

Greedy Algorithms

CPCS324 Project - Phase two

Instructor: Dr. Bassma Saleh Alsulami
Group #5

<i>Student Name</i>	<i>Section</i>
<i>Dimah Abdullah Alolayan</i>	<i>DAR</i>
<i>Majd Saeed Gezan</i>	<i>DAR</i>
<i>Razan Muhammed Aljuhani</i>	<i>DAR</i>

Table of Contents

1. Introduction	3
2. Empirical Analysis of the Algorithm	3
2.1 Empirical Purpose.....	3
2.2 The Method to Measure the Efficiency	4
2.3 Input Characteristics	4
2.4 Algorithms.....	4
2.4.1. The Prim's using priority queue Algorithm	5
2.4.2. The Prim's using min-heap Algorithm	5
2.4.3. The Kruskal using priority queue Algorithm	6
2.5 Data Observed from Running the Algorithms	6
2.6 Analyze the Data Obtained	8
3. Dijkstra Algorithm	8
4. Difficulties.....	9
5. Conclusion.....	9
6. Appendix	9
7. References	10

Figures

Figure 1: Prim's using priority queue Algorithm.....	5
Figure 2: Prim's using min-heap Algorithm.....	5
Figure 3: Kruskal using priority queue Algorithm	6
Figure 4: Comparison between Prim algorithm's running time for both implementation (Priority queue and Min heap)	7
Figure 5: Comparison between Prim priority queue and Kruskal algorithm running time.....	7
Figure 6: Dijkstra Algorithm	8
Figure 7: Output of the running time of the algorithms	9
Figure 8: Kruskal output 2.....	10
Figure 9: Kruskal Output 2	10
Figure 10: Dijkstra final result.....	10

Tables

Table 1: Prim Priority queue Runtime.....	6
Table 2: Prim Min heap Runtime	6
Table 3: Kruskal Runtime	6

1. Introduction

There are different algorithms techniques to solve problems that are discovered by computer scientists, in this report we are discussing one of the commonly used techniques which is Greedy Algorithm, this type of algorithm makes the optimal choice at each step with the hope of finding the overall optimal way for the whole problem. This report focuses on three of the Greedy Algorithm techniques, which are Prim's, Kruskal's and Dijkstra's algorithms.

The Prim's and Kruskal's algorithms work in a different manner to find minimum spanning tree of a directed/undirected weighted graph. Therefore, prim's algorithm will be implemented using two data structures Priority queue and min-heap, then Kruskal's algorithm will be implemented and compared with Prim's in term of efficiency.

The Dijkstra algorithm solves single-source-shortest-paths problem by finding the shortest paths between a source node and every other node through a directed/undirected graph with nonnegative weight only.

As an advantage of greedy algorithms, analyzing the run time will generally be much easier than for other techniques. With that being said, in this report we will find performance of Greedy algorithms Prim's & Kruskal's by empirical analysis. Also, we will be going to implement Dijkstra's algorithm then discuss the performance of it.

2. Empirical Analysis of the Algorithm

2.1 Empirical Purpose

The main purpose of using the empirical analysis in this experiment is to compare the performance of Prim's (priority queue) & Kruskal's algorithms and compare Prim's (priority queue) and Prim's (min-heap), prim's algorithm is implemented using priority queue and min-heap. Then, Kruskal's algorithm is implemented using priority queue. various graphs will be used that are generated randomly as an input of code. Runtime will be measured for each algorithm. this experiment results a conclusion on which algorithms are better.

2.2 The Method to Measure the Efficiency

There are two approaches to measure the efficiency of any algorithm, The first way is a straightforward way, it works by counting how many times the basic operation is executed by implementing a counter to the code, the second way is to use physical time measure which is the actual time that the program takes to execute the algorithm.

In this experiment the time measuring approach has been chosen because we need to have a specific information about the algorithm's performance in a real computing environment. the code was implemented using java programming language, `currentTimeMillis()` function is used to determine the physical time.

2.3 Input Characteristics

The data structure used for all algorithms to represent the graph is the adjacency list, the adjacency list is filled with a connected graph that is generated randomly according to user choice from the 10 cases of `Make_Graph()` function. The purpose of taking a big range of sizes is to test the performance more accurately on a wide variety of inputs.

Input specification

- edge weights are integers and between $1 \leq \text{weights} \leq 10$
- There are 10 cases to randomly generate Graphs. where n represent the number of nodes, and m represent the number of edges.
 - $n=1000$ and $m=\{10000, 15000, 25000\}$,
 - $n=5000$ and $m=\{15000, 25000\}$,
 - $n=10000$ and $m=\{15000, 25000\}$,
 - $n=20000$ and $m=\{200000, 300000\}$,
 - $n=50000$ and $m=1,000,000$.
- No edge will appear more than once in the same test case.

2.4 Algorithms

In this section, Prim's algorithm is shown how it is implemented by different data structures which are priority queue and min-heap and how Kruskal's algorithms works implemented by priority queue.

2.4.1. The Prim's using priority queue Algorithm

ALGORITHM *Prim(G)*

```
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G = \langle V, E \rangle$ 
//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$ 
 $V_T \leftarrow \{v_0\}$  //the set of tree vertices can be initialized with any vertex
 $E_T \leftarrow \emptyset$ 
for  $i \leftarrow 1$  to  $|V| - 1$  do
    find a minimum-weight edge  $e^* = (v^*, u^*)$  among all the edges  $(v, u)$ 
    such that  $v$  is in  $V_T$  and  $u$  is in  $V - V_T$ 
     $V_T \leftarrow V_T \cup \{u^*\}$ 
     $E_T \leftarrow E_T \cup \{e^*\}$ 
return  $E_T$ 
```

Figure 1: Prim's using priority queue Algorithm

From: Textbook - Introduction to The Design and Analysis of Algorithms Chapter9, Section 9.1, Page 319

2.4.2. The Prim's using min-heap Algorithm

```
Algorithm Prim2
Input Weighted connected graph  $G = \langle V, E \rangle$ 
Output  $E_T$ , a minimum spanning tree of  $G$ 

// initialize a priority queue — 1: Initialize( $Q$ )
2: foreach  $u \in V$  adjacent to  $v_0$  do
3:   Insert( $Q, u, w_u$ )
// initialize the tree — 4:  $V_T \leftarrow \{v_0\}$ 
5:  $E_T \leftarrow \phi$ 
6: for  $i \leftarrow 1$  to  $|V| - 1$  do
    // FIND A MINIMUM-WEIGHT EDGE  $e^* = (v^*, u^*)$  AMONG ALL  $(v, u)$ 
    // SUCH THAT  $v$  IS IN  $V_T$  AND  $u$  IS IN  $V - V_T$ 
// get next tree vertex — 7:  $u^* \leftarrow \text{DeleteMin}(Q)$ 
// update tree — 8:  $V_T \leftarrow V_T \cup \{u^*\}$ 
9:  $E_T \leftarrow E_T \cup \{e^*\}$ 
// update priority queue — 10: foreach  $u \in V - V_T$  adjacent to  $u^*$  do
11:   Insert( $Q, u, w_u$ )
12: return  $E_T$ 
```

Figure 2: Prim's using min-heap Algorithm

From: slides of Comparison – Link: https://lms.kau.edu.sa/bbcswebdav/pid-8384982-dt-content-rid-61015261_1/courses/202102_CPCS324_DAR_14449_EL/comparison.pdf

2.4.3. The Kruskal using priority queue Algorithm

ALGORITHM *Kruskal*(G)

```

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G = \langle V, E \rangle$ 
//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$ 
sort  $E$  in nondecreasing order of the edge weights  $w(e_{i_1}) \leq \dots \leq w(e_{i_{|E|}})$ 
 $E_T \leftarrow \emptyset$ ;  $ecounter \leftarrow 0$     //initialize the set of tree edges and its size
 $k \leftarrow 0$                         //initialize the number of processed edges
while  $ecounter < |V| - 1$  do
     $k \leftarrow k + 1$ 
    if  $E_T \cup \{e_{i_k}\}$  is acyclic
         $E_T \leftarrow E_T \cup \{e_{i_k}\}$ ;  $ecounter \leftarrow ecounter + 1$ 
return  $E_T$ 

```

Figure 3: Kruskal using priority queue Algorithm

From: Textbook - Introduction to The Design and Analysis of Algorithms Chapter9, Section 9.2, Page 329

2.5 Data Observed from Running the Algorithms

The tools used for analyzing the algorithms and comparing between them:

- **JAVA programming language:** used to implement the code.
- **NetBeans IDE 8.2:** used to run the code and show the output.
- **Microsoft Excel:** used to represent result data.

The illustrated figures and tabels below contain records of 10 different cases, followed by graphs representing these data.

Prim Priority-queue		
n	m	Runtime pq
1000	10000	0.058
	15000	0.06
	25000	0.064
5000	15000	0.059
	25000	0.069
10000	15000	0.068
	25000	0.069
20000	200000	0.193
	300000	0.263
50000	1000000	0.552

Table 2: Prim Priority queue Runtime

Prim Min Heap		
n	m	Runtime MinHeap
1000	10000	0.006
	15000	0.006
	25000	0.011
5000	15000	0.012
	25000	0.013
10000	15000	0.015
	25000	0.018
20000	200000	0.06
	300000	0.087
50000	1000000	0.193

Table 3: Prim Min heap Runtime

Kruskal		
n	m	Runtime Kruskal
1000	10000	0.023
	15000	0.026
	25000	0.031
5000	15000	0.09
	25000	0.115
10000	15000	0.232
	25000	0.331
20000	200000	2.087
	300000	2.649
50000	1000000	19.237

Table 1: Kruskal Runtime

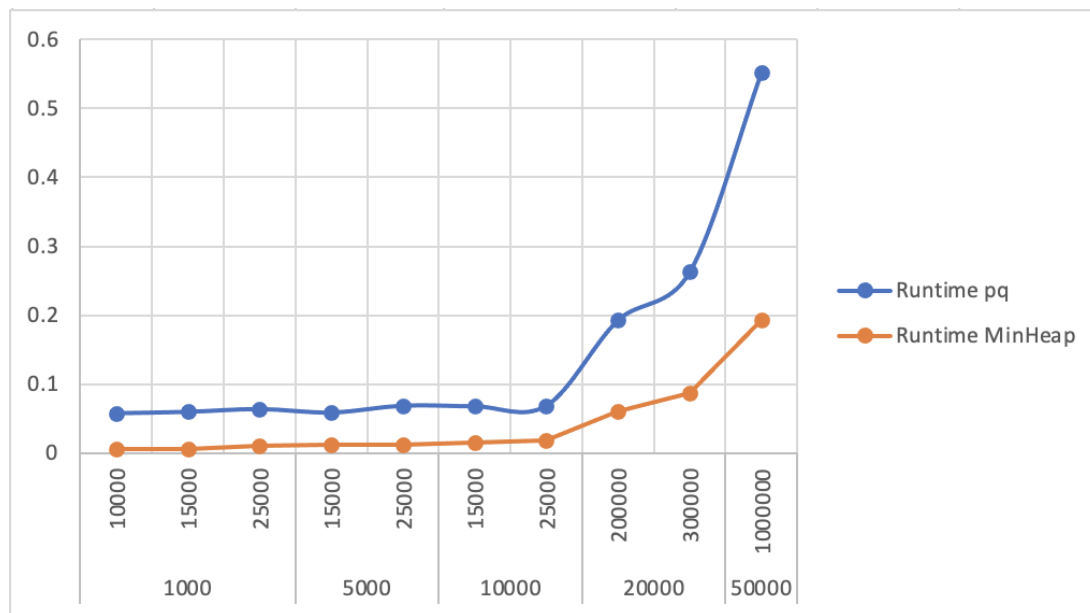


Figure 4: Comparison between Prim algorithm's running time for both implementation (Priority queue and Min heap)

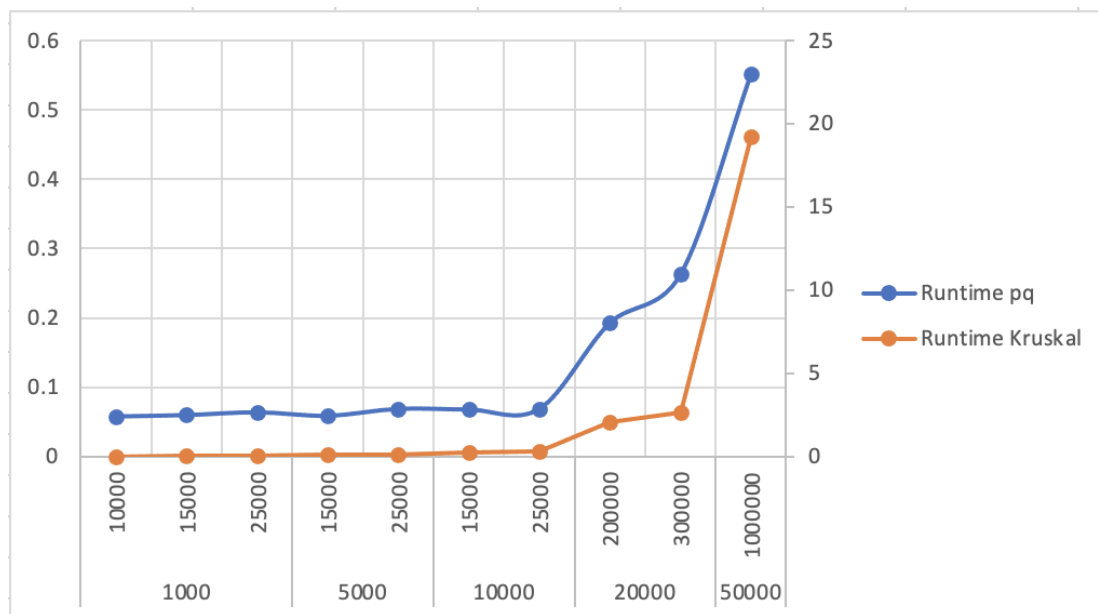


Figure 5: Comparison between Prim priority queue and Kruskal algorithm running time

2.6 Analyze the Data Obtained

As illustrated in the figures above in the previous section and from data observation we can say that Prim's using min-heap algorithm runs faster than the other algorithms. However, using large or small vertices and number of edges affects the running time for every algorithm. In the empirical analysis to find the efficiency class we used the scatter plot graph type to represent the comparison between algorithms. clearly, we can see the result of the comparison in the graph and the tables that Prim's using min-heap algorithm is more slightly more efficient than Prim's using priority queue algorithm and undoubtedly more efficient than kurskul's algorithm.

the efficiency class of each one of the algorithms is represented as:

- Prim's using min-heap: $O(\log n)$
- Prim's using priority queue: $O(|E| \log |V|)$
- Kurskul's: $O(|E| \log |E|)$

3. Dijkstra Algorithm

Dijkstra's Algorithm works for finding the shortest paths from the single source vertex to all other nodes in the graph. In this report Dijkstra's Algorithm graph was implemented using adjacency list and priority queue, the time Complexity is in $O(|E| \log |V|)$ and the space complexity is $O(V)$ where, E is the number of edges and V is the number of vertices.

ALGORITHM *Dijkstra(G, s)*

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph  $G = (V, E)$  with nonnegative weights
//      and its vertex  $s$  ( $s \in V$ )
//Output: The length  $d_v$  of a shortest path from  $s$  to  $v$   $\forall v \in V$ :  $d_v$  the length of the shortest path from  $s$  to  $v$ 
//      and its penultimate vertex  $p_v$  for every vertex  $v$  in  $V$   $p_v$  (next to last vertex in path)
Initialize(Q) //initialize priority queue to empty
for every vertex  $v$  in  $V$ 
     $d_v \leftarrow \infty$ ;  $p_v \leftarrow \text{null}$ 
    Insert(Q,  $v, d_v$ ) //initialize vertex priority in the priority queue
 $d_s \leftarrow 0$ ; Decrease(Q,  $s, d_s$ ) //update priority of  $s$  with  $d_s$ 
 $V_T \leftarrow \emptyset$ 
for  $i \leftarrow 0$  to  $|V| - 1$  do
     $u^* \leftarrow \text{DeleteMin}(Q)$  //delete the minimum priority element // pick the fringe vertex with min distance
     $V_T \leftarrow V_T \cup \{u^*\}$ 
    for every vertex  $u$  in  $V - V_T$  that is adjacent to  $u^*$  do //Update fringe set after adding  $u^*$ 
        if  $d_{u^*} + w(u^*, u) < d_u$ 
             $d_u \leftarrow d_{u^*} + w(u^*, u)$ ;  $p_u \leftarrow u^*$ 
            Decrease(Q,  $u, d_u$ )
```

Figure 6: Dijkstra Algorithm

From: Textbook - Introduction to The Design and Analysis of Algorithms Chapter9, Section 9.3, Page 363

4. Difficulties

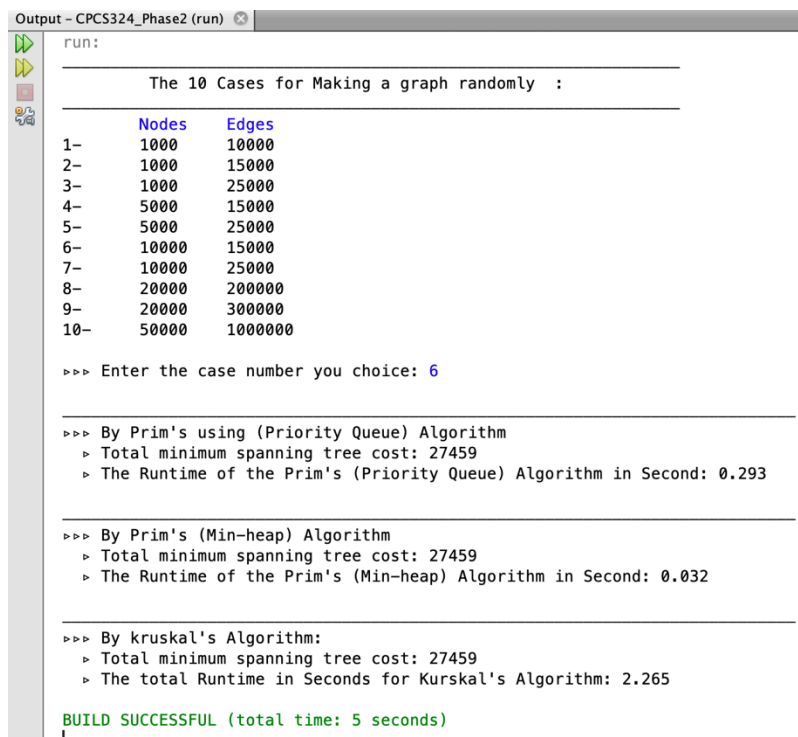
The difficulties that faced us during this project phase was tracing a huge graph, we overcame it by testing the algorithm using static small graphs. Another difficulty we faced was to screenshot the whole output it was too long and not clear. instead, we took a screenshot of the start of each graph's output.

5. Conclusion

To conclude, this experiment's purpose was to use the empirical analysis to compare between different greedy algorithm techniques, However, we had a brief discussion about the empirical analysis and Dijkstra's Algorithm implementation process. Also, this report went through many sections to discuss the observation steps. After all the observation and running-time analysis we noticed that Prim's using min-heap algorithm is more efficient that Prim's using priority queue algorithm and kruskul's algorithm.

6. Appendix

In Prim's Case it took way too long to print the tracing we waited more than an hour and the running was not finished yet, so we kept the printing statement as a comment in case it was needed to be tested.



```
Output - CPCS324_Phase2 (run)
run:
The 10 Cases for Making a graph randomly :
Nodes Edges
1- 1000 10000
2- 1000 15000
3- 1000 25000
4- 5000 15000
5- 5000 25000
6- 10000 15000
7- 10000 25000
8- 20000 200000
9- 20000 300000
10- 50000 1000000

>>> Enter the case number you choice: 6

>>> By Prim's using (Priority Queue) Algorithm
> Total minimum spanning tree cost: 27459
> The Runtime of the Prim's (Priority Queue) Algorithm in Second: 0.293

>>> By Prim's (Min-heap) Algorithm
> Total minimum spanning tree cost: 27459
> The Runtime of the Prim's (Min-heap) Algorithm in Second: 0.032

>>> By kruskal's Algorithm:
> Total minimum spanning tree cost: 27459
> The total Runtime in Seconds for Kurskal's Algorithm: 2.265

BUILD SUCCESSFUL (total time: 5 seconds)
```

Figure 7: Output of the running time of the algorithms

```
>>> By kruskal's Algorithm:
Edge: 0-922 cost: 1
Edge: 1-356 cost: 1
Edge: 2-599 cost: 1
Edge: 3-159 cost: 1
Edge: 4-461 cost: 1
Edge: 5-25 cost: 1
Edge: 6-23 cost: 1
Edge: 7-788 cost: 1
Edge: 8-179 cost: 1
Edge: 9-983 cost: 1
Edge: 10-63 cost: 1
Edge: 11-268 cost: 1
Edge: 12-429 cost: 1
Edge: 13-775 cost: 1
Edge: 14-965 cost: 1
Edge: 15-181 cost: 1
Edge: 16-73 cost: 1
Edge: 17-777 cost: 1
Edge: 18-166 cost: 1
Edge: 19-29 cost: 1
Edge: 20-309 cost: 1
Edge: 21-487 cost: 1
Edge: 22-919 cost: 1
Edge: 23-264 cost: 1
Edge: 24-188 cost: 1
Edge: 25-687 cost: 1
Edge: 26-183 cost: 1
Edge: 27-244 cost: 1
Edge: 28-914 cost: 1
Edge: 29-345 cost: 1
Edge: 30-216 cost: 1
Edge: 31-295 cost: 1
Edge: 32-38 cost: 1
```

Figure 8: Kruskal output 2

```
Output - CPCS324_Phase2 (run)
Edge: 972-988 cost: 2
Edge: 973-300 cost: 2
Edge: 974-363 cost: 2
Edge: 975-506 cost: 2
Edge: 976-679 cost: 2
Edge: 977-307 cost: 2
Edge: 978-273 cost: 2
Edge: 979-726 cost: 2
Edge: 980-533 cost: 2
Edge: 981-713 cost: 2
Edge: 982-611 cost: 2
Edge: 983-79 cost: 2
Edge: 984-579 cost: 2
Edge: 985-309 cost: 2
Edge: 986-669 cost: 2
Edge: 987-358 cost: 2
Edge: 988-350 cost: 2
Edge: 989-926 cost: 3
Edge: 990-600 cost: 3
Edge: 991-117 cost: 3
Edge: 992-551 cost: 3
Edge: 993-574 cost: 3
Edge: 994-285 cost: 3
Edge: 995-460 cost: 3
Edge: 996-65 cost: 3
Edge: 997-294 cost: 3
Edge: 998-373 cost: 3
  > Total minimum spanning tree cost: 1133
  > The total Runtime in Seconds for Kruskal's Algorithm: 3.066
```

Figure 9: Kruskal Output 2

```
Output - CPCS324_Dijkstra (run)
  > Dammam(Jeddah,1343)
-----
Tree Vertices VT
  > Jeddah(-,0)
  > Makkah(Jeddah,79)
  > Taif(Jeddah,167)
  > Madinah(Jeddah,420)
  > Abha(Jeddah,625)
  > Jizan(Jeddah,710)
  > Hail(Jeddah,777)
  > Qasim(Jeddah,863)
  > Najran(Jeddah,905)
  > Riyadh(Jeddah,949)
  > Tabuk(Jeddah,1024)

Remaining Vertices V-VT
  > Dammam(Jeddah,1343)
-----

Tree Vertices VT
  > Jeddah(-,0)
  > Makkah(Jeddah,79)
  > Taif(Jeddah,167)
  > Madinah(Jeddah,420)
  > Abha(Jeddah,625)
  > Jizan(Jeddah,710)
  > Hail(Jeddah,777)
  > Qasim(Jeddah,863)
  > Najran(Jeddah,905)
  > Riyadh(Jeddah,949)
  > Tabuk(Jeddah,1024)
  > Dammam(Jeddah,1343)

Remaining Vertices V-VT
-----

BUILD SUCCESS
|
```

Figure 10: Dijkstra final result

7. References

1. Levitin, A. (2012,2007,2003). *Introduction to The Design & Analysis of Algorithms*. New Jersey: Addison-Wesley.
2. <https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/>
3. <https://algorithms.tutorialhorizon.com/prims-minimum-spanning-tree-mst-using-adjacency-list-and-priority-queue-without-decrease-key-in-oelgv/>
4. <https://algorithms.tutorialhorizon.com/prims-minimum-spanning-tree-mst-using-adjacency-list-and-min-heap/>
5. <https://algorithms.tutorialhorizon.com/kruskals-algorithm-minimum-spanning-tree-mst-complete-java-implementation/>
6. <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-in-java-using-priorityqueue/>