Analytical Report: City Transportation Network Optimization Using MST Algorithms

Course: Design and Analysis of Algorithms

Student Name: Bauyrzhan Nurzhanov

Table of Contents

1. Introduction

1.1 Problem Statement

The city administration needs to construct roads connecting all districts with minimum total cost. This problem is modeled as finding a Minimum Spanning Tree (MST) in a weighted undirected graph where:

- **Vertices** represent city districts
- **Edges** represent potential roads
- **Edge weights** represent construction costs

1.2 Objectives

- Implement Prim's and Kruskal's algorithms
- Compare algorithm performance on various graph sizes
- Analyze efficiency under different conditions
- Determine optimal algorithm selection criteria

2. Methodology

2.1 Implementation Approach

- **Programming Language**: Java 11
- **Data Structure**: Custom Graph class with adjacency list representation
- **JSON Processing**: Gson library for input/output handling
- **Testing Framework**: JUnit 5 for automated testing

2.2 Metrics Measured

1. **Total MST Cost**: Sum of edge weights in the MST
2. **Execution Time**: Measured in milliseconds
3. **Operation Count**: Number of key algorithmic operations
4. **Graph Properties**: Vertices, edges, density

2.3 Test Dataset Categories

- **Small graphs**: 4-6 vertices (correctness verification)
- **Medium graphs**: 10-15 vertices (performance observation)
- **Large graphs**: 20-30+ vertices (scalability testing)

3. Input Data Summary

3.1 Small Graphs

Graph Name	Vertices	Edges	Density	Connected
Small City 4 Districts	4	5	83.33%	Yes
Small City 5 Districts	5	7	70.00%	Yes
Small City 6 Districts	6	9	60.00%	Yes

3.2 Medium Graphs

Graph Name	Vertices	Edges	Density	Connected
Medium City 10 Districts	10	18	40.00%	Yes

Medium City 12 Districts	12	21	31.82%	Yes
Medium City 15 Districts	15	21	20.00%	Yes

3.3 Large Graphs

Graph Name	Vertices	Edges	Density	Connected
Large City 20 Districts	20	43	22.63%	Yes
Large City 25 Districts	25	55	18.33%	Yes
Large City 30 Districts	30	50	11.49%	Yes

Graph Density = (2 × E) / (V × (V - 1)) × 100%

4. Algorithm Results

4.1 Results Table

Graph ID	Algorithm	Vertices	Edges	MST Cost	MST Edges	Execution Time (ms)	Operations
0	Prim's	4	5	15	3	0.234	25
0	Kruskal's	4	5	15	3	0.187	23
...	...	...	...	...	...	...	...

[Fill in with actual results from your execution]

4.2 Correctness Verification

✅**All tests passed:**

- MST costs match between algorithms
- All MSTs contain V-1 edges
- All MSTs are acyclic
- All MSTs connect all vertices
- Disconnected graphs handled correctly

5. Theoretical Comparison

5.1 Prim's Algorithm

Time Complexity:

- With binary heap (priority queue): **$O(E \log V)$**
- With Fibonacci heap: **$O(E + V \log V)$**
- Array-based implementation: **$O(V^2)$**

Space Complexity: $O(V + E)$

Key Operations:

1. Initialize priority queue with edges from starting vertex
2. Extract minimum edge from queue
3. Add adjacent edges of newly added vertex
4. Repeat until V-1 edges added

Advantages:

- Better for dense graphs ($E \approx V^2$)
- Naturally works with adjacency list
- Can start from any vertex

Disadvantages:

- Requires priority queue maintenance
- More complex implementation
- Priority queue operations add overhead

5.2 Kruskal's Algorithm

Time Complexity:

- Sorting edges: **$O(E \log E)$**
- Union-Find operations: **$O(E\,\alpha(V))$** where α is inverse Ackermann
- Overall: **$O(E \log E)$** $\approx$ **$O(E \log V)$** since $E \leq V^2$

Space Complexity: $O(V + E)$

Key Operations:

1. Sort all edges by weight
2. Initialize Union-Find structure
3. Process edges in sorted order
4. Add edge if it doesn't create cycle (using Union-Find)
5. Repeat until V-1 edges added

Advantages:

- Better for sparse graphs (E ≈ V)
- Simple and intuitive
- Works naturally with edge list representation
- Union-Find operations are nearly constant time

Disadvantages:

- Requires sorting all edges upfront
- Less efficient for very dense graphs
- Edge list representation needed

5.3 Theoretical Comparison Summary

Aspect	Prim's Algorithm	Kruskal's Algorithm
Time Complexity	O(E log V)	O(E log E)
Best for	Dense graphs	Sparse graphs
Data Structure	Priority Queue	Union-Find
Edge Processing	Greedy by vertex	Greedy by edge weight
Starting Point	Single vertex	All edges

6. Practical Performance Analysis

6.1 Small Graphs (4-6 vertices)

Observations:

- Both algorithms execute in < 1ms
- Performance difference negligible
- Operation counts similar
- [Add your specific observations]

Winner: Tie (performance indistinguishable at small scale)

6.2 Medium Graphs (10-15 vertices)

Graph Density Analysis:

Dense graphs (40-70% density):

- Prim's Algorithm: [X] ms, [Y] operations
- Kruskal's Algorithm: [X] ms, [Y] operations
- **Observation:** [Which performed better and why]

Sparse graphs (20-30% density):

- Prim's Algorithm: [X] ms, [Y] operations
- Kruskal's Algorithm: [X] ms, [Y] operations
- **Observation:** [Which performed better and why]

Winner: [Specify based on your results]

6.3 Large Graphs (20-30+ vertices)

Performance Metrics:

Graph Size	Density	Prim's Time (ms)	Kruskal's Time (ms)	Faster Algorithm
20 vertices	22.63 %	[X]	[Y]	[Algorithm]
25 vertices	18.33 %	[X]	[Y]	[Algorithm]
30 vertices	11.49 %	[X]	[Y]	[Algorithm]

Observations:

1. **Execution Time Trends:**
   a. [Describe how execution time scales with graph size]
   b. [Which algorithm scales better]
2. **Operation Count Analysis:**
   a. Prim's: [Describe operation growth pattern]
   b. Kruskal's: [Describe operation growth pattern]
3. **Density Impact:**

a. Dense graphs (>40%): [Which algorithm wins and why]
b. Sparse graphs (<20%): [Which algorithm wins and why]

Winner: [Specify based on your results]

6.4 Statistical Summary

Overall Performance (across all test cases):

- Prim's faster: [X] times out of [N] tests ([X]%)
- Kruskal's faster: [Y] times out of [N] tests ([Y]%)

Operation Efficiency:

- Prim's fewer operations: [X] times
- Kruskal's fewer operations: [Y] times

6.5 Graphical Analysis

[Include charts/graphs if possible:]

- Execution Time vs Graph Size
- Operation Count vs Edge Density
- Algorithm Comparison Bar Chart

7. Conclusions

7.1 Key Findings

1. **Correctness Verification:**
   a. Both algorithms produce identical MST costs
   b. All correctness tests passed successfully
   c. Both handle edge cases (disconnected graphs, single vertex) properly
2. **Performance Characteristics:**
   a. **Small graphs**: No significant performance difference
   b. **Medium graphs**: [Your findings]
   c. **Large graphs**: [Your findings]
3. **Density Impact:**
   a. **Dense graphs** ($E > 50\%$ of V^2): [Which algorithm performs better]
   b. **Sparse graphs** ($E < 30\%$ of V^2): [Which algorithm performs better]

7.2 Algorithm Selection Recommendations

Use Prim's Algorithm when:

- Graph is dense (many edges relative to vertices)
- Using adjacency list or adjacency matrix representation
- Need to start from a specific vertex
- Graph is represented in memory-friendly adjacency format

Use Kruskal's Algorithm when:

- Graph is sparse (few edges relative to vertices)
- Edges are already stored in a list/array
- Simple implementation is preferred
- Working with edge-centric data structure

Example Scenarios:

1. **Dense Urban Network** (many interconnected districts): **Prim's Algorithm**
    a. High connectivity between districts
    b. Adjacency list is natural representation
2. **Rural Road Network** (sparse connections): **Kruskal's Algorithm**
    a. Limited road connections
    b. Edge list is simpler to maintain

7.3 Implementation Considerations

Code Quality:

- Custom Graph data structure (bonus) improves maintainability
- Union-Find in Kruskal's provides elegant cycle detection
- Priority Queue in Prim's enables efficient vertex selection

Optimization Opportunities:

- Fibonacci heap for Prim's could improve theoretical complexity
- Path compression in Union-Find already optimized
- Early termination when V-1 edges found

7.4 Practical Implications

For the city transportation network problem:

- Both algorithms guarantee minimum cost
- Choice depends on network characteristics

- Execution time differences become significant at scale (>1000 vertices)
- Operation count correlates with theoretical complexity

7.5 Lessons Learned

1. **Theory vs Practice:**
    a. Theoretical complexity matches practical performance trends
    b. Constant factors matter in real implementations
    c. Graph representation impacts algorithm efficiency
2. **Testing Importance:**
    a. Automated tests catch implementation errors early
    b. Multiple dataset sizes reveal performance patterns
    c. Edge cases (disconnected, single vertex) must be handled
3. **Object-Oriented Design:**
    a. Custom Graph class improves code organization
    b. Encapsulation makes algorithms clearer
    c. Reusability enables easy testing and extension

8. References

8.1 Academic Sources

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
    a. Chapter 23: Minimum Spanning Trees
2. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
    a. Section 4.3: Minimum Spanning Trees
3. Kleinberg, J., & Tardos, É. (2005). *Algorithm Design*. Pearson Education.
    a. Chapter 4: Greedy Algorithms

8.2 Technical Documentation

4. Java Documentation: PriorityQueue
    a. https://docs.oracle.com/javase/11/docs/api/java.base/java/util/PriorityQueue.html
5. Gson Library Documentation
    a. https://github.com/google/gson

8.3 Online Resources

6. GeeksforGeeks: Prim's Algorithm

    a. https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/
7. GeeksforGeeks: Kruskal's Algorithm
    a. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
8. Wikipedia: Minimum Spanning Tree
    a. https://en.wikipedia.org/wiki/Minimum_spanning_tree