**Kruskal’s lab manual**

import java.util.\*;

class Graph {

// A class to represent a graph edge

class Edge implements Comparable<Edge> {

int src, dest, weight;

// Comparator function used for sorting edges based on their weight

public int compareTo(Edge compareEdge) {

return this.weight - compareEdge.weight;

}

}

// A class to represent a subset for union-find

class Subset {

int parent, rank;

}

int V, E; // V -> number of vertices, E -> number of edges

Edge[] edge; // collection of all edges

// Constructor to create a graph with V vertices and E edges

Graph(int v, int e) {

V = v;

E = e;

edge = new Edge[e];

for (int i = 0; i < e; ++i)

edge[i] = new Edge();

}

// A utility function to find set of an element i (uses path compression technique)

int find(Subset[] subsets, int i) {

// find root and make root as parent of i (path compression)

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// A function that does union of two sets of x and y (uses union by rank)

void Union(Subset[] subsets, int x, int y) {

int xroot = find(subsets, x);

int yroot = find(subsets, y);

// Attach smaller rank tree under root of high rank tree (Union by Rank)

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

// The main function to construct MST using Kruskal's algorithm

void KruskalMST() {

// This will store the resultant MST

Edge[] result = new Edge[V];

int e = 0; // An index variable used for result[]

int i = 0; // An index variable used for sorted edges

// Step 1: Sort all the edges in non-decreasing order of their weight.

Arrays.sort(edge);

// Allocate memory for creating V subsets

Subset[] subsets = new Subset[V];

for (i = 0; i < V; ++i)

subsets[i] = new Subset();

// Create V subsets with single elements

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

subsets[v].parent = v;

subsets[v].rank = 0;

}

i = 0; // Index used to pick the next edge

// Number of edges to be taken is equal to V-1

while (e < V - 1) {

// Step 2: Pick the smallest edge. And increment the index for next iteration

Edge next\_edge = edge[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

// If including this edge does not cause a cycle, include it in the result and increment the index of result

if (x != y) {

result[e++] = next\_edge;

Union(subsets, x, y);

}

// Else discard the next\_edge

}

// Print the contents of result[] to display the built MST

System.out.println("Following are the edges in the constructed MST");

int minimumCost = 0;

for (i = 0; i < e; ++i) {

System.out.println(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);

minimumCost += result[i].weight;

}

System.out.println("Minimum Cost Spanning Tree " + minimumCost);

}

// Driver Code

public static void main(String[] args) {

/\* Let us create the following weighted graph

10

0 ------- 1

| \ |

6| 5\ |15

| \ |

2 ------- 3

4

\*/

int V = 4; // Number of vertices in graph

int E = 5; // Number of edges in graph

Graph graph = new Graph(V, E);

// add edge 0-1

graph.edge[0].src = 0;

graph.edge[0].dest = 1;

graph.edge[0].weight = 10;

// add edge 0-2

graph.edge[1].src = 0;

graph.edge[1].dest = 2;

graph.edge[1].weight = 6;

// add edge 0-3

graph.edge[2].src = 0;

graph.edge[2].dest = 3;

graph.edge[2].weight = 5;

// add edge 1-3

graph.edge[3].src = 1;

graph.edge[3].dest = 3;

graph.edge[3].weight = 15;

// add edge 2-3

graph.edge[4].src = 2;

graph.edge[4].dest = 3;

graph.edge[4].weight = 4;

// Function call

graph.KruskalMST();

}

}

**• Find the number of distinct minimum spanning trees for a given weighted graph.**

import java.util.\*;

class NumDistinct {

// A class to represent a graph edge

class Edge implements Comparable<Edge> {

int src, dest, weight;

// Comparator function used for sorting edges based on their weight

public int compareTo(Edge compareEdge) {

return this.weight - compareEdge.weight;

}

}

// A class to represent a subset for union-find

class Subset {

int parent, rank;

}

int V, E; // V -> number of vertices, E -> number of edges

Edge[] edge; // collection of all edges

// Constructor to create a graph with V vertices and E edges

NumDistinct(int v, int e) {

V = v;

E = e;

edge = new Edge[e];

for (int i = 0; i < e; ++i)

edge[i] = new Edge();

}

// A utility function to find set of an element i (uses path compression technique)

int find(Subset[] subsets, int i) {

// find root and make root as parent of i (path compression)

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// A function that does union of two sets of x and y (uses union by rank)

void Union(Subset[] subsets, int x, int y) {

int xroot = find(subsets, x);

int yroot = find(subsets, y);

// Attach smaller rank tree under root of high rank tree (Union by Rank)

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

// The main function to construct MST using Kruskal's algorithm and count distinct MSTs

int countDistinctMSTs() {

// Sort all edges in non-decreasing order of their weight.

Arrays.sort(edge);

// Allocate memory for creating V subsets

Subset[] subsets = new Subset[V];

for (int i = 0; i < V; ++i)

subsets[i] = new Subset();

// Create V subsets with single elements

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

subsets[v].parent = v;

subsets[v].rank = 0;

}

int mstWeight = 0;

int edgesInMST = 0;

int numDistinctMSTs = 1; // Start with 1 possible MST

// Traverse through the sorted edges

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

Edge next\_edge = edge[i];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

// If including this edge does not cause a cycle, include it in the MST

if (x != y) {

mstWeight += next\_edge.weight;

edgesInMST++;

Union(subsets, x, y);

} else {

// If the edge causes a cycle, it may still be part of a distinct MST.

// Check if there are multiple edges with the same weight.

int weight = next\_edge.weight;

int count = 0;

// Count edges with the same weight

while (i + count < E && edge[i + count].weight == weight) {

count++;

}

// If there are multiple edges with the same weight, we have multiple options

if (count > 1) {

numDistinctMSTs \*= count; // Multiply the number of ways to choose the edges

}

i += count - 1; // Skip all edges with the same weight

}

// If the MST has exactly V-1 edges, stop.

if (edgesInMST == V - 1)

break;

}

// If we couldn't include exactly V-1 edges, it means no MST exists.

if (edgesInMST != V - 1)

return 0;

return numDistinctMSTs;

}

// Driver Code

public static void main(String[] args) {

/\* Let us create the following weighted graph:

10

0 --------- 1

| \ / |

| \ 5 / |

6| \ / | 15

| \|/ |

2 --------- 3

4

\*/

int V = 4; // Number of vertices in graph

int E = 5; // Number of edges in graph

NumDistinct graph = new NumDistinct(V, E);

// Add edge 0-1

graph.edge[0].src = 0;

graph.edge[0].dest = 1;

graph.edge[0].weight = 10;

// Add edge 0-2

graph.edge[1].src = 0;

graph.edge[1].dest = 2;

graph.edge[1].weight = 6;

// Add edge 0-3

graph.edge[2].src = 0;

graph.edge[2].dest = 3;

graph.edge[2].weight = 5;

// Add edge 1-3

graph.edge[3].src = 1;

graph.edge[3].dest = 3;

graph.edge[3].weight = 15;

// Add edge 2-3

graph.edge[4].src = 2;

graph.edge[4].dest = 3;

graph.edge[4].weight = 4;

// Function call to count distinct MSTs

int distinctMSTs = graph.countDistinctMSTs();

System.out.println("Number of distinct MSTs: " + distinctMSTs);

}

}

**Prim’s:**

import java.util.\*;

class MST {

// Number of vertices in the graph

private static final int V = 5;

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

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] == false && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

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

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[i][parent[i]]);

}

}

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

void primMST(int graph[][]) {

// Array to store constructed MST

int parent[] = new int[V];

// Key values used to pick minimum weight edge in cut

int key[] = new int[V];

// To represent set of vertices included in MST

Boolean mstSet[] = new Boolean[V];

// Initialize all keys as INFINITE

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

key[i] = Integer.MAX\_VALUE;

mstSet[i] = false;

}

// Always include the first vertex in MST.

key[0] = 0; // Make key 0 so that this vertex is picked as the first vertex

parent[0] = -1; // First node is always root of MST

// 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 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 of 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] == false && graph[u][v] < key[v]) {

parent[v] = u;

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

}

}

}

// Print the constructed MST

printMST(parent, graph);

}

public static void main(String[] args) {

// Let us create the following graph

// 23

// (0)---(1)---(2)

// |/\|

// 6|8/\5|7

// |/ \|

// (3)-------(4)

// 9

MST t = new MST();

int graph[][] = new int[][]{

{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}

};

// Print the solution

t.primMST(graph);

}

}

**Dijkstra’s:**

import java.util.\*;

class ShortestPath {

static final int V = 5;

int minDistance(int dist[], Boolean sptSet[]) {

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

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

if (sptSet[v] == false && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

void printSolution(int dist[]) {

System.out.println("Vertex \t\t Distance from Source");

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

System.out.println(i + "\t\t " + dist[i]);

}

}

void dijkstra(int graph[][], int src) {

int dist[] = new int[V];

Boolean sptSet[] = new Boolean[V];

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

dist[i] = Integer.MAX\_VALUE;

sptSet[i] = false;

}

dist[src] = 0;

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

int u = minDistance(dist, sptSet);

sptSet[u] = true;

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

if (!sptSet[v] && graph[u][v] != 0 && dist[u] != Integer.MAX\_VALUE && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

printSolution(dist);

}

public static void main(String[] args) {

int graph[][] = new int[][] {

{0, 10, 5, 0, 0},

{0, 0, 2, 1, 0},

{0, 3, 0, 9, 2},

{0, 0, 0, 0, 4},

{7, 0, 0, 6, 0}

};

ShortestPath t = new ShortestPath();

t.dijkstra(graph, 0);

}

}