

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

**Laboratory work 7:
Empirical analysis of Prim's and Kruskal's
algorithms**

Elaborated:
st. gr. FAF-213

Konjevic Alexandra

Verified:
asist. univ.

Fiștic Cristofor

Chișinău – 2023

ALGORITHM ANALYSIS

Objective:

Study the greedy algorithm design technique. After that, implement Prim's and Kruskal's algorithms.

Tasks:

1. Implement the listed algorithms in a programming language
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.

Introduction:

Graph theory is a branch of mathematics that deals with the study of graphs, which are structures used to represent relationships between objects. In computer science, graphs are used to model a wide range of systems, such as computer networks, social networks, transportation systems, and more. One of the fundamental problems in graph theory is the minimum spanning tree (MST) problem, which involves finding the smallest possible tree that spans all the vertices of a graph.

Two of the most popular algorithms for solving the MST problem are Prim's algorithm and Kruskal's algorithm. Both algorithms are greedy algorithms that work by iteratively selecting edges that form the minimum spanning tree. Prim's algorithm starts with a single vertex and iteratively adds the edge with the smallest weight that connects

the current tree to a new vertex. Kruskal's algorithm, on the other hand, starts with a set of disconnected vertices and iteratively adds the edge with the smallest weight that connects two disjoint sets of vertices.

While both algorithms solve the same problem, they have different time and space complexities and are better suited for different types of graphs. In this report, we will explore the theoretical underpinnings of Prim's and Kruskal's algorithms, analyze their time and space complexities, and compare their strengths and weaknesses.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm ($T(n)$).

Input Format:

I generated random graphs of lengths from 100 vertexes to 700, with the step 100. For each graph I measured the time of executing of functions `prim` and `kruskal`.

IMPLEMENTATION

First of all, I used python library `networkx` to generate random graphs with edges, that have a random weight from 0 to 100, via the method:

```
def generate_random_graph(num_nodes):
    G = nx.Graph()
    for i in range(num_nodes):
        G.add_node(i)
    for i in range(num_nodes):
        for j in range(i+1, num_nodes):
            if random.random() < 0.5: # Add an edge with probability 0.5
                weight = random.randint(1, 100) # Generate a random weight
                G.add_weighted_edges_from([(i, j, weight)])
    return G
```

Figure 1. Method to generate random graphs in python

1. Kruskal method implementation

```
def kruskal(G: nx.Graph) -> List[nx.Graph]:  
    mst = []  
    sets = [{u} for u in G.nodes()]  
    edges = list(G.edges(data="weight"))  
    edges.sort(key=lambda x: x[2])  
    for u, v, w in edges:  
        set_u = None  
        set_v = None  
        for s in sets:  
            if u in s:  
                set_u = s  
            if v in s:  
                set_v = s  
        if set_u != set_v:  
            mst.append((u, v))  
            set_u |= set_v  
            sets.remove(set_v)  
        if len(mst) == len(G) - 1:  
            break  
    return [G.edge_subgraph(mst)]
```

Figure 2. Kruskal implementation in python

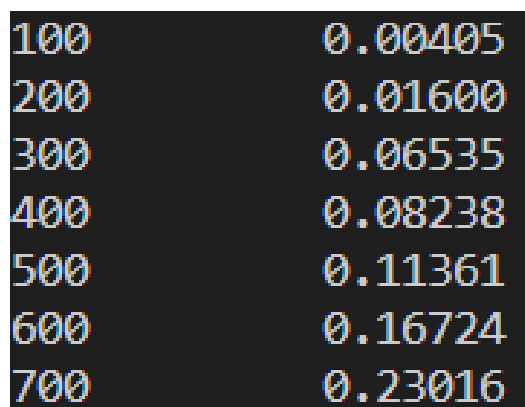
Algorithm Description:

Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, weighted graph. The algorithm starts by sorting the edges of the graph in ascending order by weight. It then iteratively adds the edges to the MST, starting with the smallest-weight edge and continuing until all vertices are connected. In each iteration, the algorithm checks whether adding the next edge creates a cycle in the MST. If the edge does not create a cycle, it is

added to the MST. If the edge creates a cycle, it is discarded.

The code above is an implementation of Kruskal's algorithm. It takes as input a NetworkX graph object and returns a list of one or more MSTs. The implementation uses a list of sets to keep track of the connected components of the graph. In each iteration, it finds the sets that contain the endpoints of the next edge and checks whether they are the same. If they are different, the sets are merged, and the edge is added to the MST. The algorithm continues until all vertices are connected or all edges have been considered.

The implementation first initializes an empty list to store the edges of the MST and a list of sets, each containing a single vertex. It then constructs a list of edges, sorted by weight, using the edges method of the NetworkX graph object. In each iteration, the algorithm selects the next edge from the sorted list and finds the sets that contain its endpoints using a loop over the list of sets. If the sets are different, the edge is added to the MST, and the sets are merged using the union (\cup) operator. The algorithm stops when the MST contains $n-1$ edges, where n is the number of vertices in the graph. Finally, the implementation constructs a subgraph of the input graph using only the edges of the MST and returns it as a list containing a single element.



100	0.00405
200	0.01600
300	0.06535
400	0.08238
500	0.11361
600	0.16724
700	0.23016

Figure 3. Results for Kruskal

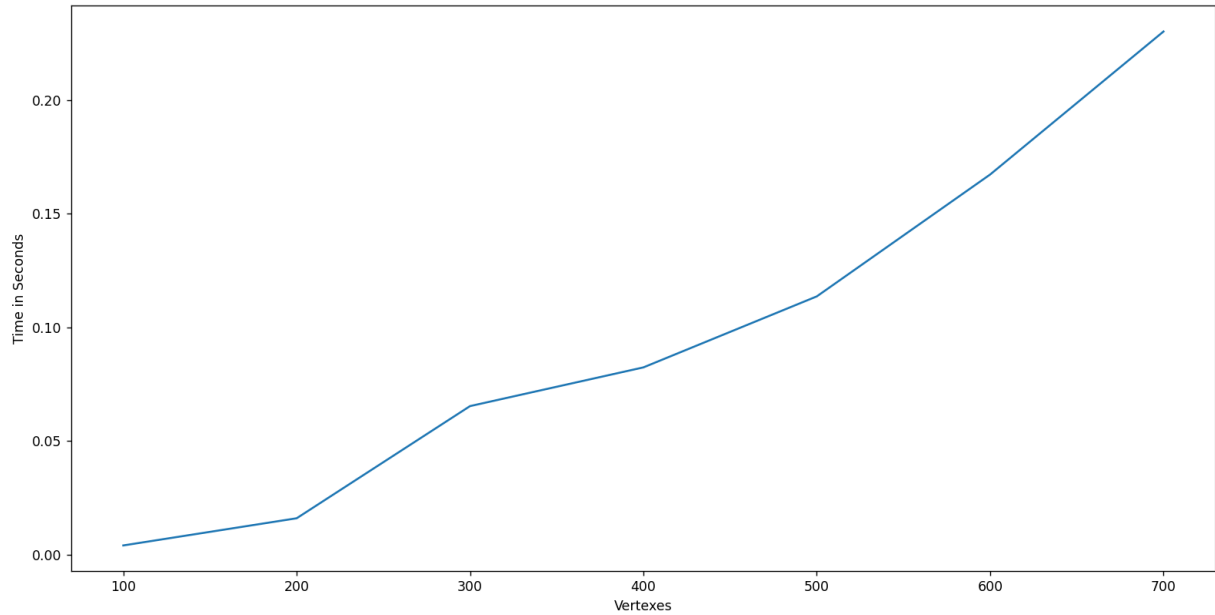


Figure 4. Graph for Kruskal

2. Prim method implementation

```
def prim(graph: nx.Graph) -> nx.Graph:
    T = nx.Graph()
    T.add_node(0)
    while len(T) < len(graph):
        min_edge = None
        min_weight = float("inf")
        for u in T.nodes():
            for v in graph.neighbors(u):
                if v not in T:
                    weight = graph.edges[u, v]["weight"]
                    if weight < min_weight:
                        min_edge = (u, v)
                        min_weight = weight
        T.add_node(min_edge[1])
        T.add_edge(*min_edge, weight=min_weight)
```

Figure 5. Prim implementation in python

Algorithm Description:

Prim's algorithm is another greedy algorithm used to find the minimum spanning tree (MST) of a connected, weighted graph. The algorithm starts by selecting an arbitrary vertex to be the root of the MST. It then iteratively adds the closest vertex not yet in the MST to the MST. In each iteration, the algorithm selects the closest vertex to the MST using a priority queue or a similar data structure. The priority of each vertex is defined as the weight of the edge connecting it to the MST.

The code above is an implementation of Prim's algorithm. It takes as input a NetworkX graph object and returns the MST as another NetworkX graph object. The implementation initializes an empty graph T and adds a single node to it, arbitrarily chosen to be node 0. It then iteratively adds nodes to the MST until all nodes are included. In each iteration, the implementation finds the edge with the smallest weight that connects a node in T to a node not yet in T . It then adds the new node and the selected edge to T .

The implementation uses a nested loop to iterate over all nodes in T and their neighbors in graph. It selects the edge with the smallest weight among the edges connecting nodes in T to nodes not in T . It then adds the endpoint of the selected edge not in T to T and adds the edge to T . The implementation continues until all nodes are included in T . The resulting graph T is returned as the MST of the input graph.

Overall, this implementation of Prim's algorithm is simple and intuitive, using only basic data structures such as sets and loops. However, its time complexity is $O(|V|^2)$, which can be slow for large graphs. More efficient implementations use a priority queue or a similar data structure to speed up the selection of the next vertex to add to the MST.

100	0.08800
200	0.74792
300	2.92043
400	6.28503
500	12.85280
600	23.76577
700	38.20629

Figure 6. Results for Prim

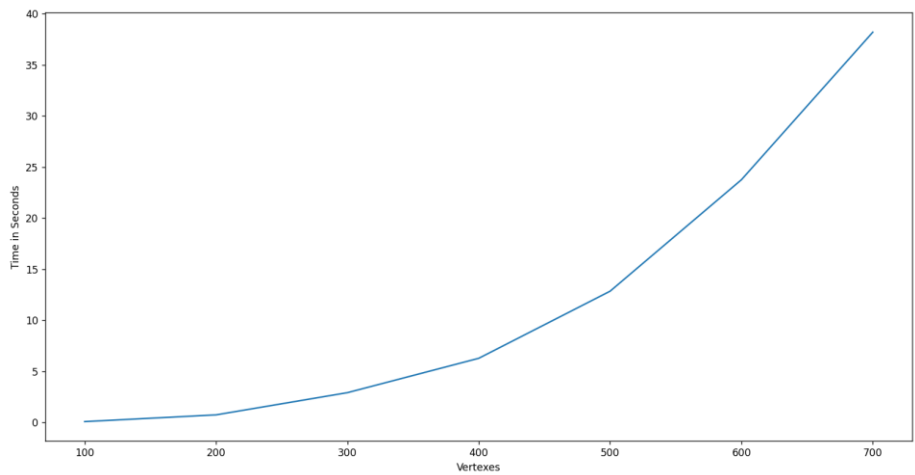


Figure 9. Graph for Prim

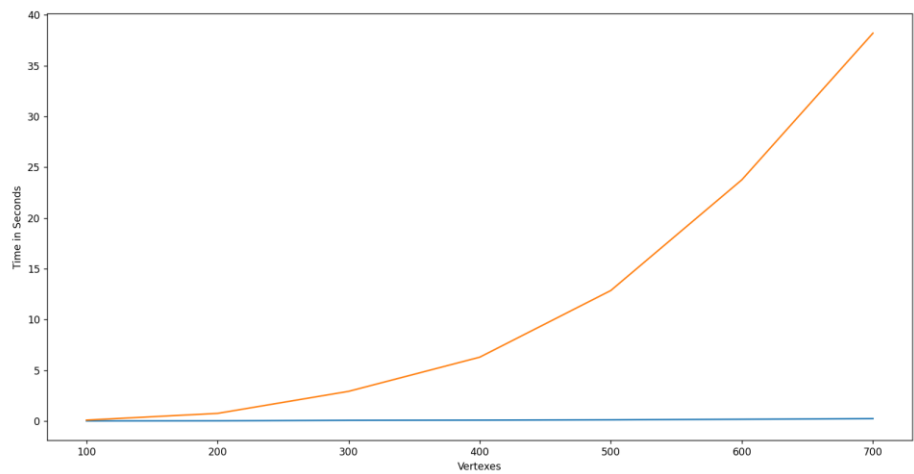


Figure 7. Graph with both Dijkstra and Floyd-Warshall

● – Prim ● – Kruskal

Conclusion

In conclusion, Prim's and Kruskal's algorithms are two widely used methods for finding the minimum spanning tree of a weighted, connected graph. Both algorithms are greedy and produce optimal solutions, but they differ in their approach to selecting edges for the MST. Prim's algorithm starts from a single vertex and iteratively adds the closest vertex not yet in the MST, while Kruskal's algorithm starts with an empty MST and iteratively adds the smallest-weight edge that does not create a cycle.

The choice of algorithm depends on the specific requirements of the problem, such as the size of the graph, the structure of the data, and the available computing resources. For small graphs, the simple implementation of Prim's algorithm provided in this report can be a good choice due to its ease of implementation and clarity. For larger graphs, more efficient implementations, such as using a priority queue, are recommended to reduce the time complexity and improve the performance.

Overall, the study of minimum spanning trees and their algorithms is an important topic in graph theory and computer science, with many real-world applications in network design, transportation planning, and resource allocation. Understanding these algorithms and their strengths and limitations can provide valuable insights into solving complex optimization problems in various fields.

Github repository: <https://github.com/alya1007/Labs-semester-4/tree/master/AA>