Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Algorithm Analysis

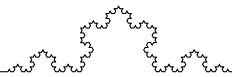
Laboratory work #5

Greedy Algorithms

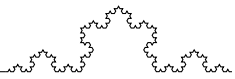
Author: Timur Cravtov
std. gr. FAF-231

Verified by: Cristofor Fistic
asist. univ

Chișinău
2025



Contents



1 Algorithm Analysis

1.1 Objective

Study and analyze different algorithms for building minimum spanning trees. The analysis should be performed on the basis of the following algorithms:

1. Prim's algorithm
2. Kruskal's algorithm

1.2 Task

1. Study the greedy algorithm design technique
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms;
4. Perform empirical analysis of the proposed algorithms;
5. Make a graphical presentation of the data obtained;
6. Make a conclusion on the work done.

1.3 Introduction

Greedy algorithms are a class of algorithms that make a series of choices, each of which looks best at the moment. The idea is to make a locally optimal choice at each step with the hope that these choices will lead to a globally optimal solution. Greedy algorithms are often used for optimization problems, where the goal is to find the best solution among many possible solutions.

One of the application of Greedy algorithm is building a minimum spanning tree (MST) of a weighted undirected graph. A minimum spanning tree is a subset of the edges of the graph that connects all the vertices together without any cycles and with the minimum possible total edge weight. In this laboratory work, we will analyze two greedy algorithms for building a minimum spanning tree: Prim's algorithm and Kruskal's algorithm.

2 Implementation

2.1 Prim's algorithm

Algorithm description

Prims algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. It works by starting with a single vertex and adding edges to the tree until all vertices are included. The algorithm maintains a priority queue of edges, and at each step, it adds the edge with the smallest weight that connects a vertex in the tree to a vertex outside the tree.

```python
import networkx as nx
import heapq


def min_span_tree_prim(G: nx.Graph) -> nx.Graph:
    if not G.nodes:
        return nx.Graph()

    mst = nx.Graph()
    visited = set()
    min_heap = []

    # Start from an arbitrary node
    start_node = list(G.nodes)[0]
    visited.add(start_node)

    # Push all edges from the start node into the heap
    for neighbor in G.neighbors(start_node):
        weight = G[start_node][neighbor]['weight']
        heapq.heappush(min_heap, (weight, start_node, neighbor))

    while min_heap and len(visited) < G.number_of_nodes():
        weight, u, v = heapq.heappop(min_heap)

        if v not in visited:
            visited.add(v)
            mst.add_edge(u, v, weight=weight)

            for neighbor in G.neighbors(v):
                if neighbor not in visited:
                    heapq.heappush(min_heap, (G[v][neighbor]['
                        weight'], v, neighbor))

```

```
33      return mst
```

Prim algorithm

2.2 Kruskal's algorithm

Algorithm description

Kruskal algorithm works by sorting all the edges in the graph by their weights and adding them to the minimum spanning tree one by one, as long as they do not form a cycle. The algorithm uses a disjoint-set data structure to keep track of which vertices are in which components.

```
1  import networkx as nx
2
3
4  def min_span_tree_kruskal(G: nx.Graph) -> nx.Graph:
5      if not G.nodes:
6          return nx.Graph()
7
8      # Create a list of edges with weights
9      edges = sorted(G.edges(data=True), key=lambda x: x[2]['
           weight'])
10
11     parent = {node: node for node in G.nodes}
12     rank = {node: 0 for node in G.nodes}
13
14     def find(v):
15         if parent[v] != v:
16             parent[v] = find(parent[v])  # Path compression
17         return parent[v]
18
19     def union(u, v):
20         root_u = find(u)
21         root_v = find(v)
22         if root_u != root_v:
23             if rank[root_u] > rank[root_v]:
24                 parent[root_v] = root_u
25             elif rank[root_u] < rank[root_v]:
26                 parent[root_u] = root_v
27             else:
```

```
                parent[root_v] = root_u
                rank[root_u] += 1
            return True
        return False

    mst = nx.Graph()

    for u, v, data in edges:
        if union(u, v):
            mst.add_edge(u, v, weight=data['weight'])

    return mst
```

Kruskal's algorithm

2.3 Algorithms execution

For verifying the correctness of the algorithms, we will use a simple random generated graph.

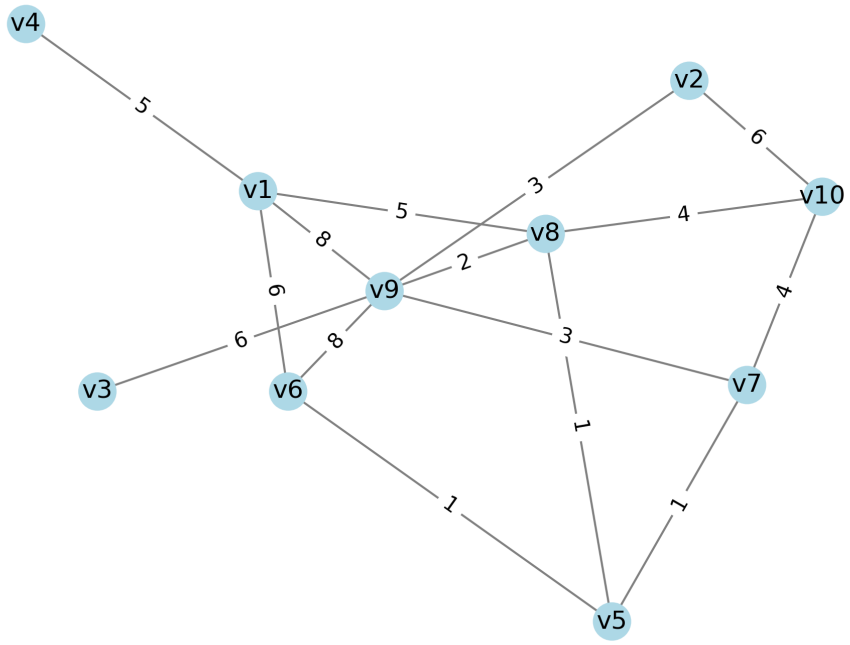


Figure 1: Random graph

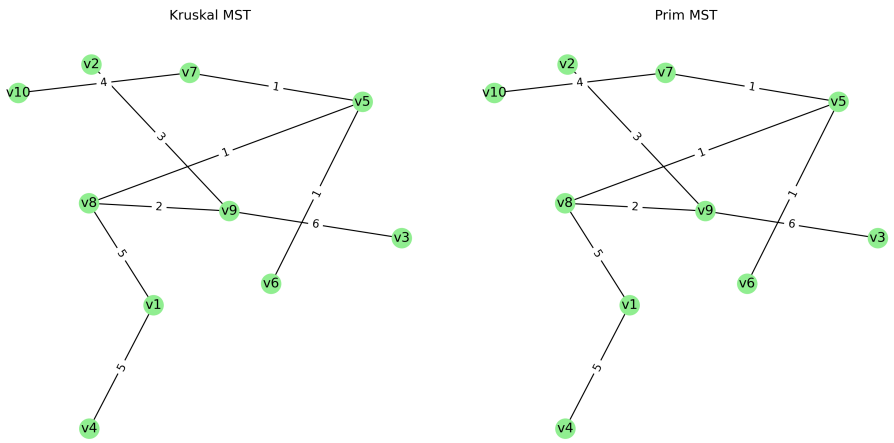


Figure 2: Graph of the minimum spanning tree

As one can see, the minimum spanning trees produced by both algorithms are the same. This is because both algorithms are designed to find the minimum spanning tree of a graph, and they both use different approaches to achieve the same result.

The minimum spanning tree is a subset of the edges of the graph that connects all the vertices together without any cycles and with the minimum possible total edge weight. In this case, both algorithms produced the same minimum spanning tree, which is expected since they are both designed to find the same result despite their different approaches.

2.4 Comparative analysis

For the comparative analysis of the algorithms, we will use the following metrics:

- Time complexity: The time complexity of an algorithm is a measure of the amount of time it takes to run as a function of the size of the input. We will analyze the time complexity of both Prim's and Kruskal's algorithms.
- Number of edges: The number of edges in the minimum spanning tree is another important metric. We will compare the number of edges in the minimum spanning trees produced by both algorithms.

The figure 3 shows the performance of the algorithms in terms of time complexity and number of edges. There was considered 4 versions of the graph: dense graph, graph with fixed vertices and sparse graphs.

The time complexity of Prim's algorithm is $O(ElogV)$, where E is the number of edges and V is the number of vertices. The time complexity of Kruskal's algorithm is $O(ElogE)$, which can be simplified to $O(ElogV)$ since E is at most V^2.

From graphs below, we can see that the performance of both algorithms is similar for small graphs. However, as the size of the graph increases, Kruskal's algorithm tends

to perform worse than Prim's algorithm. This is because Kruskal's algorithm requires sorting the edges, which can be expensive for large graphs. In contrast, Prim's algorithm uses a priority queue to maintain the edges, which allows it to add edges more efficiently.

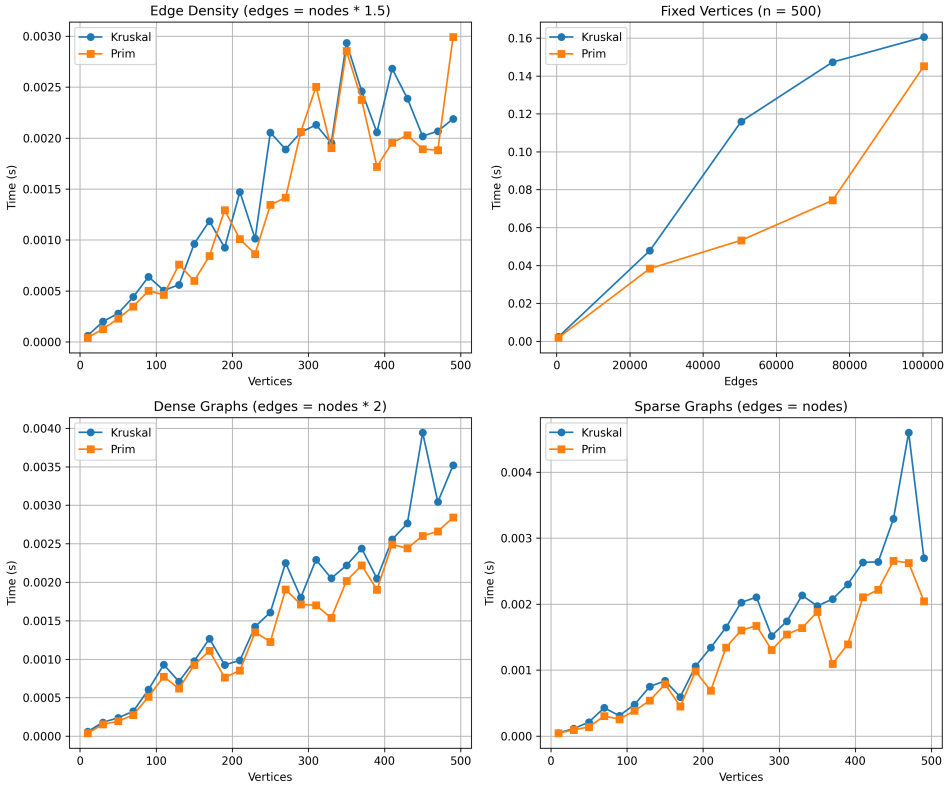
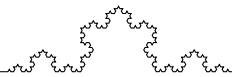


Figure 3: Graph of the minimum spanning tree

3 Conclusions

During this laboratory work, we studied and analyzed two greedy algorithms for building a minimum spanning tree: Prim's algorithm and Kruskal's algorithm. We implemented both algorithms in Python and verified their correctness using a simple random generated graph. We performed a comparative analysis of the algorithms based on their time complexity and the number of edges in the minimum spanning tree. We found that both algorithms produced the same minimum spanning tree, which is expected since they are both designed to find the same result. However, we also found that Prim's algorithm tends to perform slightly better than Kruskal's algorithm for larger graphs. This is because Prim's algorithm uses a priority queue to maintain the edges, which allows it to add edges more efficiently. In contrast, Kruskal's algorithm requires sorting the edges, which can be expensive for large graphs.

Overall, we concluded that both algorithms are effective for building a minimum spanning tree, but Prim's algorithm is generally more efficient for larger graphs. This



laboratory work helped us to understand the greedy algorithm design technique and its application in building minimum spanning trees.

References

[1] https://github.com/TimurCravtov/AA-labs