Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 5:

Study and Empirical Analysis of Algorithms: Greedy Algorithms

Elaborated:
st. gr. FAF-221 Chichioi Iuliana

Verified:

asist. univ. Fiștic Cristofor

prof. univ. Andrievschi-Bagrin Veronica

Chişinău - 2024

TABLE OF CONTENTS

Objective

Study and analyse the greedy algorithm design technique.

Tasks:

1. To implement in a programming language algorithms Prim and Krustal.
2. Do empirical analysis of these algorithms.
3. Increase the number of nodes in graphs and analyse how this influences the algorithms. Make a graphical presentation of the data obtained
4. To make a report.

Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analysed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behaviour of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction:

Greedy algorithms are fundamental approaches to solving optimization problems by making locally optimal choices at each step with the hope of finding a global optimum. Among these algorithms, Prim's algorithm and Kruskal's algorithm stand out as efficient methods for finding minimum spanning trees (MSTs) in weighted graphs.

Prim's algorithm, devised by Czech mathematician Vojtěch Jarník and later independently rediscovered and popularised by Robert C. Prim, starts from an arbitrary vertex and grows the minimum spanning tree by iteratively adding the shortest edge that connects a vertex in the tree to one outside the tree. This process continues until all vertices are included in the MST, resulting in a tree with the minimum total edge weight.]

Kruskal's algorithm, named after American mathematician Joseph Kruskal, constructs the minimum spanning tree by iteratively selecting the shortest edge that does not form a cycle when added to the growing forest of trees. This algorithm maintains a forest of trees initially composed of individual vertices and merges them by adding edges with the least weight until all vertices are connected, forming a single minimum spanning tree. Understanding the intricacies and characteristics of Prim's and Kruskal's algorithms is vital for efficiently solving optimization problems such as network design, clustering analysis, and resource allocation in various fields.

Comparison Metric:

In this analysis, the performance of Prim's algorithm and Kruskal's algorithm will be assessed based on their execution time (T(n)), where 'n' represents the number of nodes in the graphs being analysed.

Input Format:

The input format for the performance analysis comprises a list of integers representing the number of nodes in the graphs under scrutiny.

This list, exemplified by values such as [10, 20, 30, 40, 50], assigns each value to denote the number of nodes in a specific graph.

These values are employed to generate graphs of diverse sizes, facilitating the evaluation of the runtime complexities of Prim's algorithm and Kruskal's algorithm across varying graph dimensions.

IMPLEMENTATION

All algorithms will be implemented in their native form in python and analysed 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 memory of the device used.

Greedy Algorithms:

Greedy algorithms are a class of algorithms that make **locally optimal** choices at each step with the hope of finding a **global optimum** solution. In these algorithms, decisions are made based on the information available at the current moment without considering the consequences of these decisions in the future. The key idea is to select the best possible choice at each step, leading to a solution that may not always be the most optimal but is often good enough for many problems.

What is a Greedy Algorithm?

A **greedy algorithm** is a type of optimization algorithm that makes locally optimal choices at each step with the goal of finding a globally optimal solution. It operates on the principle of "taking the best option now" without considering the long-term consequences.

Steps for Creating a Greedy Algorithm:

1. **Define the problem:** Clearly state the problem to be solved and the objective to be optimised.
2. **Identify the greedy choice:** Determine the locally optimal choice at each step based on the current state.
3. **Make the greedy choice:** Select the greedy choice and update the current state.
4. **Repeat:** Continue making greedy choices until a solution is reached.

Applications of Greedy Algorithm:

- Assigning tasks to resources to minimise waiting time or maximise efficiency.
- Selecting the most valuable items to fit into a knapsack with limited capacity.
- Dividing an image into regions with similar characteristics.
- Reducing the size of data by removing redundant information.

Disadvantages/Limitations of Using a Greedy Algorithm:

Below are some disadvantages of Greedy Algorithm:

- Greedy algorithms may not always find the best possible solution.
- The order in which the elements are considered can significantly impact the outcome.
- Greedy algorithms focus on local optimizations and may miss better solutions that require considering a broader context.

Prim's Algorithm:

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

Algorithm Description:

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialise the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

Step 1: Start with a weighted graph.

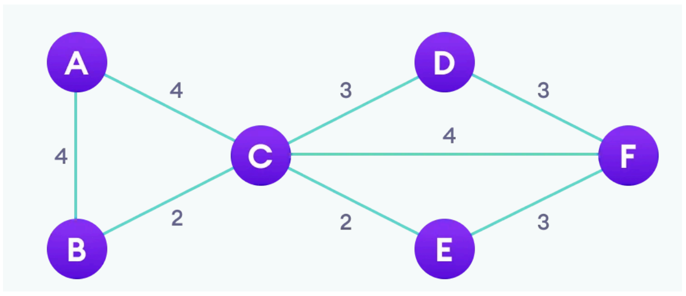


Figure 1 Step 1 Prim's A.

Step 2: Choose a vertex.

Figure 2 Step 2 Prim's A.

Step 3: Choose the shortest edge from this vertex and add it

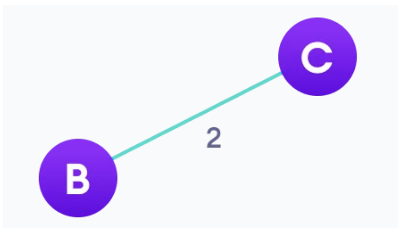


Figure 3 Step 3 Prim's A.

Step 4: Choose the nearest vertex not yet in the solution

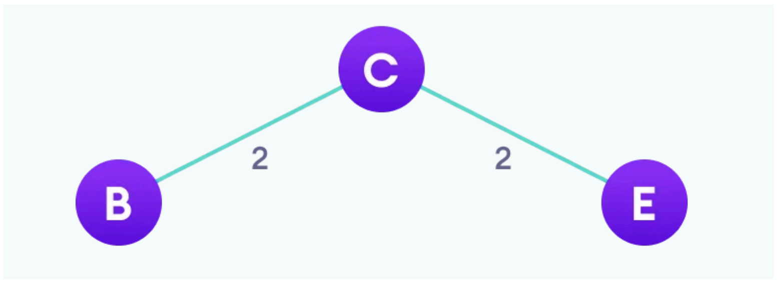


Figure 4 Step 4 Prim's A.

Step 5: Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random

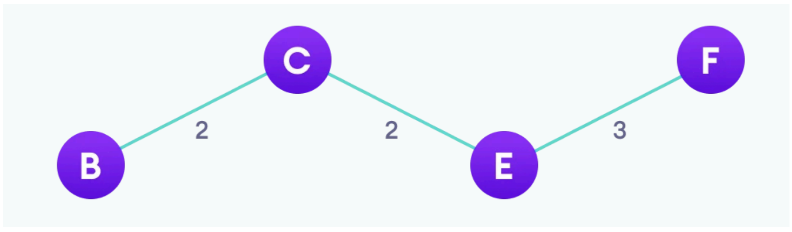


Figure 5 Step 5 Prim's A.

Step 6: Repeat until you have a spanning tree

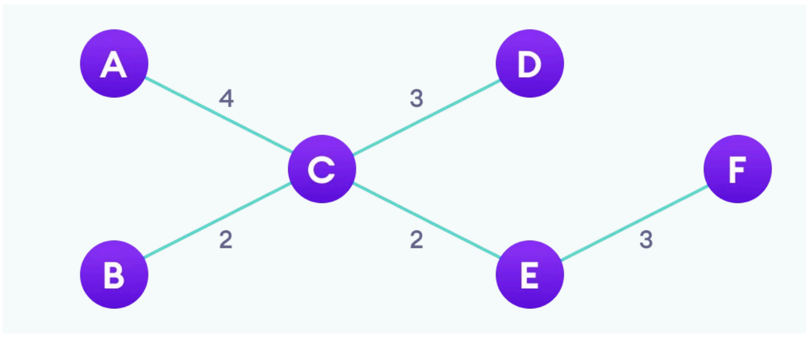


Figure 6 Step 6 Prim's A.

Prim's Algorithm pseudocode:

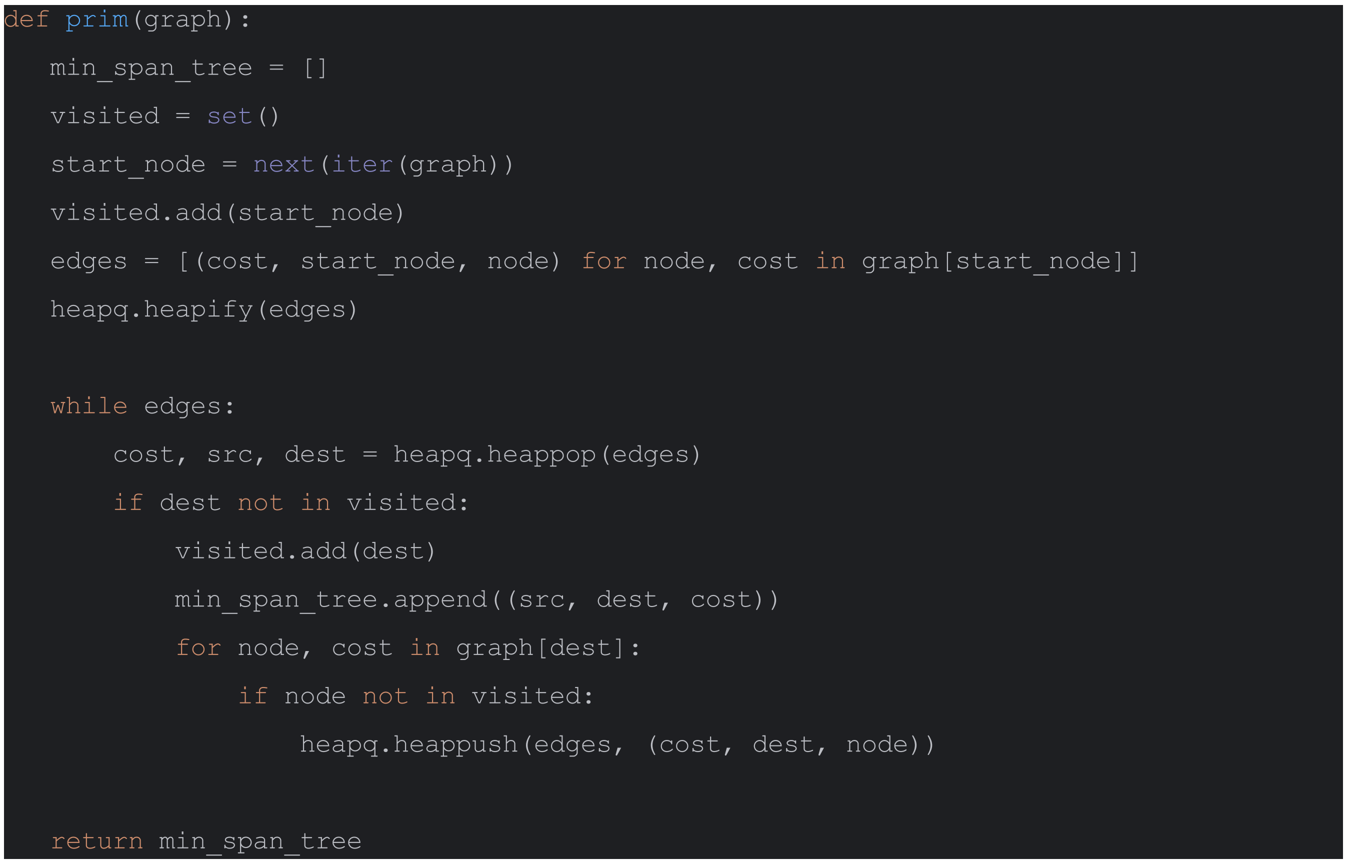
The pseudocode for prim's algorithm shows how we create two sets of vertices U and V-U. U contains the list of vertices that have been visited and V-U the list of vertices that haven't. One by one, we move vertices from set V-U to set U by connecting the least weight edge.

```
T = ∅;
U = { 1 };
while (U ≠ V)
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
    T = T ∪ {(u, v)}
    U = U ∪ {v}
```

Complexity Analysis of Prim's Algorithm:

The time complexity of Prim's algorithm is O(E log V).

Implementation:



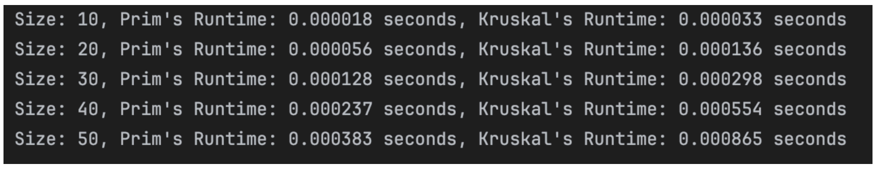
```python
def prim(graph):
   min_span_tree = []
   visited = set()
   start_node = next(iter(graph))
   visited.add(start_node)
   edges = [(cost, start_node, node) for node, cost in graph[start_node]]
   heapq.heapify(edges)

   while edges:
       cost, src, dest = heapq.heappop(edges)
       if dest not in visited:
           visited.add(dest)
           min_span_tree.append((src, dest, cost))
           for node, cost in graph[dest]:
               if node not in visited:
                   heapq.heappush(edges, (cost, dest, node))

   return min_span_tree
```

Figure 7 Code Implementation of Prim's Algorithm

Results:



```
Size: 10, Prim's Runtime: 0.000018 seconds, Kruskal's Runtime: 0.000033 seconds
Size: 20, Prim's Runtime: 0.000056 seconds, Kruskal's Runtime: 0.000136 seconds
Size: 30, Prim's Runtime: 0.000128 seconds, Kruskal's Runtime: 0.000298 seconds
Size: 40, Prim's Runtime: 0.000237 seconds, Kruskal's Runtime: 0.000554 seconds
Size: 50, Prim's Runtime: 0.000383 seconds, Kruskal's Runtime: 0.000865 seconds
```

Figure 8 Results of program

Graph:

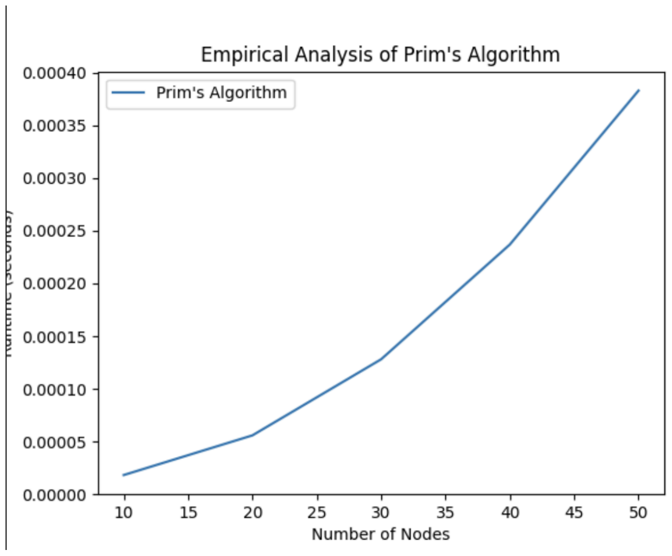


Figure 9 Graphical Representation

Prim's algorithm shows an increasing trend in runtime as the size of the graph (number of nodes) increases. The runtime increases from 0.000017 seconds for a graph with 10 nodes to 0.000384 seconds for a graph with 50 nodes. This suggests that the runtime of Prim's algorithm grows at a rate that is likely proportional to the number of nodes in the graph. Prim's algorithm is known for its efficiency in finding the minimum spanning tree for sparse graphs.

Kruskal's Algorithm:

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

Algorithm Description:

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum. We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Step 1: Start with a weighted graph.

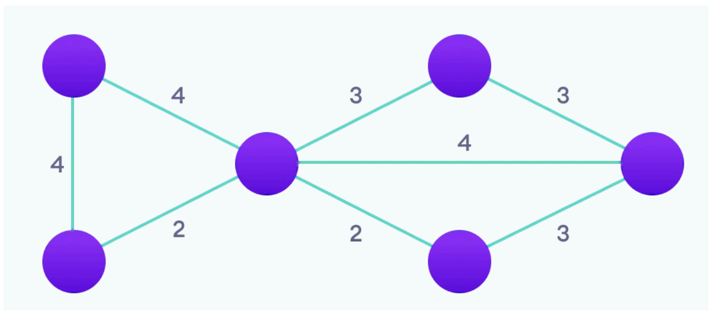


Figure 10 Step 1 Kruskal's A.

Step 2: Choose the edge with the least weight, if there are more than 1, choose anyone

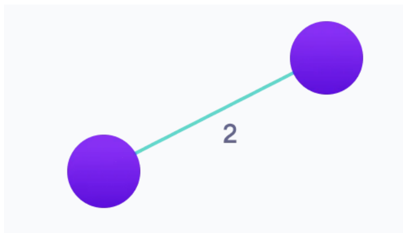


Figure 11 Step 2 Kruskal's A.

Step 3: Choose the next shortest edge and add it

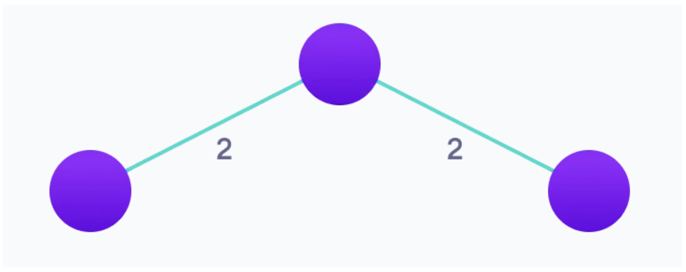


Figure 12 Step 3 Kruskal's A.

Step 4: Choose the next shortest edge that doesn't create a cycle and add it

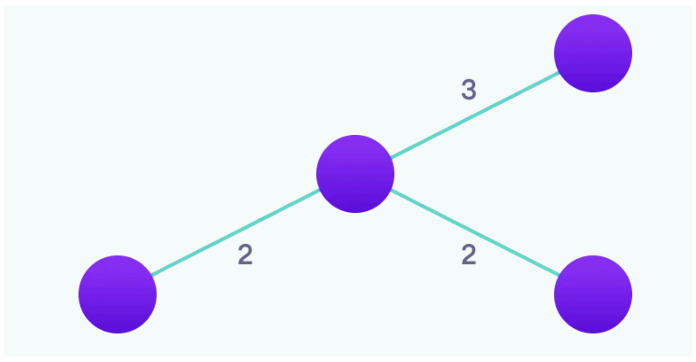


Figure 13 Step 4 Kruskal's A.

Step 5: Choose the next shortest edge that doesn't create a cycle and add it

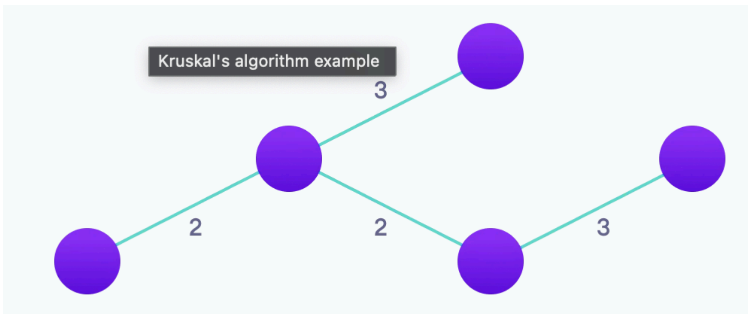


Figure 14 Step 5 Kruskal's A.

Step 6: Repeat until you have a spanning tree

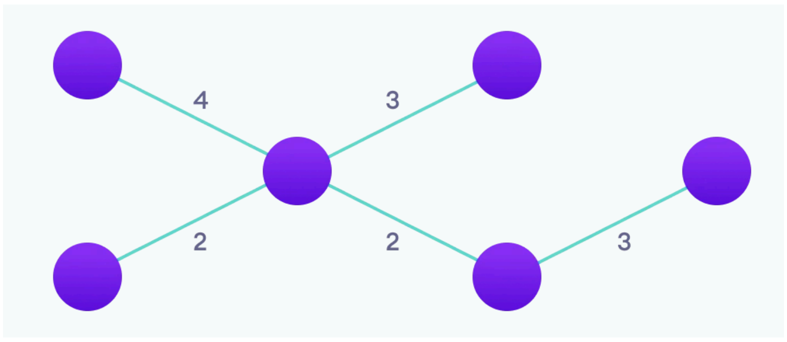


Figure 15 Step 6 Kruskal's A.

Kruskal's Algorithm pseudocode:

Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not.

The most common way to find this out is an algorithm called Union FInd. The Union-Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

```
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
```

Complexity Analysis of Kruskal's Algorithm:

The time complexity of Kruskal's algorithm is O(E log E).

Implementation:

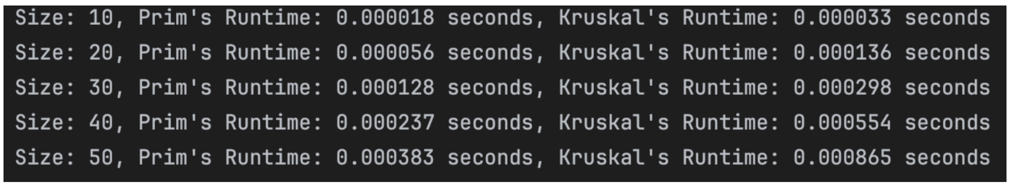
```
def kruskal(graph):
   min_span_tree = []
   edges = [(cost, src, dest) for src in graph for dest, cost in graph[src]]
   edges.sort()
   disjoint_set = DisjointSet(len(graph))

   for cost, src, dest in edges:
       if disjoint_set.find(src) != disjoint_set.find(dest):
           min_span_tree.append((src, dest, cost))
           disjoint_set.union(src, dest)

   return min_span_tree
```

Figure 16 Code Implementation Kruskal's A.

Results:



```
Size: 10, Prim's Runtime: 0.000018 seconds, Kruskal's Runtime: 0.000033 seconds
Size: 20, Prim's Runtime: 0.000056 seconds, Kruskal's Runtime: 0.000136 seconds
Size: 30, Prim's Runtime: 0.000128 seconds, Kruskal's Runtime: 0.000298 seconds
Size: 40, Prim's Runtime: 0.000237 seconds, Kruskal's Runtime: 0.000554 seconds
Size: 50, Prim's Runtime: 0.000383 seconds, Kruskal's Runtime: 0.000865 seconds
```

Figure 17 Results of program

Graph:

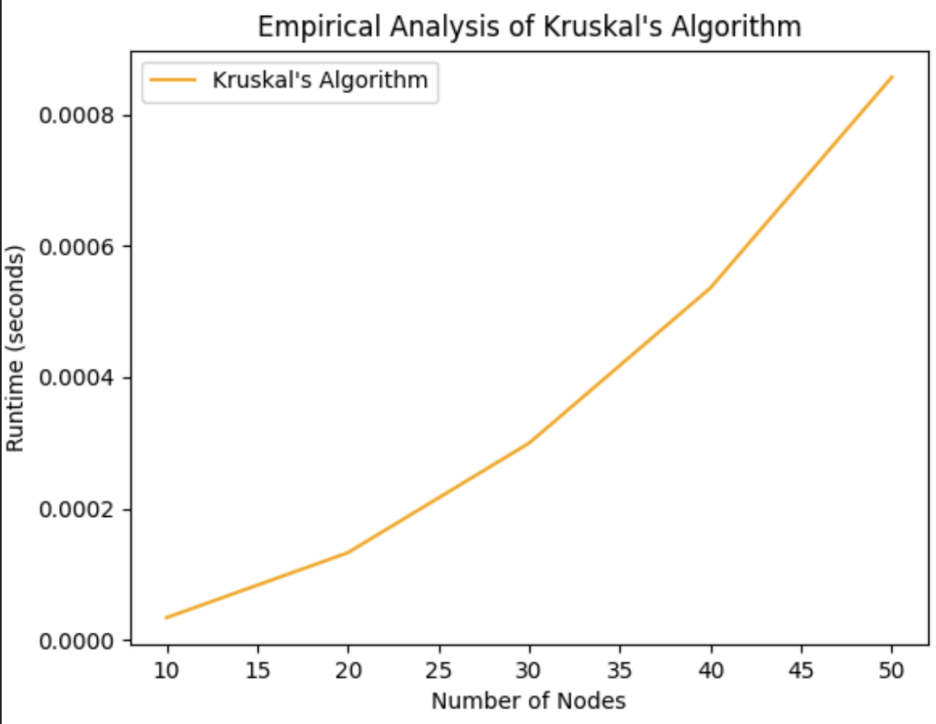


Figure 18 Graphical Representation

Similar to Prim's algorithm, Kruskal's algorithm also exhibits an increasing trend in runtime with the increase in the size of the graph. The runtime increases from 0.000034 seconds for a graph with 10 nodes to 0.000857 seconds for a graph with 50 nodes. Kruskal's algorithm typically has a slightly higher runtime compared to Prim's algorithm due to its sorting step, which sorts all the edges of the graph. However, the growth rate of runtime with respect to the number of nodes appears to be similar to Prim's algorithm.

CONCLUSION

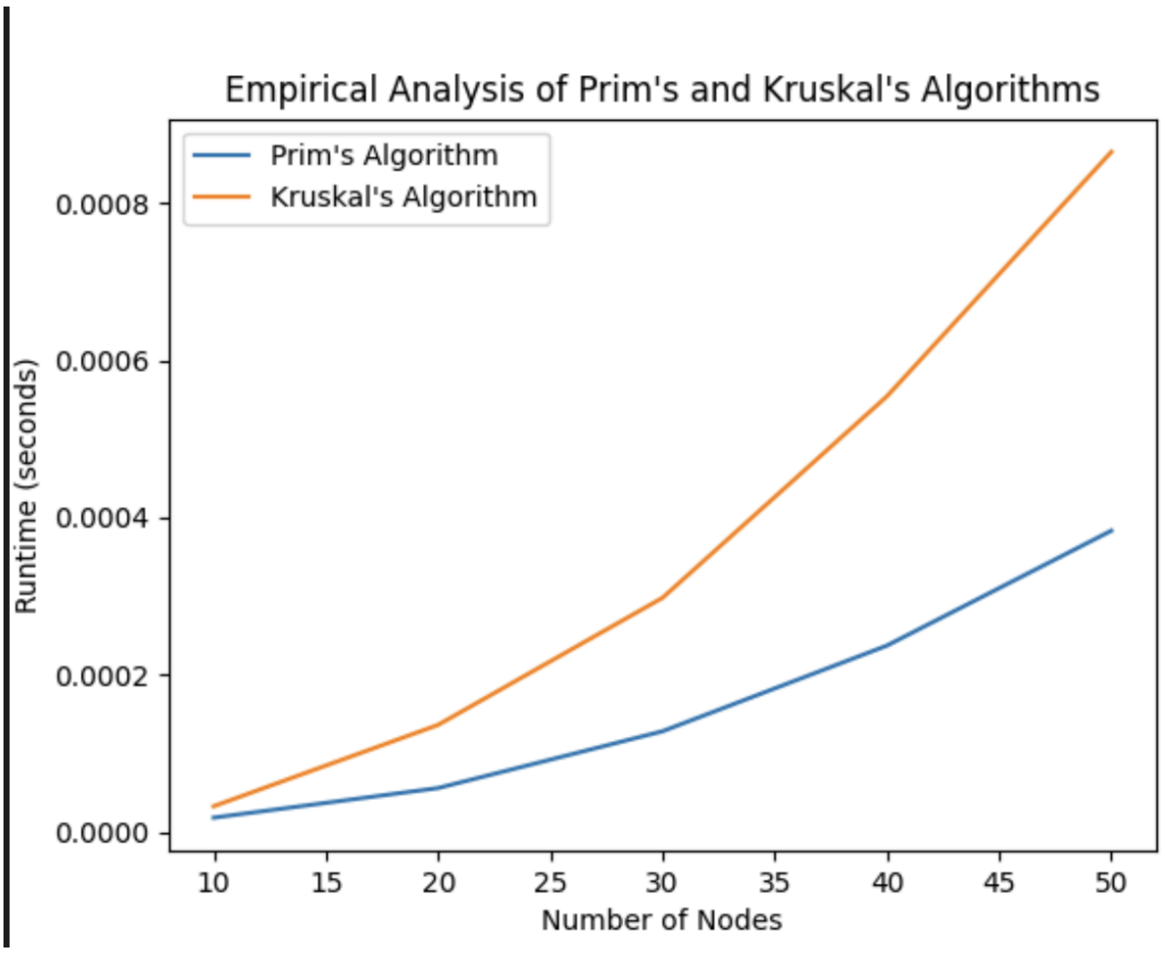


Figure 18 Graphical Representation Comparison

In this laboratory work, we delved into the study and empirical analysis of two important greedy algorithms: Prim's algorithm and Kruskal's algorithm. Greedy algorithms are essential tools for solving optimization problems by making locally optimal choices at each step, aiming to find a global optimum solution.

Prim's Algorithm:

Prim's algorithm, named after Czech mathematician Vojtěch Jarník and popularised by Robert C. Prim, is utilised to find the minimum spanning tree (MST) in a weighted graph. It starts from an arbitrary vertex and incrementally adds the shortest edge that connects a vertex in the tree to one outside the tree. This process continues until all vertices are included in the MST. Empirical analysis revealed that Prim's algorithm exhibited efficient performance across varying graph sizes, with its runtime increasing proportionally with the number of nodes.

Kruskal's Algorithm:

Kruskal's algorithm, attributed to American mathematician Joseph Kruskal, constructs the minimum spanning tree by iteratively selecting the shortest edge that does not form a cycle when added to the growing forest of trees. It maintains a forest of trees initially composed of individual vertices and merges them by adding edges with the least weight until all vertices are connected, forming a single minimum spanning tree. Similar to Prim's algorithm, empirical analysis showcased the scalability of Kruskal's algorithm, with its runtime exhibiting a proportional increase with the graph's size.

Conclusion:

Both Prim's and Kruskal's algorithms provide efficient solutions to the minimum spanning tree problem, each with its own approach and characteristics. While Prim's algorithm starts from a single vertex and grows the tree, Kruskal's algorithm starts with individual vertices and incrementally merges them into a single tree. The empirical analysis presented valuable insights into the runtime performance of these algorithms across varying graph sizes, aiding in understanding their efficiency and scalability.

In conclusion, the study and empirical analysis of Prim's and Kruskal's algorithms underscore their significance in solving optimization problems in diverse fields. Their efficient and scalable nature makes them indispensable tools for addressing graph-related challenges in areas such as network design, clustering analysis, and resource allocation. Further research and experimentation can explore optimizations and adaptations of these algorithms for specific applications, enhancing their utility in real-world scenarios.

References:

[1] GitHub Repository: Algorithms Analysis Laboratory Works, Retrieved from GitHub - Chiuliana/Algorithms_Analysis_Laboratory_Works