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

Laboratory work 5:

Study and empirical analysis of Kruskal and Prim algorithms.

Elaborated:
st. gr. FAF-233 Ploteanu Augustin

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and empirical analysis of Minimum Spanning Tree algorithms. Analysis of Kruskal and Prim.

Tasks:

1. Implement the algorithms listed above 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.

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 analyzed.

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 behavior 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:

Minimum Spanning Tree (MST) algorithms are essential for optimizing connectivity in weighted, undirected graphs. Applications range from network design and clustering to image segmentation and approximation algorithms.

In this laboratory, we focus on two well-known MST algorithms:

Kruskal's Algorithm – A greedy approach that builds the MST by adding the smallest weight edges in increasing order, ensuring no cycles are formed. It uses a disjoint-set (union-find) structure to track connected components.

Prim's Algorithm – Also greedy, but it grows the MST by always choosing the smallest edge that connects a visited node to an unvisited one, often using a priority queue for efficient selection.

Though both algorithms achieve the same goal, they differ in strategy, data structure usage, and performance under different graph structures. This lab investigates their runtime behavior across various graph types and sizes.

Comparison Metric:

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

Input Format:

To evaluate the performance of graph algorithms under varying structural conditions, eight distinct graph types of increasing size will be analyzed. These graph types include:

-Complete graphs, where every node is connected to every other node.

-Sparse graphs, which maintain minimal connectivity.

-Dense graphs, with a high but non-complete number of edges.

-Trees, which are acyclic connected graphs.

-Bipartite graphs, where nodes are divided into two sets with edges only between sets.

-Cyclic graphs, containing multiple cycles.

-Acyclic graphs, such as directed acyclic graphs (DAGs).

-Grid graphs, structured like 2D matrices.

Each graph type will be generated at six different sizes, ranging from 10 to 400 nodes. For weighted and directed variants, edges will be assigned random weights and directions.

IMPLEMENTATION

Both algorithms will be implemented in python and analyzed 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 report with input will vary depending on memory of the device used.

Kruskal's Algorithm:

Kruskal's Algorithm constructs a Minimum Spanning Tree (MST) by selecting edges in order of increasing weight, ensuring that no cycles are formed. It uses a disjoint-set (union-find) structure to efficiently track which components are connected. With a time complexity of O(E log E), Kruskal is especially effective on sparse graphs. It guarantees a globally optimal MST and is useful in network design and clustering applications.

Algorithm Description:

Kruskal's Algorithm follows the algorithm as shown in the next pseudocode:

```
KRUSKAL(edges, num_nodes)
    sort edges by ascending weight
    create parent array where parent[i] = i
    create empty list MST

    for each (u, v, weight) in edges do
        if FIND(u) ≠ FIND(v) then
            UNION(u, v)
            add (u, v, weight) to MST

    return MST

FIND(x)
    if parent[x] ≠ x then
        parent[x] = FIND(parent[x])
    return parent[x]

UNION(x, y)
    rootX = FIND(x)
    rootY = FIND(y)
    if rootX ≠ rootY then
        parent[rootY] = rootX
```

Implementation:

```python
def kruskal(edges, num_nodes):
    parent = list(range(num_nodes))

    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]

    def union(x, y):
        rootX = find(x)
        rootY = find(y)
        if rootX != rootY:
            parent[rootY] = rootX
            return True
        return False

    mst = []
    edges.sort(key=lambda x: x[2])

    for u, v, weight in edges:
        if union(u, v):
            mst.append((u, v, weight))

    return mst
```

Figure 1 Kruskal's algorithm in Python

Results:

The numerical results after execution of the algorithms on the aforementioned graphs can be seen in the appendix. Kruskal shows strong performance across all graph types, particularly excelling in sparse

and tree graphs where fewer edge comparisons are needed. Its efficiency slightly diminishes on dense graphs due to the increased number of edge sorting and union-find operations, but it remains competitive overall.

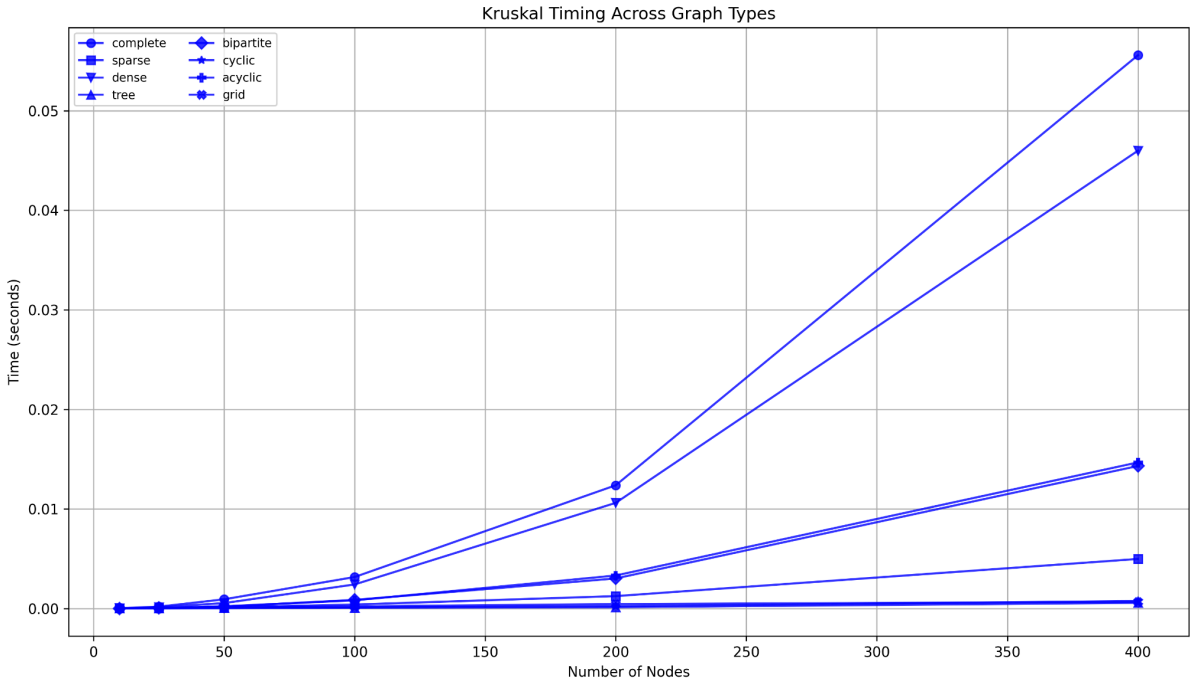

Figure 2 Kruskal's algorithm graph

Prim's Algorithm:

Prim's Algorithm builds a Minimum Spanning Tree by growing the tree from a starting vertex and always selecting the minimum-weight edge connecting a visited node to an unvisited one. It typically uses a priority queue (min-heap) to find the next optimal edge efficiently. The algorithm has a time complexity of O((V + E) log V) and is especially well-suited for dense graphs. Prim's approach is greedy and ensures an optimal MST for connected weighted graphs.

Algorithm Description:

Prim's Algorithm follows the algorithm as shown in the next pseudocode:

```
PRIM(graph, start)
    create empty set visited
    create min-priority queue Q
    insert (0, start, None) into Q
    create empty list MST

    while Q is not empty do
        (weight, node, parent) = extract_min(Q)
        if node not in visited then
            add node to visited
            if parent ≠ None then
                add (parent, node, weight) to MST
            for each (neighbor, edge_weight) in graph[node] do
                if neighbor not in visited then
                    insert (edge_weight, neighbor, node) into Q

    return MST
```

Implementation:

```python
def prim(graph, start):
    visited = set()
    min_heap = [(0, start, None)]
    mst = []

    while min_heap:
        weight, node, parent = heapq.heappop(min_heap)
        if node not in visited:
            visited.add(node)
            if parent is not None:
                mst.append((parent, node, weight))
            for neighbor, edge_weight in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(min_heap, (edge_weight, neighbor, node))

    return mst
```

Figure 4 Prim's algorithm in Python

Results:

The numerical results after execution of the algorithms on the aforementioned graphs can be seen in the appendix. Prim performs similarly to Kruskal in small graphs but tends to be slower in dense graphs where its edge selection process becomes more expensive. It scales reasonably well but exhibits higher runtime variance depending on graph connectivity, performing best when the number of candidate edges is limited.

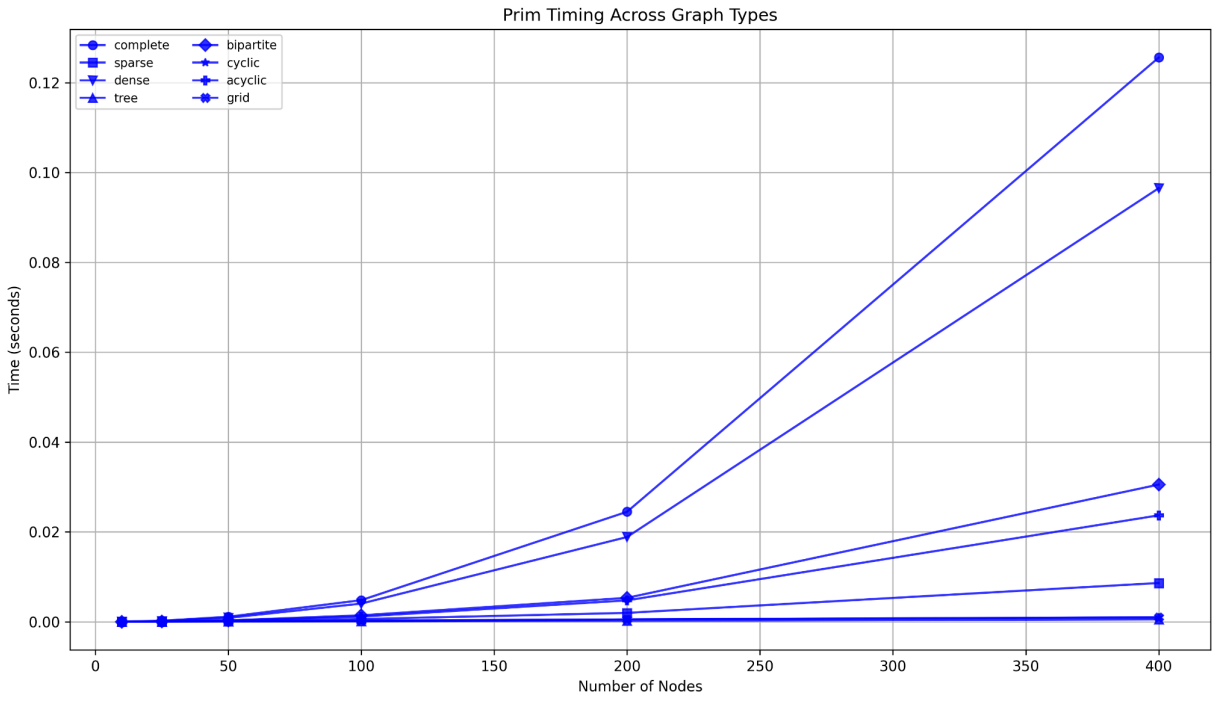

Figure 6 Prim's algorithm graph

CONCLUSION

Through empirical analysis, this study has evaluated the efficiency of two Minimum Spanning Tree algorithms, Kruskal and Prim, by analyzing their execution time under different input conditions. The goal was to identify the scenarios in which each algorithm performs optimally and to determine their feasibility for various applications. The results provide insights into the strengths and limitations of each approach, as well as opportunities for further optimizations.

Kruskal's algorithm was most efficient in sparse, tree, acyclic, and bipartite graphs, where the limited number of edges makes its edge-sorting and disjoint-set operations quick and effective. It processes edges globally, making it independent of the starting node and ideal for graphs with many disconnected components.

Prim's algorithm, on the other hand, performed better in dense, complete, and grid graphs, where its vertex-oriented edge growth rapidly connects new nodes through a priority queue. It efficiently expands the MST from a seed node and is less affected by global edge sorting.

Neither Kruskal nor Prim handles directed graphs or negative edge weights; MSTs are only defined for undirected, positively weighted graphs. Both are applicable to weighted networks but are sensitive to density and connectivity.

In conclusion, Kruskal is the better choice for sparse and tree-like networks, offering fast performance due to minimal edge comparisons. Prim excels in dense and grid-like graphs, where its local edge expansion is more effective. The optimal choice depends on graph density, edge distribution, and whether a node- or edge-centric approach is more appropriate.

Numerical Results Appendix:

https://github.com/augustin-ploteanu/AA_Labs/blob/main/Lab3-4-5/results.txt