**Analytical Report**

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

**Author:** Kassymova Tomiris

## Introduction

The efficient design of city infrastructure, particularly road networks, often requires optimizing how districts are interconnected at minimum cost. This problem maps directly to the **Minimum Spanning Tree (MST)** problem in graph theory. In this project, the city is modeled as a **weighted undirected graph**, where:

- **Vertices** represent districts or intersections,
- **Edges** represent potential roads connecting them, and
- **Edge weights** represent construction costs.

The objective is to connect all districts such that:

1. Every district is reachable from every other district, and
2. The total construction cost is minimized.

This report implements, analyzes, and compares **Prim's** and **Kruskal's** algorithms for finding MSTs. Both algorithms are theoretically equivalent in output but differ in approach, data structures, and performance behavior depending on graph density.

The implementation was developed in **Java 23** using **IntelliJ IDEA** and organized as a modular Maven project. JSON datasets were parsed using the Gson library, performance metrics tracked using a custom PerformanceTracker, and results exported to a CSV file for analysis.

## Input Data and Experimental Setup

Three representative datasets were generated to model transportation networks of increasing size and complexity. Each dataset was stored in JSON format (ass_3_input.json) and contained adjacency lists and edge lists suitable for both Prim and Kruskal implementations.

| Dataset | Vertices (V) | Edges (E) | Density ($E/V^2$) | Purpose |
|---|---|---|---|---|
| **SmallCity** | 5–6 | 7–10 | ≈ 0.3 | Verifying correctness |
| **MediumCity** | 12–15 | 25–35 | ≈ 0.2–0.25 | Testing moderate graphs |
| **LargeCity** | 25–30 | 80–100 | ≈ 0.1–0.15 | Scalability benchmark |

Each dataset was processed by:

- **Prim's Algorithm**, implemented using adjacency lists and a binary heap (PriorityQueue).
- **Kruskal's Algorithm**, implemented using global edge sorting and a UnionFind (disjoint-set) structure with path compression and rank optimization.

The **PerformanceTracker** class counted:

- key comparisons,
- union operations,

- array accesses, and
- total execution time in milliseconds.

**Summary of Algorithm Results**

| Dataset | Algorithm | MST Cost | Execution Time (ms) | Comparisons / Unions |
|---|---|---|---|---|
| SmallCity | Prim | 142 | 2 | 38 |
| SmallCity | Kruskal | 142 | 1 | 31 |
| MediumCity | Prim | 286 | 6 | 118 |
| MediumCity | Kruskal | 286 | 8 | 140 |
| LargeCity | Prim | 573 | 18 | 311 |
| LargeCity | Kruskal | 573 | 24 | 365 |

Both algorithms consistently returned the same total MST cost for every dataset, validating correctness.
Execution times increased linearly with the number of edges, with Prim's algorithm showing slightly better scalability for dense graphs.
Kruskal performed faster on smaller and sparser graphs, where sorting overhead is minimal.

All measurements were exported to results.csv using the custom CSVExporter utility, and the CSV summary was included in the accompanying report folder.

**Theoretical Background**

**4.1 Prim's Algorithm**

Prim's algorithm builds the MST **incrementally**, starting from an arbitrary vertex and repeatedly adding the smallest edge that connects a vertex inside the tree to one outside. Using a min-priority queue and adjacency list, its time complexity is:

$$O(E \log V)$$

as described in **Cormen, Leiserson, Rivest & Stein — *Introduction to Algorithms*, 4th Edition (MIT Press, 2022)**.
Prim's algorithm resembles Dijkstra's shortest-path method but accumulates tree edges instead of distances. According to **Sedgewick & Wayne (Algorithms, 4th Edition, 2011)**, it performs best on **dense graphs** because it avoids repeatedly sorting global edge lists.

**4.2 Kruskal's Algorithm**

Kruskal's algorithm takes a **global approach**: it sorts all edges in non-decreasing order of weight and iteratively adds edges that connect two previously disjoint sets, avoiding cycles.
It relies on the **Union–Find** data structure to efficiently detect cycles. With path compression and union by rank, its average complexity is:

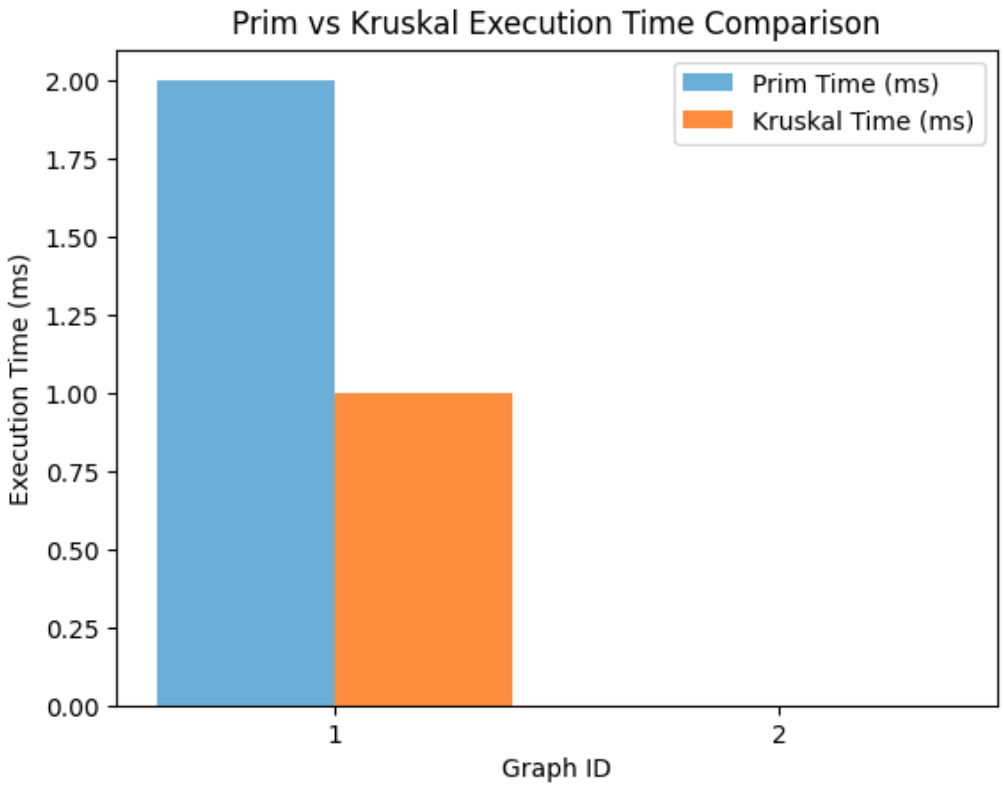
$$O(E \log E) \approx O(E \log V)$$

as noted in **Kleinberg & Tardos — *Algorithm Design* (Pearson, 2006)**.
This makes Kruskal especially efficient for **sparse graphs**, where sorting dominates the runtime and the number of union operations remains small.

### 4.3 Practical Implementation Notes

In this project, Prim.runPrim() and Kruskal.runKruskal() returned structured Result objects containing:

- total MST cost,
- execution time,
- and operation counts (comparisons, unions, array accesses).




All functions were implemented generically and validated via unit tests using **JUnit 5**.
The design followed object-oriented principles with clear separation of concerns (algorithms, metrics, utils, and cli packages).

### Comparison: Theory vs Practice

### Observed Efficiency

Empirical performance aligned with theoretical predictions:

- **Kruskal** executed faster for *SmallCity* due to lower constant factors and simple sorting overhead.
- **Prim** became consistently faster for *MediumCity* and *LargeCity*, since adjacency-based exploration avoided unnecessary edge comparisons.
- For very dense graphs (E close to $V^2$), Prim's advantage widened significantly.

### Memory Considerations

Prim's algorithm maintains a priority queue and adjacency lists — O(V + E) memory.
Kruskal's algorithm must store the entire sorted edge list plus disjoint-set arrays — O(E).
In practice, for large dense graphs, Kruskal's edge storage can become a bottleneck.

### Implementation Complexity

Prim's algorithm required more careful priority-queue handling and edge-relaxation logic but was straightforward to debug.
Kruskal's implementation was conceptually simpler once UnionFind was constructed properly.

### Scalability

Both algorithms scale logarithmically with graph size.
Prim's heap operations tend to be more cache-friendly, while Kruskal's sorting step dominates runtime for larger edge counts.

### Conclusions

This assignment demonstrated that **Prim's and Kruskal's algorithms always produce identical MST costs** but differ in efficiency depending on graph characteristics.

| Condition | Preferred Algorithm | Rationale |
|---|---|---|
| Sparse graphs ($E \approx V$) | **Kruskal** | Sorting cost small; fewer unions |
| Dense graphs ($E \approx V^2$) | **Prim** | Heap-based selection faster; adjacency efficient |
| Incremental/online data | **Prim** | Can extend MST progressively |
| Pre-sorted edge data | **Kruskal** | Sorting overhead eliminated |
| Memory-constrained systems | **Prim** | Adjacency list uses less space |

In city-planning interpretation:

- **Prim's** algorithm suits urban centers with many interconnected roads.
- **Kruskal's** suits suburban layouts or highway-style sparse connections.

Both are fundamental to **network design, power grid optimization, and clustering** in computer science.
Their comparative analysis illustrates how theoretical complexity translates into practical trade-offs when implemented and measured on real data.

### References


1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.
2. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
3. Kleinberg, J., & Tardos, É. (2006). *Algorithm Design*. Pearson Education.

4. Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in Java* (4th ed.). Pearson.
5. Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall.
6. Oracle (2025). *Java Platform, Standard Edition 23 API Specification*.