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

REPORT

Laboratory work no. 7
Prim & Kruskal

Elaborated:
st. gr. FAF-213 Botnari Ciprian

Verified:
asist. univ. Fiștic Cristofor

Chișinău – 2023

Table of Contents

ALGORITHM ANALYSIS

Objective

Study and analyze 2 algorithms for finding the minimum spanning tree (MST): Prim and Kruskal.

Tasks

1. Implement 2 algorithms for finding the minimum spanning tree (MST)
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical notes

An alternative approach to evaluating the complexity of an algorithm is through empirical analysis, which can provide insights on the complexity class of the algorithm, comparison of efficiency between algorithms solving the same problem, comparison of different implementations of the same algorithm, and performance on specific computers.

The process of empirical analysis typically involves:

1. Establishing the purpose of the analysis
2. Choosing the efficiency metric (number of operations or execution time)
3. Defining the properties of the input data
4. Implementing the algorithm in a programming language
5. Generating input data
6. Running the program with each set of data
7. Analyzing the results

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

A minimum spanning tree (**MST**) is a subset of the edges of a weighted undirected graph that connects all the vertices with the minimum total edge weight. In other words, it is a tree that spans all the vertices of the graph with the smallest possible sum of edge weights.

Properties of a Minimum Spanning Tree:

1. **Connectivity**: A minimum spanning tree ensures that all vertices of the graph are connected. There is a path between any two vertices in the MST.
2. **Acyclic**: A minimum spanning tree does not contain any cycles. Adding any edge to the MST would create a cycle.
3. V-1 **Edges**: A minimum spanning tree has exactly V-1 edges, where V is the number of vertices in the graph.
4. **Minimal Total Weight**: Among all possible spanning trees of the graph, the minimum spanning tree has the smallest total weight.

Applications of Minimum Spanning Trees:

1. **Network Design**: MSTs are used in designing efficient network infrastructures, such as connecting cities with minimum cost in telecommunication networks or establishing reliable connections in computer networks.
2. **Cluster Analysis**: MSTs can be used to analyze and group data points in cluster analysis, where the goal is to identify clusters of similar objects.
3. **Approximation Algorithms**: MSTs are utilized as a part of approximation algorithms to find near-optimal solutions for optimization problems, such as the traveling salesman problem and facility location problems.
4. **Image Segmentation**: MSTs can be applied in image processing and computer vision tasks, specifically for segmenting images into regions based on the similarity of pixel intensities or other features.
5. **Spanning Tree Protocols**: MSTs are used in network protocols like Rapid Spanning Tree Protocol (RSTP) and Multiple Spanning Tree Protocol (MSTP) to prevent loops and ensure efficient communication in network topologies.

Algorithms for Finding Minimum Spanning Trees: Two popular algorithms for finding minimum spanning trees are Prim's algorithm and Kruskal's algorithm, as mentioned earlier. These algorithms use different approaches but ultimately achieve the same result by selecting edges based on their weights and connectivity. Other algorithms, such as Boruvka's algorithm and Jarník's algorithm (also known as Prim's algorithm with a different implementation), are also used to find minimum spanning trees. The choice of algorithm depends on the specific requirements of the problem and the characteristics of the input graph.

Comparison Metric

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

Input Format

The algorithms will take graphs with nodes:

- 10
- 50
- 100
- 500
- 1000
- 5000
- 10000
- 50000
- 100000
- 500000

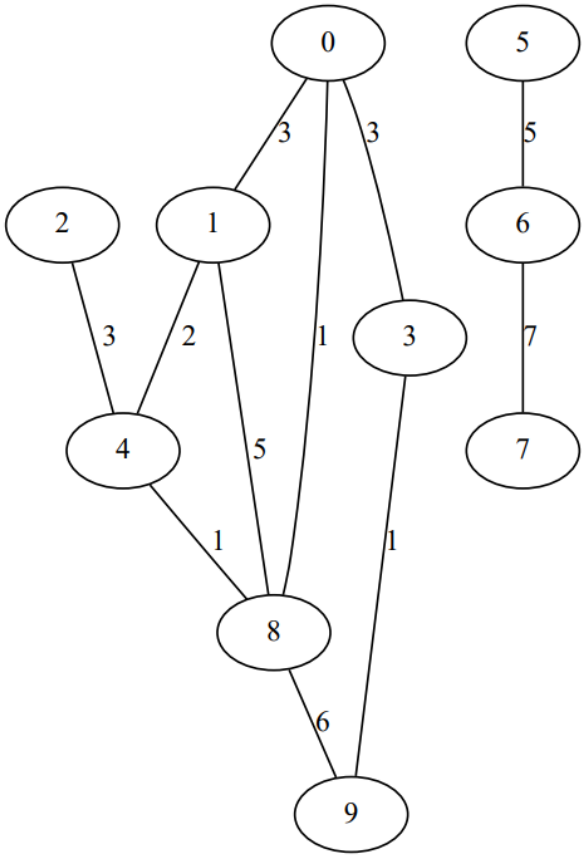


Figure 1. Random graph

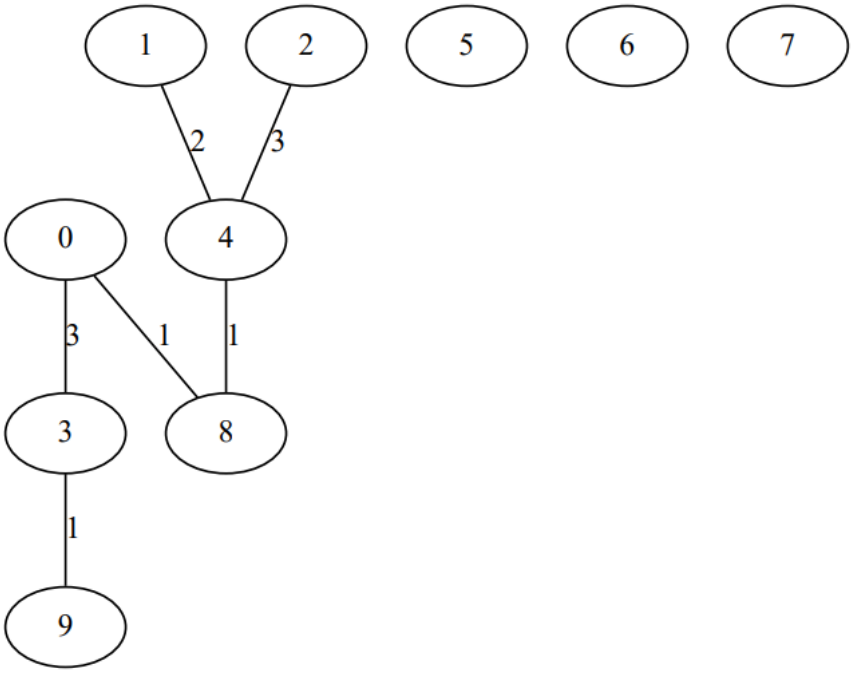


Figure 2. Prim MST

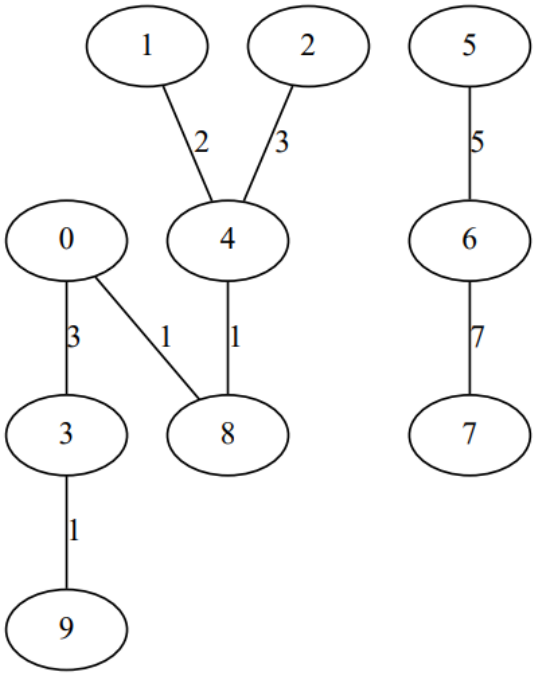


Figure 3. Kruskal MST

IMPLEMENTATION

All algorithms will be put into practice in Python and objectively evaluated depending on how long it takes to complete each one. The particular efficiency in report with input will vary based on the memory of the device utilized, even though the overall trend of the results may be similar to other experimental data. As determined by experimental measurement, the error margin will be a few seconds.

Prim

Algorithm Description:

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a weighted undirected graph. The minimum spanning tree is a subset of the graph's edges that connects all the vertices with the minimum total edge weight. Prim's algorithm starts with an arbitrary vertex and iteratively adds the closest vertex to the current MST, until all vertices are included, forming a tree.

Implementation

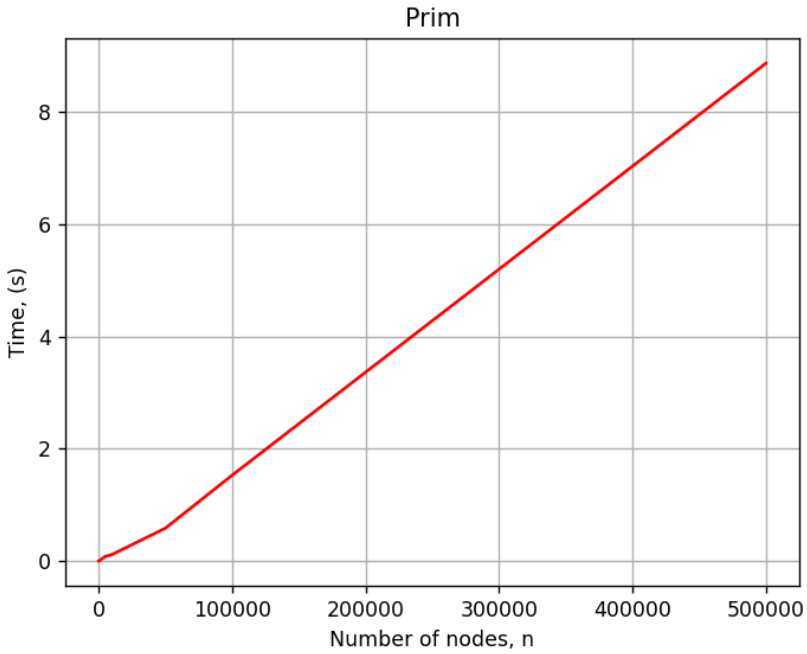
```python
def Prim(graph):
    visited = set()
    mst = []

    start_vertex = next(iter(graph))
    visited.add(start_vertex)
    edges = [
        (cost, start_vertex, end_vertex)
        for end_vertex, cost in graph[start_vertex].items()
    ]

    heapq.heapify(edges)

    while edges:
        cost, start_vertex, end_vertex = heapq.heappop(edges)
        if end_vertex not in visited:
            visited.add(end_vertex)
            mst.append((start_vertex, end_vertex, cost))
            for next_vertex, cost in graph[end_vertex].items():
                if next_vertex not in visited:
                    heapq.heappush(edges, (cost, end_vertex, next_vertex))
    return mst
```

Results

Minimum Spanning Tree / n	10	50	100	500	1000	5000	10000	50000	100000	500000
Prim	9e-05	0.00043	0.00075	0.00614	0.01376	0.08095	0.11449	0.58387	1.52818	8.87321
Kruskal	0.00036	0.00194	0.0131	0.01915	0.05083	0.18188	0.27145	0.9963	2.62755	11.42579

Prim's algorithm guarantees that the resulting tree will be a minimum spanning tree. The time complexity of Prim's algorithm is $O(V^2)$ using an adjacency matrix representation, and O(E log V) using an adjacency list representation, where V is the number of vertices and E is the number of edges in the graph.

Graph



Kruskal

Algorithm Description:

Kruskal's algorithm is another greedy algorithm used to find the minimum spanning tree (MST) of a weighted undirected graph. It starts by sorting the edges of the graph in ascending order of their weights and then iteratively adds the smallest weighted edge that does not create a cycle in the MST. Kruskal's algorithm keeps adding edges until there are V-1 edges in the MST, where V is the number of vertices in the graph.

Implementation

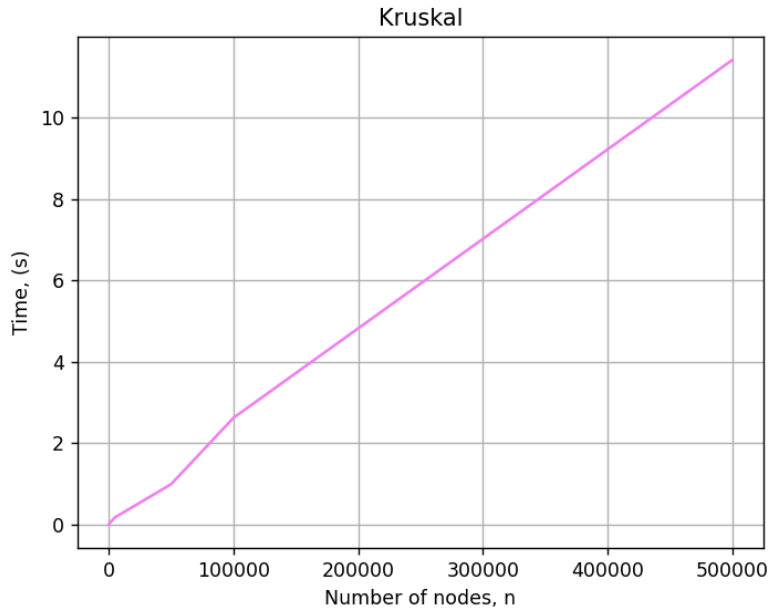
```python
def Kruskal(graph):
    mst = []
    ds = DisjointSet(len(graph))
    edges = [(cost, start_vertex, end_vertex) for start_vertex, edges in graph.items() for end_vertex, cost in edges.items()]
    edges.sort()

    for cost, start_vertex, end_vertex in edges:
        if ds.union(int(start_vertex), int(end_vertex)):
            mst.append((start_vertex, end_vertex, cost))

    return mst
```

Results

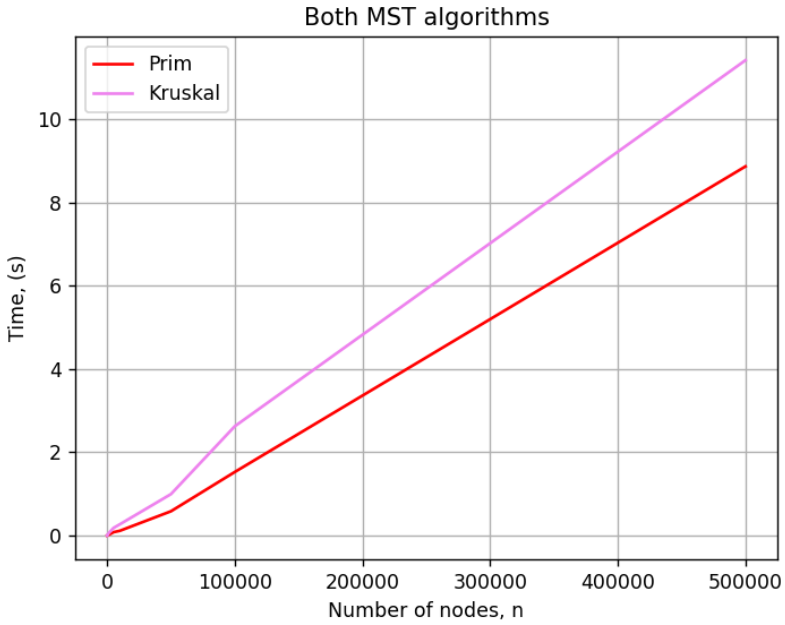
Minimum Spanning Tree / n	10	50	100	500	1000	5000	10000	50000	100000	500000
Prim	9e-05	0.00043	0.00075	0.00614	0.01376	0.08095	0.11449	0.58387	1.52818	8.87321
Kruskal	0.00036	0.00194	0.0131	0.01915	0.05083	0.18188	0.27145	0.9963	2.62755	11.42579

Kruskal's algorithm also guarantees that the resulting tree will be a minimum spanning tree. The time complexity of Kruskal's algorithm is 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, depending on the sorting algorithm used for the edges.

Graph



Comparison



Prim's Algorithm and Kruskal's Algorithm are both used to find minimum spanning trees (MSTs) in weighted undirected graphs. While they achieve the same result, they differ in their approach and can have different performance characteristics depending on the input graph.

1. **Approach**:
   - Prim's Algorithm: It follows a vertex-centric approach. It starts with an arbitrary vertex and iteratively grows the MST by adding the closest vertex to the current MST.
   - Kruskal's Algorithm: It follows an edge-centric approach. It sorts all the edges and greedily adds them to the MST if they do not create a cycle.

2. **Edge Selection**:
   - Prim's Algorithm: It selects edges based on the weights of the adjacent vertices. It always adds the minimum-weight edge connected to the current MST.
   - Kruskal's Algorithm: It sorts all the edges by weight and iterates through them in ascending order. It adds the edges one by one if they do not create a cycle.

3. **Performance**:
    - Prim's Algorithm: The time complexity of Prim's algorithm is O(V^2) using an adjacency matrix representation, and O(E log V) using an adjacency list representation. It performs better when the graph has a dense or semi-dense structure (many edges) since the adjacency matrix representation allows for faster access to adjacent vertices.
    - Kruskal's Algorithm: The time complexity of Kruskal's algorithm is O(E log E) or O(E log V), depending on the sorting algorithm used for the edges. It performs better when the graph has a sparse structure (fewer edges) since it relies on sorting the edges.

4. **Memory Usage**:
    - Prim's Algorithm: It typically requires a priority queue or a min-heap to efficiently select the minimum-weight edge. The memory usage is generally proportional to the number of vertices (V).
    - Kruskal's Algorithm: It requires a disjoint-set data structure to check for cycles. The memory usage is proportional to the number of edges (E).

5. **Connectivity**:
    - Prim's Algorithm: It guarantees that the MST is connected from the starting vertex. If the graph is disconnected, multiple MSTs can be formed by running the algorithm from different starting vertices.
    - Kruskal's Algorithm: It also guarantees connectivity since it adds edges that do not create cycles. The resulting MST may not be connected from a specific vertex, but it will be a forest of MSTs.

CONCLUSION

In this laboratory work, we explored two popular algorithms for finding minimum spanning trees (MSTs) in weighted undirected graphs: Prim's Algorithm and Kruskal's Algorithm. We compared and analyzed their approaches, discussed when to use each algorithm, and identified their advantages and disadvantages.

Prim's Algorithm follows a vertex-centric approach, starting with an arbitrary vertex and iteratively growing the MST by adding the closest vertex to the current MST. It selects edges based on the weights of the adjacent vertices, guaranteeing connectivity from the starting vertex. Prim's Algorithm performs better when the graph has a dense or semi-dense structure, as its time complexity is $O(V^2)$ using an adjacency matrix representation and O(E log V) using an adjacency list representation.

Kruskal's Algorithm, on the other hand, follows an edge-centric approach. It sorts all the edges by weight and greedily adds them to the MST if they do not create a cycle. Kruskal's Algorithm performs better when the graph has a sparse structure, as its time complexity is O(E log E) or O(E log V), depending on the sorting algorithm used. It guarantees connectivity by adding edges that do not create cycles but may result in a forest of MSTs.

When choosing between Prim's and Kruskal's algorithms, several factors should be considered. Prim's Algorithm is suitable when the starting vertex is predefined or needs to be specified, and when the graph is dense or semi-dense. It guarantees connectivity from the starting vertex and has a more efficient time complexity in such cases. Kruskal's Algorithm is advantageous when the graph is sparse and has fewer edges. It works well in situations where the starting vertex is not crucial and can handle disconnected graphs.

In terms of memory usage, Prim's Algorithm typically requires a priority queue or min-heap, with memory usage proportional to the number of vertices. Kruskal's Algorithm requires a disjoint-set data structure, with memory usage proportional to the number of edges.

Both algorithms provide a minimum spanning tree, ensuring the minimum total weight and connectivity. However, Prim's Algorithm guarantees a connected MST from the starting vertex, while Kruskal's Algorithm may produce a forest of MSTs. The choice between the two algorithms ultimately

depends on the characteristics of the input graph, available data structures, and specific requirements of the problem at hand.

Finally, understanding Prim's Algorithm and Kruskal's Algorithm provides valuable tools for solving optimization problems and analyzing network structures. By applying these algorithms, we can efficiently find minimum spanning trees, enabling us to make informed decisions in various practical scenarios.

GITHUB

Sufferal/Algorithms: Analysis of algorithms (github.com)