Minimum Spanning Tree and Shortest Path Algorithms Analysis

Executive Summary

This report presents a comprehensive analysis of Minimum Spanning Tree (MST) and Shortest Path algorithms implemented and tested on various graph datasets. The algorithms implemented include Prim's and Kruskal's algorithms for MST, and Dijkstra's algorithm for shortest path computation.

The analysis reveals that for small graphs (fewer than 10 vertices), all algorithms perform efficiently with negligible execution times. However, for larger graphs (200+ vertices), significant performance differences emerge. Kruskal's algorithm shows higher execution times compared to Prim's algorithm for MST computation, primarily due to the edge sorting step. Dijkstra's algorithm demonstrates efficient performance for shortest path computation, comparable to Prim's algorithm.

Based on our findings, we recommend using Prim's algorithm for MST computation in dense graphs and Dijkstra's algorithm for shortest path problems. Kruskal's algorithm may be more suitable for very sparse graphs despite its higher overhead from sorting edges.

1. Introduction

Graph algorithms are fundamental in computer science and have numerous applications in network design, transportation planning, circuit design, and many other fields. This report focuses on two important classes of graph algorithms:

1. **Minimum Spanning Tree (MST) Algorithms**: These algorithms find a subset of edges that connect all vertices in a graph with the minimum possible total edge weight.
2. **Shortest Path Algorithms**: These algorithms find the path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

The primary objectives of this analysis are:

- To implement and compare the performance of Prim's and Kruskal's algorithms for MST computation
- To implement and analyze Dijkstra's algorithm for shortest path computation

- To evaluate the performance of these algorithms on various graph datasets
- To identify factors that affect algorithm performance

2. Algorithm Descriptions

2.1 Prim's Algorithm for MST

Prim's algorithm builds the MST by adding edges one at a time, starting from an arbitrary vertex and always adding the lowest-weight edge that connects a vertex in the MST to a vertex outside the MST.

Pseudocode:

```
function Prim(G, start):
    Initialize empty MST
    Initialize priority queue PQ
    For each vertex v in G:
        Set key[v] = infinity
        Set parent[v] = null
    Set key[start] = 0
    Add all vertices to PQ with their key values

    While PQ is not empty:
        u = Extract-Min(PQ)
        Add u to MST
        If parent[u] is not null:
            Add edge (parent[u], u) to MST

        For each neighbor v of u:
            If v is in PQ and weight(u,v) < key[v]:
                Set parent[v] = u
                Set key[v] = weight(u,v)
                Update v's key in PQ

    Return MST
```

Time Complexity: $O(V^2)$ without a binary heap, $O(E \log V)$ with a binary heap

2.2 Kruskal's Algorithm for MST

Kruskal's algorithm builds the MST by considering edges in ascending order of weight and adding them to the MST if they don't create a cycle.

Pseudocode:

```
function Kruskal(G):
    Initialize empty MST
    Sort all edges in G in non-decreasing order of weight
    Initialize disjoint-set data structure for each vertex

    For each edge (u,v) in sorted edges:
        If Find-Set(u) != Find-Set(v):
            Add edge (u,v) to MST
            Union(u,v)

    Return MST
```

Time Complexity: O(E log E) or O(E log V)

2.3 Dijkstra's Algorithm for Shortest Path

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights.

Pseudocode:

```
function Dijkstra(G, start):
    Initialize distance[v] = infinity for all vertices v
    Set distance[start] = 0
    Initialize priority queue PQ with (0, start)
    Initialize empty set S of visited vertices

    While PQ is not empty:
        (dist, u) = Extract-Min(PQ)

        If u is in S:
            Continue

        Add u to S

        For each neighbor v of u:
            If distance[u] + weight(u,v) < distance[v]:
                Set distance[v] = distance[u] + weight(u,v)
                Add (distance[v], v) to PQ

    Return distance
```

Time Complexity: $O(V^2)$ without a binary heap, O(E log V) with a binary heap

3. Implementation Details

All algorithms were implemented in Python using the following data structures:

- **Graph Representation**: Adjacency list representation using dictionaries
- **Priority Queue**: Binary heap implementation using Python's `heapq` module
- **Disjoint-Set**: Union-find data structure with path compression and union by rank optimizations

The implementation includes:

1. A `Graph` class with methods for adding vertices and edges, and for running the MST and shortest path algorithms
2. Functions for loading graph data from various file formats
3. Analysis scripts for measuring and comparing algorithm performance

4. Graph Datasets

The analysis was performed on the following graph datasets:

1. **Cities Graph**: A graph representing major European cities with 10 vertices and 28 edges. Edge weights represent distances between cities.
2. **Cyclic Graph**: A simple cyclic graph with 6 vertices and 12 edges.
3. **Random Graph**: A randomly generated graph with 8 vertices and 26 edges.
4. **Standard Graph**: A standard test graph with 6 vertices and 18 edges.
5. **Large Graph**: A randomly generated large graph with 200 vertices and 3972 edges.

5. Performance Results

5.1 Execution Times (in nanoseconds)

Graph Name	Vertices	Edges	Prim (ns)	Kruskal (ns)	Dijkstra (ns)
Cities Graph	10	28	18,014.19	15,537.74	13,411.52
Cyclic Graph	6	12	5,001.78	9,430.41	5,914.69
Random Graph	8	26	9,119.75	18,768.79	9,112.12
Standard Graph	6	18	5,031.11	16,478.78	6,213.19

Graph Name	Vertices	Edges	Prim (ns)	Kruskal (ns)	Dijkstra (ns)
Large Graph	200	3972	1,008,510.59	136,692,667.01	1,009,988.78

Note: Times are measured in nanoseconds (ns) for maximum precision (1 nanosecond = 0.001 microsecond = 0.000001 millisecond).

5.2 Execution Times (in microseconds for easier reading)

Graph Name	Vertices	Edges	Prim (μs)	Kruskal (μs)	Dijkstra (μs)
Cities Graph	10	28	18.01	15.54	13.41
Cyclic Graph	6	12	5.00	9.43	5.91
Random Graph	8	26	9.12	18.77	9.11
Standard Graph	6	18	5.03	16.48	6.21
Large Graph	200	3972	1,008.51	136,692.67	1,009.99

5.3 Relative Performance

To better visualize the relative performance differences, we can normalize the execution times relative to the fastest algorithm for each graph:

Graph Name	Prim (relative)	Kruskal (relative)	Dijkstra (relative)
Cities Graph	1.34	1.16	1.00
Cyclic Graph	1.00	1.89	1.18
Random Graph	1.00	2.06	1.00
Standard Graph	1.00	3.28	1.24
Large Graph	1.00	135.54	1.00

Note: Values are normalized relative to the fastest algorithm for each graph (lower is better).

6. Analysis of Factors Affecting Performance

6.1 Number of Vertices

The number of vertices in a graph significantly impacts the performance of graph algorithms:

- **Prim's Algorithm**: $O(V^2)$ without a binary heap, $O(E \log V)$ with a binary heap
- **Kruskal's Algorithm**: $O(E \log E)$ or $O(E \log V)$
- **Dijkstra's Algorithm**: $O(V^2)$ without a binary heap, $O(E \log V)$ with a binary heap

With our nanosecond-precision measurements, we can now observe meaningful execution times even for small graphs. For graphs with fewer than 10 vertices, execution times range from 5,000 to 19,000 nanoseconds (5-19 microseconds). For the large graph with 200 vertices, execution times increase significantly, particularly for Kruskal's algorithm.

6.2 Number of Edges

The number of edges plays a crucial role in algorithm performance:

- **Prim's Algorithm**: Performance is more affected by the number of vertices than edges when using a binary heap.
- **Kruskal's Algorithm**: Significantly affected by the number of edges since it sorts all edges by weight.
- **Dijkstra's Algorithm**: Similar to Prim's, it's more affected by vertices than edges when using a binary heap.

In our large graph with 3972 edges, Kruskal's algorithm showed the highest execution time (136,692,667 nanoseconds or ~136.7 milliseconds) compared to Prim's (1,008,510 nanoseconds or ~1.0 milliseconds) and Dijkstra's (1,009,988 nanoseconds or ~1.0 milliseconds). This represents a 135.5x performance difference between Kruskal's and the other algorithms, which aligns with theoretical expectations since Kruskal's algorithm needs to sort all edges.

Interestingly, our nanosecond-precision measurements revealed that for smaller graphs, the performance characteristics are more nuanced than previously observed:

- For the Cities Graph (28 edges): Dijkstra's algorithm is actually the fastest (13,411 ns), followed by Kruskal's (15,537 ns) and then Prim's (18,014 ns)
- For the Cyclic Graph (12 edges): Prim's algorithm is the fastest (5,001 ns), followed by Dijkstra's (5,914 ns) and then Kruskal's (9,430 ns)
- For the Random Graph (26 edges): Dijkstra's algorithm is marginally faster (9,112 ns) than Prim's (9,119 ns), with Kruskal's being significantly slower (18,768 ns)
- For the Standard Graph (18 edges): Prim's algorithm is the fastest (5,031 ns), followed by Dijkstra's (6,213 ns) and then Kruskal's (16,478 ns)

This pattern confirms that the edge sorting step in Kruskal's algorithm has a significant impact on performance as the number of edges increases.

6.3 Graph Structure

The structure of the graph can significantly impact algorithm performance:

- **Cyclic vs. Acyclic**: Cycles require additional checks in MST algorithms, particularly in Kruskal's algorithm where the union-find data structure is used to detect cycles.
- **Connected vs. Disconnected**: Disconnected graphs may require special handling, especially for MST algorithms that assume connectivity.
- **Weight Distribution**: Uniform weights vs. varied weights can affect the performance of priority queue operations.

6.4 Implementation Details

Implementation details can significantly affect performance:

- **Data Structures**: Our implementation uses binary heaps (via Python's heapq) for Prim's and Dijkstra's algorithms, and a disjoint-set data structure for Kruskal's algorithm.
- **Language and Runtime**: Python's interpreted nature may introduce overhead compared to compiled languages.

7. Conclusion

Based on our detailed analysis with nanosecond-precision measurements:

1. **For Small Graphs** (< 100 vertices):

2. All three algorithms perform efficiently, with execution times ranging from 5,000 to 19,000 nanoseconds (5-19 microseconds).

3. The performance ranking varies by graph structure:

    - Prim's algorithm is fastest for graphs with fewer edges (Cyclic and Standard graphs)
    - Dijkstra's algorithm is fastest for graphs with more edges (Cities and Random graphs)
    - Kruskal's algorithm is generally slower than both Prim's and Dijkstra's for most small graphs, except for the Cities graph where it outperforms Prim's

4. **For Large Graphs** (≥ 100 vertices):

5. **Prim's Algorithm** is highly efficient for both sparse and dense graphs, with execution times around 1 millisecond (1,008,510 nanoseconds) for our 200-vertex graph.

6. **Kruskal's Algorithm** shows dramatically higher execution times (135.5x slower than Prim's) due to the edge sorting step, making it unsuitable for graphs with many edges.

7. **Dijkstra's Algorithm** performs almost identically to Prim's algorithm on large graphs, with execution times around 1 millisecond (1,009,988 nanoseconds) for our 200-vertex graph.

8. **Trade-offs**:

9. **Time vs. Space**: Prim's algorithm with a binary heap offers excellent time complexity but requires more space for the priority queue.

10. **Implementation Complexity**: Kruskal's algorithm requires implementing a union-find data structure, which adds complexity.

11. **Edge Density Impact**: The performance gap between Prim's and Kruskal's algorithms widens dramatically as the number of edges increases.

12. **Recommendations**:

13. For small graphs with fewer edges (E < 20), Prim's algorithm is generally the fastest choice for MST computation.
14. For small graphs with more edges (E ≥ 20), Dijkstra's algorithm surprisingly outperforms both MST algorithms for single-source shortest paths.
15. For large graphs, both Prim's and Dijkstra's algorithms provide excellent performance, while Kruskal's algorithm should be avoided due to its O(E log E) sorting step.
16. When graph structure is unknown, Dijkstra's algorithm offers the most consistent performance across different graph types and sizes.

8. Future Work

1. Implement and compare other shortest path algorithms (e.g., Bellman-Ford, Floyd-Warshall).
2. Test on even larger graphs to better observe performance differences.
3. Optimize implementations for better performance.
4. Analyze memory usage alongside execution time.
5. Implement parallel versions of these algorithms to leverage multi-core processors.

Appendix A: Code Implementation

The implementation of the algorithms can be found in the following files:

1. `graph.py` : Contains the Graph class and algorithm implementations
2. `graph_loader.py` : Contains functions for loading graph data from files
3. `analyze_graphs.py` : Contains code for analyzing algorithm performance
4. `visualize_graphs.py` : Contains code for visualizing graphs and MSTs

MST and Shortest Path Algorithms Analysis Report