REPORT

Project 2: Graph Algorithms

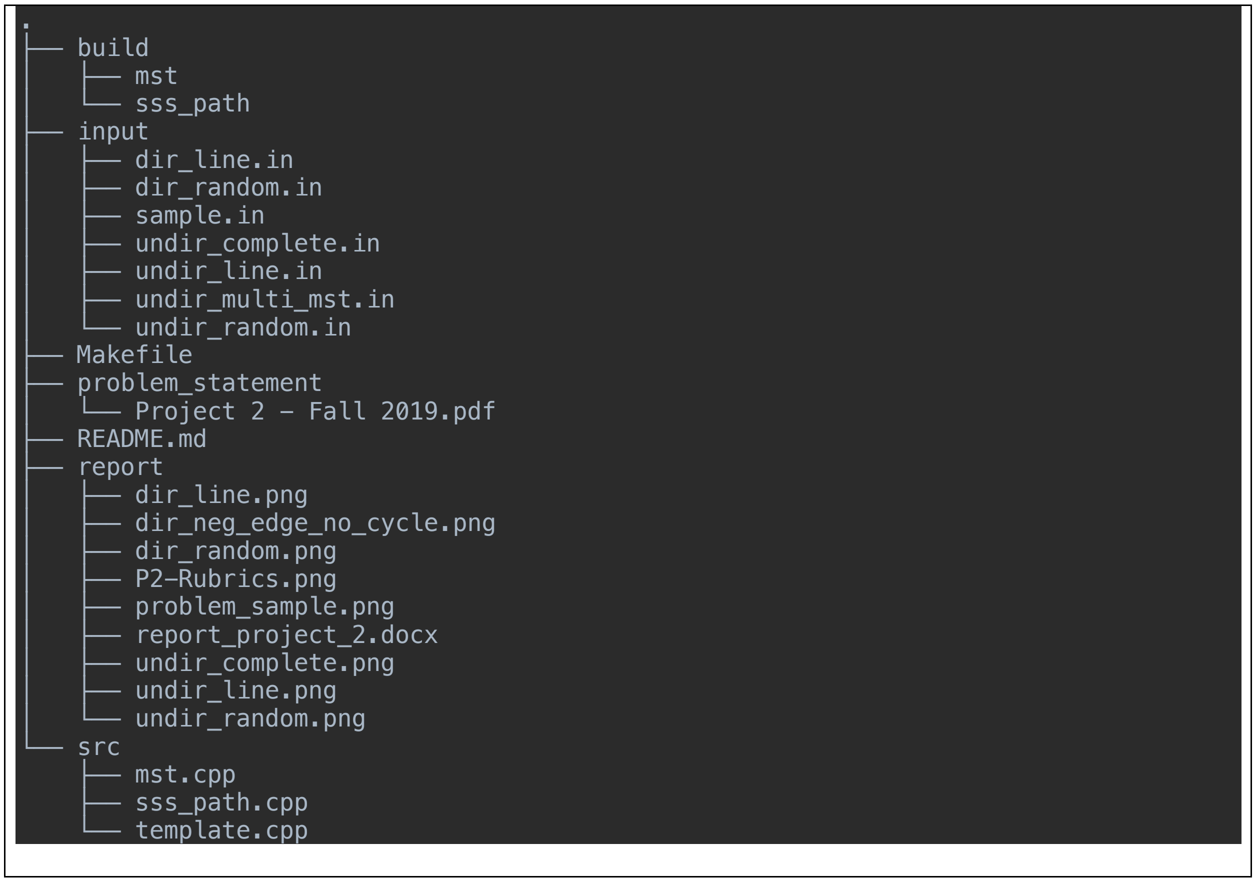
Single-source shortest path algorithm and Minimum Spanning Tree (MST)

Submitted By

Abdullah Al Raqibul Islam (ID# 801151189)

Code Repository Overview

I have used C++ as the programming language for this project. The project structure looks like this



```
.
├── build
│   ├── mst
│   └── sss_path
├── input
│   ├── dir_line.in
│   ├── dir_random.in
│   ├── sample.in
│   ├── undir_complete.in
│   ├── undir_line.in
│   ├── undir_multi_mst.in
│   └── undir_random.in
├── Makefile
├── problem_statement
│   └── Project 2 - Fall 2019.pdf
├── README.md
├── report
│   ├── dir_line.png
│   ├── dir_neg_edge_no_cycle.png
│   ├── dir_random.png
│   ├── P2-Rubrics.png
│   ├── problem_sample.png
│   ├── report_project_2.docx
│   ├── undir_complete.png
│   ├── undir_line.png
│   └── undir_random.png
└── src
    ├── mst.cpp
    ├── sss_path.cpp
    └── template.cpp
```

Here I am going to give brief description of the purpose of the sub-directories and files of this project:

- /build directory contains the executable program files
- /input directory contains the sample graph inputs
- Makefile will help compiling the project programs and place output executable files in the /build directory
- /problem_statement holds the problem statement of this project
- README.md contains instructions to build the project and way of running the programs to test further
- /report contains the project reports and helper files for it
- /src holds the programs to solve the problems of this project

Note

For runtime analysis I have used standard notations, i.e. B will represent number of nodes and **E** will represent number of edges.

Project Run Instruction

```
# go to project directory
$ cd project_2
# build the project
$ make
# command to run solution for problem 1
#  general  instruction:  $  ./build/sss_path  <  input_file  >
output_file
# here is a sample
$ ./build/sss_path < input/sample.in > sample_sssp.out
# command to run solution for problem 2
# general instruction: $ ./build/mst < input_file > output_file
# here is a sample
$ ./build/mst < input/sample.in > sample_mst.out
```

Problem 1: Single Source Shortest Path

Short description: In this task, I have implemented Dijkstra's algorithm for finding the shortest paths from a source node to all the other nodes of the graph. This algorithm will work on both directed and undirected weighted graph. The limitation of this algorithm is that, it will not work on graph contains negative edges.

Dijkstra's algorithm applies greedy strategy to find shortest path from a single source node. Dijkstra's algorithm use priority queue to decide (based on the already discovered path-cost) which node to be considered to relax the shortest path from the source node, is typically considered as a greedy approach. By nature, single source shortest path problem has satisfactory optimization substructure since if node A is connected to node B, node B is connected to node C, and the path must go through node A and node B to reach the destination node C, then the shortest path from node A to node B and the shortest path from node B to node C must be a part of the shortest path from node A to node C. So, the optimal answers from the subproblems do contribute to the optimal answer for the total problem.

Data-structure: In my solution, I have used priority queue from C++ Standard Template Library with custom comparator, where it will cost O(log n) to push/pop an item into/from it (if the total number of elements we push to the priority queue is n). To store graph, I used adjacency list implemented by array of C++ vector. I have used array to store other supporting data, i.e. graph node visited marking, distance of shortest paths from the sources, and parent information to reconstruct shortest paths, etc.

Runtime analysis: To read input, we need cost E. The Dijkstra's algorithm needs standard cost [(V+E) x logV]. The path construction will cost (V x V). Overall the runtime will be O(E x logV).

Judgements for test I/O: I have chosen the following directed and undirected weighted graphs to test my solution for this problem,

- Directed line graph (Fig 1)
  - Line graph is always interesting to test single source shortest path algorithm.
- Directed random graph (Fig 5)
  - To test the behavior of the implementation in a randomly generated graph.
- Undirected complete graph (Fig 4)
  - To test the implementation will perform worst in complete graph.
- Sample input given in the problem statement (Fig: 6)
  - To test the behavior of the implementation in the sample graph provided in the problem statement.

Sample I/O:

Sample Input	Sample Output
10 9 D A B 2 B C 2 C D 2 D E 2 E F 2	from [A] to [A], min weight to reach: 0 path: A from [A] to [B], min weight to reach: 2

F G 2 G H 2 H I 2 I J 2 A	path: A -> B from [A] to [C], min weight to reach: 4 path: A -> B -> C from [A] to [D], min weight to reach: 6 path: A -> B -> C -> D from [A] to [E], min weight to reach: 8 path: A -> B -> C -> D -> E from [A] to [F], min weight to reach: 10 path: A -> B -> C -> D -> E -> F from [A] to [G], min weight to reach: 12 path: A -> B -> C -> D -> E -> F -> G from [A] to [H], min weight to reach: 14 path: A -> B -> C -> D -> E -> F -> G -> H from [A] to [I], min weight to reach: 16 path: A -> B -> C -> D -> E -> F -> G -> H -> I from [A] to [J], min weight to reach: 18 path: A -> B -> C -> D -> E -> F -> G -> H -> I -> J
10 14 D A B 4 A H 8 B H 11 B C 8 H I 7 H G 1 C I 2 I G 6 C D 7 C F 4 G F 2 D F 14 D E 9 E F 10 A	from [A] to [A], min weight to reach: 0 path: A from [A] to [B], min weight to reach: 4 path: A -> B from [A] to [C], min weight to reach: 12 path: A -> B -> C from [A] to [D], min weight to reach: 19 path: A -> B -> C -> D from [A] to [E], min weight to reach: 28 path: A -> B -> C -> D -> E from [A] to [F], min weight to reach: 11 path: A -> H -> G -> F

	from [A] to [G], min weight to reach: 9 path: A -> H -> G from [A] to [H], min weight to reach: 8 path: A -> H from [A] to [I], min weight to reach: 14 path: A -> B -> C -> I from [A] to [J], min weight to reach: unreachable
7 21 U A B 2 A C 4 A D 5 A E 7 A F 9 A G 12 B C 7 B D 13 B E 16 B F 5 B G 3 C D 9 C E 11 C F 8 C G 7 D E 5 D F 3 D G 7 E F 9 E G 14 F G 19 A	from [A] to [A], min weight to reach: 0 path: A from [A] to [B], min weight to reach: 2 path: A -> B from [A] to [C], min weight to reach: 4 path: A -> C from [A] to [D], min weight to reach: 5 path: A -> D from [A] to [E], min weight to reach: 7 path: A -> E from [A] to [F], min weight to reach: 7 path: A -> B -> F from [A] to [G], min weight to reach: 5 path: A -> B -> G
6 10 U A B 1 A C 2 B C 1 B D 3 B E 2 C D 1 C E 2 D E 4 D F 3	from [A] to [A], min weight to reach: 0 path: A from [A] to [B], min weight to reach: 1 path: A -> B from [A] to [C], min weight to reach: 2 path: A -> C from [A] to [D], min weight to reach: 3

E F 3 A	path: A -> C -> D from [A] to [E], min weight to reach: 3 path: A -> B -> E from [A] to [F], min weight to reach: 6 path: A -> B -> E -> F

Sample Graphs

Here is the graphical demonstration of my sample input graphs,

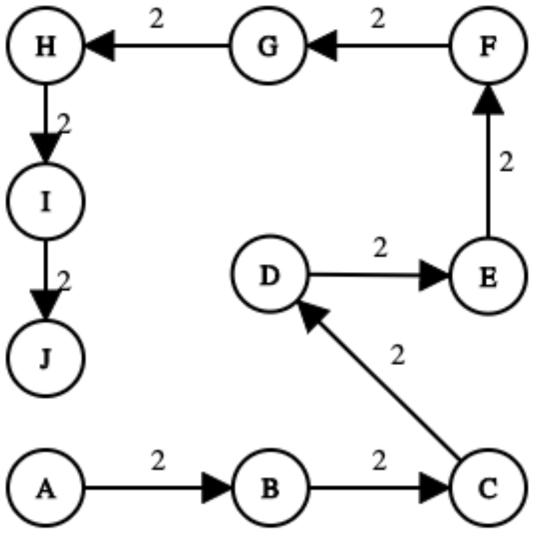


Fig 1: Directed line graph

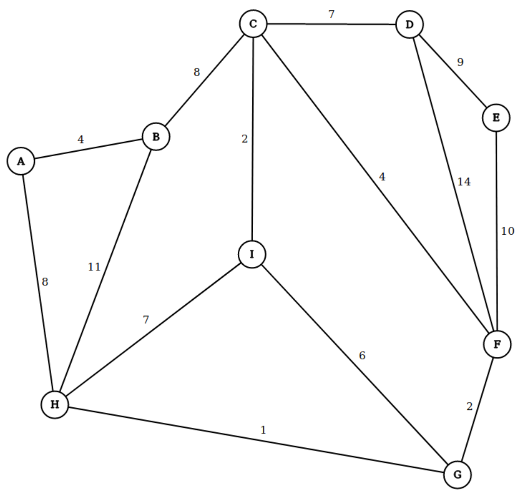


Fig 2: Undirected random graph

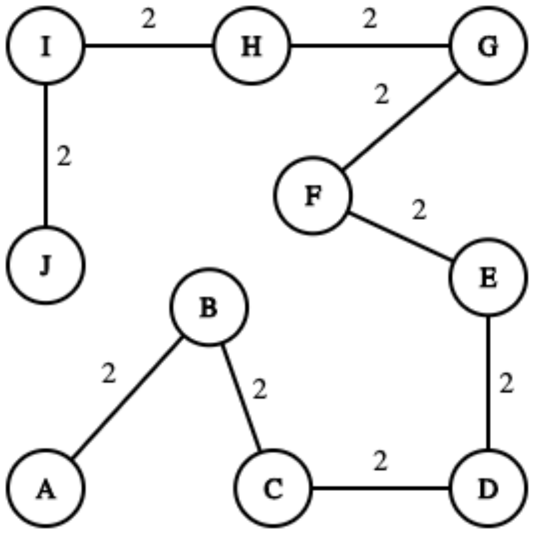


Fig 3: Undirected line graph

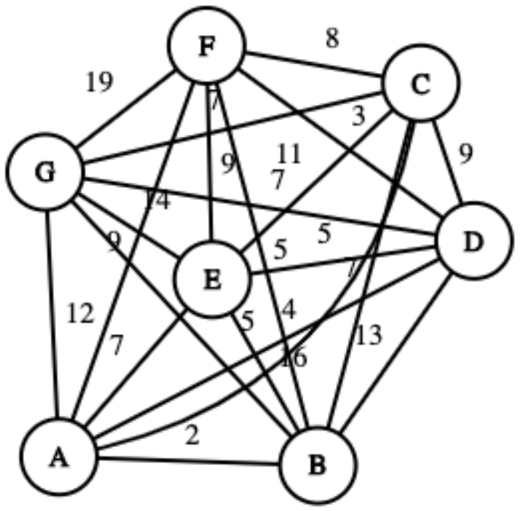


Fig 4: Undirected complete graph

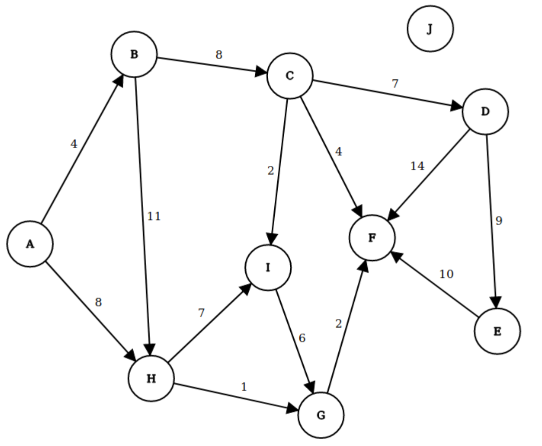


Fig 5: Directed random graph

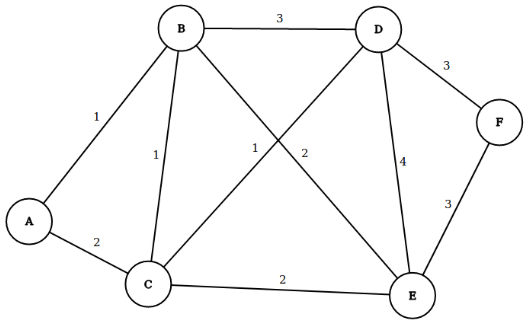


Fig 6: Sample graph given in the problem statement

Problem 2: Minimum Spanning Tree (MST)

Short description: For this task, I have implemented Kruskal's algorithm to find minimum spanning tree of an undirected graph. This algorithm will only work on undirected graph. The working principal of Kruskal's algorithm is to choose the edge with minimum weight (greedy approach) that connects any two trees in the forest.

Eventually it will become a single tree (only if it is a connected graph) which connects all the nodes of the graph.

Data-structure: I have used disjoint set data-structure to track the set-id of a node. Initially every node assigned to his own set-id. When we found an edge, which connects two nodes from two separate sets, we make a connection between them by applying Union operation. I applied two technique called **path-compression** in find and **union-by-rank** to reduce the time complexity of each operation done on disjoint set data-structure. This two techniques actually complement each other and make the amortized time complexity even smaller than O(log n) and becomes constant.

In my solution I used edge list (with the support of vector from C++ Standard Template Library) to store the edge information. To sort the edge list by weight, I use sort function provided by C++ standard library which requires O(E x log(E)) in the worst case scenario (to sort E edges).

Runtime analysis: To read input, we need cost E. In Kruskal's algorithm,

- Initialize the disjoint-set costs V
- Edge list shorting costs [E x log(E)]
- (2 x E) find_set call with total cost approximately c x (2 x E), where c is a constant
- V union_by_rank call with total cost approximately c x V, where c is a constant

Overall running time O(E x log E)

Judgements for test I/O: I have chosen the following connected, undirected, weighted graphs to test my solution for this problem,

- Undirected complete graph (Fig 4)
    - Runtime should be the worst for this graph, as we will need to sort more edges for this graph.
- Undirected line graph (Fig 3)
    - Execution time should be fastest for this graph.
- Undirected graph with multiple MSTs (Fig 6)
    - To test the behavior of the MST implementation in graph which has more than one MST.
- Undirected random graph (Fig 2)

- To test the behavior of the MST implementation in randomly generated graph.

Sample I/O:

Sample Input	Sample Output
7 21 U A B 2 A C 4 A D 5 A E 7 A F 9 A G 12 B C 7 B D 13 B E 16 B F 5 B G 3 C D 9 C E 11 C F 8 C G 7 D E 5 D F 3 D G 7 E F 9 E G 14 F G 19 A	Cost for building mst: 22 A B -> 2 B G -> 3 D F -> 3 A C -> 4 A D -> 5 D E -> 5
10 9 U A B 2 B C 2 C D 2 D E 2 E F 2 F G 2 G H 2 H I 2 I J 2 A	Cost for building mst: 18 A B -> 2 B C -> 2 C D -> 2 D E -> 2 E F -> 2 F G -> 2 G H -> 2 H I -> 2 I J -> 2

6 10 U A B 1 A C 2 B C 1 B D 3 B E 2 C D 1 C E 2 D E 4 D F 3 E F 3 A	Cost for building mst: 8 A B -> 1 B C -> 1 C D -> 1 B E -> 2 D F -> 3
9 14 U A B 4 A H 8 B H 11 B C 8 H I 7 H G 1 C I 2 I G 6 C D 7 C F 4 G F 2 D F 14 D E 9 E F 10 A	Cost for building mst: 37 H G -> 1 C I -> 2 G F -> 2 A B -> 4 C F -> 4 C D -> 7 A H -> 8 D E -> 9

Test platform

- Processor: Intel(R) Xeon(R) CPU E5-2620 2.00GHz (12 Core)
- Linux version: 5.0.0-27-generic
- g++ version: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0

References

- Online graph drawing tool: https://csacademy.com/app/graph_editor/

Code

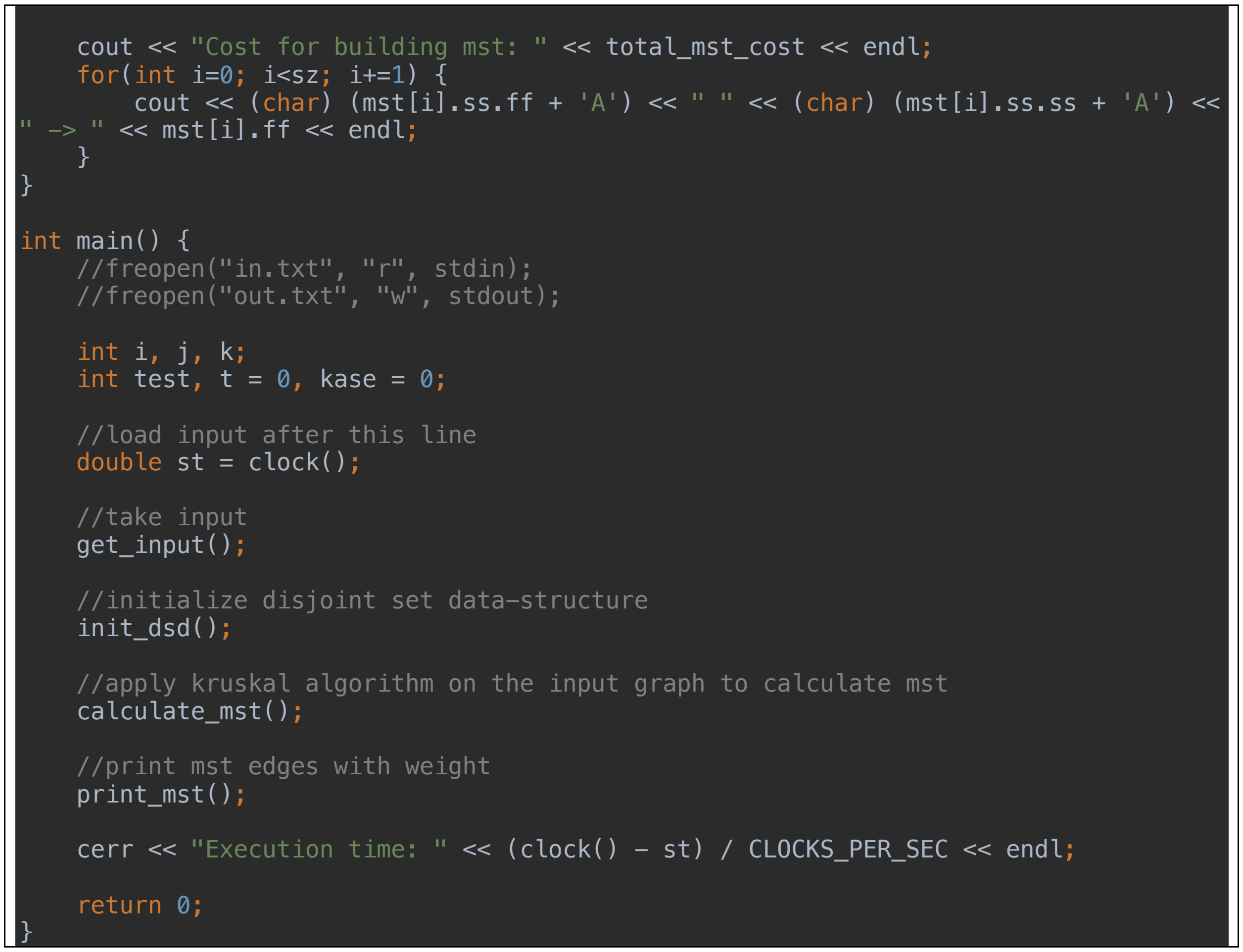
Program for problem 1:

```
#define ff first
#define ss second

const int inf = (1 << 28);
const int MAX = 1005;

struct comp {
    bool operator() (const pii &a, const pii &b) {
        return a.ss > b.ss;
    }
};

priority_queue< pii, vector< pii >, comp > pq;
vector<pii> graph[MAX];
int dist[MAX], nodes, edges, st, parent[MAX];
bool vis[MAX], is_directed;

void get_input() {
    string line, dir, u_str, v_str;
    int u, v, w;

    getline(cin, line);
    stringstream ss(line);
    ss >> nodes;
    ss >> edges;
    ss >> dir;

    if(dir[0] == 'U') is_directed = false;
    else is_directed = true;

    for(int i=0; i<edges; i+=1) {
        getline(cin, line);
        stringstream ss(line);
        ss >> u_str;
        ss >> v_str;
        ss >> w;
        u = u_str[0] - 'A';
        v = v_str[0] - 'A';

        graph[u].push_back(pii(v, w));
        if(!is_directed) graph[v].push_back(pii(u, w));
    }

    getline(cin, line);
    stringstream ss_source(line);
    ss_source >> u_str;
    st = u_str[0] - 'A';

    /*for(int i=0; i<nodes; i+=1) {
        int sz = graph[i].size();
        for(int j=0; j<sz; j+=1) {
            cout << i << " " << graph[i][j].ff << " " << graph[i][j].ss << endl;
        }
    }
    cout << "starting node: " << st << endl;*/
}
```

```
void run_dijkstra () {
    int u, v, w, sz;
    for(int i=0; i<nodes; i+=1) {
        dist[i] = inf;
        vis[i] = false;
    }
    dist[st] = 0;
    parent[st] = -1;
    pq.push(pii(st, 0));

    while(!pq.empty()) {
        u = pq.top().ff;
        pq.pop();
        if(vis[u]) continue;
        sz = graph[u].size();
        for(int i=0; i<sz; i+=1) {
            v = graph[u][i].ff;
            w = graph[u][i].ss;
            if(!vis[v] && dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                parent[v] = u;
                pq.push(pii(v, dist[v]));
            }
        }
        vis[u] = true;
    }
}

void print_path_recursive(int u) {
    // base case!
    if(parent[u] == -1) return;
    print_path_recursive(parent[u]);
    cout << " -> " << (char) (u + 'A');
}

void print_shortest_paths() {
    for(int i=0; i<nodes; i++) {
        cout << "from [" << (char) (st + 'A') << "] to [" << (char) (i + 'A') << "],
min weight to reach: ";

        if(dist[i] >= inf) {
            cout << "unreachable" << endl;
        }
        else {
            cout << dist[i] << endl;
            cout << "path: " << (char) (st + 'A');
            print_path_recursive(i);
            cout << endl;
        }
    }
}

int main() {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;

    //load input after this line
    double st = clock();

    get_input();
    run_dijkstra();
    print_shortest_paths();

    cerr << "Execution time: " << (clock() - st) / CLOCKS_PER_SEC << endl;

    return 0;
}
```

Program for problem 2:

```
#define ff first
#define ss second

#define pii pair< int, int >

const int inf = (1 << 28);
const int MAX = 1005;

struct disjoint_set {
    int parent, rank;
};

int total_mst_cost, nodes, edges;
bool is_directed;

vector< pair< int, pii > > edge_list, mst;
struct disjoint_set dset[MAX];

void init_dsd() {
    for(int i=0; i<nodes; i+=1) {
        dset[i].parent = i;
        dset[i].rank = 0;
    }
}

void get_input() {
    string line, dir, u_str, v_str;
    int u, v, w;

    getline(cin, line);
    stringstream ss(line);
    ss >> nodes;
    ss >> edges;
    ss >> dir;

    if(dir[0] == 'U') is_directed = false;
    else is_directed = true;

    for(int i=0; i<edges; i+=1) {
        getline(cin, line);
        stringstream ss(line);
        ss >> u_str;
        ss >> v_str;
        ss >> w;
        u = u_str[0] - 'A';
        v = v_str[0] - 'A';
        edge_list.push_back(make_pair(w, pii(u, v)));
    }

    if(is_directed) {
        cout << "Kruskal's algorithm for finding MST can't work on directed graph."
<< endl;
        exit(0);
    }

    /*for(int i=0; i<edges; i+=1) {
        cout << edge_list[i].ss.ff << " " << edge_list[i].ss.ss << " " <<
edge_list[i].ff << endl;
    }*/
}

/*find root of the disjoint set with path compression*/
int find_set(int x) {
    if(x != dset[x].parent) dset[x].parent = find_set(dset[x].parent);
    return dset[x].parent;
}

/*union disjoint sets by rank*/
void union_by_rank(int px, int py) {
    if(dset[px].rank < dset[py].rank) {
        dset[px].parent = py;
    }
    else if(dset[px].rank > dset[py].rank) {
        dset[py].parent = px;
    }
    else {
        dset[py].parent = px;
        dset[px].rank += 1;
    }
}

void calculate_mst() {
    int u, v, pu, pv;

    //sort edges by increasing order
    sort(edge_list.begin(), edge_list.end());

    for(int i=0; i<edges; i+=1) {
        u = edge_list[i].ss.ff;
        v = edge_list[i].ss.ss;

        pu = find_set(u);
        pv = find_set(v);

        /*u and v are not in same set, connect them by this edge and add a link
among them*/
        if(pu != pv) {
            mst.push_back(edge_list[i]);
            total_mst_cost += edge_list[i].ff;
            union_by_rank(pu, pv);
        }
    }
}

void print_mst() {
    int sz = mst.size();
```



```
    cout << "Cost for building mst: " << total_mst_cost << endl;
    for(int i=0; i<sz; i+=1) {
        cout << (char) (mst[i].ss.ff + 'A') << " " << (char) (mst[i].ss.ss + 'A') <<
" -> " << mst[i].ff << endl;
    }
}

int main() {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;

    //load input after this line
    double st = clock();

    //take input
    get_input();

    //initialize disjoint set data-structure
    init_dsd();

    //apply kruskal algorithm on the input graph to calculate mst
    calculate_mst();

    //print mst edges with weight
    print_mst();

    cerr << "Execution time: " << (clock() - st) / CLOCKS_PER_SEC << endl;

    return 0;
}
```