### **Table of Contents**

#### **Graph Traversal and Related Algorithms**

1. **Graph using Adjacency Matrix – 1**
2. **DFS using Stack – 2**
3. **DFS using Recursion – 3**
4. **BFS Algorithm – 4**
5. **Path Exists Between Two Vertices – 5**
6. **All Paths Between Two Vertices – 6**
7. **All Paths from Source to Destination – 7**
8. **Distance Between Source and Destination – 8**
9. **Cycle in Directed Graph – 9**
10. **Cycle in Undirected Graph using Disjoint Set – 10**
11. **Strongly Connected Directed Graph – 11**

#### **String Matching Algorithms**

1. **String Matching using Brute Force – 12**
2. **String Matching using Rabin-Karp Algorithm – 13**

#### **Divide and Conquer Algorithms**

1. **Merge Sort – 14**
2. **Count Inversions Using Merge Sort – 15**
3. **Quick Sort – 16**

#### **Minimum Spanning Tree and Shortest Path Algorithms**

1. **Kruskal’s Algorithm – 17**
2. **Prim’s MST – 19**
3. **Dijkstra’s Shortest Distance – 21**

**BFS DFS**

**1. Graph using Adjacency Matrix**

public class GraphMatrix {

int[][] adjMatrix;

int vertices;

GraphMatrix(int v) {

vertices = v;

adjMatrix = new int[v][v];}

void addEdge(int i, int j) {

adjMatrix[i][j] = 1;

adjMatrix[j][i] = 1; }

void display() {

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

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

System.out.print(adjMatrix[i][j] + " "); }

System.out.println();} }

public static void main(String[] args) {

GraphMatrix g = new GraphMatrix(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.display();

}}

**2. DFS using Stack (Example 12.3)**

import java.util.\*;

public class DFSStack {

private int vertices;

private LinkedList<Integer>[] adj;

DFSStack(int v) {

vertices = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

void addEdge(int v, int w) {

adj[v].add(w);

}

void DFS(int start) {

boolean[] visited = new boolean[vertices];

Stack<Integer> stack = new Stack<>();

stack.push(start);

while (!stack.empty()) {

int node = stack.pop();

if (!visited[node]) {

System.out.print(node + " ");

visited[node] = true;

}

for (int neighbor : adj[node]) {

if (!visited[neighbor])

stack.push(neighbor);

}

}

}

public static void main(String[] args) {

DFSStack g = new DFSStack(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.DFS(0);

}

}

**3. DFS using Recursion**

import java.util.\*;

public class DFSRecursive {

private int vertices;

private LinkedList<Integer>[] adj;

DFSRecursive(int v) {

vertices = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

void addEdge(int v, int w) {

adj[v].add(w);

}

void DFSUtil(int v, boolean[] visited) {

visited[v] = true;

System.out.print(v + " ");

for (int n : adj[v]) {

if (!visited[n])

DFSUtil(n, visited);

}

}

void DFS(int v) {

boolean[] visited = new boolean[vertices];

DFSUtil(v, visited);

}

public static void main(String[] args) {

DFSRecursive g = new DFSRecursive(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.DFS(0);

}

}

**4. BFS Algorithm**

import java.util.\*;

public class BFS {

private int vertices;

private LinkedList<Integer>[] adj;

BFS(int v) {

vertices = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

void addEdge(int v, int w) {

adj[v].add(w);

}

void BFSAlgo(int s) {

boolean[] visited = new boolean[vertices];

Queue<Integer> queue = new LinkedList<>();

visited[s] = true;

queue.add(s);

while (!queue.isEmpty()) {

s = queue.poll();

System.out.print(s + " ");

for (int n : adj[s]) {

if (!visited[n]) {

visited[n] = true;

queue.add(n);

}

}

}

}

public static void main(String[] args) {

BFS g = new BFS(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.BFSAlgo(0);

}

}

**5. Path Exists Between Two Vertices**

import java.util.\*;

public class PathExists {

private int vertices;

private LinkedList<Integer>[] adj;

PathExists(int v) {

vertices = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

void addEdge(int v, int w) {

adj[v].add(w);

}

boolean isReachable(int s, int d) {

boolean[] visited = new boolean[vertices];

Queue<Integer> queue = new LinkedList<>();

visited[s] = true;

queue.add(s);

while (!queue.isEmpty()) {

int n = queue.poll();

if (n == d)

return true;

for (int i : adj[n]) {

if (!visited[i]) {

visited[i] = true;

queue.add(i);

}

}

}

return false;

}

public static void main(String[] args) {

PathExists g = new PathExists(4);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 3);

System.out.println("Path exists: " + g.isReachable(0, 3));

}

}

**6. All Paths Between Two Vertices**

import java.util.\*;

public class AllPaths {

private int vertices;

private LinkedList<Integer>[] adj;

AllPaths(int v) {

vertices = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

void addEdge(int v, int w) {

adj[v].add(w);

}

void printAllPaths(int s, int d) {

boolean[] visited = new boolean[vertices];

ArrayList<Integer> pathList = new ArrayList<>();

pathList.add(s);

printAllPathsUtil(s, d, visited, pathList);

}

void printAllPathsUtil(Integer u, Integer d, boolean[] visited, List<Integer> localPathList) {

visited[u] = true;

if (u.equals(d)) {

System.out.println(localPathList);

visited[u] = false;

return;

}

for (Integer i : adj[u]) {

if (!visited[i]) {

localPathList.add(i);

printAllPathsUtil(i, d, visited, localPathList);

localPathList.remove(i);

}}

visited[u] = false; }

public static void main(String[] args) {

AllPaths g = new AllPaths(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

g.printAllPaths(2, 3); }}

**7. All Paths from Source to Destination**

import java.util.\*;

public class AllPathsSD {

private int vertices;

private LinkedList<Integer>[] adj;

AllPathsSD(int v) {

vertices = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

void addEdge(int v, int w) {

adj[v].add(w);

}

void printAllPaths(int s, int d) {

boolean[] visited = new boolean[vertices];

List<Integer> pathList = new ArrayList<>();

pathList.add(s);

printAllPathsUtil(s, d, visited, pathList);

}

private void printAllPathsUtil(int u, int d, boolean[] visited, List<Integer> path) {

visited[u] = true;

if (u == d) {

System.out.println(path);

} else {

for (Integer i : adj[u]) {

if (!visited[i]) {

path.add(i);

printAllPathsUtil(i, d, visited, path);

path.remove(i);

}

}

}

visited[u] = false;

}

public static void main(String[] args) {

AllPathsSD g = new AllPathsSD(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 3);

g.printAllPaths(0, 3);

}

}

**8. Distance Between Source and Destination**

import java.util.\*;

public class ShortestPathBFS {

private int vertices;

private LinkedList<Integer>[] adj;

ShortestPathBFS(int v) {

vertices = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

void addEdge(int v, int w) {

adj[v].add(w);

}

int shortestDistance(int src, int dest) {

boolean[] visited = new boolean[vertices];

int[] distance = new int[vertices];

Queue<Integer> queue = new LinkedList<>();

visited[src] = true;

queue.add(src);

distance[src] = 0;

while (!queue.isEmpty()) {

int u = queue.poll();

for (int v : adj[u]) {

if (!visited[v]) {

visited[v] = true;

distance[v] = distance[u] + 1;

queue.add(v);

}

}

}

return distance[dest];

}

public static void main(String[] args) {

ShortestPathBFS g = new ShortestPathBFS(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 3);

g.addEdge(3, 4);

System.out.println("Shortest Distance: " + g.shortestDistance(0, 4));

}

}

**9. Cycle in Directed Graph**

import java.util.\*;

public class CycleInDirectedGraph {

private int vertices;

private LinkedList<Integer>[] adj;

CycleInDirectedGraph(int v) {

vertices = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

void addEdge(int u, int v) {

adj[u].add(v);

}

boolean isCyclic() {

boolean[] visited = new boolean[vertices];

boolean[] recStack = new boolean[vertices];

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

if (isCyclicUtil(i, visited, recStack))

return true;

return false;

}

private boolean isCyclicUtil(int v, boolean[] visited, boolean[] recStack) {

if (recStack[v]) return true;

if (visited[v]) return false;

visited[v] = true;

recStack[v] = true;

for (Integer neighbor : adj[v])

if (isCyclicUtil(neighbor, visited, recStack))

return true;

recStack[v] = false;

return false;

}

public static void main(String[] args) {

CycleInDirectedGraph g = new CycleInDirectedGraph(4);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 0); // Cycle

g.addEdge(2, 3);

System.out.println("Graph has cycle: " + g.isCyclic());

}

}

**10. Cycle in Undirected Graph using Disjoint Set**

public class CycleUndirectedDisjointSet {

int[] parent;

int find(int i) {

if (parent[i] == -1)

return i;

return find(parent[i]);

}

void union(int x, int y) {

int xset = find(x);

int yset = find(y);

if (xset != yset)

parent[xset] = yset;

}

boolean isCycle(int[][] edges, int V) {

parent = new int[V];

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

parent[i] = -1;

for (int[] edge : edges) {

int x = find(edge[0]);

int y = find(edge[1]);

if (x == y)

return true;

union(x, y);

}

return false;

}

public static void main(String[] args) {

CycleUndirectedDisjointSet g = new CycleUndirectedDisjointSet();

int[][] edges = {{0, 1}, {1, 2}, {2, 0}}; // Cycle

System.out.println("Graph has cycle: " + g.isCycle(edges, 3));

}

}

**11. Strongly Connected Directed Graph**

import java.util.\*;

public class StronglyConnectedGraph {

int vertices;

LinkedList<Integer>[] adj;

StronglyConnectedGraph(int v) {

vertices = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>(); }

void addEdge(int v, int w) {

adj[v].add(w); }

void DFSUtil(int v, boolean[] visited) {

visited[v] = true;

for (int i : adj[v])

if (!visited[i])

DFSUtil(i, visited);}

StronglyConnectedGraph getTranspose() {

StronglyConnectedGraph g = new StronglyConnectedGraph(vertices);

for (int v = 0; v < vertices; v++)

for (int i : adj[v])

g.adj[i].add(v);

return g; }

boolean isStronglyConnected() {

boolean[] visited = new boolean[vertices];

DFSUtil(0, visited);

for (boolean v : visited)

if (!v)

return false;

StronglyConnectedGraph gr = getTranspose();

visited = new boolean[vertices];

gr.DFSUtil(0, visited);

for (boolean v : visited)

if (!v)

return false;

return true; }

public static void main(String[] args) {

StronglyConnectedGraph g = new StronglyConnectedGraph(5);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 3);

g.addEdge(3, 0);

g.addEdge(2, 4); g.addEdge(4, 2);

System.out.println("Graph is strongly connected: " + g.isStronglyConnected()); }}

**String Matching Algorithms**

**1. String Matching using Brute Force**

public class BruteForceMatch {

public static int bruteForceSearch(String text, String pattern) {

int n = text.length();

int m = pattern.length();

for (int i = 0; i <= n - m; i++) {

int j;

for (j = 0; j < m; j++) {

if (text.charAt(i + j) != pattern.charAt(j))

break;

}

if (j == m)

return i; // Match found

}

return -1; // No match

}

public static void main(String[] args) {

String text = "abcdefghij";

String pattern = "def";

int index = bruteForceSearch(text, pattern);

if (index != -1)

System.out.println("Pattern found at index: " + index);

else

System.out.println("Pattern not found");

}

}

**2. String Matching using Rabin-Karp Algorithm**

public class RabinKarp {

public static final int d = 256; // number of characters in input alphabet

public static final int q = 101; // a prime number

public static void search(String pattern, String text) {

int m = pattern.length();

int n = text.length();

int i, j;

int p = 0; // hash value for pattern

int t = 0; // hash value for text

int h = 1;

// The value of h would be "pow(d, M-1)%q"

for (i = 0; i < m - 1; i++)

h = (h \* d) % q;

// Calculate hash value for pattern and first window of text

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

p = (d \* p + pattern.charAt(i)) % q;

t = (d \* t + text.charAt(i)) % q;

}

// Slide the pattern over text

for (i = 0; i <= n - m; i++) {

if (p == t) {

// Check for characters one by one

for (j = 0; j < m; j++) {

if (text.charAt(i + j) != pattern.charAt(j))

break;

}

if (j == m)

System.out.println("Pattern found at index " + i);

}

// Calculate hash value for next window

if (i < n - m) {

t = (d \* (t - text.charAt(i) \* h) + text.charAt(i + m)) % q;

// We might get negative value of t, convert it to positive

if (t < 0)

t = (t + q);

}

}

}

public static void main(String[] args) {

String text = "abcdefghij";

String pattern = "def";

search(pattern, text);

}}

**Divide and Conquer**

**1. Merge Sort**

import java.util.Arrays;

public class MergeSort {

public static void merge(int[] arr, int l, int m, int r) {

int[] left = Arrays.copyOfRange(arr, l, m + 1);

int[] right = Arrays.copyOfRange(arr, m + 1, r + 1);

int i = 0, j = 0, k = l;

while (i < left.length && j < right.length) {

if (left[i] <= right[j]) arr[k++] = left[i++];

else arr[k++] = right[j++];

}

while (i < left.length) arr[k++] = left[i++];

while (j < right.length) arr[k++] = right[j++];

}

public static void mergeSort(int[] arr, int l, int r) {

if (l < r) {

int m = (l + r) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

public static void main(String[] args) {

int[] arr = {5, 2, 9, 1, 5, 6};

mergeSort(arr, 0, arr.length - 1);

System.out.println("Sorted: " + Arrays.toString(arr));

}

}

**2. Count Inversions Using Merge Sort**  
import java.util.Arrays;

public class CountInversions {

private static int mergeAndCount(int[] arr, int l, int m, int r) {

int[] left = Arrays.copyOfRange(arr, l, m + 1);

int[] right = Arrays.copyOfRange(arr, m + 1, r + 1);

int i = 0, j = 0, k = l, invCount = 0;

while (i < left.length && j < right.length) {

if (left[i] <= right[j]) {

arr[k++] = left[i++];

} else {

arr[k++] = right[j++];

invCount += (left.length - i);

}

}

while (i < left.length) arr[k++] = left[i++];

while (j < right.length) arr[k++] = right[j++];

return invCount;

}

private static int sortAndCount(int[] arr, int l, int r) {

int count = 0;

if (l < r) {

int m = (l + r) / 2;

count += sortAndCount(arr, l, m);

count += sortAndCount(arr, m + 1, r);

count += mergeAndCount(arr, l, m, r);

}

return count;

}

public static int inversionCount(int[] arr) {

return sortAndCount(arr.clone(), 0, arr.length - 1);

}

public static void main(String[] args) {

int[] arr = {1, 20, 6, 4, 5};

System.out.println("Sorted: " + Arrays.toString(arr));

System.out.println("Inversions: " + inversionCount(arr)); // output: 5

}

}

**3.Quick Sort**

import java.util.Arrays;

public class QuickSort {

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1); // Left part

quickSort(arr, pivotIndex + 1, high); // Right part

}

}

private static int partition(int[] arr, int low, int high) {

int pivot = arr[high]; // pivot is last element

int i = low - 1; // index of smaller element

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

// swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// swap arr[i+1] and pivot (arr[high])

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1; // return pivot index

}

public static void main(String[] args) {

int[] arr = {10, 7, 8, 9, 1, 5};

System.out.println("Original: " + Arrays.toString(arr));

quickSort(arr, 0, arr.length - 1);

System.out.println("Sorted: " + Arrays.toString(arr));

}

}

**Kruskal, Prim, Dijkstra**

**1.Kruskal’s Algorithm:**  
import java.util.\*;

// Edge class

class Edge implements Comparable<Edge> {

int src, dest, weight;

Edge(int s, int d, int w) {

src = s;

dest = d;

weight = w;

}

public int compareTo(Edge compareEdge) {

return this.weight - compareEdge.weight; // ascending order

}

}

// Disjoint Set (Union-Find)

class Subset {

int parent;

int rank;

}

public class KruskalAlgorithm {

int vertices; // number of vertices

List<Edge> edges = new ArrayList<>();

KruskalAlgorithm(int v) {

vertices = v;

}

void addEdge(int src, int dest, int weight) {

edges.add(new Edge(src, dest, weight));

}

// Find root of set

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

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

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

return subsets[i].parent;

}

// Union of two sets by rank

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

int xroot = find(subsets, x);

int yroot = find(subsets, y);

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++;

}

}

void kruskalMST() {

List<Edge> result = new ArrayList<>();

Collections.sort(edges); // sort all edges by weight

Subset[] subsets = new Subset[vertices];

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

subsets[i] = new Subset();

subsets[i].parent = i;

subsets[i].rank = 0;

}

int e = 0, i = 0;

while (e < vertices - 1 && i < edges.size()) {

Edge next = edges.get(i++);

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

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

if (x != y) {

result.add(next);

union(subsets, x, y);

e++;

}

}

System.out.println("Edges in MST:");

for (Edge edge : result) {

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

}

}

public static void main(String[] args) {

KruskalAlgorithm g = new KruskalAlgorithm(4);

g.addEdge(0, 1, 10);

g.addEdge(0, 2, 6);

g.addEdge(0, 3, 5);

g.addEdge(1, 3, 15);

g.addEdge(2, 3, 4);

g.kruskalMST();

}

}

**2.Prim’s MST**

import java.util.\*;

public class PrimsMST {

private static final int V = 5; // Number of vertices in the graph

// 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[]) {

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

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

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

min = key[v];

minIndex = v;

}

}

return minIndex;

}

// 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

void primMST(int graph[][]) {

int parent[] = new int[V]; // Array to store constructed MST

int key[] = new int[V]; // Key values used to pick minimum weight edge

boolean mstSet[] = new boolean[V]; // To represent set of vertices included in MST

// Initialize all keys as INFINITE

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

key[i] = Integer.MAX\_VALUE;

mstSet[i] = false;

}

// Always include first vertex in MST

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

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 not yet included in MST

int u = minKey(key, mstSet);

mstSet[u] = true;

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

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

// graph[u][v] is non zero only for adjacent vertices of u

// 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);

}

public static void main(String[] args) {

PrimsMST t = new PrimsMST();

/\* Example graph represented as adjacency matrix

0 means no edge \*/

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},

};

t.primMST(graph);

}

}

**3.Dijkstra’s Shortest Distance**

import java.util.\*;

public class Dijkstra {

private static final int V = 5; // Number of vertices

// Function to find the vertex with minimum distance value, from the set of vertices not yet processed

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

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

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

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

min = dist[v];

minIndex = v;

}

}

return minIndex;

}

// Function to print the constructed distance array

void printSolution(int dist[]) {

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

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

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

}

// Function that implements Dijkstra's single source shortest path algorithm

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

int dist[] = new int[V]; // The output array. dist[i] will hold shortest distance from src to i

boolean sptSet[] = new boolean[V]; // sptSet[i] will be true if vertex i is included in shortest path tree

// Initialize all distances as INFINITE and sptSet[] as false

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

dist[i] = Integer.MAX\_VALUE;

sptSet[i] = false;

}

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

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

// Pick the minimum distance vertex from the set of vertices not yet processed

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex

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

// Update dist[v] only if it's not in sptSet, there is an edge from u to v,

// and total weight of path from src to v through u is smaller than current dist[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];

}

}

}

// Print the constructed distance array

printSolution(dist);

}

public static void main(String[] args) {

Dijkstra t = new Dijkstra();

/\* Example graph represented as adjacency matrix \*/

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

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

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

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

{7, 0, 6, 0, 0},

{0, 3, 9, 2, 0}

};

int source = 0;

t.dijkstra(graph, source);

}

}