MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Algorithm Analysis

Laboratory work 7: Greedy Algorithms

Elaborated:
st.gr. FAF-211 Coreţchi Mihai
Verified:
asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

Greedy algorithms are a class of problem-solving approaches that prioritize making locally optimal decisions at each step in the hopes of finding the best overall solution. These algorithms are widely used in optimization problems and are characterized by their focus on immediate gains rather than long-term consequences.The main advantage of greedy algorithms is their computational efficiency. Unlike more complex algorithms that require extensive computations, greedy algorithms often have a linear or near-linear runtime complexity. This makes them well-suited for solving problems with large input sizes, where speed and scalability are important. However, it's important to note that greedy algorithms do not always guarantee optimal solutions. Their myopic nature may lead to overlooking certain possibilities or making suboptimal choices. The correctness of a greedy algorithm depends on the specific problem and its characteristics. In this introduction, we will explore various aspects of greedy algorithms, including their underlying principles, common applications, and limitations. We will examine the design strategies used by greedy algorithms, such as the greedy choice property and optimal substructure property. Additionally, we will delve into examples of well-known greedy algorithms and discuss techniques to evaluate and analyze their efficiency. By understanding the fundamentals of greedy algorithms, you will acquire a valuable set of tools to tackle optimization problems in various fields, including computer science, operations research, and economics. Let's embark on this journey to uncover the strengths and limitations of greedy algorithms and discover how they can assist us in finding near-optimal solutions in real-world scenarios.

Objectives

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. Make a graphical presentation of the data obtained.
5. To make a report.

Prim's algorithm

Prim's algorithm is a greedy algorithm that aims to find the minimum spanning tree (MST) of a connected weighted graph. The MST is a tree that spans all the vertices of the graph while minimizing the total weight of its edges. The algorithm begins by selecting an arbitrary vertex as the starting point. It then gradually grows the MST by iteratively adding the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST. At each step, the algorithm maintains two sets: the MST set, which initially contains only the starting vertex, and the boundary set, which contains the vertices not yet included in the MST. The algorithm also uses a priority queue to keep track of the candidate edges. Starting from the initial vertex, the algorithm considers all the edges connected to it and adds them to the priority queue. The edges in the priority queue are sorted based on their weights, with the minimum weight edge at the front. In each iteration, the algorithm extracts the edge with the minimum weight from the priority queue. If the extracted edge connects a vertex already in the MST to a vertex outside the MST, it is discarded since adding it would create a cycle. However, if the edge connects a vertex in the MST to a vertex in the boundary set, it is added to the MST, and the newly added vertex is included in the MST set. The algorithm also adds all the edges connected to the newly added vertex to the priority queue. The process continues until all vertices are included in the MST, i.e., the boundary set becomes empty. At this point, the algorithm terminates, and the resulting MST is the optimal tree with the minimum total weight. Prim's algorithm guarantees that the resulting MST is optimal, meaning it has the minimum total weight among all possible spanning trees of the graph. It achieves this optimality by greedily selecting the edge with the minimum weight at each step. The runtime complexity of Prim's algorithm is typically $O(V^2)$ for adjacency matrix representation or O(E log V) for adjacency list representation, where V is the number of vertices and E is the number of edges in the graph. Prim's algorithm is commonly used in various applications, such as network design, clustering, and image segmentation, where finding the optimal tree structure is essential.

Code:

```
Prim(G):
   Select an arbitrary starting vertex s
   Initialize an empty set MST to store the minimum spanning tree
   Initialize a priority queue PQ to store candidate edges
   Mark all vertices as not visited

   Add s to MST
   Mark s as visited
```

```
For each edge e connected to s:
   Add e to PQ with its weight as the priority

While PQ is not empty:
   Extract the edge with the minimum weight from PQ as (u, v)

   If v is not visited:
      Add (u, v) to MST
      Add v to MST
      Mark v as visited

      For each edge e connected to v:
         If the neighboring vertex w of v is not visited:
            Add e to PQ with its weight as the priority

Return MST
```

Kruskal's algorithm

Kruskal's algorithm is a greedy algorithm that aims to find the minimum spanning tree (MST) of a connected weighted graph. The MST is a tree that spans all the vertices of the graph while minimizing the total weight of its edges. The algorithm begins by sorting all the edges of the graph in non-decreasing order of their weights. This sorting step ensures that we process the edges in ascending order of their weights. Next, the algorithm initializes an empty set called MST to store the minimum spanning tree. The MST set starts empty and will gradually grow as we add edges to it. The algorithm then iterates through the sorted edges, considering them one by one. For each edge, the algorithm checks if adding it to the MST will create a cycle. This check ensures that we do not include edges that would result in a cycle in the final MST. To perform the cycle check, Kruskal's algorithm employs a disjoint set data structure or a union-find data structure. This data structure keeps track of the connected components of the vertices. Initially, each vertex is in its own disjoint set. When we consider an edge, we check if the endpoints of the edge belong to the same connected component (disjoint set) or not. If they do not, adding the edge to the MST will not create a cycle, and we add it to the MST. Otherwise, we skip the edge and move on to the next one. The algorithm continues this process until it has processed all the edges or until the MST contains V-1 edges, where V is the number of vertices in the graph. Once this condition is met, we stop the algorithm. At the end of the

algorithm, the MST set will contain the edges that form the minimum spanning tree of the graph. Kruskal's algorithm guarantees that the resulting MST is optimal, meaning it has the minimum total weight among all possible spanning trees of the graph. It achieves this optimality by greedily selecting edges with the smallest weights that do not create cycles. The runtime complexity of Kruskal's algorithm is typically O(E log E) or O(E log V), where E is the number of edges and V is the number of vertices in the graph. Kruskal's algorithm is commonly used in various applications, such as network design, transportation planning, and circuit design, where finding the optimal tree structure is essential.

Code:

```
Kruskal(G):
   Sort all edges of G in non-decreasing order of their weights
   Initialize an empty set MST to store the minimum spanning tree

   For each vertex v in G:
      Create a disjoint set with v as the representative

   For each edge (u, v) in the sorted edges:
      If the vertices u and v are in different disjoint sets:
         Add the edge (u, v) to MST
         Merge the disjoint sets of u and v

   Return MST
```

Implementation

```
def kruskal(graph):
    nodes = list(graph.nodes)
    edges = list(graph.edges.data("weight", default=1))
    edges.sort(key=lambda x: x[2])  # sort edges by weight
    tree = nx.Graph()
    tree.add_nodes_from(nodes)
    for u, v, w in edges:
        if nx.algorithms.tree.recognition.is_forest(tree):
            tree.add_edge(u, v, weight=w)
            if not nx.is_connected(tree):
                tree.remove_edge(u, v)
    return tree



def prim(graph):
    nodes = list(graph.nodes)
    tree = nx.Graph()
    tree.add_node(nodes[0])
    while set(tree.nodes) != set(nodes):
        fringe = [(u, v, d) for u, v, d in
            graph.edges(tree.nodes, data='weight') if v not in tree.nodes]
        if not fringe:
            print("Graph is not connected. Can't apply Prim's algorithm.")
            return
        u, v, w = min(fringe, key=lambda x: x[2])
        tree.add_edge(u, v, weight=w)
    return tree
```

Screenshot:

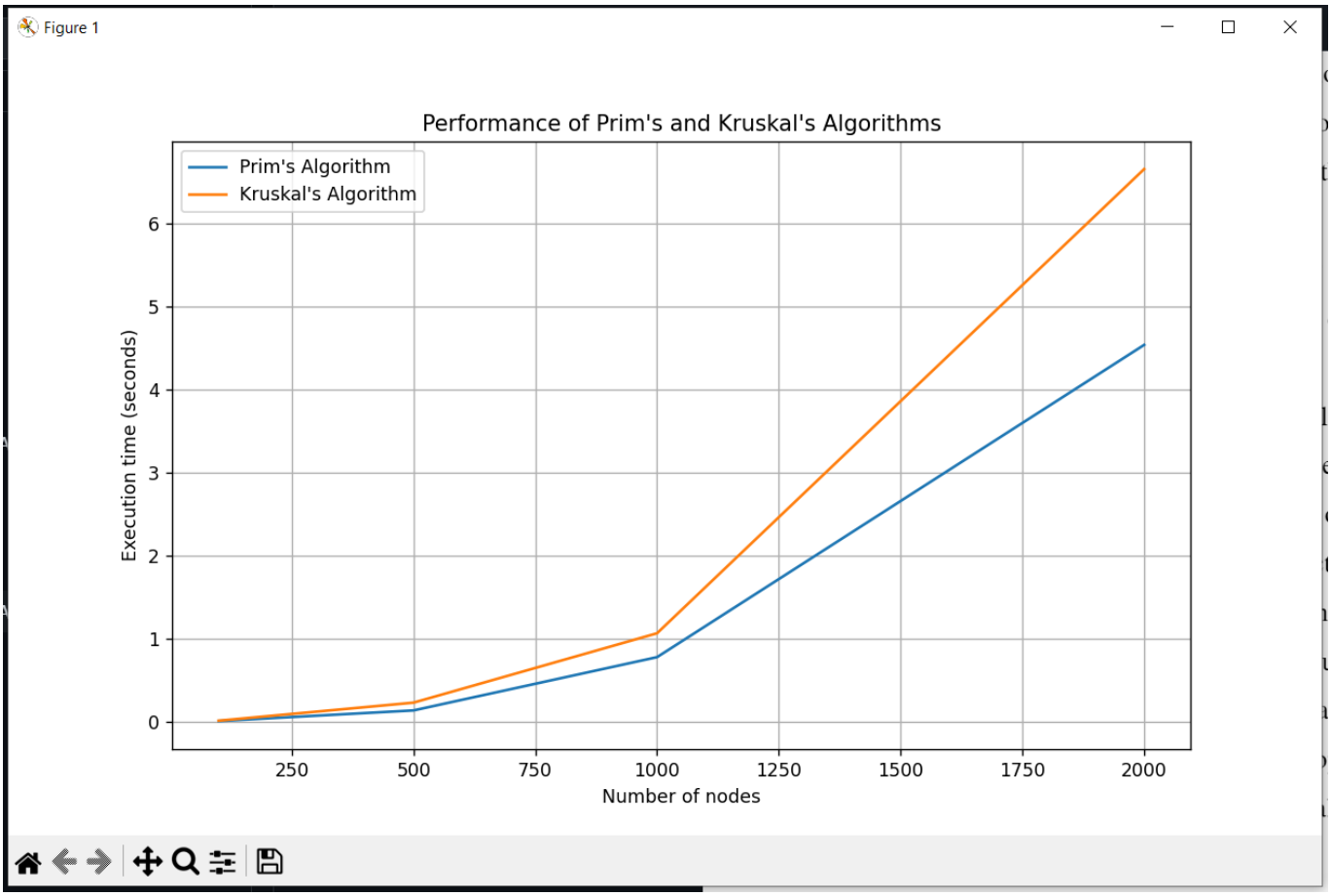
Figure 1
Performance of Prim's and Kruskal's Algorithms
Prim's Algorithm
Kruskal's Algorithm
Execution time (seconds)
0
1
2
3
4
5
6
250
500
750
1000
1250
1500
1750
2000
Number of nodes

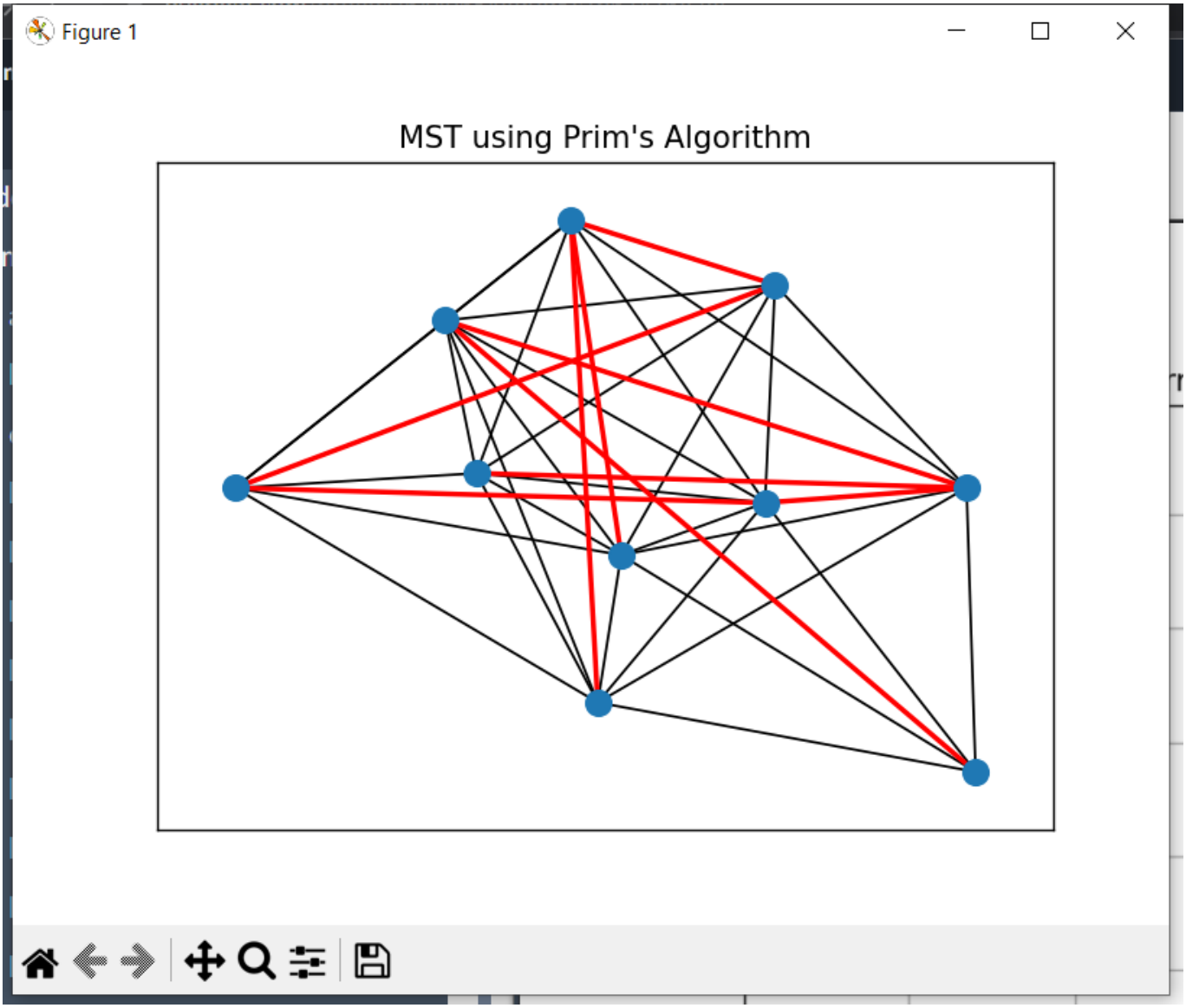
Figure 1
MST using Prim's Algorithm

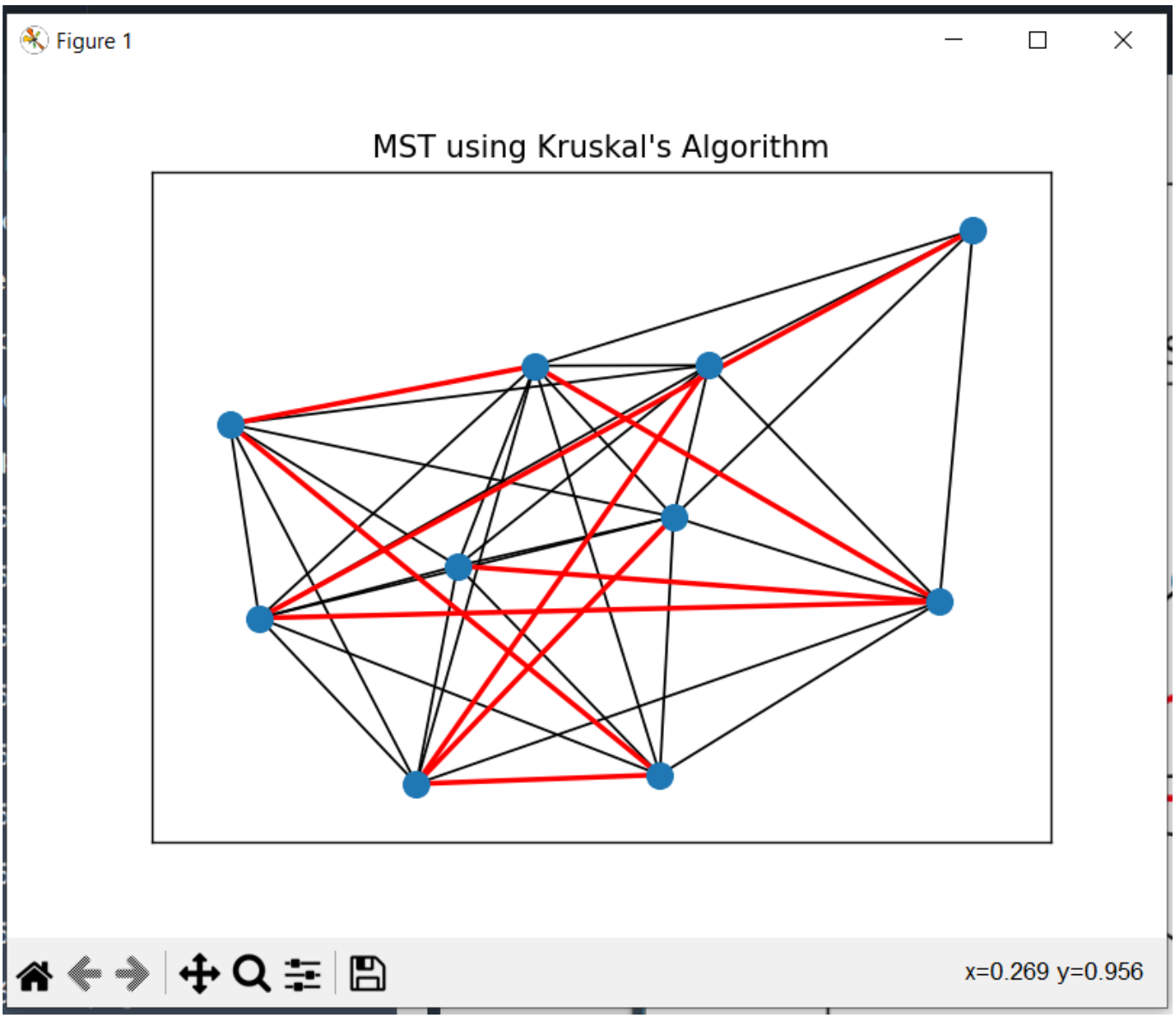
Figure 1
MST using Kruskal's Algorithm
x=0.269 y=0.956

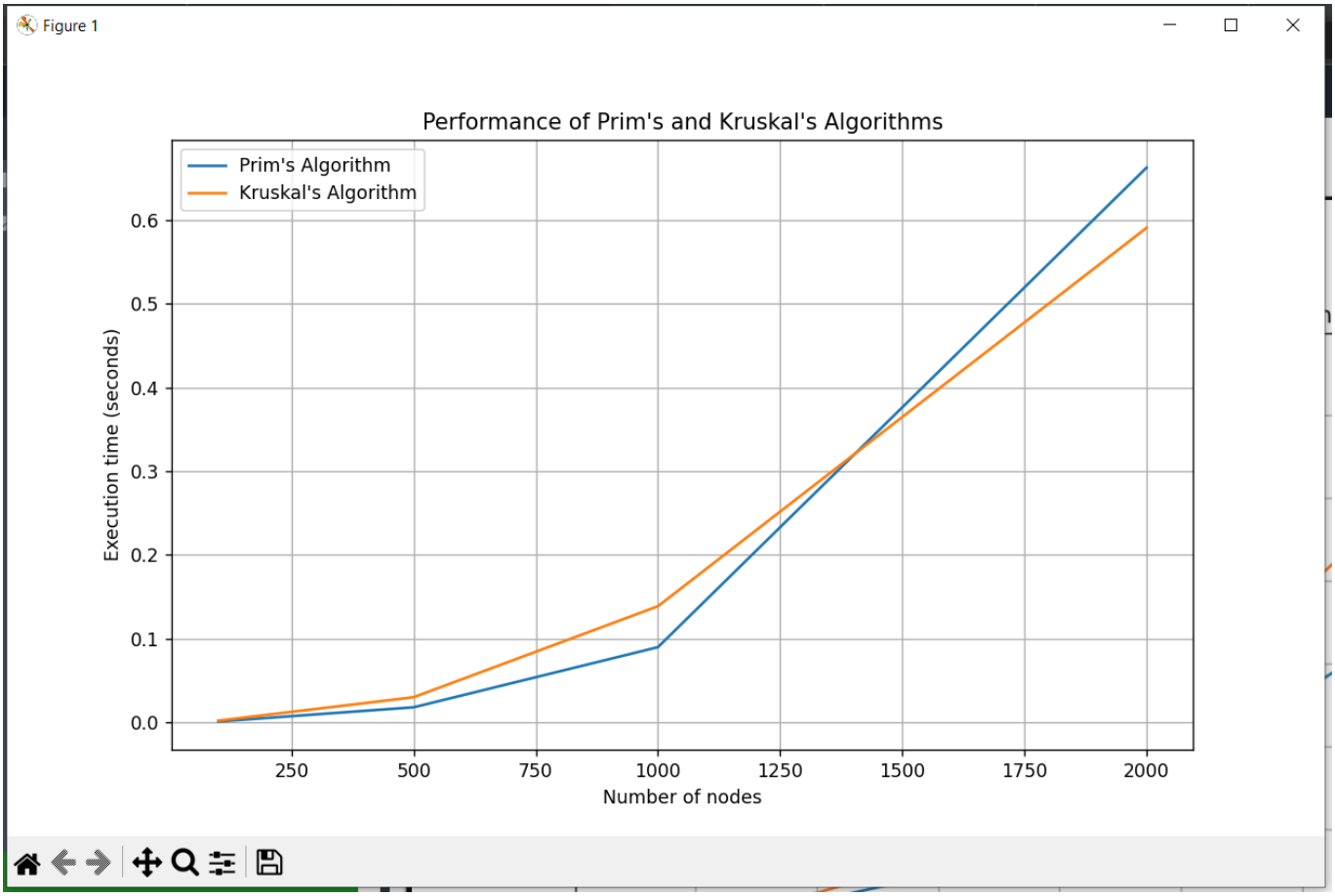
Figure 1
Performance of Prim's and Kruskal's Algorithms
Prim's Algorithm
Kruskal's Algorithm
Execution time (seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
250
500
750
1000
1250
1500
1750
2000
Number of nodes

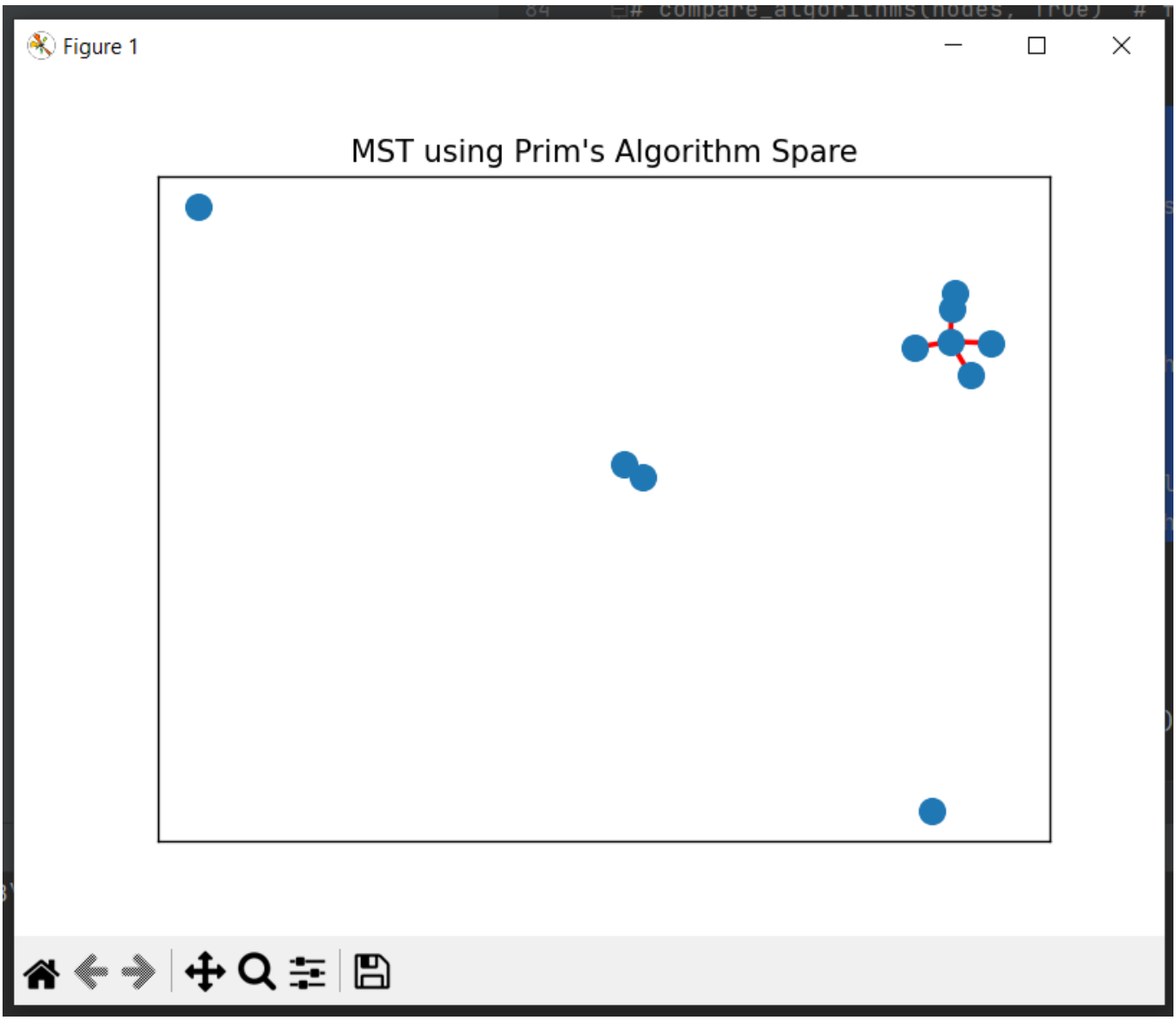
Figure 1
MST using Prim's Algorithm Spare

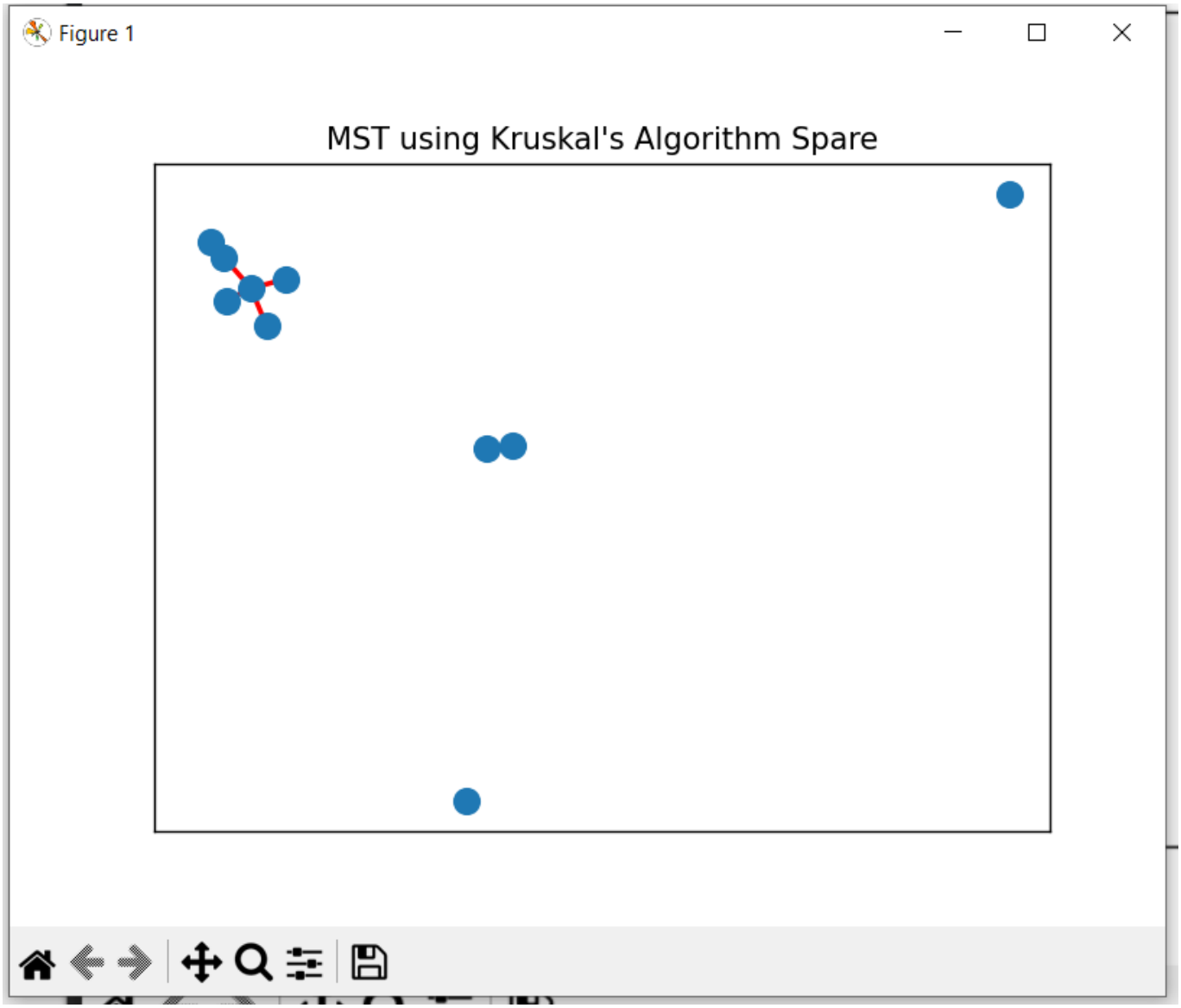
Figure 1
MST using Kruskal's Algorithm Spare

Conclusion

In this laboratory work, I have examined and compared the performance of Prim's and Kruskal's algorithms for finding the minimum spanning tree (MST) of a graph. These two algorithms provide different strategies for constructing the MST and offer distinct advantages and considerations. Prim's algorithm is a greedy approach that starts with an arbitrary vertex and incrementally grows the MST by adding the edge with the minimum weight that connects a vertex in the MST to a vertex outside. This algorithm guarantees an optimal solution and has a runtime complexity of $O(V^2)$ for adjacency matrix representation or $O(ElogV)$ for adjacency list representation, where V is the number of vertices and E is the number of edges in the graph. Prim's algorithm performs well for both sparse and dense graphs. On the other hand, Kruskal's algorithm is also a greedy approach but focuses on sorting and considering edges in non-decreasing order of their weights. It checks if adding each edge to the MST creates a cycle and proceeds accordingly. Kruskal's algorithm also guarantees an optimal solution and has a runtime complexity of $O(ElogE)$ or $O(ElogV)$, depending on the implementation. This algorithm is particularly suitable for sparse graphs with many edges. Comparing the two algorithms, we find that Prim's algorithm often performs better in dense graphs due to its efficient representation using adjacency matrix. Its time complexity of $O(V^2)$ allows it to handle larger dense graphs more efficiently than Kruskal's algorithm. On the other hand, Kruskal's algorithm, with its time complexity of $O(ElogE)$ or $O(ElogV)$, tends to outperform Prim's algorithm in sparse graphs where the number of edges is much smaller than the number of vertices. In summary, both Prim's and Kruskal's algorithms offer reliable solutions for finding the minimum spanning tree. When deciding which algorithm to use, the graph's characteristics, such as sparsity, should be considered. Prim's algorithm is a strong choice for dense graphs, while Kruskal's algorithm excels in sparse graphs. By understanding these nuances, we can select the most appropriate algorithm for our specific graph structure and optimize the efficiency of our MST computations.

Github repository: $https://github.com/eamtcPROG/AA/tree/main/Lab7$