



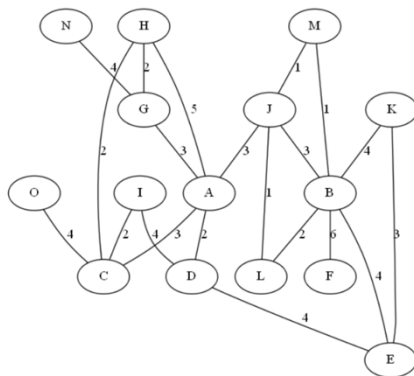
## **Railway Network DV1490**

# Table of contents

<b>1. Introduction</b>	<b>3</b>
<b>2. Description of the problem</b>	<b>3</b>
<b>2.1 Problem</b>	<b>3</b>
<b>2.2 Brief introduction to the solution approach</b>	<b>3</b>
<b>3. Description of the approach</b>	<b>4</b>
<b>3.1 Data processing from the input file</b>	<b>4</b>
<b>3.2 Kruskal's Algorithm</b>	<b>7</b>
<b>3.2.1 Algorithmic steps</b>	
<b>3.2.2 Implementation details</b>	
<b>3.3 Prim's Algorithm</b>	<b>9</b>
<b>3.3.1 Algorithmic steps</b>	
<b>3.3.2 Implementation steps</b>	
<b>4. Analysis of implementation</b>	<b>12</b>
<b>4.1 Data Processing analysis</b>	<b>12</b>
<b>4.2 Kruskal's analysis</b>	<b>12</b>
<b>4.3 Prim's analysis</b>	<b>12</b>
<b>5. Conclusion</b>	<b>13</b>
<b>5.1 Comparison of Prim's and Kruskal's algorithm</b>	<b>13</b>
<b>7. References</b>	<b>14</b>

# 1. Introduction

A railway network is given in the form of a graph in which every vertex represents a station & there are some links between some pairs of stations which represent the cost of constructing a railway track between the pairs. See Figure 1 for reference. The railway network has to be constructed in such a way that the cost of construction is minimum & all the stations are connected to each other.



[Figure-1]

## 2. Description of the problem

### 2.1 Problem

1. The most cost-efficient way of constructing the railway tracks has to be found from the input given in Figure-1.
2. The railway network has to be constructed in such a way that all the stations are connected to each other i.e. if we select a station then there is always a way to go to any other station in the network.
3. File handling part for the input of the network that would be provided to us in the form of a text file is to be handled.

### 2.2 Brief introduction to the solution approach

1. Take the input graph from the text file by using the FileInputStream class of Java.
2. Store the graph in an appropriate data structure.
3. For the input graph, find one of its minimum spanning trees i.e. a tree constructed out of the graph for which the sum of the weights of the edges is minimum.

4. Use Prim's [1] as well as Kruskal's [2] algorithm implementation in order to find the minimum spanning tree of the input graph.
5. Output the final minimum spanning tree formed in the Answer.txt file using the PrintStream class of Java.

## 3. Description of the approach

### 3.1 Data processing from the input file

For taking input from the input file:

For example, let the input file be named Nodes3.txt with the contents as follows:

Alpha		
Beta		
Gamma		
Delta		
Epsilon		
Alpha	Epsilon	3
Alpha	Beta	2
Beta	Gamma	3
Gamma	Delta	5
Delta	Epsilon	3

The following snippet contains code for the file handling part:

```
public static void main(String[] args) throws IOException{
    // Final list of edges which are present in the mst to be stored in file:
    Answer.txt
    PrintStream ps = new PrintStream(new File("Answer.txt"));
    // taking input from the file provided as a command line argument
    if(args[0]==null){
        System.out.println("Please provide the input file as a command line
argument");
        return;
    }
    InputStream is = new FileInputStream(args[0]);
    System.setIn(is);
    System.setOut(ps);
}
```

Here it can be seen that the `FileInputStream` class for taking the input from the file and the input file name is passed from the console by using the command line arguments and can be found in `args[0]`, 'is' object contains the file input stream & passed in the `System.setIn(is)` as the argument so that it can take the input from the file in the same way the input is taken from the command line. For the output file, the 'ps' object of the `PrintStream` class has been initialised & passed in the object in the `System.setOut(ps)` in order to print in the file by using standard `System.out.println()` function.

For the file `Nodes3.txt` the code is run with the command:

➤ `java Kruskal Nodes3.txt`

For this file, the starting lines consist of the names of the vertices until an empty line is followed.

```
List<String> nodes = new ArrayList<String>();
while(!(line = sc.nextLine()).isEmpty()) {
    nodes.add(line);
}
int n = nodes.size(); // number of vertices
```

The names of the vertices is being stored in a list of strings called `nodes` and the input is taken by using the object of the scanner class "sc" by using the function `sc.nextLine()` until the next line is empty as per the condition of the while loop.

In the end, all the vertices names are stored in the `nodes` list and the variable 'n' stores the number of vertices.

As the vertices names are in the form of strings, for the convenience of the implementation. Each string name of the vertex is represented in the form of an integer like this:

Alpha-0
Beta-1
Gamma-2
Delta-3
Epsilon-4

For this, a `HashMap` that maps the string name to the integer is used for the convenience of the implementation.

```
Map<String,Integer> verticesToNumber = new HashMap<String,Integer>();
for(int i=0;i<n;i++){
```

```
verticesToNumber.put(nodes.get(i),i);
}
```

In the next lines of the file Nodes3.txt, the connections of the undirected edges are given in the format:

➤ x      y      weight

```
List<Edge> edges = new ArrayList<Edge>();
while(sc.hasNext()){
    String src = sc.next();
    String dest = sc.next();
    int weight = Integer.parseInt(sc.next());
    Edge edge = new
Edge(verticesToNumber.get(src),verticesToNumber.get(dest),weight);
    edges.add(edge);
}
int e = edges.size(); // number of edges
```

A list of Edges is used, edges class scheme is as follows:

```
static class Edge implements Comparable<Edge>{
    int src = -1;
    int dest = -1;
    int weight = -1;
    public Edge(int src,int dest,int weight){
        this.src = src;
        this.dest = dest;
        this.weight = weight;
    }
    public int compareTo(Edge other){
        return this.weight - other.weight;
    }
}
```

The Edge class is made comparable as this list has to be sorted in the algorithm.

The input of edges is taken one by one and is stored in a list named 'edges'. The variable 'e' stores the number of edges.

Note: This input implementation remains the same for both the algorithms i.e. the Prim's as well as the Kruskal's algorithm.

For the output, the following snippet is used:

```

List<Edge> mst = new Prim().findMst(graph,n,e);
    // Printing the edges in the mst as required in the output file
    for(String node:nodes){
        System.out.println(node);
    }
    System.out.println();
    for(Edge edge:mst){
        System.out.println(nodes.get(edge.src) + "\t" + nodes.get(edge.dest)
+ "\t" + edge.weight);
    }

```

Here, assume that in the List 'mst' has the final edges. For a detailed, Edge class, see the further section.

## 3.2 Kruskal's Algorithm

### 3.2.1 Algorithmic steps

The algorithm[2] works in the following steps:

1. The edges of the graph are sorted in the ascending order of their weights.
2. Assume that all the vertices of the graph are independent at the starting.
3. Traverse the sorted edges and select the edge and include it in the minimum spanning tree only if the edge included doesn't create a cycle in the graph.
4. Check if the graph contains a cycle by using the disjoint set data structure & its union function after including an edge in the Minimum spanning tree.

### 3.2.2 Implementation details

For sorting the edges, the edge class is made comparable and used the Collections.sort(edges) method to sort it. This sorting function is better than the bubble/insertion sort as it would take  $O(n \log n)$  time in comparison to the  $O(n^2)$  time taken by the later algorithms.

```

private static List<Edge> findMst(List<Edge> edges,int n,int e){
    // Sorting the edges in the increasing order of their weights
    Collections.sort(edges);
    // mst is the list of edges which are present in the mst
    List<Edge> mst = new ArrayList<Edge>();
    // selected variable is to keep track of the number of vertices which are already
included in the mst
    int selected = 0;
    // max selected can be n-1
    // Initially all the vertices belong to one set
    // Initially all the vertices are not selected
    // Disjoint set union is used to find if there is a cycle in graph or not
    List<Subset> subsets = new ArrayList<Subset>();
    for(int i=0;i<n;i++){

```

```

        // Initial parent of each vertex is itself & rank is 0
        subsets.add(new Subset(i,0));
    }
    // Iterating over the edges
    for(int i=0;i<e;i++){
        // If n-1 edges are already included in the mst, then we can break the loop
        if(selected == n-1) break;
        // Getting the edge from the list of edges
        int src = edges.get(i).src;
        int dest = edges.get(i).dest;
        // Finding the subset of the source vertex
        int parSrc = findPar(src, subsets);
        // Finding the subset of the destination vertex
        int parDest = findPar(dest, subsets);
        // If the source and destination are in different sets, then we can add the
        // edge to the mst as there would be no cycle after adding it to the mst
        if(parSrc != parDest){
            // Adding the edge to the mst
            mst.add(edges.get(i));
            // Merging the subsets of the source and destination vertices
            Union(subsets,parSrc,parDest);
            // Incrementing the number of vertices which are already included in the
            mst
            selected++;
        }
    }
    // Returning the list of edges which are present in the mst
    return mst;
}

```

The final minimum spanning tree edges are stored in the 'mst' list.

Working on the Disjoint-set union data structure:

1. The disjoint-set union data structure is used with the union on the basis of rank [3]. This is better than the normal union operation of this data structure as the normal union would have made a linear chain & each cycle checking operation would have become  $O(n)$ . But with the union operation by rank, the complexity is on average  $O(\log(n))$ .
2. In a disjoint set union, initially, each vertex is independent. If we try to include an edge in the minimum spanning tree, we would check whether the two vertexes in the edge selected belong to the same set or not.
3. In order to do so, each vertex is associated with a parent, of its set and if the parents of both the selected vertexes are the same, it means they belong to the same set and cannot be included in the final minimum spanning tree.



```

private static void Union(List<Subset> subsets, int x, int y)
{
    int xroot = findPar(x,subsets);
    int yroot = findPar(y,subsets);
    // If the rank of the first subset is greater than the rank of the second
subset, then we can merge the second subset into the first subset
    if(subsets.get(xroot).rank < subsets.get(yroot).rank){
        subsets.get(xroot).parent = yroot;
    }
    // If the rank of the first subset is less than the rank of the second subset,
then we can merge the first subset into the second subset
    else if(subsets.get(xroot).rank > subsets.get(yroot).rank){
        subsets.get(yroot).parent = xroot;
    }
    // else if the rank of the first subset is equal to the rank of the second
subset, then we can merge the first subset into the second subset
    else{
        subsets.get(yroot).parent = xroot;
        subsets.get(xroot).rank++;
    }
}

private static int findPar(int v,List<Subset> subset){
    if(subset.get(v).parent==v) return v;
    else return findPar(subset.get(v).parent,subset);
}

```

### 3.3 Prim's Algorithm

#### 3.3.1 Algorithmic steps

The algorithm works in the following steps:

1. First, initialize an MST with the randomly chosen vertex. Now, we have to find all the edges that connect the tree in the above step with the new vertices.
2. From the edges found, select the minimum edge and add it to the tree.
3. Repeat step 2 until the minimum spanning tree is formed.

#### 3.3.2 Implementation details

1. The Adjacency list data structure is used to store the graph in addition to the input.

The graph scheme is as follows:

```

/*
    Graph class which contains the information about the graph.
    Scheme:
    n: number of vertices in the graph

```

```

adjList: adjacency list of the graph
function insertEdge: inserts an edge into the graph(undirected)
*/

```

```

static class Graph{
    int n;
    LinkedList<Edge> adjList[];
    public Graph(int n){
        this.n = n;
        adjList = new LinkedList[n];
        for(int i=0;i<n;i++){
            adjList[i] = new LinkedList<Edge>();
        }
    }
    public void insertEdge(int src,int dest,int weight){
        Edge edge = new Edge(src,dest,weight);
        adjList[src].add(edge);
        edge = new Edge(dest,src,weight);
        adjList[dest].add(edge);
    }
}

```

2. In this algorithm, the source vertex (in this case we are using the vertex '0' as the source vertex) is taken.
3. Three array data structures are used viz. parent, key, visited. Parent[n] array store the parent of the ith vertex in the mst constructed. Key[n] stores the current key value of the ith vertex. Visited[n] stores whether the ith node is included in the mst or not. Visited array is a Boolean array.
4. The minimum edge is found with the use of a max-heap data structure. This data structure instead of iterating over all the edges connected to the current mst as it would become an  $O(e)$  operation in the case of iteration. While in the case of a max-heap, this can be done in  $O(\log(e))$  time.

In java, the implementation is as follows:

```

/*
    Class Item which contains the information about the item.
    Item is used to store the vertex and it's key value that are to be used as an
    object for the priority queue
    later used in the Prim's Algorithms for mst.
    Scheme:
    vertex: vertex of the graph
    key: key value of the vertex
    */
class Item{

```

```

        int vertex = -1;
        int key = -1;
        public Item(int vertex,int key){
            this.vertex = vertex;
            this.key = key;
        }
    }

    PriorityQueue<Item> pq = new PriorityQueue<Item>(n,new Comparator<Item>(){
        public int compare(Item p1,Item p2){
            return p1.key - p2.key;
        }
    });
    // Adding the source vertex '0' to the priority queue
    pq.add(new Item(0,0));

```

Then the priority queue iteration is done as follows:

```

while(!pq.isEmpty()){
    // Getting the vertex with the minimum key value
    Item item = pq.poll();
    int u = item.vertex;
    // If the vertex is already present in the mst, then ignore it
    if(visited[u]==true){
        continue;
    }
    // Marking the vertex as present in the mst
    visited[u] = true;
    // Iterating over the adjacency list of the vertex
    for(Edge edge:graph.adjList[u]){
        int v = edge.dest;
        // If the vertex is already present in the mst, then ignore it else
        // update the key value if the edge weight is less than the current key value
        if(!visited[v] && edge.weight < key[v]){
            key[v] = edge.weight;
            parent[v] = u;
            pq.add(new Item(v,key[v]));
        }
    }
}

```

5. The edges included in the mst can be found from the parent array as follows:

```

// Creating the list of edges which are present in the mst
List<Edge> mst = new ArrayList<Edge>();
for(int i=1;i<n;i++){
    mst.add(new Edge(parent[i],i,key[i]));
}

```

```
}
```

6. Then, the list 'mst' of edges can be returned.

## 4. Analysis of implementation

### 4.1 Data Processing analysis

Assumptions:  $n$ =number of vertices in the graph,  $e$ =number of edges in the graph.

#### Input analysis:

As  $n$  vertices and  $e$  edges are read, so the time complexity for this operation is  $O(n+e)$ .

### 4.2 Kruskal's analysis

1. The edges list is being sorted according to the increasing order of their weights. As there are  $e$  edges, so this operation takes  $O(e \log(e))$  time as the inbuilt Collections sort function in Java uses merge/quicksort implementation which runs in  $O(e \log e)$  time.
2. Iteration is done through all the edges in sorted order and it is found whether the edge, if included forms a cycle or not in the current minimum spanning tree which is an  $O(\log(n))$  operation in the worst case due to the rank-based Union algorithm. We do this  $e$  times as we are iterating over the edges.

Proof [3]: In the case of the Union operation in union without using rank, it could happen that in the worst case all the vertices are linked together in a chain of length ' $n$ '. In that case, the query for finding the parent would be  $O(n)$  operation. But, when the union by size/rank is used, the forming of a single long chain is ruled out & in the worst case the operation for finding the parent becomes  $O(\log(n))$ .

**So, the overall time complexity of the algorithm is  $O(e \log e + e \log n)$ .**

If, in place of the rank-based union, if the normal union operation would have been used, then each union operation would have cost  $O(n)$  time in the worst case as explained in the proof. This would have made the algorithm  $O(e \log(e) + e.n)$  time  $\sim O(ne)$  time which is  $\gg O(e \log e + e \log n)$  that we used. That's why the rank-based union function is preferred.

### 4.3 Prim's analysis

1. The creation of the adjacency list is an  $O(e)$  operation as the list of edges of size ' $e$ ' is traversed.
2. The initialisation of the three arrays viz. parent, key, visited is also an  $O(n)$  operation as each array of size ' $n$ '

3. As the priority queue implementation of Java collections is used which uses a binary heap data structure implementation, the following are the time complexities for the various operation on it.

In binary heap, the insertion operation is  $O(\log(n))$  in worst case and the removal operation of the smallest element is also  $O(\log(n))$  as the root element of the heap is removed & then the max-heap is rebalanced for making the next smallest element as the root.

As 'n' number of vertices are being inserted in the priority queue, so the time complexity of the priority queue loop is  $O(n\log n)$ .

The other method for finding the minimum key value would have to iterate over all the key values in the array & find the smallest one out of them for a vertex that has not been included in the minimum spanning tree yet, that would have made the time complexity as  $O(n.e) \gg O(n\log n)$ , that is why the priority queue data structure is preferred.

**So, the overall time complexity =  $O(e+n\log n)$**

## 5. Conclusion

In order to solve the problem mention in section '1', the two well known algorithms viz. Prim's & Kruskal's were used in their most efficient implementation. The following time complexities were achieved:

- i) Kruskal algorithm:  $O(e\log e + e\log n)$ .
- ii) Prim algorithm:  $O(e+n\log(n))$

### 5.1 Comparison of Prim's and Kruskal's algorithm

Prim's algorithm time complexity is better in comparison to Kruskal's algorithm in the case when the graph is dense i.e. it has a large number of edges as can be seen in the final time complexities of both the algorithms.

Prim's algorithm would not offer much control when the edges having the same weights are repeated as those edges are stored in the queue rather than all the edges as in the case of Kruskal's algorithm.

## 6. References

- [1] [https://cp-algorithms.com/graph/mst\\_prim.html](https://cp-algorithms.com/graph/mst_prim.html)
- [2] [https://cp-algorithms.com/graph/mst\\_kruskal.html](https://cp-algorithms.com/graph/mst_kruskal.html)
- [3] [https://cp-algorithms.com/graph/mst\\_kruskal\\_with\\_dsu.html](https://cp-algorithms.com/graph/mst_kruskal_with_dsu.html)