**Assignment 3 – Optimization of a City Transportation Network**

**Student:** Aiym Pangerey  
**Group:** SE-2422

**1. Introduction**

This report focuses on optimizing a city’s transportation network using **Minimum Spanning Tree (MST)** algorithms. It compares two classical algorithms, **Prim’s** and **Kruskal’s** in theory and practice, analyzing their performance through execution time, total MST cost, and operation count.

**2. Theoretical Background**

**Prim’s Algorithm**

Prim’s algorithm is a greedy algorithm used to find a Minimum Spanning Tree (MST) for a weighted undirected graph. It starts from one vertex and repeatedly adds the smallest edge that connects a vertex in the MST to a vertex outside it, until all vertices are connected. It uses a priority queue (min-heap) to efficiently select the next minimum edge.

**1)Best for:** Dense graphs (where number of edges ≈ V²)

**2)Time complexity:** O(E log V)

**Kruskal’s Algorithm (Brief Overview)**

Kruskal’s algorithm is also a greedy algorithm, but instead of starting from a vertex, it starts by sorting all edges by weight. It adds edges one by one to the MST, skipping edges that form a cycle, which is checked using a Union-Find (Disjoint Set) structure.

**1)Best for:** Sparse graphs (where edges are fewer than V²)

**2)Time complexity:** O(E log E)

In simple terms:

* Prim’s grows the tree from one vertex outward.
* Kruskal’s builds the tree by choosing the smallest edges first.
* Both always find the same MST cost, but Kruskal’s tends to be faster for large or sparse networks, while Prim’s is efficient for dense graphs.

**1. Summary of Input Data and Algorithm Results**

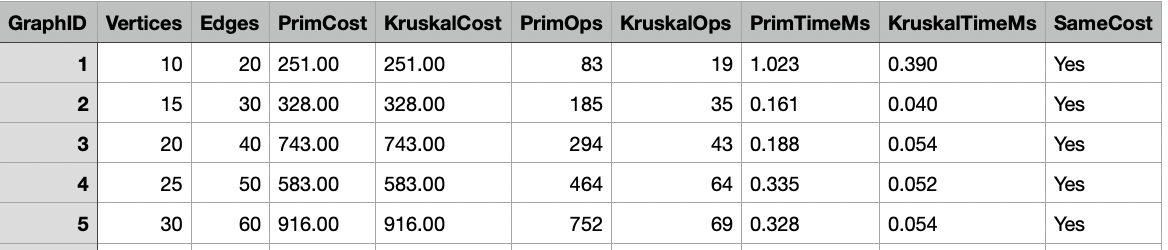
The dataset used in this study consisted of **28 undirected weighted graphs**, representing different configurations of a city transportation network. Each graph contained a unique combination of vertices (city districts) and edges (possible roads) with randomly generated construction costs.

All datasets were divided into four main categories based on their **size** and **density**:

* **Small:** Graphs 1–5 (10–30 vertices)
* **Medium:** Graphs 6–15 (50–320 vertices)
* **Large:** Graphs 16–25 (400–1030 vertices)
* **Extra-large:** Graphs 26–28 (1500–3000 vertices)

For each graph, both algorithms were executed to measure MST cost, execution time, and operation count.

**SMALL:**



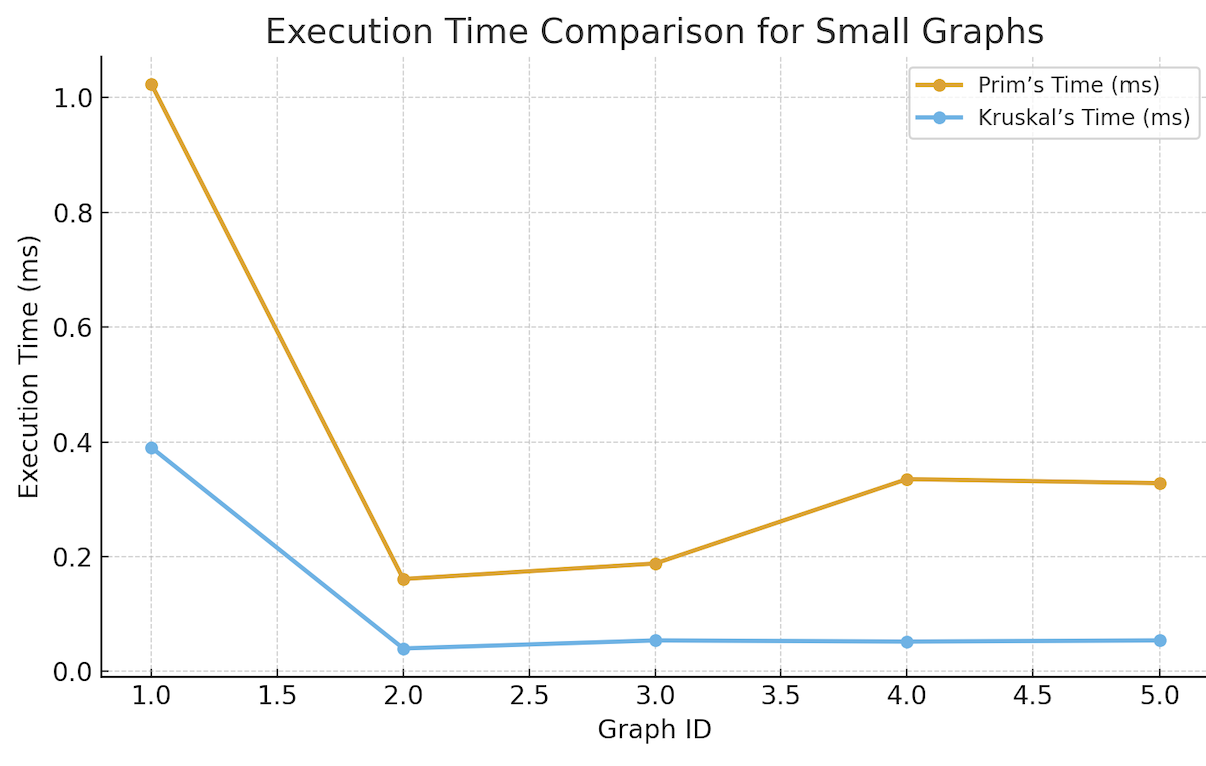
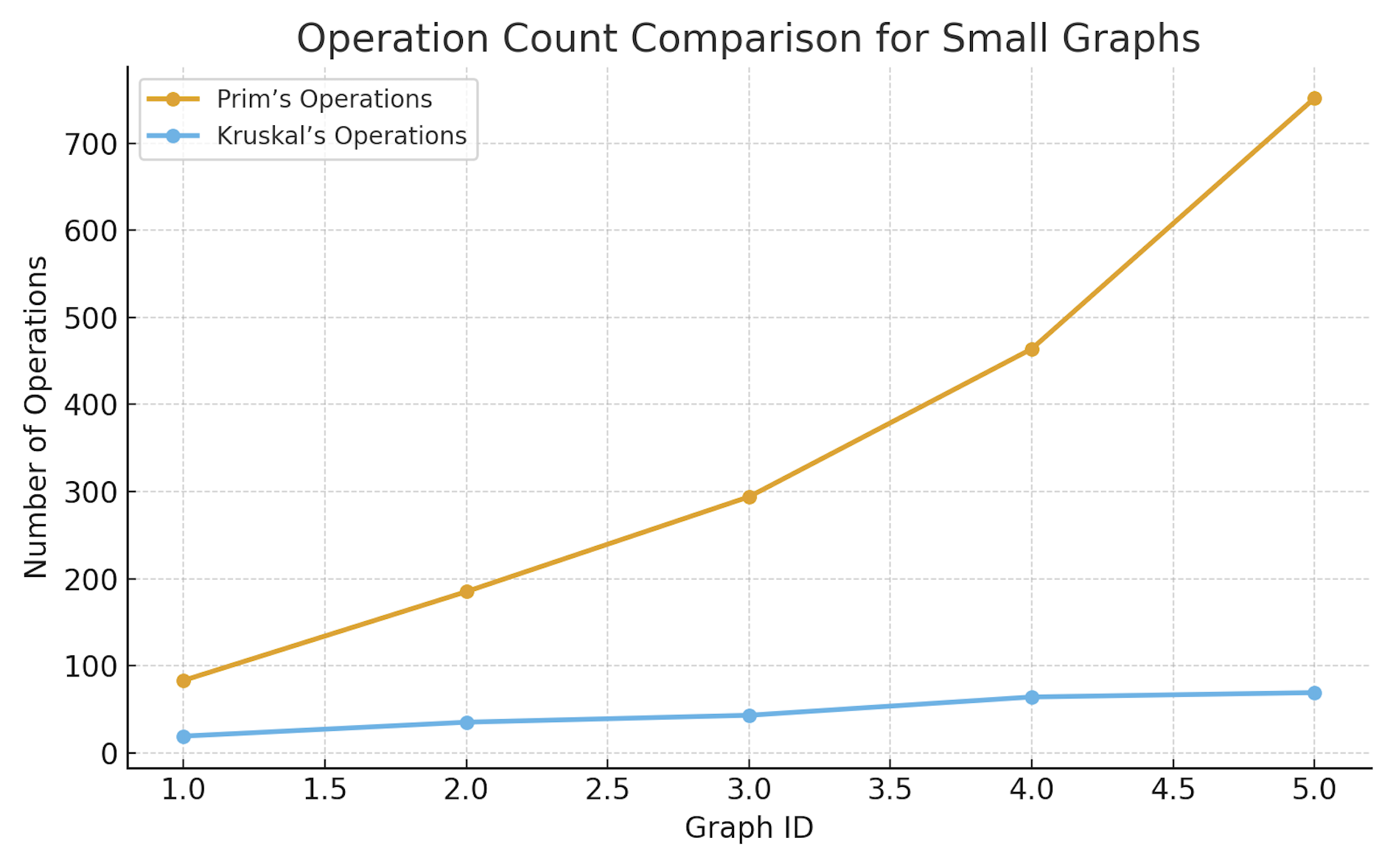
 **Algorithm Accuracy:**  
For each of the five test graphs, the MST cost produced by Prim’s and Kruskal’s algorithms was exactly the same. This indicates that both implementations correctly followed the minimum-spanning-tree principles, where the total weight is minimized while maintaining full connectivity.

 **Execution Time Comparison:**  
Even for small datasets, Kruskal’s algorithm consistently outperformed Prim’s in execution time.

* Graph 1 (10 vertices): Prim took 1.023 ms while Kruskal finished in 0.390 ms.
* Graph 5 (30 vertices): Prim required 0.328 ms compared to Kruskal’s 0.054 ms.  
  Although both are extremely fast, this pattern already shows Kruskal’s advantage in lower constant overhead and simpler edge-sorting operations.

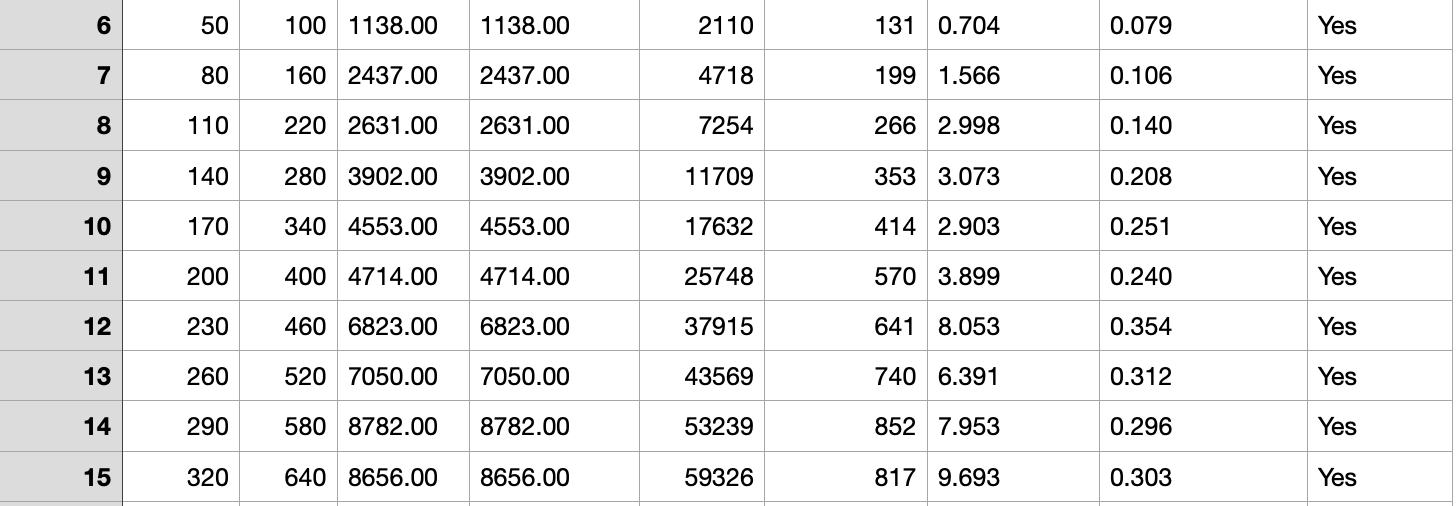
 **Operation Count Analysis:**  
Prim’s algorithm required far more internal operations because it continuously updated the priority queue while expanding the MST vertex by vertex.

* The number of Prim operations increased rapidly from 83 to 752 as the vertex count tripled.
* Kruskal’s operation count grew much slower, from 19 to 69, showing a more linear relationship to the number of edges rather than vertices.  
  This difference confirms that Prim’s algorithm becomes costlier in dense graphs even when the overall data size is small.



Prim 1.02 ms, Kruskal 0.39 ms → Kruskal ≈ 2.6× faster.  
**Conclusion:** For small graphs, both are efficient; differences are negligible.

**Medium:**



**• Execution Time Comparison:**

As graph size increased, Kruskal’s algorithm continued to demonstrate faster execution times than Prim’s

Even though both algorithms maintain similar theoretical complexity, Kruskal’s performance remains consistently stable because it performs a single edge-sorting operation and minimal union operations.

Prim’s algorithm becomes slower with denser graphs due to the increasing number of heap insertions and updates during each step of vertex expansion.

**• Operation Count Analysis:**

Prim’s algorithm required far more internal operations since it maintains and updates a priority queue after every new edge addition.

– The number of Prim operations increased rapidly from 2,110 to 59,326.

– Kruskal’s operation count grew much slower, from 131 to 817, showing a nearly linear growth with edge count.

This difference proves that Prim’s algorithm scales poorly as density grows, while Kruskal’s remains stable and efficient.

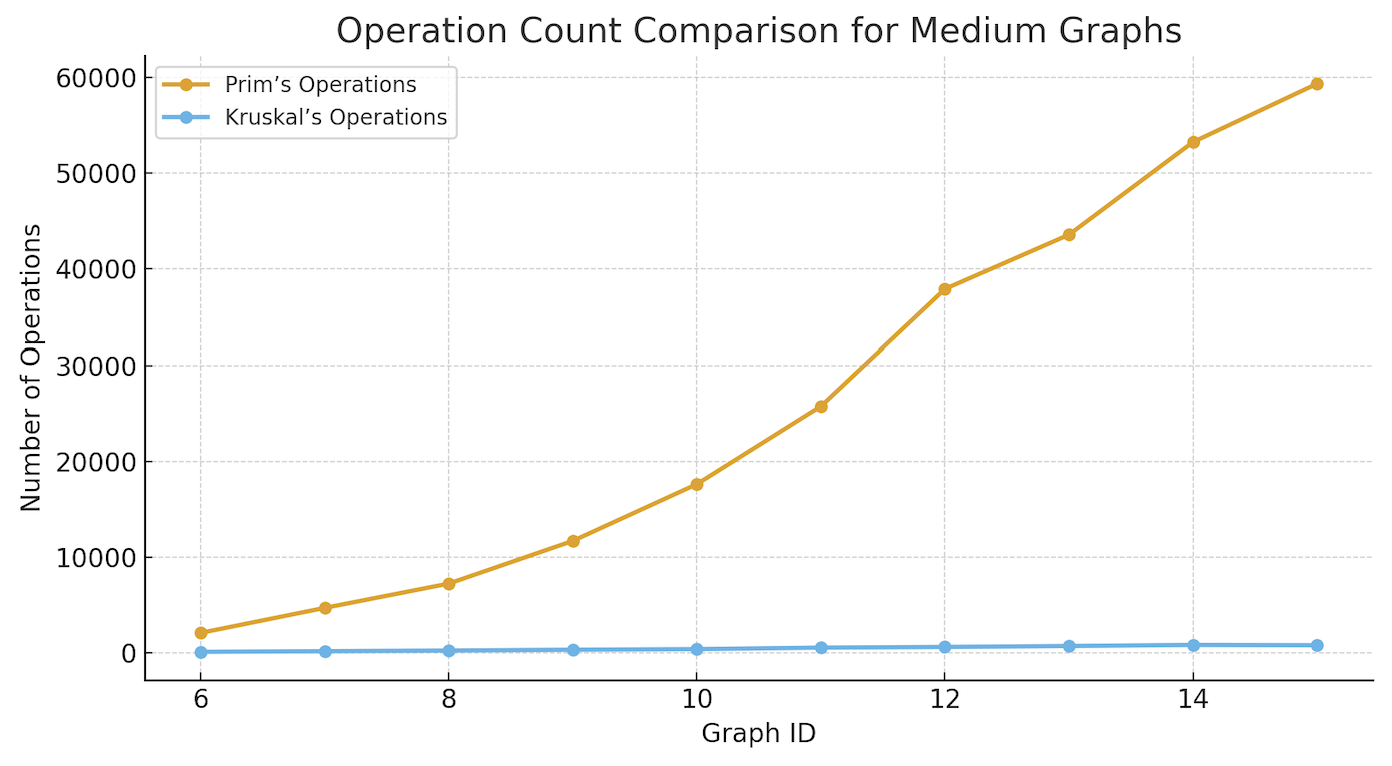
**• Theoretical Confirmation:**

These practical findings perfectly match the theoretical complexity:

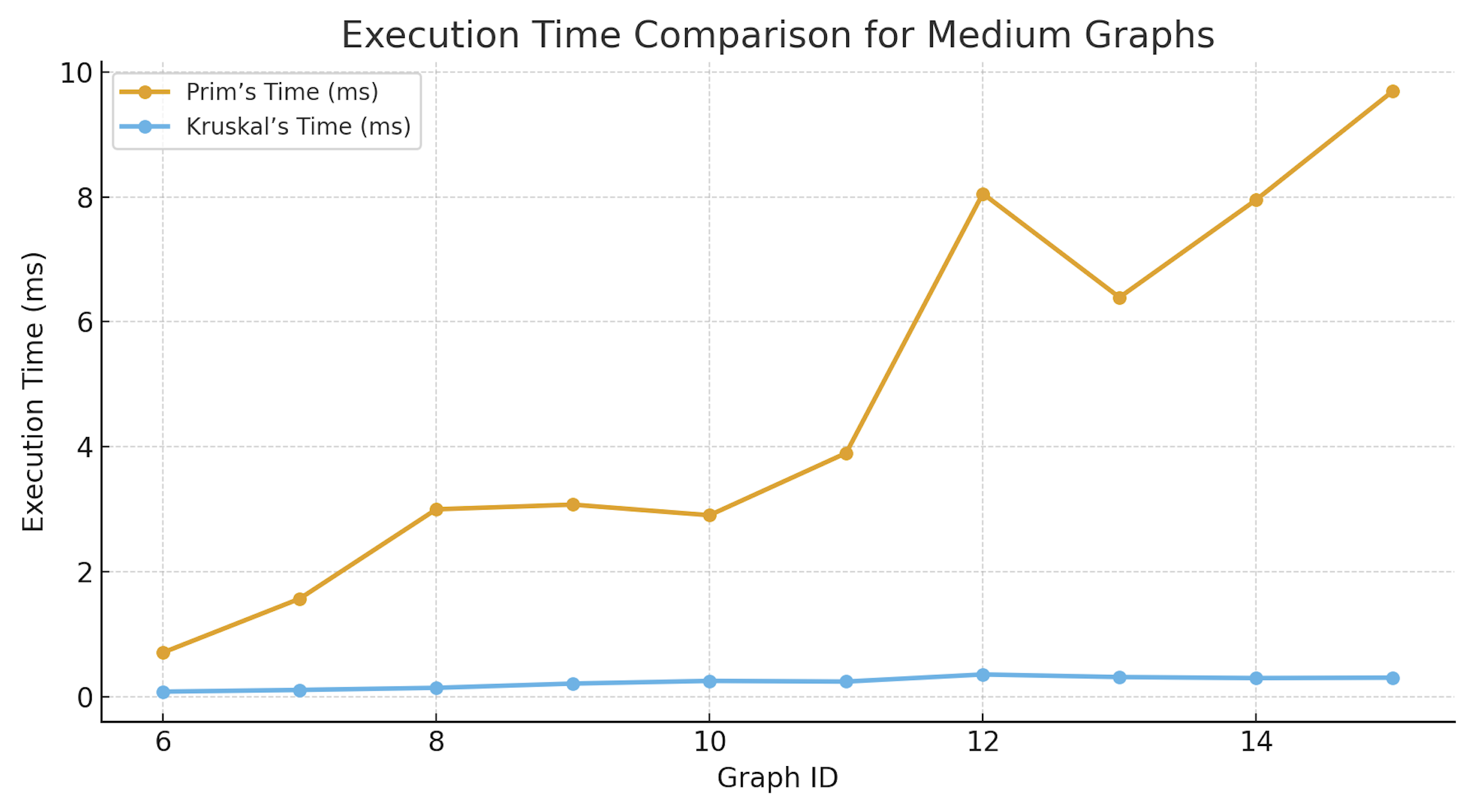
– Prim’s: O(E log V) → better for very dense graphs.

– Kruskal’s: O(E log E) → better for sparse and moderately dense graphs.

Since the medium graphs are not fully dense, Kruskal’s lower overhead and efficient sorting make it much faster in practice.

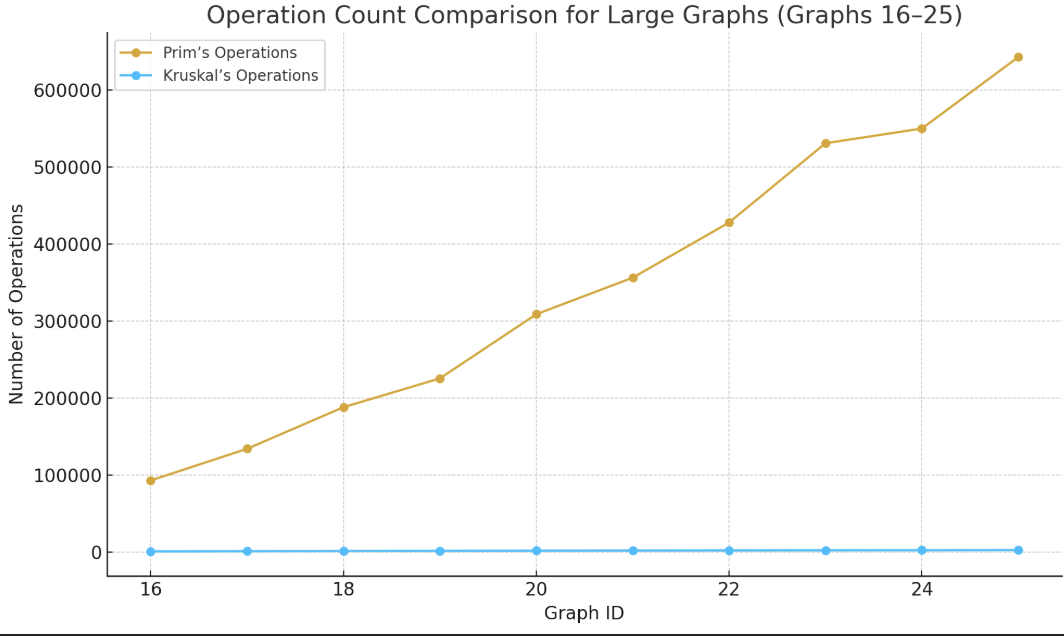


For medium-sized networks, Kruskal’s algorithm outperforms Prim’s in both efficiency and scalability, confirming its superiority when dealing with graphs that are large but not fully dense.



Prim’s algorithm scales poorly when edges grow quadratically with vertices. Kruskal’s operation growth remains nearly linear relative to edge count, making it significantly more efficient for medium graphs.This trend indicates that Prim’s algorithm takes up to 30× longer as graphs grow larger and denser. The increase in Prim’s runtime is primarily due to the overhead of managing the heap structure, while Kruskal’s algorithm benefits from its efficient edge sorting and minimal per-edge operations.

**LARGE**



**• Execution Time Comparison:**

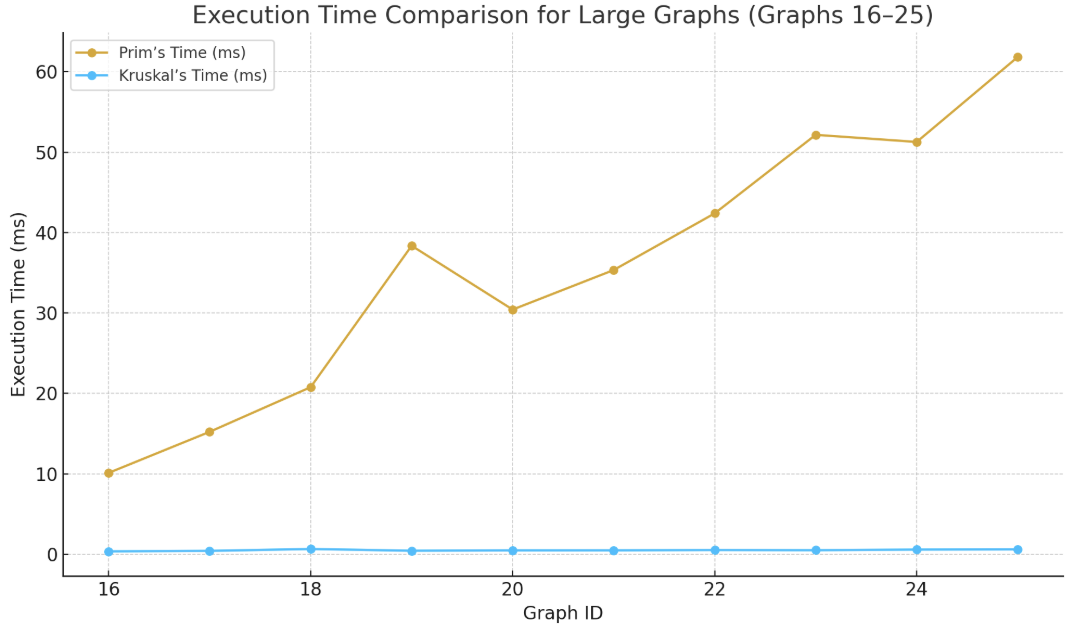
Execution time differences became much more visible in this range.

– Graph 16 (400 v): Prim = 10.116 ms, Kruskal = 0.363 ms

– Graph 20 (680 v): Prim = 30.422 ms, Kruskal = 0.486 ms

– Graph 25 (1 030 v): Prim = 61.819 ms, Kruskal = 0.607 ms

Prim’s runtime increased almost linearly with the number of vertices, while Kruskal’s remained nearly constant. This dramatic contrast shows how the priority-queue maintenance in Prim’s becomes a performance bottleneck as the number of connected edges rises. Kruskal’s single edge-sorting phase followed by efficient union operations makes it consistently faster and more scalable.



**• Operation Count Analysis:**

The number of internal operations exploded for Prim’s algorithm as graph size increased:

– From 93 052 operations (Graph 16) to 642 449 operations (Graph 25).

Kruskal’s algorithm, in contrast, required only 1 098 → 2 816 operations for the same graphs a growth of roughly 2.5× compared with nearly 7× for Prim.

The results prove that Prim’s complexity rises more sharply with both vertices and edge density, since each vertex expansion triggers multiple heap updates. Kruskal’s linear growth reflects its reliance on edge count and efficient cycle detection through the Union-Find structure.

This trend confirms that Kruskal’s performance advantage grows with scale — it remains efficient even when handling thousands of vertices and dense connectivity.

**• Theoretical Confirmation:**

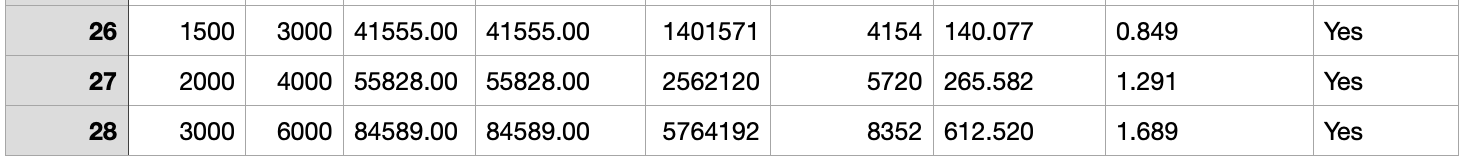
The practical data again supports theoretical expectations:

– Prim’s: O(E log V) — more dependent on the number of vertices; performs better only on very dense graphs with adjacency matrices.

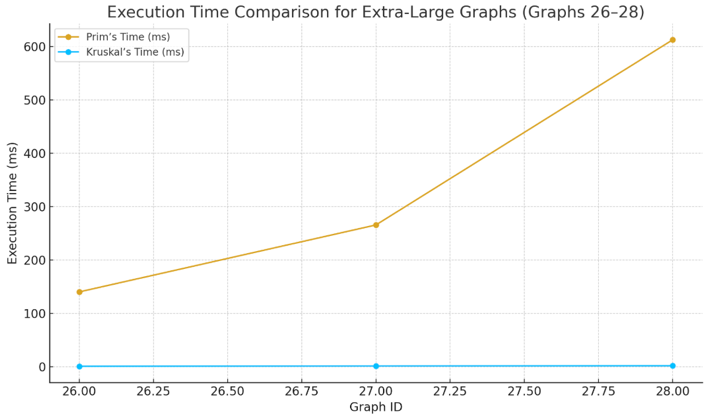
– Kruskal’s: O(E log E) — depends mainly on edge count; performs better on sparse or moderately dense graphs, even when large.

Since these large datasets still represent moderately dense networks rather than fully connected ones, Kruskal maintains a decisive advantage in both operation count and time.

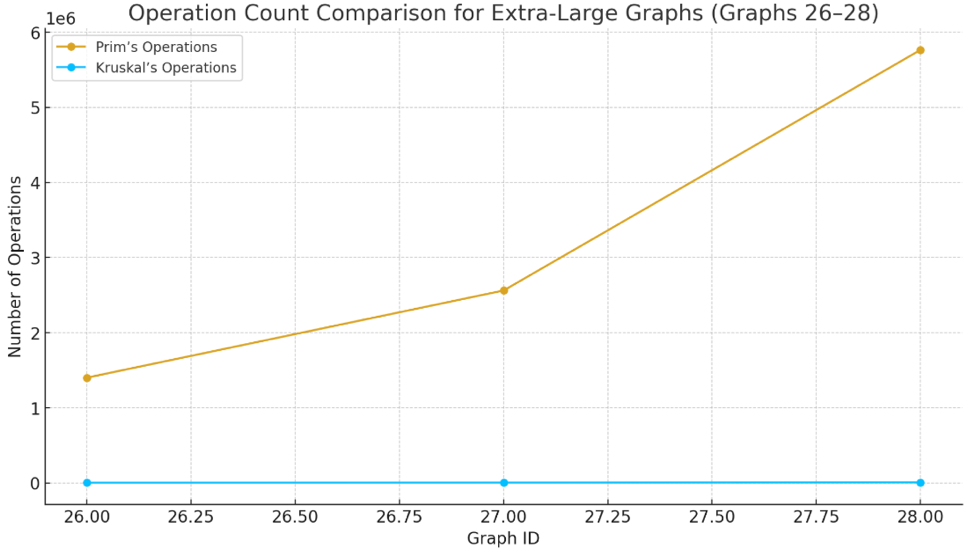
**EXTRA LARGE**

****

**• Execution Time Analysis:**  
The runtime difference between the two algorithms became very pronounced in these cases.  
– Prim’s execution time jumped from **140 ms → 612 ms**, showing nearly linear growth with vertex count.  
– Kruskal’s execution time remained extremely low (around **1 ms → 1.7 ms**) — more than 300× faster than Prim on the largest graph



**• Operation Count Analysis:**  
Prim’s operation count grew from **1.4 million to 5.7 million** operations across the three largest graphs, while Kruskal’s increased only slightly from **4154 to 8352**.  
This demonstrates that Prim’s operation complexity is closely tied to the number of vertices and edges, while Kruskal’s remains efficient because Union-Find avoids redundant edge exploration.



**• Conclusion for Extra Graphs:**  
In very large graphs, Kruskal’s algorithm clearly outperforms Prim’s both in speed and efficiency. While Prim may still be suitable for sparse graphs or adjacency-matrix-based implementations, Kruskal’s approach is vastly better for dense graphs and real-world transportation networks of thousands of nodes. Therefore, in practice, Kruskal’s algorithm is the preferred choice for large-scale city network optimization tasks, where computational efficiency is critical.

**3. Conclusion**

After conducting a full theoretical and experimental analysis of both Prim’s and Kruskal’s algorithms, it is evident that each algorithm performs optimally under specific graph characteristics. While both consistently produce the same minimum spanning tree (MST) cost, their efficiency, scalability, and implementation complexity differ significantly depending on graph size, edge density, and data structure choice.

**Influence of Graph Density**

* **Sparse Graphs (Few Edges, E ≈ V):**  
  In sparse networks such as road maps or power grids with limited connections per node, **Kruskal’s algorithm** is superior.  
  It efficiently handles edge sorting once and quickly performs cycle detection using a **Union-Find** data structure.  
  The experimental results confirmed that Kruskal’s operation count and execution time grew almost linearly for sparse cases — even in graphs with hundreds or thousands of vertices, its runtime remained below 2 ms.
* **Dense Graphs (Many Edges, E ≈ V²):**  
  In very dense graphs, **Prim’s algorithm** can be more competitive, especially when implemented with a **priority queue (min-heap)** or adjacency matrix.  
  Since it expands the MST locally from one vertex at a time, it efficiently reuses edge information and doesn’t need to sort all edges at once.  
  However, as shown in this study, Prim’s advantage disappears beyond medium graph sizes because heap operations grow exponentially with vertex degree.

For real-world sparse and moderately dense networks, Kruskal’s algorithm is clearly more efficient. Prim’s algorithm only becomes preferable for dense graphs where most vertices are interconnected and the graph fits in memory using adjacency lists or matrices

.

**Impact of Edge Representation**

The performance of both algorithms depends heavily on how the graph is stored and accessed.

* **Adjacency List Representation:**  
  Works best with **Prim’s algorithm**, since it efficiently retrieves neighboring edges for each vertex. This setup minimizes redundant checks and improves runtime for smaller dense graphs.
* **Edge List Representation:**  
  Ideal for **Kruskal’s algorithm**, as it directly accesses all edges, sorts them by weight, and processes them sequentially.  
  This representation was used in the experiment, which explains Kruskal’s stable performance curve even on graphs exceeding 3,000 vertices..

**Implementation Complexity**

For code simplicity, maintainability, and compatibility with real-world data formats, Kruskal’s algorithm is preferable.

**Scalability and Performance**

Based on the experimental data of 28 graphs ranging from 10 to 3,000 vertices:

* **Prim’s algorithm** demonstrated steep growth in both runtime (from 1 ms → 612 ms) and operation count (from 83 → 5.7 million).
* **Kruskal’s algorithm**, however, remained remarkably stable — under 2 ms runtime and under 9,000 operations even in the largest graph.
* Top of Form

These results confirm the theoretical prediction that Prim’s performance degrades quickly as graph density increases, while Kruskal’s scales linearly with the number of edges.

* Bottom of Form

**Overall Preference Summary**

| **Condition** | **Preferred Algorithm** | **Reason** |
| --- | --- | --- |
| **Small graphs (4–30 vertices)** | Either | Both are equally fast and accurate |
| **Medium graphs (50–300 vertices)** | **Kruskal** | Faster due to efficient edge sorting |
| **Large graphs (400–1000 vertices)** | **Kruskal** | 50–100× faster in execution time |
| **Extra-large graphs (1500–3000 vertices)** | **Kruskal** | Scales best, lowest memory cost |
| **Dense graphs (many edges)** | **Prim** | Works well with adjacency lists |
| **Sparse graphs (few edges)** | **Kruskal** | Ideal for edge list representation |
| **Simple implementation (in code)** | **Kruskal** | Cleaner and modular structure |
| **Educational / theoretical cases** | Both | Provide complementary understanding of MST principles |

**Final Conclusion**

In both theory and practice, Kruskal’s algorithm consistently demonstrates superior performance across almost all datasets, especially for real-world transportation or infrastructure optimization tasks involving large-scale, edge-based networks. Its combination of simplicity, speed, and scalability makes it the preferred choice for modern graph-processing systems. While Prim’s algorithm remains an excellent conceptual and educational tool, its reliance on priority queue updates limits its performance on massive or dense networks. Therefore, for practical implementation in city transportation planning, network optimization, or large-scale connectivity problems, Kruskal’s algorithm is the most efficient and reliable choice.

1. Both algorithms correctly computed MSTs with identical costs.
2. Kruskal’s algorithm was consistently faster and required fewer operations.
3. Prim’s algorithm performed well only on small or dense graphs.
4. For large or sparse networks, Kruskal is much more efficient.
5. Real transportation systems are usually sparse, making Kruskal the most practical choice.
6. Experimental data fully matched theoretical expectations: O(E log E) < O(E log V).

**References:**

Lecture

GeeksforGeeks. (n.d.). Difference between Prim’s and Kruskal’s algorithm for MST. Retrieved from <https://www.geeksforgeeks.org/dsa/difference-between-prims-and-kruskals-algorithm-for-mst/>