Ratchanon Hsu (許如來)
111006261

Data Structure's Final Project

Approach to Problem 1

```
void Problem1::insert(int id, int s, Set D, int t, Graph &G, Tree &MTid) {
    /* Store your output graph and multicast tree into G and MTid */

    /* Write your code here. */

    /* clear memory */
    tr.V.clear();
    tr.E.clear();

    /* initialize */
    tr.s = s;
    tr.id = id;
    tr.ct = 0;

    /* Prim's algorithm */
    primMST(s - 1, t, D);

    /* update */
    bandwidth_id.push_back(make_pair(id, t));
    G = gr;
    MTid = tr;
    fo.trees.push_back(tr);
    fo.size++;
    sort(fo.trees.begin(), fo.trees.end(), TreeIdComparator());
    return;
```

Fig. 1: Insert function

Explanation

The objective for this insert function is to implement four essential tasks, which are 1. Clear memory, 2. Initialize, 3. Prim's Algorithm, 4. Update.

Each time the insert function is called, the memory of the multicast tree's edges and vertices from the previous operation are cleared. After that, we begin initializing various variables based on the multicast tree data, such as the multicast tree's source node, id, and total cost.

Additionally, we apply Prim's algorithm to the graph by passing parameters s (source vertex), t (remaining bandwidth), and D (set of destination nodes) in order to find the minimum spanning tree of the multicast tree to all destination vertices. The primMST function will update the vertices, edge, and total cost of the multicast tree in addition to updating the remaining bandwidth of the edge graph that has begun broadcasting.

Lastly, we employ a vector data structure in this update part to preserve the bandwidth that this multicast tree has used. The multicast tree's ID and consumed bandwidth will be stored in this vector in pairs. Next, update to MTid, return our temporary graph, and save it in the forest. Finally, but just as importantly, we arrange our tree in the forest according to id in ascending order.

Time and space complexity

The complexity of this function is primarily determined by primMST function, which implements Prim's algorithm. The time complexity of Prim's algorithm is generally O((V + E) * log(V)), where V is the number of vertices and E is the number of edges in the graph.

The space complexity of this function is determined by the additional data structures used in function. These include the array included[] being O(V), the priority queue Q being O(E), and the vectors tr.V and tr.E which used to store vertices and edges of the multicast tree being O(V + E).

In conclusion, insert function has an O(V + E) * log(V) time complexity and an O(V + E) space complexity.

```
void Problem1::stop(int id, Graph &G, Forest &MTidForest) {
    /* Store your output graph and multicast tree forest into G and MTidForest
       Note: Please "only" include mutlicast trees that you added nodes in MTidForest. */

    /* Write your code here. */

    /* initialize */
    Forest tempFo;

    /* find bandwidth according to id */
    int bandwidth = find_erase_Bandwidth(id);

    /* release bandwidth according to id */
    releaseBandwidth(id, bandwidth);

    /* Prim's algorithm (add edge to existing tree)*/
    primMST_PLUS(tempFo);

    /* update */
    G = gr;
    MTidForest = tempFo;
    return;
}
```

Fig. 2: Stop function

Explanation

The objective for this stop function is to implement five essential tasks, which are 1. Initialize, 2. Find bandwidth according to id, 3. Release bandwidth according to id, 4. Prim's algorithm (add edge to existing tree), 5. Update.

Every time the stop function is called, we will initialize temporary forest to store existing multicast tree after added addition edge(s). After that, we begin to find a bandwidth according to the multicast tree's ID

that consumed bandwidth in the stored vector array. Since we can access value of bandwidth from vector array, now we will release the bandwidth that has been used in the graph and update it.

Moreover, we apply Prim's algorithm to all of the multicast again after we update remaining bandwidth in graph. Then try to connect more vertices of current running multicast trees in the forest by using least additional cost, starting from the smallest id.

Finally, update and return our graph and temporary forest to G and MTid, respectively.

Time and space complexity

The complexity of this function is primarily determined by primMST_PLUS function, which implements Prim's algorithm to modifies an existing tree by adding edges to it. Since it need to be run though all n trees in the forest. The time complexity of this Prim's algorithm is generally O((V + E) * log(V)), but with this new Prim's algorithm has to modifies an existing tree by adding edges to all n trees in the forest, so the time complexity of this Prim's algorithm is O(n * ((V + E) * log(V))), where n is the number of trees in forest, V is the number of vertices, and E is the number of edges in the graph.

The space complexity of this function is determined by the additional data structures used in function. These include the array included[] being O(V), the priority queue Q being O(E), and the vectors tempFo which used to store trees of the multicast tree being O(n(V + E)).

In conclusion, insert function has an O(n * ((V + E) * log(V))) time complexity and an O(n(V + E)) space complexity.

```
void Problem1::rearrange(Graph &G, Forest &MTidForest) {
    /* Store your output graph and multicast tree forest into G and MTidForest
       Note: Please include "all" active mutlicast trees in MTidForest. */

    /* Write your code here. */

    /* initialize */
    Forest tempFo;

    /* reset graph */
    resetGraph();

    /* Prim's algorithm */
    primMST_PLUS_ULTRA(tempFo);

    /* update */
    G = gr;
    fo = tempFo;
    MTidForest = tempFo;

    return;
}
```

Fig. 3: Rearrange function

Ratchanon Hsu (許如來)
111006261

Explanation

The objective for this insert function is to implement four essential tasks, which are 1. Initialize, 2. Reset graph, 3. Prim's Algorithm, 4. Update.

Each time the rearrange function is called, we will remove all of the active multicast trees in both graph and forest, then insert all of them again according to id in ascending order.

First, we will initialize temporary forest to store new multicast trees. Then reset the graph, which mean we're update the remaining bandwidth to the original state of the graph.

Additionally, we apply Prim's algorithm to the graph by using fo(forest in problem 1 class that has store all of the active multicast trees) to insert each tree into the graph again.

Lastly, update and return our graph to G, and temporary forest to fo and MTid.

Time and space complexity

The complexity of this function is determined by primMST_PLUS_ULTRA function, which implements Prim's algorithm to modifies an existing tree by adding edges to it. Since it need to be run thought all n trees in the forest. The time complexity of this Prim's algorithm is generally O((V + E) * log(V)), but with this new Prim's algorithm has to modifies an existing tree by adding edges to all n trees in the forest, so the time complexity of this Prim's algorithm is O(n * ((V + E) * log(V))), where n is the number of trees in forest, V is the number of vertices, and E is the number of edges in the graph.

Moreover, it also determined by resetGraph function, which iterates over all edges and resets their bandwidth being O(E).

The space complexity of this function is determined by the additional data structures used in function. These include the array included[] being O(V), the priority queue Q being O(E), and the vectors tempFo which used to store trees of the multicast tree being O(n(V + E)).

In conclusion, insert function has an O(E + (n * ((V + E) * log(V)))) time complexity and an O(n(V + E)) space complexity.

Ratchanon Hsu (許如來)
111006261

Classes for Problem 1

```
class Problem1 {
private:
    int V;
    int E;
    vector< pair<int, int>> bandwidth_id;
    Graph gr;
    Tree tr;
    Forest fo;
    bool* included;
    void primMST(int s, int t, Set D);
    void primMST_PLUS(Forest &tempFo);
    void primMST_PLUS_ULTRA(Forest &tempFo);
    void updateGraphNegative(vector< graphEdge>::iterator itr, int t);
    void updateGraphPositive(vector< graphEdge>::iterator itr, int t);
    void updateTree(int u, int v, int t, graphEdge gE);
    void updateTreePLUS(int u, int v, int t, graphEdge gE, vector< Tree>::iterator itr_tr);
    int find_erase_Bandwidth(int id);
    int find_Bandwidth(int id);
    void releaseBandwidth(int id, int t);
    void resetGraph();
public:
    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);
};
```

Fig. 4: Classes for Problem 1

Public Method

1. Problem1(Graph G)

```
Problem1::Problem1(Graph G) {
    /* Write your code here. */
    V = G.V.size();
    E = G.E.size();
    gr = G;
    included = new bool [V];
}
```

Fig. 5: Constructor

A constructor that initializes the variables in the Problem1 object with given graph. For instance, V is the number of vertices, E is the number of edges, gr is the duplicate graph that store original graph, and included[] is arrays that store visited vertices in Prim's algorithm.

2. ~Problem1()

```
Problem1::~Problem1() {
    /* Write your code here. */
    delete [] included;

    /* clear graph */
    gr.V.clear();
    gr.E.clear();

    /* clear tree */
    for(vector< Tree>::iterator itr_tr = fo.trees.begin(); itr_tr != fo.trees.end(); ++itr_tr){
        itr_tr -> V.clear();
        itr_tr -> E.clear();
    }

    /* clear forest */
    fo.trees.clear();
}
```

Fig. 6: Destructor

A destructor that frees allocated memory and clears all data structures that it is used. For instance, 1. Free included[] arrays that store visited vertices in Prim's algorithm, 2. Free gr where V is the number of vertices, and E is the number of edges in this duplicate graph. 3. Free duplicate trees in the forest by use iterator to free each tree's vertices and edges of it. 4. Free the duplicate forest.

3. insert(int id, int s, Set D, int t, Graph &G, Tree &MTid)

```
void Problem1::insert(int id, int s, Set D, int t, Graph &G, Tree &MTid) {
    /* Store your output graph and multicast tree into G and MTid */

    /* Write your code here. */

    /* clear memory */
    tr.V.clear();
    tr.E.clear();

    /* initialize */
    tr.s = s;
    tr.id = id;
    tr.ct = 0;

    /* Prim's algorithm */
    primMST(s - 1, t, D);

    /* update */
    bandwidth_id.push_back(make_pair(id, t));
    G = gr;
    MTid = tr;
    fo.trees.push_back(tr);
    fo.size++;
    sort(fo.trees.begin(), fo.trees.end(), TreeIdComparator());
    return;
}
```

Fig. 7: Insert function

The insert function that inserts a multicast tree into the graph. For instance, 1. Clear all memory of previous multicast tree's vertices and edges before store new multicast tree. 2. Initialize s (source vertex), id (id according to the multicast tree's id), ct (total cost of that multicast tree). 3. Apply Prim's algorithm by passing parameters s (source vertex), t (remaining bandwidth), and D (set of destination nodes) though the primMST function. 4. Update bandwidth _id vector that stored id (id according to the multicast tree's id) and t (remaining bandwidth according to multi cast tree's id), update duplicate graph to graph, update duplicate tree to MTid, update duplicate forest by adding duplicate tree in it, update size of forest, sort tree's id in ascending order.

4. stop(int id, Graph &G, Forest &MTidForest)

```
void Problem1::stop(int id, Graph &G, Forest &MTidForest) {
    /* Store your output graph and multicast tree forest into G and MTidForest
       Note: Please "only" include mutlicast trees that you added nodes in MTidForest. */

    /* Write your code here. */

    /* initialize */
    Forest tempFo;

    /* find bandwidth according to id */
    int bandwidth = find_erase_Bandwidth(id);

    /* release bandwidth according to id */
    releaseBandwidth(id, bandwidth);

    /* Prim's algorithm (add edge to existing tree)*/
    primMST_PLUS(tempFo);

    /* update */
    G = gr;
    MTidForest = tempFo;
    return;
}
```

Fig. 8: Stop function

The stop function that stops a multicast tree by releasing its allocated bandwidth and update the graph. For instance, 1. Initialize temporary forest to store existing multicast tree after added addition edge(s). 2. Initialize bandwidth variable to store multicast tree's allocated bandwidth according to multicast tree's id and delete it in bandwidth vector array. 3. Release multicast tree's allocated bandwidth and update to graph. 4. Apply Prim's algorithm by passing temporary forest though function, then try to connect more vertices of current running multicast trees in the forest by using least additional cost, starting from the smallest id. 5. Update and return our graph and temporary forest to G and MTid, respectively.

5. rearrange(Graph &G, Forest &MTidForest)

```
void Problem1::rearrange(Graph &G, Forest &MTidForest) {
    /* Store your output graph and multicast tree forest into G and MTidForest
       Note: Please include "all" active mutlicast trees in MTidForest. */

    /* Write your code here. */

    /* initialize */
    Forest tempFo;

    /* reset graph */
    resetGraph();

    /* Prim's algorithm */
    primMST_PLUS_ULTRA(tempFo);

    /* update */
    G = gr;
    fo = tempFo;
    MTidForest = tempFo;

    return;
}
```

Fig. 9: Rearrange function

The rearrange function that remove all the active multicast trees and insert, then insert all of them again according to id in ascending order. For instance, 1. Initialize temporary forest to store new multicast trees. 2. Reset the graph, which mean we're update the remaining bandwidth to the original state of the graph. 3. Apply Prim's algorithm to the graph by using fo(forest in problem 1 class that has store all of the active multicast trees) to insert each tree into the graph again. 4. update and return our graph to G, and temporary forest to fo and MTid.

Public Method

1. primMST(int s, int t, Set D)

```
void Problem1::primMST(int s, int t, Set D){
    for(int i = 0; i < V; i++){
        included[i] = false;
        //cout << "included " << i << " : " << included[i] << endl;
    }
    priority_queue < pair< int, graphEdge>, vector< pair< int, graphEdge>>, PairComparator> Q;
    for(vector<graphEdge>::iterator itr = gr.E.begin(); itr != gr.E.end(); itr++){
        if((itr -> vertex[0] != s + 1) && itr -> vertex[1] != s + 1){
            continue;
        }
        else if(((itr -> vertex[0] == s + 1) || (itr -> vertex[1] == s + 1)) && (itr -> b >= t)){
            Q.push(make_pair(itr -> ce, *itr));
            //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
        }
    }
    included[s] = true; // visit first source node
    //cout << "included " << s << " : " << included[s] << endl;
    tr.V.push_back(s + 1); // push first source node into the tree
    while(!Q.empty()){
        graphEdge gE = Q.top().second;
        int u = (included[gE.vertex[0] - 1] == true) ? gE.vertex[0] - 1 : gE.vertex[1] - 1;
        int v = (included[gE.vertex[0] - 1] == true) ? gE.vertex[1] - 1 : gE.vertex[0] - 1;
        Q.pop();
        if((included[u] == true) && (included[v] == true)){
            //cout << "included " << u << " : " << included[u] << " / included " << v << " : " << included[v] << endl;
            //cout << "skip\n";
            continue; // edge has been visited
        }
        else if((included[v] == false) && (gE.b >= t)){
            //cout << "valid\n";
            included[v] = true; // visit destination node
            updateTree(u, v, t, gE);
            for(vector< graphEdge>::iterator itr = gr.E.begin(); itr != gr.E.end(); ++itr){
                if((itr -> vertex[0] != v + 1) && itr -> vertex[1] != v + 1){
                    continue;
                }
                if((itr -> vertex[0] == u + 1) && (itr -> vertex[1] == v + 1) || (itr -> vertex[1] == u + 1) && (itr -> vertex[0] == v + 1)){
                    updateGraphNegative(itr, t);
                }
                if(((itr -> vertex[0] == v + 1) || (itr -> vertex[1] == v + 1)) && ((itr -> vertex[0] != u + 1) && (itr -> vertex[1] != u + 1)) && (itr -> b >= t)){
                    Q.push(make_pair(itr -> ce, *itr));
                    //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
                }
            }
        }
    }
}
```

Fig. 10: Prim's algorithm function for insert function

The primMST function for insert function that implement Prim's algorithm to find minimum spanning tree. For instance, 1. Initialize array included[] as all vertices as non-visited, and priority queue Q to stored graph edge cost and graph edge as a pair. 2. Set first source vertex as visited and push it back to multicast tree's vertices array. 3. In while loop, take graph edge at first queue of Q and find whether it can span to the other vertex with minimum cost and enough remaining bandwidth or not. 4. Update included[] array if vertex has been visited, then update multicast tree and graph edge's remaining bandwidth. 5. Push new valid graph edge if it can span to other vertex with minimum cost and valid remaining bandwidth into Q priority queue.

2. primMST_PLUS(Forest &tempFo)

```cpp
void Problem1::primMST_PLUS(Forest &tempFo){
    for(vector< Tree>::iterator itr_tr = fo.trees.begin(); itr_tr != fo.trees.end(); ++itr_tr){
        //cout << "Tree no: " << itr_tr - fo.trees.begin() + 1 << endl;
        int flag = false;
        int t = find_Bandwidth(itr_tr -> id);
        for(int i = 0; i < V; i++){
            included[i] = false;
            //cout << "included " << i << " : " << included[i] << endl;
        }
        for(vector< int>::iterator itr = itr_tr -> V.begin(); itr != itr_tr -> V.end(); ++itr){
            included[(*itr) - 1] = true; // marked node that tree has already visited
            //cout << "included " << (*itr) - 1 << " : " << included[(*itr) - 1] << endl;
        }
        priority_queue < pair< int, graphEdge>, vector< pair< int, graphEdge>>, PairComparator> Q;
        for(vector<graphEdge>::iterator itr_gE = gr.E.begin(); itr_gE != gr.E.end(); itr_gE++){
            if((included[itr_gE -> vertex[0] - 1] == true) && (included[itr_gE -> vertex[1] - 1] == true)){
                continue;
            }
            else if(((included[itr_gE -> vertex[0] - 1] == true) || (included[itr_gE -> vertex[1] - 1] == true)) && (itr_gE -> b >= t)){
                Q.push(make_pair(itr_gE -> ce, *itr_gE));
                //cout << "cost : " << itr_gE -> ce << " / vertex[0] : " << itr_gE -> vertex[0] << " / vertex[1] : " << itr_gE -> vertex[1] << endl;
            }
        }
        while(!Q.empty()){
            graphEdge gE = Q.top().second;
            int u = (included[gE.vertex[0] - 1] == true) ? gE.vertex[0] - 1 : gE.vertex[1] - 1;
            int v = (included[gE.vertex[0] - 1] == true) ? gE.vertex[1] - 1 : gE.vertex[0] - 1;
            Q.pop();
            if((included[u] == true) && (included[v] == true)){
                //cout << "included " << u << " : " << included[u] << " / included " << v << " : " << included[v] << endl;
                //cout << "skip\n";
                continue; // edge has been visited
            }
            else if((included[v] == false) && (gE.b >= t)){
                //cout << "valid\n";
                included[v] = true; // visit destination node
                updateTreePLUS(u, v, t, gE, itr_tr);
                flag = true;
                for(vector< graphEdge>::iterator itr = gr.E.begin(); itr != gr.E.end(); ++itr){
                    if((itr -> vertex[0] != v + 1) && itr -> vertex[1] != v + 1){
                        continue;
                    }
                    if((itr -> vertex[0] == u + 1) && (itr -> vertex[1] == v + 1) || (itr -> vertex[1] == u + 1) && (itr -> vertex[0] == v + 1)){
                        updateGraphNegative(itr, t);
                    }
                    if(((itr -> vertex[0] == v + 1) || (itr -> vertex[1] == v + 1)) && ((itr -> vertex[0] != u + 1) && (itr -> vertex[1] != u + 1)) && (itr -> b >= t)){
                        Q.push(make_pair(itr -> ce, *itr));
                        //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
                    }
                }
            }
        }
        if(flag){
            tempFo.trees.push_back((*itr_tr));
            fo.trees.erase(itr_tr);
            fo.trees.push_back((*itr_tr));
            //cout << "total new cost : " << itr_tr -> ct << endl;
        }
    }
}
```

Fig. 11: Prim's algorithm function for stop function

The primMST_PLUS function for stop function that implement Prim's algorithm to find minimum spanning tree to modifies an existing tree by adding edges new valid graph edge to it. This functionality of this function is similar to primMST function, but this primMST_PLUS has to iterate all trees in the forest in order to check whether it has to add valid graph edge or not (find minimum spanning tree if it possible to reach all destination vertices).

3. primMST_PLUS_ULTRA(Forest &tempFo)

```
void Problem1::primMST_PLUS_ULTRA(Forest &tempFo){
    for(vector< Tree>::iterator itr_tr = fo.trees.begin(); itr_tr != fo.trees.end(); ++itr_tr){
        //cout << "Tree no: " << itr_tr - fo.trees.begin() + 1 << endl;
        /* clear memory */
        itr_tr -> V.clear();
        itr_tr -> E.clear();
        /* initialize */
        itr_tr -> ct = 0;
        int s = itr_tr -> s - 1;
        //cout << "source : " << s << endl;
        int id = itr_tr -> id;
        //cout << "id : " << id << endl;
        int t = find_Bandwidth(id);
        //cout << "bandwidth : " << t << endl;
        for(int i = 0; i < V; i++){
            included[i] = false;
            //cout << "included " << i << " : " << included[i] << endl;
        }
        priority_queue < pair< int, graphEdge>, vector< pair< int, graphEdge>>, PairComparator> Q;
        for(vector<graphEdge>::iterator itr_gE = gr.E.begin(); itr_gE  != gr.E.end(); itr_gE ++){
            if((itr_gE  -> vertex[0] != s + 1) && itr_gE  -> vertex[1] != s + 1){
                continue;
            }
            else if(((itr_gE  -> vertex[0] == s + 1) || (itr_gE  -> vertex[1] == s + 1)) && (itr_gE  -> b >= t)){
                Q.push(make_pair(itr_gE  -> ce, *itr_gE ));
                //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
            }
        }
        included[s] = true; // visit first source node
        //cout << "included " << s << " : " << included[s] << endl;
        itr_tr -> V.push_back(s + 1); // push first source node into the tree
        while(!Q.empty()){
            graphEdge gE = Q.top().second;
            int u = (included[gE.vertex[0] - 1] == true) ? gE.vertex[0] - 1 : gE.vertex[1] - 1;
            int v = (included[gE.vertex[0] - 1] == true) ? gE.vertex[1] - 1 : gE.vertex[0] - 1;
            Q.pop();
            if((included[u] == true) && (included[v] == true)){
                //cout << "included " << u << " : " << included[u] << " / included " << v << " : " << included[v] << endl;
                //cout << "skip\n";
                continue; // edge has been visited
            }
            else if((included[v] == false) && (gE.b >= t)){
                //cout << "valid\n";
                included[v] = true; // visit destination node
                updateTreePLUS(u, v, t, gE, itr_tr);
                for(vector< graphEdge>::iterator itr = gr.E.begin(); itr != gr.E.end(); ++itr){
                    if((itr -> vertex[0] != v + 1) && itr -> vertex[1] != v + 1){
                        continue;
                    }
                    if((itr -> vertex[0] == u + 1) && (itr -> vertex[1] == v + 1) || (itr -> vertex[1] == u + 1) && (itr -> vertex[0] == v + 1)){
                        updateGraphNegative(itr, t);
                    }
                    if(((itr -> vertex[0] == v + 1) || (itr -> vertex[1] == v + 1)) && ((itr -> vertex[0] != u + 1) && (itr -> vertex[1] != u + 1)) && (itr -> b >= t)){
                        Q.push(make_pair(itr -> ce, *itr));
                        //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
                    }
                }
            }
        }
        tempFo.trees.push_back((*itr_tr));
    }
}
```

Fig. 12: Prim's algorithm function for rearrange function

The primMST_PLUS_ULTRA function for rearrange function that implement Prim's algorithm to find minimum spanning tree. This functionality of this function is similar to primMST function, but this primMST_PLUS_ULTRA need to iterate all multicast trees in the forest in order to clear all of the memory of the multicast tree into the original state, and apply Prim's algorithm, then store this new multicast tree in the temporary forest and return it.

4. updateGraphPositive(vector< graphEdge>::iterator itr, int t)

```
void Problem1::updateGraphPositive(vector< graphEdge>::iterator itr, int t){
    itr -> b += t; // add remaining bandwidth to the edge
}
```

Fig. 13: Update graph by adding graph edge's remaining bandwidth

Updates the graph by adding graph edge's remaining bandwidth.

5. updateGraphNegative(vector< graphEdge>::iterator itr, int t)

```
void Problem1::updateGraphNegative(vector< graphEdge>::iterator itr, int t){
    itr -> b -= t; // deduct remaining bandwidth to the edge
}
```

Fig. 14: Updates the graph by adding graph edge's remaining bandwidth.

Updates the graph by deducting graph edge's remaining bandwidth.

6. updateTree(int u, int v, int t, graphEdge gE)

```
void Problem1::updateTreePLUS(int u, int v, int t, graphEdge gE, vector< Tree>::iterator itr_tr){
    treeEdge tE;
    tE.vertex[0] = u + 1;
    tE.vertex[1] = v + 1;
    //cout << "tree vertex 0 : " << tE.vertex[0] << " / tree vertex 1 : " << tE.vertex[1] << endl;
    itr_tr -> ct += (gE.ce * t);
    //cout << "cost ++ : " << tr.ct << endl;
    itr_tr -> V.push_back(v + 1); // push destination node into tree
    itr_tr -> E.push_back(tE); // push edge into tree
}
```

Fig. 15: Update tree function for primMST

The updateTree function for primMST that update multicast tree's destination vertex, edge, total cost.

7. updateTreePLUS(int u, int v, int t, graphEdge gE, vector< Tree>::iterator itr_tr)

```
void Problem1::updateTree(int u, int v, int t, graphEdge gE){
    treeEdge tE;
    tE.vertex[0] = u + 1;
    tE.vertex[1] = v + 1;
    //cout << "tree vertex 0 : " << tE.vertex[0] << " / tree vertex 1 : " << tE.vertex[1] << endl;
    tr.ct += (gE.ce * t);
    //cout << "cost ++ : " << tr.ct << endl;
    tr.V.push_back(v + 1); // push destination node into tree
    tr.E.push_back(tE); // push edge into tree
}
```

Fig. 16: Update tree function for primMST_PLUS and primMST_PLUS_ULTRA

The updateTree function for primMST_PLUS and primMST_PLUS_ULTRA that update multicast tree's destination vertex, edge, total cost according to iterator point to which id of the multicast tree.

8. find_erase_Bandwidth(int id)

```
int Problem1::find_erase_Bandwidth(int id){
    int bandwidth = 0;
    auto iter = find_if(bandwidth_id.begin(), bandwidth_id.end(), [id](pair<int, int> &element){return element.first == id;});
    if (iter != bandwidth_id.end()){
        bandwidth = iter->second;
        bandwidth_id.erase(iter);
        return bandwidth;
    }
    return 0;
}
```

Fig. 17: Find bandwidth and erase bandwidth according to multicast tree's id

The find_erase_Bandwidth function that find the allocated bandwidth of multicast tree according to multicast' id and erase it in bandwidth_it vectors array.

9. find_Bandwidth(int id)

```
int Problem1::find_Bandwidth(int id){
    int bandwidth = 0;
    auto iter = find_if(bandwidth_id.begin(), bandwidth_id.end(), [id](pair<int, int> &element){return element.first == id;});
    if (iter != bandwidth_id.end()){
        bandwidth = iter->second;
        return bandwidth;
    }
    return 0;
}
```

Fig. 18: Find bandwidth according to multicast tree's id

The find_Bandwidth function that find the allocated bandwidth of multicast tree according to multicast' id

10. releaseBandwidth(int id, int t)

```
void Problem1::releaseBandwidth(int id, int t){
    int num = 0;
    for(vector< Tree>::iterator itr_tr = fo.trees.begin(); itr_tr != fo.trees.end(); ++itr_tr){
        if(itr_tr -> id == id){
            for(vector< graphEdge>::iterator itr_gE = gr.E.begin(); itr_gE != gr.E.end(); ++itr_gE){
                if(num == itr_tr -> E.size()){
                    break;
                }
                for(vector< treeEdge>::iterator itr_tE = itr_tr -> E.begin(); itr_tE != itr_tr -> E.end(); ++itr_tE){
                    if(((itr_gE -> vertex[0] == itr_tE -> vertex[0]) && (itr_gE -> vertex[1] == itr_tE -> vertex[1])) ||
                    ((itr_gE -> vertex[0] == itr_tE -> vertex[1]) && (itr_gE -> vertex[1] == itr_tE -> vertex[0]))){
                        updateGraphPositive(itr_gE, t);
                        num++;
                    }
                }
            }
            //cout << "total old cost : " << itr_tr -> ct << endl;
            itr_tr -> V.clear();
            itr_tr -> E.clear();
            fo.trees.erase(itr_tr);
            fo.size--;
            break;
        }
    }
}
```

Fig. 19: Release bandwidth of graph edge

The releaseBanwidth function that iterate over all of the multicast trees in the forest to find which graph edge need to be update. If it updates all of the graph edge, then its clear memory and erase multicast tree according to the stop id.

11. resetGraph()

```
void Problem1::resetGraph(){
    for(vector< graphEdge>::iterator itr_gE = gr.E.begin(); itr_gE != gr.E.end(); ++itr_gE){
        itr_gE -> b = itr_gE -> be;
    }
}
```

Fig. 20: Reset Graph remaining bandwidth to original state

The resetGraph function that iterate to all over the graph edge to reset remaining bandwidth to the original state.

Approach to Problem 2

```
bool Problem2::insert(int id, int s, Set D, int t, Graph &G, Tree &MTid) {
    /* Store your output graph and multicast tree into G and MTid */

    /* Write your code here. */

    /* clear memory */
    tr.V.clear();
    tr.E.clear();

    /* initialize */
    tr.s = s;
    tr.id = id;
    tr.ct = 0;

    /* Prim's algorithm */
    primMST(s - 1, t, D);

    /* update */
    bandwidth_id.push_back(make_pair(id, t));
    G = gr;
    MTid = tr;
    fo.trees.push_back(tr);
    fo.size++;
    sort(fo.trees.begin(), fo.trees.end(), TreeIdComparator());
    /* You should return true or false according the insertion result */
    int flag = 1;
    for(int i = 0; i < D.size; i++){
        int found = 0;
        for(vector< int>::iterator itr = tr.V.begin(); itr != tr.V.end(); ++itr){
            //cout << "dest : " << D.destinationVertices[i] << " / *itr : " << *itr << endl;
            if((D.destinationVertices[i] == (*itr))){
                found = 1;
                break;
            }
        }
        if(!found){
            flag = 0;
            break;
        }
    }
    if(flag){
        return true;
    }
    else{
        return false;
    }
}
```

Fig. 20: Insert function

Ratchanon Hsu (許如來)
111006261

Explanation

The objective for this insert function is to implement four essential tasks, which are 1. Clear memory, 2. Initialize, 3. Prim's Algorithm, 4. Update.

Each time the insert function is called, the memory of the multicast tree's edges and vertices from the previous operation are cleared. After that, we begin initializing various variables based on the multicast tree data, such as the multicast tree's source node, id, and total cost.

Additionally, we apply Prim's algorithm to the graph by passing parameters s (source vertex), t (remaining bandwidth), and D (set of destination nodes) in order to find the minimum spanning tree of the multicast tree to **specific** destination vertices. The primMST function will update the vertices, edge, and total cost of the multicast tree in addition to updating the remaining bandwidth of the edge graph that has begun broadcasting.

Moreover, we employ a vector data structure in this update part to preserve the bandwidth that this multicast tree has used. The multicast tree's ID and consumed bandwidth will be stored in this vector in pairs. Next, update to MTid, return our temporary graph, and save it in the forest. Finally, but just as importantly, we arrange our tree in the forest according to id in ascending order.

Lastly, the last section is to check whether this insertion tree is valid or not. If it return, it's mean it's accept this this request multicast tree, vice versa.

Time and space complexity

The complexity of this function is primarily determined by primMST function, which implements Prim's algorithm. The time complexity of Prim's algorithm is generally O((V + E) * log(V)), where V is the number of vertices and E is the number of edges in the graph.

The space complexity of this function is determined by the additional data structures used in function. These include the array included[] being O(V), the priority queue Q being O(E), and the vectors tr.V and tr.E which used to store vertices and edges of the multicast tree being O(V + E).

In conclusion, insert function has an O(V + E) * log(V) time complexity and an O(V + E) space complexity.

```
void Problem2::stop(int id, Graph &G, Forest &MTidForest) {
    /* Store your output graph and multicast tree forest into G and MTidForest
       Note: Please "only" include mutlicast trees that you added nodes in MTidForest. */

    /* Write your code here. */

    /* initialize */
    Forest tempFo;

    /* find bandwidth according to id */
    int bandwidth = find_erase_Bandwidth(id);

    /* release bandwidth according to id */
    releaseBandwidth(id, bandwidth);

    /* Prim's algorithm (add edge to existing tree)*/
    primMST_PLUS(tempFo);

    /* update */
    G = gr;
    MTidForest = tempFo;
    return;
}
```

Fig. 21: Stop function

Explanation

The objective for this stop function is to implement five essential tasks, which are 1. Initialize, 2. Find bandwidth according to id, 3. Release bandwidth according to id, 4. Prim's algorithm (add edge to existing tree), 5. Update.

Every time the stop function is called, we will initialize temporary forest to store existing multicast tree after added addition edge(s). After that, we begin to find a bandwidth according to the multicast tree's ID that consumed bandwidth in the stored vector array. Since we can access value of bandwidth from vector array, now we will release the bandwidth that has been used in the graph and update it.

Moreover, we apply Prim’s algorithm to all of the multicast again after we update remaining bandwidth in graph. Then try to connect more vertices of current running multicast trees in the forest by using least additional cost, starting from the smallest id.

Finally, update and return our graph and temporary forest to G and MTid, respectively.

Time and space complexity

The complexity of this function is primarily determined by primMST_PLUS function, which implements Prim’s algorithm to modifies an existing tree by adding edges to it. Since it need to be run though all n trees in the forest. The time complexity of this Prim’s algorithm is generally O((V + E) * log(V)), but with this new Prim’s algorithm has to modifies an existing tree by adding edges to all n trees

in the forest, so the time complexity of this Prim's algorithm is O(n * ((V + E) * log(V))), where n is the number of trees in forest, V is the number of vertices, and E is the number of edges in the graph.

The space complexity of this function is determined by the additional data structures used in function. These include the array included[] being O(V), the priority queue Q being O(E), and the vectors tempFo which used to store trees of the multicast tree being O(n(V + E)).

In conclusion, insert function has an O(n * ((V + E) * log(V)))) time complexity and an O(n(V + E)) space complexity.

```
void Problem2::rearrange(Graph &G, Forest &MTidForest) {
    /* Store your output graph and multicast tree forest into G and MTidForest
       Note: Please include "all" active mutlicast trees in MTidForest. */

    /* Write your code here. */

    Forest tempFo;

    /* reset graph */
    resetGraph();

    /* Prim's algorithm */
    primMST_PLUS_ULTRA(tempFo);

    /* update */
    G = gr;
    fo = tempFo;
    MTidForest = tempFo;

    return;
}
```

Fig. 22: Rearrange function

Explanation

The objective for this insert function is to implement four essential tasks, which are 1. Initialize, 2. Reset graph, 3. Prim's Algorithm, 4. Update.

Each time the rearrange function is called, we will remove all of the active multicast trees in both graph and forest, then insert all of them again according to id in ascending order.

First, we will initialize temporary forest to store new multicast trees. Then reset the graph, which mean we're update the remaining bandwidth to the original state of the graph.

Additionally, we apply Prim's algorithm to the graph by using fo(forest in problem 2 class that has store all of the active multicast trees) to insert each tree into the graph again.

Lastly, update and return our graph to G, and temporary forest to fo and MTid.

Ratchanon Hsu (許如來)
111006261

Time and space complexity

The complexity of this function is determined by primMST_PLUS_ULTRA function, which implements Prim's algorithm to modifies an existing tree by adding edges to it. Since it need to be run thought all n trees in the forest. The time complexity of this Prim's algorithm is generally O((V + E) * log(V)), but with this new Prim's algorithm has to modifies an existing tree by adding edges to all n trees in the forest, so the time complexity of this Prim's algorithm is O(n * ((V + E) * log(V))), where n is the number of trees in forest, V is the number of vertices, and E is the number of edges in the graph.

Moreover, it also determined by resetGraph function, which iterates over all edges and resets their bandwidth being O(E).

The space complexity of this function is determined by the additional data structures used in function. These include the array included[] being O(V), the priority queue Q being O(E), and the vectors tempFo which used to store trees of the multicast tree being O(n(V + E)).

In conclusion, insert function has an O(E + (n * ((V + E) * log(V)))) time complexity and an O(n(V + E)) space complexity.

Classes for Problem 2

```
class Problem2 {
private:
    int V;
    int E;
    vector< pair<int, int>> bandwidth_id;
    Graph gr;
    Tree tr;
    Forest fo;
    Set se;
    vector<Set> se_vector;
    bool* included;
    void primMST(int s, int t, Set D);
    void primMST_PLUS(Forest &tempFo);
    void primMST_PLUS_ULTRA(Forest &tempFo);
    bool isInArray(Set D, int num);
    bool isInSubset(Set D, int num);
    void updateGraphNegative(vector< graphEdge>::iterator itr, int t);
    void updateGraphPositive(vector< graphEdge>::iterator itr, int t);
    void updateTree(int u, int v, int t, graphEdge gE);
    void updateTreePLUS(int u, int v, int t, graphEdge gE, vector< Tree>::iterator itr_tr);
    int find_erase_Bandwidth(int id);
    int find_Bandwidth(int id);
    void releaseBandwidth(int id, int t);
    void resetGraph();
public:
    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);
};
```

Fig. 23: Classes for Problem 2

Public Method

1. Problem2(Graph G)

```
Problem2::Problem2(Graph G) {
    /* Write your code here. */
    V = G.V.size();
    E = G.E.size();
    gr = G;
    included = new bool [V];
}
```

Fig. 24: Constructor

A constructor that initializes the variables in the Problem1 object with given graph. For instance, V is the number of vertices, E is the number of edges, gr is the duplicate graph that store original graph, and included[] is arrays that store visited vertices in Prim's algorithm.

2. ~Problem2()

```
Problem2::~Problem2() {
    /* Write your code here. */
    delete [] included;

    /* clear graph */
    gr.V.clear();
    gr.E.clear();

    /* clear tree */
    for(vector< Tree>::iterator itr_tr = fo.trees.begin(); itr_tr != fo.trees.end(); ++itr_tr){
        itr_tr -> V.clear();
        itr_tr -> E.clear();
    }

    /* clear forest */
    fo.trees.clear();
}
```

Fig. 25: Destructor

A destructor that frees allocated memory and clears all data structures that it is used. For instance, 1. Free included[] arrays that store visited vertices in Prim's algorithm, 2. Free gr where V is the number of vertices, and E is the number of edges in this duplicate graph. 3. Free duplicate trees in the forest by use iterator to free each tree's vertices and edges of it. 4. Free the duplicate forest.

3. insert(int id, int s, Set D, int t, Graph &G, Tree &MTid)

```cpp
bool Problem2::insert(int id, int s, Set D, int t, Graph &G, Tree &MTid) {
    /* Store your output graph and multicast tree into G and MTid */

    /* Write your code here. */

    /* clear memory */
    tr.V.clear();
    tr.E.clear();

    /* initialize */
    tr.s = s;
    tr.id = id;
    tr.ct = 0;

    /* Prim's algorithm */
    primMST(s - 1, t, D);

    /* update */
    bandwidth_id.push_back(make_pair(id, t));
    G = gr;
    MTid = tr;
    fo.trees.push_back(tr);
    fo.size++;
    sort(fo.trees.begin(), fo.trees.end(), TreeIdComparator());
    /* You should return true or false according the insertion result */
    int flag = 1;
    for(int i = 0; i < D.size; i++){
        int found = 0;
        for(vector< int>::iterator itr = tr.V.begin(); itr != tr.V.end(); ++itr){
            //cout << "dest : " << D.destinationVertices[i] << " / *itr : " << *itr << endl;
            if((D.destinationVertices[i] == (*itr))){
                found = 1;
                break;
            }
        }
        if(!found){
            flag = 0;
            break;
        }
    }
    if(flag){
        return true;
    }
    else{
        return false;
    }
}
```

Fig. 26: Insert function

The insert function that inserts a multicast tree into the graph. For instance, 1. Clear all memory of previous multicast tree's vertices and edges before store new multicast tree. 2. Initialize s (source vertex), id (id according to the multicast tree's id), ct (total cost of that multicast tree). 3. Apply Prim's algorithm by passing parameters s (source vertex), t (remaining bandwidth), and D (set of destination nodes) though the primMST function. 4. Update bandwidth _id vector that stored id (id according to the multicast tree's id) and t (remaining bandwidth according to multi cast tree's id), update duplicate graph to graph, update duplicate tree to MTid, update duplicate forest by adding duplicate tree in it, update size of forest, sort tree's id in ascending order.

3. stop(int id, Graph &G, Forest &MTidForest)

```
void Problem2::stop(int id, Graph &G, Forest &MTidForest) {
    /* Store your output graph and multicast tree forest into G and MTidForest
       Note: Please "only" include mutlicast trees that you added nodes in MTidForest. */

    /* Write your code here. */

    /* initialize */
    Forest tempFo;

    /* find bandwidth according to id */
    int bandwidth = find_erase_Bandwidth(id);

    /* release bandwidth according to id */
    releaseBandwidth(id, bandwidth);

    /* Prim's algorithm (add edge to existing tree)*/
    primMST_PLUS(tempFo);

    /* update */
    G = gr;
    MTidForest = tempFo;
    return;
}
```

Fig. 27: Stop function

The stop function that stops a multicast tree by releasing its allocated bandwidth and update the graph. For instance, 1. Initialize temporary forest to store existing multicast tree after added addition edge(s). 2. Initialize bandwidth variable to store multicast tree's allocated bandwidth according to multicast tree's id and delete it in bandwidth vector array. 3. Release multicast tree's allocated bandwidth and update to graph. 4. Apply Prim's algorithm by passing temporary forest though function, then try to connect more vertices of current running multicast trees in the forest by using least additional cost, starting from the smallest id. 5. Update and return our graph and temporary forest to G and MTid, respectively.

4. rearrange(Graph &G, Forest &MTidForest)

```
void Problem2::rearrange(Graph &G, Forest &MTidForest) {
    /* Store your output graph and multicast tree forest into G and MTidForest
       Note: Please include "all" active mutlicast trees in MTidForest. */

    /* Write your code here. */

    Forest tempFo;

    /* reset graph */
    resetGraph();

    /* Prim's algorithm */
    primMST_PLUS_ULTRA(tempFo);

    /* update */
    G = gr;
    fo = tempFo;
    MTidForest = tempFo;

    return;
}
```

Fig. 28: Rearrange function

The rearrange function that remove all the active multicast trees and insert, then insert all of them again according to id in ascending order. For instance, 1. Initialize temporary forest to store new multicast trees. 2. Reset the graph, which mean we're update the remaining bandwidth to the original state of the graph. 3. Apply Prim's algorithm to the graph by using fo(forest in problem 2 class that has store all of the active multicast trees) to insert each tree into the graph again. 4. update and return our graph to G, and temporary forest to fo and MTid.

Public Method

1. primMST(int s, int t, Set D)

```
void Problem2::primMST(int s, int t, Set D){
    for(int i = 0; i < V; i++){
        included[i] = false;
        //cout << "included " << i << " : " << included[i] << endl;
    }
    priority_queue < pair< int, graphEdge>, vector< pair< int, graphEdge>>, PairComparator> Q;
    for(vector<graphEdge>::iterator itr = gr.E.begin(); itr != gr.E.end(); itr++){
        if((itr -> vertex[0] != s + 1) && itr -> vertex[1] != s + 1){
            continue;
        }
        else if(((itr -> vertex[0] == s + 1) || (itr -> vertex[1] == s + 1)) && (itr -> b >= t) && isInArray(D, itr -> vertex[0]) || isInArray(D, itr -> vertex[1])){
            Q.push(make_pair(itr -> ce, *itr));
            //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
        }
    }
    included[s] = true; // visit first source node
    //cout << "included " << s << " : " << included[s] << endl;
    tr.V.push_back(s + 1); // push first source node into the tree
    while(!Q.empty()){
        graphEdge gE = Q.top().second;
        int u = (included[gE.vertex[0] - 1] == true) ? gE.vertex[0] - 1 : gE.vertex[1] - 1;
        int v = (included[gE.vertex[0] - 1] == true) ? gE.vertex[1] - 1 : gE.vertex[0] - 1;
        Q.pop();
        if((included[u] == true) && (included[v] == true)){
            //cout << "included " << u << " : " << included[u] << " / included " << v << " : " << included[v] << endl;
            //cout << "skip\n";
            continue; // edge has been visited
        }
        else if((included[v] == false) && (gE.b >= t)){
            //cout << "valid\n";
            //cout << "cost : " << gE.ce << " / u : " << u << " / v : " << v << endl;
            included[v] = true; // visit destination node
            updateTree(u, v, t, gE);
            for(vector< graphEdge>::iterator itr = gr.E.begin(); itr != gr.E.end(); ++itr){
                if((itr -> vertex[0] != v + 1) && itr -> vertex[1] != v + 1){
                    continue;
                }
                if((itr -> vertex[0] == u + 1) && (itr -> vertex[1] == v + 1) || (itr -> vertex[1] == u + 1) && (itr -> vertex[0] == v + 1)){
                    updateGraphNegative(itr, t);
                }
                if(((itr -> vertex[0] == v + 1) || (itr -> vertex[1] == v + 1)) && ((itr -> vertex[0] != u + 1) && (itr -> vertex[1] != u + 1)) && (itr -> b >= t)&& isInArray(D, itr -> vertex[0]) || isInArray(D, itr -> vertex[1])){
                    Q.push(make_pair(itr -> ce, *itr));
                    //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
                }
            }
        }
    }
    //cout << "tr.V.size() : " << tr.V.size() << endl;
    //cout << "D.size : " << D.size << endl;
}
```

Fig. 29: Prim's algorithm function for insert function

The primMST function for insert function that implement Prim's algorithm to find minimum spanning tree. For instance, 1. Initialize array included[] as all vertices as non-visited, and priority queue Q to stored graph edge cost and graph edge as a pair. 2. Set first source vertex as visited and push it back to multicast tree's vertices array. 3. In while loop, take graph edge at first queue of Q and find whether it can span to the other vertex with minimum cost and enough remaining bandwidth or not. 4. Update included[] array if vertex has been visited, then update multicast tree and graph edge's remaining bandwidth. 5. Push new valid graph edge if it can span to other vertex with minimum cost and valid remaining bandwidth into Q priority queue.

2. primMST_PLUS(Forest &tempFo)

```
void Problem2::primMST_PLUS(Forest &tempFo){
    for(vector< Tree>::iterator itr_tr = fo.trees.begin(); itr_tr != fo.trees.end(); ++itr_tr){
        //cout << "Tree no: " << itr_tr - fo.trees.begin() + 1 << endl;
        int flag = false;
        int t = find_Bandwidth(itr_tr -> id);
        for(int i = 0; i < V; i++){
            included[i] = false;
            //cout << "included " << i << " : " << included[i] << endl;
        }
        for(vector< int>::iterator itr = itr_tr -> V.begin(); itr != itr_tr -> V.end(); ++itr){
            included[(*itr) - 1] = true; // marked node that tree has already visited
            //cout << "included " << (*itr) - 1 << " : " << included[(*itr) - 1] << endl;
        }
        priority_queue < pair< int, graphEdge>, vector< pair< int, graphEdge>>, PairComparator> Q;
        for(vector<graphEdge>::iterator itr_gE = gr.E.begin(); itr_gE != gr.E.end(); itr_gE++){
            if((included[itr_gE -> vertex[0] - 1] == true) && (included[itr_gE -> vertex[1] - 1] == true)){
                continue;
            }
            else if(((included[itr_gE -> vertex[0] - 1] == true) || (included[itr_gE -> vertex[1] - 1] == true)) && (itr_gE -> b >= t)){
                Q.push(make_pair(itr_gE -> ce, *itr_gE));
                //cout << "cost : " << itr_gE -> ce << " / vertex[0] : " << itr_gE -> vertex[0] << " / vertex[1] : " << itr_gE -> vertex[1] << endl;
            }
        }
        while(!Q.empty()){
            graphEdge gE = Q.top().second;
            int u = (included[gE.vertex[0] - 1] == true) ? gE.vertex[0] - 1 : gE.vertex[1] - 1;
            int v = (included[gE.vertex[0] - 1] == true) ? gE.vertex[1] - 1 : gE.vertex[0] - 1;
            Q.pop();
            if((included[u] == true) && (included[v] == true)){
                //cout << "included " << u << " : " << included[u] << " / included " << v << " : " << included[v] << endl;
                //cout << "skip\n";
                continue; // edge has been visited
            }
            else if((included[v] == false) && (gE.b >= t)){
                //cout << "valid\n";
                included[v] = true; // visit destination node
                updateTreePLUS(u, v, t, gE, itr_tr);
                flag = true;
                for(vector< graphEdge>::iterator itr = gr.E.begin(); itr != gr.E.end(); ++itr){
                    if((itr -> vertex[0] != v + 1) && itr -> vertex[1] != v + 1){
                        continue;
                    }
                    if((itr -> vertex[0] == u + 1) && (itr -> vertex[1] == v + 1) || (itr -> vertex[1] == u + 1) && (itr -> vertex[0] == v + 1)){
                        updateGraphNegative(itr, t);
                    }
                    if(((itr -> vertex[0] == v + 1) || (itr -> vertex[1] == v + 1)) && ((itr -> vertex[0] != u + 1) && (itr -> vertex[1] != u + 1)) && (itr -> b >= t)){
                        Q.push(make_pair(itr -> ce, *itr));
                        //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
                    }
                }
            }
        }
        if(flag){
            tempFo.trees.push_back((*itr_tr));
            fo.trees.erase(itr_tr);
            fo.trees.push_back((*itr_tr));
            //cout << "total new cost : " << itr_tr -> ct << endl;
        }
    }
}
```

Fig. 30: Prim's algorithm function for stop function

The primMST_PLUS function for stop function that implement Prim's algorithm to find minimum spanning tree to modifies an existing tree by adding edges new valid graph edge to it. This functionality of this function is similar to primMST function, but this primMST_PLUS has to iterate all trees in the forest in order to check whether it has to add valid graph edge or not (find minimum spanning tree if it possible to reach all destination vertices).

3. primMST_PLUS_ULTRA(Forest &tempFo)

```
void Problem2::primMST_PLUS_ULTRA(Forest &tempFo){
    for(vector< Tree>::iterator itr_tr = fo.trees.begin(); itr_tr != fo.trees.end(); ++itr_tr){
        //cout << "Tree no: " << itr_tr - fo.trees.begin() + 1 << endl;
        /* clear memory */
        itr_tr -> V.clear();
        itr_tr -> E.clear();
        /* initialize */
        itr_tr -> ct = 0;
        int s = itr_tr -> s - 1;
        //cout << "source : " << s << endl;
        int id = itr_tr -> id;
        //cout << "id : " << id << endl;
        int t = find_Bandwidth(id);
        //cout << "bandwidth : " << t << endl;
        for(int i = 0; i < V; i++){
            included[i] = false;
            //cout << "included " << i << " : " << included[i] << endl;
        }
        priority_queue < pair< int, graphEdge>, vector< pair< int, graphEdge>>, PairComparator> Q;
        for(vector<graphEdge>::iterator itr_gE = gr.E.begin(); itr_gE  != gr.E.end(); itr_gE ++){
            if((itr_gE  -> vertex[0] != s + 1) && itr_gE  -> vertex[1] != s + 1){
                continue;
            }
            else if(((itr_gE  -> vertex[0] == s + 1) || (itr_gE  -> vertex[1] == s + 1)) && (itr_gE  -> b >= t)){
                Q.push(make_pair(itr_gE  -> ce, *itr_gE ));
                //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
            }
        }
        included[s] = true; // visit first source node
        //cout << "included " << s << " : " << included[s] << endl;
        itr_tr -> V.push_back(s + 1); // push first source node into the tree
        while(!Q.empty()){
            graphEdge gE = Q.top().second;
            int u = (included[gE.vertex[0] - 1] == true) ? gE.vertex[0] - 1 : gE.vertex[1] - 1;
            int v = (included[gE.vertex[0] - 1] == true) ? gE.vertex[1] - 1 : gE.vertex[0] - 1;
            Q.pop();
            if((included[u] == true) && (included[v] == true)){
                //cout << "included " << u << " : " << included[u] << " / included " << v << " : " << included[v] << endl;
                //cout << "skip\n";
                continue; // edge has been visited
            }
            else if((included[v] == false) && (gE.b >= t)){
                //cout << "valid\n";
                included[v] = true; // visit destination node
                updateTreePLUS(u, v, t, gE, itr_tr);
                for(vector< graphEdge>::iterator itr = gr.E.begin(); itr != gr.E.end(); ++itr){
                    if((itr -> vertex[0] != v + 1) && itr -> vertex[1] != v + 1){
                        continue;
                    }
                    if((itr -> vertex[0] == u + 1) && (itr -> vertex[1] == v + 1) || (itr -> vertex[1] == u + 1) && (itr -> vertex[0] == v + 1)){
                        updateGraphNegative(itr, t);
                    }
                    if(((itr -> vertex[0] == v + 1) || (itr -> vertex[1] == v + 1)) && ((itr -> vertex[0] != u + 1) && (itr -> vertex[1] != u + 1)) && (itr -> b >= t)){
                        Q.push(make_pair(itr -> ce, *itr));
                        //cout << "cost : " << itr -> ce << " / vertex[0] : " << itr -> vertex[0] << " / vertex[1] : " << itr -> vertex[1] << endl;
                    }
                }
            }
        }
        tempFo.trees.push_back((*itr_tr));
    }
}
```

Fig. 31: Prim's algorithm function for rearramge function

The primMST_PLUS_ULTRA function for rearrange function that implement Prim's algorithm to find minimum spanning tree. This functionality of this function is similar to primMST function, but this primMST_PLUS_ULTRA need to iterate all multicast trees in the forest in order to clear all of the memory of the multicast tree into the original state, and apply Prim's algorithm, then store this new multicast tree in the temporary forest and return it.

4. updateGraphPositive(vector< graphEdge>::iterator itr, int t)

```
void Problem2::updateGraphPositive(vector< graphEdge>::iterator itr, int t){
    itr -> b += t; // add remaining bandwidth to the edge
}
```

Fig. 32: Update graph by adding graph edge's remaining bandwidth

Updates the graph by adding graph edge's remaining bandwidth.

5. updateGraphNegative(vector< graphEdge>::iterator itr, int t)

```
void Problem2::updateGraphNegative(vector< graphEdge>::iterator itr, int t){
    itr -> b -= t; // deduct remaining bandwidth to the edge
}
```

Fig. 33: Updates the graph by adding graph edge's remaining bandwidth.

Updates the graph by deducting graph edge's remaining bandwidth.

6. updateTree(int u, int v, int t, graphEdge gE)

```
void Problem2::updateTree(int u, int v, int t, graphEdge gE){
    treeEdge tE;
    tE.vertex[0] = u + 1;
    tE.vertex[1] = v + 1;
    //cout << "tree vertex 0 : " << tE.vertex[0] << " / tree vertex 1 : " << tE.vertex[1] << endl;
    tr.ct += (gE.ce * t);
    //cout << "cost ++ : " << tr.ct << endl;
    tr.V.push_back(v + 1); // push destination node into tree
    tr.E.push_back(tE); // push edge into tree
}
```

Fig. 34: Update tree function for primMST

The updateTree function for primMST that update multicast tree's destination vertex, edge, total cost.

7. updateTreePLUS(int u, int v, int t, graphEdge gE, vector< Tree>::iterator itr_tr)

```
void Problem2::updateTreePLUS(int u, int v, int t, graphEdge gE, vector< Tree>::iterator itr_tr){
    treeEdge tE;
    tE.vertex[0] = u + 1;
    tE.vertex[1] = v + 1;
    //cout << "tree vertex 0 : " << tE.vertex[0] << " / tree vertex 1 : " << tE.vertex[1] << endl;
    itr_tr -> ct += (gE.ce * t);
    //cout << "cost ++ : " << tr.ct << endl;
    itr_tr -> V.push_back(v + 1); // push destination node into tree
    itr_tr -> E.push_back(tE); // push edge into tree
}
```

Fig. 35: Update tree function for primMST_PLUS and primMST_PLUS_ULTRA

The updateTree function for primMST_PLUS and primMST_PLUS_ULTRA that update multicast tree's destination vertex, edge, total cost according to iterator point to which id of the multicast tree.

8. find_erase_Bandwidth(int id)

```
int Problem2::find_erase_Bandwidth(int id){
    int bandwidth = 0;
    auto iter = find_if(bandwidth_id.begin(), bandwidth_id.end(), [id](pair<int, int> &element){return element.first == id;});
    if (iter != bandwidth_id.end()){
        bandwidth = iter->second;
        bandwidth_id.erase(iter);
        return bandwidth;
    }
    return 0;
}
```

Fig. 36: Find bandwidth and erase bandwidth according to multicast tree's id

The find_erase_Bandwidth function that find the allocated bandwidth of multicast tree according to multicast' id and erase it in bandwidth_it vectors array.

9. find_Bandwidth(int id)

```
int Problem2::find_Bandwidth(int id){
    int bandwidth = 0;
    auto iter = find_if(bandwidth_id.begin(), bandwidth_id.end(), [id](pair<int, int> &element){return element.first == id;});
    if (iter != bandwidth_id.end()){
        bandwidth = iter->second;
        return bandwidth;
    }
    return 0;
}
```

Fig. 37: Find bandwidth according to multicast tree's id

The find_Bandwidth function that find the allocated bandwidth of multicast tree according to multicast' id

10. releaseBandwidth(int id, int t)

```
void Problem2::releaseBandwidth(int id, int t){
    int num = 0;
    for(vector< Tree>::iterator itr_tr = fo.trees.begin(); itr_tr != fo.trees.end(); ++itr_tr){
        if(itr_tr -> id == id){
            for(vector< graphEdge>::iterator itr_gE = gr.E.begin(); itr_gE != gr.E.end(); ++itr_gE){
                if(num == itr_tr -> E.size()){
                    break;
                }
                for(vector< treeEdge>::iterator itr_tE = itr_tr -> E.begin(); itr_tE != itr_tr -> E.end(); ++itr_tE){
                    if(((itr_gE -> vertex[0] == itr_tE -> vertex[0]) && (itr_gE -> vertex[1] == itr_tE -> vertex[1])) ||
                    ((itr_gE -> vertex[0] == itr_tE -> vertex[1]) && (itr_gE -> vertex[1] == itr_tE -> vertex[0]))){
                        updateGraphPositive(itr_gE, t);
                        num++;
                    }
                }
            }
            //cout << "total old cost : " << itr_tr -> ct << endl;
            itr_tr -> V.clear();
            itr_tr -> E.clear();
            fo.trees.erase(itr_tr);
            fo.size--;
            break;
        }
    }
}
```

Fig. 38: Release bandwidth of graph edge

The releaseBanwidth function that iterate over all of the multicast trees in the forest to find which graph edge need to be update. If it updates all of the graph edge, then its clear memory and erase multicast tree according to the stop id.

11. resetGraph()

```
void Problem2::resetGraph(){
    for(vector< graphEdge>::iterator itr_gE = gr.E.begin(); itr_gE != gr.E.end(); ++itr_gE){
        itr_gE -> b = itr_gE -> be;
    }
}
```

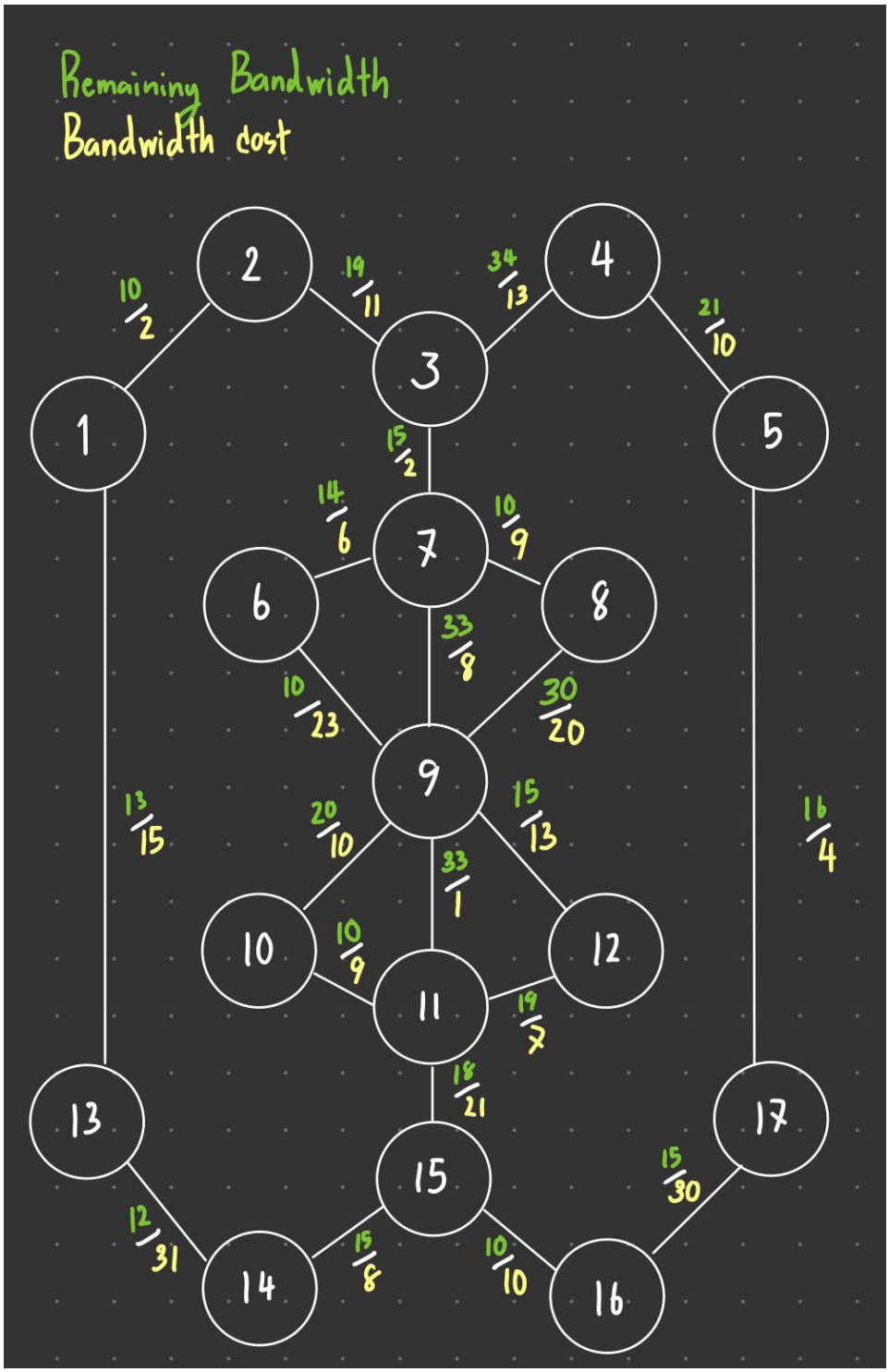
Fig. 39: Reset Graph remaining bandwidth to original state

The resetGraph function that iterate to all over the graph edge to reset remaining bandwidth to the original state.

Ratchanon Hsu (許如來)
111006261

Test Case Design

Test Case Design for Problem 1



Ratchanon Hsu (許如來)
111006261

17 22

1 2 10 2

1 13 13 15

2 3 19 11

3 4 34 13

3 7 15 2

4 5 21 10

5 17 16 4

6 7 14 6

6 9 10 23

7 8 10 9

7 9 33 8

8 9 30 20

9 10 20 10

9 11 33 1

9 12 15 13

10 11 10 9

11 12 19 7

11 15 18 21

13 14 12 31

14 15 15 8

15 16 10 10

16 17 15 30

insert 4 9 { 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 } 10

insert 3 3 { 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 } 5

insert 5 13 { 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 } 3

stop 3

insert 7 8 { 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 } 2

rearrange

Explanation

This test case contains 17 vertices and 22 edge with 4 insert function call with not arrange order of id, 1 stop function call and 1 rearrange.

Ratchanon Hsu (許如來)
111006261

Test Case Design for Problem 2

4 4

1 2 5 1

1 3 3 2

1 4 4 3

3 4 2 4

insert 1 1 { 1 2 3 4 } 2

insert 2 1 { 1 2 3 } 2

insert 3 1 { 1 2 3 4 } 2

stop 2

rearrange