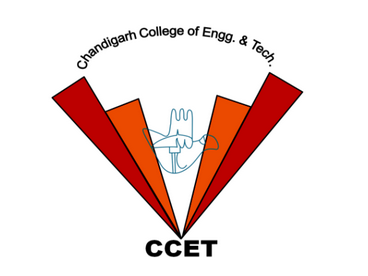
**CHANDIGARH COLLEGE OF ENGINEERING & TECHNOLOGY (DEGREE WING)**

****

Government institute under Chandigarh (UT) Administration, affiliated to Punjab University, Chandigarh

Department of Computer Science & Engineering

**Semester**: CSE 3rd

**SUBJECT:** Data Structures Practical (CS351)

**Problem 8: Case Study of Graphs**

**Submitted by: Submitted to:**

Bhavyam Dhand Dr. R.B. Patel

(CO23316) (Professor)

**CODE**

#include <bits/stdc++.h>

#include <vector>

using namespace std;

const int V = 7, MAX = 100;

// Enum to represent vertices

enum Vertex { A, B, C, D, E, F, G};

struct GraphStruct {

    vector<vector<int>> matrix = vector<vector<int>>(V, vector<int>(V, 0)); // Initialize matrix to 0

    vector<vector<int>> edgelist;

};

// Array to map enum values to labels

char vertexLabels[V] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};

void displayMatrix(GraphStruct &graph) {

    cout << "  ";

    for (int i = 0; i < V; i++) {

        cout << vertexLabels[i] << " ";

    }

    cout << endl;

    for (int i = 0; i < V; i++) {

        cout << vertexLabels[i] << " ";

        for (int j = 0; j < V; j++) {

            cout << graph.matrix[i][j] << " ";

        }

        cout << endl;

    }

}

void AddEdge(GraphStruct &graph, Vertex i, Vertex j, int weight) {

    graph.matrix[i][j] = weight;

    graph.matrix[j][i] = weight;

    graph.edgelist.push\_back({weight, i, j});

}

void BreadthFirstSearch(GraphStruct &graph, Vertex source) {

    int Q[MAX], front = 0, rear = front;

    bool Visited[V] = {false};

    Visited[source] = true;

    Q[rear++] = source;

    cout << "BFS traversal starting from " << vertexLabels[source] << ": ";

    while (front < rear) {

        int x = Q[front++];

        cout << vertexLabels[x] << " ";

        for (int i = 0; i < V; i++) {

            if (graph.matrix[x][i] != 0 && !Visited[i]) {

                Visited[i] = true;

                Q[rear++] = i;

            }

        }

    }

    cout << endl;

}

void DepthFirstSearch(GraphStruct &graph, int source, vector<bool>& visited) {

    cout << vertexLabels[source] << " ";

    visited[source] = true;

    for (int i = 0; i < graph.matrix[source].size(); i++) {

        if (graph.matrix[source][i] != 0 && !visited[i]) {

            DepthFirstSearch(graph, i, visited);

        }

    }

}

// Prim's Algorithm

int MinKey(vector<int>& key, vector<bool>& MSTSet) {

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < V; v++) {

        if (!MSTSet[v] && key[v] < min) {

            min = key[v];

            min\_index = v;

        }

    }

    return min\_index;

}

void PrimMST(GraphStruct &graph) {

    vector<int> parent(V);

    vector<int> key(V, INT\_MAX);

    vector<bool> MSTSet(V, false);

    key[0] = 0;

    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {

        int u = MinKey(key, MSTSet);

        MSTSet[u] = true;

        for (int v = 0; v < V; v++) {

            if (graph.matrix[u][v] && !MSTSet[v] && graph.matrix[u][v] < key[v]) {

                parent[v] = u;

                key[v] = graph.matrix[u][v];

            }

        }

    }

    int Cost;

    cout << "Edge \tWeight" << endl;

    for (int i = 1; i < V; i++){

        cout << vertexLabels[parent[i]] << " - " << vertexLabels[i] << " \t" << graph.matrix[i][parent[i]] << endl;

        Cost+=graph.matrix[i][parent[i]];

    }

    cout << "Weight Cost of Minimum Spanning Tree: " << Cost << endl;

}

// Kruskal's Algorithm

int findParent(vector<int>& parent, int x) {

    if (parent[x] == x)

        return x;

    return parent[x] = findParent(parent, parent[x]);

}

void UnionSet(int u, int v, vector<int>& parent, vector<int>& rank) {

    u = findParent(parent, u);

    v = findParent(parent, v);

    if (rank[u] < rank[v]) {

        parent[u] = v;

    } else if (rank[u] > rank[v]) {

        parent[v] = u;

    } else {

        parent[v] = u;

        rank[u]++;

    }

}

void KruskalMST(GraphStruct &graph) {

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

    vector<int> parent(V), rank(V, 0);

    for (int i = 0; i < V; i++)

        parent[i] = i;

    int minCost = 0;

    cout << "Edgelist for the MST:" << endl;

    for (auto& edge : graph.edgelist) {

        int wt = edge[0];

        int u = edge[1];

        int v = edge[2];

        int v1 = findParent(parent, u);

        int v2 = findParent(parent, v);

        if (v1 != v2) {

            UnionSet(v1, v2, parent, rank);

            minCost += wt;

            cout << vertexLabels[u] << " -- " << vertexLabels[v] << " == " << wt << endl;

        }

    }

    cout << "Weight Cost of Minimum Spanning Tree: " << minCost << endl;

}

// User Interface

int main() {

    GraphStruct graph;

    vector<bool> visited(V, false);

    // Adding edges

    AddEdge(graph, A, B, 1);

    AddEdge(graph, A, C, 4);

    AddEdge(graph, B, C, 2);

    AddEdge(graph, B, D, 3);

    AddEdge(graph, B, E, 10);

    AddEdge(graph, C, D, 6);

    AddEdge(graph, D, E, 5);

    AddEdge(graph, D, G, 1);

    AddEdge(graph, E, G, 2);

    AddEdge(graph, E, F, 7);

    AddEdge(graph, F, G, 5);

    int choice;

    do {

        cout << "\nGraph Menu:\n";

        cout << "1. Display Adjacency Matrix\n";

        cout << "2. Breadth-First Search (BFS)\n";

        cout << "3. Depth-First Search (DFS)\n";

        cout << "4. Prim's Minimum Spanning Tree\n";

        cout << "5. Kruskal's Minimum Spanning Tree\n";

        cout << "6. Quit\n";

        cout << "Enter choice: ";

        cin >> choice;

        switch (choice) {

            case 1:

                displayMatrix(graph);

                break;

            case 2:

                BreadthFirstSearch(graph, A); // BFS starting from vertex A

                break;

            case 3:

                fill(visited.begin(), visited.end(), false);

                cout << "DFS traversal starting from A: ";

                DepthFirstSearch(graph, A, visited); // DFS starting from vertex A

                cout << endl;

                break;

            case 4:

                PrimMST(graph);

                break;

            case 5:

                KruskalMST(graph);

                break;

            case 6:

                cout << "Exiting program.\n";

                break;

            default:

                cout << "Invalid choice. Try again.\n";

        }

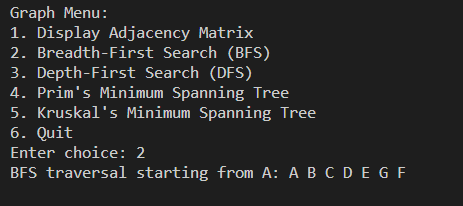
    } while (choice != 6);

    return 0;

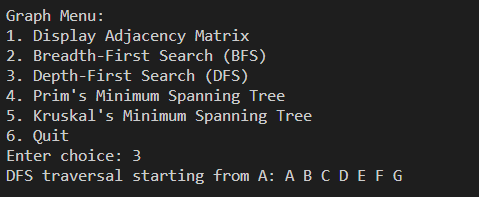
}

**CODE OUTPUT**

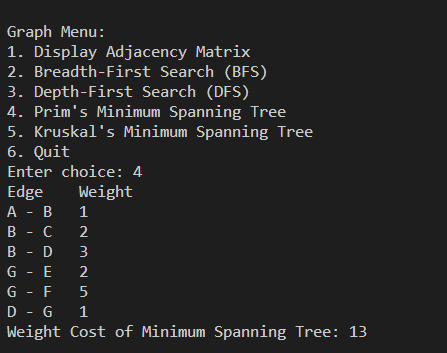
1. **Breadth First Search:**



1. **Depth First Search**



1. **Prim’s MST Algorithm:**



1. **Kruskal’s MST Algorithm**

