TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**JAVA**

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÀI TẬP LỚN/ĐỒ ÁN CUỐI KÌ MÔN TOÁN TỔ HỢP VÀ ĐỒ THỊ**

**Combinatorial Optimization**

****

*Người thực hiện*: **HUỲNH GIA THIỆN – 51702186**

**LÊ HOÀNG PHÚC - 51702161**

Lớp **: 17050201**

Khoá  **: 21**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2019**

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**JAVA**

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÀI TẬP LỚN/ĐỒ ÁN CUỐI KÌ MÔN TOÁN TỔ HỢP VÀ ĐỒ THỊ**

**Combinatorial Optimization**

Người thực hiện: **HUỲNH GIA THIỆN - 51702186**

**LÊ HOÀNG PHÚC 51702161**

Lớp **: 17050201**

Khoá  **: 21**



**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2019**

MỤC LỤC

[MỤC LỤC 1](#_Toc6787253)

[DANH MỤC CÁC BẢNG BIỂU, HÌNH VẼ, ĐỒ THỊ 9](#_Toc6787254)

[I.INTRODUCTION 10](#_Toc6787255)

[1.1 Concept 10](#_Toc6787256)

[1.1.1 Maximum Network Flow 10](#_Toc6787257)

[1.1.2 Shortest Path 11](#_Toc6787258)

[1.1.3 Minimum Spanning Tree 12](#_Toc6787259)

[1.2 History 13](#_Toc6787260)

[1.2.1 Maximum Network Flow 13](#_Toc6787261)

[1.2.2 Shortest Path 13](#_Toc6787262)

[1.2.3 Minimum Spanning Tree 13](#_Toc6787263)

[II.STATE OF THE ART 13](#_Toc6787264)

[2.1 Maximum Network Flow 13](#_Toc6787265)

[2.2 Shortest Path 14](#_Toc6787266)

[2.3 Minimum Spanning Tree 15](#_Toc6787267)

[III.APPROACH 15](#_Toc6787268)

[IV.EXPEREMENTS AND RESULTS 15](#_Toc6787269)

[4.1 Maximum Network Flow 15](#_Toc6787270)

[4.1.1 Code 15](#_Toc6787271)

[4.1.2 Result 20](#_Toc6787272)

[20](#_Toc6787273)

[4.2 Shortest Path 20](#_Toc6787274)

[4.2.1 Code 20](#_Toc6787275)

[import java.util.\*; 20](#_Toc6787276)

[public class ShortestPath { 20](#_Toc6787277)

[private int dist[]; 20](#_Toc6787278)

[private Set<Integer> settled; 21](#_Toc6787279)

[private PriorityQueue<Node> pq; 21](#_Toc6787280)

[private int V; // Number of vertices 21](#_Toc6787281)

[List<List<Node> > ob; 21](#_Toc6787282)

[public ShortestPath(int V) 21](#_Toc6787283)

[{ 21](#_Toc6787284)

[this.V = V; 21](#_Toc6787285)

[dist = new int[V]; 21](#_Toc6787286)

[settled = new HashSet<Integer>(); 21](#_Toc6787287)

[pq = new PriorityQueue<Node>(V, new Node()); 21](#_Toc6787288)

[} 21](#_Toc6787289)

[public void dijkstra(List<List<Node> > ob, int source) 21](#_Toc6787290)

[{ 21](#_Toc6787291)

[this.ob = ob; 21](#_Toc6787292)

[for (int i = 0; i < V; i++) 21](#_Toc6787293)

[dist[i] = Integer.MAX\_VALUE; 21](#_Toc6787294)

[// Add source node to the priority queue 21](#_Toc6787295)

[pq.add(new Node(source, 0)); 21](#_Toc6787296)

[// Distance to the source is 0 22](#_Toc6787297)

[dist[0] = 0; 22](#_Toc6787298)

[while (settled.size() != V) { 22](#_Toc6787299)

[int u = pq.remove().node; 22](#_Toc6787300)

[settled.add(u); 22](#_Toc6787301)

[e\_Neighbours(u); 22](#_Toc6787302)

[} 22](#_Toc6787303)

[} 22](#_Toc6787304)

[// Function to process all the neighbours 22](#_Toc6787305)

[// of the passed node 22](#_Toc6787306)

[private void e\_Neighbours(int u) 22](#_Toc6787307)

[{ 22](#_Toc6787308)

[int edgeDistance = -1; 22](#_Toc6787309)

[int newDistance = -1; 22](#_Toc6787310)

[for (int i = 0; i < ob.get(u).size(); i++) { 22](#_Toc6787311)

[Node v = ob.get(u).get(i); 22](#_Toc6787312)

[if (!settled.contains(v.node)) { 22](#_Toc6787313)

[edgeDistance = v.cost; 23](#_Toc6787314)

[newDistance = dist[u] + edgeDistance; 23](#_Toc6787315)

[if (newDistance < dist[v.node]) 23](#_Toc6787316)

[dist[v.node] = newDistance; 23](#_Toc6787317)

[pq.add(new Node(v.node, dist[v.node])); 23](#_Toc6787318)

[} 23](#_Toc6787319)

[} 23](#_Toc6787320)

[} 23](#_Toc6787321)

[public static void main(String arg[]) 23](#_Toc6787322)

[{ 23](#_Toc6787323)

[int V = 5; 23](#_Toc6787324)

[int source = 0; 23](#_Toc6787325)

[List<List<Node> > ob = new ArrayList<List<Node> >(); 23](#_Toc6787326)

[// Initialize list for every node 23](#_Toc6787327)

[for (int i = 0; i < V; i++) { 23](#_Toc6787328)

[List<Node> item = new ArrayList<Node>(); 23](#_Toc6787329)

[ob.add(item); 24](#_Toc6787330)

[} 24](#_Toc6787331)

[// Inputs for the ShortestPath graph 24](#_Toc6787332)

[ob.get(0).add(new Node(1, 9)); 24](#_Toc6787333)

[ob.get(0).add(new Node(2, 6)); 24](#_Toc6787334)

[ob.get(0).add(new Node(3, 5)); 24](#_Toc6787335)

[ob.get(0).add(new Node(4, 3)); 24](#_Toc6787336)

[ob.get(2).add(new Node(1, 2)); 24](#_Toc6787337)

[ob.get(2).add(new Node(3, 4)); 24](#_Toc6787338)

[// Calculate the single source shortest path 24](#_Toc6787339)

[ShortestPath dpq = new ShortestPath(V); 24](#_Toc6787340)

[dpq.dijkstra(ob, source); 24](#_Toc6787341)

[// Print the shortest path to all the nodes 24](#_Toc6787342)

[// from the source node 24](#_Toc6787343)

[System.out.println("The shorted path from node :"); 24](#_Toc6787344)

[for (int i = 0; i < dpq.dist.length; i++) 24](#_Toc6787345)

[System.out.println("From point " + source + " to point " + i + " is " + dpq.dist[i]); 24](#_Toc6787346)

[} 25](#_Toc6787347)

[} 25](#_Toc6787348)

[class Node implements Comparator<Node> { 25](#_Toc6787349)

[public int node; 25](#_Toc6787350)

[public int cost; 25](#_Toc6787351)

[public Node() 25](#_Toc6787352)

[{ 25](#_Toc6787353)

[} 25](#_Toc6787354)

[public Node(int node, int cost) 25](#_Toc6787355)

[{ 25](#_Toc6787356)

[this.node = node; 25](#_Toc6787357)

[this.cost = cost; 25](#_Toc6787358)

[} 25](#_Toc6787359)

[@Override 25](#_Toc6787360)

[public int compare(Node node1, Node node2) 25](#_Toc6787361)

[{ 25](#_Toc6787362)

[if (node1.cost < node2.cost) 25](#_Toc6787363)

[return -1; 26](#_Toc6787364)

[if (node1.cost > node2.cost) 26](#_Toc6787365)

[return 1; 26](#_Toc6787366)

[return 0; 26](#_Toc6787367)

[} 26](#_Toc6787368)

[} 26](#_Toc6787369)

[4.2.2 Result 26](#_Toc6787370)

[4.3 Minimum Spanning Tree 26](#_Toc6787371)

[4.3.1 Code 26](#_Toc6787372)

[import java.util.ArrayList; 26](#_Toc6787373)

[import java.util.Arrays; 26](#_Toc6787374)

[import java.util.\*; 26](#_Toc6787375)

[import java.util.PriorityQueue; 26](#_Toc6787376)

[import java.util.Queue; 26](#_Toc6787377)

[public class Prim{ 26](#_Toc6787378)

[public static void main(String[] args) { 26](#_Toc6787379)

[int V = 9; 26](#_Toc6787380)

[Prim p = new Prim(); 26](#_Toc6787381)

[Graph graph = p.createGraph(V); 27](#_Toc6787382)

[p.addEdge(graph, 0, 1, 4); 27](#_Toc6787383)

[p.addEdge(graph, 0, 7, 8); 27](#_Toc6787384)

[p.addEdge(graph, 1, 2, 8); 27](#_Toc6787385)

[p.addEdge(graph, 1, 7, 11); 27](#_Toc6787386)

[p.addEdge(graph, 2, 3, 7); 27](#_Toc6787387)

[p.addEdge(graph, 2, 8, 2); 27](#_Toc6787388)

[p.addEdge(graph, 2, 5, 4); 27](#_Toc6787389)

[p.addEdge(graph, 3, 4, 9); 27](#_Toc6787390)

[p.addEdge(graph, 3, 5, 14); 27](#_Toc6787391)

[p.addEdge(graph, 4, 5, 10); 27](#_Toc6787392)

[p.addEdge(graph, 5, 6, 2); 27](#_Toc6787393)

[p.addEdge(graph, 6, 7, 1); 27](#_Toc6787394)

[p.addEdge(graph, 6, 8, 6); 27](#_Toc6787395)

[p.addEdge(graph, 7, 8, 7); 27](#_Toc6787396)

[p.printGraph(graph); 27](#_Toc6787397)

[System.out.println(); 27](#_Toc6787398)

[p.getPrim2(graph); 27](#_Toc6787399)

[} 27](#_Toc6787400)

[public class Node implements Comparable<Node> { 27](#_Toc6787401)

[int vertice, key; 28](#_Toc6787402)

[Node(int vertice, int key) { 28](#_Toc6787403)

[this.vertice = vertice; 28](#_Toc6787404)

[this.key = key; 28](#_Toc6787405)

[} 28](#_Toc6787406)

[@Override 28](#_Toc6787407)

[public int compareTo(Node o) { 28](#_Toc6787408)

[return this.key - o.key; 28](#_Toc6787409)

[} 28](#_Toc6787410)

[} 28](#_Toc6787411)

[public class AdjList { 28](#_Toc6787412)

[ArrayList<Node> nodes; 28](#_Toc6787413)

[} 28](#_Toc6787414)

[public class Graph { 28](#_Toc6787415)

[int V; 28](#_Toc6787416)

[AdjList[] adjLists; 28](#_Toc6787417)

[} 28](#_Toc6787418)

[public Graph createGraph(int v) { 28](#_Toc6787419)

[Graph graph = new Graph(); 28](#_Toc6787420)

[graph.V = v; 29](#_Toc6787421)

[graph.adjLists = new AdjList[v]; 29](#_Toc6787422)

[for (int i = 0; i < v; i++) { 29](#_Toc6787423)

[AdjList adjList = new AdjList(); 29](#_Toc6787424)

[adjList.nodes = new ArrayList<Node>(); 29](#_Toc6787425)

[graph.adjLists[i] = adjList; 29](#_Toc6787426)

[} 29](#_Toc6787427)

[return graph; 29](#_Toc6787428)

[} 29](#_Toc6787429)

[public void addEdge(Graph graph, int src, int dest, int key) { 29](#_Toc6787430)

[Node srcNode = new Node(src, key); 29](#_Toc6787431)

[Node destNode = new Node(dest, key); 29](#_Toc6787432)

[graph.adjLists[src].nodes.add(destNode); 29](#_Toc6787433)

[graph.adjLists[dest].nodes.add(srcNode); 29](#_Toc6787434)

[} 29](#_Toc6787435)

[public void printGraph(Graph graph) { 29](#_Toc6787436)

[System.out.println("Minimum Spanning tree: "); 29](#_Toc6787437)

[for (int i = 0; i < graph.V; i++) { 29](#_Toc6787438)

[System.out.print(i + " ->"); 29](#_Toc6787439)

[for (Node node : graph.adjLists[i].nodes) { 29](#_Toc6787440)

[System.out.print(" " + node.vertice); 30](#_Toc6787441)

[} 30](#_Toc6787442)

[System.out.println(); 30](#_Toc6787443)

[} 30](#_Toc6787444)

[} 30](#_Toc6787445)

[public void getPrim2(Graph graph) { 30](#_Toc6787446)

[Node keys[] = new Node[graph.V]; 30](#_Toc6787447)

[int parent[] = new int[graph.V]; 30](#_Toc6787448)

[boolean mstSet[] = new boolean[graph.V]; 30](#_Toc6787449)

[for (int i = 0; i < graph.V; i++) { 30](#_Toc6787450)

[keys[i] = new Node(i, Integer.MAX\_VALUE); 30](#_Toc6787451)

[parent[i] = -1; 30](#_Toc6787452)

[mstSet[i] = false; 30](#_Toc6787453)

[} 30](#_Toc6787454)

[keys[0].key = 0; 30](#_Toc6787455)

[Queue<Node> pQueue = new PriorityQueue<>(); 30](#_Toc6787456)

[pQueue.addAll(Arrays.asList(keys)); 30](#_Toc6787457)

[while (pQueue.size() > 1) { 30](#_Toc6787458)

[Node u = pQueue.remove(); 31](#_Toc6787459)

[mstSet[u.vertice] = true; 31](#_Toc6787460)

[for (Node node : graph.adjLists[u.vertice].nodes) { 31](#_Toc6787461)

[if (mstSet[node.vertice] == false && node.key < keys[node.vertice].key) { 31](#_Toc6787462)

[pQueue.remove(keys[node.vertice]); 31](#_Toc6787463)

[keys[node.vertice].key = node.key; 31](#_Toc6787464)

[parent[node.vertice] = u.vertice; 31](#_Toc6787465)

[pQueue.add(keys[node.vertice]); 31](#_Toc6787466)

[} 31](#_Toc6787467)

[} 31](#_Toc6787468)

[} 31](#_Toc6787469)

[print\_mst(keys, parent, graph); 31](#_Toc6787470)

[} 31](#_Toc6787471)

[public void print\_mst(Node[] keys, int[] parent, Graph graph) { 31](#_Toc6787472)

[for (int i = 1; i < graph.V; i++) { 31](#_Toc6787473)

[System.out.println("From point: " + parent[keys[i].vertice] + " to point : " + keys[i].vertice + " Weight = " +keys[i].key); 31](#_Toc6787474)

[} 32](#_Toc6787475)

[} 32](#_Toc6787476)

[} 32](#_Toc6787477)

[4.3.2 Result 32](#_Toc6787478)

[V.CONCLUSION 32](#_Toc6787479)

DANH MỤC CÁC BẢNG BIỂU, HÌNH VẼ, ĐỒ THỊ

**DANH MỤC HÌNH**

[Image 1. 1: Maximum NetWork Flow 4](file:///E:\TTHVDT\HuynhGiaThien-51702186-LeHoangPhuc-51702161.docx.docx#_Toc6785624)

[Image 1. 2 : Shortest Path 5](#_Toc6785625)

[Image 1. 3: Minimum Spanning Tree 6](file:///E:\TTHVDT\HuynhGiaThien-51702186-LeHoangPhuc-51702161.docx.docx#_Toc6785626)

[Image 4. 1: Maximum Network Flow result 13](#_Toc6785634)

[Image 4. 2: Shortest Path result 19](file:///E:\TTHVDT\HuynhGiaThien-51702186-LeHoangPhuc-51702161.docx.docx#_Toc6785635)

[Image 4. 3: Minmum Spanning Tree result 25](file:///E:\TTHVDT\HuynhGiaThien-51702186-LeHoangPhuc-51702161.docx.docx#_Toc6785636)

I.INTRODUCTION

In research activity, Combinatorial Optimization is a topic that covers finding an optimal method from a set of finite implementations. In many such issues, comprehensive search is not possible. It works on the domain of such optimization problems, in which the set of possible solutions is discrete or mutable of discrete, with the goal of finding the best solution.

Combinatorial Optimization is a subset of mathematical optimization related to operational research, algorithm theory and complex computational theory. It has important applications in a number of areas, including artificial intelligence, machine learning, auction theory and software technology.

1.1 Concept

1.1.1 Maximum Network Flow

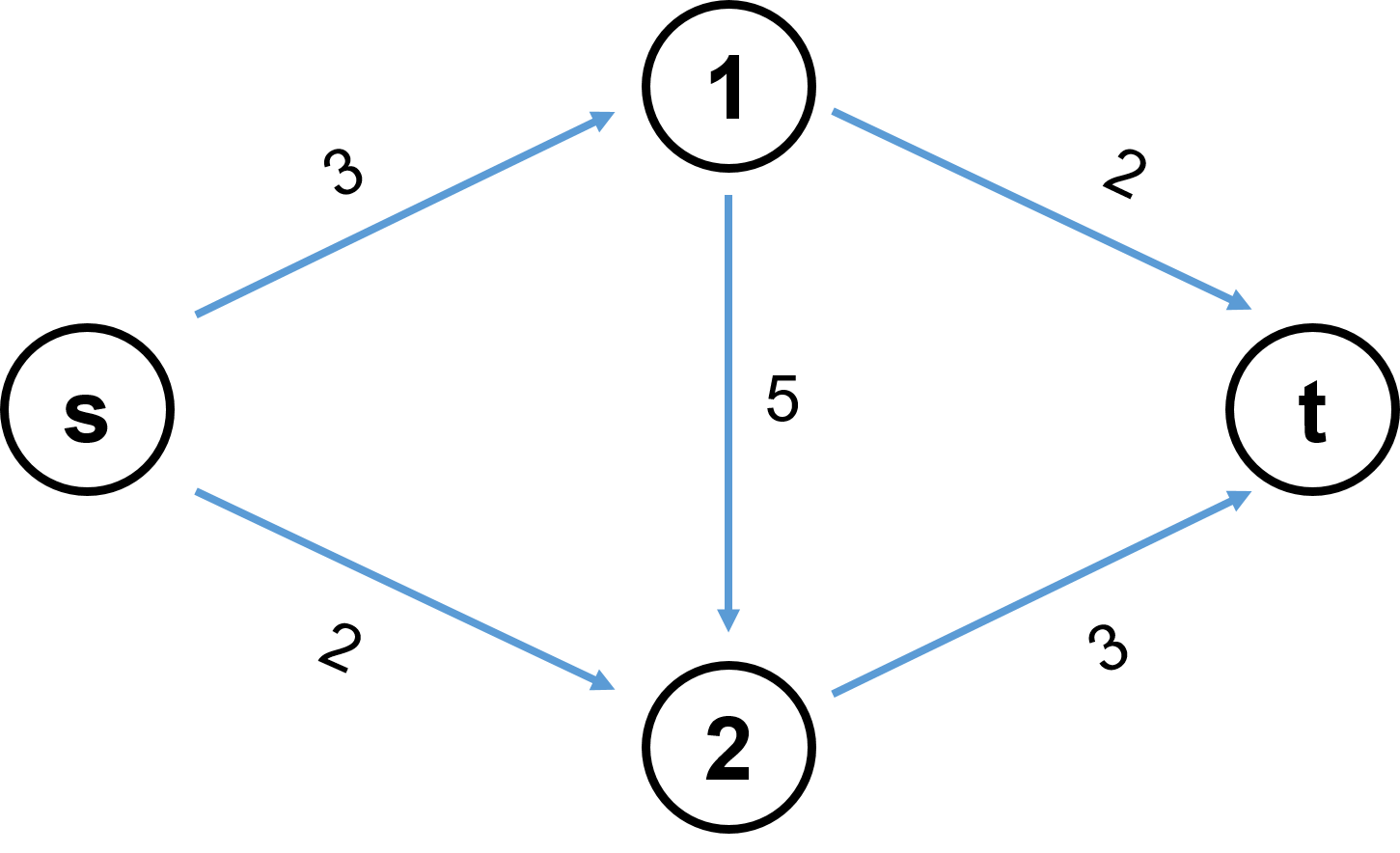
Maximum flow problems related to finding a feasible stream through a single traffic network, is a flow with maximum traffic.

Image 1. 1: Maximum NetWork Flow

1.1.2 Shortest Path

The shortest path is the problem of finding the path between two vertices (or nodes) in the graph so that the total weight of its constituent edges is the lowest.

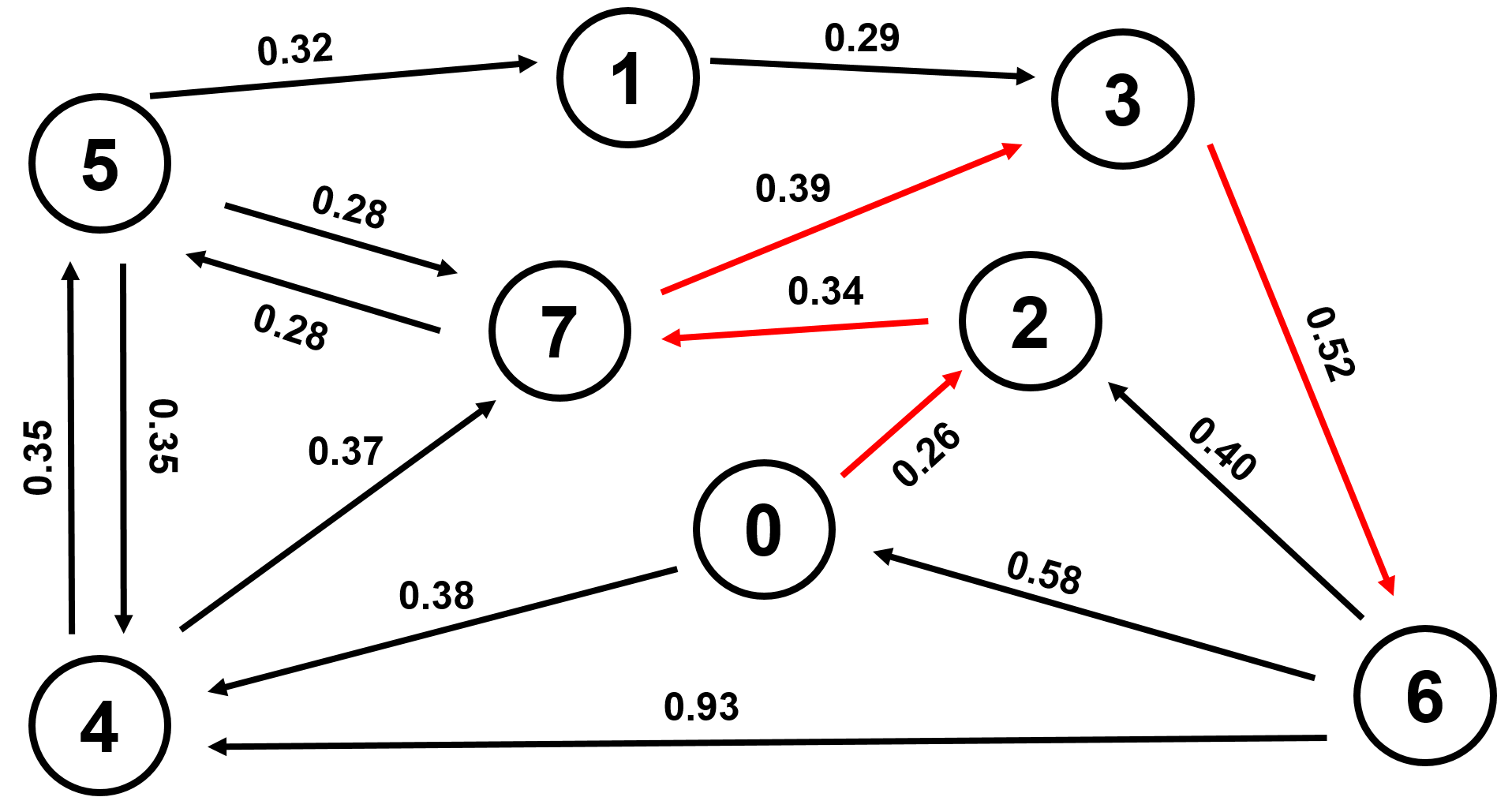
****Shorter paths can be modeled as a special case of the shortest path problem in a diagram, where vertices correspond to intersections and edges corresponding to road segments, each weighting in length. of the segment.

Image 1. 2 : Shortest Path

1.1.3 Minimum Spanning Tree

Minimum Spanning Tree is a subset of the edges of a weightless scalar graph that connects all vertices together, with no cycles and with the total possible minimum weight.

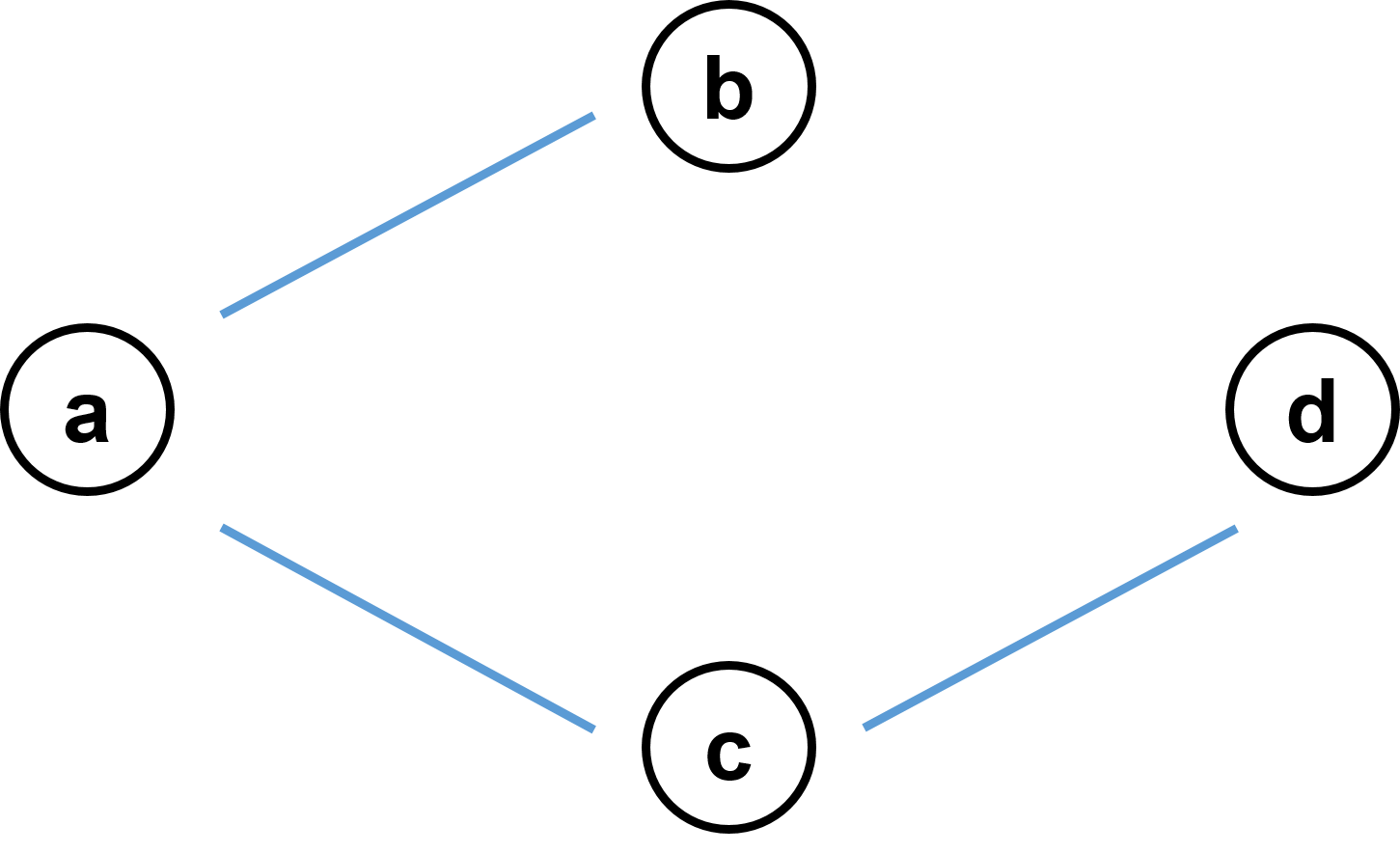
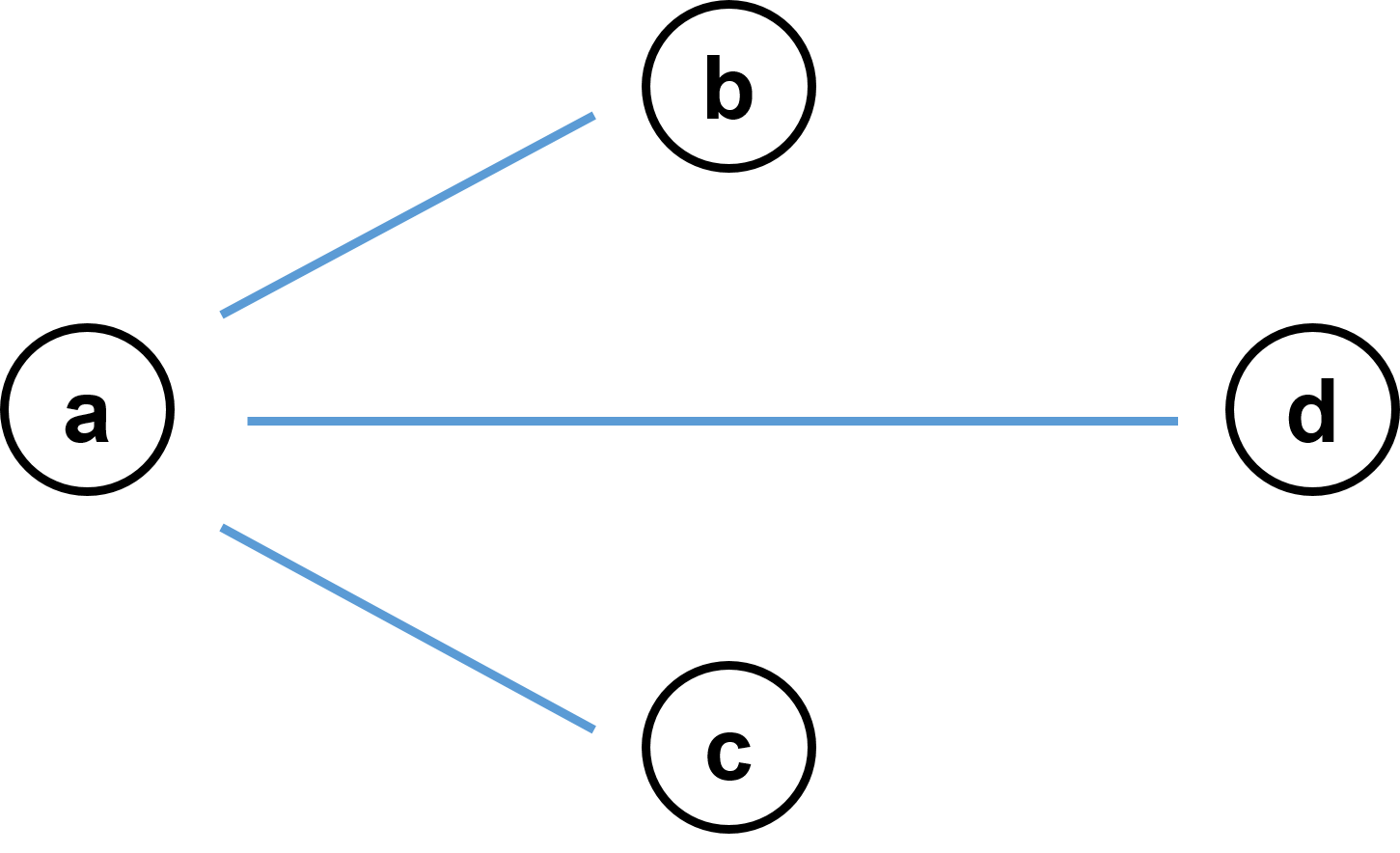


Image 1. 3: Minimum Spanning Tree

1.2 History

1.2.1 Maximum Network Flow

Maximum flow problem was first reported by TE Harris and FS Ross in 1945 to model railway traffic status in the Soviet Union. It was part of Maximum NetWork Flow. And then 10 years later, Lester R. Ford , Jr. and Delbert R. Fulkerson relied on that model to create the first algorithm, an algorithm called Ford – Fulkerson.

1.2.2 Shortest Path

Shortest Path is a widely useful problem-solving model for controlling robots, texture mapping, and typesetting in TEX, etc. The first Shortest Path algorithm was reported by Shimbel in 1954.

1.2.3 Minimum Spanning Tree

The Minimum Spanning Tree algorithm was first discovered by Czech scientist Otakar Borůvka in 1926, in order to obtain an effective electrical coverage of Moravia.

The fastest current Spanning Tree algorithm currently developed by Bernard Chazelle. Algorithms based on soft heap, approximate priority queues

II.STATE OF THE ART

2.1 Maximum Network Flow

The newest algorithm of maximum network traffic is Dinic's Algorithm.

In 1970, Y. A. Dinitz developed a faster algorithm to calculate the maximum flow across networks. It includes building level charts and residual charts and finding incremental paths with blocking flow.

The level chart is one where the value of each node is the shortest distance from the source.

Block the flow including finding a new path from the bottleneck.

Residual charts and increased paths are discussed previously.

The dummy code for Dinic's algorithm is given below.

The required inputs are network graph G, power node S and sink node T.

Function: DinicMaxFlow (G, S Button, T Button):

Initialize threads at all edges to 0, F = 0

Build a level chart

while (there exists an increasing path in the level chart):

Find the blocking line f in the graph

F = F + f

Update level chart

back to F

Update level chart including removing edges with full capacity. Remove nodes that do not sink and are dead ends. An illustration of the operation of the Dinic algorithm is shown below with the help of diagrams.

2.2 Shortest Path

Dijkstra’s algorithm given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

2.3 Minimum Spanning Tree

Kruskal's algorithm is the minimum spanning tree algorithm, finding the smallest weight edge that can connect any two trees in the forest. This is a greedy algorithm in graph theory because it finds a minimum spanning tree for a connected weight graph, adding incremental cost arcs at each step. If the chart is not connected, it will find a minimum spanning forest (a minimum spanning tree for each connected component).

III.APPROACH

IV.EXPEREMENTS AND RESULTS

4.1 Maximum Network Flow

4.1.1 Code

// Java program for implementation of Ford Fulkerson algorithm

import java.util.\*;

import java.lang.\*;

import java.io.\*;

import java.util.LinkedList;

class MaximumNetworkFlow

{

static final int V = 6;

private int maxFlowvalue;

private int[][] graph;

/\*\*

\* @return the maxFlowvalue

\*/

public int getMaxFlow\_value() {

return maxFlowvalue;

}

private boolean bfs(int rGraph[][], int s, int t, int parent[])

{

boolean visited[] = new boolean[V];

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

visited[i]=false;

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

queue.add(s);

visited[s] = true;

parent[s]=-1;

while (queue.size()!=0)

{

int u = queue.poll();

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

{

if (visited[v]==false && rGraph[u][v] > 0)

{

queue.add(v);

parent[v] = u;

visited[v] = true;

}

}

}

return (visited[t] == true);

}

public void fordFulkerson(int graph[][], int V)

{

this.graph = graph;

int s = 0, t = V - 1;

int u, v;

int rGraph[][] = new int[V][V];

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

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

rGraph[u][v] = graph[u][v];

int parent[] = new int[V];

int max\_flow = 0;

while (bfs(rGraph, s, t, parent))

{

int path\_flow = Integer.MAX\_VALUE;

for (v=t; v!=s; v=parent[v])

{

u = parent[v];

path\_flow = Math.min(path\_flow, rGraph[u][v]);

}

for (v=t; v != s; v=parent[v])

{

u = parent[v];

rGraph[u][v] -= path\_flow;

rGraph[v][u] += path\_flow;

}

max\_flow += path\_flow;

}

maxFlowvalue = max\_flow;

}

public String flowofMax()

{

String flow = "0";

boolean visited[] = new boolean[V];

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

visited[i]=false;

int current = 0;

visited[current] = true;

while(current < V - 1)

{

int max = graph[current][0];

int v = 0;

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

if(graph[current][i] > max && visited[i] == false){

max = graph[current][i];

v = i;

}

}

current = v;

visited[current] = true;

flow += " -> " + current;

}

return flow;

}

public static void main (String[] args) throws java.lang.Exception

{

//call run() before call another method

int graph[][] =new int[][] { {0, 16, 18, 0, 0, 0},

{0, 0, 10, 12, 0, 0},

{0, 4, 0, 0, 14, 0},

{0, 0, 9, 8, 0, 20},

{0, 2, 0, 7, 0, 4},

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

};

MaximumNetworkFlow m = new MaximumNetworkFlow();

m.fordFulkerson(graph, V);

System.out.println("The maximum possible flow is "

+m.getMaxFlow\_value());

System.out.println("The flow of maximum value is "

+m.flowofMax());

}

}

4.1.2 Result

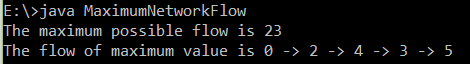


Image 4. 1: Maximum Network Flow result

4.2 Shortest Path

4.2.1 Code

import java.util.\*;

public class ShortestPath {

private int dist[];

private Set<Integer> settled;

private PriorityQueue<Node> pq;

private int V; // Number of vertices

List<List<Node> > ob;

public ShortestPath(int V)

{

this.V = V;

dist = new int[V];

settled = new HashSet<Integer>();

pq = new PriorityQueue<Node>(V, new Node());

}

public void dijkstra(List<List<Node> > ob, int source)

{

this.ob = ob;

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

dist[i] = Integer.MAX\_VALUE;

// Add source node to the priority queue

pq.add(new Node(source, 0));

// Distance to the source is 0

dist[0] = 0;

while (settled.size() != V) {

int u = pq.remove().node;

settled.add(u);

e\_Neighbours(u);

}

}

// Function to process all the neighbours

// of the passed node

private void e\_Neighbours(int u)

{

int edgeDistance = -1;

int newDistance = -1;

for (int i = 0; i < ob.get(u).size(); i++) {

Node v = ob.get(u).get(i);

if (!settled.contains(v.node)) {

edgeDistance = v.cost;

newDistance = dist[u] + edgeDistance;

if (newDistance < dist[v.node])

dist[v.node] = newDistance;

pq.add(new Node(v.node, dist[v.node]));

}

}

}

public static void main(String arg[])

{

int V = 5;

int source = 0;

List<List<Node> > ob = new ArrayList<List<Node> >();

// Initialize list for every node

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

List<Node> item = new ArrayList<Node>();

ob.add(item);

}

// Inputs for the ShortestPath graph

ob.get(0).add(new Node(1, 9));

ob.get(0).add(new Node(2, 6));

ob.get(0).add(new Node(3, 5));

ob.get(0).add(new Node(4, 3));

ob.get(2).add(new Node(1, 2));

ob.get(2).add(new Node(3, 4));

// Calculate the single source shortest path

ShortestPath dpq = new ShortestPath(V);

dpq.dijkstra(ob, source);

// Print the shortest path to all the nodes

// from the source node

System.out.println("The shorted path from node :");

for (int i = 0; i < dpq.dist.length; i++)

System.out.println("From point " + source + " to point " + i + " is " + dpq.dist[i]);

}

}

class Node implements Comparator<Node> {

public int node;

public int cost;

public Node()

{

}

public Node(int node, int cost)

{

this.node = node;

this.cost = cost;

}

@Override

public int compare(Node node1, Node node2)

{

if (node1.cost < node2.cost)

return -1;

if (node1.cost > node2.cost)

return 1;

return 0;

}

}

4.2.2 Result

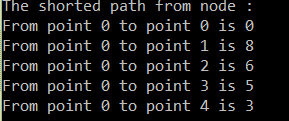
4.3 Minimum Spanning Tree

Image 4. 2: Shortest Path result

4.3.1 Code

import java.util.ArrayList;

import java.util.Arrays;

import java.util.\*;

import java.util.PriorityQueue;

import java.util.Queue;

public class Prim{

public static void main(String[] args) {

int V = 9;

Prim p = new Prim();

Graph graph = p.createGraph(V);

p.addEdge(graph, 0, 1, 4);

p.addEdge(graph, 0, 7, 8);

p.addEdge(graph, 1, 2, 8);

p.addEdge(graph, 1, 7, 11);

p.addEdge(graph, 2, 3, 7);

p.addEdge(graph, 2, 8, 2);

p.addEdge(graph, 2, 5, 4);

p.addEdge(graph, 3, 4, 9);

p.addEdge(graph, 3, 5, 14);

p.addEdge(graph, 4, 5, 10);

p.addEdge(graph, 5, 6, 2);

p.addEdge(graph, 6, 7, 1);

p.addEdge(graph, 6, 8, 6);

p.addEdge(graph, 7, 8, 7);

p.printGraph(graph);

System.out.println();

p.getPrim2(graph);

}

public class Node implements Comparable<Node> {

int vertice, key;

Node(int vertice, int key) {

this.vertice = vertice;

this.key = key;

}

@Override

public int compareTo(Node o) {

return this.key - o.key;

}

}

public class AdjList {

ArrayList<Node> nodes;

}

public class Graph {

int V;

AdjList[] adjLists;

}

public Graph createGraph(int v) {

Graph graph = new Graph();

graph.V = v;

graph.adjLists = new AdjList[v];

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

AdjList adjList = new AdjList();

adjList.nodes = new ArrayList<Node>();

graph.adjLists[i] = adjList;

}

return graph;

}

public void addEdge(Graph graph, int src, int dest, int key) {

Node srcNode = new Node(src, key);

Node destNode = new Node(dest, key);

graph.adjLists[src].nodes.add(destNode);

graph.adjLists[dest].nodes.add(srcNode);

}

public void printGraph(Graph graph) {

System.out.println("Minimum Spanning tree: ");

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

System.out.print(i + " ->");

for (Node node : graph.adjLists[i].nodes) {

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

}

System.out.println();

}

}

public void getPrim2(Graph graph) {

Node keys[] = new Node[graph.V];

int parent[] = new int[graph.V];

boolean mstSet[] = new boolean[graph.V];

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

keys[i] = new Node(i, Integer.MAX\_VALUE);

parent[i] = -1;

mstSet[i] = false;

}

keys[0].key = 0;

Queue<Node> pQueue = new PriorityQueue<>();

pQueue.addAll(Arrays.asList(keys));

while (pQueue.size() > 1) {

Node u = pQueue.remove();

mstSet[u.vertice] = true;

for (Node node : graph.adjLists[u.vertice].nodes) {

if (mstSet[node.vertice] == false && node.key < keys[node.vertice].key) {

pQueue.remove(keys[node.vertice]);

keys[node.vertice].key = node.key;

parent[node.vertice] = u.vertice;

pQueue.add(keys[node.vertice]);

}

}

}

print\_mst(keys, parent, graph);

}

public void print\_mst(Node[] keys, int[] parent, Graph graph) {

for (int i = 1; i < graph.V; i++) {

System.out.println("From point: " + parent[keys[i].vertice] + " to point : " + keys[i].vertice + " Weight = " +keys[i].key);

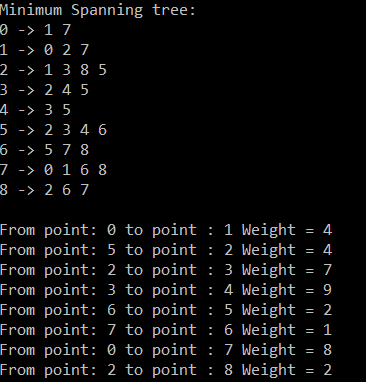
}

}

}

4.3.2 Result

Image 4. 3: Minmum Spanning Tree result

V.CONCLUSION

The best algorithm to use won’t be left up to you to decide, rather it will be dependant upon the type of graph you are using and the shortest path problem that is being solved.

When it comes down to it, many aspects of these algorithms are the same, however, when you look at performance and use, that’s where the differences come to light. There for, the best algorithm of Combinatorial Optimization is up your problem.

**TÀI LIỆU THAM KHẢO**

1. <https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/>

2. <https://www.hackerearth.com/fr/practice/algorithms/graphs/>