Name: Sai Gulve

PRN No: 202201040145

Batch: A4

Roll No: 63

Problem Statement:

Given a connected, weighted, and undirected graph with **V** vertices and **E** edges, implement **Prim's Algorithm** using the **Greedy Method** to construct a **Minimum Spanning Tree (MST)**. The goal is to find a subset of edges that connect all vertices with the **minimum possible total edge weight** and **no cycles**.

Introduction:

A **Minimum Spanning Tree (MST)** of a graph is a subset of edges that:

- Connect **all vertices** without any cycles.
- Have **minimum total weight** among all possible spanning trees.

Prim's Algorithm is a **Greedy Algorithm** that builds the MST **incrementally**, always selecting the **minimum-weight edge** that expands the tree without forming a cycle.

Approach & Algorithm:

1. **Initialize MST with a starting vertex** (usually vertex 0).
2. Maintain a **visited array** to track included vertices.
3. Select the **minimum-weight edge** connecting an included vertex to an unvisited vertex.
4. **Repeat until all vertices are included** in the MST.
5. **Measure execution time** using the chrono library.

Code :-

```
#include <bits/stdc++.h>

#include <chrono>


#define V 5

#define E 10 // Maximum number of edges

using namespace std;

using namespace std::chrono;
```

```
// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Structure to represent a graph for Kruskal's Algorithm
struct Graph {
    int V, E;
    Edge edges[E]; // Fixed-size array instead of vector
};

// Find function for Disjoint Set (with Path Compression)
int find(int parent[], int i) {
    if (parent[i] == -1)
        return i;
    return parent[i] = find(parent, parent[i]); // Path compression
}

// Union function for Disjoint Set
void Union(int parent[], int x, int y) {
    int xroot = find(parent, x);
    int yroot = find(parent, y);
    if (xroot != yroot)
        parent[xroot] = yroot;
}

// Function to find the vertex with the minimum key value
int minKey(int key[], 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;
}

// Function to print the constructed MST
void printMST(int parent[], int graph[V][V]) {
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t" << graph[parent[i]][i] << "\n";
}

// Function to implement Prim's Algorithm
void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    bool mstSet[V];

    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = 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[u][v] && !mstSet[v] && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
        }
```

```
    }

    printMST(parent, graph);
}

// Function to implement Kruskal's Algorithm
void kruskalMST(Graph& graph) {
    Edge result[V - 1]; // Array to store MST edges
    int parent[V];
    memset(parent, -1, sizeof(parent));

    // Sorting edges by weight using simple selection sort (as we are using arrays)
    for (int i = 0; i < graph.E - 1; i++) {
        for (int j = i + 1; j < graph.E; j++) {
            if (graph.edges[i].weight > graph.edges[j].weight) {
                swap(graph.edges[i], graph.edges[j]);
            }
        }
    }

    int edgeCount = 0;
    for (int i = 0; i < graph.E && edgeCount < V - 1; i++) {
        Edge e = graph.edges[i];
        int x = find(parent, e.src);
        int y = find(parent, e.dest);

        if (x != y) {
            result[edgeCount++] = e;
            Union(parent, x, y);
        }
    }
```

```
    cout << "Edge \tWeight\n";
    for (int i = 0; i < edgeCount; i++)
        cout << result[i].src << " - " << result[i].dest << " \t" << result[i].weight << "\n";
}

int main() {
    int graph[V][V];

    cout << "Enter the adjacency matrix (" << V << "x" << V << "):\n";
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            cin >> graph[i][j];

    // Construct Graph for Kruskal's Algorithm
    Graph g;
    g.V = V;
    g.E = 0;

    for (int i = 0; i < V; i++) {
        for (int j = i + 1; j < V; j++) {
            if (graph[i][j] > 0 && graph[i][j] != INT_MAX) { // Avoid 0-weight and infinite-weight edges
                g.edges[g.E++] = {i, j, graph[i][j]};
            }
        }
    }

    auto start_prim = high_resolution_clock::now();
    cout << "\nPrim's Algorithm:\n";
    primMST(graph);
    auto end_prim = high_resolution_clock::now();
```

```
    auto duration_prim = duration_cast<microseconds>(end_prim - start_prim);


    auto start_kruskal = high_resolution_clock::now();
    cout << "\nKruskal's Algorithm:\n";
    kruskalMST(g);
    auto end_kruskal = high_resolution_clock::now();
    auto duration_kruskal = duration_cast<microseconds>(end_kruskal - start_kruskal);


    cout << "\nExecution time:\n";
    cout << "Prim's Algorithm: " << duration_prim.count() << " microseconds\n";
    cout << "Kruskal's Algorithm: " << duration_kruskal.count() << " microseconds\n";


    return 0;
}
```

Output :-

```
Enter the adjacency matrix (3x3):
4
5
6
7
8
9
4
5
6

Prim's Algorithm - Minimum Spanning Tree:
Edge    Weight
0 - 1   7
0 - 2   4

--------------------------------
Process exited after 9.417 seconds with return value 0
Press any key to continue . . .
```