Algorithm Assignment – 7

1(a)

This problem of MBTA bus routing application can be represented as a graph where each stop can be represented as nodes in the graph and the route taken by the bus as edges of the graph. Weight of each edge will be the time taken by a bus to reach the stop from the previous stop.

The time taken by walk is found by third party application. After reaching the bus stop, Our application needs to provide the fastest way to reach the destination by using MBTA bus.

In order to solve this problem, we use Dijkstra's algorithm which is used to find the shortest route from source node to the destination node.

While calculating the shortest distance from the source node to the destination, we need to take in to consideration the

(i) waiting period of the customer.
(ii) switch between the buses in the transit point if it will help us to reach the destination faster.

We need to compute the earliest time we can reach a stop, starting from the nearest stop to the "from" address given.

Graph formed will have stops represented as nodes in the graph. We can maintain an edge for path taken by bus. Edges here are directed which shows the direction in which bus travels. If two paths have same stops as their next point of travel, then we can have a common edge.

Each edge is associated with a path cost which represents the time taken by the bus to reach from one stop to another, this can be fetched using an third party application which gives the data based on the current traffic. This value can be dynamic.

Path cost also includes the waiting period of the customer, which is
Equal to (Departure-time of the bus – Arrival time of the customer)

Each Node i.e. stop will hold information of all the buses departing from that stop i.e. both their time of arrival and their time of departure.

While solving this problem we will have to consider only the buses whose departure time is on or after the customer's arrival time.

We can make the edges of all the bus routes whose departure time is after the customers time as an active edges of graph, This will improve the efficiency of the algorithm

(b) Technical problem here is to find single source single destination shortest path, Given a graph with nodes, edges, source node and destination node information.

In this algorithm, Stops form the nodes of the graph, edges are the path taken by a bus from one stop to another, weights on the edges are the path cost which includes the waiting time and the time taken to reach a stop from another stop by bus based on the current traffic condition.

(c)
In order to solve this in an efficient way, we use Dijkstra's algorithma greedy algorithm to find the shortest or the fastest route possible from source node to destination.

Following data needs to be maintained to solve this problem:

1. Waiting time of the customer at each stop.
   Waiting Time = (Departure time of the bus – Arrival time of the customer)

At every stop we need to keep track of the time the customer reached a given stop. Waiting time is computed for all the buses

Additional context variable "prev" is maintained to keep track of the stop from which the customer reached the current stop with least total travel duration and also the bus details.

Algorithm:

Input: Graph G which consists of stops and edges formed between stops if there is one or more buses travelling in that direction.
Source which will be starting point of travel
Destination to be reached, Departure time of the customer.

Ouptut: Array **prev** which keeps track of the previous stop in the optimal

Path, Array **time** which keeps track of the time taken to reach a stop, Array **bus** which keeps track of the bus that needs to be travelled in, to reach each stop at earliest.

```
FindShortestPath( G, source,  departureTime)
{
    // Let Q be a minimum priority queue which holds all stops in graph G
    S.T the stop with minimum travel time is on top of the Queue

    // Adding all the stops to a minimum priority Queue Q
    For all stops  s in Graph G
        time[s] = INF  // total time taken to reach  stop s from source stop
        prev [s] is set to  undefined // Previous stop before reaching stop s
        bus[s] = undefined // bus taken to reach stop s from source stop
        add s to Q

    // To start from source node
    set time[source] to 0

    while Q is not empty

        // stop in minimum priority queue with least total reach time
        x= stop with minimum reach time in Q
        For all the stops s connected to x

            // consider the buses whose departure  time is within 24
            //hours of customer arrival time
            For all the buses b in s


                // compute waiting period of the customer for all the
                //buses
                waitingTime = (b.departureTime -
                                    x.arrivalTmeOfCustomer)
                totalTime = time[x] + time(x, s)+ waitingTime
                if( time[s] > totalTime)
                {
                    time[s] = totalTime
                    prev[s]  = x
                    bus[s] = b
                    s.arrivalTmeOfCustomer  = totalTime
```

```
                    }

        Return time[], prev[], bus[]

}
```

We can print the itinerary starting to back track from the destination stop using prev[], time[] and bus[] as shown below:

```
// This prints itinerary in reverse order

Let stop = prev[destination]
While (stop != source)
{
        Print ( bus[stop].busNo )  // Prints /bus number
        Print ( bus[stop].departureTime ) // Prints departure time of the bus
        stop = prev[stop]
}
Print (bus[source].busNo)
Print( bus[source].departureTime)
```

(d)

Time complexity of Djikstra's algorithm depends on the implementation of minimum priority Queue. Time taken for creating a min priority queue is O(v), DecreasingKey or IncreasingKey will be O(log V) and extracting minimum will be O(log v). This will result total time taken to be **(|E|* |V| log v)** for all edges E. Since all the stops are reachable by the source node time complexity will be O**(|E| log v)**

If V is the number of stops and E is the numbers of edges established between stops and B is the total number of buses that depart from a stop then the time taken for finding the fastest route from source to destination is:

T(n) = O(|E|*|B| log v)

(e) The above algorithm uses Dijkstra's algorithm with few modifications to consider all the buses that travel from a given stop. This ensures that we are Considering even bus switching to reach the destination at the earliest possible.

Waiting time is also considered while calculating time taken to reach a stop from source. All buses that depart from a stop are considered whose waiting time < 24 hours after customer reaches the stop.

Keeping track of the previous stop and the bus that we traversed in will help us print the itinerary by back tracking.

Algorithm 7.2>

a> A minimum spanning tree is a tree in which all the nodes in the graph are connected by (V-1) edges.

In order to convert a given minimum spanning tree to another minimum spanning tree, we need to consider following cases.

(i) An edge which does not exist in new MST (T) but exists in MST B i.e. (B -T)

(ii) An edge which does not exists in the minimum spanning tree B but exists in T (T- B)

Adding an edge to MST will create a cycle. So we need to find an edge which is not part of new MST but part for from the old MST which results in cycle creation when new edge is added and move it to the new edge position in MST B.

In order to accomplish this, we can a do a depth first search starting from one of the vertices of the newly added edge and find all the edges which is in search path, by using recursion stack and remove the first edge which needs to be deleted.

We can identify all the common edges between the two MST's by making use of hash map "mapB". For all the edge in B, create a mapping entry with edge (u, v) as a key and 1 as value.

Now loop through all the edge (u, v) in T and see if there exists a key in Hashmap "mapB" with entry (u, v). If so then add this edge to the HashMap "commonMap" which keeps track of all the common edges between B and T with value set to 1.

Steps to convert from B to T where B is the old minimum spanning tree and T is the new minimum spanning tree:

Algorithm:

Given : Two Minimum spanning trees B and T

ConvertMST(B, T):

Step1 :

Identify all the common edges between B and T, new edges that need to be added to T, old edges in B that needs to be replaced by new ones.

Let V is the number of nodes in the MST and E is the No of edges in B and V' and E' be the number of nodes and edges in T

Let mapB be an hashmap which holds all the edges in MST B

```
for  each edge (u, v) in E
{
        mapB.put((u,v),1)
}
```

Let commonMap be an hashmap which keeps track of all the common edges in B and T

Let newEdgeArr arrayList which keep track of all the edges that needs to be added to the MST

```
for each edge (u, v) in E'
{

        if(map.containsKey((u,v))
        {
                commonMap.put((u,v), 1)
        }
        else
        {
                newEdgeArr.add(u,v)
        }

}
```

All the edges in B which do not exist in the MST T can be maintained using an HashMap deletedEdgesMap.

```
for each edge in (u,v) in B
{
        if (commonMap does not contain (u,v))
        {
                deletedEdgesMap.put((u,v),1)
        }
}
```

Step 2 :

Now to move an old edge (x,y) to the new edge (u,v) place, we need to do a depth first traversal starting from the node u and search for node v. Keep track of all the edges that you come along this path.

Find the first edge (x,y) which is part of deletedEdgesMap from the edges obtained from the above step. Move this edge (x,y) to (u,v). This ensures there is no cycle formed while moving the edges and ensures that the intermediate trees will remain as spanning tree during this conversion.

Repeat this process for all the new edges that are supposed to be added. Once we are done with all the all the edges, we have a new minimum spanning tree obtained in minimum number of steps.

Let recusrionStack keep track of all the edges we come across during the DFS traversal

```
for ( (u,v) in newEdgesArr)
```

Search the minimum spanning tree B for the node v starting from u using DFS.

The traversal path which involves the vertex v will result in recursion Stack with all the edges in that path, by adding edges to recursion stack during traversal.

```
For (x,y) in recursionSatck
        If deletedEdgeMap.contains((x, y))
                move(x,y) to (u,v) in B
                break;
```

If E and V are the number of edges and vertices in B respectively and E' and V' are the number of edges and vertices in T respectively. This algorithm involves traversing all the edges in B and T to find common edges, deleted edges and the new edges. This will take O(E) and O(E') time.

In the worst case if all the edges are to be removed and replaced by different edges to form new minimum spanning tree then, O(E (E+V)), where E is looping through all new edges and O(|E|+|V|) will be the time taken to traverse the graph to find the edge that needs to moved by Depth first traversal. Therefore T(n) = O(E (E+V)).

2 (b) In this algorithm, We ensure that no cycles are formed while adding new edges to the existing MST by moving an edges which is not part of the new MST i.e (edge which is part of B-T) to the new edge position (i.e. an which is part of T-B)

We keep track of all the edges that needs to be deleted from the old MST B and the new edges that need to be added. So each time we want to add an new edge(u,v) we will do it by moving an old edge in the (x,y) which needs to be deleted from (x,y) to (u,v). This ensures that the tree will remain to be spanning throughout the process of transition from B to T without any creation of cycles. This ensures that the network will remain valid for broadcast.

This algorithm will terminate once all the new edges are moved to new edge positions. Adding all the new edges will ensure B is converted to T.

2 (c)

The Messaging Quality will go down during this transition period as the spanning tree formed by moving an old edge to the new edge position will not be minimum spanning tree at all points of the transitions.

We can ensure that the quality is not hampered by making changes to the algorithm, i.e. by moving the edges with the highest latency in the recursion stack which needs to deleted, to the new edge position.

This will ensure that the quality of the message is maintained during transition from B to T.