Optimization of a City Transportation Network using MST

Student: Nurdan Zhanabayev

Introduction

The goal is to optimize a city's transportation network by connecting all districts with the lowest possible construction cost.

This is achieved using **Minimum Spanning Tree (MST)** algorithms — **Prim's** and **Kruskal's** — applied to weighted undirected graphs where:

- Vertices represent city districts.
- Edges represent potential roads.
- Weights represent construction costs.

1. Summary of Input Data and Algorithm Results

Graph ID	Vertices	Edges	Connected	Algorithm	Total Cost	Operations	Execution Time (ms)
1	4	5	true	Prim	7.00	7	2.784
1	4	5	true	Kruskal	7.00	28	0.389
2	5	7	true	Prim	16.00	10	0.061
2	5	7	true	Kruskal	16.00	40	0.040
3	7	8	true	Prim	17.00	13	0.068
3	7	8	true	Kruskal	17.00	50	0.042

4	8	9	true	Prim	20.00	15	0.062
4	8	9	true	Kruskal	20.00	57	0.047

Observations:

- All graphs are connected (true).
- Both algorithms produced the same **Total Cost** for every graph, confirming algorithmic correctness.
- Kruskal's algorithm consistently performed **faster in execution time** (lower milliseconds).
- Prim's algorithm required **fewer operations** in every case, especially as graph size increased.

2. Practical Comparison of Efficiency and Performance

Criterias	Prim's Algorithm	Kruskal's Algorithm
Time complexity	$O(V^2)$ (simple) or O(E log V) (with PQ)	O(E log E) or O(E log V)
Observed execution time (ms)	Slightly higher (0.06–0.18 ms)	Slightly lower (0.04–0.10 ms)
Operations count	~3–4× fewer operations	~4× more operations (due to sorting edges)
Graph density handling	Better for dense graphs (many edges)	Better for sparse graphs (fewer edges)
Implementation	Simple with adjacency matrix or list	sorting and disjoint-set union (DSU)

Memory usage	Lower (stores adjacency structure)	Higher (stores full edge list and DSU)
Practical results	More efficient in operations, less code complexity	Faster runtime on small and medium sparse graphs

Metric	Derived From	What It Shows
Ops-to-Time Ratio	OperationsCount / ExecutionTimeMs	How many operations executed per ms — higher means tighter loops or faster operations.
Scaling Behavior	Compare Ops and Time as Vertices ↑	Shows how performance grows with graph size ($O(V^2)$ or $O(E \log V)$ patterns).

Prim (adjacency matrix)

T(V) = ∑_{i=1}^{V} O(V) = $O(V^2)$

Prim (adjacency list + min-heap)

Extract-min: $V \times O(\log V) = O(V \log V)$

Heap updates (insert/decrease-key): $\leq O(E) \times O(\log V) = O(E \log V)$

$T = O(V \log V + E \log V) = O((V+E) \log V) = O(E \log V)$

Kruskal

Sort edges: $O(E \log E)$

Union-Find (find/union) per edge: $O(E\ \alpha(V)) = O(E)$ (α = inverse Ackermann)

$T = O(E \log E + E) = O(E \log E) = O(E \log V)$ (since $\log E = \Theta(\log V)$)

Final forms:

Prim (matrix) = $O(V^2)$

Prim (heap) = $O(E \log V)$

Kruskal = $O(E \log E) \approx O(E \log V)$

Example (Graph 10):

- Prim → 47 ops, 0.186 ms
- Kruskal → 171 ops, 0.108 ms

Even though Kruskal is faster, it does ~3.6× more work, confirming that **Prim's operations are heavier but fewer**, while **Kruskal's are lighter but many**.

  - Lower operationsCount → algorithm did less computational work.
  - In your CSV, Prim uses **7–47 operations**, while Kruskal uses **28–171**, meaning Kruskal performed **3–4× more internal steps**.
  - Even if Kruskal is faster in milliseconds, Prim is more efficient per operation.

In practice:

Even though Kruskal's algorithm executed slightly faster in milliseconds, this difference is marginal and primarily influenced by the low graph sizes. Prim's algorithm performed fewer internal operations, meaning it scales better when many edges exist (dense graphs).

For sparse graphs ($E \approx V$), Kruskal's simplicity and edge-based nature make it slightly faster. For dense graphs, Prim's priority queue approach becomes preferable.

3. Conclusions

1. **Correctness:** Both algorithms consistently found identical MST costs, validating correct implementation.
2. **Efficiency:**
    a. Kruskal's algorithm was faster in raw time due to fewer edge relaxations per iteration on small datasets.
    b. Prim's algorithm executed fewer operations, suggesting higher computational efficiency for dense or large graphs.
3. **Performance by Graph Type:**
    a. **Sparse graphs ($E \approx V$):** Kruskal is preferable due to edge-based nature and efficient union-find.
    b. **Dense graphs ($E \approx V^2$):** Prim is preferable because it avoids unnecessary edge sorting.
4. **Scalability:** For large-scale networks (hundreds or thousands of vertices), a priority queue–based Prim implementation (using a min-heap) offers better asymptotic scaling.
5. **Implementation complexity:** Kruskal requires additional DSU data structures and sorting, while Prim integrates more directly with adjacency matrices/lists.

Overall:

- **Kruskal** is slightly faster on small sparse networks.
- **Prim** becomes superior for dense networks or when implemented with optimized data structures.

4. References

1. FreeCodeCamp. (2021). *Prim's Algorithm explained with pseudocode*. https://www.freecodecamp.org/news/prims-algorithm-explained-with-pseudocode/
2. GeeksforGeeks. (n.d.). *Prim's vs Kruskal's Algorithm – Comparison and Implementation*. https://www.geeksforgeeks.org/
3. OpenStax. (2020). *Data Structures and Algorithms*. OpenStax CNX. https://openstax.org/books/introduction-computer-science/pages/3-1-introduction-to-data-structures-and-algorithms?query=Prim
4. Programiz. (n.d.). *Kruskal's Algorithm*. https://www.programiz.com/dsa/kruskal-algorithm

5. Programiz. (n.d.). *Prim's Algorithm*. https://www.programiz.com/dsa/prim-algorithm