# Minimum Spanning Tree Algorithms

## Performance Analysis Report

Student: Yelzhan Zhandos  
Group: SE-2426

## 1. Executive Summary

This comprehensive analysis compares Prim's and Kruskal's algorithms for Minimum Spanning Tree computation across 28 test graphs. Kruskal's algorithm demonstrated superior performance in 85.7% of test cases with an average speed improvement of 22.6%.

### Key Findings:

* Kruskal wins: 24 out of 28 tests
* Prim wins: 4 out of 28 tests
* Correctness: 100% MST validation across all tests
* Recommendation: Kruskal for most practical applications

## 2. Experimental Methodology

### 2.1 Test Data Composition

| Category | Graphs | Vertices | Edges | Density |
| --- | --- | --- | --- | --- |
| Small | 5 | 4–6 | 3–9 | Sparse |
| Medium | 10 | 11–15 | 22–42 | Medium |
| Large | 10 | 22–31 | 69–139 | Dense |
| XLarge | 3 | 50–100 | 122–495 | Very Dense |

### 2.2 Measurement Approach

* Time Measurement: System.nanoTime() for algorithm core only
* Operation Counting: Key algorithmic operations tracked
* Validation: MST properties verified for all results

## 3. Performance Results

### 3.1 Execution Time Analysis

Table 1: Average Execution Times (ms)

| Category | Prim Time | Kruskal Time | Winner | Speed Improvement |
| --- | --- | --- | --- | --- |
| Small | 0.336 | 0.188 | Kruskal | -44.1% |
| Medium | 0.059 | 0.045 | Kruskal | -23.9% |
| Large | 0.155 | 0.121 | Kruskal | -21.8% |
| XLarge | 0.385 | 0.383 | Kruskal | -0.7% |

### 3.2 Operation Count Analysis

Table 2: Operation Count Ranges

| Graph Size | Prim Operations | Kruskal Operations | Ratio |
| --- | --- | --- | --- |
| Small | 32–70 | 50–132 | 1.6x |
| Medium | 155–274 | 263–486 | 1.8x |
| Large | 461–887 | 964–1908 | 2.2x |
| XLarge | 1842–3178 | 2026–8216 | 2.6x |

## 4. Detailed Performance Breakdown

### 4.1 Small Graphs (4–6 vertices)

* Kruskal wins: 5/5 tests (100%)
* Average improvement: 44.1%
* Best case: Graph 6 (-48.9%)

### 4.2 Medium Graphs (11–15 vertices)

* Kruskal wins: 8/10 tests (80%)
* Average improvement: 30.9%
* Best case: Graph 12 (-57.0%)

### 4.3 Large Graphs (22–31 vertices)

* Kruskal wins: 9/10 tests (90%)
* Average improvement: 19.8%
* Best case: Graph 18 (-31.3%)

### 4.4 XLarge Graphs (50–100 vertices)

* Kruskal wins: 2/3 tests (67%)
* Performance: Nearly equivalent (0.7% difference)

## 5. Algorithm Characteristics

### 5.1 Prim’s Algorithm

* Time Complexity: O(E log V)
* Space Complexity: O(V + E)
* Strengths: Consistent performance, efficient on dense graphs
* Weaknesses: Priority queue overhead

### 5.2 Kruskal’s Algorithm

* Time Complexity: O(E log E)
* Space Complexity: O(V + E)
* Strengths: Efficient Union-Find, cache-friendly processing
* Weaknesses: Sorting overhead on very dense graphs

## 6. Technical Insights

### 6.1 Why Kruskal Performs Better

* Optimized Sorting: Java’s Arrays.sort() provides efficient edge processing
* Cache Locality: Sequential edge array access benefits from CPU caching
* Union-Find Efficiency: Path compression and union-by-rank optimizations

### 6.2 Graph Density Impact

* Sparse Graphs: Kruskal’s Union-Find provides significant advantage
* Medium-Density: Kruskal maintains strong performance
* Very Dense: Performance gap narrows as sorting overhead increases

## 7. Correctness Validation

All 28 test cases passed comprehensive MST validation:

* Cost Matching: Identical MST costs between algorithms
* Edge Count: Exactly V-1 edges in all MSTs
* Connectivity: All vertices connected
* Acyclic: No cycles detected in spanning trees

## 8. Conclusions and Recommendations

### 8.1 Algorithm Selection Guidelines

Choose Kruskal when:

* Working with sparse to medium-density graphs
* Implementation simplicity is important
* Consistent performance across graph sizes needed

Choose Prim when:

* Dealing with very dense graphs
* Memory-constrained environments
* Vertex-based incremental processing required

### 8.2 Practical Recommendations

1. Default Choice: Kruskal for most applications
2. Special Cases: Consider Prim for specific dense graph scenarios
3. Production Use: Profile both algorithms for critical applications
4. Implementation: Use optimized data structures for best performance

## 9. Final Summary

### Overall Performance:

* Kruskal Superiority: 85.7% of test cases
* Average Speedup: 22.6% with Kruskal
* Scalability: Kruskal maintains advantage across sizes

### Key Takeaway:

While theoretical complexities are similar, practical performance favors Kruskal due to optimized library functions and better cache behavior. The choice between algorithms should consider graph density and specific application requirements.