line 7, by making the deletion to be done in constant time, as no need to search for edges in G because of use of pointers.
lines 4-8, DFSNEW, also takes O($|E|$) time as each edge has to be looked at once and then it is deleted.

Hence time complexity to find out the Euler Tour of G is O($|E|$) time or linear time.