Analysis of Algorithm

Practical no 7 :

Prism & kruskal’s Algorithm

Prism Algorithm

Code :

import java.util.\*;

public class PrimMST {

    // Number of vertices in the graph

    static final int V = 4;

    // A utility function to find the vertex with minimum key value, from the set of vertices not yet included in MST

    static int minKey(int key[], boolean mstSet[]) {

        // Initialize min value

        int min = Integer.MAX\_VALUE, min\_index = -1;

        for (int v = 0; v < V; v++) {

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

                min = key[v];

                min\_index = v;

            }

        }

        return min\_index;

    }

    // A utility function to print the constructed MST stored in parent[]

    static void printMST(int parent[], int graph[][]) {

        System.out.println("Edge \tWeight");

        for (int i = 1; i < V; i++) {

            System.out.println(parent[i] + " - " + i + " \t" + graph[parent[i]][i]);

        }

    }

    // Function to construct and print MST for a graph represented using adjacency matrix representation

    static void primMST(int graph[][]) {

        int parent[] = new int[V];

        int key[] = new int[V];

        boolean mstSet[] = new boolean[V];

        // Initialize all keys as INFINITE and mstSet[] as false

        Arrays.fill(key, Integer.MAX\_VALUE);

        Arrays.fill(mstSet, false);

        // Always include the first vertex in MST. Make key 0 so that this vertex is picked as the first vertex

        key[0] = 0;

        parent[0] = -1;

        // The MST will have V vertices

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

            // Pick the minimum key vertex from the set of vertices not yet included in MST

            int u = minKey(key, mstSet);

            // Add the picked vertex to the MST Set

            mstSet[u] = true;

            // Update the key value and parent index of the adjacent vertices of the picked vertex.

            // Consider only those vertices which are not yet included in MST

            for (int v = 0; v < V; v++) {

                // graph[u][v] is non-zero only for adjacent vertices, mstSet[v] is false for vertices not yet included in MST

                // Update the key only if graph[u][v] is smaller than key[v]

                if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {

                    parent[v] = u;

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

                }

            }

        }

        // Print the constructed MST

        printMST(parent, graph);

    }

    // Driver code

    public static void main(String[] args) {

        int graph[][] = {

            { 0, 2, 0, 6},

            { 2, 0, 3, 8},

            { 0, 3, 0, 0},

            { 6, 8, 0, 0},

        };

        // Print the solution

        primMST(graph);

    }

}

Output :

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

Kruskal’s Algorithm

import java.util.Arrays;

import java.util.Comparator;

// Class to represent a graph edge

class Edge {

    int src, dest, weight;

    public Edge(int src, int dest, int weight) {

        this.src = src;

        this.dest = dest;

        this.weight = weight;

    }

}

// Disjoint-set (Union-Find) data structure

class DisjointSet {

    int[] parent, rank;

    public DisjointSet(int n) {

        parent = new int[n];

        rank = new int[n]

        for (int i = 0; i < n; i++) {

            parent[i] = i;

            rank[i] = 0;

        }

    }

    // Find the representative of the set containing x

    public int find(int x) {

        if (parent[x] != x) {

            parent[x] = find(parent[x]); // Path compression

        }

        return parent[x];

    }

    // Union of two sets

    public void union(int x, int y) {

        int rootX = find(x);

        int rootY = find(y);

        // Union by rank

        if (rootX != rootY) {

            if (rank[rootX] > rank[rootY]) {

                parent[rootY] = rootX;

            } else if (rank[rootX] < rank[rootY]) {

                parent[rootX] = rootY;

            } else {

                parent[rootY] = rootX;

                rank[rootX]++;

            }

        }

    }

}

public class KruskalAlgorithm {

    // Function to perform Kruskal's algorithm to find MST

    public static void kruskalMST(Edge[] edges, int V) {

        // Sort edges based on their weight

        Arrays.sort(edges, Comparator.comparingInt(e -> e.weight));

        // Create a Disjoint-set (Union-Find) data structure

        DisjointSet ds = new DisjointSet(V);

        System.out.println("Edges in the Minimum Spanning Tree:");

        // Traverse the sorted edges and add them to the MST if they don't form a cycle

        for (Edge edge : edges) {

            int x = ds.find(edge.src);

            int y = ds.find(edge.dest);

            // If adding this edge doesn't form a cycle

            if (x != y) {

                System.out.println(edge.src + " - " + edge.dest + " : " + edge.weight);

                ds.union(x, y);  // Union the sets

            }

        }

    }

    public static void main(String[] args) {

        int V = 4; // Number of vertices

        Edge[] edges = new Edge[] {

            new Edge(0, 1, 10),

            new Edge(0, 2, 6),

            new Edge(0, 3, 5),

            new Edge(1, 3, 15),

            new Edge(2, 3, 4)

        };

        kruskalMST(edges, V); // Run Kruskal's algorithm

    }

}

Output :

Edges in the Minimum Spanning Tree:

2 - 3 : 4

0 - 3 : 5

0 - 1 : 10