|  |
| --- |
| Ionasco Gheorghe |

|  |
| --- |
| Technical University of Moldova Faculty of Computers, Informatics and Microelectronics Software Engineering and Automation Department |
| Calculabilitate şi complexitate |
| Laboratory №6 |

# Introduction

What makes an algorithm greedy? Definitions in the literature vary slightly, but most describe a greedy algorithm makes a sequence of choices, each choice being in some way the best available at that time (the term greedy refers to choosing the best). When making the sequence of choices, a greedy algorithm never goes back on earlier decisions. Because of their simplicity, greedy algorithms are frequently straightforward and efficient. They are also very versatile, being useful for different problems in many varied areas of combinatorics and beyond. Some examples of applications include data compression and DNA sequencing, finding minimum-cost spanning trees of weighted undirected graphs, computational geometry, and routing through networks, to name but a few. Some problems are impractical to solve exactly but may have a greedy algorithm that can be used as a heuristic, and solutions that are close to optimal. For other problems, greedy algorithms may produce an exact solution. Unfortunately, for any particular problem, there is no guarantee that a greedy algorithm exists to solve it exactly.

Therefore, the algorithm designer who thinks up plausible greedy strategies to solve a problem may and theory about greedy algorithms useful: correctness conditions can be tested to see whether a particular greedy algorithm provides an exact solution. Existing greedy theories have frequently addressed the following concerns:

* Expression of greedy algorithms;
* Correctness proofs of greedy algorithms;
* Characterization of greedy data structures;
* Synthesis of greedy algorithms;
* Coverage of as many greedy algorithms as possible;

Deferent greedy theories have concentrated on different selections of the above concerns. The theory of matroids, and later greedoids, models greedy algorithms using set systems, concentrating heavily on the characterization of greedy data structures (problem structures for which the greedy algorithm produces an optimal solution), but does not consider the synthesis of greedy algorithms at all.

# 2. Algorithms

2.1. Prim’s algorithm.

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

The Prim’s algorithm:

**Step 1:** Select a starting vertex

**Step 2:** Repeat Steps 3 and 4 until there are fringe vertices

**Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight

**Step 4:** Add the selected edge and the vertex to the minimum spanning tree T

**Step 5:** EXIT

Example:

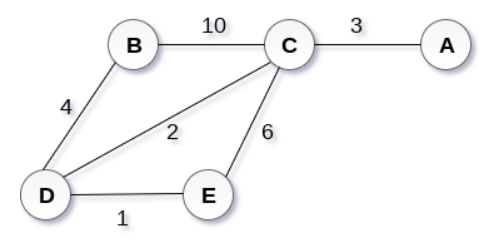


Figure 2.1.1 –Graph.

Solution:

**Step 1:** Choose a starting vertex B.



Figure 2.1.2 –Step 1.

**Step 2:** Add the vertices that are adjacent to A. the edges that connecting the vertices are shown by dotted lines.

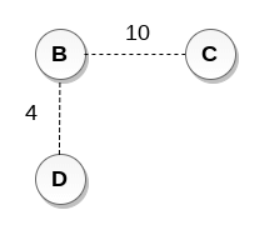


Figure 2.1.3 –Step 2.

**Step 3:** Choose the edge with the minimum weight among all. i.e., BD and add it to MST. Add the adjacent vertices of D i.e., C and E.

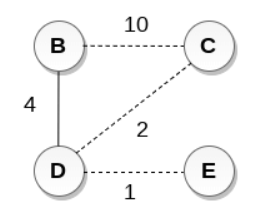


Figure 2.1.4 –Step 3.

**Step 4:** Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e., E and A.

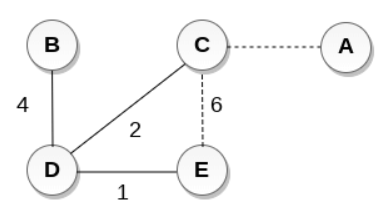


Figure 2.1.5 –Step 4.

**Step 5:** Choose the edge with the minimum weight i.e., CA. We can't choose CE as it would cause cycle in the graph.

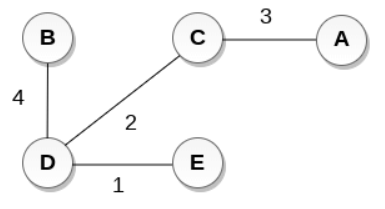


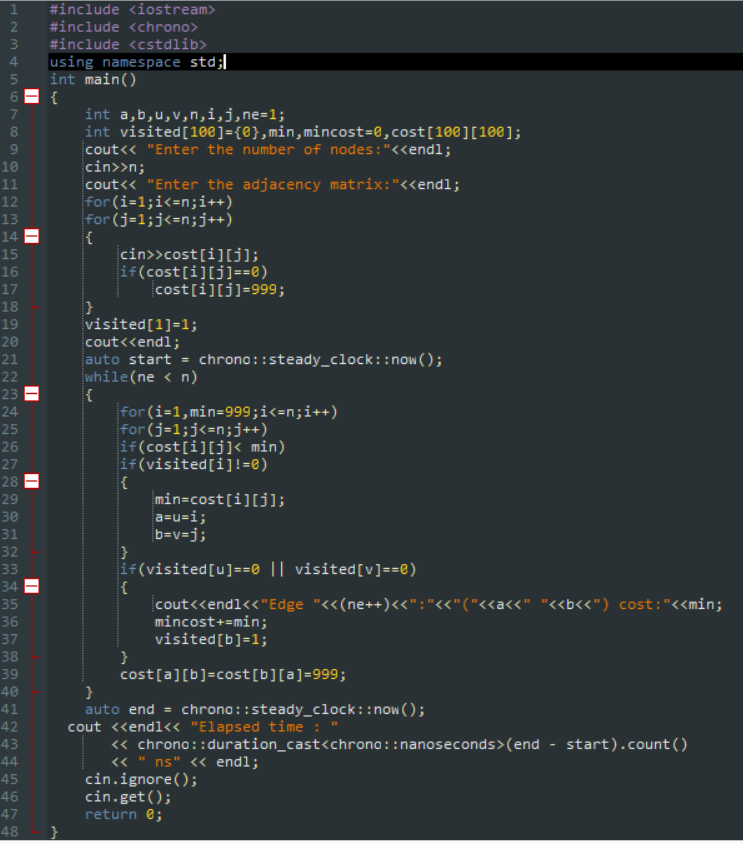
Figure 2.1.6 – Minimum spanning tree.

The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.

The cost of MST will be calculated as;

Minimum cost = 4 + 2 + 1 + 3 = 10 units.

Implementation:



2.2. Kruskal’s algorithm.

Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

The Kruskal's algorithm:

**Step 1:** Create a forest in such a way that each graph is a separate tree.

**Step 2:** Create a priority queue Q that contains all the edges of the graph.

**Step 3:** Repeat Steps 4 and 5 while Q is NOT EMPTY

**Step 4:** Remove an edge from Q

**Step 5:** IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).  
ELSE Discard the edge.

**Step 6:** END.

Example:

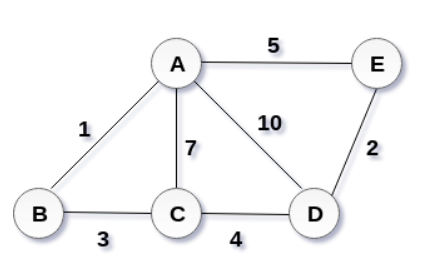
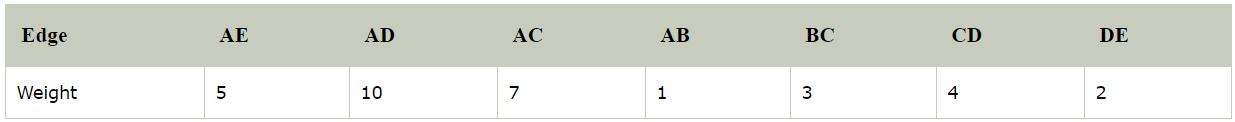


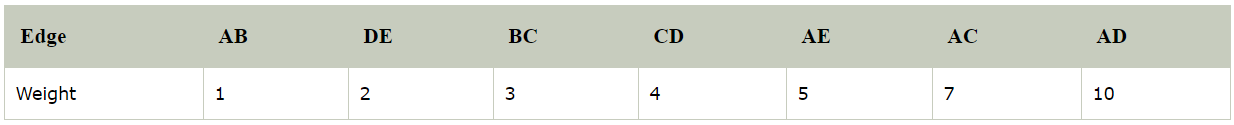
Figure 2.2.1 –Graph.

Solution:

The weight of the edges given as:



Sortation:



Start constructing the tree:

1. Add AB to the MST.

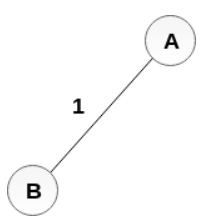


Figure 2.2.2 –Step 1.

1. Add DE to the MST.

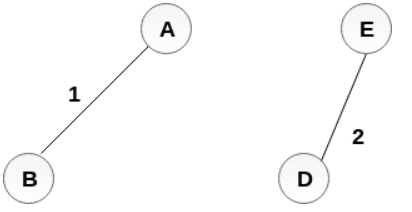


Figure 2.2.3 –Step 2.

1. Add BC to the MST;

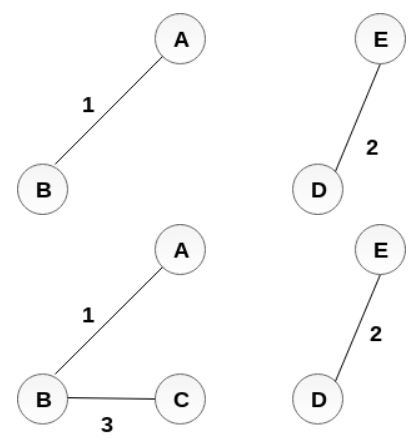


Figure 2.2.4 –Step 3.

1. The next step is to add AE, but we can't add that as it will cause a cycle.

The next edge to be added is AC, but it can't be added as it will cause a cycle.

The next edge to be added is AD, but it can't be added as it will contain a cycle.

Hence, the final MST is the one which is shown in the Figure 2.2.4.

Minimum cost= 1 + 2 + 3 + 4 = 10.

Implementation:



# Empiric analysis

Empirical analysis conducted for the number of nodes equal 3,6,9,12,15. All the values are shown in nanoseconds. Every algorithm was executed 5 times to get a medium value for every number. Every algorithm is visualized in a graph where the x-axis represents the number of nodes and the y-axis represents the time of execution of the algorithm in seconds.

Table 1.-Results

|  |  |  |
| --- | --- | --- |
| (number of nodes, number of edges) | Kruscal’s algorithm (ns) | Prim’s algorithm (ns) |
| (3,2) | 1100 | 3120200 |
| (6,6) | 2400 | 6861000 |
| (9,12) | 10400 | 6773400 |
| (12,20) | 20200 | 12032800 |
| (15,26) | 35900 | 188110800 |

Graph 1- Kruscal’s algorithm (ns).

Graph 2 –Prim’s algorithm (ns)

The graphs represent the time dependence on number of vertices. With the growing amount of number of vertices, the execution time of the algorithm increases.

Graph 3- Comparing Kruscal and Prim algorithms.

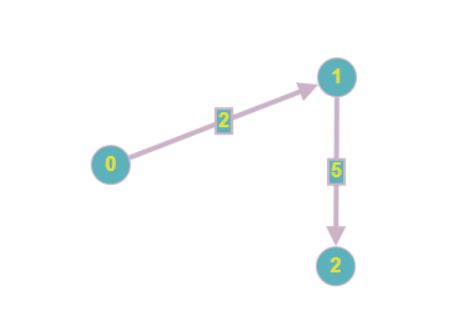
On this graph is seen the difference between the empiric time complexity of the Prim and Kruscal algorithm. The graph represent Kruscal algorithm is more efficient than Prim algorithm.

# Conclusion

The experiment examined Prim and Kruscal algorithms. Prim’s and Kruskal’s algorithms are designed for finding the minimum spanning tree of a graph. These algorithms use a different approach to solve the same problem. Prim’s algorithm works by selecting the root vertex in the beginning and then spanning from vertex to vertex adjacently, while in Kruskal’s algorithm the lowest cost edges which do not form any cycle are selected for generating the MST. These algorithms are designed for the undirected graph. The work consisted of three phases of information collection, implementation and empirical analysis. By the analysis of algorithms, was learned that time complexity of Prim is O (log n2) and of Kruskal is O (log n). During the empiric analysis, was learned, that the Kruscal algorithm is faster.

# 5. Annex

First graph:



Matrix:

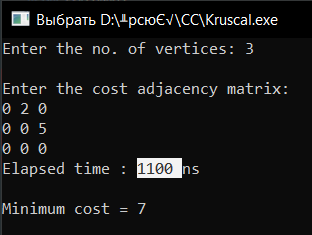
0 2 0

0 0 5

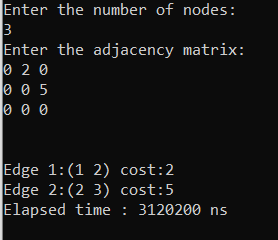
0 0 0

Result:

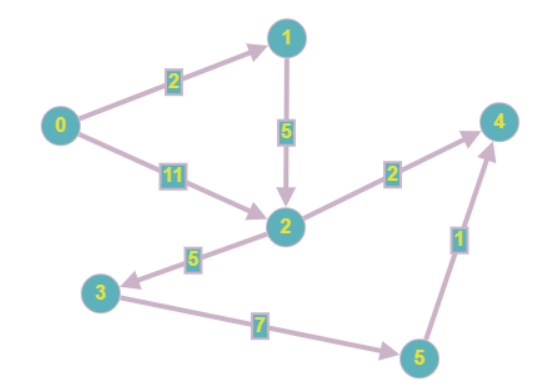
(Kruskal)



(Prim)



Second graph:



Matrix:

0 2 11 0 0 0

0 0 5 0 0 0

0 0 0 5 2 0

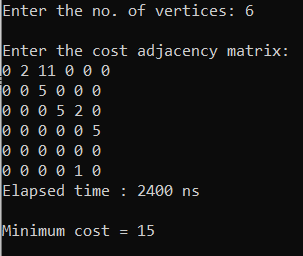
0 0 0 0 0 5

0 0 0 0 0 0

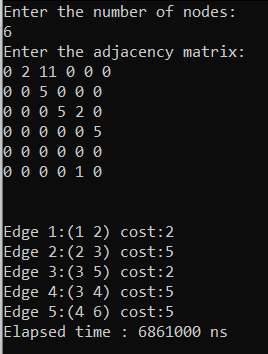
0 0 0 0 1 0

Result:

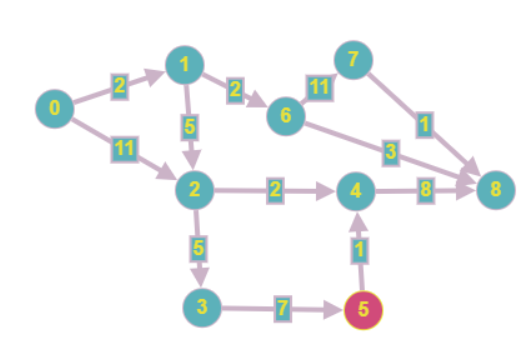
(Kruskal)



(Prim)



Third graph:



Matrix:

0 2 11 0 0 0 0 0 0

0 0 5 0 0 0 2 0 0

0 0 0 5 2 0 0 0 0

0 0 0 0 0 7 0 0 0

0 0 0 0 0 0 0 0 8

0 0 0 0 1 0 0 0 0

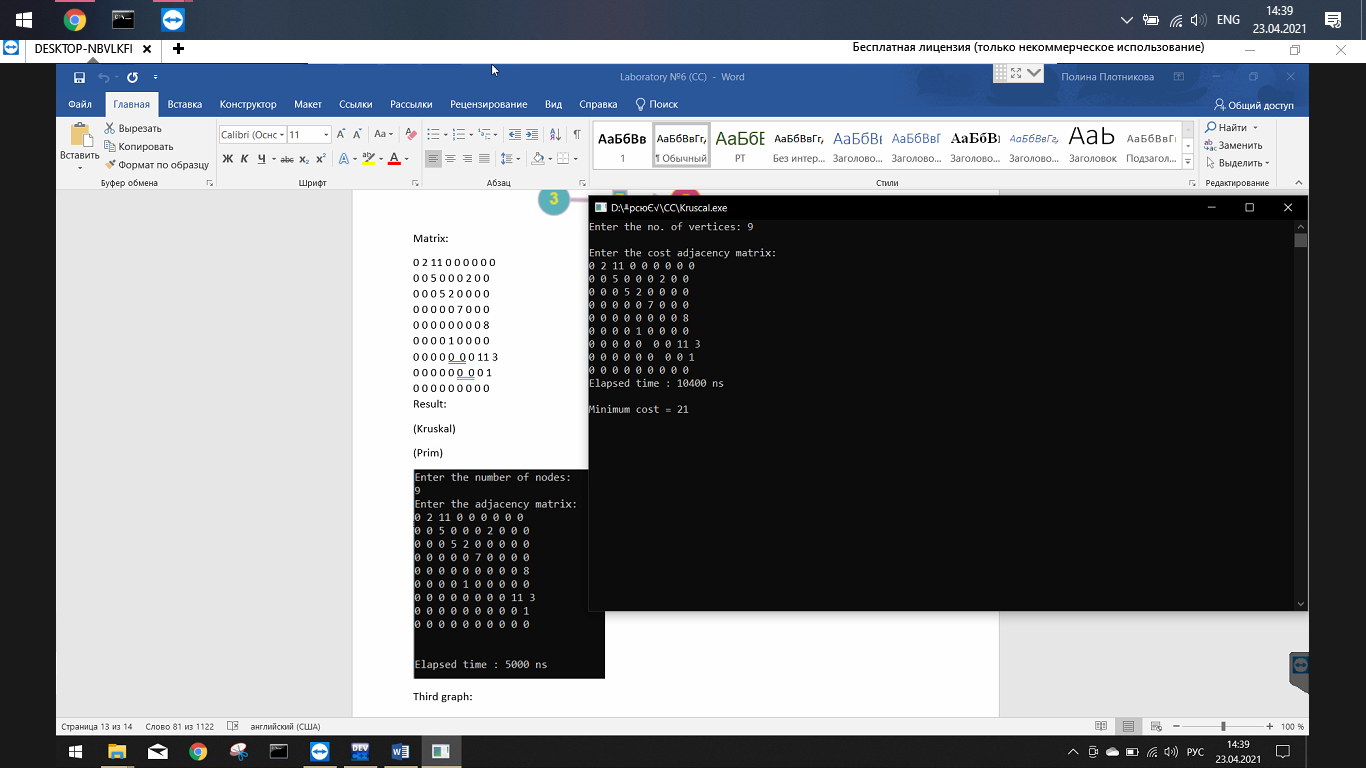
0 0 0 0 0 0 0 11 3

0 0 0 0 0 0 0 0 1

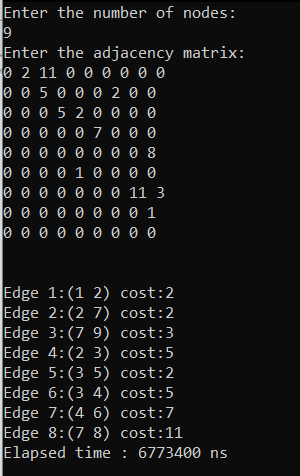
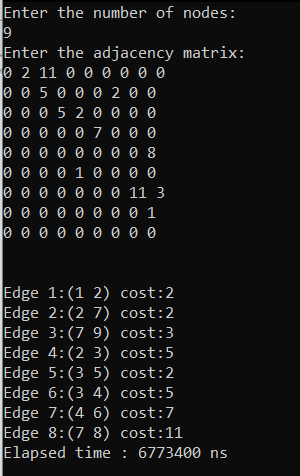
0 0 0 0 0 0 0 0 0

Result:

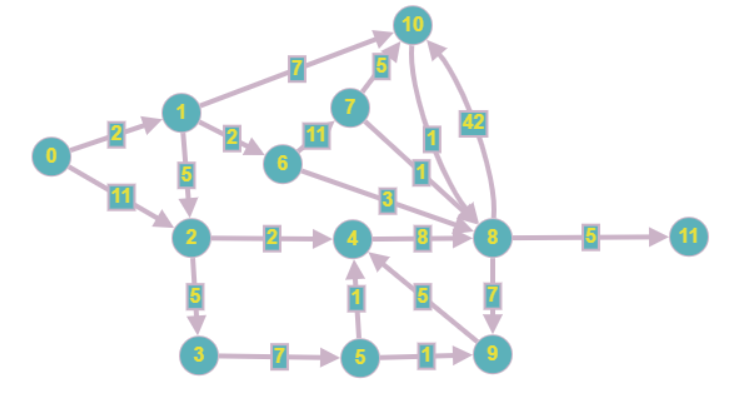
(Kruskal)



(Prim)



Fourth graph:



Matrix:

0 2 11 0 0 0 0 0 0 0 0 0

0 0 5 0 0 0 2 0 0 0 7 0

0 0 0 5 2 0 0 0 0 0 0 0

0 0 0 0 0 7 0 0 0 0 0 0

0 0 0 0 0 0 0 0 8 0 0 0

0 0 0 0 1 0 0 0 0 1 0 0

0 0 0 0 0 0 0 11 3 0 0 0

0 0 0 0 0 0 0 0 1 0 5 0

0 0 0 0 0 0 0 0 0 7 42 5

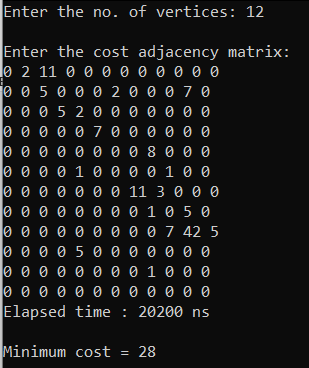
0 0 0 0 5 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0

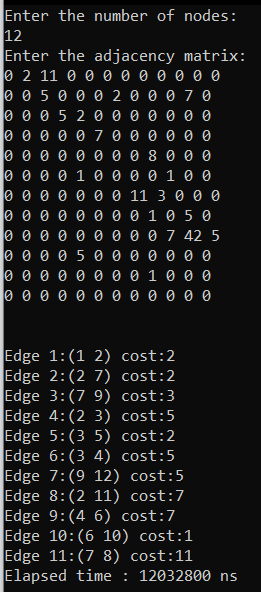
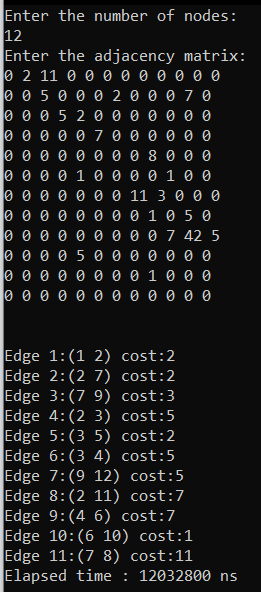
0 0 0 0 0 0 0 0 0 0 0 0

Result:

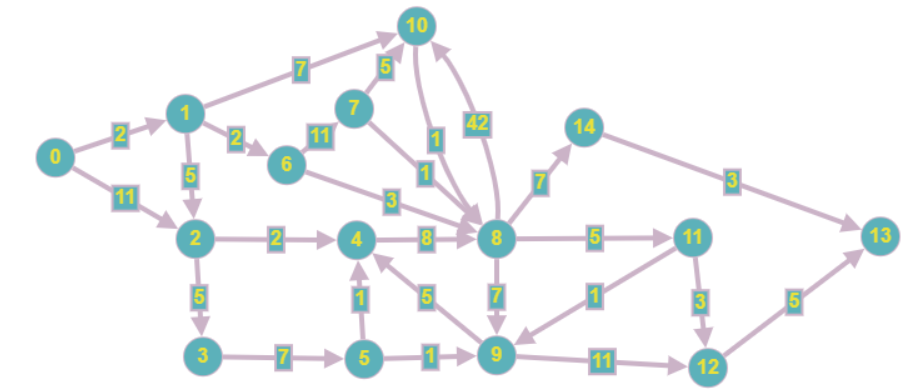
(Kruskal)



(Prim)



Fifth graph:



Matrix:

0 2 11 0 0 0 0 0 0 0 0 0 0 0 0

0 0 5 0 0 0 2 0 0 0 7 0 0 0 0

0 0 0 5 2 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 7 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 8 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 11 3 0 0 0 0 0 0

0 0 0 0 0 0 0 0 1 0 5 0 0 0 0

0 0 0 0 0 0 0 0 0 7 42 5 0 0 7

0 0 0 0 5 0 0 0 0 0 0 0 11 0 0

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1 0 3 0 0 0

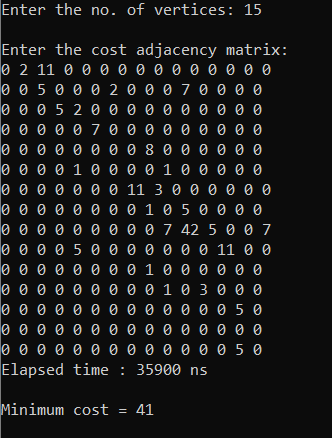
0 0 0 0 0 0 0 0 0 0 0 0 0 5 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

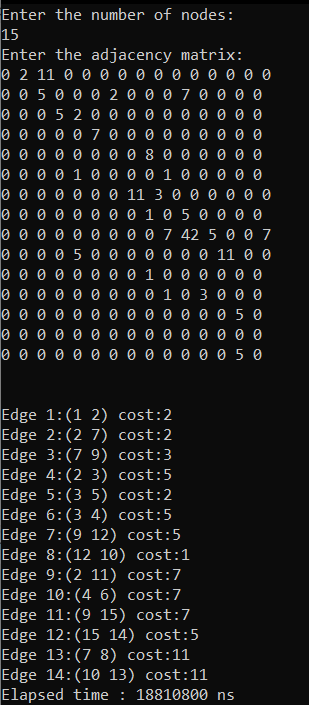
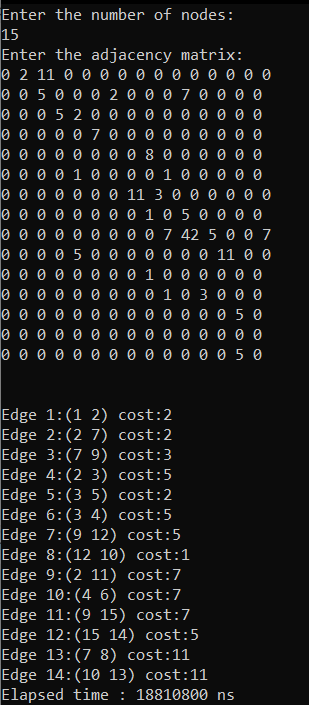
0 0 0 0 0 0 0 0 0 0 0 0 0 5 0

Result:

(Kruskal)



(Prim)



# 6. Bibliography

https://www.geeksforgeeks.org/difference-between-prims-and-kruskals-algorithm-for-mst/