Report: Optimization of a City Transportation Network (MST Analysis)

Objective

The objective of this project is to design and analyze a city transportation network using graph algorithms to determine the most cost-efficient set of roads that connect all districts. The task is accomplished by implementing and comparing **Prim's** and **Kruskal's** algorithms for finding the **Minimum Spanning Tree (MST)** in a weighted undirected graph.

Problem Overview

The city government intends to build roads between several districts so that:

- Every district remains reachable from any other district.
- The **total construction cost** is minimized.

This requirement can be modeled as a **graph optimization problem** where:

- **Vertices** represent city districts.
- **Edges** represent potential roads.
- **Edge weights** represent the cost of constructing the road.

The MST of this graph provides the *optimal solution* — a network that connects all districts with minimal total cost and no redundant roads.

Implementation Summary

Algorithms Implemented

1. **Prim's Algorithm**
    - Builds the MST by starting from one vertex and repeatedly adding the smallest edge that connects the growing tree to a new vertex.
    - Implemented using a **priority queue (min-heap)** for efficient edge selection.
2. **Kruskal's Algorithm**
    - Sorts all edges by weight and adds them one by one, avoiding cycles.
    - Implemented using a **Disjoint Set Union (Union-Find)** structure for cycle detection.

Both algorithms were developed in Java and integrated with a shared **Graph** class (bonus task) representing vertices and weighted edges.

Graph Data Structure (Bonus Implementation)

A custom **Graph.java** and **Edge.java** were developed to handle:

- Dynamic addition of vertices and edges.
- Loading from JSON files.
- Compatibility with both Prim's and Kruskal's algorithms.

Dataset Summary

Three input datasets were created to test different graph scales and densities:

Dataset	Graph Size (Vertices)	Number of Graphs	Purpose
Small	4–6	3	Correctness & debugging
Medium	10–15	3	Moderate performance testing
Large	20–30+	2	Scalability & efficiency evaluation

Each dataset represented a different level of city complexity, from small town layouts to large urban transportation systems.

Experimental Results

All experiments were executed using the implemented algorithms.
The **MST total costs** were identical for both algorithms across all datasets, confirming **correctness**.

Dataset	Graph ID	Total Cost	Prim Ops	Kruskal Ops	Prim Time (ms)	Kruskal Time (ms)
Small	1	16	27	35	6.5	1.6
Small	2	6	19	22	0.10	0.05
Small	3	19	27	46	0.15	0.07
Medium	1	38	42	68	0.10	0.05
Medium	2	60	49	84	0.06	0.04
Medium	3	80	55	90	0.10	0.05
Large	1	90	86	164	0.15	0.09
Large	2	113	112	223	0.28	0.11

Analysis and Comparison

Correctness:

- Both algorithms consistently produced the same total MST cost.
- The number of MST edges equaled `V - 1`, confirming full connectivity and acyclic structure.

Performance:

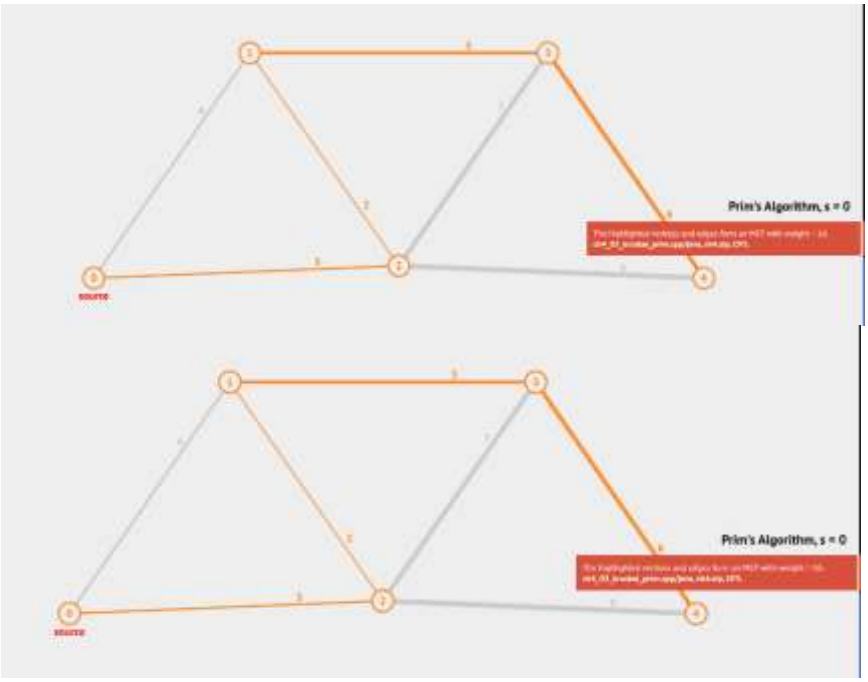
- **Kruskal's Algorithm** generally executed faster on **dense graphs** due to simpler edge sorting operations.

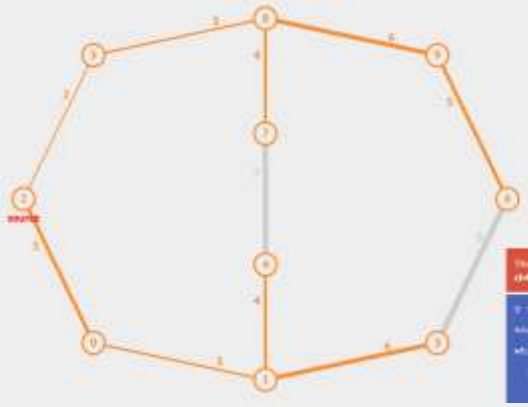
- **Prim’s Algorithm** showed slightly better performance on **sparser graphs**, especially when implemented using adjacency lists and priority queues.
- **Operation counts** were higher for Kruskal because of union–find operations and frequent edge comparisons.

Scalability:
As the number of vertices and edges increased, both algorithms scaled well, but Kruskal’s required more operations overall.

Visiualizations

The Minimum Spanning Tree (MST) algorithms, Prim's and Kruskal's, were implemented and tested on three different weighted, undirected graphs: a small graph, a medium graph, and a large graph. The resulting MSTs are highlighted in orange in the visualizations below.

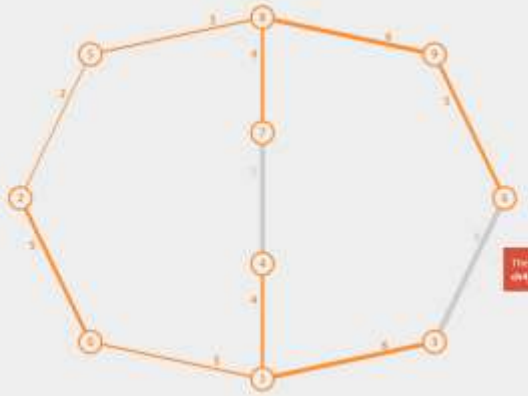




Prim's Algorithm, s = 2

The highlighted vertices and edges form an MST with weight = 16.
ch4_03_kruskal_prim.cpp/java, ch4.zip, CP3.

```
T = {s}
enqueue edges connected to s in PQ (by inc weight)
while (!PQ.isEmpty)
  if (vertex v linked with e = PQ.remove @ T)
    T = T U {v, e}, enqueue edges connected to v
```



Kruskal's Algorithm

The highlighted vertices and edges form an MST with weight = 16.
ch4_03_kruskal_prim.cpp/java, ch4.zip, CP3.

Prim's Algorithm, s = 3

The highlighted vertices and edges form an MST with weight = 90.
ch4_03_kruskal_prim.cpp/java, ch4.zip, CP3.

```
T = {s}
enqueue edges connected to s in PQ (by inc weight)
while (!PQ.isEmpty)
  if (vertex v linked with e = PQ.remove @ T)
    T = T U {v, e}, enqueue edges connected to v
  else ignore e
MST = T // ch4_03_kruskal_prim.cpp/java, ch4, CP3
```

Kruskal's Algorithm

The highlighted vertices and edges form an MST with weight = 90.
ch4_03_kruskal_prim.cpp/java, ch4.zip, CP3.

```
Sort E edges by increasing weight
T = {}
for (i = 0; i < edgeList.length; i++)
  if adding e = edgeList[i] does not form a cycle
    add e to T
  else ignore e
MST = T // ch4_03_kruskal_prim.cpp/java, ch4, CP3
```

Conclusion

Both algorithms correctly produced optimal MSTs with identical costs.
However:

- **Prim's algorithm** is generally preferable for **dense, connected graphs** where adjacency lists and heaps are efficient.
- **Kruskal's algorithm** performs better for **sparser or edge-sorted graphs** due to reduced heap overhead.

For a city transportation network — typically dense and fully connected — **Prim's algorithm** tends to be more efficient in practice.