**ET3272: Design and Analysis of Algorithms**

**Name of the student: Aditya Jannawar Roll No.: 02**

**Div.: A Batch: 1**

**Date of performance: /0 /2023**

**Experiment No. 16**

# Title: Kruskal Mst

**Theory/Description of the Problem Statement:**

A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree

In Kruskal’s algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first at the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution.

**Algorithm :**

* Create a set mstSet that keeps track of vertices already included in MST.
* Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.
* While mstSet doesn’t include all vertices
* Pick a vertex u that is not there in mstSet and has a minimum key value.
* Include u in the mstSet.
* Update the key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices.
* For every adjacent vertex v, if the weight of edge u-v is less than the previous key value of v, update the key value as the weight of u-v.

**Pseudo Code :**

* Prim(G, w, r):
* for each u ∈ V[G]
* key[u] ← ∞ // Initialize key values to infinity
* parent[u] ← NIL // Initialize parent values to NIL
* key[r] ← 0 // Make the key value of root node to 0
* Q ← V[G] // Add all vertices to a min-priority queue
* while Q is not empty
* u ← Extract-Min(Q) // Extract the vertex with the minimum key value
* for each v ∈ Adj[u]
* if v ∈ Q and w(u, v) < key[v]
* parent[v] ← u
* key[v] ← w(u, v) // Update the key value of vertex v in Q
* Decrease-Key(Q, v, key[v])
* return parent // Return the set of edges in MST

**Analysis of the Algorithm**

**Time Complexity:**

The time complexity of Kruskal's algorithm is O(ElogE), where E is the number of edges in the graph, as it involves sorting of all edges

**Space Complexity:**

The space complexity of the algorithm is O(V+E), where V is the number of vertices and E is the number of edges, as it uses an additional data structure, i.e., Disjoint Set Union (DSU) to keep track of connected components.

**Experiment and result:**

Code:

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// DSU data structure

// path compression + rank by union

class DSU {

    int\* parent;

    int\* rank;

public:

    DSU(int n)

    {

        parent = new int[n];

        rank = new int[n];

        for (int i = 0; i < n; i++) {

            parent[i] = -1;

            rank[i] = 1;

        }

    }

    // Find function

    int find(int i)

    {

        if (parent[i] == -1)

            return i;

        return parent[i] = find(parent[i]);

    }

    // Union function

    void unite(int x, int y)

    {

        int s1 = find(x);

        int s2 = find(y);

        if (s1 != s2) {

            if (rank[s1] < rank[s2]) {

                parent[s1] = s2;

            }

            else if (rank[s1] > rank[s2]) {

                parent[s2] = s1;

            }

            else {

                parent[s2] = s1;

                rank[s1] += 1;

            }

        }

    }

};

class Graph {

    vector<vector<int> > edgelist;

    int V;

public:

    Graph(int V) { this->V = V; }

    // Function to add edge in a graph

    void addEdge(int x, int y, int w)

    {

        edgelist.push\_back({ w, x, y });

    }

    void kruskals\_mst()

    {

        // Sort all edges

        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU

        DSU s(V);

        int ans = 0;

        cout << "Following are the edges in the "

                "constructed MST"

            << endl;

        for (auto edge : edgelist) {

            int w = edge[0];

            int x = edge[1];

            int y = edge[2];

            // Take this edge in MST if it does

            // not forms a cycle

            if (s.find(x) != s.find(y)) {

                s.unite(x, y);

                ans += w;

                cout << x << " -- " << y << " == " << w

                    << endl;

            }

        }

        cout << "Minimum Cost Spanning Tree: " << ans;

    }

};

// Driver code

int main()

{

    Graph g(4);

    g.addEdge(0, 1, 10);

    g.addEdge(1, 3, 15);

    g.addEdge(2, 3, 4);

    g.addEdge(2, 0, 6);

    g.addEdge(0, 3, 5);

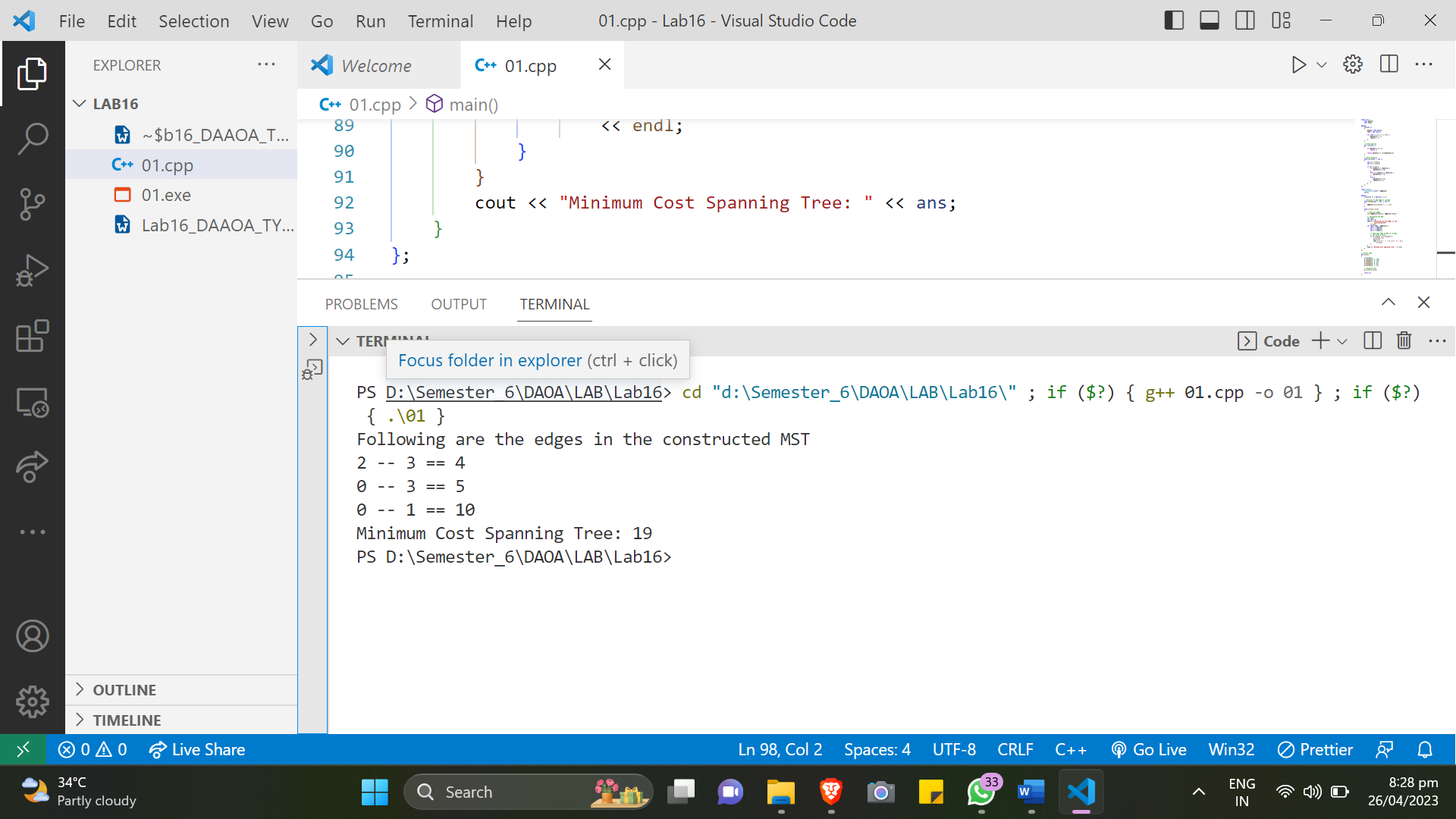
    // Function call

    g.kruskals\_mst();

    return 0;

}

Output:



**Conclusions:**

Kruskal's algorithm is an efficient algorithm to find the Minimum Spanning Tree (MST) of a given undirected, weighted graph. It has a time complexity of O(ElogE) and a space complexity of O(V+E). It uses the DSU data structure to keep track of connected components and forms the MST by selecting edges in a sorted order and checking for cycles using the DSU. Overall, Kruskal's algorithm is a good choice for finding the MST of a sparse graph with a large number of edges.