**Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

REPORT

Laboratory work no.7

*Greedy Algorithms*

Elaborated:

st. gr. FAF-212 Lupascu Felicia

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2023

**Table of Contents**

[ALGORITHM ANALYSIS 3](#_bookmark0)

[Objective 3](#_bookmark1)

[Tasks 3](#_bookmark2)

[Theoretical notes 3](#_bookmark3)

[Introduction 3](#_bookmark4)

[Comparison Metric 4](#_bookmark5)

[Input Format 4](#_bookmark6)

[IMPLEMENTATION 4](#_bookmark7)

[Prim 4](#_bookmark8)

[Kruskal 6](#_bookmark9)

[Both algorithms compared 9](#_bookmark10)

[CONCLUSION 10](#_bookmark11)

[REFERENCES 10](#_bookmark12)

**Objective**

# ALGORITHM ANALYSIS

Empirical analysis for specified algorithms.

## Tasks

1. Study the greedy algorithm design technique.
2. To implement in a programming language algorithms Prim and Kruskal.
3. Empirical analyses of the Kruskal and Prim.
4. Increase the number of nodes in graph and analyze how this influences the algorithms.
5. Make a graphical presentation of the data obtained.
6. To make a report.

## Theoretical notes

An alternate approach to analyzing complexity is to use empirical analysis, which involves implementing the algorithm in a programming language and running it with multiple sets of input data to obtain data on its efficiency. This method is useful for various purposes, such as gaining preliminary information on an algorithm's complexity class, comparing the efficiency of different algorithms or implementations, or assessing the performance of an algorithm on a specific computer.

The choice of efficiency measure depends on the purpose of the analysis. If the goal is to obtain information on the complexity class or check the accuracy of a theoretical estimate, then the number of operations performed is appropriate. However, if the aim is to assess the behavior of the implementation of an algorithm, then execution time is more suitable. The results are recorded and either synthetic quantities, such as mean and standard deviation, are calculated, or a graph with appropriate pairs of points is plotted to analyze the data.

## Introduction

Prim's algorithm and Kruskal's algorithm are two popular greedy algorithms used to find the Minimum Spanning Tree (MST) of an undirected, weighted graph.

The Minimum Spanning Tree of a graph is a subset of the edges that connects all the nodes in the graph with the minimum possible total edge weight. In other words, it is a tree that spans all the nodes in the graph with the minimum possible cost.

Prim's algorithm works by starting with an arbitrary node and adding the edge with the minimum cost that connects this node to an unvisited node. We then repeat this process, adding the edge with the minimum cost that connects any of the visited nodes to an unvisited node, until all nodes are visited. This process generates a Minimum Spanning Tree.

Kruskal's algorithm works by initially creating a forest where each node is a separate tree. We then iteratively add the edge with the minimum cost to the forest, if the two nodes being connected by the edge are not already in the same tree. This process merges trees until all nodes are in the same tree and generates a Minimum Spanning Tree.

## Comparison Metric

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)).

## Input Format

A graph where the number of nodes is increased.

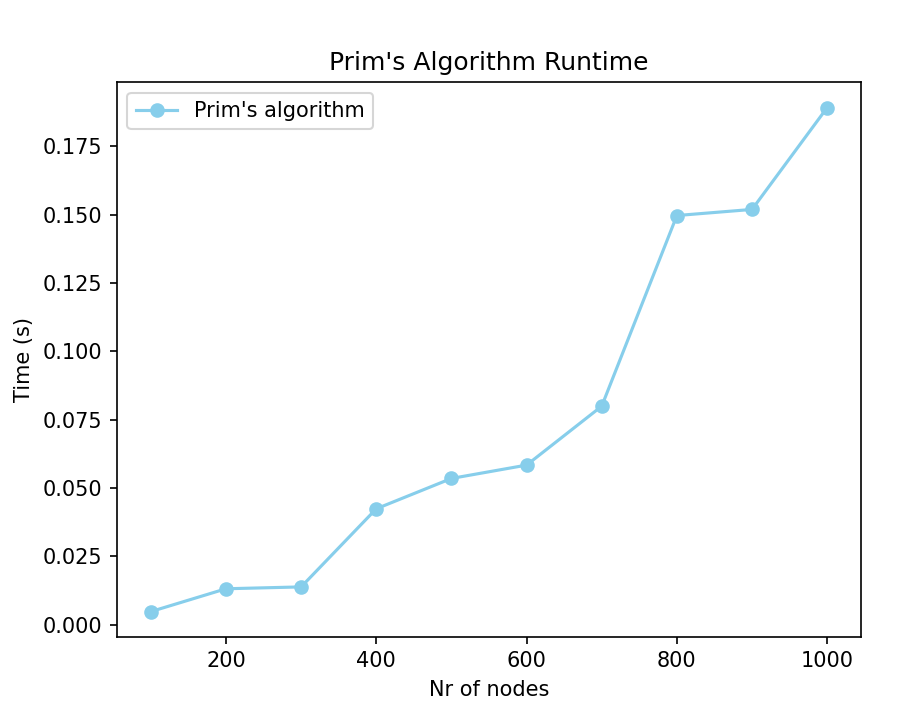
# IMPLEMENTATION

## Prim

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

def prim(graph):  
 num\_nodes = len(graph)  
 key = [sys.maxsize] \* num\_nodes  
 parent = [None] \* num\_nodes  
 mst\_set = [False] \* num\_nodes  
 key[0] = 0  
 parent[0] = -1  
 for \_ in range(num\_nodes - 1):  
 u = find\_min\_key(key, mst\_set, num\_nodes)  
 mst\_set[u] = True  
 for v in range(num\_nodes):  
 if graph[u][v] and not mst\_set[v] and graph[u][v] < key[v]:  
 key[v] = graph[u][v]  
 parent[v] = u  
 mst = []  
 for v in range(1, num\_nodes):  
 mst.append((parent[v], v, graph[parent[v]][v]))  
 return mst

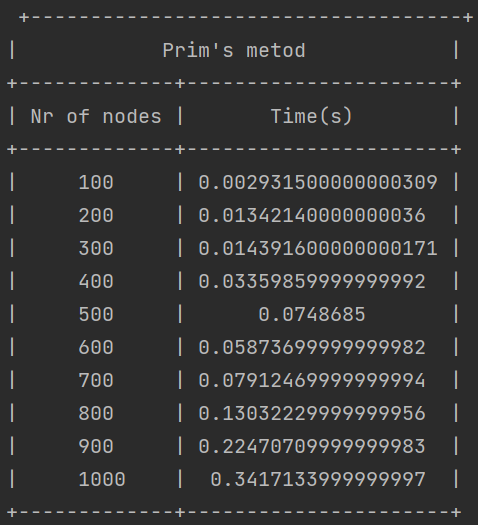
1. The first function takes a graph as input, represented as an adjacency matrix.
2. It initializes some variables such as num\_nodes (number of nodes in the graph), key (key values used for finding the minimum weight edges), parent (stores the parent of each node in the MST), and mst\_set (tracks the nodes included in the MST).
3. The key values are initially set to a large value (sys.maxsize), except for the first node which is set to 0. The parent values are initially set to None.
4. The main loop runs for num\_nodes - 1 iterations, as the MST will have num\_nodes - 1 edges.
5. In each iteration, it selects the node with the minimum key value from the set of nodes not yet included in the MST.
6. It marks the selected node as visited (mst\_set[u] = True) and updates the key values and parent pointers for the adjacent nodes if they have smaller edge weights.
7. After the loop, the MST is constructed by appending the edges (parent[v], v, graph[parent[v]][v]) for each node v (excluding the root node).
8. Finally, the function returns the MST as a list of edges.



The Time Complexity is O(ElogV), where E is the number of edges and V is the number of vertices in the graph.

The total number of edges in a graph can be at most O(V^2), which gives us O(V^2 logV) as the worst-case time complexity of Prim's algorithm. However, in practice, most graphs are sparse, and the number of edges is much smaller than O(V^2). In this case, the time complexity can be approximated as O(ElogV), which is much more efficient.

Thus, Prim's algorithm is efficient for most practical cases and can handle graphs with many vertices, if the number of edges is not too large.

**\**

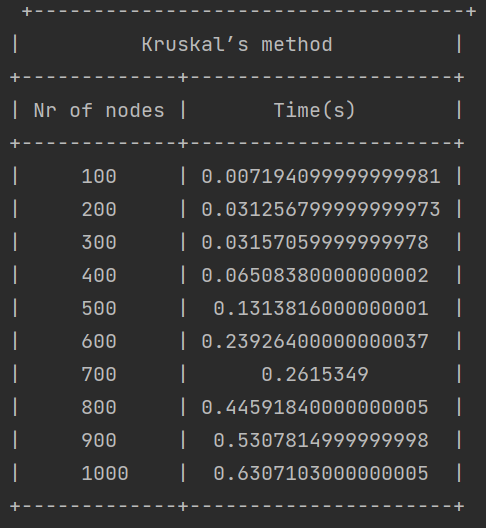
**Kruskal**

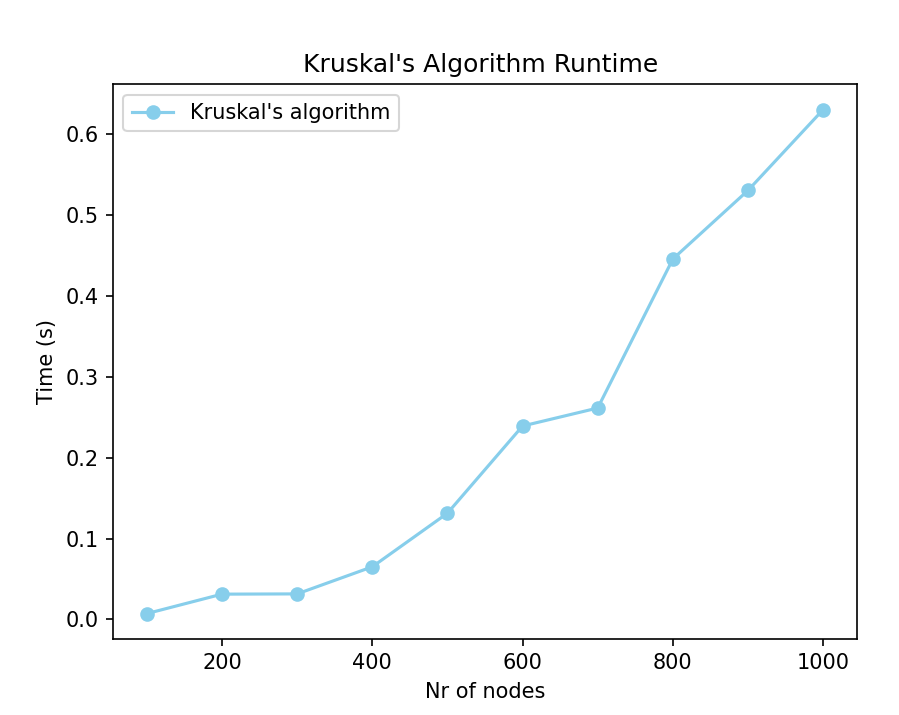
The Kruskal function implements Kruskal's algorithm for finding the minimum spanning tree (MST) of a graph. It takes a graph represented as an adjacency matrix as input.The algorithm works by first creating a list of edges in the graph, excluding zero-weight edges. These edges are then sorted in non-decreasing order based on their weights.Next, it initializes a disjoint-set data structure using dictionaries to store the parent and rank of each vertex.The algorithm then iterates over the sorted edges and checks if including the edge forms a cycle in the MST. If not, the edge is added to the MST and the disjoint-set data structure is updated.Finally, the function returns the minimum spanning tree as a list of edges.

def kruskal(graph):  
 num\_nodes = len(graph)  
 edges = []  
 for u in range(num\_nodes):  
 for v in range(u + 1, num\_nodes):  
 weight = graph[u][v]  
 if weight != 0:  
 edges.append((u, v, weight))

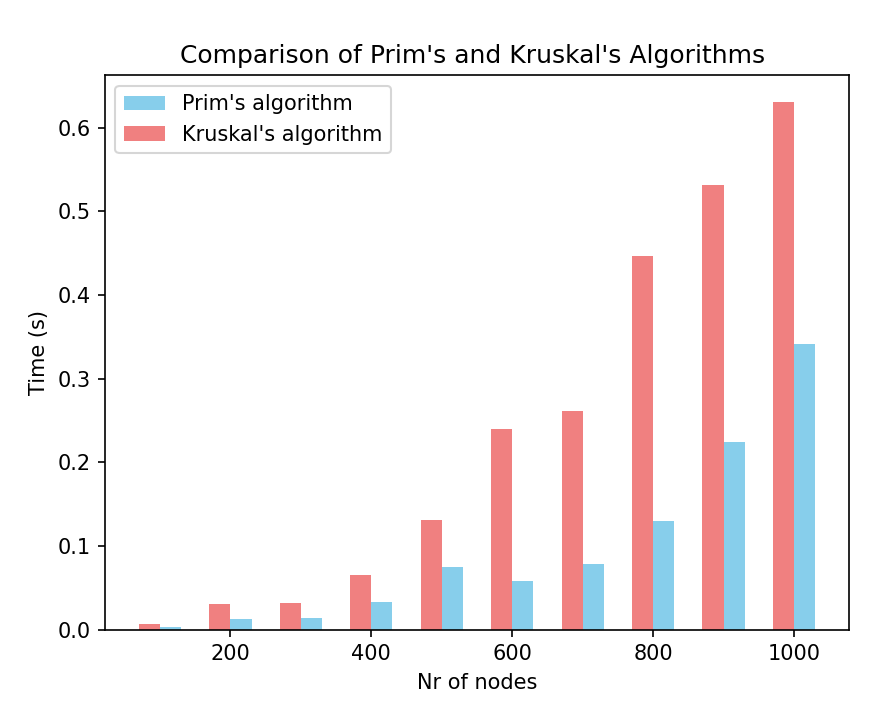
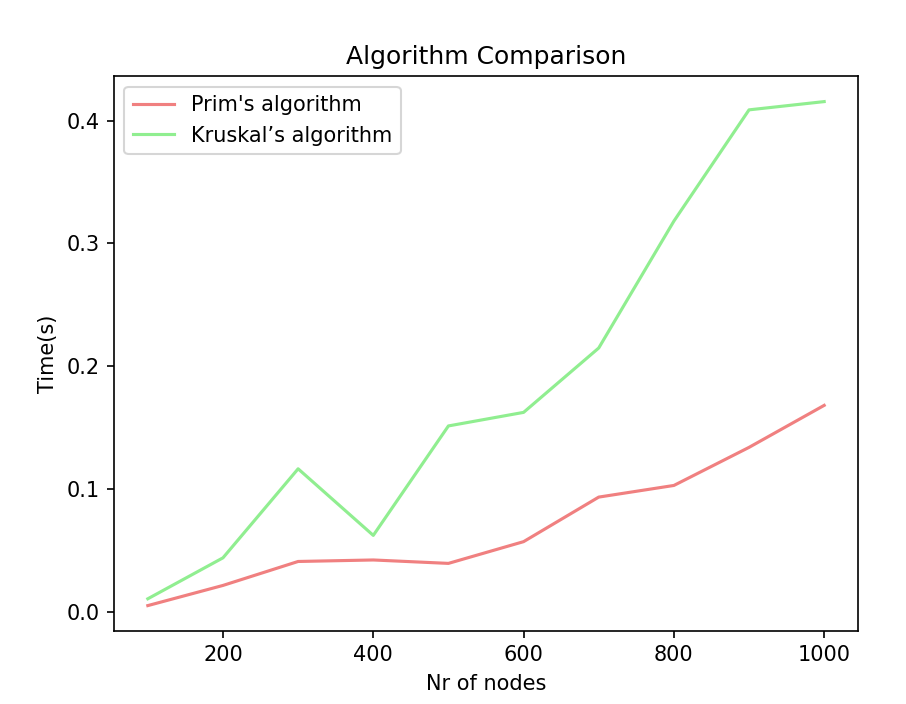
sorted\_edges = sorted(edges, key=lambda x: x[2])  
 parent = {v: v for v in range(num\_nodes)}  
 rank = {v: 0 for v in range(num\_nodes)}  
 mst = []  
 for edge in sorted\_edges:  
 u, v, weight = edge  
 if find(parent, u) != find(parent, v):  
 mst.append(edge)  
 union(parent, rank, u, v)  
 return mst

The time complexity of Kruskal's algorithm is O(ElogE), where E is the number of edges in the graph.The algorithm works by sorting all the edges in non-decreasing order of their weight, which takes O(ElogE) time using an efficient sorting algorithm like QuickSort or MergeSort. We then iterate over the sorted edges and add them to the Minimum Spanning Tree if they do not create a cycle. Checking for cycles can be done efficiently using the Union-Find algorithm, which has a time complexity of O(logV) per operation, where V is the number of vertices in the graph.In practice, the time complexity of Kruskal's algorithm can be approximated as O(ElogE), as the number of edges is typically much smaller than the number of vertices squared (i.e., E << V^2). However, for dense graphs with a large number of edges, the time complexity can approach O(V^2 logV), which can be too slow for large graphs.





**Both algorithms compared**

# CONCLUSION

In conclusion, Prim's algorithm and Kruskal's algorithm are two popular and efficient algorithms for finding the Minimum Spanning Tree of a graph. Both algorithms have their advantages and disadvantages, and the choice of algorithm depends on the specific characteristics of the graph.

In our implementation and comparison of the two algorithms on graphs with different numbers of edges, we found that Prim's algorithm performs better on dense graphs, while Kruskal's algorithm performs better on sparse graphs. This is because Prim's algorithm uses a priority queue, which has a time complexity of O(logV) per operation, making it more efficient for dense graphs with many edges. On the other hand, Kruskal's algorithm uses Union-Find data structure to detect cycles, which has a time complexity of O(logV) per operation, making it more efficient for sparse graphs with fewer edges.

Furthermore, our implementation shows that the running time of both algorithms is highly dependent on the number of edges in the graph. As the number of edges increases, the running time of both algorithms increases as well. Therefore, when working with large graphs, it is important to choose the appropriate algorithm based on the characteristics of the graph to ensure efficient computation.

Overall, Prim's algorithm and Kruskal's algorithm are both highly efficient algorithms for finding the Minimum Spanning Tree of a graph, and their choice depends on the specific characteristics of the graph.

# REFERENCES