CS 4820, Spring 2014	**Homework 2, Problem 1**
Name: Piyush Maheshwari	
NetID: pm489	
Collaborators: None	

(1) *(10 points)* Consider the following candidate algorithm for computing a minimum weight spanning tree of a graph with n vertices and m edges.

The algorithm builds a subgraph proceeding in rounds and starting from the empty subgraph. In each round, for each connected component of the current subgraph, the algorithm selects an edge of minimum weight that leaves the component. At the end of the round it adds these selected edges to the subgraph. The algorithm stops when the subgraph is connected.

(1.a) Show that the subgraph computed by the algorithm is a minimum weight spanning tree of the input graph. You may assume that the input graph is connected and that all edge weights are distinct.

(1.b) Show how to implement the algorithm in time $O(m \log n)$.

Warning: *This minimum spanning tree algorithm is moderately well known, and it is possible to find analyses of it on the Web. If you use Web sources you must cite them, and citing a Web source that contains an analysis of this algorithm will result in loss of points. For this reason, you are advised to solve the problem using your textbook and lecture notes (and office hours or Piazza, if desired), but not consulting the Web.*

Solutions

(1.a)

Let the input graph be denoted by $G = (V, E)$. Let the sub-graph be T. Initially $T = \{V, \emptyset\}$.

Lemma 1. *The minimum weight edge leaving a component as selected by the algorithm is a minimum weight edge crossing some cut S of graph G.*

Proof. Suppose at any point in the algorithm the sub-graph has components $c_1, c_2, ...c_i$. Now we choose a minimum weight edge leaving component c_j. Let this edge be (v, w). The vertex v lies in component c_j. The vertex w will lie in some component $c_k \neq c_j$ since (v, w) leaves the component c_j. We know that $V - c_j$ is not empty otherwise the algorithm would have terminated. Consider the cut c_j and $V - c_j$. Clearly edge (v, w) is lowest cost edge crossing the cut since (v, w) chosen to be the minimum cost edge with one vertex in c_j and one in $V - c_j$. We can prove this by contradiction as well. Suppose some other edge (v', w') is the minimum cost edge crossing the cut c_j. This means that edge (v', w') is the minimum cost edge leaving the cut. But we assumed earlier that (v, w) is the minimum cost edge leaving the component c_j. Hence this is contradiction which proves that the edge selected by the algorithm (v, w) is also the minimum edge cost crossing the cut c_j. □

Now the algorithm in each round picks up minimum cost edge leaving each connected component. Using Lemma 1 we know that the every such edge chosen is a minimum edge crossing some cut of the graph. Now using the cut property of MST, we know that these edges chosen by the

algorithm would the part of every MST possible. Also since we only pick up edges which connect disconnected components and we stop once the sub graph is connected, we can never have cycles in the sub graph. This means that when the algorithm terminates we will have a spanning tree. Hence the sub graph constructed by this algorithm is a minimum spanning tree of the input graph.

(1.b)

Algorithm

The algorithm starts out by running MakeUnion(V). Initially we make a priority queue for each connected component. Each priority queue contains edges coming out of that In each round we go through each of the connected components that we have, take out an edge from its priority queue and add that to MST that we are building. Then we merge the priority queues of the connected components that the graphs connect. For implementation purposes we can maintain a list of component components that keep decreasing in count and maintain the corresponding priority queue for each component.

Algorithm 1 FIND-MST

```
1:  function FIND-MST(G)
2:      G' = {V, ∅}.
3:      Make-Union(n)                                   ▷ Make a component for each vertex
4:      Make-PriorityQueue(N)   ▷ Make a PQ for each vertex by adding edges for that vertex
5:      while G' is still not connected do
6:          t[] ← ∅
7:          for all c_i in components do
8:              Extract min element u, v from PQ of c_i
9:              Add u, v to t[]
10:         for all u, v in t[] do
11:             Add u, v to G'
12:             x ← find(u)
13:             y ← find(v)
14:             Union(x,y)
15:             Merge-PQ(x, y)
16:     return S
```

If we maintain priority-queues by simple lists, the following operations can be performed as follows -

Extract Min - $O(1)$, Pick the head of the list
Merge-PQ of size m, n - $O(m+n)$, since this is same as merging two sorted lists of size m and n
Make-PQ of size n - $O(nlogn)$, this is simply sorting and storing the list.

If we represent union find data structures using forest representation as discussed in class
find (v) = $O(logn)$
union(x,y) = $O(logn)$

Complexity Analysis

1. Step 3 takes **O(n)**

2. Step 4 takes $O(n_1 log n_1) + O(n_2 log n_2)..O(n_n log n_n)$ where $n_1 + n_2..n_n$ =m (edge list size) This can be asymptotically bounded by **O(mlogm)**

3. Step 12,13,14 happens one for every edge since one an edge has been taken out of the PQ it is not added again. Hence these take **3mlog(n)**

4. Step 14 - There is a subtle issue with merging of PQ's. Since a priority queue for a component contains edges that leave that component, merging two PQ of different components might have edges which go to each other. So the finally merge PQ will have edges that have both end points within the same component. We can handle this issue while doing extract min element from PQ. Whenever we do an extract, if that edge u, v has find(u)= find(v), then we discard that edge and pick the next edge. So for now, we can simple assume that merging lists take linear time. Now we can observe that the size of a component increases by atleast half for the component of lower size. This means that the maximum number of rounds that we can have are $log(n)$. In each round we can have many components merging. But since each merge is linear, if we sum them together they take $O(m)$ time. Hence the total time spent in merge would be **mlogn**.

5. Step 8- In normal case extract min is $O(1)$ but in this case when have to do a find operation for both end points as explained in 4 above. We do a extract-min once for every edge since once an edge has been taken out of the PQ, it is never added again. Hence the total time this operation takes is **0(2mlogn)**

Total running time = **O(n)** + **O(mlogm)** + **3mlog(n)** + **mlogn** + **0(2mlogn)** = **O(mlogn)** using the fact that log(n) is equal to log(m) asymptotically.