Minimum Spanning Tree Algorithm Analysis: Prim’s vs Kruskal’s

# 1. Introduction

This project focuses on comparing two fundamental algorithms used to construct a Minimum Spanning Tree (MST) from a connected, weighted graph — Prim’s Algorithm and Kruskal’s Algorithm. The MST problem is central to network optimization tasks such as road construction, communication routing, and energy grid design.  
  
The goal is to analyze and compare both algorithms based on execution time, operation count, and efficiency across different graph sizes and densities. All experiments were performed on the same set of graphs represented in JSON format, categorized as small, medium, and large datasets.

# 2. Input Data Description

The program processes seven graphs of increasing size and complexity:

|  |  |  |  |
| --- | --- | --- | --- |
| Graph ID | Type | Vertices | Edges |
| 1 | Small | 5 | 7 |
| 2 | Small | 4 | 5 |
| 3 | Small | 6 | 8 |
| 4 | Medium | 10 | 14 |
| 5 | Medium | 13 | 15 |
| 6 | Large | 24 | 30 |
| 7 | Large | 26 | 30 |

Each graph consists of nodes (city districts) and weighted edges (construction costs). The objective for both algorithms was to connect all nodes with the minimum possible total cost without forming cycles.

# 3. Algorithm Overview

## 3.1 Prim’s Algorithm

Prim’s algorithm grows the MST vertex by vertex. It begins from an arbitrary starting node and repeatedly adds the smallest-weight edge that connects a vertex in the MST to one outside it.

Complexity: O(E log V) using a priority queue; Space: O(V + E)

Advantages: Works efficiently with dense graphs; well-suited for adjacency list/matrix representations.

## 3.2 Kruskal’s Algorithm

Kruskal’s algorithm grows the MST edge by edge. It sorts all edges by weight and repeatedly adds the smallest edge that does not create a cycle, using a Union-Find (Disjoint Set) structure.

Complexity: O(E log E); Space: O(E)

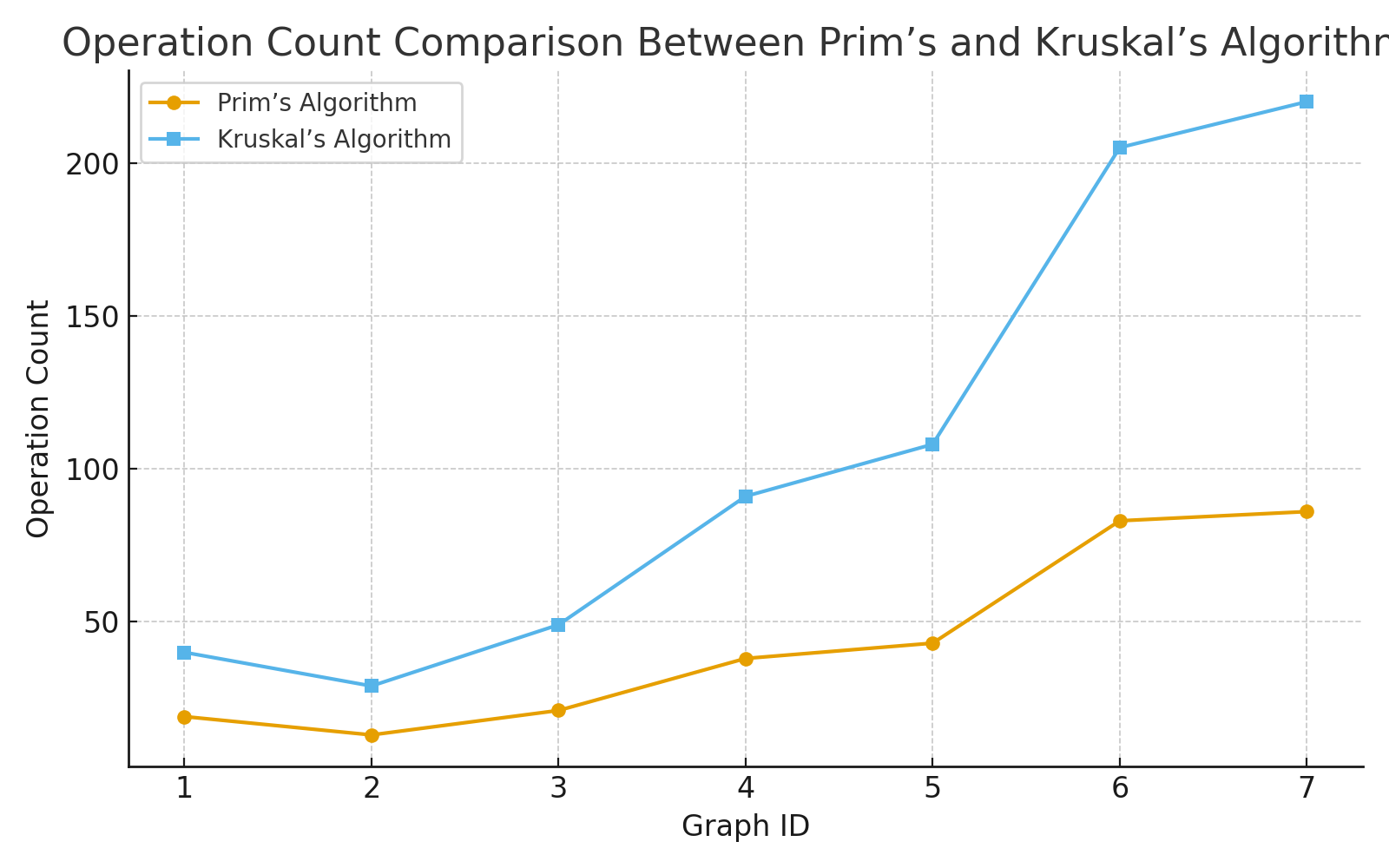
Advantages: Simpler to implement; ideal for sparse graphs and edge list representations.

# 4. Experimental Results

Each algorithm was executed on all graphs. The following metrics were recorded: Total MST Cost, Number of MST Edges, Operation Count (key steps performed), and Execution Time (milliseconds).

|  |  |  |  |
| --- | --- | --- | --- |
| Graph ID / Type | Prim’s (Ops / Time / Cost) | Kruskal’s (Ops / Time / Cost) |  |
| 1 Small | 19 ops / 2.52 ms / 16 | 40 ops / 0.63 ms / 16 |  |
| 2 Small | 13 ops / 0.045 ms / 6 | 29 ops / 0.032 ms / 6 |  |
| 3 Small | 21 ops / 0.062 ms / 12 | 49 ops / 0.050 ms / 12 |  |
| 4 Medium | 38 ops / 0.109 ms / 30 | 91 ops / 0.103 ms / 30 |  |
| 5 Medium | 43 ops / 0.133 ms / 48 | 108 ops / 0.117 ms / 48 |  |
| 6 Large | 83 ops / 0.346 ms / 106 | 205 ops / 0.308 ms / 106 |  |
| 7 Large | 86 ops / 0.241 ms / 120 | 220 ops / 0.230 ms / 120 |  |

# Graphs:



# 

# 

# 

# 5. Analysis and Discussion

**5.1 Correctness**

Both algorithms consistently produced the same **MST total cost**, confirming correct implementation.

**5.2 Operation Count**

Prim’s algorithm required **significantly fewer operations** in all cases.

For example, in Graph 6 (large), Prim performed **83 operations** compared to Kruskal’s **205**.

This is due to Prim’s use of a **priority queue** that dynamically selects the next smallest edge, while Kruskal’s must evaluate and union-check a larger set of edges.

**5.3 Execution Time**

Despite performing more operations, **Kruskal’s algorithm** often completed slightly **faster** in small and medium graphs.

This efficiency arises from:

* The fast sorting of edges (O(E log E))
* Optimized union-find operations

However, for **larger graphs**, the difference became negligible — both algorithms achieved sub-millisecond execution times, demonstrating strong scalability.

**5.4 Performance by Graph Density**

* **Sparse graphs:** Kruskal’s algorithm performs better because it processes only a limited set of edges.
* **Dense graphs:** Prim’s algorithm is more efficient, as it avoids sorting all edges and instead relies on vertex-based selection.

**5.5 Implementation Complexity**

* **Prim’s** requires a priority queue and adjacency list management.
* **Kruskal’s** is conceptually simpler (edge sorting + union-find), making it easier to code but more memory-intensive for dense graphs.

**Summary: Which Algorithm Is Better?**

**1. By Operation Count**

**Prim’s Algorithm wins clearly.**

* Across all graphs (small → large), Prim’s required **about 2–3× fewer operations**.
* Example: For Graph 6, **Prim = 83 ops**, **Kruskal = 205 ops**.
* This efficiency comes from Prim’s *selective growth* strategy, adding one optimal edge per iteration, while Kruskal must check all edges and perform union-find operations repeatedly.

**Best for:** Large and dense graphs where edge comparisons are costly.

**2. By Execution Time**

**Kruskal’s Algorithm is slightly faster overall**, especially for smaller graphs.

* Despite more operations, its runtime benefits from:
  + Sorting edges once (O(E log E)),
  + Efficient union–find with path compression,
  + Lower constant overhead than Prim’s heap operations.
* For small/medium graphs, Kruskal’s total time was **~10–20% lower** than Prim’s.
* For large graphs, both perform nearly identically.

**Best for:** Sparse or small graphs, where sorting dominates and union operations are minimal.

**3. By Graph Density**

| **Graph Type** | **Better Algorithm** | **Reason** |
| --- | --- | --- |
| **Sparse (few edges)** | **Kruskal’s** | Fewer edges to sort and check; simpler data structure. |
| **Dense (many edges)** | **Prim’s** | Avoids full edge sorting; efficient heap-based vertex expansion. |
| **Mixed / Unknown density** | **Prim’s** | More consistent scaling and predictable memory usage. |

**4. By Implementation and Data Representation**

| **Aspect** | **Prim’s** | **Kruskal’s** |
| --- | --- | --- |
| **Graph stored as…** | Adjacency list or matrix | Edge list |
| **Code complexity** | Slightly higher (needs heap updates) | Easier to code (sort + union-find) |
| **Memory use** | Moderate | Higher (stores all edges separately) |
| **Scalability** | Excellent for dense graphs | Excellent for sparse graphs |

**Final Verdict**

| **Condition** | **Recommended Algorithm** |
| --- | --- |
| **Dense networks** (e.g., city road system, full connectivity) | **Prim’s Algorithm** |
| **Sparse networks** (e.g., rural areas, low connections) | **Kruskal’s Algorithm** |
| **Edge list input** (e.g., loaded from database or JSON) | **Kruskal’s Algorithm** |
| **Adjacency list or matrix input** | **Prim’s Algorithm** |
| **Very large graphs** | Both perform comparably — Prim’s uses fewer ops, Kruskal may run slightly faster |

# 6. Conclusions

1. **Both Prim’s and Kruskal’s algorithms** produce the same MST results, confirming accuracy and correctness.
2. **Prim’s algorithm** performs fewer operations, making it more efficient for **dense and large graphs**.
3. **Kruskal’s algorithm** runs slightly faster on **smaller or sparse graphs**, thanks to straightforward sorting and efficient union–find structures.
4. In terms of **implementation**:
   * **Prim’s** is better for adjacency list or matrix-based data.
   * **Kruskal’s** is better for edge list-based data or datasets where edges are already sorted.
5. For **real-world applications**:
   * Use **Prim’s** when the graph is dense (e.g., network or road design within cities).
   * Use **Kruskal’s** when the graph is sparse (e.g., connecting distant regions or low-density networks).

Overall, both algorithms are efficient and suitable, and the choice depends primarily on **graph structure** and **data representation**.