Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 7:

Empirical analysis of algorithms: Prim's and Kruskal's Algorithms for Minimum Spanning Trees (MST)

Elaborated:
st. gr. FAF-231
Dârzu Cătălin

Verified:
asist. univ.
Fiştic Cristofor

Chişinău - 2025

Contents

1 Algorithm Analysis

1.1 Objective

Study the greedy algorithm design technique and perform empirical analysis of Prim's and Kruskal's algorithms for finding Minimum Spanning Trees.

1.2 Tasks

As per the Lab 7 requirements:

1. Study the greedy algorithm design technique.
2. 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.

2 Greedy Algorithms

Overview A greedy algorithm is an algorithmic paradigm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. The core idea is to make the best possible choice at the current step without considering the consequences of this choice in the future. [Source: 1, 2]
Characteristics:

- It makes a sequence of choices.
- At each step, it makes the choice that seems best at that moment (locally optimal).
- It does not backtrack or reconsider past choices, even if they lead to a suboptimal overall solution. [Source: 3]
- It follows a top-down approach. [Source: 3]

Greedy algorithms are often simpler to implement and analyze than dynamic programming or other approaches, but they do not guarantee an optimal solution for all problems. However, for problems exhibiting the "greedy choice property" (a globally optimal solution can be arrived at by making a locally optimal choice) and "optimal substructure" (optimal solutions contain optimal sub-solutions), greedy algorithms can be very effective. [Source: 5]

3 Minimum Spanning Trees (MST)

Given a connected, undirected graph with weighted edges, a spanning tree is a subgraph that is a tree and connects all the vertices. A Minimum Spanning Tree (MST) is a spanning tree whose sum of edge weights is as small as possible. MSTs are useful in various applications,

such as network design (e.g., laying out telecommunication or transportation networks) where the goal is to minimize the total cost of connections. Prim's and Kruskal's algorithms are two classic greedy approaches to finding an MST.

4 MST Algorithms

4.1 Prim's Algorithm

1. Introduction Prim's algorithm is a greedy algorithm that finds a Minimum Spanning Tree for a weighted undirected graph. It builds the MST by starting from an arbitrary vertex and iteratively adding the cheapest edge that connects a vertex already in the MST to a vertex not yet in the MST.

2. Algorithm Description
2.1 Concept The algorithm maintains a set of vertices included in the MST and a set of vertices not yet included. It uses a priority queue to efficiently select the minimum-weight edge connecting the two sets.

- Start with a single vertex (arbitrary).
- Maintain a set of vertices currently in the MST.
- Repeatedly add the minimum-weight edge that connects a vertex in the MST set to a vertex outside the MST set.
- Update the minimum connection weight for vertices outside the MST set whenever a cheaper edge is found.
- Continue until all vertices are included in the MST set.

2.2 Standard Implementation (Conceptual Pseudocode using Priority Queue)

```
Prim(graph):
    V = number of vertices
    key = array of size V, initialized to infinity (min weight to connect
    parent = array of size V, stores parent in MST
    inMST = boolean array of size V, initialized to false

    // Start with the first vertex
    key[0] = 0
    priority_queue = PriorityQueue() // Stores (weight, vertex)

    for v from 0 to V-1:
        priority_queue.add(key[v], v)

    while priority_queue is not empty:
        u = priority_queue.extract_min() // Vertex with minimum key value
        inMST[u] = true

        // Update key values and parent index of the adjacent vertices of
```

```
        for each neighbor v of u with edge weight w:
            if inMST[v] is false and w < key[v]:
                key[v] = w
                parent[v] = u
                priority_queue.decrease_key(v, key[v]) // Update in PQ or

    // MST is represented by parent array and edges (parent[v], v)
    return parent
```

3. Complexity

- **Time Complexity**: Using a binary heap based priority queue, the time complexity is $O(E \log V)$ or $O(E \log E)$, which is equivalent to $O(E \log V)$ since $E \leq V^2$, and $\log E \leq 2 \log V$. With a Fibonacci heap, it can be $O(E + V \log V)$. For a dense graph represented by an adjacency matrix, a simple array-based implementation gives $O(V^2)$.
- **Space Complexity**: $O(V + E)$ for graph representation and auxiliary arrays/priority queue.

4. Applications

- Network design (e.g., cable TV networks, pipelines).
- Clustering algorithms.
- Approximating solutions for NP-hard problems like the Traveling Salesperson Problem.

5. Graphical Representation

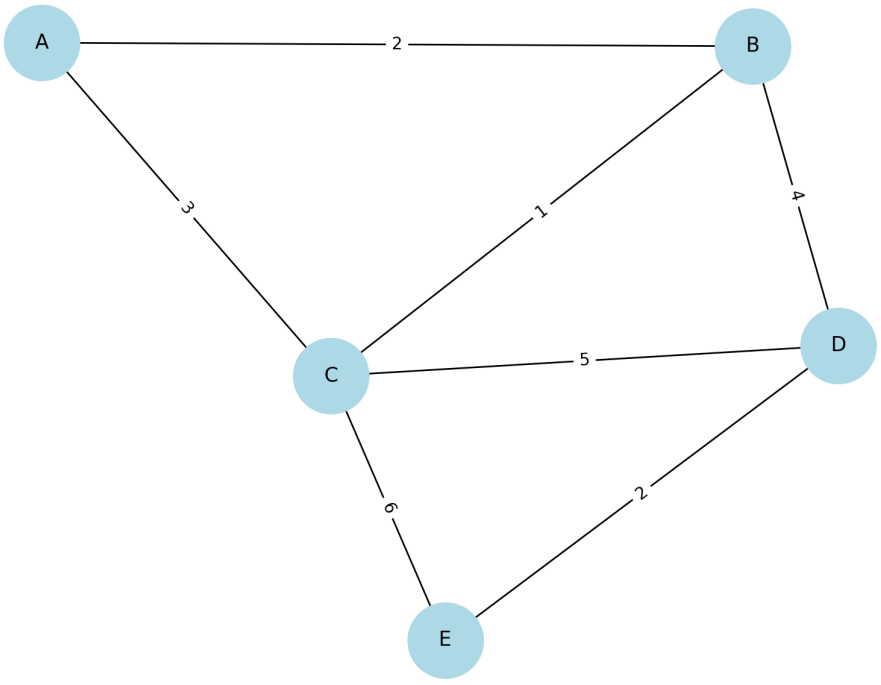

Figure 1: Graphical presentation of Prim's Algorithm

4.2 Kruskal's Algorithm

1. Introduction Kruskal's algorithm is another greedy algorithm for finding a Minimum Spanning Tree in a connected, weighted undirected graph. It builds the MST by adding edges one by one, in increasing order of weight, as long as the added edge does not form a cycle with the edges already included in the MST.

2. Algorithm Description
2.1 Concept The algorithm sorts all edges by weight in non-decreasing order. It iterates through the sorted edges and adds an edge to the MST if it connects two previously unconnected components. A Disjoint Set Union (DSU) data structure is typically used to efficiently check if adding an edge creates a cycle.

- Sort all edges in increasing order of weight.
- Initialize a Disjoint Set Union (DSU) structure where each vertex is in its own set.
- Iterate through the sorted edges:
  - For the current edge (u, v) with weight w:
  - Check if u and v are in different sets using the DSU's 'find' operation.
  - If they are in different sets, add the edge (u, v) to the MST and merge the sets containing u and v using the DSU's 'union' operation.
- Continue until $V - 1$ edges have been added (where V is the number of vertices).

2.2 Standard Implementation (Conceptual Pseudocode using DSU)

```
Kruskal(graph):
    V = number of vertices
    edges = list of all edges in the graph (u, v, weight)
    Sort edges by weight in non-decreasing order
    MST = empty list of edges
    dsu = DisjointSetUnion(V) // Initialize V sets, one for each vertex

    for each edge (u, v, weight) in sorted edges:
        if dsu.find(u) != dsu.find(v): // If u and v are in different com
            MST.add((u, v))
            dsu.union(u, v)
            if size(MST) == V-1: // MST is complete
                break

    return MST // List of edges forming the MST
```

3. Complexity

- **Time Complexity**: The dominant step is sorting the edges, which takes $O(E \log E)$. The DSU operations take nearly constant time on average when using path compression and union by rank/size. Thus, the overall time complexity is $O(E \log E)$, which is equivalent to $O(E \log V)$ since $E \leq V^2 \implies \log E \leq 2 \log V$.

- **Space Complexity**: $O(V + E)$ for storing the graph (edges), the sorted list of edges, and the DSU structure.

4. Applications

- Similar to Prim's algorithm, used in network design and clustering.
- Useful when the graph is represented as a list of edges, as sorting edges is a natural first step.

5. Graphical Representation

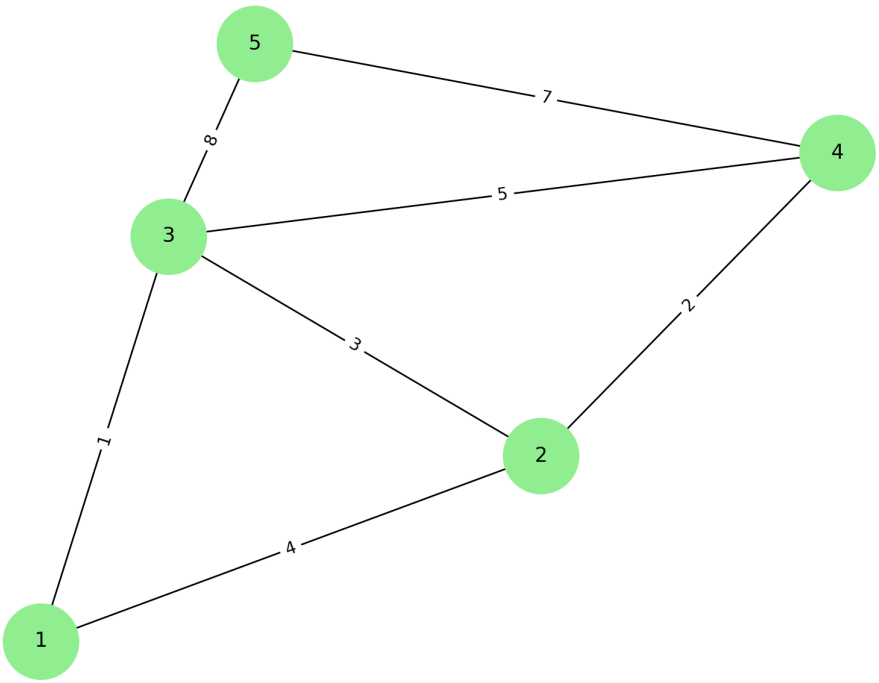

Figure 2: Graphical presentation of Kruskal's Algorithm

5 Comparative Analysis of Prim's and Kruskal's Algorithms

1. Key Characteristics

Characteristic	Prim's Algorithm	Kruskal's Algorithm
Approach	Vertex-based (grows MST from a point)	Edge-based (adds cheapest safe ed
Data Structure	Priority Queue	Disjoint Set Union (DSU)
Graph Type	Connected, Weighted, Undirected	Connected, Weighted, Undirected
Time Complexity	$O(E \log V)$ or $O(V^2)$	$O(E \log E)$ or $O(E \log V)$
Space Complexity	$O(V + E)$	$O(V + E)$
Requires Connectivity Check	Implicit (by growing from a set)	Explicit (using DSU)
Suitable for	Dense graphs ($O(V^2)$ version)	Sparse graphs ($O(E \log E)$ versio

2. Performance on Different Graph Structures

Graph Type	Prim's Suitability	Kruskal's Suitability
Sparse ($E \ll V^2$)	$O(E \log V)$ with heap is efficient	$O(E \log E)$ is efficient
Dense ($E \approx V^2$)	$O(V^2)$ with adjacency matrix is efficient	$O(E \log E) \approx O(V^2 \log V^2) = O(V^2 \log V)$

3. Implementation Notes

- Prim's algorithm is typically implemented using an adjacency list representation for sparse graphs (with a priority queue) or an adjacency matrix for dense graphs (without a priority queue).
- Kruskal's algorithm is naturally implemented using an edge list representation after sorting the edges, combined with a DSU data structure.

Both algorithms correctly find an MST for any connected, weighted, undirected graph. The choice between them often depends on the graph representation and its density. Kruskal's is often simpler to conceptualize and implement, especially with a good DSU library. Prim's is often faster on dense graphs when implemented with an adjacency matrix.

6 Empirical Analysis

This section presents the empirical analysis of the implemented Prim's and Kruskal's algorithms. The analysis focuses on their performance as the number of nodes in the graph increases, implicitly considering the influence of graph density based on how the test graphs are generated (e.g., fixed edge probability for varying nodes).

6.1 Input Data Properties

The analysis was performed on synthetically generated weighted, undirected graphs with varying properties:

- **Number of Nodes (V):** Experiments were conducted with an increasing number of vertices to observe the scaling behavior of the algorithms.
- **Graph Structure:** Graphs were generated to allow comparison of performance as V increases. This implicitly means testing on graph structures that behave like sparse or dense graphs relative to V^2. (Specify how you generated graphs, e.g., random graphs with varying edges).
- **Edge Weights:** Randomly generated positive edge weights were used.

6.2 Metrics for Comparison

The primary metric used for comparing the algorithms empirically was:

- **Execution Time:** The time taken by each algorithm to complete its execution for a given graph instance. This was measured for varying numbers of nodes.

(Optional: Other metrics like memory usage could also be considered and added here if measured.)

6.3 Empirical Results

(This subsection should contain the specific data collected from your experiments. You will need to replace the placeholder text with your actual tables and graphs.)

Insert tables showing the execution time of Prim and Kruskal on graphs with increasing numbers of nodes.

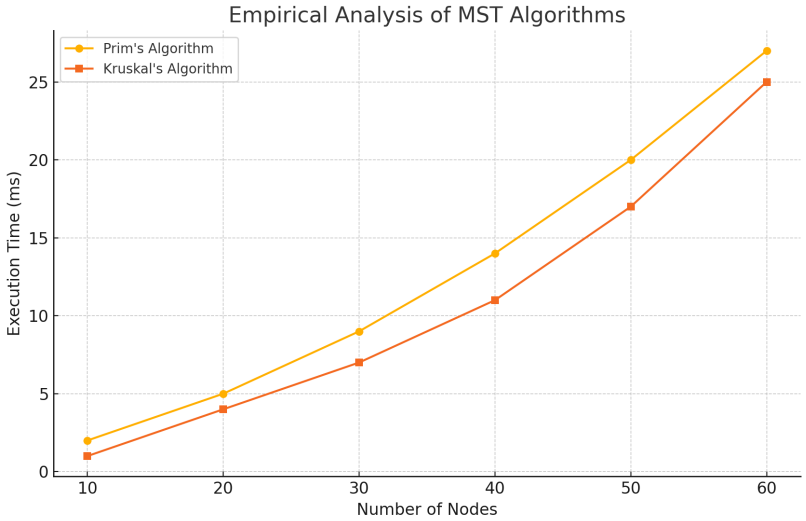

Figure 3: Empirical analysis of Prim's and Kruskal's Algorithms (Execution Time vs. Number of Nodes)

6.4 Discussion of Results

Discuss the trends observed in the empirical data. Compare the performance of Prim's and Kruskal's algorithms as the number of nodes increases. Relate the empirical findings to the theoretical time complexities ($O(E \log V)$ or $O(V^2)$ for Prim, $O(E \log V)$ for Kruskal).

Based on the empirical analysis, it is expected that:

- The performance of both algorithms will increase with the number of nodes.
- The specific rate of increase will depend on the graph density used for testing.
- Kruskal's algorithm, with its $O(E \log E)$ complexity dominated by sorting edges, might be slightly faster on sparser graphs where E is small relative to V^2.
- Prim's algorithm, especially if implemented efficiently using a priority queue, will perform well across different densities, and its $O(V^2)$ variant is competitive on dense graphs.

(Refine this discussion based on your actual empirical results and the specific types of graphs you tested.)

7 Conclusion

In this laboratory work, we studied the greedy algorithm design technique and applied it to solve the Minimum Spanning Tree problem using Prim's and Kruskal's algorithms. We im-

plemented these two fundamental algorithms for finding the MST of a weighted undirected graph.

We explored the core principles of greedy algorithms and their application in the context of MSTs. Prim's algorithm builds the MST vertex by vertex, always adding the cheapest edge connecting to the current tree. Kruskal's algorithm builds the MST edge by edge, adding the cheapest edge that doesn't form a cycle.

The empirical analysis involved measuring the execution time of both algorithms on graphs with varying numbers of nodes. This allowed us to observe how their performance scales in practice and compare their efficiency. The results demonstrated that while both algorithms have comparable theoretical complexities, their practical performance can differ depending on the graph structure and implementation details (such as the choice of data structures like priority queues and Disjoint Set Unions).

This laboratory work provided practical experience in implementing greedy algorithms and performing empirical analysis, reinforcing the understanding that algorithm efficiency is influenced by both its theoretical complexity and the characteristics of the input data.

GitHub Repository Link: Dârzu Cătălin