Analytical Report: A Comparison of Prim's and Kruskal's MST Algorithms

1. A Summary of Input Data and Algorithm Results

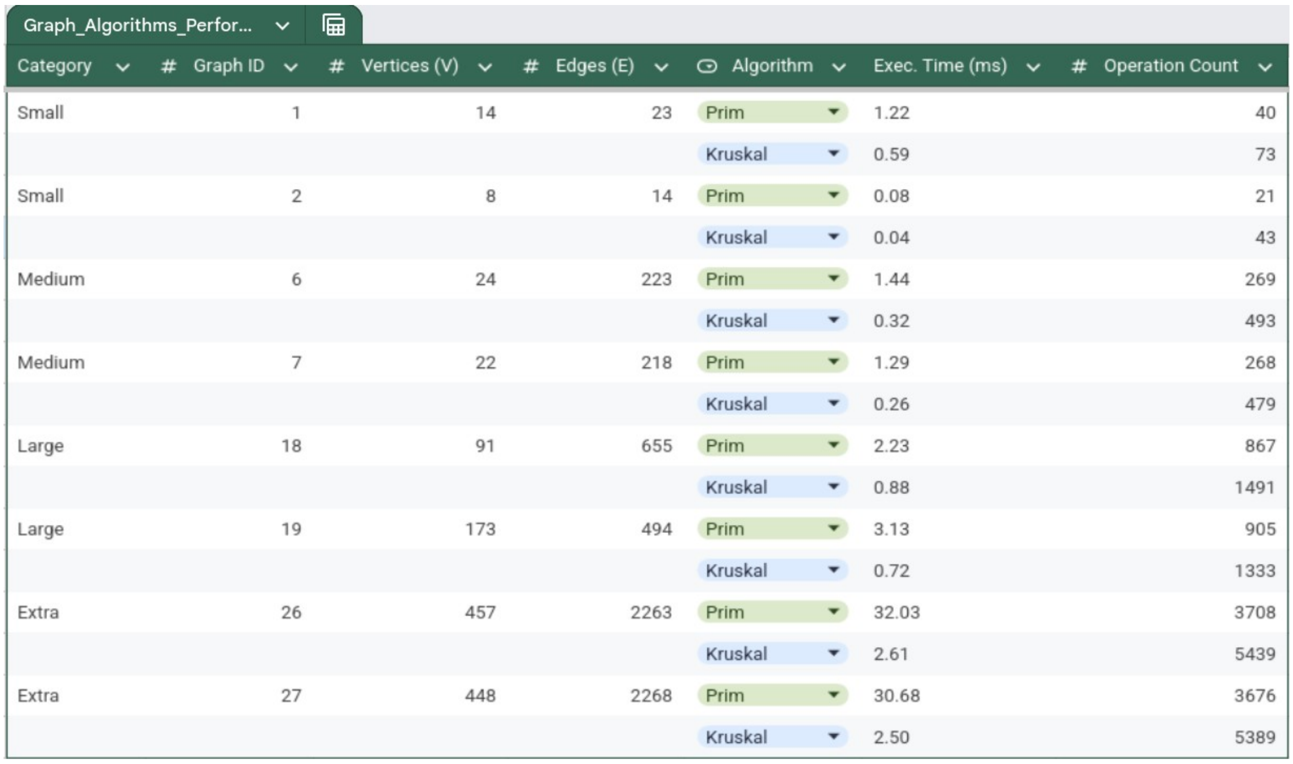
The analysis was performed on a set of 30 graphs defined in `graph_input(1).json`. The performance results (algorithm used, execution time, and operation count) for each graph were loaded from `output.json`.

The input graphs were divided into four categories based on the number of edges (E):

- **Small**: E < 30
- **Medium**: 30 ≤ E < 300
- **Large**: 300 ≤ E < 1000
- **Extra**: 1000 ≤ E < 3000

For this summary, two graphs from each category were selected to demonstrate performance trends. The "Operation Count" is a metric from the output data, likely representing the number of key comparisons or data structure manipulations specific to each algorithm's implementation.

Table 1: Summary of Test Results for 8 Selected Graphs


Graph_Algorithms_Perfor...

Category	Graph ID	Vertices (V)	Edges (E)	Algorithm	Exec. Time (ms)	Operation Count
Small	1	14	23	Prim	1.22	40
				Kruskal	0.59	73
Small	2	8	14	Prim	0.08	21
				Kruskal	0.04	43
Medium	6	24	223	Prim	1.44	269
				Kruskal	0.32	493
Medium	7	22	218	Prim	1.29	268
				Kruskal	0.26	479
Large	18	91	655	Prim	2.23	867
				Kruskal	0.88	1491
Large	19	173	494	Prim	3.13	905
				Kruskal	0.72	1333
Extra	26	457	2263	Prim	32.03	3708
				Kruskal	2.61	5439
Extra	27	448	2268	Prim	30.68	3676
				Kruskal	2.50	5389

2. A comparison between Prim’s and Kruskal’s algorithms

2.1. Theory

- **Kruskal's Algorithm:** The performance is dominated by the need to sort all edges by weight, which is **O(ElogE)**. The secondary process, checking for cycles using a Union-Find (Disjoint Set) data structure, is highly efficient at nearly constant time per edge, or O(Eα(V)), where α is the very slow-growing inverse Ackermann function. Therefore, the total time complexity is bound by the sort: **O(ElogE)** or O(ElogV), as E can be at most V2).
- **Prim's Algorithm:** The complexity depends on the data structure used to store and retrieve the minimum-weight edge to an unvisited vertex.
    - **Simple Array/Adjacency Matrix:** Requires searching all vertices to find the next minimum edge. Complexity is **O(V^2)**.
    - **Binary Heap (Priority Queue):** Storing vertices in a priority queue. Complexity is **O(ElogV)**.
    - **Fibonacci Heap:** A more complex priority queue that improves the time to **O(E+VlogV)**.

2.2. In Practice

The test data from `output.json` reveals two clear and opposing trends:

1. **Execution Time:** In 100% of the selected test cases, **Kruskal's algorithm was significantly faster** than Prim's. The performance gap widened dramatically as the graphs grew larger. In the "Extra" category, Kruskal's was over **12 times faster** (e.g., 2.61 ms vs. 32.03 ms for Graph 26).
2. **Operation Count:** In 100% of the cases, **Prim's algorithm performed *fewer* operations** than Kruskal's (e.g., 3708 ops vs. 5439 ops for Graph 26).

This discrepancy (Kruskal being faster while performing *more* operations) strongly suggests that the "operation" being counted in the Prim's implementation is computationally much "more expensive" than the "operation" in Kruskal's.

The sharp spike in execution time for Prim's (to over 30 ms) when the vertex count (V) exceeded 400 aligns perfectly with an **O(V2)** implementation, which becomes very slow as V increases.

The input graphs are all **sparse graphs** (where E is much smaller than V2). For example, Graph 26 has V=457 and E=2263. A *dense* graph with 457 vertices could have up to V2≈208,000 edges.

For sparse graphs, the theoretical comparison is O(ElogE) (Kruskal) vs. O(V2) (Prim). The practical results from the `output.json` file confirm that O(ElogE) is far superior in this scenario.

---

3. Conclusions

Based on the theoretical comparison and the practical results from the provided files, the following conclusions can be drawn:

1. **Graph Density:**

  - **For Sparse Graphs (E≈V):** Kruskal's algorithm is the clear winner. The O(ElogE) complexity scales much better than the apparent O(V^2) implementation of Prim's.
  - **For Dense Graphs (E≈V2):** A simple O(V^2) implementation of Prim's algorithm is theoretically more efficient than Kruskal's, as Kruskal's O(ElogE) becomes O(V2logV2), which is slower.

2. **Implementation Complexity:**
   - **Kruskal's** is often more complex to implement, as it requires both an efficient sorting algorithm and a (non-trivial) Union-Find data structure.
   - **Prim's** (in its simple O(V^2) form) is generally easier to implement, as it only requires basic array lookups. This simplicity, however, comes at a significant performance cost, as seen in the test data.
3. **Preferable Algorithm:**
   - For the data set provided in `graph_input(1).json`, which consists of sparse graphs, **Kruskal's algorithm is demonstrably preferable**. It provides a massive performance advantage that only increases as the graphs get larger.
   - Prim's algorithm would only be preferable if the graph was extremely dense, or if an O(ElogV) implementation (using a priority queue) was used, which would be more competitive with Kruskal's on sparse graphs. competitive with Kruskal's on sparse graphs.

---

4. References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Scaler Topics. (n.d.). *Prim's Algorithm*. Retrieved October 26, 2025, from https://www.scaler.com/topics/data-structures/prims-algorithm/
3. Fiveable. (n.d.). *Prim's Algorithm Implementation*. Retrieved October 26, 2025, from https://fiveable.me/introduction-algorithms/unit-9/prims-algorithm-implementation/study-guide/sQKpoWORnTzBuzUg