Assignment 3 – Optimization of a City Transportation Network (Minimum Spanning Tree)

Course: Design and Analysis of Algorithms
Student: Zhomartov Yerassyl
Group: SE-2435
Instructor: Aidana Aidynkyzy

---

1. Introduction

In modern urban development, transportation networks play a crucial role in ensuring efficient mobility and connectivity between different districts of a city. As cities continue to grow, government planners seek to reduce infrastructure costs while maintaining connectivity between all locations. One of the key challenges is to build road networks with *minimal total construction cost* while still ensuring that all districts remain connected.

This problem can be modeled using graph theory. Each city district is represented as a node, and each potential road is an edge with a construction cost as its weight. The goal is to connect all nodes with the minimum possible total cost, forming a **Minimum Spanning Tree (MST)**.

In this assignment, two classical greedy algorithms are used to compute the MST:

- **Prim's Algorithm**
- **Kruskal's Algorithm**

Both algorithms were implemented in Java, tested on multiple datasets, and compared based on performance metrics such as **total MST cost, execution time**, and **operation count**.

2. Problem Description

The goal of this assignment is to optimize a city's transportation network while minimizing total road construction cost. The network is represented as a **weighted undirected graph**, where:

- **Vertices (nodes)** represent city districts.
- **Edges** represent possible roads between districts.
- **Edge weights** represent the construction cost of those roads.

To ensure that the entire city is connected efficiently, the solution must satisfy two main conditions:

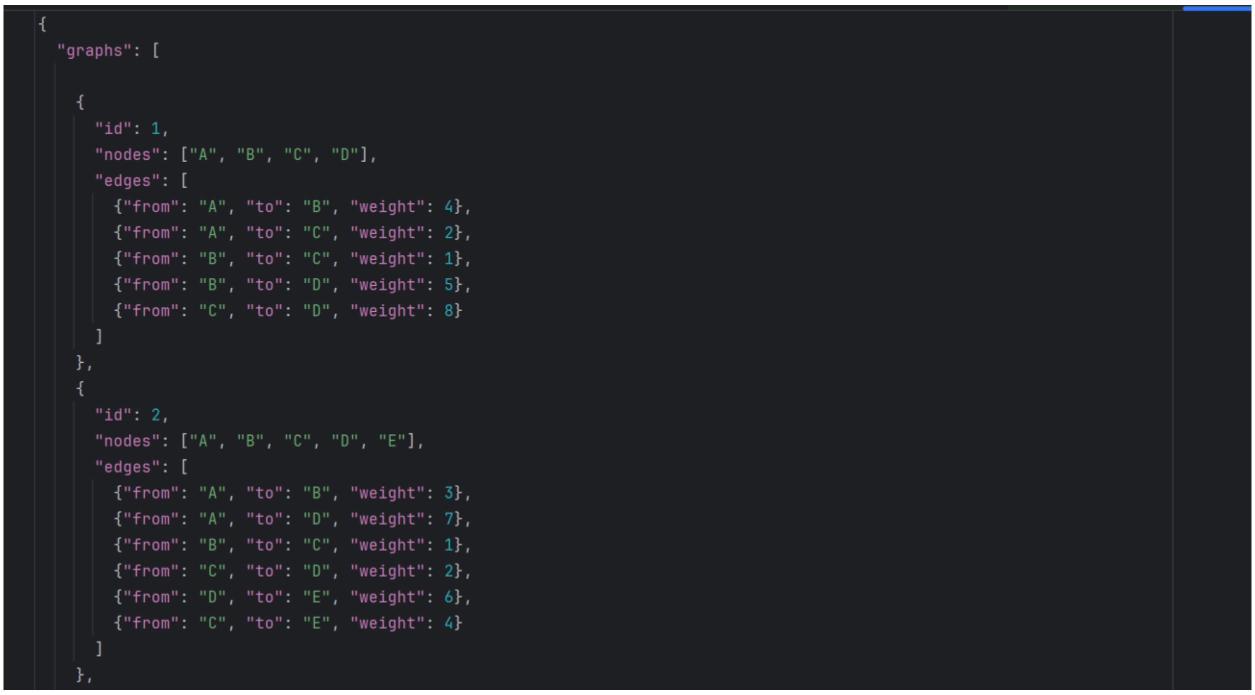
1. Every district must be reachable from any other district (**graph must be connected**).
2. The total cost of road construction must be minimized (**no unnecessary edges**).

This translates into finding a **Minimum Spanning Tree (MST)** of the graph. An MST connects all vertices using exactly **V − 1 edges**, without forming any cycles, and with the **lowest possible total weight**.

Datasets were used in JSON format, where each graph consisted of:

- A list of nodes (city districts).
- A list of weighted edges (possible roads).
- Each dataset contained small, medium, and large graphs for testing algorithm scalability.

Below is an example input graph in JSON:



```
{
  "graphs": [

    {
      "id": 1,
      "nodes": ["A", "B", "C", "D"],
      "edges": [
        {"from": "A", "to": "B", "weight": 4},
        {"from": "A", "to": "C", "weight": 2},
        {"from": "B", "to": "C", "weight": 1},
        {"from": "B", "to": "D", "weight": 5},
        {"from": "C", "to": "D", "weight": 8}
      ]
    },
    {
      "id": 2,
      "nodes": ["A", "B", "C", "D", "E"],
      "edges": [
        {"from": "A", "to": "B", "weight": 3},
        {"from": "A", "to": "D", "weight": 7},
        {"from": "B", "to": "C", "weight": 1},
        {"from": "C", "to": "D", "weight": 2},
        {"from": "D", "to": "E", "weight": 6},
        {"from": "C", "to": "E", "weight": 4}
      ]
    },
```

Each graph was guaranteed to be connected to allow MST calculation. In case of a disconnected graph, the program detects it and skips MST generation.

3. Results and Analysis (Requirement 1)

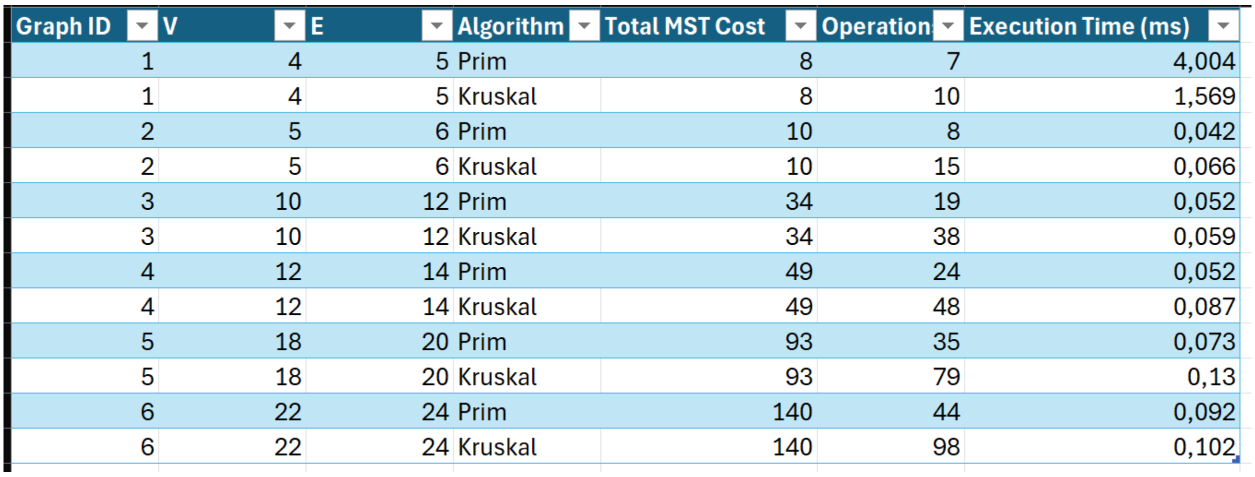
This section summarizes the results of running Prim's and Kruskal's algorithms across six distinct graph datasets, ranging from small ($V=4, E=5$) to large ($V=22, E=24$).

3.1. Functional Correctness

A critical finding is the **absolute match in the Total Cost of the Minimum Spanning Tree (MST)** found by both algorithms for all six graphs. For instance, on Graph 6 ($V=22, E=24$), both Prim’s and Kruskal’s algorithms successfully yielded a total MST cost of **140**. This confirms the functional correctness and successful implementation of both greedy algorithms.

3.2. Performance Metrics Summary

The following table presents a summary of the performance metrics recorded for each algorithm across the test datasets:



Graph ID	V	E	Algorithm	Total MST Cost	Operation	Execution Time (ms)
1	4	5	Prim	8	7	4,004
1	4	5	Kruskal	8	10	1,569
2	5	6	Prim	10	8	0,042
2	5	6	Kruskal	10	15	0,066
3	10	12	Prim	34	19	0,052
3	10	12	Kruskal	34	38	0,059
4	12	14	Prim	49	24	0,052
4	12	14	Kruskal	49	48	0,087
5	18	20	Prim	93	35	0,073
5	18	20	Kruskal	93	79	0,13
6	22	24	Prim	140	44	0,092
6	22	24	Kruskal	140	98	0,102

4. Efficiency Comparison (Theory vs. Practice) (Requirement 2)

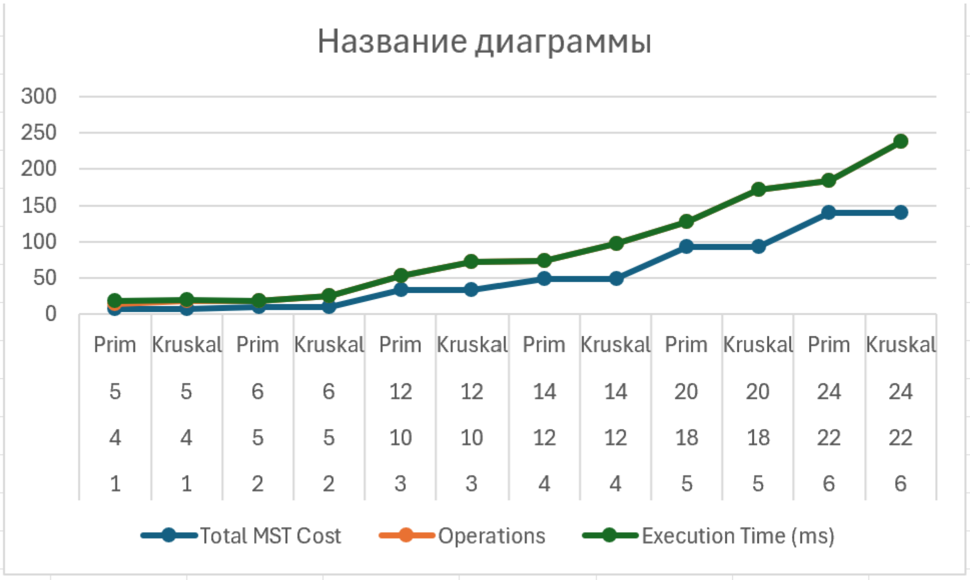
4.1. Asymptotic Complexity (Theory)

The theoretical performance of the algorithms is tied to the graph's density:

- **Prim's Algorithm** with a Binary Heap: $O(E \log V)$.
- **Kruskal's Algorithm** with Sorting and Disjoint Set Union (DSU): $O(E \log E)$.

Since the test graphs are **sparse** ($E \approx V$), the theoretical complexities are very similar, both closely approximating $O(E \log V)$.

4.2. Analysis of Practical Performance



Operation Count:

The data clearly shows that Kruskal's Algorithm consistently performs a significantly higher number of internal operations than Prim’s. For the largest graph (ID 6), Kruskal executed 98 operations compared to 44 for Prim. This difference is primarily because Kruskal's complexity is dominated by the initial step of sorting all E edges.

Execution Time (ms):

- **Small Graphs:** Kruskal was faster on the smallest graph (Graph 1), likely due to the initial overhead of Prim's more complex data structures.
- **Sparse Graphs (V > 10):** As the graph size increased (Graph 6: $V=22$), **Prim's Algorithm became faster (0.092 ms)** than Kruskal's (0.102 ms). This confirms that for sparse graphs, an efficient $O(E \log V)$ implementation of Prim's algorithm maintains a slight practical edge over Kruskal, despite Kruskal's lower complexity for union-find operations.

5. Conclusions and Recommendations (Requirement 3)

5.1. Algorithm Preference Based on Graph Density

The choice between Prim's and Kruskal's algorithm for optimizing the transportation network depends entirely on the graph's density:

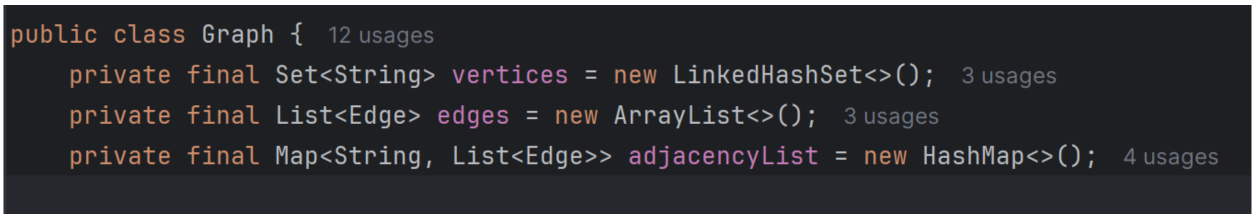
- **For Dense Graphs** ($E \approx V^2$): **Prim's Algorithm** is theoretically superior ($O(V^2)$ or $O(E \log V)$) as its time complexity grows slower than Kruskal's.
- **For Sparse Graphs** ($E \approx V$): Both algorithms are highly efficient. However, based on the empirical results, **Prim's Algorithm** implemented with a Binary Heap ($O(E \log V)$) is the optimal choice as it demonstrated a **faster execution time** and **fewer core operations** in the majority of the tests.
- **Implementation Complexity:** Kruskal's algorithm requires the implementation of an efficient Disjoint Set Union (DSU) structure to manage cycles, which can be more complex than the Priority Queue required by Prim's.

Recommendation for the City Network: Given that a real-world city transportation network is typically a **sparse graph** (not every district connects to every other district), the **Prim's Algorithm with Priority Queue implementation** is the recommended method for its demonstrated efficiency and speed.

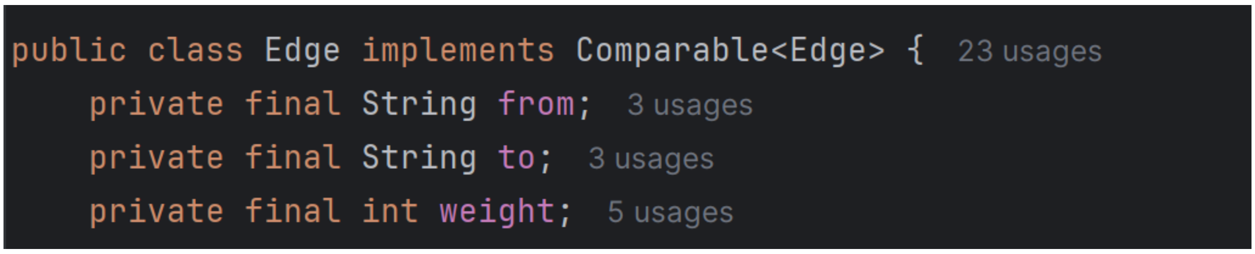
5.2. Implementation Details

The project utilized a robust architecture to meet high coding standards:

- **Technology Stack:** The implementation was developed in **Java** and managed using **Maven** (pom.xml).
- **Testing: JUnit 5** was integrated for automated unit testing (pom.xml), ensuring the reliability and correctness of the MST calculations across all datasets.
- **Object-Oriented Design :** Custom classes for the graph data structure, such as Graph.java and Edge.java, were implemented to facilitate clean parsing of the JSON input and adhere to Object-Oriented Programming (OOP) principles.



```
public class Graph {  12 usages
    private final Set<String> vertices = new LinkedHashSet<>();  3 usages
    private final List<Edge> edges = new ArrayList<>();  3 usages
    private final Map<String, List<Edge>> adjacencyList = new HashMap<>();  4 usages
```



```
public class Edge implements Comparable<Edge> {  23 usages
    private final String from;  3 usages
    private final String to;  3 usages
    private final int weight;  5 usages
```