# Roll No. COA244

# Assignment no : 7

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

#define MAX 10 // Maximum number of cities

// Structure for edges struct Edge {

int start, end, weight;

};

class Graph {

int adj[MAX][MAX]; string city[MAX]; int numCities; vector<Edge> mst;

int totalCost;

// Adjacency matrix

// City names

// Number of cities

// Minimum Spanning Tree (MST)

// Total cost of MST

public:

Graph();

void prims(int start); void showMST();

};

// Constructor

// Prim's algorithm to find MST

// Display the MST

// Constructor: Initializes the graph Graph::Graph() {

totalCost = 0;

cout << "Enter number of cities (1-" << MAX << "): "; cin >> numCities;

numCities = min(numCities, MAX);

// Get city names

for (int i = 0; i < numCities; i++) {

cout << "Enter city " << i + 1 << ": "; cin >> city[i];

}

// Initialize adjacency matrix with "infinity" (no direct connections) for (int i = 0; i < numCities; i++) {

for (int j = 0; j < numCities; j++) {

adj[i][j] = INT\_MAX; // Initialize with large value

}

}

// Get city connections (edges) int connections;

cout << "Enter number of city connections: "; cin >> connections;

// Show city codes

cout << "\nCity Codes: \n";

for (int i = 0; i < numCities; i++) { cout << i << " " << city[i] << endl;

}

// Read edges

for (int i = 0; i < connections; i++) { int x, y, cost;

cout << "Enter connection (city1 city2 cost): "; cin >> x >> y >> cost;

adj[x][y] = cost; // Undirected graph adj[y][x] = cost; // Symmetric

}

}

// Prim's Algorithm to find MST void Graph::prims(int start) {

bool visited[MAX] = {false}; // Array to track visited cities visited[start] = true; // Mark the starting city as visited

while (mst.size() < numCities - 1) { // While there are still edges to add Edge minEdge = {0, 0, INT\_MAX}; // Initialize minEdge with maximum cost

// Find the smallest edge from any visited city

for (int i = 0; i < numCities; i++) {

if (visited[i]) { // If city 'i' is visited for (int j = 0; j < numCities; j++) {

if (!visited[j] && adj[i][j] < minEdge.weight) { // If city 'j' is unvisited and edge weight is less

minEdge = {i, j, adj[i][j]}; // Update minEdge

}

}

}

}

// Add the edge to MST mst.push\_back(minEdge); totalCost += minEdge.weight;

visited[minEdge.end] = true; // Mark the destination city as visited

}

}

// Display the Minimum Spanning Tree (MST) void Graph::showMST() {

cout << "\nMost efficient network (Minimum Spanning Tree):\n"; for (Edge e : mst) {

cout << city[e.start] << " - " << city[e.end] << " (Cost: " << e.weight

<< ")\n";

}

cout << "Total network cost: " << totalCost << endl;

}

// Main function int main() {

Graph g; int start;

cout << "\nEnter starting city code: "; cin >> start;

g.prims(start); g.showMST();

return 0;

}

OUTPUT :

