# ASSIGNMENT NO. 4

**TITLE**: Minimum Spanning Tree (MST) using Prim’s and Kruskal’s Algorithms

**AIM**:

Given an undirected weighted graph with nodes and edges, find the total weight of the Minimum Spanning Tree (MST) and construct one specific MST using Prim’s or Kruskal’s algorithm. Note that there may be multiple possible MSTs, and the goal is to construct any one of them.

# Relevant Course Objectives:

* To understand the concept of Minimum Spanning Trees (MST) in undirected weighted graphs.
* To apply Prim’s and Kruskal’s algorithms for finding MSTs.
* To develop an understanding of graph traversal techniques and their applications in optimization problems.

# Theory:

* **Undirected Weighted Graphs**: In an undirected graph, edges have no direction, meaning that if there is an edge between two nodes, it can be traversed in both directions. When edges

are assigned weights, it indicates the cost, distance, or time associated with traveling between the two nodes.

* **Minimum Spanning Tree (MST)**: The MST of a graph is a subset of its edges that connects all the vertices together, without any cycles, and with the minimum possible total edge weight.

An MST is useful in various applications such as network design, minimizing costs, and optimizing paths.

# Kruskal’s Algorithm:

* + Kruskal’s algorithm is a greedy algorithm that finds an MST by sorting all edges by weight and adding the smallest edge to the MST, ensuring no cycles are formed. The algorithm uses the **Disjoint Set (Union-Find)** data structure to detect cycles.

# Steps of Kruskal’s Algorithm:

* + 1. Sort all the edges in increasing order of their weights.
    2. Pick the smallest edge. If it doesn’t form a cycle with the spanning tree, add it to the MST.
    3. Repeat the process until the number of edges in the MST equals n-1 (where n is the number of nodes).

# Prim’s Algorithm:

* + Prim’s algorithm builds the MST by starting from an arbitrary node and expanding

the tree by adding the smallest edge that connects a node inside the tree to a node outside the tree. The algorithm uses a **Priority Queue (Min-Heap)** to efficiently find the smallest edge.

# Steps of Prim’s Algorithm:

* + 1. Start with an arbitrary node and add it to the MST.
    2. Add the smallest edge that connects a node in the MST to a node outside the MST.
    3. Repeat the process until all nodes are included in the MST.
* **Multiple MSTs**: An undirected weighted graph can have multiple valid MSTs, depending on

the order in which edges are added. Both Prim’s and Kruskal’s algorithms ensure that a valid MST is constructed, but different runs of the algorithm may yield different trees.

# Code:

#include <iostream> #include <vector> #include <queue> #include <climits> using namespace std;

// Pair class to represent a node and the weight of an edge class Pair {

public:

int node, weight;

Pair(int node, int weight) { this->node = node;

this->weight = weight;

}

};

// Comparator for the priority queue to order by weight struct compareWeight {

bool operator()(const Pair& a, const Pair& b) { return a.weight > b.weight;

}

};

class Graph {

private:

int V; // Number of vertices

vector<vector<Pair>> adjList; // Adjacency list

public:

Graph(int V) { this->V = V;

adjList.resize(V);

}

// Add an edge to the graph

void addEdge(int src, int dest, int weight) { adjList[src].push\_back(Pair(dest, weight)); adjList[dest].push\_back(Pair(src, weight));

}

// Function to perform Prim's MST algorithm void primMST() {

vector<int> key(V, INT\_MAX); // Key values used to pick minimum weight edge vector<bool> visited(V, false); // To keep track of vertices included in MST

vector<int> parent(V, -1); // To store the MST

priority\_queue<Pair, vector<Pair>, compareWeight> pq; // Min-heap priority queue

key[0] = 0; // Start from the first node

pq.push(Pair(0, 0)); // Insert the first node with a weight of 0

while (!pq.empty()) {

int u = pq.top().node; // Extract the node with the minimum key value pq.pop();

visited[u] = true; // Include u in MST

// Traverse all adjacent vertices of u for (const Pair& neighbor : adjList[u]) {

int v = neighbor.node;

int weight = neighbor.weight;

// If v is not in MST and the weight of u-v is smaller than the current key of v if (!visited[v] && weight < key[v]) {

key[v] = weight;

pq.push(Pair(v, key[v])); // Add v to the priority queue parent[v] = u; // Update parent of v

}

}

}

printMST(parent, key);

}

// Function to print the MST and its total weight

void printMST(const vector<int>& parent, const vector<int>& key) { cout << "MST Graph -\n";

cout << "Edge \tWeight\n"; int totalWeight = 0;

for (int i = 1; i < V; ++i) {

cout << parent[i] << " - " << i << "\t" << key[i] << "\n"; totalWeight += key[i];

}

cout << "Total weight of MST: " << totalWeight << endl;

}

};

int main() {

int nodes, edgesCount;

cout << "Enter the number of nodes: "; cin >> nodes;

Graph graph(nodes);

cout << "Enter the number of edges: "; cin >> edgesCount;

cout << "Enter each edge in the format: src dest weight\n"; for (int i = 0; i < edgesCount; ++i) {

int src, dest, weight;

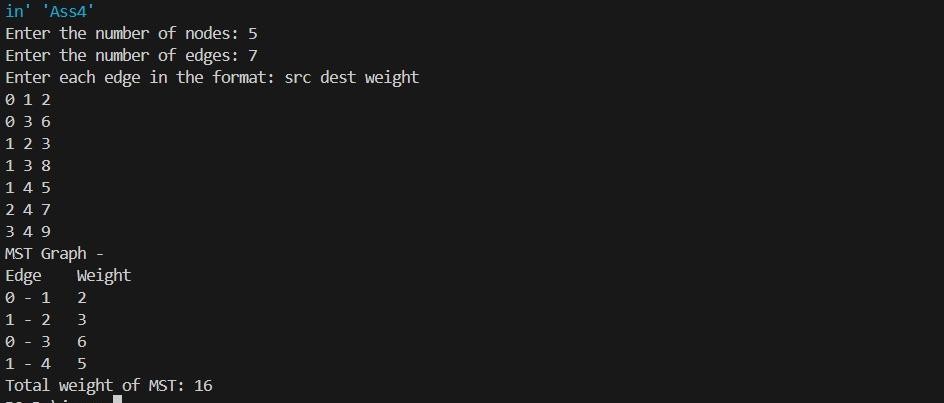
cin >> src >> dest >> weight; graph.addEdge(src, dest, weight);

}

graph.primMST(); return 0;

}

# Output:



**Case Studies**:

1. Consider a graph with 5 nodes and the following edges with weights:
   * (1, 2) with weight 10
   * (1, 3) with weight 5
   * (2, 3) with weight 4
   * (2, 4) with weight 7
   * (3, 5) with weight 3
   * (4, 5) with weight 1

Using Kruskal’s or Prim’s algorithm, find the total weight of the MST and the specific edges included in one possible MST.

# Conclusion:

This assignment provided a thorough understanding of Minimum Spanning Trees (MSTs) and how

they are used in optimizing networks and minimizing costs in graphs. By implementing both Kruskal’s and Prim’s algorithms, I gained insight into different approaches to solving the MST problem, each with its own advantages based on the nature of the graph. Kruskal’s algorithm is well-suited for sparse graphs, while Prim’s algorithm is efficient for dense graphs. The practical application of MSTs in areas such as network design, circuit design, and transportation networks demonstrates the

importance of efficient graph traversal techniques in solving real-world optimization problems.