Programming Assignment #3

B11901029 李致頡

Code Explanation

1. Class definition

   We construct a class `DisjointSet` for Kruskal's algorithm.

```
class DisjointSet {
    public:
        DisjointSet(int V);
        int Find(int x);
        void Union(int x, int y);

    private:
        vector<int> parent;
        vector<int> rank;
};
```

   And a class `CBSolver` for solving the problems and saving the graph.

```
class CBSolver {
    public:
        int V, E;
        int minRemoveWeight = 0;
        int removeWeight;
        int removeWeightLib;
        bool directed;
        int positiveCount;
        int negativeCount;
        vector<Edge2> edgeList;
        vector<vector<Edge>> adjList;
        vector<vector<Edge>> adjListLib;    // directed only
        vector<Edge2> minRemoveEdges;
        vector<Edge2> removeEdges;          // directed only
        vector<Edge2> removeEdgesLib;       // directed only

        CBSolver(int numV, int numE, bool graphtype) : V(numV), E(numE), directed(graphtype), adjList(vector<vector<Edge>>(numV)), color(vector<Color>(numV)) {}
        void MaximumSpanningTree();
        void AddbackEdge(int start);        // directed only

    private:
        vector<Color> color;
        bool DFSdetectCycle();          // directed only
        bool CycleFound(int i);         // directed only
};
```

And some structures to save nodes and edges:

```
enum Color {WHITE, GRAY, BLACK};

struct Edge {
    int v;
    short w;
};

struct Edge2 {
    int u, v;
    short w;
};
```

2. Store the graph

```
char graphtype;
inFile >> graphtype;

int numV, numE;
inFile >> numV >> numE;
CBSolver graph = CBSolver(numV, numE, (graphtype == 'd') ? true : false);

// Read in edges, add to adjacency list
for (int i = 0; i < graph.E; ++i) {
    int u, v;
    short w;
    inFile >> u >> v >> w;
    graph.edgeList.push_back({u, v, w});
    if (graph.directed) {
        graph.adjList[u].push_back({v, w});
    }
}
```

We store the graph by a list of edge for the convenience of Kruskal's algorithm. For directed graphs, we construct another adjacency list representation by STL vector. `adjList` is a vector of distinct-length vectors, where an Edge $u \to v$ is saved in the vector of `adjList[u]`.

3. Solve undirected cycle breaking problem: by maximum spanning tree.

We've been taught how to solve minimum spanning tree problem in class. We can do it by Kruskal's algorithm. To make the graph acyclic by removing edges with minimum total weight is basically the same problem as finding maximum spanning tree (an acyclic graph that contains as much weight as possible). Those edges not in this tree are the set of edges we want: adding them back to the graph will result in a cycle, so they must be removed, and their weights are also smallest.

So simply solve it by Kruskal's algorithm with the conditions reversed.

```cpp
void CBSolver::MaximumSpanningTree() {
    // Sort edges by weight in descending order
    sort(edgeList.begin(), edgeList.end(), [](Edge2 a, Edge2 b) {
        return a.w > b.w;
    });
    // Kruskal's Algorithm
    DisjointSet set(V);
    for (int i = 0; i < E; ++i) {
        Edge2 e = edgeList[i];
        if(set.Find(e.u) != set.Find(e.v)) {
            set.Union(e.u, e.v);
        }
        else {
            minRemoveEdges.push_back(e);
            minRemoveWeight += e.w;
        }
    }
    if(directed) {
        // Remove edges from adjacency list
        for (int i = 0; i < (int)minRemoveEdges.size(); ++i) {
                Edge2 e = minRemoveEdges[i];
                for(int j = 0; j < (int)adjList[e.u].size(); ++j) {
                    if(adjList[e.u][j].v == e.v) {
                        adjList[e.u].erase(adjList[e.u].begin() + j);
                        break;
                    }
                }
            }
        // Count positive and negative edges
        int i = 0;
        while (minRemoveEdges[i].w > 0 && i < (int)minRemoveEdges.size()) {
            i++;
        }
        positiveCount = i;
        negativeCount = (int)minRemoveEdges.size() - i;
    }
}
```

The time complexity of this algorithm is $O(E \lg V)$ given by the textbook.

4. Solve directed cycle breaking problem: by greedy.
   1. Treat all edges as undirected, perform Kruskal's algorithm to find an edge combination that must contain no cycle.
   2. Since edges are directed, we've actually removed too many edges, so try to add positive edges back in descending order of weights to reduce the total cost of removing.
   3. Use DFS to check if adding back an edge cause a cycle. If yes, don't add back.

   Time complexity: normal DFS cost $O(V+E)$. We try to add back every edge we've picked out to remove, so the cost is $O\big(E(V+E)\big)$.

```cpp
void CBSolver::AddbackEdge(int start) {
    // Add back positive edges
    for (int i = start; i < (int)removeEdges.size() - negativeCount; ++i) {
        Edge2 e = removeEdges[i];
        adjList[e.u].push_back({e.v, e.w});
        // Check for cycle
        if (DFSdetectCycle()) {
            adjList[e.u].pop_back();
        }
        else {
            removeEdges.erase(removeEdges.begin() + i);
            --i;
            removeWeight -= e.w;
        }
    }
    for(int i = 0; i < start; ++i) {
        Edge2 e = removeEdges[i];
        adjList[e.u].push_back({e.v, e.w});
        // Check for cycle
        if (DFSdetectCycle()) {
            adjList[e.u].pop_back();
        }
        else {
            removeEdges.erase(removeEdges.begin() + i);
            --i;
            --start;
            removeWeight -= e.w;
        }
    }
}

bool CBSolver::DFSdetectCycle() {
    // Reset for DFS
    for (int i = 0; i < V; ++i) {
        color[i] = WHITE;
    }
    // DFS: See if cycle exists
    for (int i = 0; i < V; ++i) {
        if (color[i] == WHITE && CycleFound(i)) {
            return true;
        }
    }
    return false;
}

bool CBSolver::CycleFound(int i) {
    color[i] = GRAY;
    for (int j = 0; j < (int)adjList[i].size(); ++j) {
        Edge e = adjList[i][j];
        if (color[e.v] == GRAY) {
            return true;
        }
        else if (color[e.v] == WHITE && CycleFound(e.v)) {
            return true;
        }
    }
    color[i] = BLACK;
    return false;
}
```

This is a heuristic algorithm. We can try to start adding back from different edge to see if it gives a better solution, so I use every edge to remove as a start. Since there's a runtime limit 1 min, I set a timer to stop if runtime limit approaches.

```
if(graph.directed) {
    graph.adjListLib = graph.adjList;
    graph.removeWeightLib = graph.minRemoveWeight;
    graph.removeWeight = graph.minRemoveWeight;
    graph.removeEdges = graph.minRemoveEdges;
    graph.removeEdgesLib = graph.minRemoveEdges;
    int loopCount = 0;
    auto timelimit = milliseconds(55000).count();
    auto enterStart = steady_clock::now();
    // Keep finding better solution
    while (loopCount < graph.positiveCount*2) {
        auto now = steady_clock::now();
        auto enterElapsed = duration_cast<milliseconds>(now - enterStart).count();
        auto totalElapsed = duration_cast<milliseconds>(now - totoalStart).count();
        // Stop if time is almost up
        if(loopCount != 0 && (enterElapsed / loopCount) * 8 > timelimit - totalElapsed) {
            break;
        }
        // Scheme of selecting the vertex to start from
        graph.AddbackEdge(loopCount < graph.positiveCount ? loopCount : loopCount - graph.positiveCount);
        if(loopCount == graph.positiveCount - 1) {
            sort(graph.removeEdgesLib.begin(), graph.removeEdgesLib.end(), [](Edge2 a, Edge2 b) {
                return a.w > b.w;
            });
        }
        // If the current graph has less remove weight, update min
        if (graph.removeWeight < graph.minRemoveWeight) {
            graph.minRemoveEdges = graph.removeEdges;
            graph.minRemoveWeight = graph.removeWeight;
        }
        // Reset graph to original state
        graph.adjList = graph.adjListLib;
        graph.removeWeight = graph.removeWeightLib;
        graph.removeEdges = graph.removeEdgesLib;
        loopCount++;
    }
}
```

Result

Input File	Total Weight
public_case_0.in	5
public_case_1.in	21
public_case_2.in	-3330
public_case_3.in	-21680
public_case_4.in	0
public_case_7.in	-11492
public_case_8.in	-71075