MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE AND AUTOMATION ENGINEERING

ANALYSIS OF ALGORITHMS
LABORATORY WORK #5

Study and empirical analysis of greedy algorithms: Prim's Algorithm and Kruskal's Algorithm

Author:
Andrei Chicu
std. gr. FAF–233

Verified:
Fistic Cristofor
Department of SEA, FCIM UTM

Chișinău, 2025

1 Analysis of Algorithms

github url: `https://github.com/andyp1xe1/aa_labs/tree/main/lab5`

1.1 Objective

The objective of this laboratory work is to implement and analyze two fundamental greedy algorithms for finding minimum spanning trees: Prim's Algorithm and Kruskal's Algorithm. The analysis aims to compare their performance across various graph types and sizes to understand their efficiency characteristics, time complexity behavior, and practical applications in different graph scenarios.

1.2 Tasks

1 Study the greedy algorithm design technique. 2 To implement in a programming language algorithms Prim and Kruskal. 3 Empirical analyses of the Kruskal and Prim 4 Increase the number of nodes in graph and analyze how this influences the algorithms. Make a graphical presentation of the data obtained 5 To make a report.

1.3 Theoretical Notes

This lab explores greedy algorithms for finding minimum spanning trees in graphs. Below, we discuss the theoretical background and algorithms implemented.

1.3.1 Minimum Spanning Tree

A minimum spanning tree (MST) of a connected, undirected, weighted graph is a tree that spans all vertices with the minimum possible total edge weight. MSTs have numerous practical applications in network design, circuit wiring, cluster analysis, and approximation algorithms for NP-hard problems like the Traveling Salesman Problem. The key property of an MST is that it provides the lowest-cost way to connect all vertices in a graph, ensuring network connectivity while minimizing resource usage (e.g., cable length, construction costs). Both Prim's and Kruskal's algorithms exploit the greedy property that locally optimal choices at each step lead to a globally optimal spanning tree. For any connected graph, the MST is unique if all edge weights are distinct; otherwise, multiple MSTs may exist with the same total weight.

1.3.2 Types of Graphs

In this laboratory, we analyze the algorithms on the following types of graphs:

1. Complete Graph Every vertex is connected to every other vertex, resulting in |V|(|V|-1)/2 edges in an undirected graph.

2. Dense Graph Has approximately 80% of the maximum possible edges.

3. Sparse Graph Contains relatively few edges, approximately 2|V| edges.

4. Connected Graph There exists a path between any pair of vertices.

5. Grid Graph Vertices arranged in a grid-like structure with connections primarily to adjacent nodes.

1.3.3 Greedy Algorithms

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach. This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

1.3.4 Prim's Algorithm

Prim's algorithm builds the MST one vertex at a time, starting from an arbitrary root vertex and at each step adding the lowest-weight edge that connects the tree to a vertex not yet in the tree. The algorithm works as follows:

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph

2. Grow the tree by one edge at a time: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge and add it to the tree

3. Repeat step 2 until all vertices are in the tree

Prim's algorithm has time complexity O(E log V) when implemented with a binary heap, where E is the number of edges and V is the number of vertices.

1.3.5 Kruskal's Algorithm

Kruskal's algorithm builds the MST by adding edges in order of increasing weight, skipping edges that would create a cycle. The algorithm works as follows:

1. Sort all the edges in non-decreasing order of their weight

2. Initialize an empty forest (set of trees) where each vertex is a separate tree
3. For each edge in the sorted list:
    - If adding this edge does not create a cycle (vertices are in different trees), add it to the forest and merge the trees
    - Otherwise, discard this edge
4. Continue until the forest becomes a tree (or all edges have been processed)

Kruskal's algorithm has time complexity O(E log E) or equivalently O(E log V), dominated by the sorting step, where E is the number of edges and V is the number of vertices.

1.4 Comparison Metric

For this empirical analysis, we measure the following metrics:

1. **Execution Time**: The primary metric is the actual execution time in seconds for each algorithm across different graph types and sizes. This provides an empirical measure of the algorithm's efficiency.
2. **Number of Edges in MST**: This will always be V-1 for a connected graph with V vertices, serving as a validation check.
3. **Total Weight of MST**: The sum of weights of all edges in the resulting MST. Both algorithms should produce the same total weight for the same input graph.

The algorithms are evaluated on various graph types with sizes ranging from 100 to 2500/10,000 vertices to observe how they scale with input size.

```
sizes = [10, 50, 60, 70, 80, 90, 100, 150, 200, 300]
mid_sizes = [100, 200, 300, 500, 700, 1000]
```

For each graph type and size, algorithms were executed with 5 repetitions to account for system variability and provide statistical robustness. This approach allows us to:

- Calculate mean execution times for more reliable performance comparison
- Determine standard deviation to assess result consistency and reliability
- Identify potential outliers or anomalous behavior in specific test cases

The error bars in the plots represent the standard deviation across these repetitions, providing a visual indication of measurement variability.

1.5 Input Format

In this laboratory work, graphs are represented using adjacency lists implemented as nested Python dictionaries. Each vertex is a key in the outer dictionary, and its value is another dictionary mapping neighboring vertex IDs to edge weights.

For MST algorithms, we convert the directed graph representation to an undirected one by ensuring each edge appears in both directions with the same weight.

Example input graph structure:

```
graph = {
    '0': {'1': 5, '2': 3},
    '1': {'0': 5, '3': 2, '4': 4},
    '2': {'0': 3},
    '3': {'1': 2},
    '4': {'1': 4, '2': 1}
}
```

In this representation:

- Vertices ’0’ and ’1’ are connected by an edge with weight 5
- Vertices ’0’ and ’2’ are connected by an edge with weight 3
- Vertices ’1’ and ’3’ are connected by an edge with weight 2
- Vertices ’1’ and ’4’ are connected by an edge with weight 4
- Vertices ’4’ and ’2’ are connected by an edge with weight 1

All graphs are generated with the functions provided in the uploaded code, with vertices labeled as strings from ’0’ to ’n-1’ and edge weights ranging from 1 to 10.

2 Implementation

2.1 Prim’s Algorithm

Prim’s algorithm is implemented using a priority queue to efficiently find the minimum-weight edge at each step. We use a min-heap data structure from Python’s heapq module.

```python
def prims(graph, start=None):
    import heapq

    # Extract all vertices
    vertices = list(graph.keys())

    if not vertices:
        return [], [], 0

    # Use provided start vertex or default to first vertex
    if start is None or start not in vertices:
        start = vertices[0]

    # Initialize tracking structures
    visited = {start}
    tree_edges = []
    explored_edges = []

    # Priority queue to store edges (weight, from_vertex, to_vertex)
    edge_queue = []
    for neighbor, weight in graph[start].items():
        heapq.heappush(edge_queue, (weight, start, neighbor))

    while edge_queue and len(visited) < len(vertices):
        weight, from_vertex, to_vertex = heapq.heappop(edge_queue)

        # Record this exploration
        explored_edges.append((from_vertex, to_vertex, weight))

        if to_vertex in visited:
            continue

        # Add edge to minimum spanning tree
        tree_edges.append((from_vertex, to_vertex, weight))
        visited.add(to_vertex)

        # Add all edges from the new vertex to priority queue
        for neighbor, edge_weight in graph[to_vertex].items():
            if neighbor not in visited:
                heapq.heappush(edge_queue, (edge_weight, to_vertex, neighbor))

    # Calculate total weight of MST
    mst_weight = sum(weight for _, _, weight in tree_edges)

    return tree_edges, explored_edges, mst_weight
```

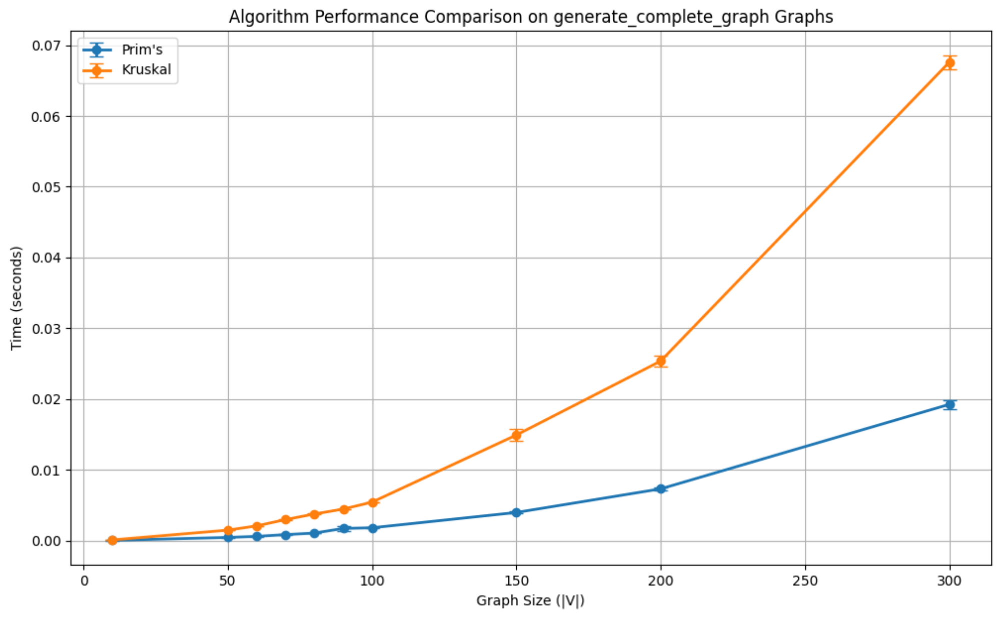
Listing 1: Implementation of Prim's Algorithm

2.2 Kruskal's Algorithm

Kruskal's algorithm is implemented using a disjoint-set data structure (Union-Find) to efficiently detect cycles. We sort edges by weight and add them to the MST if they don't create cycles.

3 Results

3.1 Complete Graph



For complete graphs, the execution time for both algorithms shows a clear polynomial growth pattern as graph size increases. From the data presented in the figure, Kruskal's algorithm consistently performs worse than Prim's algorithm across all tested graph sizes.

As we can observe, for a complete graph with 300 vertices, Prim's algorithm takes approximately 0.02 seconds, while Kruskal's algorithm requires about 0.068 seconds - more than three times longer. This significant performance gap widens as the graph size increases.

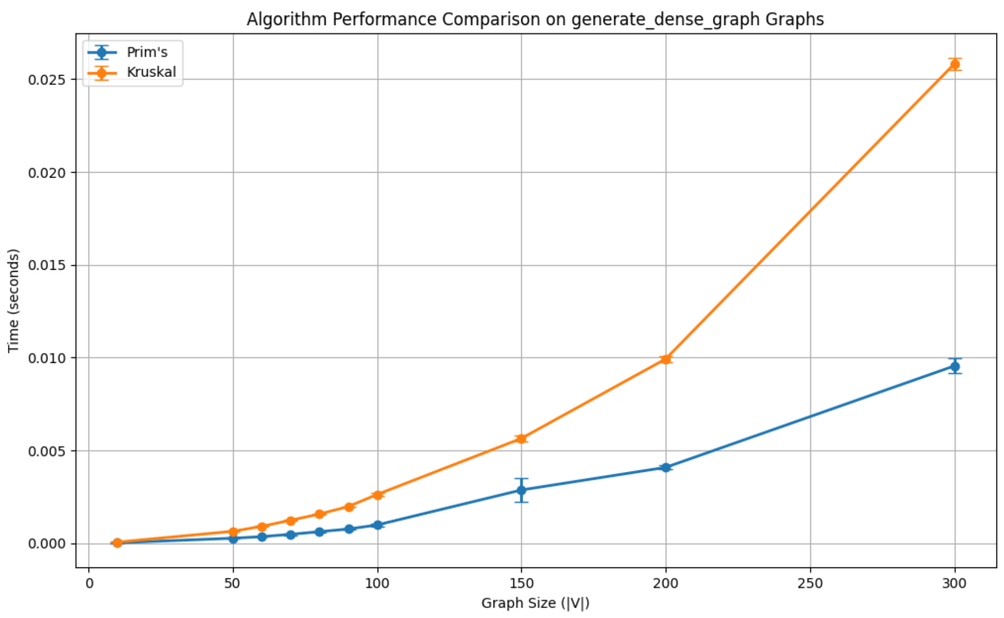
This performance difference aligns with theoretical expectations. In complete graphs with |V| vertices, there are O(V^2) edges. Kruskal's algorithm must sort all these edges, which becomes expensive as graph size increases. In contrast, Prim's algorithm benefits from its approach of directly processing edges from the adjacency list using a priority queue, avoiding the need to sort all edges upfront.

The standard deviation (shown as error bars) is relatively small for both algorithms, indicating consistent performance across test repetitions.

```python
def kruskal(graph, start=None):
    # Define disjoint-set data structure for cycle detection
    class DisjointSet:
        def __init__(self, vertices):
            self.parent = {v: v for v in vertices}
            self.rank = {v: 0 for v in vertices}

        def find(self, vertex):
            if self.parent[vertex] != vertex:
                self.parent[vertex] = self.find(self.parent[vertex])
            return self.parent[vertex]

        def union(self, vertex1, vertex2):
            root1 = self.find(vertex1)
            root2 = self.find(vertex2)

            if root1 != root2:
                if self.rank[root1] < self.rank[root2]:
                    self.parent[root1] = root2
                elif self.rank[root1] > self.rank[root2]:
                    self.parent[root2] = root1
                else:
                    self.parent[root2] = root1
                    self.rank[root1] += 1
                return True
            return False

    # Extract all vertices
    vertices = list(graph.keys())

    if not vertices:
        return [], [], 0

    # Collect all edges
    edges = []
    for vertex in vertices:
        for neighbor, weight in graph[vertex].items():
            # Prevent duplicate edges by only adding if vertex < neighbor
            if vertex < neighbor:
                edges.append((weight, vertex, neighbor))

    # Sort edges by weight
    edges.sort()

    # Initialize DisjointSet
    ds = DisjointSet(vertices)

    # Track results
    tree_edges = []
    explored_edges = []

    # Process edges in ascending order of weight
    for weight, vertex1, vertex2 in edges:
        # Record this exploration
        explored_edges.append((vertex1, vertex2, weight))

        if ds.union(vertex1, vertex2):
            # Edge is part of MST
            tree_edges.append((vertex1, vertex2, weight))

    # Calculate total weight of MST
    mst_weight = sum(weight for _, _, weight in tree_edges)

    return tree_edges, explored_edges, mst_weight
```

Listing 2: Implementation of Kruskal’s Algorithm

3.2 Dense Graph



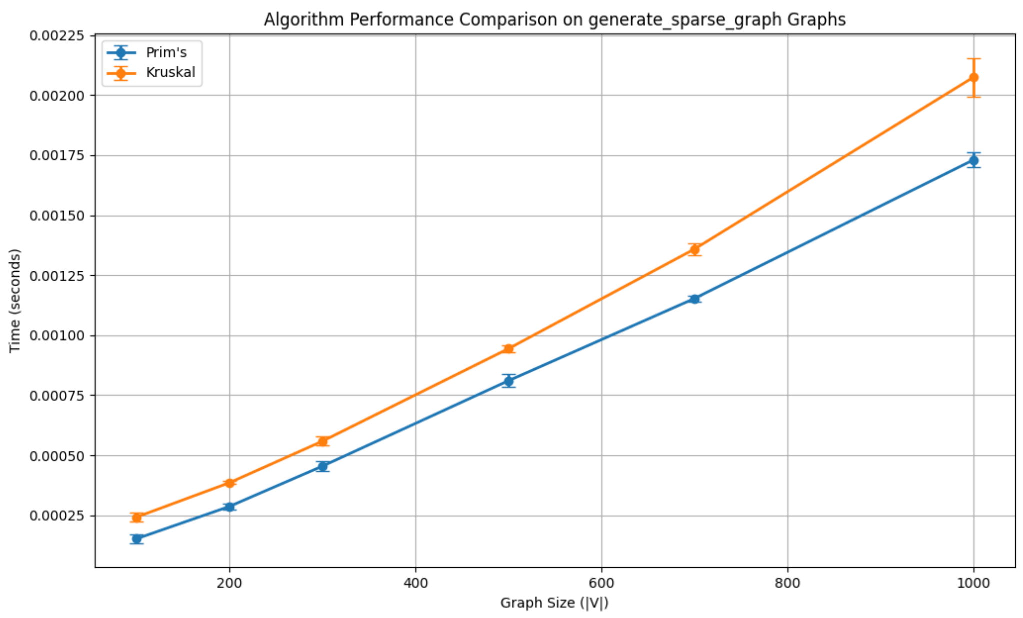
In dense graphs (approximately 80% of maximum possible edges), we observe similar performance patterns to complete graphs, but with proportionally reduced execution times.

Based on the figure, Prim's algorithm maintains a substantial advantage over Kruskal's algorithm. For a dense graph with 300 vertices, Prim's algorithm takes approximately 0.01 seconds, while Kruskal's algorithm requires about 0.026 seconds - still more than twice as long.

This performance difference is consistent with what we observed in complete graphs and reinforces that Prim's algorithm is better suited for graphs with high edge density. The algorithm's approach of growing a single tree from a starting vertex efficiently handles densely connected graphs where there are many candidate edges to consider at each step.

The error bars remain small, indicating reliable and consistent measurements across test runs.

3.3 Sparse Graph



For sparse graphs (with approximately 2|V| edges), the performance characteristics of both algorithms change significantly compared to dense graphs.

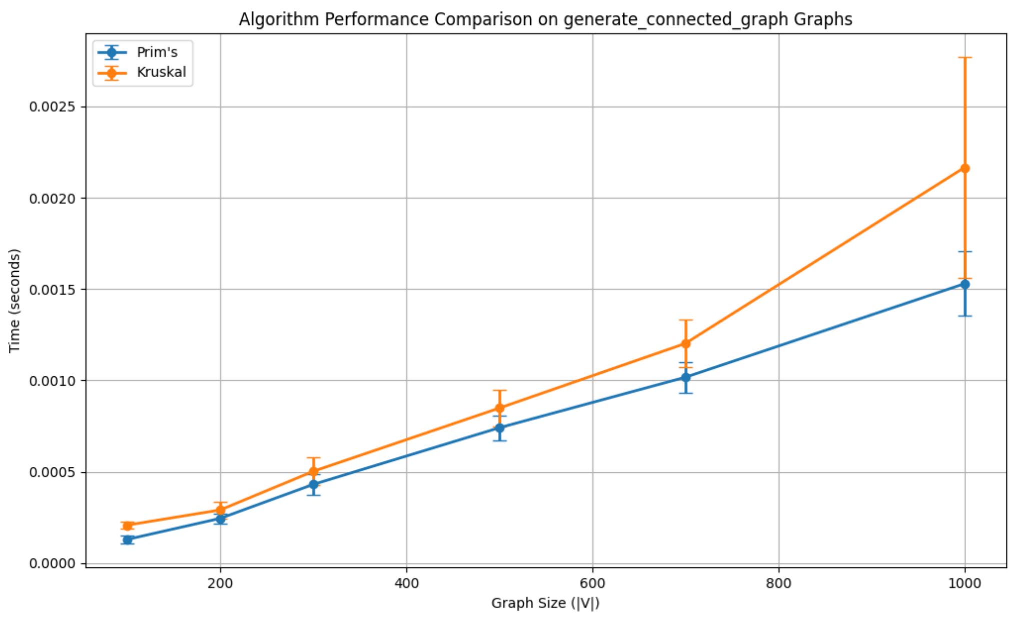
Based on the figure, both algorithms exhibit a nearly linear growth pattern with increasing graph size, which is consistent with their theoretical O(E log V) complexity when E is proportional to V.

Notably, in sparse graphs, both algorithms perform comparably, with slightly better performance from Prim's algorithm. For a sparse graph with 1000 vertices, Prim's algorithm takes approximately 0.00175 seconds, while Kruskal's algorithm requires about 0.0021 seconds.

This relatively small difference (compared to the much larger gap seen in dense graphs) can be explained by Kruskal's algorithm needing to sort far fewer edges in sparse graphs. While Prim's algorithm still maintains an advantage, the performance gap is significantly reduced.

The standard deviation is slightly higher than in dense graphs but still indicates consistent measurements across test runs.

3.4 Connected Graph



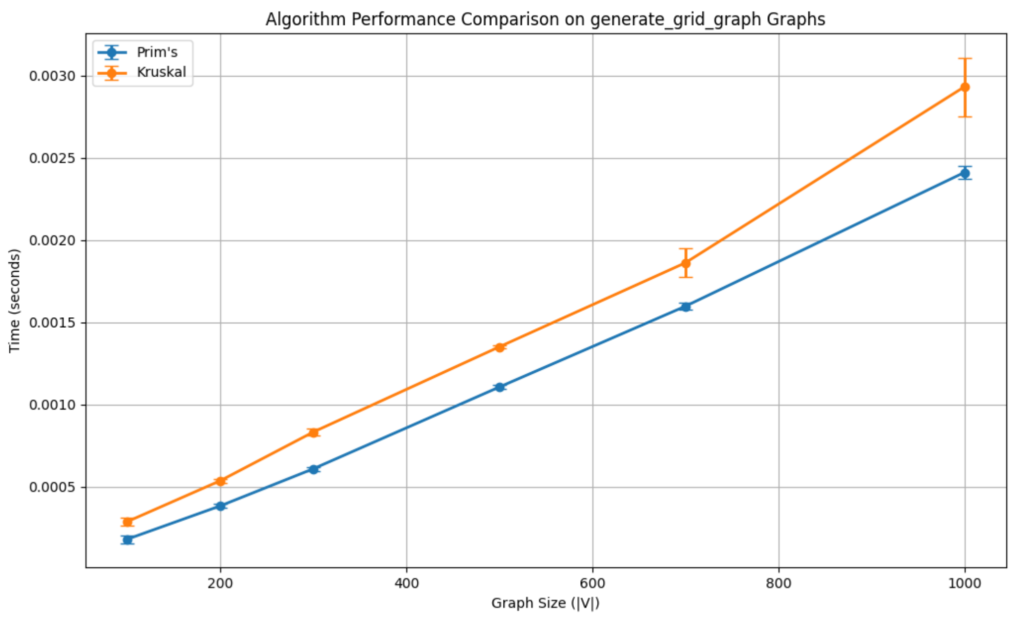
Connected graphs show a linear growth pattern for both algorithms, similar to sparse graphs.

According to the figure, both Prim's and Kruskal's algorithms perform comparably, with Kruskal's algorithm showing slightly higher execution times. For a connected graph with 1000 vertices, Prim's algorithm takes approximately 0.0015 seconds, while Kruskal's requires around 0.0022 seconds.

This performance difference is consistent but not as pronounced as in dense or complete graphs. The connected graphs used in this test likely have moderate edge density, leading to balanced performance between the two algorithms.

The error bars increase with graph size but remain relatively small, indicating reliable measurements with some expected variability.

3.5 Grid Graph



Grid graphs present an interesting case for MST algorithms due to their regular structure.

Based on the figure, both algorithms show linear growth with graph size, but Prim's algorithm consistently outperforms Kruskal's algorithm. For a grid graph with 1000 vertices, Prim's algorithm takes approximately 0.0024 seconds while Kruskal's requires around 0.0029 seconds.

The performance advantage of Prim's algorithm for grid graphs is noteworthy considering that grid graphs are relatively sparse (maximum 4 edges per vertex). This suggests that Prim's approach of growing a connected tree is particularly efficient for the structured nature of grid graphs, where edges have a predictable pattern of connectivity.

The standard deviation is small relative to the mean values, indicating consistent measurements across test runs.

4 Conclusions

This laboratory work demonstrates that the relative performance of Prim's and Kruskal's algorithms varies based on graph density and structure, despite having the same asymptotic complexity of O(E log V).

The empirical analysis reveals several key insights:

Prim's algorithm consistently outperforms Kruskal's algorithm in dense and complete graphs. For complete graphs with 300 vertices, Prim's algorithm is approximately three times faster than Kruskal's. This advantage stems from Prim's efficient approach of growing a single tree from a starting vertex, which avoids the costly sorting of all edges required by Kruskal's algorithm.

In sparse graphs, connected graphs, and grid structures, the performance difference between the two algorithms is less pronounced. Prim's algorithm still maintains an advantage in most cases, but the margin is smaller. This reduced gap is due to the smaller number of edges that Kruskal's algorithm needs to sort.

Both algorithms scale according to their theoretical complexity predictions, with execution time growing as O(E log V). The results confirm that the constant factors and actual performance characteristics vary significantly based on graph structure.

Implementation details matter significantly. Our implementation uses a binary heap for Prim's algorithm and a disjoint-set data structure for Kruskal's algorithm. Different data structures could potentially change the performance characteristics.

The consistency of results across repetitions (small standard deviations) indicates that both algorithms have stable performance characteristics, making them reliable choices for MST problems.

These findings highlight the importance of understanding both the theoretical complexity and practical performance characteristics when choosing between MST algorithms. While both algorithms are viable solutions for finding minimum spanning trees, Prim's algorithm appears to be the better choice in most scenarios tested, particularly for denser graphs.

For applications where graph density is known in advance, this analysis provides clear guidance on algorithm selection: use Prim's algorithm for dense graphs, and consider either algorithm for sparse or structured graphs with a slight preference for Prim's if performance is critical.