GRAPH ALGORITHMS
SINGLE-SOURCE SHORTEST PATH AND MINIMUM SPANNING TREE

1. SINGLE-SOURCE SHORTEST PATH:

The shortest path refers to finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. The single-source shortest path is a concept in which the shortest path with minimal weights is calculated, starting from a particular vertex as the source. The shortest paths can be calculated for both undirected and directed graphs.

1.a.DIJKSTRA’s ALGORITHM:

This algorithm is one of the famous methods to calculate the shortest path from a single-source. For the implementation, a priority queue is used here. Dijkstra's algorithm initially marks the distance from the source vertex to every other vertex(V) on the graph with infinity(implying those vertices have not been visited yet). Now, based on the available edges(E), it calculates the minimum cost required to visit each vertex. Selecting the vertex with low cost at every iteration, the process continues by updating the cost of each vertex. The final iteration shows the cost of traversing each vertex from the source vertex.

1.b.DATA STRUCTURE:

As said above, a priority queue is used here. The reason is that the operations “extract-min” and “decrease key” matches with the specialty of this data structure.

i.Main: Here, the user is asked to enter the path of the input file and inputs it into a queue. This main class takes a choice between the algorithm that is to be implemented. Based on the answer, the vertices, edges and the type of graph(directed or undirected) are all displayed. In case of choosing the single-source shortest path algorithm, it prompts us to enter the source vertex.

ii.Edge: Consists of two variable(vertices) which stores the vertex information and also a weight associated with that edge.

iii.Vertex: Prints the least cost path between two vertices by comparing the costs.

iv.Graph: A hash map is used here to map the edges with the vertices in the main class. This function also uses the priority queue to add each vertex into the queue. Eventually, it also prints the costs from the source vertex to the other vertices.

1.c.RUNTIME:

O(E log V), where E and V represent the Edges and Vertices respectively.

1.d.INSTRUCTIONS FOR EXECUTING DIJKSTRA’s:

On executing the program, it asks for the location of the input file. So save all the input files in a path such as “C:\Users\bavud\Desktop\Algorithms P2\Project 2\input1”. Next, we will be choosing between Dijkstra’s(single-source shortest path) or Kruskal’s(minimum spanning tree - explained below). In the case of Dijkstra’s, it asks for the starting vertex and upon entering lists down all the costs from the source to all the vertices.

Sample Input/Output for Dijkstra’s Algorithm:

Input 1:

7	13	U
A	B	1
A	C	2
A	G	4

B	C	1
B	D	3
B	E	2
C	D	1
C	E	2
D	E	4
D	F	3
E	F	3
G	E	11
G	F	7

SOURCE: A

```
Enter path of the text file to be given as Input:
C:\Users\bavud\Desktop\Algorithms P2\Project 2\input1
Enter:
 1. Dijkstra's Algorithm
 2. Kruskal's Algorithm
1
Number of vertices: 7
Number of edges: 13
Type of graph: U
Edge{v1='A', v2='B', weight=1}
Edge{v1='A', v2='C', weight=2}
Edge{v1='A', v2='G', weight=4}
Edge{v1='B', v2='C', weight=1}
Edge{v1='B', v2='D', weight=3}
Edge{v1='B', v2='E', weight=2}
Edge{v1='C', v2='D', weight=1}
Edge{v1='C', v2='E', weight=2}
Edge{v1='D', v2='E', weight=4}
Edge{v1='D', v2='F', weight=3}
Edge{v1='E', v2='F', weight=3}
Edge{v1='G', v2='E', weight=11}
Edge{v1='G', v2='F', weight=7}
Enter start node:
A
Line 1 7  13 U
A
A -> B(1)
A -> C(2)
```

```
A -> C(2) -> D(3)
A -> B(1) -> E(3)
A -> C(2) -> D(3) -> F(6)
A -> G(4)

SOURCE: C
Enter path of the text file to be given as Input:
C:\Users\bavud\Desktop\Algorithms P2\Project 2\input1
Enter:
 1. Dijkstra's Algorithm
 2. Kruskal's Algorithm
1
Number of vertices: 7
Number of edges: 13
Type of graph: U
Edge{v1='A', v2='B', weight=1}
Edge{v1='A', v2='C', weight=2}
Edge{v1='A', v2='G', weight=4}
Edge{v1='B', v2='C', weight=1}
Edge{v1='B', v2='D', weight=3}
Edge{v1='B', v2='E', weight=2}
Edge{v1='C', v2='D', weight=1}
Edge{v1='C', v2='E', weight=2}
Edge{v1='D', v2='E', weight=4}
Edge{v1='D', v2='F', weight=3}
Edge{v1='E', v2='F', weight=3}
Edge{v1='G', v2='E', weight=11}
Edge{v1='G', v2='F', weight=7}
Enter start node:
C
Line 1 7  13 U
C -> A(2)
C -> B(1)
C
C -> D(1)
C -> E(2)
C -> D(1) -> F(4)
C -> A(2) -> G(6)
```

Input 2:

9	16	D
A	B	3
B	C	11
B	D	9

C	A	6
D	C	7
D	E	1
D	F	5
E	B	6
E	H	4
E	G	8
F	C	2
F	E	5
G	H	11
G	I	9
H	I	12
H	F	2

SOURCE: A

```
Enter path of the text file to be given as Input:
C:\Users\bavud\Desktop\Algorithms P2\Project 2\input2
Enter:
 1. Dijkstra's Algorithm
 2. Kruskal's Algorithm
1
Number of vertices: 9
Number of edges: 16
Type of graph: D
Edge{v1='A', v2='B', weight=3}
Edge{v1='B', v2='C', weight=11}
Edge{v1='B', v2='D', weight=9}
Edge{v1='C', v2='A', weight=6}
Edge{v1='D', v2='C', weight=7}
Edge{v1='D', v2='E', weight=1}
Edge{v1='D', v2='F', weight=5}
Edge{v1='E', v2='B', weight=6}
Edge{v1='E', v2='H', weight=4}
Edge{v1='E', v2='G', weight=8}
Edge{v1='F', v2='C', weight=2}
Edge{v1='F', v2='E', weight=5}
Edge{v1='G', v2='H', weight=11}
Edge{v1='G', v2='I', weight=9}
Edge{v1='H', v2='I', weight=12}
Edge{v1='H', v2='F', weight=2}
Enter start node:
```

```
A
Line 1 9  16 D
A
A -> B(3)
A -> B(3) -> C(14)
A -> B(3) -> D(12)
A -> B(3) -> D(12) -> E(13)
A -> B(3) -> D(12) -> F(17)
A -> B(3) -> D(12) -> E(13) -> G(21)
A -> B(3) -> D(12) -> E(13) -> H(17)
A -> B(3) -> D(12) -> E(13) -> H(17) -> I(29)
```

SOURCE: F

```
Enter path of the text file to be given as Input:
C:\Users\bavud\Desktop\Algorithms P2\Project 2\input2
Enter:
 1. Dijkstra's Algorithm
 2. Kruskal's Algorithm
1
Number of vertices: 9
Number of edges: 16
Type of graph: D
Edge{v1='A', v2='B', weight=3}
Edge{v1='B', v2='C', weight=11}
Edge{v1='B', v2='D', weight=9}
Edge{v1='C', v2='A', weight=6}
Edge{v1='D', v2='C', weight=7}
Edge{v1='D', v2='E', weight=1}
Edge{v1='D', v2='F', weight=5}
Edge{v1='E', v2='B', weight=6}
Edge{v1='E', v2='H', weight=4}
Edge{v1='E', v2='G', weight=8}
Edge{v1='F', v2='C', weight=2}
Edge{v1='F', v2='E', weight=5}
Edge{v1='G', v2='H', weight=11}
Edge{v1='G', v2='I', weight=9}
Edge{v1='H', v2='I', weight=12}
Edge{v1='H', v2='F', weight=2}
Enter start node:
F
Line 1 9  16 D
F -> C(2) -> A(8)
F -> E(5) -> B(11)
F -> C(2)
F -> E(5) -> B(11) -> D(20)
F -> E(5)
F
F -> E(5) -> G(13)
F -> E(5) -> H(9)
F -> E(5) -> H(9) -> I(21)
```

Input 3:

11	20	D
A	B	7
A	G	9
A	H	10
A	I	6
A	J	12
A	K	3
B	C	10
B	E	2
B	K	7
C	D	8
C	K	5
D	E	16
D	F	12
E	F	10
E	G	6
F	G	8
G	H	6
H	I	2
I	J	11
J	K	7

SOURCE: A

```
Enter path of the text file to be given as Input:
C:\Users\bavud\Desktop\Algorithms P2\Project 2\input3
Enter:
 1. Dijkstra's Algorithm
 2. Kruskal's Algorithm
1
Number of vertices: 11
Number of edges: 20
```

```
Type of graph: D
Edge{v1='A', v2='B', weight=7}
Edge{v1='A', v2='G', weight=9}
Edge{v1='A', v2='H', weight=10}
Edge{v1='A', v2='I', weight=6}
Edge{v1='A', v2='J', weight=12}
Edge{v1='A', v2='K', weight=3}
Edge{v1='B', v2='C', weight=10}
Edge{v1='B', v2='E', weight=2}
Edge{v1='B', v2='K', weight=7}
Edge{v1='C', v2='D', weight=8}
Edge{v1='C', v2='K', weight=5}
Edge{v1='D', v2='E', weight=16}
Edge{v1='D', v2='F', weight=12}
Edge{v1='E', v2='F', weight=10}
Edge{v1='E', v2='G', weight=6}
Edge{v1='F', v2='G', weight=8}
Edge{v1='G', v2='H', weight=6}
Edge{v1='H', v2='I', weight=2}
Edge{v1='I', v2='J', weight=11}
Edge{v1='J', v2='K', weight=7}
Enter start node:
A
Line 1 11 20 D
A
A -> B(7)
A -> B(7) -> C(17)
A -> B(7) -> C(17) -> D(25)
A -> B(7) -> E(9)
A -> B(7) -> E(9) -> F(19)
A -> G(9)
A -> H(10)
A -> I(6)
A -> J(12)
A -> K(3)
```

SOURCE: I

```
Enter path of the text file to be given as Input:
C:\Users\bavud\Desktop\Algorithms P2\Project 2\input3
Enter:
 1. Dijkstra's Algorithm
 2. Kruskal's Algorithm
1
Number of vertices: 11
Number of edges: 20
Type of graph: D
Edge{v1='A', v2='B', weight=7}
Edge{v1='A', v2='G', weight=9}
```

```
Edge{v1='A', v2='H', weight=10}
Edge{v1='A', v2='I', weight=6}
Edge{v1='A', v2='J', weight=12}
Edge{v1='A', v2='K', weight=3}
Edge{v1='B', v2='C', weight=10}
Edge{v1='B', v2='E', weight=2}
Edge{v1='B', v2='K', weight=7}
Edge{v1='C', v2='D', weight=8}
Edge{v1='C', v2='K', weight=5}
Edge{v1='D', v2='E', weight=16}
Edge{v1='D', v2='F', weight=12}
Edge{v1='E', v2='F', weight=10}
Edge{v1='E', v2='G', weight=6}
Edge{v1='F', v2='G', weight=8}
Edge{v1='G', v2='H', weight=6}
Edge{v1='H', v2='I', weight=2}
Edge{v1='I', v2='J', weight=11}
Edge{v1='J', v2='K', weight=7}
Enter start node:
I
Line 1 11 20 D
A(cannot reach)
B(cannot reach)
C(cannot reach)
D(cannot reach)
E(cannot reach)
F(cannot reach)
G(cannot reach)
H(cannot reach)
I
I -> J(11)
I -> J(11) -> K(18)
```

2. MINIMUM SPANNING TREE:

A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. Here, the final goal is to find out a minimum weight spanning tree. For calculating that, we have many algorithms, of which Kruskal’s is implemented below.

2.a.KRUSKAL’s ALGORITHM:

This algorithm is one of the famous methods to calculate the minimum weight spanning tree. For the implementation, a priority queue is used here. Kruskal's algorithm finds an edge of the least possible weight that connects any two trees in the forest(undirected graph). It follows the greedy algorithm approach. The algorithm finds a subset of the edges that forms a tree that includes every vertex(V), where the total weight of all the edges(E) in the tree is minimized(Until there are V-1 edges in the minimum spanning tree).

2.b.DATA STRUCTURE:

As said above, a priority queue is used here. The reason is that the operations “extract-min” and “decrease key” matches with the specialty of this data structure.

<u>i.Main:</u> Here, the user is asked to enter the path of the input file and inputs it into a queue. This main class takes a choice between the algorithm that is to be implemented. Based on the answer, the vertices, edges and the type of graph(directed or undirected) are all displayed.

<u>ii.Edge:</u> Consists of two variable(vertices) which stores the vertex information and also a weight associated with that edge.

<u>iii.Vertex:</u> Prints the least cost path between two vertices by comparing the costs.

<u>iv.Kruskal:</u> In a similar way to Dijkstra's, here as well, a hash map is used to map the vertices and edges with their weights plus the details of their parent and depth. An array list is taken into consideration and the edge with the least weight is appended to it after sorting the edges based on their weights. Once the final graph consists of V-1 edges, the algorithm prints the edges of the minimum spanning tree.

2.c.RUNTIME:

O(E log V) or O(E log E), where E and V represent the Edges and Vertices respectively.

2.d.INSTRUCTIONS FOR EXECUTING KRUSKAL's:

On executing the program, it asks for the location of the input file. So save all the input files in a path such as "C:\Users\bavud\Desktop\Algorithms P2\Project 2\input4". Next, we will be choosing between Dijkstra's(single-source shortest path) or Kruskal's(minimum spanning tree). The output lists down all the edges and the weights along with the vertices if the given input is an undirected graph. In the case of directed graphs, a minimum spanning tree cannot be formed.

<u>Sample Input/Output for Kruskal's Algorithm:</u>

<u>Input 1:</u>

7	13	U
A	B	1
A	C	2
A	G	4
B	C	1
B	D	3
B	E	2
C	D	1
C	E	2
D	E	4
D	F	3
E	F	3
G	E	11
G	F	7

```
Enter path of the text file to be given as Input:
C:\Users\bavud\Desktop\Algorithms P2\Project 2\input1
Enter:
1. Dijkstra's Algorithm
2. Kruskal's Algorithm
2
Number of vertices: 7
Number of edges: 13
Type of graph: U
Edge{v1='A', v2='B', weight=1}
Edge{v1='A', v2='C', weight=2}
Edge{v1='A', v2='G', weight=4}
Edge{v1='B', v2='C', weight=1}
Edge{v1='B', v2='D', weight=3}
Edge{v1='B', v2='E', weight=2}
Edge{v1='C', v2='D', weight=1}
Edge{v1='C', v2='E', weight=2}
Edge{v1='D', v2='E', weight=4}
Edge{v1='D', v2='F', weight=3}
Edge{v1='E', v2='F', weight=3}
Edge{v1='G', v2='E', weight=11}
Edge{v1='G', v2='F', weight=7}
minimum spanning tree has the edges: [Edge{v1='A', v2='B', weight=1}, Edge{v1='B', v2='C', weight=1},
Edge{v1='C', v2='D', weight=1}, Edge{v1='B', v2='E', weight=2}, Edge{v1='D', v2='F', weight=3},
Edge{v1='A', v2='G', weight=4}]
Cost of the MST is 12
```

Input 4:

10	15	U
A	B	6
A	J	2
A	G	2
B	D	10
D	J	2
D	H	13
F	H	5
F	I	2
F	E	1

H	I	7
G	E	15
G	F	9
I	E	1
H	C	3
J	G	7

```
Enter path of the text file to be given as Input:
C:\Users\bavud\Desktop\Algorithms P2\Project 2\input4
Enter:
1. Dijkstra's Algorithm
2. Kruskal's Algorithm
2
Number of vertices: 10
Number of edges: 15
Type of graph: U
Edge{v1='A', v2='B', weight=6}
Edge{v1='A', v2='J', weight=2}
Edge{v1='A', v2='G', weight=2}
Edge{v1='B', v2='D', weight=10}
Edge{v1='D', v2='J', weight=2}
Edge{v1='D', v2='H', weight=13}
Edge{v1='F', v2='H', weight=5}
Edge{v1='F', v2='I', weight=2}
Edge{v1='F', v2='E', weight=1}
Edge{v1='H', v2='I', weight=7}
Edge{v1='G', v2='E', weight=15}
Edge{v1='G', v2='F', weight=9}
Edge{v1='I', v2='E', weight=1}
Edge{v1='H', v2='C', weight=3}
Edge{v1='J', v2='G', weight=7}
minimum spanning tree has the edges: [Edge{v1='F', v2='E', weight=1}, Edge{v1='I', v2='E', weight=1},
Edge{v1='A', v2='J', weight=2}, Edge{v1='A', v2='G', weight=2}, Edge{v1='D', v2='J', weight=2},
Edge{v1='H', v2='C', weight=3}, Edge{v1='F', v2='H', weight=5}, Edge{v1='A', v2='B', weight=6},
Edge{v1='G', v2='F', weight=9}]
Cost of the MST is 31
```

Input 2:

9	16	D
A	B	3
B	C	11

B	D	9
C	A	6
D	C	7
D	E	1
D	F	5
E	B	6
E	H	4
E	G	8
F	C	2
F	E	5
G	H	11
G	I	9
H	I	12
H	F	2

```
Enter path of the text file to be given as Input:
C:\Users\bavud\Desktop\Algorithms P2\Project 2\input2
Enter:
1. Dijkstra's Algorithm
2. Kruskal's Algorithm
2
Number of vertices: 9
Number of edges: 16
Type of graph: D
Edge{v1='A', v2='B', weight=3}
Edge{v1='B', v2='C', weight=11}
Edge{v1='B', v2='D', weight=9}
Edge{v1='C', v2='A', weight=6}
Edge{v1='D', v2='C', weight=7}
Edge{v1='D', v2='E', weight=1}
Edge{v1='D', v2='F', weight=5}
Edge{v1='E', v2='B', weight=6}
Edge{v1='E', v2='H', weight=4}
Edge{v1='E', v2='G', weight=8}
Edge{v1='F', v2='C', weight=2}
Edge{v1='F', v2='E', weight=5}
Edge{v1='G', v2='H', weight=11}
Edge{v1='G', v2='I', weight=9}
Edge{v1='H', v2='I', weight=12}
Edge{v1='H', v2='F', weight=2}
```

Cannot find Minimum Spanning Tree for the directed graph

SOURCE CODE:

DIJKSTRA CLASS:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Scanner;

public class DijkstrasAlgorithm {
  static HashSet<String> verticesSet = new HashSet<>();
  static List<Edge> listOfEdges = new ArrayList<>();
  static String graphType, noOfEdges, noOfVertices;
  private static Edge[] e = null;
  private static boolean undirected = false;
  static String startNode = null;
  private static int k = 0;

  public static void main(String[] args) throws IOException {
        BufferedReader br = null;
        Scanner sc = new Scanner(System.in);
        Scanner sc1 = new Scanner(System.in);
        Scanner sc2 = new Scanner(System.in);
        System.out.println("Enter path of the text file to be given as Input: ");
        String url = sc.nextLine();
        System.out.println("Enter:\n 1. Dijkstra's Algorithm \n 2. Kruskal's Algorithm");
        int a = sc1.nextInt();
        br = new BufferedReader(new FileReader(url));

        try {
        StringBuilder sb = new StringBuilder();
        String line = br.readLine();
        String line1 = line;
        noOfVertices = line1.substring(0,2).trim();
        e = new Edge[Integer.parseInt(noOfVertices)];
        noOfEdges = line1.substring(3, 5).trim();
        graphType = line1.substring(6, 7).trim();
        System.out.println("Number of vertices: " + noOfVertices);
        System.out.println("Number of edges: " + noOfEdges);
        System.out.println("Type of graph: " + graphType);

        if (graphType.toUpperCase().equals("U")) {
        undirected = true;
        } else {
        undirected = false;
        }
        int num = Integer.parseInt(noOfEdges);

        e = new Edge[num];

        while (num != 0) {
        line = br.readLine();
        createNode(line);
        num--;
        }

        if (a == 1) {

        if ((line = (br.readLine())) != null) {
        startNode = line.trim().substring(0, 1);
        } else {
        System.out.println("Enter start node: ");
        startNode = sc2.nextLine();
        startNode.toLowerCase().trim();

        }

        System.out.println("Line 1 " + line1);

        Graph g = new Graph(e, undirected);
        g.dijkstra(startNode);
        g.printAllPaths();
        } else if (a == 2) {
        if (undirected) {
        KruskalsAlgorithm kruskal = new KruskalsAlgorithm();
            String x[] = verticesSet.toArray(new String[verticesSet.size()]);

        kruskal.KruskalAlgo(x, e);

        } else {
        System.out.println("Cannot find Minimum Spanning Tree for the directed graph");
        }
        }

        }

        catch (IOException e) {
        e.printStackTrace();
        } finally {
        br.close();
        }
    }

    private static void createNode(String temp) {
        String li = temp.trim();
```

```
            String s1 = li.substring(0, 1).trim();
            String s2 = li.substring(2, 3).trim();
            String s3 = li.substring(4, li.length()).trim();

            verticesSet.add(s1);
            verticesSet.add(s2);
            int s = Integer.valueOf(s3);

            e[k] = new Edge(s1, s2, s);

            System.out.println(e[k]);
            k++;
  }

}
```

EDGE CLASS:

```
public class Edge implements Comparable<Edge> {

  public final String v1, v2;
  public final int w;

  @Override
  public String toString() {
            return "Edge{" + "v1='" + v1 + '\'' + ", v2='" + v2 + '\''
                 + ", weight=" + w + '}';
  }

  public Edge(String v1, String v2, int w) {
            this.v1 = v1;
            this.v2 = v2;
            this.w = w;
  }

  public String fromVertex() {
            return this.v1;
  }

  public String toVertex() {
            return this.v2;
  }

  @Override
  public int compareTo(Edge o) {

            return w < o.w ? -1 : (w > o.w ? 1 : 0);
  }
}
```

GRAPH CLASS:

```
import java.util.*;
```

```
class Graph {
   private final Map<String, Vertex> graph;

   public Graph(Edge[] edges, boolean undirected) {
            graph = new HashMap<>(edges.length);

            // putting the vertices in the hash map "graph"
            for (Edge edge : edges) {
            if (!graph.containsKey(edge.v1))
            graph.put(edge.v1, new Vertex(edge.v1));
            if (!graph.containsKey(edge.v2))
            graph.put(edge.v2, new Vertex(edge.v2));
            }

            for (Edge e : edges) {
            // directed graph
        graph.get(e.v1).neighbours.put(graph.get(e.v2), e.w);
            if (undirected) {
            graph.get(e.v2).neighbours.put(graph.get(e.v1), e.w);

            }
            }
   }

   // Dijkstra to run on starting vertex
   public void dijkstra(String start) {
            if (!graph.containsKey(start)) {
            System.err.printf("Graph does not have start vertex \"%s\"\n",
                  start);
            return;
            }
            final Vertex source = graph.get(start);

            // Priority Queue to store vertices
            PriorityQueue<Vertex> q1 = new PriorityQueue<>();

            // set-up vertices
            for (Vertex v : graph.values()) {
               v.prev = (v == source) ? source : null;
            v.val = (v == source) ? 0 : Integer.MAX_VALUE; // setting to
                                   // infinite if it is
                                   // not source
            q1.add(v);

            }

            implementDijkstra(q1); // implementing Dijkstra algorithm on the priority

   }
```

```
// / Implementation of dijkstra's algorithm using a queue
private void implementDijkstra(final PriorityQueue<Vertex> q) {
        Vertex u, v;
        // Checking till queue is not empty
        while (!q.isEmpty()) {
        u = q.poll(); // vertex with shortest cost will be polled out of the
                // queue, for first time it will be start
        if (u.val == Integer.MAX_VALUE) {
          break; // we can ignore u (and any other remaining vertices)
        // since they are unreachable
        }

        // checking cost of each neighboring vertices
        for (Map.Entry<Vertex, Integer> a : u.neighbours.entrySet()) {
            v = a.getKey();

        final int tempDist = u.val + a.getValue();
        if (tempDist < v.val) { // if shortest path found to
                        // neighboring vertex
        q.remove(v); // remove v
        v.val = tempDist;
        v.prev = u;
        q.add(v); // add new v
        }
        }
        }
}

// prints a path from the vertex (source) to the end vertex
// This method was used for testing purpose only
public void printPath(String last) {
        if (!graph.containsKey(last)) {
        System.err
        .printf("Graph doesn't contain end vertex \"%s\"\n", last);
        return;
        }

        graph.get(last).print();
        System.out.println();
}

// prints all the path from the source vertex to other vertices
public void printAllPaths() {
        for (Vertex v : graph.values()) {
        v.print();
        System.out.println();
        }
}
```

```
}
```

VERTEX CLASS:

```
import java.util.HashMap;
import java.util.Map;

public class Vertex implements Comparable<Vertex> {
   Vertex prev = null;
   public int val = Integer.MAX_VALUE;
   public final String node;
   public final Map<Vertex, Integer> neighbours = new HashMap<>();



   public int compareTo(Vertex temp) {
      if (val == temp.val)
            return node.compareTo(temp.node);

            return Integer.compare(val, temp.val);
   }

   public void print() {
            if (this == this.prev) {
            System.out.printf("%s", this.node);
            } else if (this.prev == null) {
            System.out.printf("%s(cannot reach)", this.node);
            } else {
            this.prev.print();
            System.out.printf(" -> %s(%d)", this.node, this.val);
            }
   }

   public Vertex(String node) {
            this.node = node;
   }

   @Override
   public String toString()

   {
            return "(" + node + ", " + val + ")";
   }
}
```

KRUSKAL’s CLASS:

```
import java.util.*;
public class KruskalsAlgorithm {
    private static Map<String, String> PARENT;
    private static Map<String, Integer> DEPTH;
    private static int w=0;
    public static void initialize(String[] universe) {
        PARENT = new HashMap<String, String>();
        DEPTH = new HashMap<>();
        for (String x : universe) {
            PARENT.put(x, x);
            DEPTH.put(x, 1);
        }
    }
    public static String Find(String i) {
        String parent = PARENT.get(i);
        if (parent == i)
            return i;
        else
            return Find(parent);
    }
    public static void Union(String a, String b) {
        String parent1, parent2;
        while ((parent1 = PARENT.get(a)) != a) {
            a = parent1;
        }
        while ((parent2 = PARENT.get(b)) != b) {
            b = parent2;
        }
        int depthFirst = DEPTH.get(a), depthSecond = DEPTH.get(b);
        if (depthFirst > depthSecond) {
            PARENT.put(b, a);
            updateDepthUp(b);
        } else if (depthSecond > depthFirst) {
            PARENT.put(a, b);
            updateDepthUp(a);
        } else {
            PARENT.put(b, a);
            updateDepthUp(b);
        }
    }
    public static void updateDepthUp(String cur) {
        int tempDepth = DEPTH.get(cur);
        String curParent = PARENT.get(cur);
        int parentsDepth = DEPTH.get(curParent);
        if (!(tempDepth < parentsDepth || curParent == cur)) {
            DEPTH.put(curParent, tempDepth + 1);
            updateDepthUp(curParent);
        }
    }
    public ArrayList<Edge> KruskalAlgo(String[] vertices, Edge[] edges) {
        ArrayList<Edge> min = new ArrayList<>();
        initialize(vertices);
        Arrays.sort(edges);
        for (Edge e : edges) {
            if (Find(e.v1) != Find(e.v2)) {
                min.add(e);
                w=w+e.w;
                Union(e.v1, e.v2);
            }
        }
        // Display the minimum spanning tree
        System.out.println("minimum spanning tree has the edges: " + min);
        System.out.println("Cost of the MST is "+w);
        return min;
    }
}
```