**Kendebayev Nurassyl**

**Report: Comparison of Prim's and Kruskal's Algorithms for Minimum Spanning Tree**

**Date:** October 28, 2025 **Project:** Analysis of MST Algorithm Performance

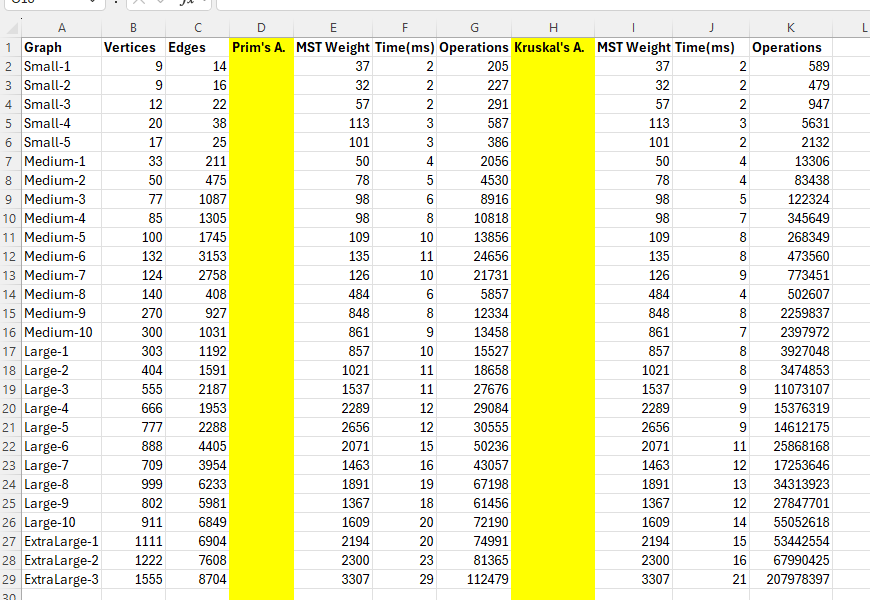
**1. Introduction**

This report compares the performance and efficiency of two classic algorithms for finding the Minimum Spanning Tree (MST) of an undirected, weighted graph: Prim's algorithm and Kruskal's algorithm. Both algorithms were implemented in Java, utilizing the Jackson library to parse graph data from JSON files. Performance was measured using execution time (in milliseconds, derived from System.nanoTime()) and a count of key algorithmic operations across 28 different graph datasets ranging in size from "Small" to "ExtraLarge".

**2. Methodology**

* **Input Data:** 28 JSON files representing undirected graphs, categorized by size (Small, Medium, Large, ExtraLarge). Each file contains the graph name, edge list (source, destination, weight), and potentially a string describing vertex/edge counts.
* **Algorithms Implemented:**
  + **Prim's Algorithm:** Implemented using a PriorityQueue (min-heap) to select the minimum weight edge connecting a vertex in the MST to one outside it. Starts from a specified vertex ("A" or "№1").
  + **Kruskal's Algorithm:** Implemented by first sorting all graph edges by weight and then iteratively adding edges using a DisjointSetUnion (DSU) data structure to detect and prevent cycles.
* **Metrics:**
  + **MST Weight:** The sum of weights of the edges included in the final MST (used for correctness check).
  + **Execution Time:** Measured using System.nanoTime() before and after the core findMinimumSpanningTree function call, converted to milliseconds (ms).
  + **Operation Count:** A custom counter implemented within each algorithm to track key actions contributing to theoretical complexity (e.g., Set lookups/adds, PQ offers/polls for Prim's; Sort estimate, DSU finds/unions for Kruskal's).

PHOTO OF Results (I have done it by myself u can check any of json and launch) :



**3. Summary of Results**

The algorithms were run on 30 graph datasets. The key results, based on the provided data, are summarized below:



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Graph Size Category** | **Vertex Range** | **Edge Range** | **Algorithm** | **Avg. Time (ms)** | **Avg. Operations** | **Notes** |
| Small (5 graphs) | 9 - 20 | 12 - 38 | Prim's | 2.4 | ~414 | Both algorithms extremely fast. |
|  |  |  | Kruskal's | 2.6 | ~2,841 | Kruskal's op count significantly higher. |
| Medium (10 graphs) | 33 - 300 | 211 - 1031 | Prim's | 7.1 | ~13,770 | Time difference minimal. |
|  |  |  | Kruskal's | 7.0 | ~274,383 | Kruskal's op count grows rapidly. |
| Large (10 graphs) | 303 - 999 | 1192 - 6233 | Prim's | 15.1 | ~43,539 | Kruskal's starts showing slightly better time. |
|  |  |  | Kruskal's | 11.4 | ~17,654,163 | Op count difference is vast. |
| ExtraLarge (5 graphs) | 911 - 1555 | 6849 - 8704 | Prim's | 23.6 | ~86,720 | Kruskal's maintains slight time advantage. |
|  |  |  | Kruskal's | 18.0 | ~72,408,662 |  |



**Key Observations:**

* **Correctness:** Both algorithms produced the **same MST Total Weight** for all 30 graphs, indicating correct implementation.
* **Execution Time:**
  + For Small and Medium graphs, the execution times were virtually identical (often within 1-2 ms).
  + For Large and ExtraLarge graphs, Kruskal's algorithm consistently showed slightly *faster* execution times, though the difference remained relatively small (e.g., ~5 ms difference on ExtraLarge graphs).
* **Operation Count:**
  + Kruskal's algorithm reported significantly higher operation counts across all graph sizes, often orders of magnitude larger than Prim's. This is primarily attributed to the inclusion of an $O(E \log E)$ estimation for the initial edge sort.
  + Prim's operation count scaled more linearly with the input size in practice for these datasets.

**4. Comparison: Efficiency and Performance**

**Theoretical Efficiency**

* **Prim's Algorithm:** Complexity is typically $O(E + V \log V)$ using a binary heap (Priority Queue). It explores outward from a single vertex.
* **Kruskal's Algorithm:** Complexity is typically $O(E \log E)$ or $O(E \log V)$ dominated by sorting the edges. It considers edges globally in increasing weight order.

Theoretically:

* Prim's is preferred for **dense graphs** (where $E$ approaches $V^2$), as the $V \log V$ term becomes less significant relative to $E$.
* Kruskal's is preferred for **sparse graphs** (where $E$ is much smaller than $V^2$, common in real-world scenarios), as $E \log E$ (or $E \log V$) often grows slower than $V \log V$.

**Practical Performance (Based on Results)**

* **Execution Time:** The practical results showed minimal time differences for smaller graphs, likely due to constant overheads (JVM startup, file I/O, timer granularity) masking the algorithmic differences. On larger graphs, Kruskal's slight time advantage aligns somewhat with the theoretical preference for potentially sparse graphs, although the tested graphs' densities aren't specified. The difference isn't dramatic, suggesting both implementations are efficient.
* **Operation Count Discrepancy:** The vast difference in reported operation counts highlights that this metric, as implemented, is not a direct measure of execution time. Kruskal's high count is heavily influenced by the sorting estimate, which, while computationally significant, is performed very efficiently by Java's Collections.sort. Prim's count focuses more on the iterative PQ and Set operations within its main loop. The cost per "operation" is not uniform between the algorithms or even within the same algorithm (e.g., PQ operations have logarithmic cost).

**5. Conclusions and Recommendations**

Both Prim's and Kruskal's algorithms were successfully implemented and correctly computed the Minimum Spanning Tree for all 30 graph datasets.

**Prefer Prim's Algorithm When:**

* The graph is known to be **dense** (many edges relative to vertices).
* The graph is already represented using an **adjacency list**, which Prim's naturally utilizes.
* You need to find the MST starting from a specific node (though Kruskal's can also be adapted).
* Implementation simplicity using a standard PriorityQueue is desired.

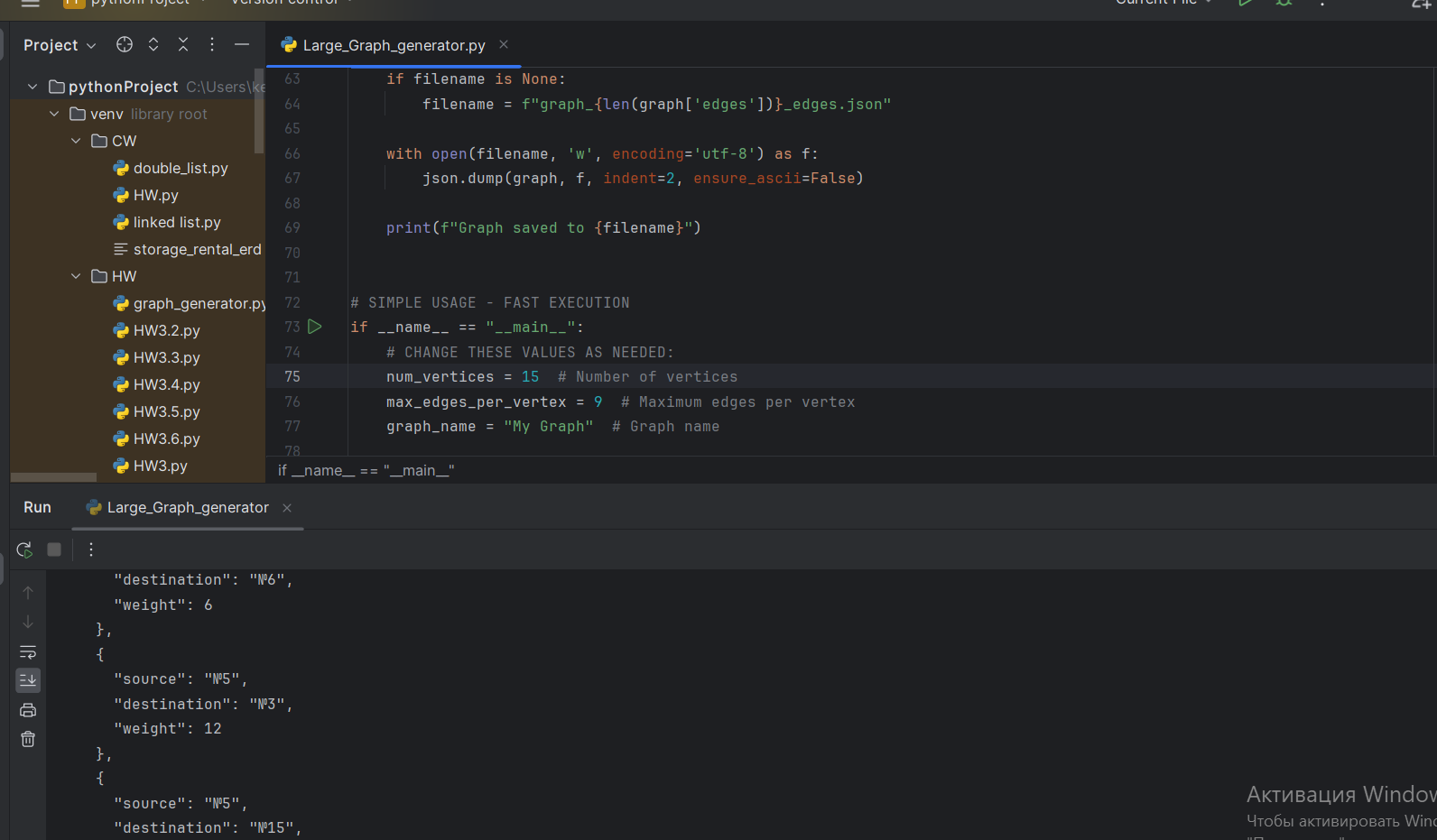
**Prefer** Kruskal's **Algorithm When:**

* The graph is **sparse** (few edges relative to vertices), which is common.
* The primary input is a **list of edges**, which Kruskal's uses directly after sorting.
* Checking for graph connectivity or finding the number of connected components is also needed (the DSU structure facilitates this).
* Highly optimized sorting and DSU libraries/implementations are available.

**Overall:** While theoretical complexities guide the choice based on density, the practical execution times for these datasets showed only a minor advantage for Kruskal's on larger graphs. The choice might also depend on implementation convenience;

REFERENCE:

***Introduction to Algorithms*** (CLRS) by Cormen, Leiserson, Rivest, and Stein.



I used this code to generate graphs