Name: Yan, Zi
Course: CIS 502
Assignment: HW8

Question 1

The algorithm is for each node v and its left child v' and right child v'', if $l_{(v,v')} \neq l_{(v,v'')}$, then adjust the lengths of two edges such that let $l_{(v,v')} = l_{(v,v'')} = \max\{l_{(v,v')}, l_{(v,v'')}\}$. After we visit each node by using BFS and adjust corresponding edges, the resulting binary tree has zero skew and the total edge length is as small as possible.

Observation 1: For any subtree rooted at v with all its descendent leaves, x_1 to x_k, after we make every path from v to x_i, where $i \in [1..k]$, equal, there always is a path from v to a leaf x_m, none of whose edges will be increased, for the case that the total length is as small as possible,

Proof. Suppose not. Let l_i denote the length of the path from v to x_i, where $i \in [1..k]$. Suppose we add $\Delta_i > 0$ to each path from v to x_i, such that all $l_i + \Delta_i$ are equal and the total length is as small as possible. But we can find the smallest Δ_m and subtract Δ_m from each Δ_i, such that for each path from v to x_i, $(l_i + \Delta_i - \Delta_m)$ is still equal to each other and get the a smaller total length. Additionally, because the length of the path from v to x_m is not changed, therefore no edges along the path from v to x_m is increased. □

Proof. Suppose not. Then there are other solutions better than the algorithm provided. Let v be any node that do not follow the above algorithm. There are three alternate solutions that can be better,

1) we let $l_{(v,v')} = l_{(v,v'')} = l' > \max\{l_{(v,v')}, l_{(v,v'')}\}$. Then at node v, all paths from v are increased, which contradicts the observation 1.
2) we let $l_{(v,v')} > l_{(v,v'')}$ after we change the lengths of the corresponding edges. Therefore, the length of each path from v'' to a leaf has to be increased by $l_{(v,v')} - l_{(v,v'')}$, in order to keep zero skew from root to leaves. Then all paths from v'' are increased, which contradicts the observation 1.
3) we let $l_{(v,v')} < l_{(v,v'')}$ after we change the lengths of the corresponding edges. Symmetrically, we will get a contradiction at node v'.

In sum, any alternate solution is no better than the provided one. So the provided algorithm is optimal. □

The BFS takes $O(n)$ for the tree traverse, and at each node the operation is constant time, therefore, the total runtime is $O(n)$.

Question 2

First, we make all weights of X-edges larger than Y-edges, and use Prim's algorithm to find a minimum spanning T_1 with least number of X-edges x_{min}. Second, we make all weights of X-edges smaller than Y-edges, and use Prim's algorithm to find a minimum spanning tree T_2 with largest number of X-edges x_{max}.

If $k < x_{min}$ or $k > x_{max}$, we can conclude that no such tree that with exactly k edges labeled X exists. If $k = x_{min} = x_{max}$, then the minimum spanning tree produced above is the spanning tree with exactly k edges labeled X. For the case that $x_{min} < k < x_{max}$, we need to find a tree with exactly k edges labeled X by swapping some edges of the minimum spanning tree, T_1 or T_2 produced above. The swapping method is described below.

We choose to start from the spanning tree T_1 with x_{min} edges. First, we pick a edge $e \in T_2 - T_1$ and the graph $T_1 \cup \{e\}$ contains a cycle C. Then the cycle must contain a edge $e' \notin T_2$. We form a new graph $T_1' = T_1 \cup \{e\} - \{e'\}$ and T_1' has at least one edge belongs to T_2. Because T_1 can have at most $n-1$ edges different from T_2, namely T_1 and T_2 have no common edge, therefore, after at most $n-1$ step mentioned above, T_1 will become T_2, and the number of X-edges in the spanning tree will be increased from x_{min} to x_{max}. In the process, each time only one edge is swapped, therefore the number of X-edges in the spanning tree will increase at most 1. As a result, we can get a exactly k X-edges spanning tree during the swapping process.

The minimum spanning tree algorithm will take $O(m \log n)$. Each swapping will take $O(m)$, and the whole swapping will take $O(mn)$. Therefore the total runtime will be $O(mn)$

Question 3

a) First, the general Resource Reservation Problem is NP, because, given any allocation schema, by checking each process whether its request is satisfied, we can tell whether there are k active processes.

Second, we use independent set problem, and if we can show Independent Set $\leq_P$ Resource Reservation Problem, then we can conclude that the general Resource Reservation Problem is NP-hard.

Given any arbitrary instance of independent set problem, where the graph is $G = (V, E)$ and the size of independent set is k, we will first construct an equivalent instance of resource reservation problem. We let each node $v \in V$ denote one process, and there is an edge (u, v) if process u and process v both request at least one common resource. We can find a k-sized independent set in G, if and only if the corresponding nodes are the active processes in resource reservation problem.

Proof. First, we prove "$\Rightarrow$". When we have a k-sized independent set, we have k nodes are independent, namely k processes request no common resources. Therefore, these k processes can be active.

Second, we prove "$\Leftarrow$". When we have k active processes, these processes will request no common resources at all. Then, in the corresponding graph G, these k nodes will have no edge between any two of them. Therefore, these k nodes form a k-sized independent set. $\square$

Hereby, we have resource reservation problem is both NP and NP-hard, then we can conclude that resource reservation problem is NP-complete.

b) We can simply check every pair of processes to see whether their requested resources are disjoint, and this takes $O(n^2)$.

c) It is a bipartite problem. One type of resources, or say people, is denoted as one set of nodes, and the other type, or say equipment, is denoted as the other set, and there is an edge (u, v) when a process requests people u and equipment v. We just need to find k matchings.

d) It is still an NP-complete problem. Because it is a special case of a) above, when an IS problem is reduce to it, there at most two edges adjacent to a single node, and we still need to find k-sized independent set.

Question 4

We first show this problem is NP. Given a set of requests $P_1, P_2, ..., P_c$, we examine each pair of these requests, in $O(c^2)$, we can tell whether k paths exist shared no common nodes.

Then, we need to show it is NP-hard by proving independent set $\leq_P$ path selection problem. Given any arbitrary instance of independent set problem, where the graph is $G' = (V', E')$ and the size of independent set is k, we need to construct an equivalent instance of path selection problem. We let each node $v_i \in V'$ denote a path P_i, and there is an edge (v_i, v_j) if path P_i and path P_j share at least one common node. We can find a k-sized independent set in G' if and only if there are k paths in G share no common node.

Proof. "$\Rightarrow$". We have a k-sized independent set, namely there are k nodes that are independent in G'. Therefore, the paths denoted by these nodes share no common node in G, according to the definition of edge in G'.

"$\Leftarrow$". We have k paths in G which share no common node, therefore in G', the corresponding k nodes do not have any edge between any pair, namely a k-sized independent set. $\square$

Hereby, we have path selection problem is both NP and NP-hard, so we say path selection problem is NP-complete.