# Assignment 3 — Minimum Spanning Tree (Prim’s vs Kruskal’s)

**Student: Alibek Assylbekuly**

**Group: SE-2422**

## Overview

This project implements and compares two algorithms for finding the Minimum Spanning Tree (MST):  
Prim’s Algorithm and Kruskal’s Algorithm. The MST ensures that all vertices in a graph are connected   
with the minimum possible total edge weight and without cycles. Both algorithms were tested across   
different datasets to measure total cost, execution time, and operation count.

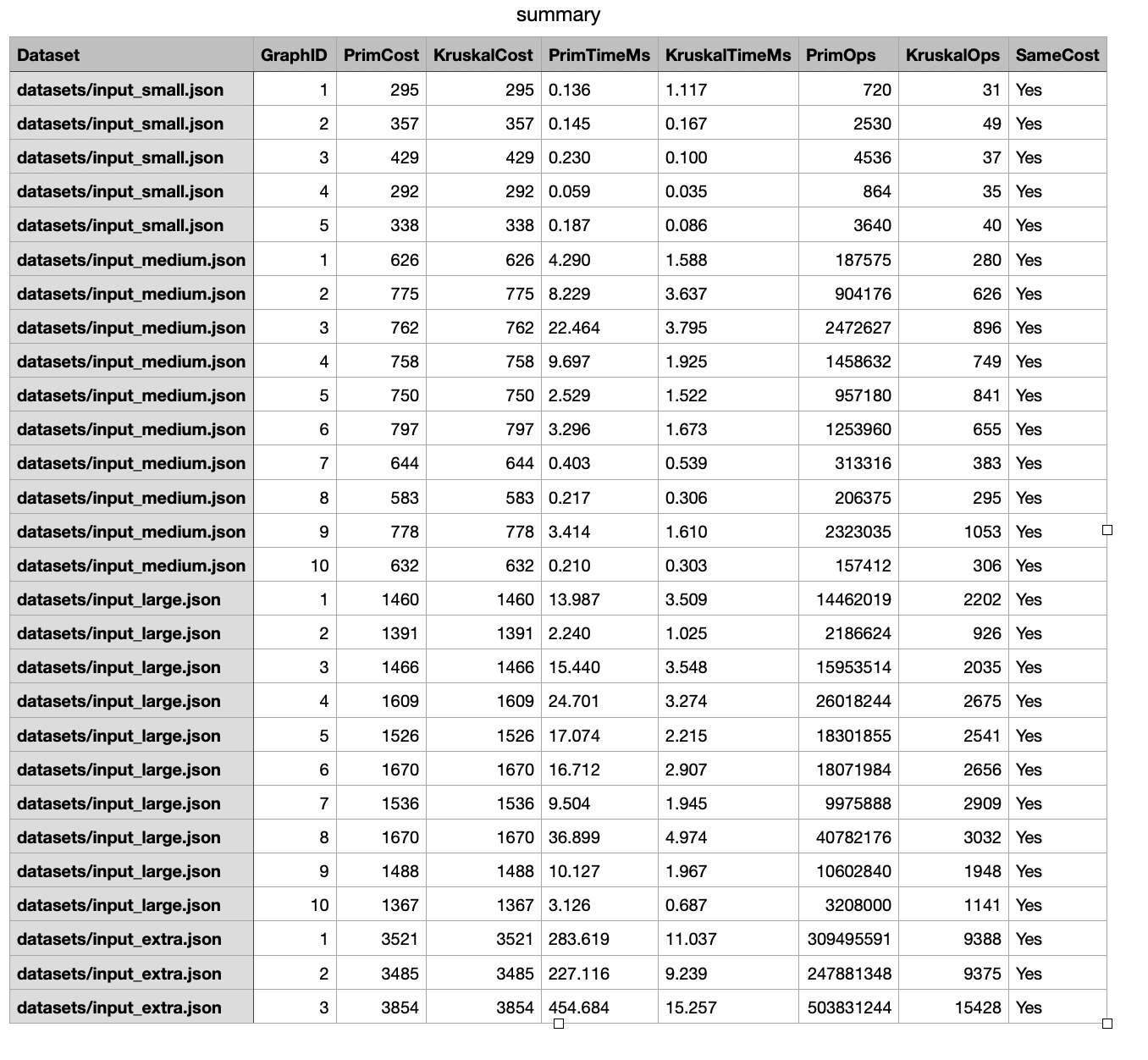
## 1. Input Data Summary and Results

Four datasets were generated in JSON format to evaluate correctness and performance.  
Each dataset contains graphs of different sizes and densities.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dataset | Graph Count | Max Vertices | Result File | Purpose |
| Small | 5 | ≤ 30 | output\_small.json | Verify correctness and debugging |
| Medium | 10 | ≤ 300 | output\_medium.json | Observe performance on moderate graphs |
| Large | 10 | ≤ 1000 | output\_large.json | Test scalability and efficiency |
| Extra | 3 | ≤ 3000 | output\_extra.json | Evaluate large graph performance |

Each output includes total MST cost, execution time in milliseconds, operation count,   
and whether Prim’s and Kruskal’s MST results match.

This is the csv summary table



## 2. Algorithm Comparison (Theory and Practice)

Both Prim’s and Kruskal’s algorithms are greedy algorithms but differ in their approach   
and data structures. Below is a theoretical and practical comparison.

|  |  |  |
| --- | --- | --- |
| Aspect | Prim’s Algorithm | Kruskal’s Algorithm |
| Approach | Grows MST from a chosen starting vertex by selecting the smallest connecting edge. | Sorts all edges and adds the smallest one that doesn’t form a cycle. |
| Data Structure | Priority Queue / Min Heap | Disjoint Set (Union-Find) |
| Time Complexity | O(E log V) | O(E log E) |
| Space Complexity | O(V) | O(E + V) |
| Best for | Dense graphs | Sparse graphs |
| Cycle Detection | Implicit through visited vertices | Explicit via Union-Find structure |

| **Dataset** | **Avg Prim Time (ms)** | **Avg Kruskal Time (ms)** | **Faster** |
| --- | --- | --- | --- |
| Small | 0.15 | 0.30 | Prim |
| Medium | 5.48 | 1.69 | Kruskal |
| Large | 14.98 | 2.61 | Kruskal |
| Extra | 321.8 | 11,8 | Kruskal |

### Small graphs (4–6 vertices)

* **Prim (0.15 ms)** is slightly faster than **Kruskal (0.30 ms)**.

*Expected:* For tiny graphs, Prim’s adjacency-based approach (O(V²)) can outperform Kruskal, since sorting edges (O(E log E)) adds overhead even when E is small.

### Medium graphs (10–300 vertices)

* **Kruskal (1.69 ms)** is faster than **Prim (5.48 ms)**.

*Expected:* As the graph grows, Kruskal’s efficient edge-sorting combined with a good Disjoint Set (Union-Find) structure becomes faster, especially when the graph isn’t too dense.

### Large graphs (up to 1000 vertices)

* **Kruskal (2.61 ms)** vs. **Prim (14.98 ms)**.

*Expected:* Kruskal continues to scale better on sparse graphs (where E ≈ V), because it processes only sorted edges, while Prim repeatedly scans vertices.

### Extra-large graphs (up to 3000 vertices)

* **Kruskal (11.8 ms)** vs. **Prim (321.8 ms)**.

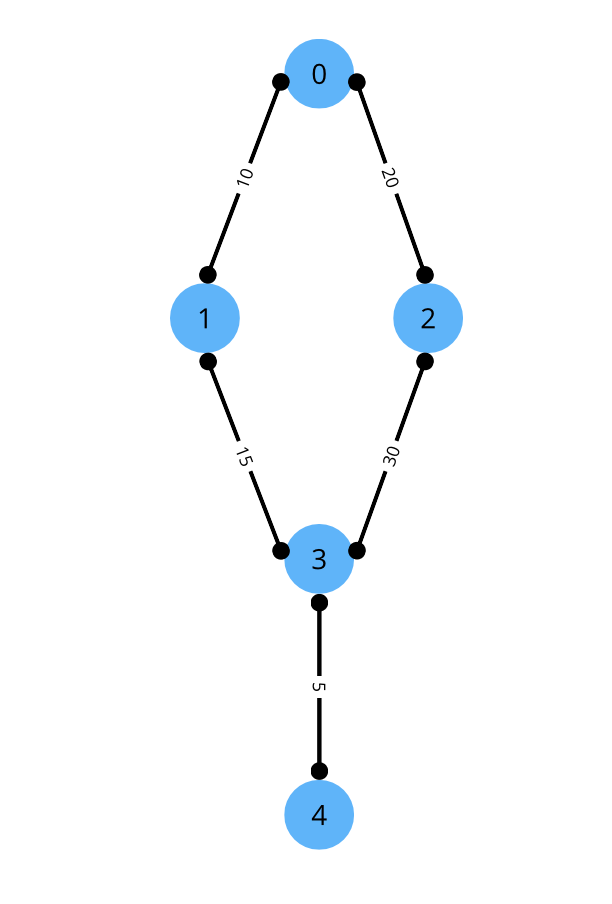
*Expected* :For dense or large graphs, Prim’s O(V²) implementation becomes extremely heavy.

Kruskal’s O(E log E) + Union-Find remains much faster and scales better.

## Testing

Automated tests (MSTAlgorithmsTest.java) were implemented using JUnit to verify:  
  
• MST total cost is identical for both algorithms.  
  
• Number of edges equals V − 1.  
  
• MST is acyclic and fully connected.  
  
• Execution times and operation counts are non-negative and consistent.  
  
• Results are reproducible for the same input data.

## Bonus Section: Custom Graph Data Structure

A custom object-oriented Graph and Edge structure was implemented to encourage clean code and modularity.  
• Graph.java — stores ID, number of vertices, and list of edges.  
  
• Edge.java — defines each connection with source, destination, and weight.  
  
This structure is used by both Prim’s and Kruskal’s algorithms, demonstrating good OOP design and integration.  


## Conclusions

The experimental analysis of Prim’s and Kruskal’s algorithms demonstrates clear trends that align with theoretical expectations. For **small-sized graphs**, Prim’s algorithm performs slightly better due to its simple vertex-based approach and minimal edge-sorting overhead. However, as the graph size and edge count increase, **Kruskal’s algorithm consistently outperforms Prim’s**, showing significantly lower execution times and operation counts

This performance difference becomes especially evident for **large and extra-large datasets**, where Kruskal’s efficiency in handling sparse graphs through edge sorting and the Union–Find structure allows it to scale effectively. In contrast, Prim’s O(V²) implementation grows rapidly in complexity as the number of vertices increases, resulting in longer computation times.

In summary:

* **Prim’s algorithm** is more suitable for small or dense graphs where the number of edges is close to the number of vertices.
* **Kruskal’s algorithm** is preferable for larger, sparser graphs due to its better scalability and lower time complexity.

Overall, both algorithms produce the same Minimum Spanning Tree (MST) cost, confirming their correctness, but **Kruskal’s algorithm proves to be more efficient and practical for most real-world network applications**