CSCI 570 Spring 2020 Xinrui Ying (USCID 1092350440)
Homework 2 Problem

Solution:

1. We initialize result as empty and the greedy algorithm is as following:

   - Run BFS to find if there exist a path from s to t, if existed, store the path as result. Else if not, we return stored result.
   - Sort all the edges in non-decreasing order of their weight and delete the set of edges with the smallest weight.
   - Repeat the above two steps.

   At most we will loop $|E|$ times and each time BFS takes $O(V+E)$. The total complexity is $O((V+E)\cdot E)$.

   Proof of correctness:
   Suppose there exists a path p' from s to t with minimum weight edge greater than the minimum weight edge in path p we derived from greedy algorithm. From our algorithm, we know that if we delete all edges in G that weight of them is $\leq$ to the minimum edge weight in p, we aren't able to find any path from s to t. This contradicts with having a path p' from s to t with minimum weight edge greater than the minimum weight edge in path p. Thus, our greedy algorithm must returns the optimal solution.

2. Prove by contradiction, suppose T and T' are both minimum spinning tree for G but they contains different edges $\implies$ weight of T is equal to weight of T'.

   Let E' be the edges that are both in T and T', let $e_i = \{s,t\}$ be the minimum weighted edge that's in T or T' that not in E'.

   Assume e_i is in T, then there must be a path from s to t in T'. Adding e_i to graph T', we will have a cycle in T' (property of tree). Deleting any edge in that cycle we will still get a spinning tree. Because edges have distinct weights, we need to delete the large weighted edge in that cycle to have the weight of the spinning tree to be small. Given T' is a minimum spinning tree, e_i must the most weighted edge in that cycle. Then, all other edges in the cycle should be in T and T' because e_i is the minimum weighted edge that are not in both T and T'. T will have e_i and all other edges in the cycle $\implies$ T has a cycle. Contradiction to T is a minimum spinning tree $\implies$ there is a unique minimum spinning tree in G.

3. Initialize an empty hashmap H to check if a node is visited, an empty hashmap H' to check if an edge is visited and a queue Q. Choosing any arbitrary node as root r, we push it to Q.

While Q isn't empty, we pop the first node in Q as v. We traverse edges that has v as one of its vertex and hasn't been visited yet (which is not in H'). Every time we traverse an edge $e_i = \{v, t\}$ that's not in H', we first push e_i to H'. Then we check if the other vertex t for that edge is in H. If it's not in the hashmap H, we simply push t to Q, and push t into H. Else, we find v's and t's least common ancestor x. We choose the most weighted edge in path v to x, t to x and edge e_i, and we delete the most weighted edge.

Popping from Queue take $O(1)$ and we iterate n times. Checking if it's in the hashmap or insert it into hashmap takes $O(1)$, we are going to do that $O(n)$ times as number of edges is $O(n)$. If we detected cycle, finding least common ancestor take $O(n)$, and delete the most weighted edge in the cycle takes $O(1)$. However, given $n+8$ edges, we are going to detected cycle at most 9 times. So the total running time will be $O(1)*n+9*O(n) = O(n)$

4. Make two pointers, first on point to the first element in S', and second pointer point to the first element in S.

   If the first pointer passed the last character of S', we return True. Else if second pointer has passed the last character of S, we return False. While neither pointer finished traversing their string, we make the comparison if the character pointed by the first pointer is the same as the character pointed by the second pointer. If they are the same, move both pointer one step to the right. Else, move only the second pointer in S one step to the right. We go back and keep the same process.

   Proof:
   Base case: If $|S'| = 1$, from the algorithm described above, we keep moving the second pointer in S until the character matches $S'[0]$. Because $|S'| = 1$, it's same as if S contains S'. If it contains, algorithm will output true and S' is the subsequence of S. Else, algorithm will output false and S' is not the subsequence of S.

   Suppose the algorithm works for $|S'| = 1, 2, 3, .., k$, we want to prove that it also works for $|S'| = k+1$.

   Because it works for $|S'| = k$, we first use the algorithm to find if the first k characters in S' is a subsequence in S. If it cannot find the first k characters, S' is not a subsequence for S and our algorithm will return False. Else if it finds the first k characters, the problem becomes finding the last element in S' in the remaining S. This is same as the base case we proved, so our algorithm works for $|S'| = k+1$. Hence, we proved our algorithm works.

5. We initialize with two empty hashmaps H and H', H for store the time at visiting the node and H' for edge is visited. We also set s as the starting node, we insert s into H with value of 0 (the start $t = 0$). We would also implement a hashmap R for store the parent node in the path.
   The algorithm is as following:

- For all out-coming edges from s, if the edge e is not in H', we insert it into H' and put the edge into a min binary heap with value $f_e(H[s])$
- We do popMin on the heap to get the minimum weighted edge e'. Let v and v' be the nodes for e', if v and v' are in H, we repeat the popMin process; else, we assume v is in H and v' is not, we change s to v', insert v' into R with value v, and insert v' into H with value $f_{e'}(H[v])$.
- We check if v' is the destination, if v' is the destination, we find the previous path through R and return the path;else, we continue the iteration and back to the first step.

The time complexity for building heap and popMin in total will be at most $O(E \log E)$, hashmap insert and check will take $O(1)$ so in total $O(V+E)$. Retrieve path will take linear time. Therefore, the total time complexity will be $O(E \log E)$.

6. We reverse the direction of all edges, use $W(e)$ as weight function for edge e and create a hashmap H for store weight of path as value ($H[v]$ is the total weight of choosen path from t to v). We create an empty path result first.

   First, We find $d(s,t)$, then start from t, we traverse all outgoing edges. For each edge, if $W(e_i = (t,v)) + d(s,v) = d(s,t)$, we store the path $t->v$ in result and store the sum of weight from $t->v$ in hashmap $H[t]$, then we continue on v.

   For all inter median steps, we start from a node v, find all outgoing edges. For each edge, if $W(e_j = (v,v')) + d(s,v') + H[v] = d(s,t)$, we append the edge to the result and continue iteration on v'.

   Once we reached s, we return the reverse of result. The total running time for reverse is $O(E)$ and running time for traverse is $O(E+V)$.

7. Greedy algorithm is as following:

   - From USC or any gas station location, if we can use the remaining fuel to arrive at the next gas station or destination, we choose to drive instead of fueling. Otherwise, we go to the gas station and fill up gas.

   Proof of correctness:
   Let our algorithm return a sequence of g with size n, suppose there exist a sequence g' has size $< n$. Let $g[i] = g'[i]$ for all $i < k$, we look at the index k where $g[k] \neq g'[k]$. Because our algorithm makes sure that $g[k]$ will be the farthest gas station it will make $\implies d_{g'[k]} < d_{g[k]}$. Similarly, let the size of g' be m, then $d_{g'[m]} < d_{g[m]}$. We know that $g[m]$ cannot make it to the destination because the car has to stop at $g[i]$ $\forall i <= n$ based on our greedy algorithm. Thus, $g'[m]$ cannot make it to the destination. Therefore, our algorithm outputs the optimal solution.

8. First we check if the removed edge e is in T, if it's not in T, we return T as the minimum spinning tree for G_1. Else, we remove the edge e in T, then we a hashmap H. After removing the edge, T has two disconnected components. Label two components a and b, traverse all nodes in the T, if nodes is in a component, we push it into H with value a; else we push it into H with value b.

   We loop through all edges in G_1, only keep the edge that has one node in a and the other in b. We find the minimum weighted edge that satisfied this requirement. We add this edge to T and return T as minimum spinning tree for G_1.

   The total running time will be looping through all edges and nodes, which takes $O(V+E) \implies$ linear.

9.
   - $T(n) = 9T(\frac{n}{3}) + n + \log n$
     By Master Theorem, $a = 9$ and $b = 3 \implies c = 2 \geq 1$. Satisfy the case 1 condition, $T(n) = \Theta(n^2)$

   - $T(n) = 3T(\frac{n}{4}) + n\log n$
     By Master Theorem, $a = 3$ and $b = 4 \implies c = \log_4 3 \leq 1$. Satisfy the case 3 condition, $T(n) = \Theta(n \log n)$

   - $T(n) = 8T(\frac{n}{2}) + (n+1)^2 - 10n$
     By Master Theorem, $a = 8$ and $b = 2 \implies c = \log_2 8 = 3 \geq 2$. Satisfy the case 1 condition, $T(n) = \Theta(n^3)$

   - $T(n) = T(\frac{2n}{3}) + 1$
     By Master Theorem, $a = 1$ and $b = 3/2 \implies c = 0 = 0$. Satisfy the case 2 condition, $T(n) = \Theta(\log n)$

   - $T(n) = \sqrt{7}T(\frac{n}{2}) + n^{\sqrt{3}}$
     By Master Theorem, $a = \sqrt{7}$ and $b = 2 \implies c = \log_2 \sqrt{7} \leq \sqrt{3}$. Satisfy the case 3 condition, $T(n) = \Theta(n^{\sqrt{3}})$

10. $T(n) = 7T(\frac{n}{2}) + n^2$ is:

$$\sum_{i=0}^{\log_2 n} 7^i (\frac{n}{2^i})^2 = n^2 \sum_{i=0}^{\log_2 n} (\frac{7^i}{4^i}) = \frac{n^{\log_2 7}}{n^2} \cdot n^2 = \theta(n^{\log_2 7})$$

    $T'(n) = aT'(\frac{n}{4}) + n^2 \log n$ is:
    $\sum_{i=0}^{\log_4 n} a^i \cdot (\frac{n}{4^i})^2 \cdot \log(\frac{n}{4^i}) = \sum_{i=0}^{\log_4 n} a^i \cdot \frac{n^2}{16^i} \cdot (\log n - \log 4^i) = \sum_{i=0}^{\log_4 n} a^i \cdot \frac{n^2}{16^i} \cdot \log n - i \cdot a^i \cdot \frac{n^2}{16^i} \cdot \log 4 = n^{\log_4 a} \cdot \log n - n^{\log_4 a} \cdot \log_4 n + n^{\log_4 a} = \theta(n^{\log_4 a})$

    To have $\theta(n^{\log_4 a}) < \theta(n^{\log_2 7})$, a can at most be no greater than 49 that $T'(n)$ is asymptotically faster than $T(n)$

■