500509944

COMP2123 Assignment 4

1. We can prove the correctness of this algorithm by first showing that our algorithm maintains the invariant that **T** is always a spanning binary tree (SBT) for a number of vertices unless it is not possible. Now initially, **T** is empty, and our invariant trivially holds. Assuming that **T** is a SBT for n iterations/vertices, we can observe that our invariant still holds on its n+1th iteration, as we will add a new vertex to **T** unless there are no edges present in **S**, in which case we would violate the binary constraint and we return **infeasible.**

   Now we can prove that if such a tree exists, our algorithm finds the one of minimum weight. Suppose that our algorithm returns a tree **T** that is not the minimum weight SBT (MWSBT). Let **M** be the actual MWSBT and **(u, v)** be the first edge in **T** that is not in **M**. Since T is not the MWSBT, there must exist an alternate path from u to v (P) in **G** which has a smaller weight than (u, v). Now there must exist an edge (x, y) in P such that x is S and y is in R since it wasn't chosen. However, in the iteration of our algorithm where we consider (x, y), It would have been chosen since it has a smaller weight than (u, v). This contradicts our initial assumption that (u, v) is the first edge in T not in M and therefore M cannot exist and therefore our algorithm correctly returns the MWSBT.

2. a) We first create an empty priority queue of size mkl , a distance array **dist** of size mkl to store the minimum cost to reach each stop and an array **prev** of size mkl to store the previous stop and instance for each stop and instance. Now we initialize all elements in **dis** to **infinity** except for the origin stop **a**, which is set to 0 and all elements in **prev** to **null**. First, we insert all buses that stop at origin stop **a** with a departure time at or after **t** into the priority queue with their respective costs . While the priority queue is not empty, we do the following:

   - Extract the stop **(cur)** with the minimum cost from the priority queue.
   - For each neighbouring stop **(next)** reachable from the same bus line, calculate the cost which is the cost of **dist[cur].** If the cost is lower than the current cost stored in **dist[next]** and the arrival time at **next** is before or at the **t'**, update **dist[next]** with the new cost and enqueue next into our priority queue. Set **prev[next]** to **cur** to track the previous stop and instance.
   - Repeat the previous step but for each neighbouring stop reachable from a different bus line, where we calculate cost by adding **dist[cur]** with the cost of **next**.

   If the loop terminates and the value at **dist[b]** is still infinity, then we return an empty array as it is not feasible.

Else, we start by initializing a list-based stack **reverse** of size l. While our destination stop **b** is not the origin stop **a**, we push **b** to the list and set b to the previous stop by accessing **prev[b]**. Finally initialize another stack **res** of size l and first push the origin stop **a .** Then we push the elements of **reverse** in reverse and return **res.**

b) To prove the algorithms correctness, we can use the fact that our algorithm utilizes Dijkstra's Algorithm by treating the timetable as a directed graph, where each node represents the triplet of form (stop, arrival, departure) and the edge weights are the fares between stops. Since the fares are non-negative, our algorithm can be applied to find the cheapest path (since our edges are costs) from the starting stop-time pair to the destination stop-time pair.

c) Our algorithm is dominated by Dijkstra's Algorithm which means it takes **mkl**(log(**mkl**)) time.

3. a) We can solve this problem utilizing a greedy algorithm with backtracking. We initialize two variables, one to represent the minimum cost (**minCost**) which is initially infinity and a list-based implementation of a stack (**traversals**) of size **n**. We define a recursive function (**search)** that takes arguments for the current shape (**shape)**, a stack of traversals (**curTraversals**), the current cost (**cur**) and its current depth (**depth**) as follows:

   - If shape is **B**, we compare **cur** with **minCost**. If it is smaller, we update **minCost** to **cur**, set **traversal** to **curTraversals** and return.
   - If **depth** is equal to **n**, then we return as we have exhausted all shapes for this current traversal.
   - Else, we iterate over every shape (**nextShape**) expect **shape** in our collection of shapes and do the following:

     - Check if **nextShape** overlaps with **shape** and if **shape** is a square, **nextShape** is not a square.
     - If both of these conditions are met, then we calculate the area of **nextShape**, add it to **cur** and push **nextShape** to **curTraversals**.
     - We recursively call search(**nextShape**, **curTraversals**, **cur**, **depth** + 1)
     - Finally, we pop **nextShape** from **curTraversals** for backtracking purposes.

   We would call search with parameters **A**, an empty stack, the area of **A** and 0. Then return **traversals** as the solution.

   b) We can argue the correctness of this algorithm by maintaining an invariant that our recursive function search correctly explores all valid traversals from **nextShape** to **B** considering only the remaining **n - depth** shapes. This holds for our base case where if the current shape is **B**, we update **minCost** and **traversals** if necessary. Assuming that our invariant holds up to the (**depth-1)**th shape, we can observe that it also holds for the (**depth)**th shape, as we consider all valid shapes within the given

constraints and since we assume that the invariant holds for the (**depth-1**)th recursive call, we can conclude that during the (**depth-1**)th call, all valid partial traversals from the (**depth-1**)th shape to shape **B** were correctly explored. Our algorithm is also guaranteed to terminate as we either terminate when the current shape is **B**, or if **depth** is equal to **n.**

c) Our algorithm takes O(n^2) time since for shape, we consider all other shapes and make recursive calls to explore the traversal.