HW 10 Solution

CS/ECE 374 B: Algorithms & Models of Computation, Spring 2020 Version: **1.0**

1 Problem 28:

Solution 28.A:

We claim that the minimum amount we should decrease the weight of $e = (p, q)$ by $w(p, q) - d(s, q) + d(s, p) + 1$ so that the original shortest-path tree would become invalid. In other words, the new weight of this edge should be $w'(p, q) = d(s, q) - d(s, p) - 1$, and this edge would become an edge in the new shortest-path tree.

We will first prove that the original shortest-path tree is not valid. Suppose the new graph is G', with the decreased edge $e = (p, q)$ with weight $w'(p, q) = d(s, q) - d(s, p) - 1$. In this case, $d_{G'}(s, p) = d_G(s, p)$ since before vertex p no edges have been changed, thus the shortest path from s to p remains the same. $d_{G'}(s, q) \leq d_{G'}(s, p) + w'(p, q) = d_G(s, p) + d_G(s, q) - d_G(s, p) - 1 < d_G(s, q)$. Therefore, there exists a tree with shorter total weights than the original T, and thus, T is invalid.

Then we will need to prove that this is smallest amount we need to decrease to make T invalid. That is, if we decrease by $w(p, q) - d(s, q) + d(s, p)$ or less, the original shortest-path tree would still remain valid. In this case, the new weight of this edge would be $w'(p, q) \geq d(s, q) - d(s, p)$. With the same reasoning from above, $d_G(s, p) = d_{G'}(s, p)$ and with the new weights of $(p, q), d_{G'}(s, q) \geq d_{G'}(s, p) + d_G(s, q) - d_G(s, p) = d_G(s, p) + d_G(s, q) - d_G(s, p) = d_G(s, q)$. Therefore, if we decrease by less amount than claimed, the new path tree would have longer total length than the original shortest-path tree. In this way, we proved that the original shortest-path tree is still valid.

In conclusion, by proving these two conditions, it is enough to decrease the weight of $e = (p, q)$ by $w(p, q) - d(s, q) + d(s, p) + 1$ so that the original shortest-path tree would become invalid.

Solution 28.B:

We claim that the minimum amount of increase of $l(e)$ is defined by the following function:

Algorithm 1 Find the minimum increase amount of $l(e)$ algorithm

Input : Graph G = (V,E)
Output: The minimum increase amount

```
Function minAmount(G):
    Use BFS on the original shortest-path tree T and find the subtree rooted at q
    retval ⟵ ∞
    for e = (u, v) ∈ E and u ∉ subtree rooted at q, and v ∈ subtree rooted at qand do
        if e ≠ (p, q) then
            retval ⟵ min{retval, d(s, u) + w(u, v) − d(s, v) + 1}
    end
    return retval
```

Then we would need to prove the value returned by this function is sufficient and necessary conditions for the original shortest-path tree to become invalid.

First, we want to show that we can only consider the edges in the subtree rooted at q, given the original

shortest-path tree. Since $e = (p, q)$ is the only edge we changed, the new shortest-path tree would not contain this edge. In this case, we can leave the edges whose both ends are outside of the subtree rooted at q since the shortest path from s to these vertices would not go through e, and by then would not change. For the edges whose both ends are within the subtree rooted at q, $d(s, t) = d(s, q) + d(q, t), t \in$ subtree rooted at$q$. Therefore, $d(q, t)$ would not be affected by the change of $e = (p, q)$. As a conclusion, we can only consider the edges which come from a vertex that is not in subtree rooted at q and ends within this subtree.

Then after finding all the edges possessing this property by traversing all edges in $O(|V|+|E|)$, we know that one of these edges would be contained in the new shortest-path tree. Therefore, suppose this edge as $e' = (u, v)$, and suppose the new graph as G'. In this case, for this edge to be contained in the new shortest-path tree, we would want the following equation, $d_G(s, v) \geq d_{G'}(s, u) + w(u, v) + 1$ so that the original shortest-path tree would not be valid since there exists a shorter shortest-path tree. And by traversing all such edges to find the minimum increased amount to ensure such situation would be enough. Therefore, the correctness of this algorithm is proved.

Since this algorithm only use BFS once to find all the edges whose start is outside the subtree rooted at q and ends within this subtree and then iterative check every edge like this, the total running time of this algorithm is $O(|V| + |E| + |E|) = O(m + n)$, which takes linear time.

2 Problem 29:

Solution 29:

Suppose we are given a sequence of the positions $x_1, x_2, ...x_n$ and r. And the smallest number of stations needed to plant is defined by the following function

Greedy Algorithm:

Algorithm 2 Find the smallest number of base stations algorithm

Input : $x_1, x_2...x_n, r$
Output: The smallest number of base stations

```
Function minNumber(x_i, x_{i+1}...x_n, r):
    currentPosition ⟵ x_i
    Plant a base station at currentPosition + r
    if currentPosition + 2r > x_n then
        return 0
    While x_i ≤ currentPosition + 2r, increment i
    return 1 + minNumber(x_i, x_{i+1}, ...x_n, r)
```

The result we want is minNumber$(x_1, x_2...x_n, r)$. Since we are traversing the whole array of positions x and each position will only be visited once, the running time of this algorithm is $O(n)$.

Proof:

We will show that there is an optimal solution that contains the position of the leftmost base station this greedy algorithm chose. Suppose this position is $y_1 = x_1 + r$. Then suppose OPT is any optimal solution given the requirements. Let k_1 be the leftmost position of this solution.

- If $k_1 = y_1$, then the proof is complete.
- If $k_1 > y_1$, i.e., k_1 is on the right of y_1. By the definition of our greedy algorithm, y_1 would be the rightmost position that contains x_1. Therefore, if k_1 is on the right of y_1, there is no way that k_1 can cover x_1. In this case, either k_1 is not the leftmost position of base station or OPT is not an optimal solution, which both are contradiction to the assumptions.
- If $k_1 < y_1$, i.e., k_1 is on the left of y_1. Since x_1 is the leftmost position that we need to cover by the base station. Therefore, any customers k_1 covers, they are also covered by y_1. In other words, we can replace k_1 with y_1 in this solution without the loss of optimality. Thus, if OPT is an optimal solution, then $(\text{OPT} \setminus k_1) \cup y_1$ is also an optimal solution.

Then we will prove by induction on the number of customers that this greedy algorithm returns an optimal solution.

- *Base Case:* When there is only one customer, we would need one base station, then it is trivially true.
- *Inductive hypothesis:* Suppose for $n \leq k-1$ customers, $k \in \mathbb{N}$, the recursive greedy algorithm returns an optimal solution.
- *Inductive step:* Now let $x_1, x_2, ...x_n, r$ be given. For $x_k, ...x_n$, where x_k is the smallest position that $x_k > x_1 + 2r$, by the inductive hypothesis, the greedy algorithm gives an optimal solution for this sequence, i.e, currently the leftmost position of the base stations is $x_k + r$. In this case, the minimum number of base stations to cover all customers $x_1...x_{k-1}$ is one, at $x_1 + r$ since $x_1 \leq x_k \leq x_1 + 2r$. Therefore when there are n customers, the algorithm returns an optimal solution.

3 Problem 30:

Solution 30.A:

Since all edge weights are different, we will prove that the "local-optimum tree" is the unique of this graph G. Let the MST T of G be given and the "local-optimum tree" be T'.

- If T = T', then the proof is complete.
- Suppose T and T' have at least one edge different. Let $e = (u, v)$ be an edge in T and not in T'. By the definition of local optimum tree, for the path $p_{T'}(u, v)$, every edge on this path has lower weights than e. Therefore, we can always replace e with some edge $e' \in p_{T'}(u, v)$.
- Then we will need to prove that any edge that is not in "local-optimum tree" cannot be in any MST, that is, is unsafe. As the same before, let $e = (u, v)$ be an edge in T and not in T'. Then we consider any partition that would cause e as a cross edge in the partition, that is, partitioning the vertices into S and $V \setminus S$ and $e = (u, v), u \in S, v \in V \setminus S$. By the result of e not in T', there exists another edge $e'' \in p_{T'}(u, v)$ that is also an edge crossing the partition. Therefore, with the same reasoning, $w(e'') < w(e)$. Therefore, for any partition with e as a crossing edge, e does not have the minimum weight of all crossing edges. Therefore, e is not a safe edge and by the definition of MST, $e \notin$ any MST. Thus, this is a contradiction with the assumption that e is in T and not in T'.

Therefore, as a result, we proved that T' is a more optimal solution than any given "MST" and any edge that is not in T' cannot be in MST. Hence, the "local-optimum tree" is an MST. And with different edge weights, it is the unique MST.

Solution 30.B:

By the ***Boruvka's algorithm***, each time we find the smallest edge weights between two connected components in a n-vertices graph, we reduce the calculation of the next step to $O(\log n)$ since each iteration of Boruvka, in the worst case, would reduce the number of connected components to one half. Therefore, we can use ***Boruvka's algorithm*** with $\log \log n$ iterations to calculate the connected components after these iterations. Since each iteration of Boruvka requires $O(m)$ complexity, the running time of this step would be $O(m \log \log n)$.

Then we can simply run DFS on the graph after $\log \log n$ iterations of ***Boruvka's algorithm*** to compute all the connected components, which takes $O(m + n)$ time to finish.

After getting the calculated components with $\log \log n$ iterations, we can construct a new graph that consider each components as a single vertex and insert an edge between connected component c_1 and c_2 if there exists between any vertex in these two components, and the weight of this edge is the minimum of all edges that originate from c_1 and end in c_2. On the new constructed graph, we run ***Prim's algorithm*** to find the minimal spanning tree. Since each iteration of ***Boruvka's algorithm*** would reduce the number of connected components to $\log n$, after $\log \log n$ iterations, the number of connected components, i.e., the number of vertices in the new graph, would be $\frac{n}{2^{\log \log n}} = \frac{n}{\log \log 2^n} = \frac{n}{\log n}$. And the number of edges in the new graph would still be $O(m)$. Therefore, running ***Prim's algorithm*** on this new graph requires $O(|E| + |V| \log |V|) = O(m + \frac{n}{\log n} \log \frac{n}{\log n}) = O(m + \frac{n}{\log n}(\log n - \log \log n)) = O(m + n)$.

Therefore, the total running time of this algorithm is $O(m \log n \log n)$.