Math 182 HW2 Jun Ryu

Section 4

8) We prove this by contradiction.
Suppose the minimum spanning tree is NOT unique: call them T and T', respectively.
We know that T' possesses an edge e' that is not in T, meaning if e' is added to T, it will create a cycle. Suppose we take an edge in this cycle with the most amount of weight. This edge will not be part of any minimum spanning tree, contradicting our original fact that this edge is included.

10) a) Denote this new edge that's being added by e. By adding e, we complete a cycle with the original $v-w$ path in T.
Thus, an efficient algorithm would be to check if every other edge other than e in the cycle has a cost less than c. If that is the case,

then T remains the same. However, if at least 1 edge in the cycle has a cost greater than c, T changes by including e. This algorithm will be run in $O(|E|)$ time since it needs to check all edges in that cycle.

b) As stated above, if T is no longer the min-cost spanning tree, we grab the most expensive edge in the v-w path and replace it with e as defined above. This will take $O(|E|)$ time.

11) We use a trick to make each edge's cost distinct. Let δ be the min. difference between the costs of 2 non-equal edges, and now we subtract δi from each e_i's. Now, we sort these distinct edges, which will remain the same ordering as it was before modifying. Thus, now if we apply Kruskal's algorithm, it should return a unique minimal spanning tree.

18) We slightly modify Dijkstra's algorithm to count for the fact that travel time varies.

Algorithm:

Let S be set of explored nodes

For each $u \in S$, we store $d(u)$, the earliest time we can arrive at u, and $r(u)$. the last site before u

Initially $S = \{s\}$ and $d(s) = 0$

While $S \neq V$

Select a node $v \notin S$ s.t.

$d'(v) = \min_{e=(u,v): u \in S} f_e(d(u))$ is small as possible

Add v to S and set $d(v) = d'(v)$ and $r(v) = u$

End.

Now, we know this algorithm works very similar to Dijkstra's, meaning when a node is picked, we observe all edges connected to that node $\Rightarrow$ take $O(\log n)$ per edge $\Rightarrow$ total time complexity is $O(m \log n)$ polynomial time.

2a) If any one of d_i's $= 0$, we know that node will be isolated.
If all d_i's are > 0, we proceed to sort the numbers s.t.
$d_1 \geq d_2 \geq \dots \geq d_n > 0$. Now,

Assume v_n with degree d_n is removed, giving us a new degree list of
$\{d_1 - 1, d_2 - 1 \dots d_{d_n} - 1 \dots d_{n-2}, d_n - 1\}$.
This works because v_n will be connected to d_n other points, so WLOG we assume v_n is connected to $v_1 \dots v_{d_n}$. Thus, removing v_n will make $v_1 \dots v_{d_n}$ lose a degree. Now, the same process can be repeated to further simplify the graph by removing another point and removing 1 more from the other points' degrees.
This will, in total, take polynomial time since the algorithm is dependent on the number of points in the original list.

Section 5

1) We try to approach this problem recursively:

input : n, a, b

median (n, a, b) :

if $n=1$, then return $\min(A(a+k), B(b+k))$

$k = \lceil \frac{1}{2} n \rceil$

if $A(a+k) < B(b+k)$, then return

median $(k, a + \lfloor \frac{1}{2} n \rfloor, b)$

else return median $(k, a, b + \lfloor \frac{1}{2} n \rfloor)$

What this essentially does is find the median of $A[a+1 ; a+n] \cup B[b+1 : b+n]$.

Since we can't necessarily delete specific numbers from the dataset, we manipulate the fact that we can access individual entries, which is why we keep updating the recursive function using either $a + \lfloor \frac{1}{2} n \rfloor$ or $b + \lfloor \frac{1}{2} n \rfloor$. Notice that the number of queries in total comes out to be $Q(n) = Q(\lceil \frac{1}{2} n \rceil) + 2 = 2 \lceil \log n \rceil = \mathcal{O}(\log n)$

2) We just slightly modify the original divide and conquer algorithm.

We have our formal algorithm:

let $k = \lfloor n/2 \rfloor$

sort $(a_1 \dots a_k) \rightarrow$ return N_1 and $(b_1 \dots b_k)$

sort $(a_{k+1} \dots a_n) \rightarrow$ return N_2 and $(b_{k+1} \dots b_n)$

count the # of significant inversions $\rightarrow$ return N_3

return $N = N_1 + N_2 + N_3$ and merge $(b_1 \dots b_n)$

Now, we modify this algorithm:

if $b_k \leq 2b_n$, then

if $n > k+1$, decrease n by 1

if $n = k+1$, return N_3

if $b_k \geq 2b_n$, then increase N_3 by $n-k$.

if $k > 1$, decrease k by 1

if $k = 1$, return N_3

This works because in the first if loop, no sig. inversions are found, meaning we return the value of N_3 as it is without modifying it. In the second if loop, however, we have counted $n-k$ sig. inversions.

6) We define the algorithm to be the following:

Start with the root r

if r has a lesser value than its two children, r is the local min.

otherwise, move to any smaller child and repeat the loop

We know this alg. will terminate because either a parent will have a smaller value than both its children or we will eventually reach a leaf. In the former case, the "parent" is the local min; in the latter, the leaf is the local min.

In either case, we know it works because the chosen minimum's parent will be of a greater value (hence why the iteration was continued)

7) We borrow a similar idea from above. Essentially, we start with a node on the border of the minimum value. If this node is a corner node, then it is the local min. If not, we extend the path from this node into a neighbor that does not lie on the same border. If the original node has a lesser value than the one traversed, the original node is the local min. Otherwise, we move to the node traversed and recursively explore the adjacent nodes not on the same border. Using this recursive search, we have

$$T(n) = O(n) + T(n/2) = O(n).$$