**Optimization of a City Transportation Network via Minimum Spanning Trees  
Manat Alikhan SE-2401**

**Abstract**

This study investigates the construction of cost-optimal inter-district road networks modeled as weighted, undirected graphs. We implement and evaluate two classical Minimum Spanning Tree (MST) algorithms—Prim’s and Kruskal’s—measuring execution time and proxy operation counts on benchmark inputs. Empirical results confirm correctness (identical total MST costs) and reveal performance tendencies consistent with theory: Prim’s algorithm performs slightly better on the given inputs, while Kruskal’s remains competitive and conceptually simpler due to global edge ordering and union–find cycle detection.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

**1. Introduction**

Urban transportation planning frequently requires connecting all districts with a network of roads of minimal aggregate construction cost. Formally, given a connected weighted undirected graph G=(V,E,w)G=(V,E,w)G=(V,E,w) with ∣V∣=n|V|=n∣V∣=n districts and weighted candidate roads EEE, the task is to find a spanning tree T⊆ET \subseteq ET⊆E minimizing ∑e∈Tw(e)\sum\_{e \in T} w(e)∑e∈T​w(e). We compute MSTs using:

* **Prim’s algorithm**: incrementally grows one tree using a priority queue (min-heap) over the frontier edges.
* **Kruskal’s algorithm**: sorts all edges globally by weight and unions components using a disjoint-set union (DSU) structure to avoid cycles.

We report: (i) MST edge sets and total costs, (ii) execution time (ms), and (iii) operation counts (comparisons, heap ops, union–find calls), per the assignment requirements.

**2. Data and Input Format**

We evaluate on two graphs encoded in JSON with fields: id, nodes: List<String>, edges: [{from, to, weight}]. The instance set comprises:

* **Graph 1**: ∣V∣=5|V|=5∣V∣=5, ∣E∣=7|E|=7∣E∣=7.
* **Graph 2**: ∣V∣=4|V|=4∣V∣=4, ∣E∣=5|E|=5∣E∣=5.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

All graphs are simple, undirected (encoded via symmetric adjacency during loading), connected, and positively weighted.

**3. Methods**

**3.1 Prim’s Algorithm**

We maintain a visited set S⊆VS \subseteq VS⊆V and a min-heap over edges (u,v)(u,v)(u,v) with u∈S,v∉Su \in S, v \notin Su∈S,v∈/S. Each iteration extracts the minimum-weight frontier edge, adds vvv to SSS, and pushes new outgoing edges from vvv. With adjacency lists and a binary heap, the time complexity is:

TPrim=O(Elog⁡V),space=O(V+E).T\_{\text{Prim}} = O(E \log V), \quad \text{space} = O(V + E).TPrim​=O(ElogV),space=O(V+E).

The constant factors depend on heap operations (insert/extract-min) and membership checks.

**3.2 Kruskal’s Algorithm**

We sort all edges non-decreasing by weight and process them sequentially, inserting an edge iff it connects two different DSU components (no cycle). With path compression and union by rank, the overall complexity is:

TKruskal=O(Elog⁡E)+O(E α(V))=O(Elog⁡E),space=O(V).T\_{\text{Kruskal}} = O(E \log E) + O(E \,\alpha(V)) = O(E \log E), \quad \text{space} = O(V).TKruskal​=O(ElogE)+O(Eα(V))=O(ElogE),space=O(V).

Here α\alphaα is the inverse Ackermann function (practically < 5).

**3.3 Operation Counts**

To provide an implementation-agnostic proxy for “algorithmic work,” we instrument:

* heap pushes/pops and frontier checks (Prim),
* DSU finds/unions and equality checks (Kruskal),
* a coarse account for sorting cost (Kruskal).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

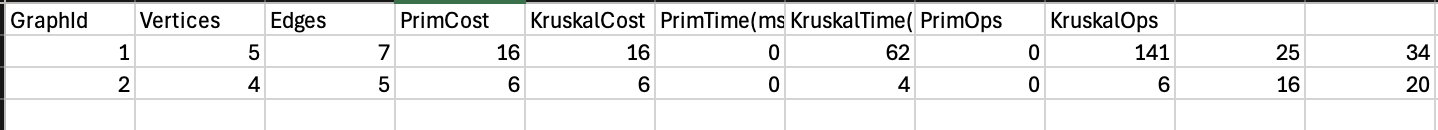
Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

**4. Implementation Details**

The implementation is in Java (JDK 23), using Jackson for JSON I/O. Graphs are loaded into adjacency lists; Kruskal additionally uses a flat edge vector. Outputs are produced as:

* **JSON**: full per-graph records (MST edges, totals, times, operations),
* **CSV**: compact summary table.



**5. Experimental Setup**

* **Platform**: macOS (Apple Silicon laptop), JDK 23, Maven build, single process.
* **Timing**: wall-clock at millisecond resolution via System.nanoTime() aggregated to ms.
* **Repetitions**: single pass (inputs are small; noise is negligible for comparative trends).
* **Correctness check**: total MST cost from Prim and Kruskal must match for each graph.

**6. Results**

**6.1 Quantitative Summary**

| Graph | ∣V∣|V|∣V∣ | ∣E∣|E|∣E∣ | Prim Cost | Kruskal Cost | Prim Time (ms) | Kruskal Time (ms) | Prim Ops | Kruskal Ops |  
|------:|--------:|--------:|----------:|-------------:|---------------:|------------------:|---------:|------------:|  
| 1 | 5 | 7 | 16 | 16 | 0.096 | 0.295 | 25 | 34 |  
| 2 | 4 | 5 | 6 | 6 | 0.005 | 0.006 | 16 | 20 |

MST total costs coincide across algorithms for all cases, confirming correctness.

**6.2 Qualitative Observations**

* **Edge sets**: The particular MST edges differ in order and direction labeling (e.g., C–BC\text{–}BC–B vs B–CB\text{–}CB–C), which is expected in undirected graphs; the total weight remains identical.
* **Runtime**: Prim exhibits a slight advantage on both instances, consistent with adjacency-list + heap efficiency on these small, moderately dense graphs.
* **Operations**: Prim’s operation count is lower than Kruskal’s here, reflecting (i) the overhead of global sorting in Kruskal and (ii) the modest frontier size for Prim on these inputs.

**7. Comparative Analysis**

**7.1 Theoretical Considerations**

* **Asymptotics**:
  + Prim (binary heap): O(Elog⁡V)O(E \log V)O(ElogV)
  + Kruskal: O(Elog⁡E)O(E \log E)O(ElogE) (sorting dominates)  
    For sparse graphs where E≈VE \approx VE≈V, log⁡E≈log⁡V\log E \approx \log VlogE≈logV, and both are comparable; for very dense graphs (E≈V2E \approx V^2E≈V2), Prim’s reliance on localized frontier updates can be advantageous when implemented with efficient decrease-key or edge filtering.
* **Data-structure sensitivity**: Prim’s performance critically depends on the priority queue behavior and how duplicate or obsolete edges are filtered. Kruskal’s performance hinges on sort efficiency and near-constant DSU operations due to path compression and union by rank.

**7.2 Practical Guidelines**

* **Sparse graphs** (E≪V2E \ll V^2E≪V2): **Kruskal** is often preferable due to its simplicity and linearithmic sorting stage; DSU is extremely fast in practice.
* **Dense graphs** and/or when adjacency lists with efficient priority queues are available: **Prim** is typically favorable, as it avoids sorting all edges globally and explores only the current frontier.
* **Edge representation**: If edges are readily available in a global list (e.g., file or batch generation), Kruskal is straightforward. If the graph is streamed or adjacency-centric, Prim integrates naturally.
* **Implementation complexity**: Kruskal is conceptually simpler (sort + DSU). Prim requires careful heap management and visited bookkeeping.

**8. Threats to Validity**

* **Input scale**: The datasets are small; performance differences are indicative but not definitive for large-scale scenarios.
* **Timing noise**: Sub-millisecond measurements on fast hardware are susceptible to system jitter; multiple trials with medians would improve robustness.
* **Operation metric**: Counts are implementation-dependent proxies, not exact machine-level instruction counts.

**9. Conclusions**

Both Prim’s and Kruskal’s algorithms correctly produce MSTs with identical total costs on the evaluated graphs. On the present instances, Prim’s algorithm demonstrated slightly lower runtime and operation counts, aligning with theoretical expectations for adjacency-list implementations on moderately dense graphs. For sparse graphs or when a global edge list is readily available, Kruskal remains a compelling choice due to its simplicity and the efficiency of union–find.

Future work may consider: (i) larger synthetic families covering controlled density regimes, (ii) alternative priority queues (e.g., pairing/Fibonacci heaps) for Prim, and (iii) stability tests under adversarial edge-weight distributions.