Data Structures Final Project Report

Achita Chitraphan [曾裕興]

111006213

Chapter 1: Problem 1

1.1 Approach to Problem 1

1.1.1 Insert ()

In this problem we're tasked with creating a multi-cast tree to all given destination nodes, which are all nodes in a connected undirected graph. In this problem, I noticed that the requirement of the task can be dealt with using Minimum Spanning Tree algorithms, which led to two options in my case, either using Kruskal or Prim's algorithm [1]. In the end the latter algorithm is chosen due to the anticipation that the graph will probably be dense thus Prim's algorithm may come out to be more efficient. Furthermore, we can employ min-heap priority queue which will sort the edges which have the least cost to the front of the queue [2]. To implement the minimum spanning tree algorithm for the insert () function, I first created a duplicate graph and forest in the structure of Problem 1. During the constructor call, we will then assign our duplicate graph structure to be that of the graph passed into the constructor call. This is done so that we can also have a copy of the current graph and keep track of the changes in real time. The data structure inside the Graph structure contains two vectors, one vector is for maintaining the total vertices while the other contains the graph edges.

```
Problem1::Problem1(Graph G) {
    /* Write your code here. */
    //initialize params??
    this->graph = G;
    //this->original = G;
}
```

Figure 1 *Problem1 constructor*

From there within the insert () function, we can use the duplicate graph that we created to solve the problem. First, we will create a temporary tree, this tree will be used to push the new multi-cast tree into the forest that we created in the Problem1 class. We would then assign the tree with its respective id, source node, and the starting total cost (0). We would then save its traffic cost on a separate map (from Map Standard Library [3]) which will have the id as its key and the t cost as its element. Now to begin the Prim's algorithm, we would first create a min-heap priority queue and a Boolean vector to track the visited nodes. Which can be done in the following manner.

```
Tree tree;
MTidTraffic[id] = t;
tree.ct = 0;
tree.id = id;
tree.s = s;
priority_queue<pair<int,graphEdge>, vector<pair<int,graphEdge>>, PairComparator> pq; //min-heap with prior queue.
vector<bool> inMST(D.size, false); //track visited nodes
```

Figure 2 *Priority queue and visited nodes vector*

```
struct PairComparator {
    bool operator()(const std::pair<int, graphEdge>& lhs, const std::pair<int, graphEdge>& rhs) const {
        // Compare based on the first element of the pair
        return lhs.first > rhs.first;
    }
};
```

Figure 3 *Function for comparing pairs in the priority queue*

We then iterate over the graph and push any edges which contain the source node in either its vertex[0] or vertex[1] and the remaining bandwidth is greater than or equal to our traffic cost, this way we will have all the traversable path from source node to every other nodes to select from which will be sorted starting from the least cost. From there we then assign the source node as visited in our Boolean vector and push the source into our tree's 'V' vector. We can then begin the while-loop which will be performed until the priority queue runs out of edges or the number of vectors included into the tree is the same as the size of the destination Set. During each iteration of the while-loop we will first take the top edge in the priority queue, we must then identify the source and the destination nodes of that edge before popping the edge out of the queue (the edge is traversed). We then check whether the destination node has been visited or not, if not then we can traverse to that node through our minimum cost edge and push the said vertex into our tree's 'V' vector. Likewise, we then record the edge into our tree's 'E' vector before updating the cost through adding the multiplication result between the edge's cost and the traffic cost 't'.

Once we've "travelled" to our destination node, we must then update our queue with all the traversable adjacent edges at our destination node, that is, the remaining bandwidth is greater than or equal to the traffic cost. During this loop we would then also deduct the bandwidth of the edge with the traffic cost of the multi-cast tree.

```
vector<graphEdge>::iterator it;
for(it = this->graph.E.begin(); it!=this->graph.E.end(); ++it){
    if(((*it).vertex[0] != s) && ((*it).vertex[1] != s)) continue;
    else if((((*it).vertex[0] == s) || ((*it).vertex[1] == s)) && (*it).b >= t) pq.push(make_pair((*it).ce, *it));
}

inMST[s] = true;
tree.V.push_back(s);
while(!pq.empty() && (tree.V.size() < D.size)){
    graphEdge U = pq.top().second;
    int u = (inMST[U.vertex[0]] == true)? U.vertex[0] : U.vertex[1]; //source
    int v = (inMST[U.vertex[0]] == true)? U.vertex[1] : U.vertex[0]; //destination
    pq.pop();

    if((inMST[u] == true) && (inMST[v] == true)){
        continue; //both vertice are visited.
    }
    else if(inMST[v] == false && (U.b >= t)){

        inMST[v] = true; //destination node visited
        tree.V.push_back(v);

        treeEdge edge;
        edge.vertex[0] = u;
        edge.vertex[1] = v;
        tree.E.push_back(edge);
        tree.ct += U.ce * t;
        for(it = this->graph.E.begin(); it != this->graph.E.end(); ++it)
        {
            if(((*it).vertex[0] != v) && ((*it).vertex[1] != v)) continue;
            if((((*it).vertex[0] == u) && (*it).vertex[1] == v) || (((*it).vertex[0] == v) && (*it).vertex[1] == u)){
                (*it).b -= t; //deduct remaining bandwidth
            }
            if((((*it).vertex[0] == v) || ((*it).vertex[1] == v)) && ((*it).vertex[0] != u && (*it).vertex[1] != u) && ((*it).b >= t)){
                // if one of the vertex is the current destination.
                // and the other vertex is not current source.
                // and remaining bandwidth is >= traffic cost.
                pq.push(make_pair((*it).ce, *it));
            }
        }
    }
}
```

Figure 4 *Pathfinding algorithm*

After the while-loop is terminated, we would have a tree that will be inserted into the multi-cast graph. We can then push this tree into the forest that we maintain, however, before pushing the tree into the forest, it is our task to check whether the tree to be inserted has an id greater than the last tree in the forest's 'trees' vector or not. If not then the vector should be sorted using std::sort out so that all trees' id are sorted from the least id value to the greatest using the Algorithm Standard Library (this would aid us in stop () and rearrange () later on) [4]. Once pushed, the size of the forest will be incremented accordingly, and the resulting graph and forest can be overwritten into the Graph 'G' and Forest 'MTidForest'.

The total expected time complexity of using the Prim's algorithm approach with a priority queue is $O(E \log V)$ while hosting a space complexity of $O(V + E)$ where the 'inMST' vector takes up space equal to the number of vertices while the priority queue has at most E edges in our priority queue. Using Prim's algorithm with a min-heap priority queue is a time-efficient method for implementing a Prim's algorithm. Std::Sort is a combination of quicksort, heapsort and insertion which combined into an introsort algorithm which is an unstable sorting algorithm, however, since all IDs will be unique there is no need to be concern about it. The time and space complexity with introsort is $O(n \log(n))$ and $O(\log(n))$ respectively [5].

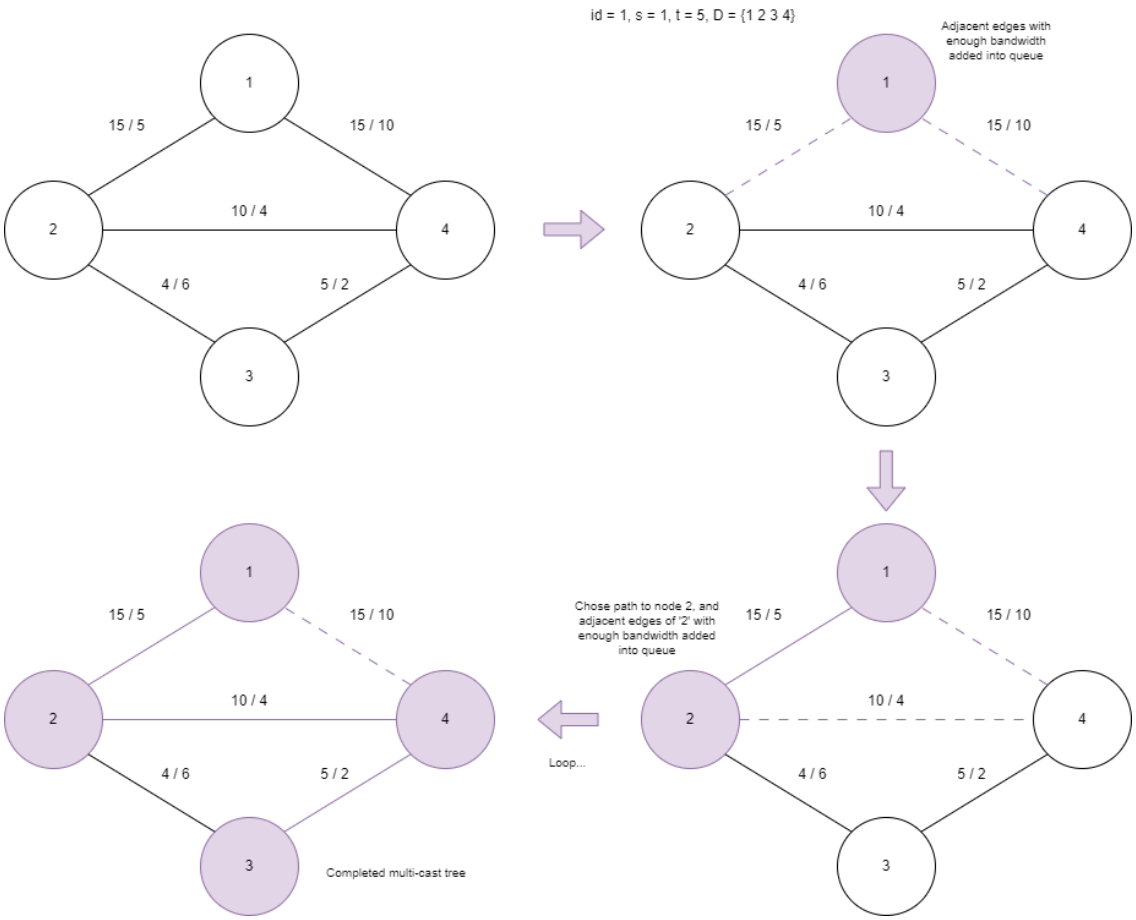


Figure 5 *A brief visualization of the steps taken*

1.1.2 Stop ()

In this function, we must first return the cost of the tree to the graph before removing the multi-cast tree from the forest (stopping broadcast). To do this, I first iterate through the forest that we kept track of, stopping the search once the id of the tree is the same as the provided id in the parameter ($O(N)$ time complexity). Once the tree is found we can access the traffic cost that we kept in a map using the tree's id as the key for searching, making sure to erase the key and its element once found (since the tree will now be stopped). We can then iterate over the graph's 'graphEdge' vector and return the traffic cost accordingly when we found edges that is contained in the Tree class's 'E' vector. Once the cost returns is

done, we then remove the tree from the forest of active trees and decrement the forest size. I then create a temporary forest to store the trees which will have added nodes during the operation of the stop () function where we must add nodes to multi-cast trees that possibly can have added nodes to them starting from the smallest ID number.

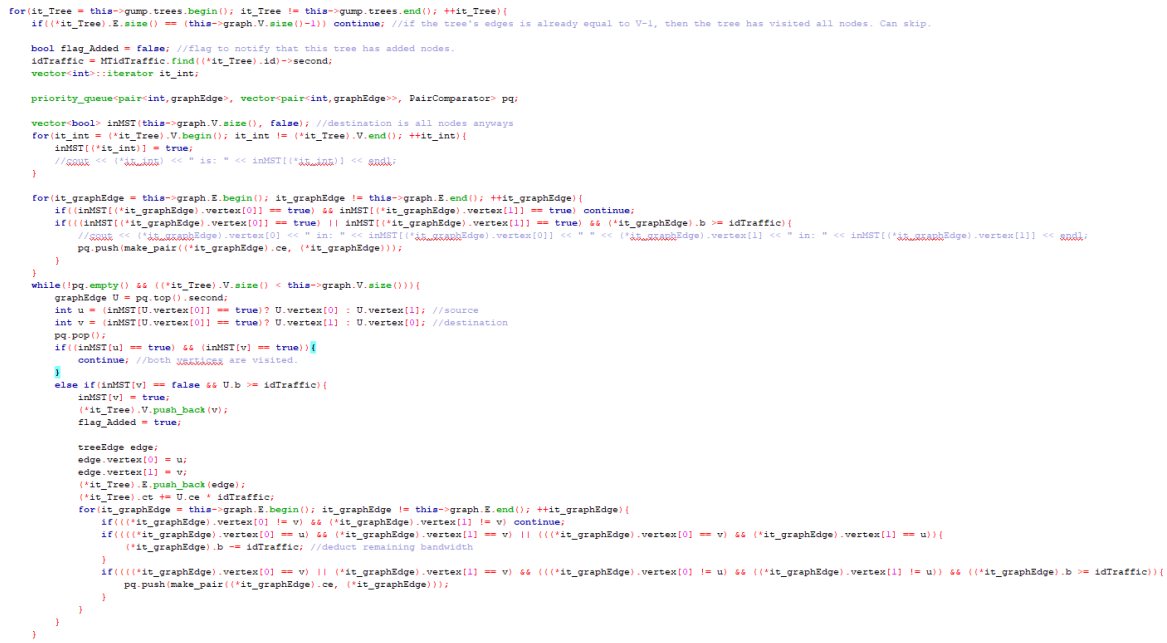
```
////////////////////////////////////
//FIND TREE & ID's TRAFFIC COST
vector<Tree>::iterator it_Tree;
for(it_Tree = gump.trees.begin(); it_Tree!=gump.trees.end(); ++it_Tree){
    if((*it_Tree).id == id)break;
}
int idTraffic = MTidTraffic.find(id)->second;
//cout << "this is t: " << idTraffic << endl;
MTidTraffic.erase(id);

////////////////////////////////////
// RETURN TRAFFIC COST
vector<treeEdge>::iterator it_treeEdge;
vector<graphEdge>::iterator it_graphEdge;
int checked = 0;
for(it_graphEdge = this->graph.E.begin(); it_graphEdge!=this->graph.E.end(); ++it_graphEdge){
    if(checked == (*it_Tree).E.size()){
        //cout << "break stop cycle: " << checked << " E.size: " << (*it_Tree).E.size() << endl;
        break;
    }
    int u = (*it_graphEdge).vertex[0];
    int v = (*it_graphEdge).vertex[1];

    for(it_treeEdge = (*it_Tree).E.begin(); it_treeEdge != (*it_Tree).E.end(); ++it_treeEdge){
        if(((*it_treeEdge).vertex[0] == u && (*it_treeEdge).vertex[1] == v) || ((*it_treeEdge).vertex[0] == v && (*it_treeEdge).vertex[1] == u)){
            (*it_graphEdge).b += idTraffic;
            //cout << (*it_treeEdge).vertex[0] << " " << (*it_treeEdge).vertex[1] << " " << (*it_graphEdge).b << endl;
            checked++;
        }
    }
}
//clear memory
(*it_Tree).V.clear();
(*it_Tree).E.clear();
this->gump.trees.erase(it_Tree);
this->gump.size--;
```

Figure 6 *Returning traffic cost to graph*

To add least additional cost routes to existing trees, we first iterate over the forest which already have all trees in its 'tree' vector **sorted** from least tree ID to the greatest ID value. Here we can first check whether the tree we are iterating on has all vectors included already by checking it 'V' vector's size compared to the total number of vertices in the graph. If the size is less than the total number of vertices existing in the graph then an attempt to route more paths will take place, otherwise continue to the next tree. Following the filter, we would create a Boolean value which will act as our flag to indicate whether the tree has been updated and must be added to the return forest once the operation is complete. We would then fetch the tree's id through map once more and similar to the insert () function, perform an MST algorithm (Prim's). However, this time instead of beginning at the source node, we would in fact include all traversed vertices into our 'visited' vector denoted as 'inMST'. We would also include all edges which is adjacent to our currently included vertices that has enough bandwidth for our tree's traffic cost to our priority queue during our initialization phase of the algorithm. Once each tree has been updated, we can either push the updated tree into the return forest or discard it if the tree does not have any additional nodes added. Finally, we output the current graph and the temp forest to Graph 'G' and Forest 'MTidForest' respectively.

In this function the estimated time complexity of returning the traffic costs to the graph would be $O(T \cdot E)$ where T represents the tree's edges and E represents the graph edges. As for the time complexity of the re-routing algorithm, since we are iterating over the graph edges and visiting all vertices the time complexity of the algorithm would be $O(E \log V)$. As for the space complexity of the function, since the

function's complexity is dependent on the numbers of trees, graph edges, and vertices/edges in the tree. The space complexity is expected to be $O(N + E + T)$ where N is the number of trees, E the number of graph edges, and T the number of tree's edges.



```
for(it_Tree = this->gump.trees.begin(); it_Tree != this->gump.trees.end(); ++it_Tree){
    if((*it_Tree).E.size() == (this->graph.V.size()-1)) continue; //if the tree's edges is already equal to V-1, then the tree has visited all nodes. Can skip.

    bool flag_Added = false; //flag to notify that this tree has added nodes.
    idTraffic = MTidTraffic.find((*it_Tree).id)->second;
    vector<int>::iterator it_int;

    priority_queue<pair<int,graphEdge>, vector<pair<int,graphEdge>>, PairComparator> pq;

    vector<bool> inMST(this->graph.V.size(), false); //destination is all nodes anyways
    for(it_int = (*it_Tree).V.begin(); it_int != (*it_Tree).V.end(); ++it_int){
        inMST[(*it_int)] = true;
        //cout << (*it_int) << " is: " << inMST[(*it_int)] << endl;
    }

    for(it_graphEdge = this->graph.E.begin(); it_graphEdge != this->graph.E.end(); ++it_graphEdge){
        if((inMST[(*it_graphEdge).vertex[0]] == true) && inMST[(*it_graphEdge).vertex[1]] == true) continue;
        if(((inMST[(*it_graphEdge).vertex[0]] == true) || inMST[(*it_graphEdge).vertex[1]] == true) && (*it_graphEdge).b >= idTraffic){
            //cout << (*it_graphEdge).vertex[0] << " in: " << inMST[(*it_graphEdge).vertex[0]] << " " << (*it_graphEdge).vertex[1] << " in: " << inMST[(*it_graphEdge).vertex[1]] << endl;
            pq.push(make_pair((*it_graphEdge).ce, (*it_graphEdge)));
        }
    }
    while(!pq.empty() && ((*it_Tree).V.size() < this->graph.V.size())){
        graphEdge U = pq.top().second;
        int u = (inMST[U.vertex[0]] == true)? U.vertex[0] : U.vertex[1]; //source
        int v = (inMST[U.vertex[0]] == true)? U.vertex[1] : U.vertex[0]; //destination
        pq.pop();
        if((inMST[u] == true) && (inMST[v] == true)){
            continue; //both vertices are visited.
        }
        else if(inMST[v] == false && U.b >= idTraffic){
            inMST[v] = true;
            (*it_Tree).V.push_back(v);
            flag_Added = true;

            treeEdge edge;
            edge.vertex[0] = u;
            edge.vertex[1] = v;
            (*it_Tree).E.push_back(edge);
            (*it_Tree).ct += U.ce * idTraffic;
            for(it_graphEdge = this->graph.E.begin(); it_graphEdge != this->graph.E.end(); ++it_graphEdge){
                if(((*it_graphEdge).vertex[0] != v) && ((*it_graphEdge).vertex[1] != v)) continue;
                if((((*it_graphEdge).vertex[0] == u) && ((*it_graphEdge).vertex[1] == v)) || (((*it_graphEdge).vertex[0] == v) && ((*it_graphEdge).vertex[1] == u))){
                    (*it_graphEdge).b -= idTraffic; //deduct remaining bandwidth
                }
                if((((*it_graphEdge).vertex[0] == v) || ((*it_graphEdge).vertex[1] == v) && (((*it_graphEdge).vertex[0] != u) && ((*it_graphEdge).vertex[1] != u)) && ((*it_graphEdge).b >= idTraffic)){
                    pq.push(make_pair((*it_graphEdge).ce, (*it_graphEdge)));
                }
            }
        }
    }
```

Figure 7 *Stop () main additional least cost pathfinding algorithm*

1.1.3 Rearrange ()

This function is in-fact quite similar to the previous insert () and stop (), however, first we must return the cost of all active trees in the forest, this can be done by resetting the graph using an extra Graph type member denoted as 'original'. The member contains the original graph structure before any trees consumed its bandwidth. By performing this operation the space complexity taken is $O(E + V)$ where E is the graph edges and V are the vertices, however, we would trade it with a time complexity of $O(1)$ instead of iterating over the graph to return its costs which would take up $O(E)$ which in my opinion is a fair trade off. Once the graph has been reset, we would then perform a similar approach to the stop () function where we will iterate over each active tree, however, this time the trees will be reset as in its total cost set to 0, 'V', and 'E' vectors will be removed using the clear function which is a part of the std::vector library so as to not use any additional spaces. Following that the Prim's algorithm will be employed and a similar path finding algorithm approach as the insert () function will be employed starting from the first tree stored in our forest (note that, the forest's trees were sorted from the least ID to the greatest from the insertion). The time and space complexity of this would then be $O(N \cdot E \log V)$ where N would be the number of active trees in the forest that we have to rearrange and $O(V + E)$. Finally, since the trees in our forest are all active, we can simply overwrite our own copy of forest onto the Forest 'MTidForest' along with our graph to Graph 'G'.

Figure 8 *Rearrange () function*

1.2 Classes for Problem 1

In the first problem the Problem1 class contains the following members which can be listed as seen below:

```
class Problem1 {
public:
    Graph graph; //duplicate graph
    Graph original; //original graph
    Forest gump; //duplicate forest
    map<int,int> MTidTraffic;
    Problem1(Graph G);  //constructor
    ~Problem1();        //destructor
    void insert(int id, int s, Set D, int t, Graph &G, Tree &MTid);
    void stop(int id, Graph &G, Forest &MTidForest);
    void rearrange(Graph &G, Forest &MTidForest);
};
```

Figure 9 *Problem1 class*

In this class, the Graph graph will be used as a duplicate graph to keep track of the changes made to the graph. The program will use this graph as a reference when performing insertion, stop, or rearrange. The next member is Forest gump, this member is used to track the changes made to the forest of the multi-cast tree and will be used as a reference for the program when accessing the trees in the forest. Both previously mentioned members will be used to provide an output graph or forest when asked. Following that, the Graph original is used to store the original unmodified graph and will be reserved for the rearrange operation where the whole graph will be reset and reinserted. The Graph graph will be assigned the values of the Graph original during rearrange (). Lastly, the map will be used to keep a dictionary of each tree's traffic using the tree's unique id as a key.

The constructor function of the class will be used to copy the values of the Graph G to our graph members. While the insert function will take the tree id we'd like to insert along with the sources, and the destination to insert a multi-cast tree and outputting the resulting graph and forest to the parameters that were passed in by reference. Similar approaches took place in the stop and rearrange functions.

Following that two additional structs were used in this problem to sort the min-heap priority queue and the forest.

```
struct PairComparator {
    bool operator()(const std::pair<int, graphEdge>& lhs, const std::pair<int, graphEdge>& rhs) const {
        // Compare based on the first element of the pair
        return lhs.first > rhs.first;
    }
};

struct TreeComparator {
    bool operator()(const Tree &lhs, const Tree &rhs) const{
        // Compare based on the tree's id.
        return lhs.id < rhs.id;
    }
};
```

Figure 10 *Structs for comparators*

Here the pair comparator struct is used to return Boolean values back to the priority queue, the struct is used as a sorting logic for the queue where if the preceding value is greater than the proceeding value, a swap will take place. For tree comparator structure, the opposite is true, a True Boolean value will be returned if the preceding tree's id has a lower value than the proceeding tree's id. The struct is used as a comparison logic for the std::sort calls.

Chapter 2: Problem 2

2.1 Approach to Problem 2

2.1.1 Insert ()

In the theory, the insert function can utilize a Steiner Tree problem [6], which have a few approaches, but in all cases the steps taken include the algorithm first find a shortest path from each node to each other in the graph, the paths would then be trimmed to exclude unnecessary path until there is only the least cost path to all nodes.

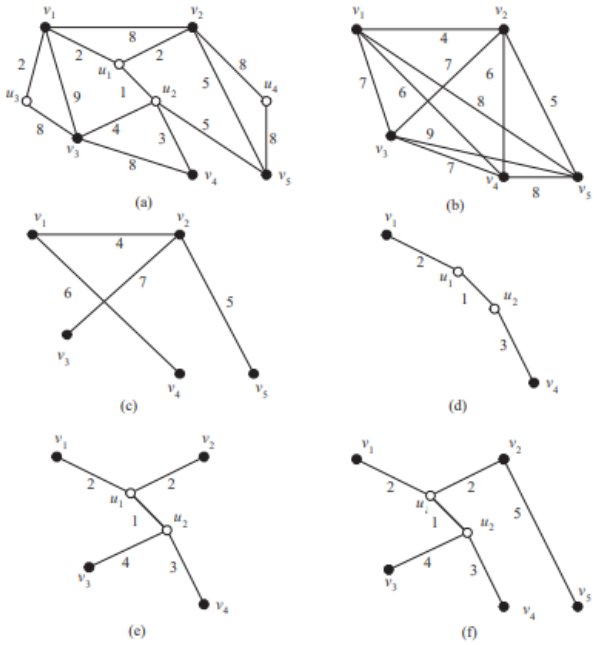


Figure 11 *An example of a Steiner Tree algorithm, MST-Steiner* [7]

However, due to the limited time, I was unable to implement similar algorithms for the tree, instead I decided to use the approach of a Prim's modified algorithm. Here, we first span the tree like a regular MST Prim's algorithm. However, in this case we're trying to minimize the tree to only reach out to the destination nodes, thus we would always check in each iteration of the Prim's algorithm whether all destination nodes have been reached. By the end of the algorithm, we would have an MST of all nodes reachable. Here, if all the destination nodes are in the resulting tree's 'V' vector, then the tree is considered valid and can be pushed into the forest, but first, the tree can undergo trimming in order to remove unnecessary branches that does not lead to or connect with a destination node. The graph would then return '**true**' if all destination nodes were reached. Otherwise, a false would be returned, and the graph would not allocate the traffic of the request while storing the tree into the forest marked as 'inactive'.

The attempt is not entirely correct nor efficient, but it has been a good attempt to solve the problem by modifying the Prim's algorithm and attempting to find a way around it. Here, the function uses additional temporary graphs to store the initial graph before changes were made to the graph during insertion. And by using the same Prim's algorithm to perform which dominates the other operations the time complexity is expected to be $O(E \log V)$ with a space complexity being dependent on the number of vertices and edges in the graph, $O(V + E)$.

```
priority_queue<pair<int,graphEdge>, vector<pair<int,graphEdge>>, PairComparator> pq; //min-heap with prior queue.
vector<bool> inMST(D.size, false); //track visited nodes

vector<graphEdge>::iterator it;
for(it = this->graph.E.begin(); it!=this->graph.E.end(); ++it){
    if(((*it).vertex[0] != s) && ((*it).vertex[1] != s)) continue;
    else if((((*it).vertex[0] == s) || ((*it).vertex[1] == s)) && (*it).b >= t) pq.push(make_pair((*it).ce, *it));
}
vector<int>::iterator it_int;
inMST[s] = true;
tree.V.push_back(s);
int connected = 0;
for(it_int = D.destinationVertices.begin(); it_int != D.destinationVertices.end(); ++it_int){
    if(*it_int == s) connected++;
}
while(!pq.empty() && (connected < D.size)){
    graphEdge U = pq.top().second;
    int u = (inMST[U.vertex[0]] == true)? U.vertex[0] : U.vertex[1]; //source
    int v = (inMST[U.vertex[0]] == true)? U.vertex[1] : U.vertex[0]; //destination
    pq.pop();

    if((inMST[u] == true) && (inMST[v] == true)){
        continue; //both vertices are visited.
    }
    else if(inMST[v] == false && (U.b >= t)){

        inMST[v] = true; //destination node visited
        tree.V.push_back(v);
        for(it_int = D.destinationVertices.begin(); it_int != D.destinationVertices.end(); ++it_int){
            if(*it_int == v) connected++;
        }
        treeEdge edge;
        edge.vertex[0] = u;
        edge.vertex[1] = v;
        tree.E.push_back(edge);
        tree.ct += U.ce * t;
        for(it = this->graph.E.begin(); it != this->graph.E.end(); ++it)
        {
            if(((*it).vertex[0] != v) && ((*it).vertex[1] != v)) continue;
            if((((*it).vertex[0] == u) && (*it).vertex[1] == v) || (((*it).vertex[0] == v) && (*it).vertex[1] == u)){
                (*it).b -= t; //deduct remaining bandwidth
            }
            if((((*it).vertex[0] == v) || ((*it).vertex[1] == v)) && ((*it).vertex[0] != u && (*it).vertex[1] != u) && ((*it).b >= t)){
                // if one of the vertex is the current destination.
                // and the other vertex is not current source.
                // and remaining bandwidth is >= traffic cost.
                pq.push(make_pair((*it).ce, *it));
                //cout << pq.top().first << endl;
            }
        }
    }
}
```

Figure 12 *Insert () main operations*

2.1.2 Stop ()

In the stop () function, similar operations to Problem 1 will take place, the cost will first be returned to the graph. Followed by iterating through the forest starting from the tree with the lowest ID value, we will then check whether the tree is currently active or not, done by having a map which stores a Boolean value, either true or false. By having the value 'true', the graph is considered as active and will not need any additional edges to make the graph reach its destinations. On the other hand, if a tree is not active, then it should be re-routed so as to see whether the tree can now reach all of its destinations.

In this function the estimated time complexity of returning the traffic costs to the graph would be $O(T \cdot E)$ where T represents the tree's edges and E represents the graph edges. As for the time complexity of the re-routing algorithm, since we are iterating over the graph edges and visiting all vertices the time complexity of the algorithm would be $O(E \log V)$.

2.1.3 Rearrange ()

In this function, a similar approach to Problem 1's rearrange () and Problem 2's stop () were used. Unfortunately, I was unable to complete this function in time and ended up rearranging the trees by attempting to find additional trees that could possibly still be added to the graph instead without rerouting previously successful trees.

2.2 Classes for Problem 2

In the first problem the Problem1 class contains the following members which can be listed as seen below:

```
class Problem2 {
public:
    Graph graph;
    Forest gump;
    map<int,int> MTidTraffic;
    map<int,bool> MTactive;
    map<int,Set> MTdestinations;
    Problem2(Graph G);  //constructor
    ~Problem2();        //destructor
    bool insert(int id, int s, Set D, int t, Graph &G, Tree &MTid);
    void stop(int id, Graph &G, Forest &MTidForest);
    void rearrange(Graph &G, Forest &MTidForest);
};
```

Figure 13 *Problem2 class*

In this class, the Graph graph will be used as a duplicate graph to keep track of the changes made to the graph. The program will use this graph as a reference when performing insertion, stop, or rearrange. The next member is Forest gump, this member is used to track the changes made to the forest of the multi-cast tree and will be used as a reference for the program when accessing the trees in the forest. Both previously mentioned members will be used to provide an output graph or forest when asked. The map 'MTidTraffic' will be used to keep a dictionary of each tree's traffic using the tree's unique id as a key. As for 'MTactive', it will be used to track trees' status whether active or inactive. Lastly, 'MTdestinations' will be used to keep record of the destinations the request id needs to satisfy.

Following that two additional structs were used in this problem to sort the min-heap priority queue and the forest.

```
struct PairComparator {
    bool operator()(const std::pair<int, graphEdge>& lhs, const std::pair<int, graphEdge>& rhs) const {
        // Compare based on the first element of the pair
        return lhs.first > rhs.first;
    }
};

struct TreeComparator {
    bool operator()(const Tree &lhs, const Tree &rhs) const{
        // Compare based on the tree's id.
        return lhs.id < rhs.id;
    }
};
```

Figure 14 *Structs for comparators*

Here the pair comparator struct is used to return Boolean values back to the priority queue, the struct is used as a sorting logic for the queue where if the preceding value is greater than the proceeding value, a swap will take place. For tree comparator structure, the opposite is true, a True Boolean value will be returned if the preceding tree's id has a lower value than the proceeding tree's id. The struct is used as a comparison logic for the std::sort calls.

Chapter 3: Test-case Contribution

3.1 Problem 1

For the first problem, I find that the possible points which should be considered are the use of algorithms and data structures. Using an incorrect data structure or approach may lead to RLE or even incorrect answers. It is also important to point out that the request IDs may not be input in ascending or descending order. To begin with, I first create a draft of a possible graph structure, a sparse graph is chosen as I believe that most algorithms will be implemented based on Prim's. The design would have 32 vertices, a third of the constraint's limit, during the design process I first draw out the vertices onto the graph. Edges were then drawn to connect the vertices at random. Following that bandwidths and edge costs were assigned to each edge, here the edges were designed such that there will be some path which has a high amount of bandwidth while others have a lower bandwidth limit. Meanwhile, the cost will also increase in some cases depending on how far it can connect the vertex to other areas of the graph. For example, if the path connects two nodes from two opposite ends of the graph, then the initial available bandwidth may be a little lower and the transmission cost may be higher (like how a real network would work).

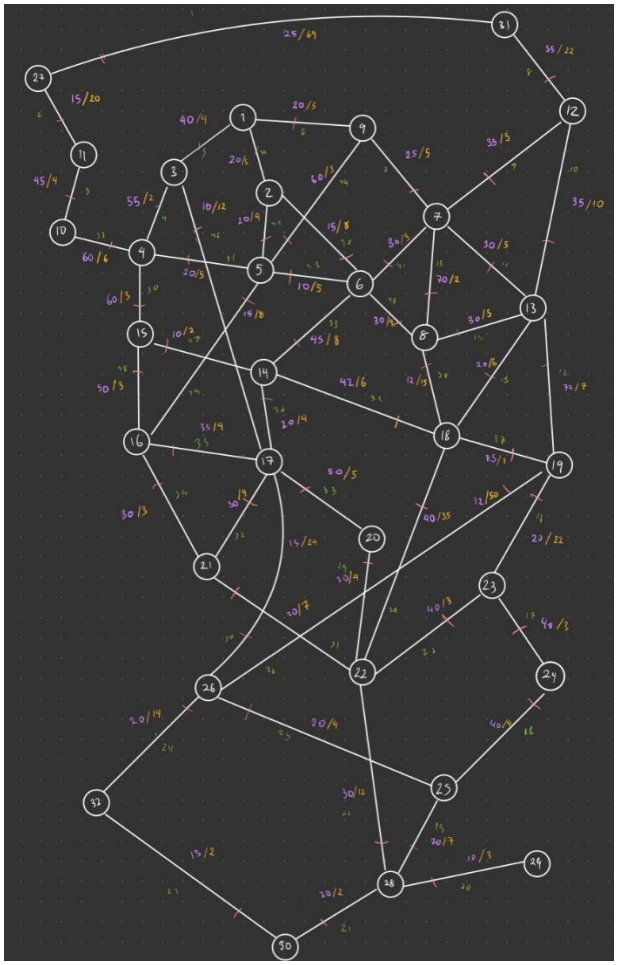


Figure 15 *Graph draft and visualization*

In the draft, the graph spans $|V|$ = 32, and $|E|$ = 54, which is a third of the allowed vertices and is theoretically a sparse graph. I then create the testcase initialization according to the marked bandwidth

and edge cost of the draft graph. Following that, insert functions were called in increasing orders before skipping back and forth. I found that many people may not consider the case that the request ID is not in incrementing order since the TA has miscommunicated during the design of the questions. Thus, a good testcase should consider whether the programmer has correctly handled unordered id requests. We then perform stop calls on different request IDs from different sources to see if the program has successfully handled the stop calls. Following that more insert functions will be called with random but unique request IDs. Here, I must make sure that the IDs called were never used before and that the IDs of stop have been inserted before. Following that rearrange can be called, here I have a feeling that the resetting of graph may take up a lot of time if the algorithm is not efficient enough, thus a lot of rearrange calls were made. The total number of testcases then ranged to about 15,000 function calls.

3.2 Problem 2

Similar approaches were taken for the second problem, by reusing the graph, I first perform simple insert function calls to the program. Here, the tree will span over the whole graph, we can then start stopping a few trees to test their correctness followed by more advanced inserts where the tree will only span to a set of destinations where the members of set D < $|V|$, but the destinations are clustered together. Here, we can test the correctness of the multi-cast tree where not all vertices are a destination. We can then call more stop functions and rearrange functions to test the correctness of the graph further. Lastly, by adding more advanced insert functions where the destination of the trees is far apart from each other (in the opposite ends of the graph) we can check the efficiency of the path finding algorithm. By using less vertices in the graph and providing the graph with less bandwidth limits, I believe that it will be better at checking the ability of the program to search and optimize its path to create the most efficient multi-cast tree routes such that it can accept as many multi-cast trees as possible. An amount of rearrange functions were then called to test the program's ability to make changes. Here the test case is more focused on the tested program's efficiency in routing and rearranging its routes when the test case spammed rearrange function calls. Which I believe can reinforce missing testcases in other contributors who may forget to consider the fact that rearrangements could sometimes come in an endless stream.

We can then perform a test case validation using the Python program that the TA provided, which is a first filtration of whether our test case should be acceptable in the contest.

```
/Problem1_test_case.txt: Test case is valid.
/Problem2_test_case.txt: Test case is valid.
```

Figure 16 *Validation result; Pass*

Chapter 4: References

[1] Wikipedia, Minimum Spanning Tree, https://en.wikipedia.org/wiki/Minimum_spanning_tree

[2] Geeksforgeeks, Prim's MST with priority queue, https://www.geeksforgeeks.org/prims-algorithm-using-priority_queue-stl/

[3] cplusplus.com, C++ STL references, https://cplusplus.com

[4] Geeksforgeeks, Sorting vectors C++, https://www.geeksforgeeks.org/sorting-a-vector-in-c/

[5] Geeksforgeeks, Details of std::sort, https://www.geeksforgeeks.org/internal-details-of-stdsort-in-c/

[6] Wikipedia, Steiner Tree problem, https://en.wikipedia.org/wiki/Steiner_tree_problem

[7] Steiner Minimal Tree, Kun-Mao Chao, https://www.csie.ntu.edu.tw/~kmchao/tree10spr/Steiner.pdf