**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-213 Iațco Sorin

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

# Table of Contents

# ALGORITHM ANALYSIS

## Objective

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, as long as 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

As input, there is given n, the nth decimal digit of pi to determine.

## IMPLEMENTATION

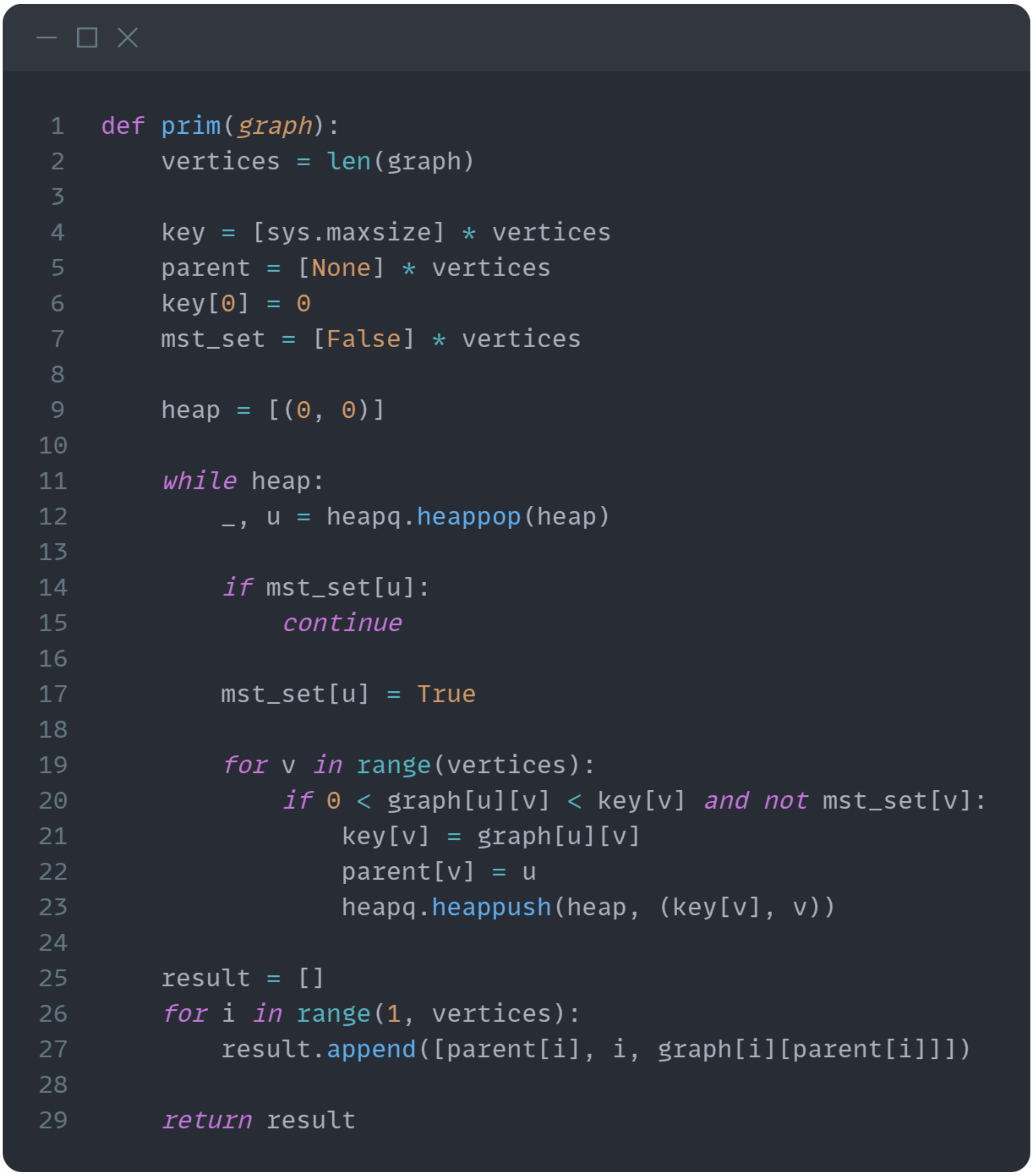
Both algorithms will be implemented in their naive form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on performance of the device used.

## Prim

*Pseudocode:*

```
prim(graph):
  mst = empty set
  startVertex = first vertex in graph
  mst.add(startVertex)
  edges = edges connected to startVertex
  while mst has fewer vertices than graph:
    minEdge, minWeight = findMinEdge(edges)
    mst.add(minEdge)
    for edge in edges connected to minEdge:
      if edge is not in mst:
        edges.add(edge)
    edges.remove(minEdge)
  return mst as an array
```

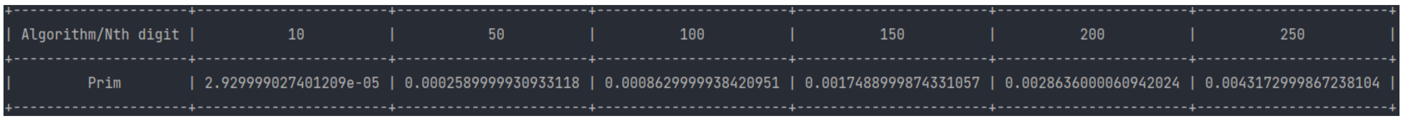
## Implementation:



```python
def prim(graph):
    vertices = len(graph)

    key = [sys.maxsize] * vertices
    parent = [None] * vertices
    key[0] = 0
    mst_set = [False] * vertices

    heap = [(0, 0)]

    while heap:
        _, u = heapq.heappop(heap)

        if mst_set[u]:
            continue

        mst_set[u] = True

        for v in range(vertices):
            if 0 < graph[u][v] < key[v] and not mst_set[v]:
                key[v] = graph[u][v]
                parent[v] = u
                heapq.heappush(heap, (key[v], v))

    result = []
    for i in range(1, vertices):
        result.append([parent[i], i, graph[i][parent[i]]])

    return result
```

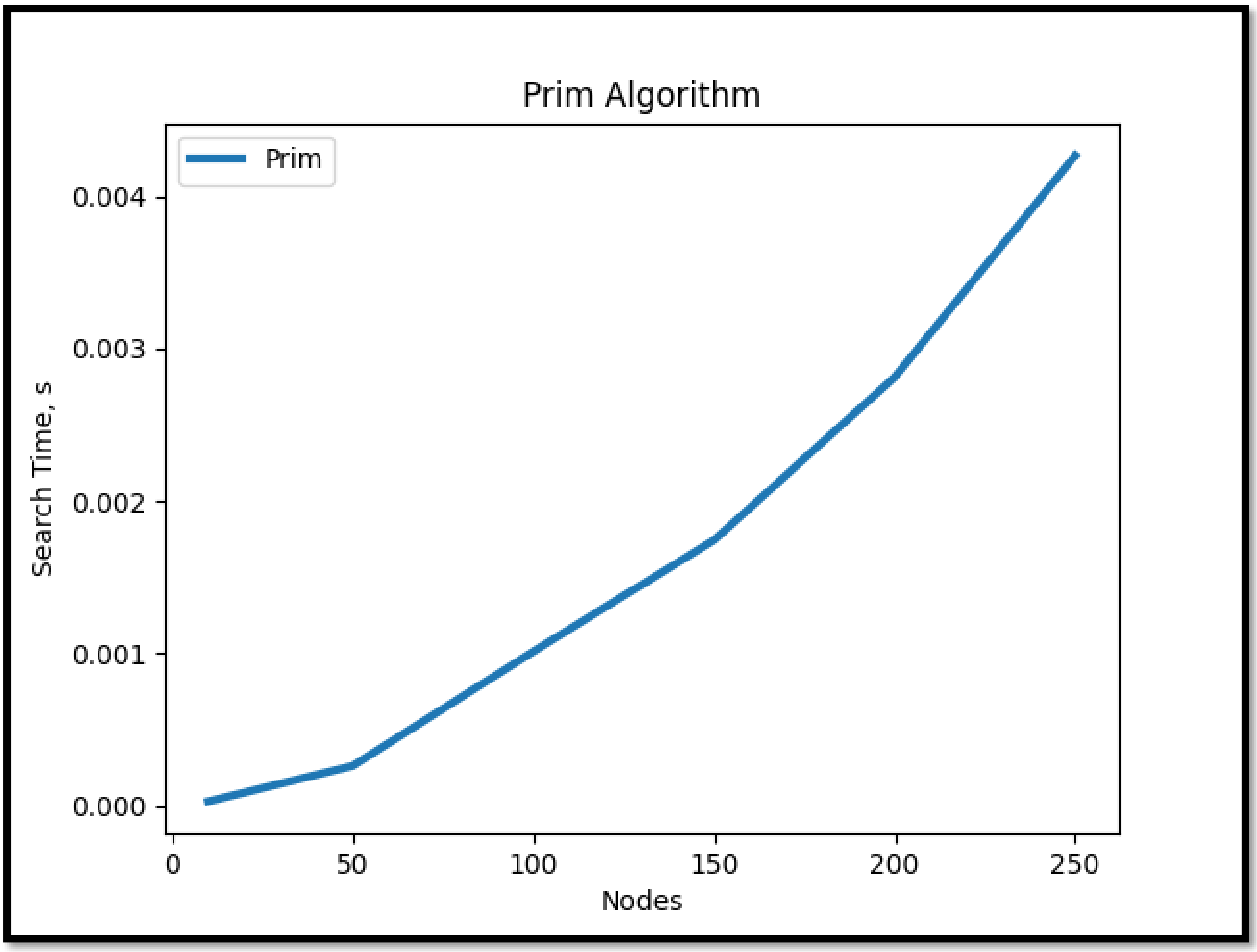
The time complexity of Prim's algorithm is O(ElogV), where E is the number of edges and V is the number of vertices in the graph.
The total number of edges in a graph can be at most O(V^2), which gives us O(V^2 logV) as the worst-case time complexity of Prim's algorithm. However, in practice, most graphs 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 a large number of vertices, as long as the number of edges is not too large.

## Results:



```
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
| Algorithm/Nth digit |          10         |          50         |         100         |         150         |         200         |         250         |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
|         Prim        | 2.929999027401209e-05 | 0.0002589999930933118 | 0.0008629999938420951 | 0.0017488999874331057 | 0.0028636000060942024 | 0.0043172999867238104 |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
```

## Graph:

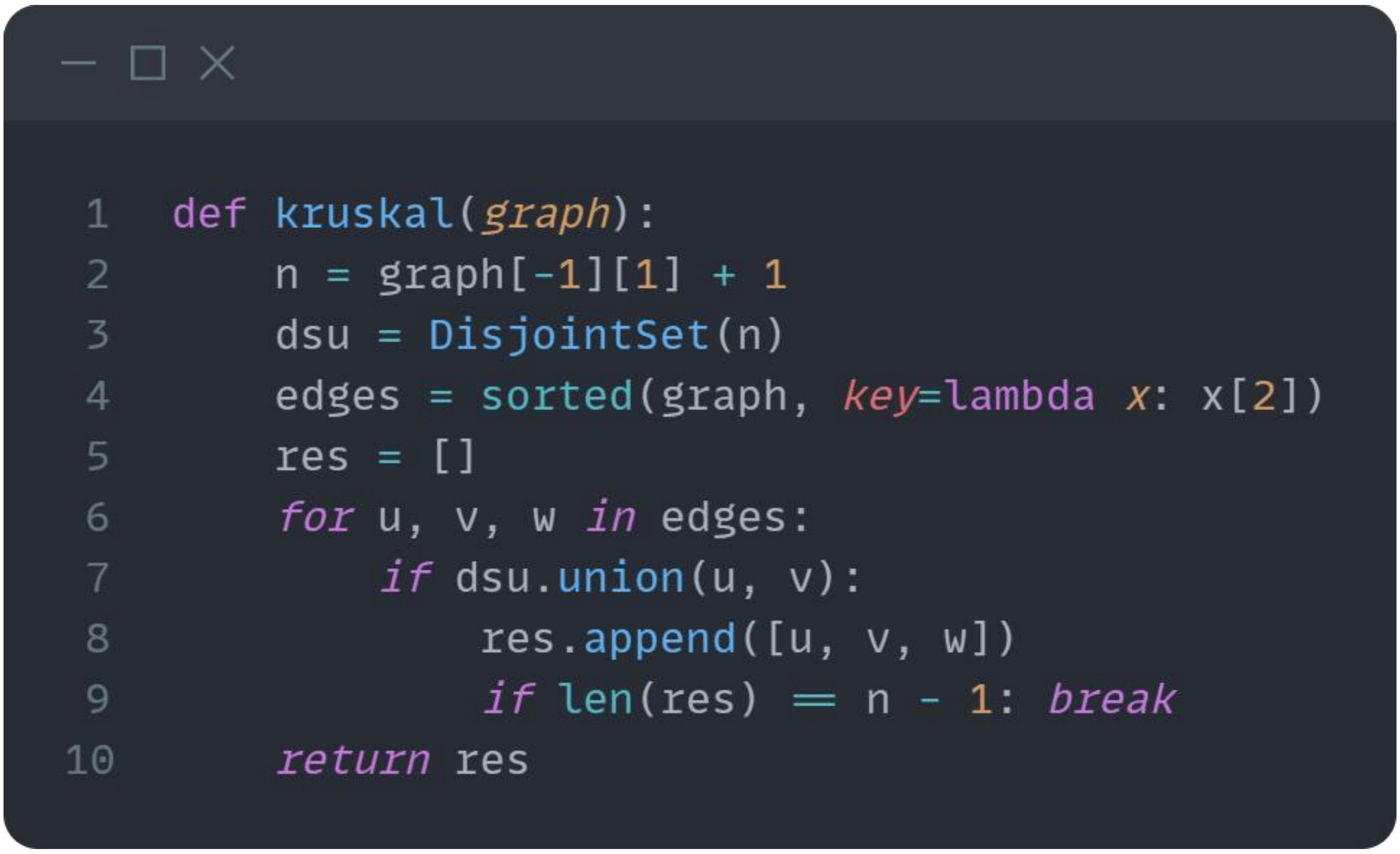


## Kruskal

*Pseudocode:*

```
kruskal(G):
A = ∅
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
    A = A ∪ {(u, v)}
    UNION(u, v)
return A
```

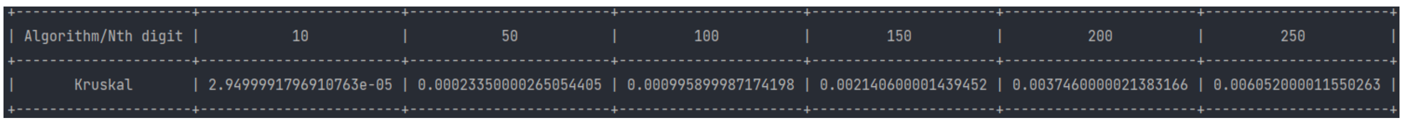
## Implementation:



```python
def kruskal(graph):
    n = graph[-1][1] + 1
    dsu = DisjointSet(n)
    edges = sorted(graph, key=lambda x: x[2])
    res = []
    for u, v, w in edges:
        if dsu.union(u, v):
            res.append([u, v, w])
            if len(res) == n - 1: break
    return res
```

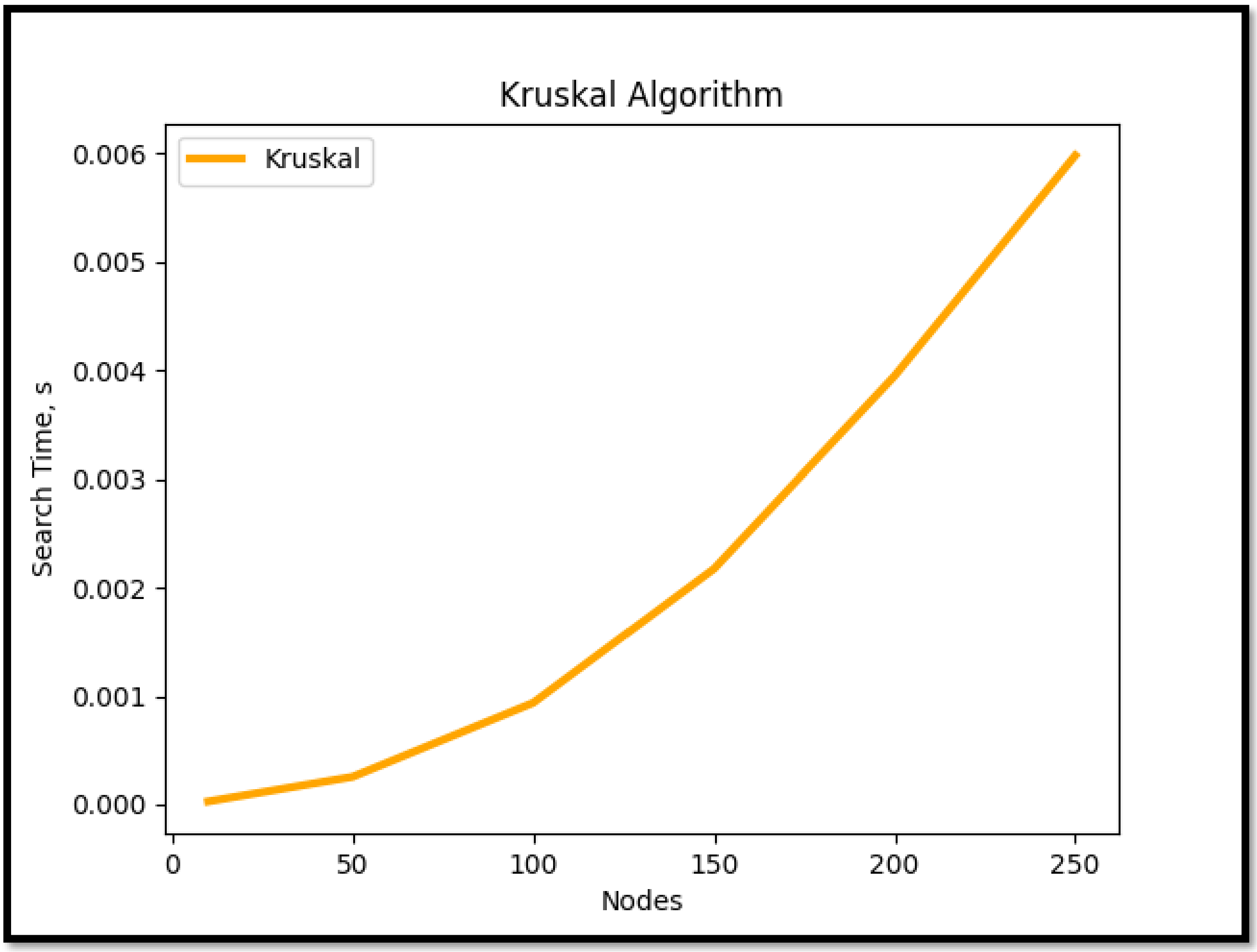
The time complexity of Kruskal's algorithm is O(ElogE), where E is the number of edges in the graph.
The algorithm works by sorting all the edges in non-decreasing order of their weight, which takes O(ElogE) time using an efficient sorting algorithm like QuickSort or MergeSort. We then iterate over the 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.

## Results:



```
+--------------------+-----------------------+-----------------------+----------------------+----------------------+----------------------+----------------------+
| Algorithm/Nth digit |          10           |          50           |         100          |         150          |         200          |         250          |
+--------------------+-----------------------+-----------------------+----------------------+----------------------+----------------------+----------------------+
|      Kruskal       | 2.9499991796910763e-05 | 0.00023350000265054405 | 0.000995899987174198 | 0.002140600001439452 | 0.0037460000021383166 | 0.006052000011550263 |
+--------------------+-----------------------+-----------------------+----------------------+----------------------+----------------------+----------------------+
```

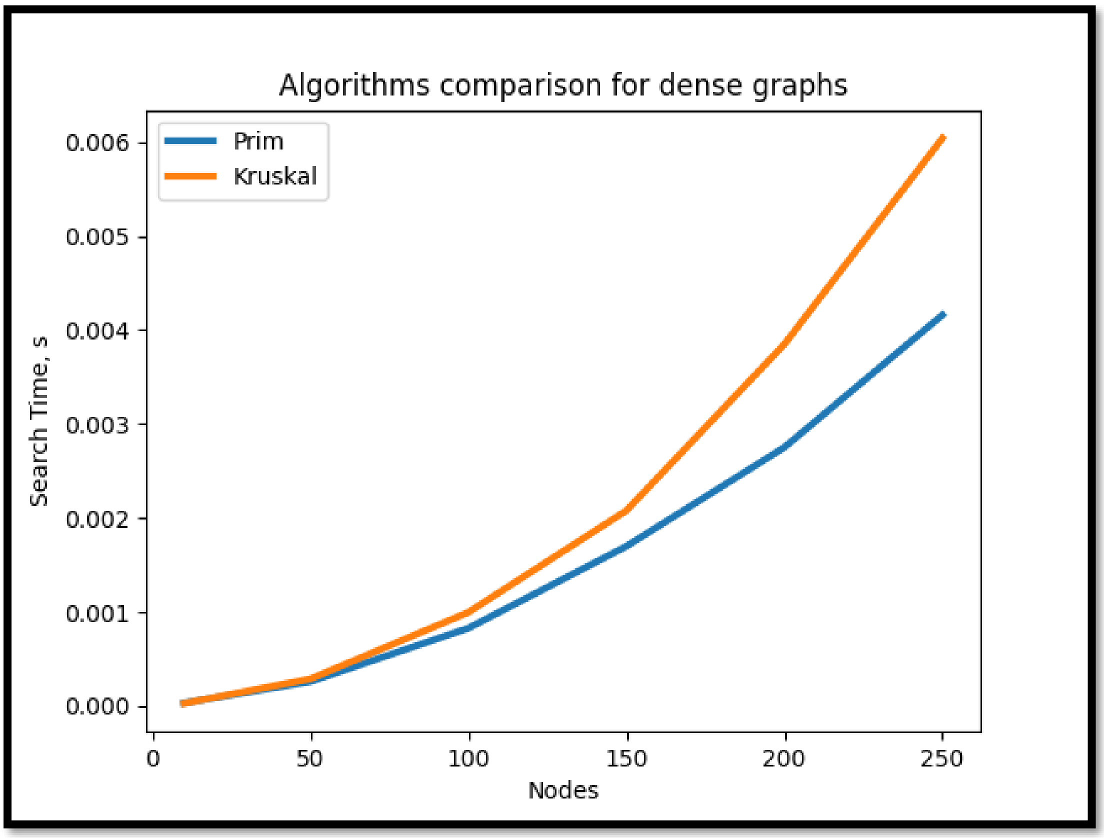
**Graph:**

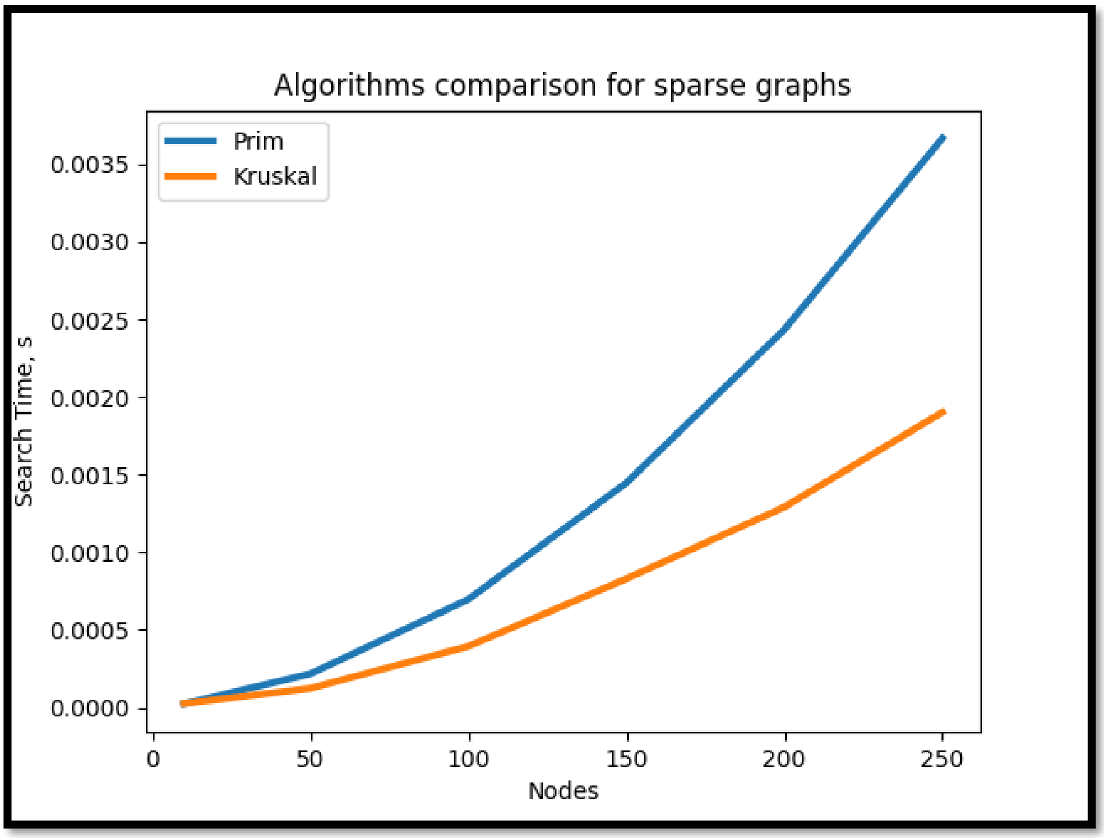


Kruskal Algorithm
Kruskal
Search Time, s
0.006
0.005
0.004
0.003
0.002
0.001
0.000
0
50
100
150
200
250
Nodes

## Both algorithms compared

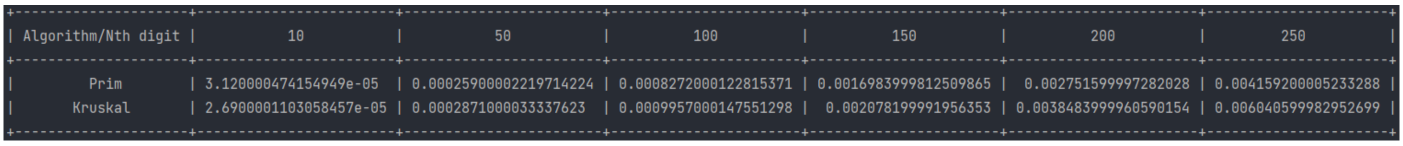
## Graphs:





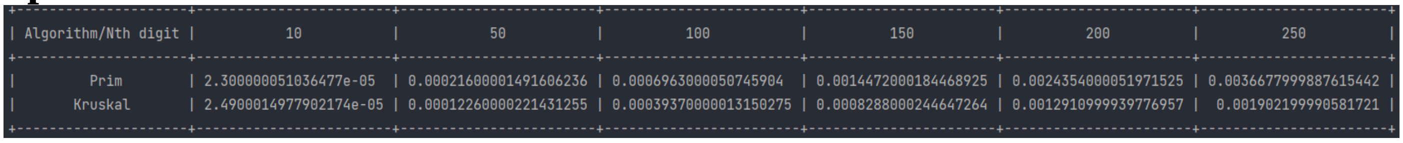
## Time Tables:

### Dense:



```
+--------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+
| Algorithm/Nth digit |           10            |           50            |           100           |           150           |           200           |           250           |
+--------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+
|        Prim        | 3.120000474154949e-05  | 0.00025900002219714224 | 0.0008272000122815371 | 0.0016983999812509865 |  0.002751599997282028 | 0.004159200005233288 |
|      Kruskal       | 2.6900001103058457e-05 | 0.0002871000033337623  | 0.0009957000147551298 |  0.002078199991956353 | 0.0038483999960590154 | 0.006040599982952699 |
+--------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+
```

### Sparse:



```
+--------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+
| Algorithm/Nth digit |           10            |           50            |           100           |           150           |           200           |           250           |
+--------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+
|        Prim        | 2.300000051036477e-05  | 0.00021600001491606236 | 0.0006963000050745904  | 0.0014472000184468925 | 0.0024354000051971525 | 0.0036677999887615442 |
|      Kruskal       | 2.4900014977902174e-05 | 0.00012260000221431255 | 0.0003937000013150275 | 0.0008288000244647264 | 0.0012910999939776957 |  0.001902199990581721 |
+--------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+
```

## 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

AA-Labs/Lab7 at main · Syn4z/AA-Labs · GitHub