Ministry of Education and Research of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Algorithm Analysis

Laboratory work #5

Greedy Algorithms

Author:
Mihaela Catan
std. gr. FAF-231

Verified:
Cristofor Fiștic

Chișinău 2025

Contents

1 Objective

Study and analyze the following Greedy algorithms: Prim's Algorithm and Kruskal's Algorithm.

2 The Task

1. Study the greedy algorithm design technique;
2. Implement in a programming language the algorithms Prim and Kruskal;
3. Perform Empirical Analysis of Prim and Kruskal;
4. Perform empirical analysis of the proposed algorithms;
5. Increase the number of nodes in graph and analyze how this influences the algorithms. Make a graphical presentation of the data obtained;
6. Make a conclusion on the work done.

3 Introduction

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. The algorithm never reverses the earlier decision, even if the choice is wrong. It works in a top-down approach. This algorithm may not produce the best result for all problems. It's because it always goes for the local best choice to produce the global best result.

4 Types of Graph Greedy Algorithms

Some of the most popular Greedy algorithms are:

- **Prim's Algorithm**

  Prim's algorithm finds the Minimum Spanning Tree (MST) in a connected and undirected graph. The MST found by Prim's algorithm is the collection of edges in a graph, that connects all vertices, with a minimum sum of edge weights. Prim's algorithm finds the MST by first including a random vertex to the MST. The algorithm then finds the vertex with the lowest edge weight from the current MST, and includes that to the MST. Prim's algorithm keeps doing this until all nodes are included in the MST. Prim's algorithm is greedy, and has a straightforward way to create a minimum spanning tree.

- **Depth First Search**

  The MST found by Prim's algorithm is the collection of edges in a graph, that connects all vertices, with a minimum sum of edge weights. The MST (or MSTs) found by Kruskal's algorithm is the collection of edges that connect all vertices (or as many as possible) with the minimum total edge weight. Kruskal's algorithm adds edges to the MST (or Minimum Spanning Forest), starting with the edges with the lowest edge weights. Edges that would create a cycle are not added to the MST. These are the red blinking lines in the animation above.

5 Complexity Analysis

In this analysis, the algorithms are evaluated based on their time complexity and space complexity.

- **Time Complexity:** The number of operations performed as a function of the input size n.
- **Space Complexity:** The amount of additional memory required by the algorithm.

6 Technical Implementation

This analysis implements two algorithms and evaluates their time and space complexities on 8 types of graphs: Sparse, Dense, Grid, Cycle, Star, Complete, Barabasi, Watts-Strogatz with sizes: 50, 100, 200, 300, 400, 500. The experiments are conducted on a backend hosted on Google Compute Engine, utilizing Python 3 and approximately 12 GB of RAM. While this environment offers a suitable platform for algorithmic analysis, several limitations and potential sources of error must be acknowledged. These include the possibility that certain algorithms may exhibit suboptimal performance on larger input sizes, as well as inherent inaccuracies in time and memory measurements due to factors such as Python's interpreter overhead and the effects of garbage collection.

6.1 Test Cases

A sparse graph is a type of graph in which the number of edges is significantly less than the maximum number of possible edges. In other words, only a few nodes (or vertices) are connected to each other compared to the total number of connections that could exist.

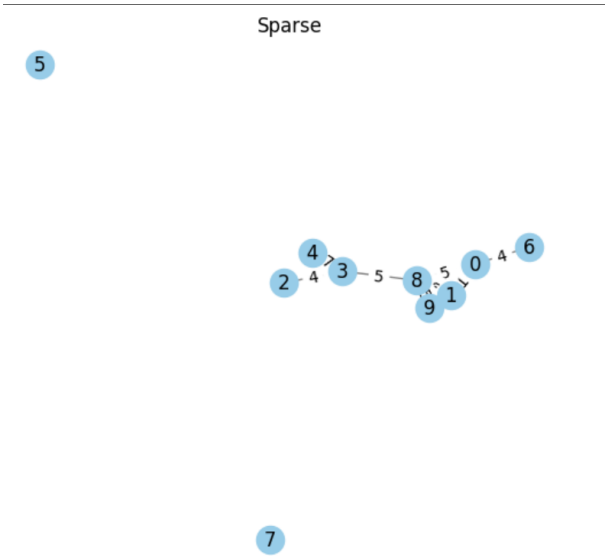

Figure 1: Sparse Graph with 10 nodes

A dense graph is a type of graph where the number of edges is close to the maximum number of possible edges. In simple terms, most of the vertices are connected to each other and leading to a high level of connectivity.

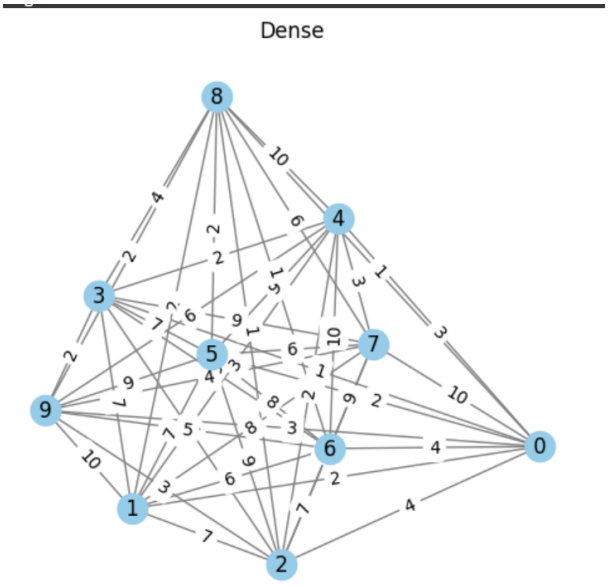

Figure 2: Dense Graph with 10 nodes

A grid graph is a type of graph in mathematics and computer science that represents a grid or lattice structure. It consists of vertices and edges arranged in a rectangular or square grid pattern. Each vertex corresponds to a point in the grid, and edges connect vertices that are adjacent horizontally or vertically. Grid graphs are often used in problems involving pathfinding, optimization, and spatial analysis.

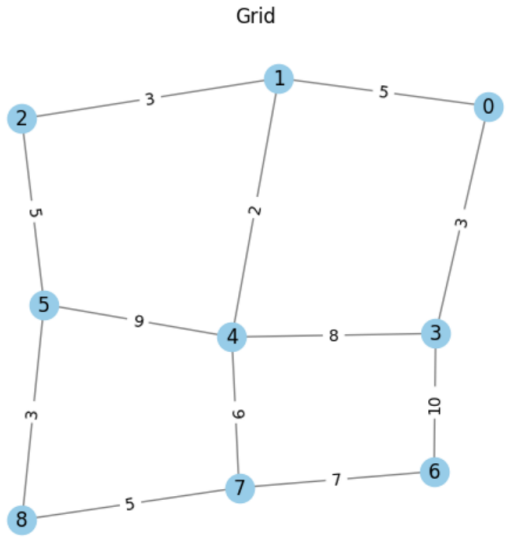

Figure 3: Grid Graph with 10 nodes

A cycle graph is a type of graph in mathematics and graph theory that consists of a single cycle. In simpler terms, it is a closed chain of vertices where each vertex is connected to exactly two other vertices.

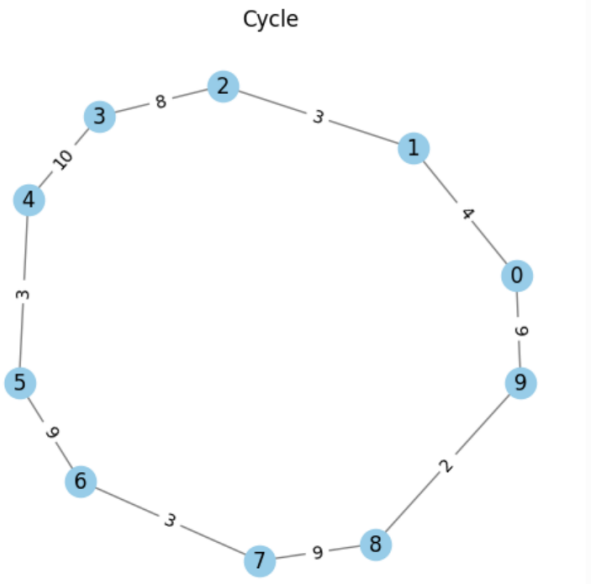

Figure 4: Cycle Graph with 10 nodes

A star graph is a type of graph in mathematics and graph theory that resembles a star shape. It consists of one central vertex connected to several outer vertices, with no connections between the outer vertices themselves.

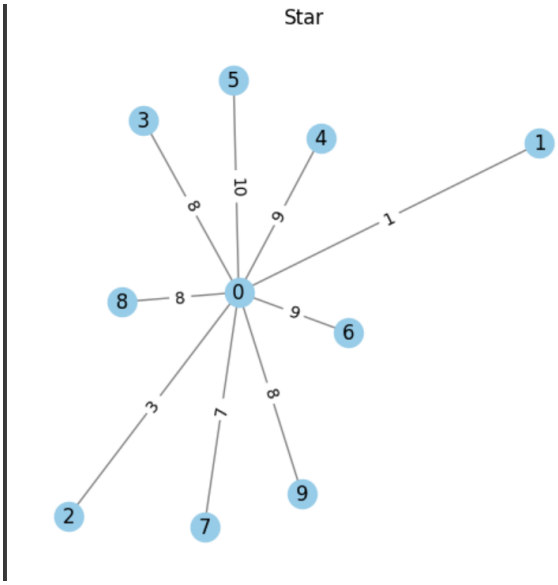

Figure 5: Star Graph with 10 nodes

A complete graph is a type of graph in mathematics and graph theory where every pair of distinct vertices is connected by a unique edge.

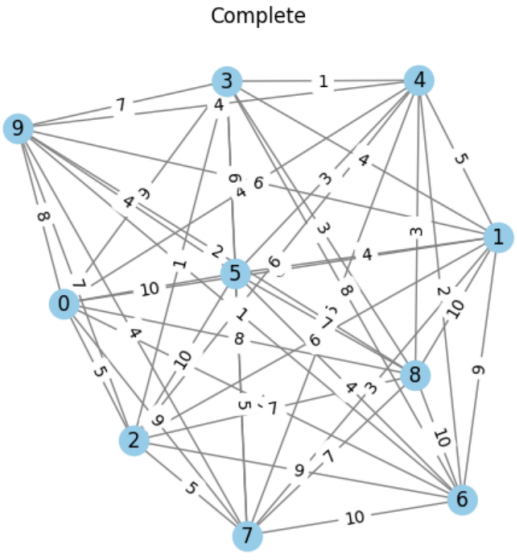

Figure 6: Complete Graph with 10 nodes

A Barabási graph, often referred to as a Barabási–Albert graph, is a type of graph generated using the Barabási–Albert model. This model is used to create scale-free networks, which are networks where some nodes (called hubs) have significantly more connections than others. The graph grows over time, and new nodes preferentially attach to existing nodes with higher degrees (more connections). This mechanism is known as preferential attachment.

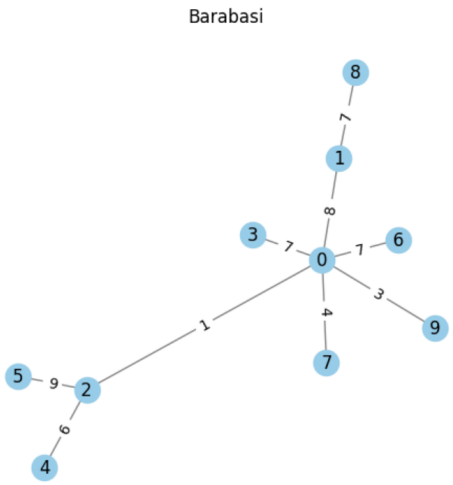

Figure 7: Barabasi Graph with 10 nodes

A Watts-Strogatz graph is a type of graph generated using the Watts-Strogatz model, which creates small-world networks. These networks are characterized by short average path lengths and high clustering, resembling many real-world networks like social networks or neural networks.

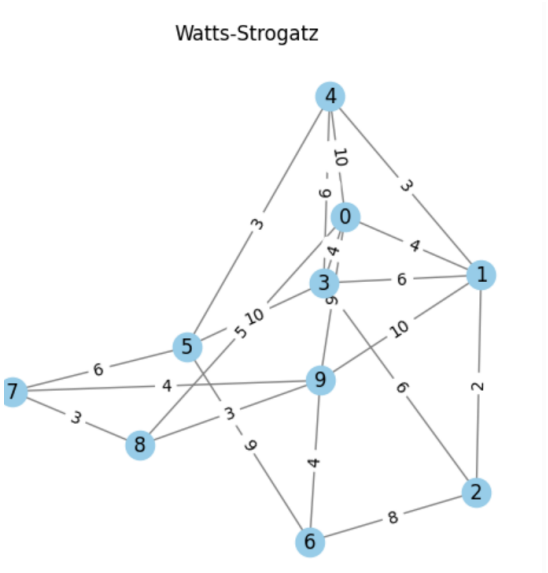

Figure 8: Watts-Strogatz Graph with 10 nodes

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

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.

6.2.1 How it works:

1. Initialize 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.

6.2.2 Python Implementation:

```
def prim_optimized(graph, start):
  visited = set([start])
  edges = []
  mst = []

  for dest, weight in graph[start].items():
    edges.append((weight, start, dest))
  edges.sort()
  while edges:
    weight, src, dest = edges.pop(0)

    if dest in visited:
      continue

    mst.append((src, dest, weight))
    visited.add(dest)

    for next_dest, next_weight in graph[dest].items():
      if next_dest not in visited:
```

```
        edges.append((next_weight, dest, next_dest))

    edges.sort()

  return mst
```

The time complexity of Prim's algorithm is O(V$\hat{2}$).

6.3 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 and has the minimum sum of weights among all the trees that can be formed from the graph

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.

6.3.1 How it works:

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.

6.3.2 Python Implementation:

```
def kruskal_optimized(graph):
  edges = []
  for node in graph:
    for neighbor, weight in graph[node].items():
      edges.append((weight, node, neighbor))

  edges.sort()

  parent = {node: node for node in graph}
  rank = {node: 0 for node in graph}
  mst = []
```

```
  for weight, node1, node2 in edges:
    if find_optimized(parent, node1) != find_optimized(parent
       , node2):
      union_optimized(parent, rank, node1, node2)
      mst.append((node1, node2, weight))

  return mst

def find_optimized(parent, node):
  if parent[node] != node:
    parent[node] = find_optimized(parent, parent[node])
  return parent[node]

def union_optimized(parent, rank, node1, node2):
  root1 = find_optimized(parent, node1)
  root2 = find_optimized(parent, node2)

  if root1 != root2:
    if rank[root1] > rank[root2]:
      parent[root2] = root1
    elif rank[root1] < rank[root2]:
      parent[root1] = root2
    else:
      parent[root2] = root1
      rank[root1] += 1
```

The time complexity Of Kruskal's Algorithm is: O(E log E).

7 Performance Results:

In Sparse graphs, Kruskal outperforms Prim in both execution time and memory usage starting from size 100. Sparse graphs have a low density of edges, which allows Kruskal's edge-centric approach to process efficiently, while Prim's vertex-priority queue handling incurs additional overhead in both time and memory.

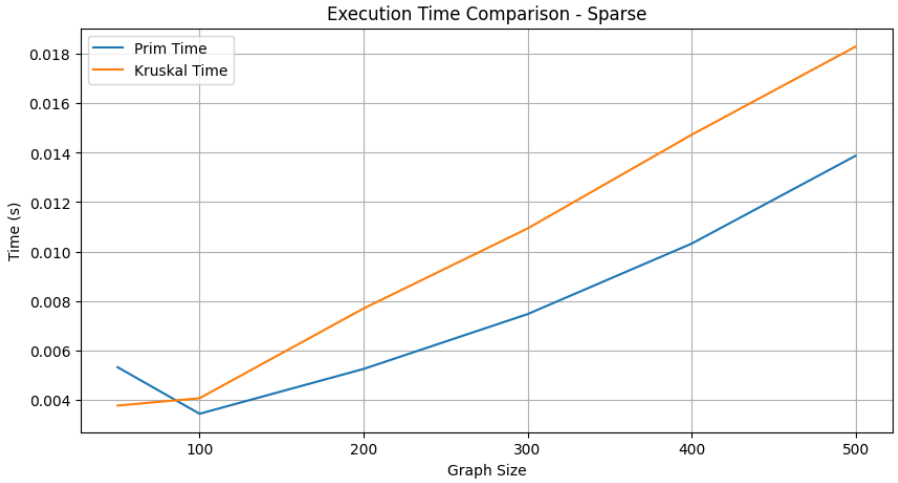

Figure 9: Time Usage on Sparse Graph

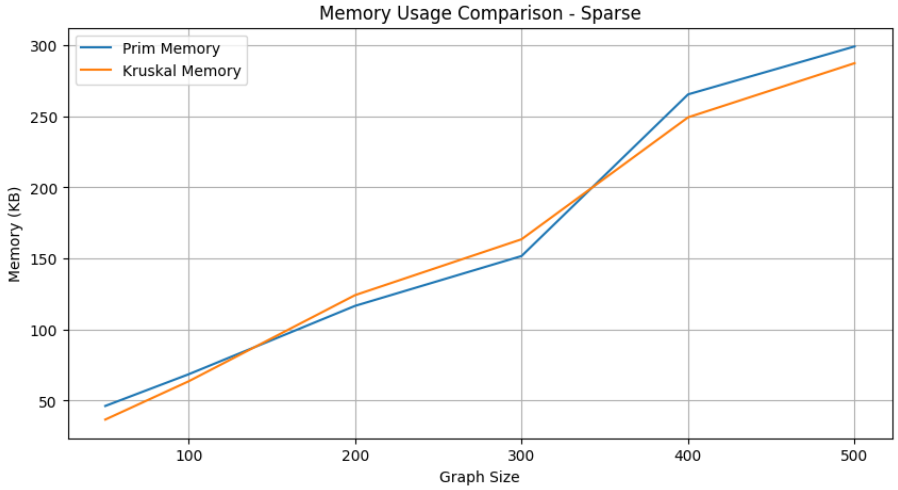

Figure 10: Space Usage on Sparse Graph

In Dense graphs, Kruskal demonstrates better performance in both time and memory from size 50 onward. The dense connectivity of these graphs means that Kruskal's systematic edge-sorting mechanism operates more effectively, whereas Prim struggles with the substantial memory and computational demands required to maintain and update its priority queues.

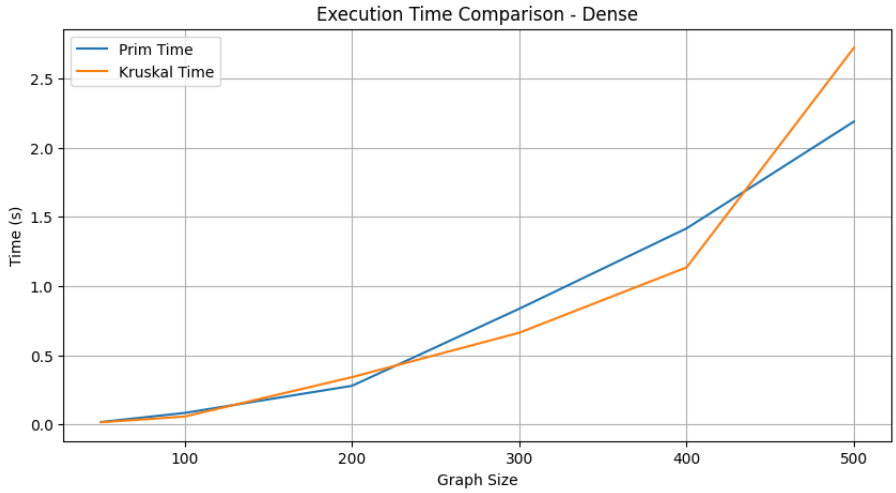

Figure 11: Time Usage on Dense Graph

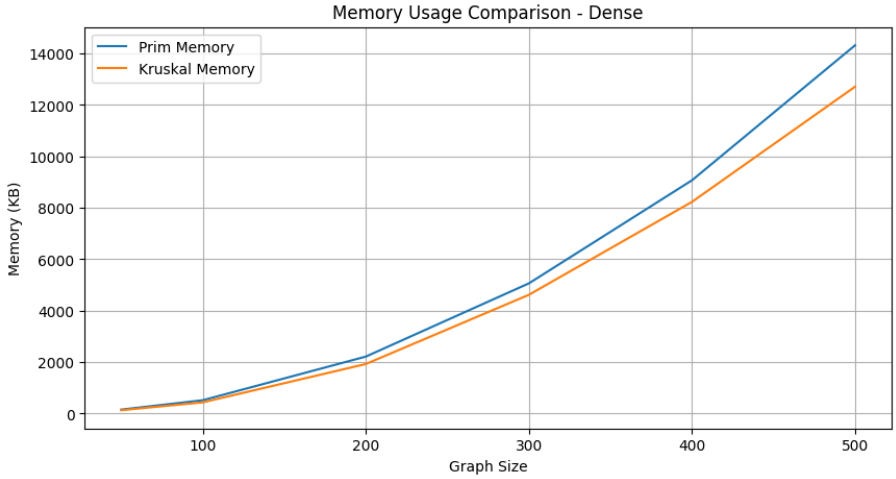

Figure 12: Space Usage on Dense Graph

For Grid graphs, memory efficiency is where Kruskal starts to surpass Prim from size 300. Grid structures, with their evenly distributed connections, are managed more simply through Kruskal's edge-focused methodology. However, in terms of time performance, the difference between the algorithms is less pronounced.

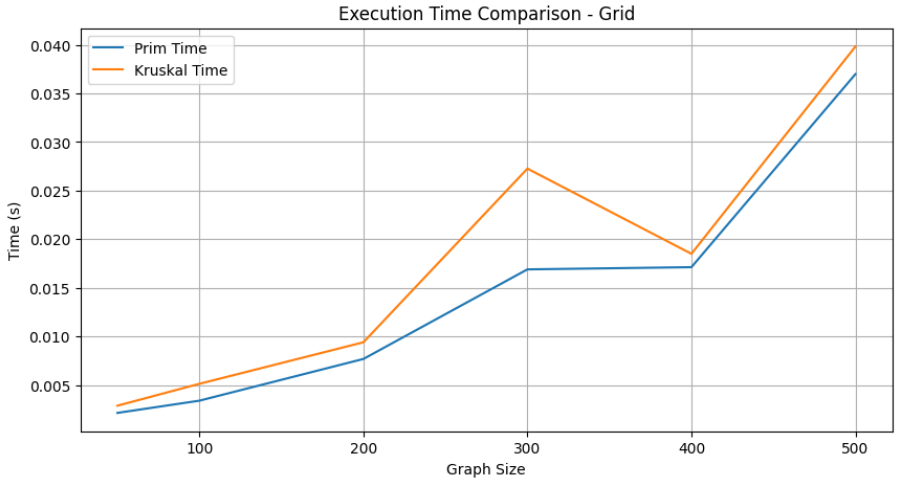

Figure 13: Time Usage on Grid Graph

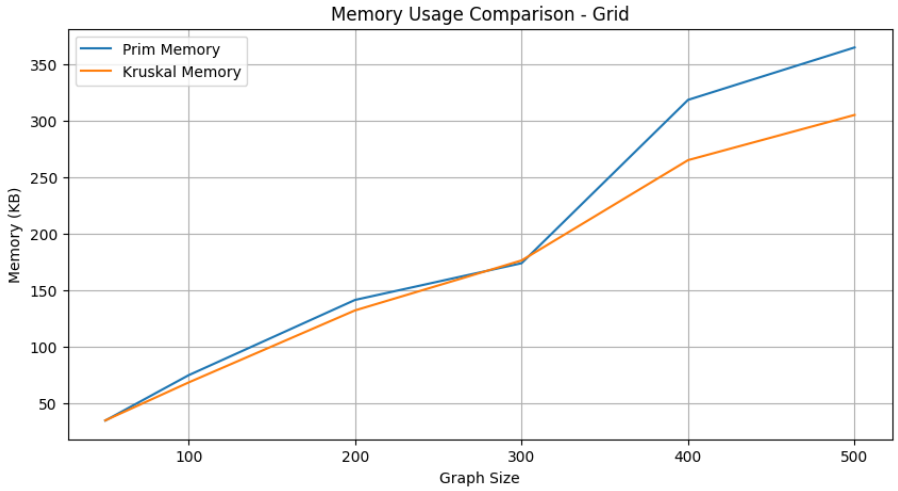

Figure 14: Space Usage on Grid Graph

In Cycle graphs, Kruskal outperforms Prim in memory usage starting from size 300. Cycle graphs, with their repetitive and linear structure, allow Kruskal to maintain efficient edge-based processing. Prim, on the other hand, faces higher memory demands due to its vertex-priority mechanism, which is less effective in such simple layouts.

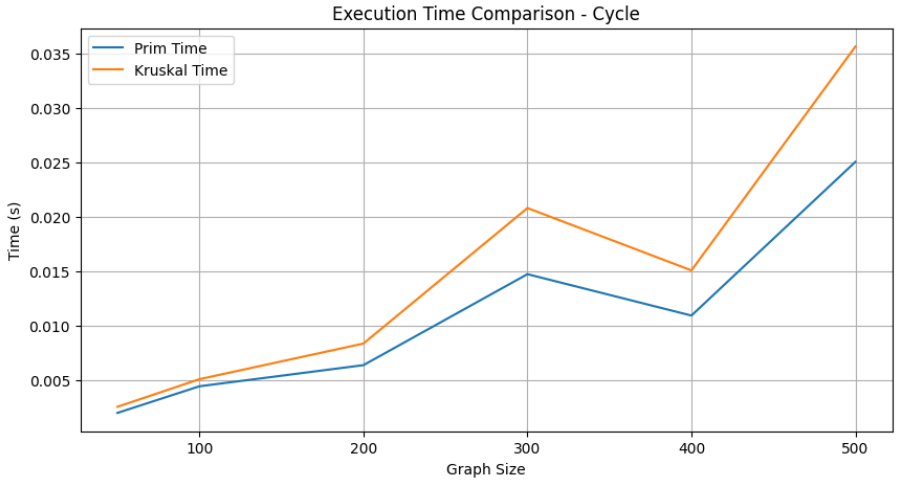

Figure 15: Time Usage on Cycle Graph

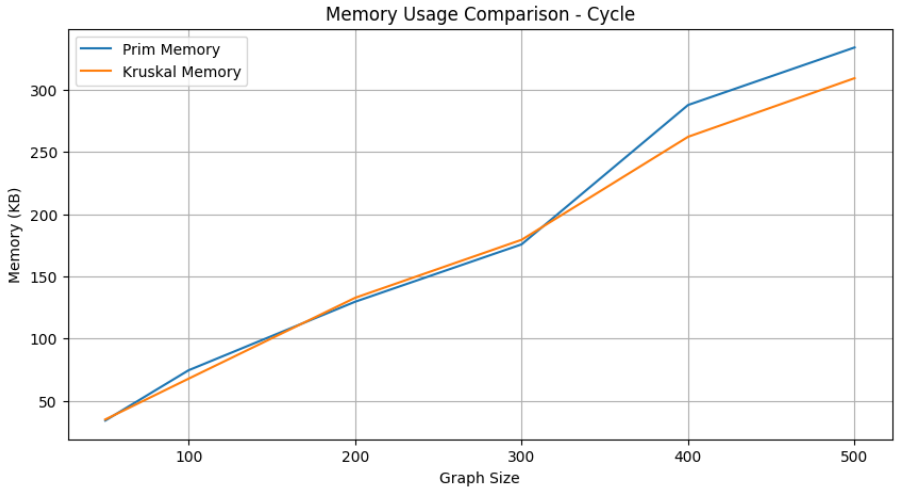

Figure 16: Space Usage on Cycle Graph

Star graphs reveal Kruskal's memory advantage over Prim beginning at size 50. The centralized hub-and-spoke structure of star graphs is handled effectively by Kruskal's edge-processing approach, avoiding the depth-first recursions that make Prim's memory usage more intensive. However, the time performance difference between the two algorithms remains less significant.

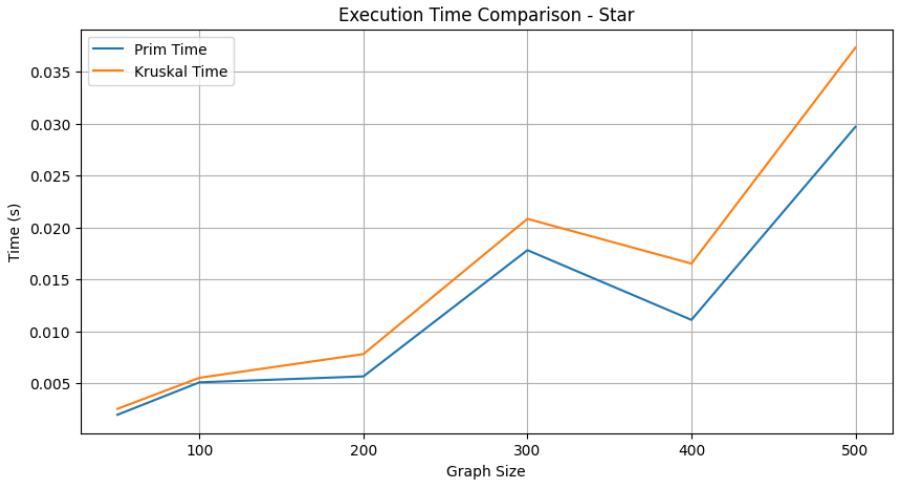

Figure 17: Time Usage on Star Graph

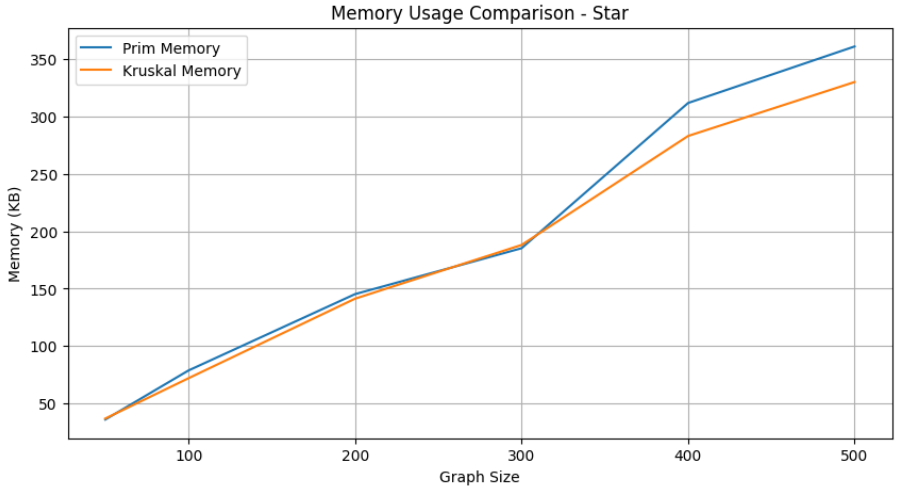

Figure 18: Space Usage on Star Graph

Complete graphs highlight Kruskal's superiority in both time and memory, with Kruskal surpassing Prim in memory usage starting from size 50 and in execution time from size 100 onward. The exhaustive edge connectivity of complete graphs makes Kruskal's edge-centric sorting approach more efficient, while Prim's vertex-priority operations become a significant bottleneck.

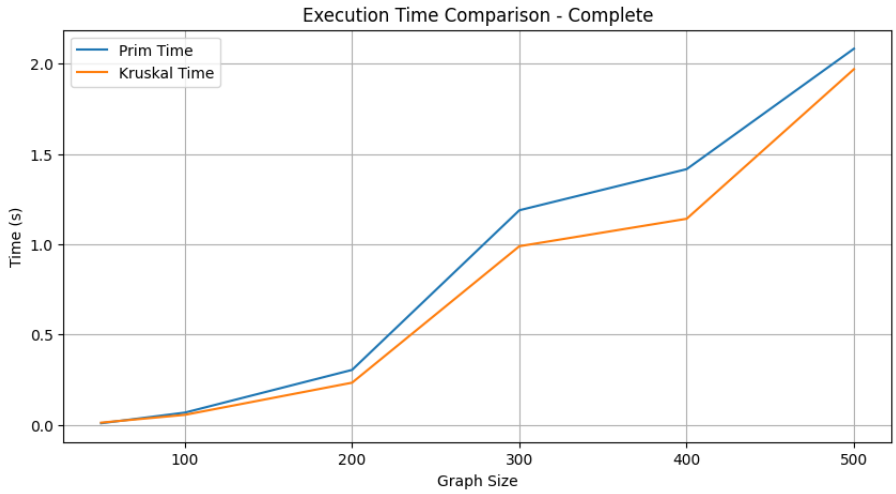

Figure 19: Time Usage on Complete Graph

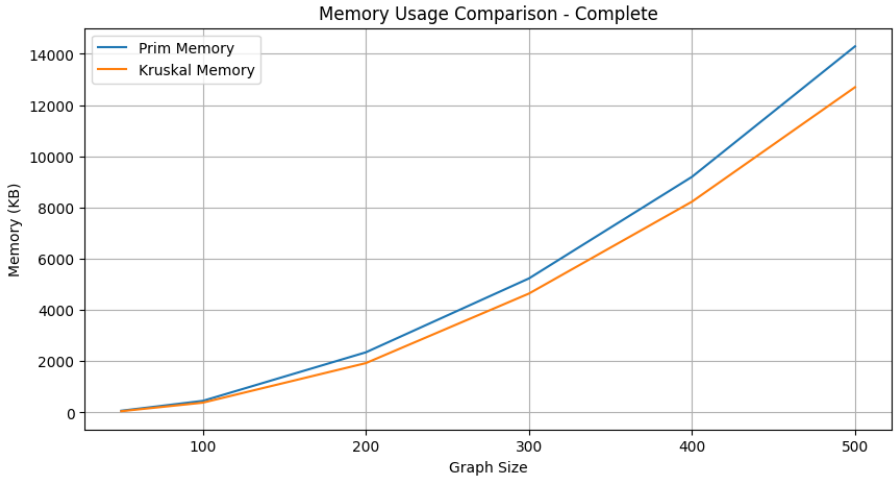

Figure 20: Space Usage on Complete Graph

In Barabási graphs, Kruskal shows a time performance advantage starting from size 300, while memory efficiency becomes apparent at size 100. The hub-dominant structure characteristic of Barabási graphs enables Kruskal to handle the highly connected nodes efficiently, whereas Prim's memory usage grows due to its less tailored approach for handling such topologies.

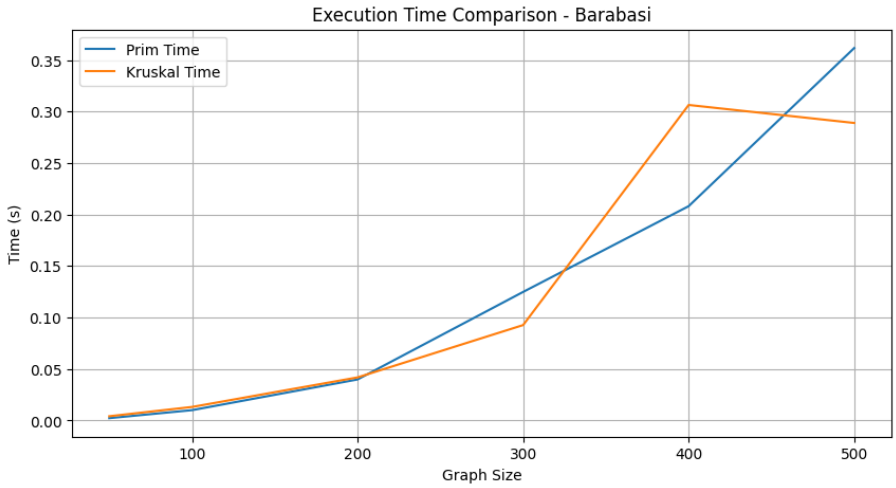

Figure 21: Time Usage on Barabasi Graph

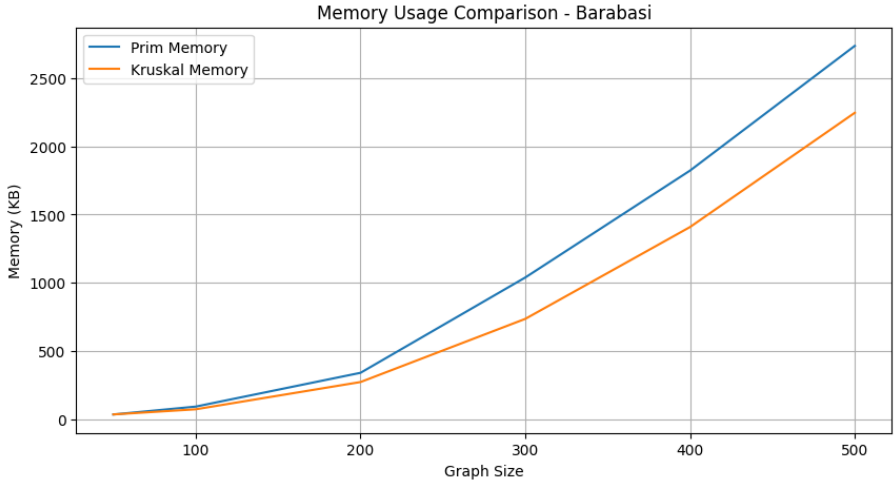

Figure 22: Space Usage on Barabasi Graph

Watts-Strogatz graphs demonstrate Kruskal's memory efficiency beginning at size 100. The small-world properties, with short average path lengths and local clustering, are better suited to Kruskal's edge-focused traversal. Prim, constrained by its vertex-priority methodology, incurs higher memory demands for maintaining complex updates in such structures.

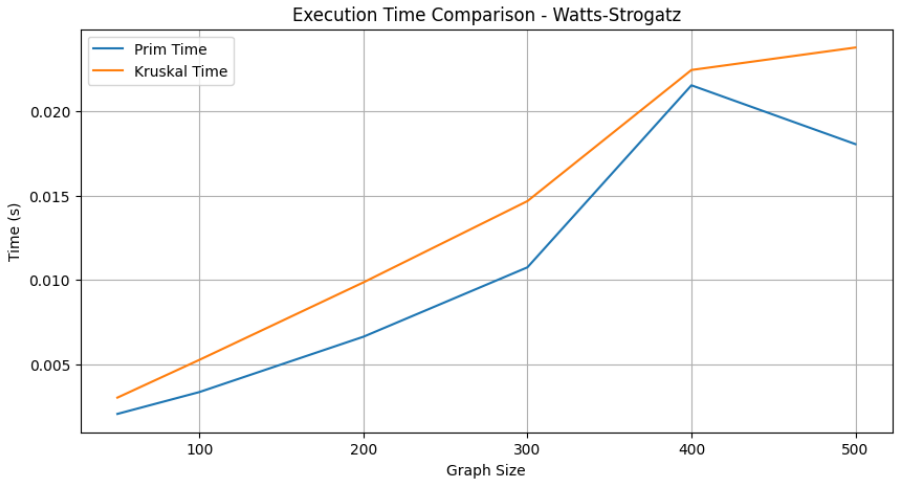

Figure 23: Time Usage on Watts-Strogatz Graph

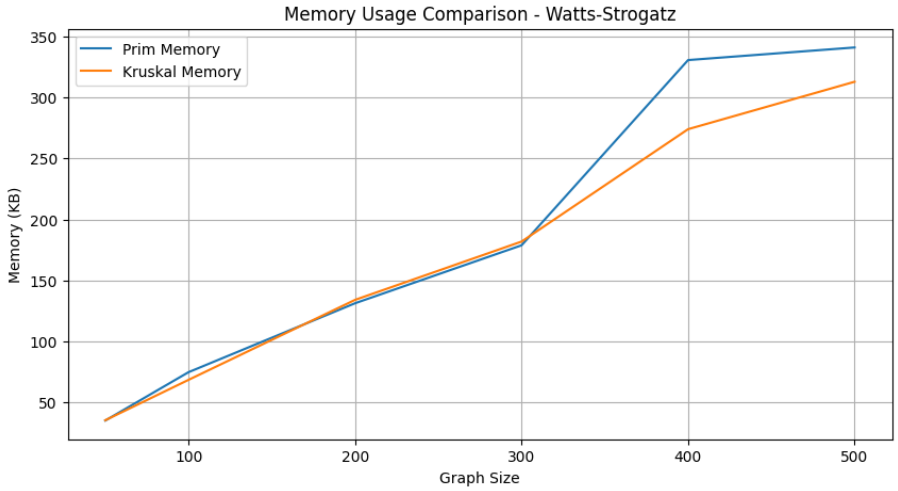

Figure 24: Space Usage on Watts-Strogatz Graph

Conclusion

Kruskal consistently outperforms Prim in both execution time and memory usage across graph types, particularly as graph size increases. This highlights Kruskal's efficiency in handling edge-heavy or complex graph structures, while Prim tends to struggle with higher memory demands due to its vertex-priority queue mechanisms. This study underscores the importance of understanding graph topology and algorithm characteristics when selecting the optimal Minimum Spanning Tree algorithm.

8 Bibliography

- The code is available at: Google Colab Notebook
- GitHub Repository: GitHub