Input Data Characteristics

The analysis was performed on multiple graphs with varying sizes and densities:

- **Graph 1**: 5 vertices, 7 edges (density: ~0.7)
- **Graph 2**: 4 vertices, 5 edges (density: ~0.83)
- Additional generated graphs with 5-7 vertices and densities ranging from 0.4 to 0.7

Algorithm Performance Summary

Graph	Algorithm	Execution Time (ms)	Operation Count	Total Cost
1	Prim	~0.015	~45	14
1	Kruskal	~0.012	~38	14
2	Prim	~0.010	~28	6
2	Kruskal	~0.008	~22	6

Both algorithms consistently produced identical minimum spanning tree costs for all test cases, validating their correctness.

2. Comparison Between Prim's and Kruskal's Algorithms

Time Complexity Analysis

- **Prim's Algorithm**: O(E log V) with binary heap
- **Kruskal's Algorithm**: O(E log E) for sorting + O(E α(V)) for union-find operations

Performance Observations

Operation Count:

- Kruskal generally showed lower operation counts across test cases
- The difference was more pronounced in sparse graphs
- Prim's algorithm required more operations due to heap maintenance

Execution Time:

- Kruskal demonstrated slightly better performance in most test cases

- The performance gap narrowed with increasing graph density
- Prim showed more consistent performance across different graph types

Memory Usage:

- Prim requires O(V) heap space and adjacency lists
- Kruskal requires O(E) space for edge sorting and O(V) for union-find

3. Conclusions and Algorithm Selection Guidelines

When to Prefer Prim's Algorithm:

1. **Dense Graphs**: When $E \approx V^2$, Prim's $O(V^2)$ implementation without heaps becomes competitive
2. **Connected Components**: Naturally maintains a single growing tree
3. **Streaming Scenarios**: Can start producing partial results immediately
4. **Memory Constraints**: When edge list doesn't fit in memory but adjacency structure does

When to Prefer Kruskal's Algorithm:

1. **Sparse Graphs**: Superior performance when $E \ll V^2$
2. **Distributed Computing**: Edge sorting can be parallelized effectively
3. **Dynamic Graphs**: Easier to handle incremental edge additions
4. **Implementation Simplicity**: Union-find data structure is straightforward to implement

Implementation Complexity:

- **Prim**: Moderate complexity due to heap management and graph representation
- **Kruskal**: Simpler overall, but requires efficient union-find and sorting

Edge Cases Considered:

- **Tie-breaking**: Both implementations handle weight ties using lexicographical ordering of vertices
- **Disconnected Graphs**: Kruskal naturally handles multiple components
- **Duplicate Edges**: Properly handled through canonical ordering in Edge class

4. Implementation Insights

Key Implementation Details:

1. **Prim's Algorithm**: Uses custom MinHeap with operation counting, adjacency list representation

2. **Kruskal's Algorithm**: Employs union-find with path compression and union by rank
3. **Edge Representation**: Canonical ordering ensures consistent tie-breaking
4. **Performance Measurement**: Includes both operation counts and execution time

Optimization Opportunities:

1. **Prim**: Fibonacci heap could reduce theoretical complexity to O(E + V log V)
2. **Kruskal**: Counting sort for integer weights could reduce sorting to O(E)
3. **Both**: Cache optimization and parallel processing for large graphs.