Ministerul Educației și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory Work 5

Study and empirical analysis of sorting algorithms
Analysis of Prim, Kruskal

Elaborated:
st. gr. FAF-233 Moraru Patricia

Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

Contents

Chapter 1

Introduction

1.1 Objective

The primary objective of this laboratory work is to conduct a comprehensive empirical analysis of two fundamental algorithms used for generating **Minimum Spanning Trees (MST)** in weighted graphs: **Prim's Algorithm** and **Kruskal's Algorithm**. These algorithms play a critical role in network design, clustering, and circuit design, among other applications. By analyzing their performance on a wide variety of graph configurations, we aim to develop an informed understanding of how each algorithm behaves under practical constraints such as graph size, density, and connectivity.

The study involves implementing both algorithms, executing them on various classes of graphs, and measuring performance metrics such as **execution time** and **memory usage**. The analysis seeks to determine not only which algorithm is faster or more efficient in different scenarios but also how graph structure influences their overall computational behavior.

1.2 Tasks

1. Implement Prim's and Kruskal's algorithms using a consistent programming framework.
2. Construct input graphs with various structural properties including sparsity, density, and connectivity.
3. Identify and select relevant performance metrics (execution time, memory usage).
4. Perform multiple empirical tests on graphs with increasing size and complexity.
5. Visualize results using graphical plots to highlight key differences and trends.
6. Formulate conclusions regarding the suitability of each algorithm for specific graph characteristics.

1.3 Mathematical vs. Empirical Analysis

While theoretical analysis provides insights into worst-case and average-case complexities, empirical analysis reveals practical behavior under real-world constraints. This dual approach ensures a balanced understanding of algorithm efficiency.

Mathematical Analysis	**Empirical Analysis**
The algorithm is analyzed via theoretical derivations, independent of specific input.	The algorithm is executed on actual data samples to observe real-world performance.
Ideal for estimating asymptotic behavior and proving correctness.	Crucial for understanding behavior in practice, including system-dependent effects.
Provides general expectations about scalability and limits.	Offers insights into real execution times, memory demands, and bottlenecks.

Table 1.1: Comparison of Mathematical and Empirical Analysis

1.4 Theoretical Notes on Empirical Analysis

Empirical analysis helps to bridge the gap between theory and practice. It complements Big-O complexity by observing actual runtime characteristics, which are influenced by constant factors, memory access patterns, and processor architecture.

1.4.1 Purpose of Empirical Analysis

- To validate theoretical expectations with practical measurements.
- To compare algorithmic performance across different graph models.
- To assess real-world behavior of algorithms in resource-constrained environments.
- To identify scenarios where one algorithm clearly outperforms the other.

1.5 Complexity of Algorithms

1.5.1 Prim's Algorithm

Prim's algorithm grows the MST by starting from a random node and greedily adding the lightest edge that expands the current tree. When implemented with a min-priority queue (e.g., a binary heap), its performance is well-optimized for dense graphs.

- **Time Complexity:** $O(E \log V)$ using a priority queue.
- **Space Complexity:** $O(V + E)$ to store the graph and queue.

1.5.2 Kruskal's Algorithm

Kruskal's algorithm begins by sorting all edges in non-decreasing order of weight and repeatedly adds the smallest edge that does not form a cycle, using the union-find data structure to detect cycles efficiently.

- **Time Complexity:** $O(E \log E)$ dominated by edge sorting and union-find operations.
- **Space Complexity:** $O(V + E)$ for storing edges and disjoint sets.

1.6 Empirical Analysis Process

1. **Define the Objective:** Compare and contrast Prim's and Kruskal's performance across different graph configurations in constructing MSTs.
2. **Choose Performance Metrics:** Execution time and peak memory usage during algorithm execution.
3. **Determine Graph Variations:**
   - Sparse vs. Dense
   - Connected vs. Disconnected
   - Cyclic vs. Acyclic
   - Trees (as minimal connected acyclic graphs)
   - Only connected graphs (valid input for MST generation)
4. **Implementation and Tools:** Graphs were generated programmatically using reproducible seeds to ensure consistency. Execution time was measured with the `timeit` module, and memory usage tracked with `tracemalloc`.
5. **Analysis and Visualization:** Results were plotted using Python visualization libraries to observe performance scaling and bottlenecks.

1.7 Minimum Spanning Tree Algorithms

Prim's Algorithm is typically more efficient on dense graphs due to fewer edge comparisons, especially when paired with a Fibonacci heap. It maintains a growing set of visited vertices and considers only the edges from the MST to the rest of the graph.

Kruskal's Algorithm, on the other hand, is particularly effective on sparse graphs due to its edge-centric nature. It doesn't require a fully connected graph from the start, which makes it suitable for working with disjoint sets and forests.

Understanding the behavior of these two algorithms under various graph structures is essential for selecting the right algorithm in practical applications such as network design, geographical routing, and image segmentation.

1.8 Comparison Metrics

To assess performance, the following criteria were used:

- **Execution Time:** The total time required to compute the MST from a given input graph, averaged over multiple runs.
- **Memory Usage:** The peak memory consumption during execution, which reflects the auxiliary data structures used by each algorithm.

1.9 Input Format

All input graphs were undirected and weighted, with sizes ranging over:

$$\{10, 15, 50, 75, 100, 250, 500\} \tag{1.1}$$

Each graph was generated under different structural constraints:

- **Sparse vs Dense:** Varying edge densities from $O(n)$ to $O(n^2)$.
- **Connected vs Disconnected:** Only connected components were retained for valid MST generation.
- **Cyclic vs Acyclic:** Testing edge redundancy and union-find efficiency.
- **Trees:** Edge cases where the graph is already a spanning tree ($E = V - 1$).
- **Only Connected (Weighted):** Filtered scenarios ensuring algorithm validity.

These input types enabled a holistic evaluation of algorithm robustness and adaptability under diverse graph topologies.

Chapter 2

Implementation

This section outlines the implementation details of the **Prim** and **Kruskal** algorithms used to compute Minimum Spanning Trees (MST) in weighted graphs. It also describes the experimental setup adopted for conducting a systematic empirical evaluation across various graph topologies.

Experimental Setup

To ensure the reliability and consistency of performance measurements, the experiments were conducted under controlled conditions with the following setup:

- **Execution Environment:** All tests were performed on a local personal computer with fixed hardware configuration to eliminate variability.
- **Graph Generator:** A custom graph generation module was employed to create graphs with varying characteristics—such as sparse/dense, connected/disconnected, cyclic/acyclic, and tree-structured. All graphs were weighted and undirected to ensure compatibility with MST algorithms.
- **Performance Metrics:** For every configuration, two primary metrics were recorded:
  - **Execution Time** using Python's `timeit` module.
  - **Memory Usage** using the `tracemalloc` library.
- **Repetition and Averaging:** Each test case was executed multiple times to ensure consistency and account for measurement noise. The final values reflect the average of those runs.

Graph Representations Used

In this implementation, the `networkx` library was used to construct and manage graphs. Although explicit data structures like adjacency matrices or custom adjacency lists were not manually implemented, the internal representation of `networkx.Graph()` closely mimics an adjacency list model and supports edge-centric operations.

Adjacency-Like Representation (networkx)

- Each node maps to a dictionary of its neighbors and edge attributes, enabling efficient neighbor iteration and weight access.
- The syntax `graph[u].items()` allows iterating through adjacent nodes of vertex u.
- This structure supports both Prim's algorithm variants by providing fast access to neighboring vertices and edge weights.

Edge List Representation

- For Kruskal's algorithm, the graph's edges were retrieved using `graph.edges(data=True)` and sorted by edge weight.
- This yields a flat edge list with metadata, making it suitable for greedy MST construction.
- The disjoint-set structure was used in conjunction to prevent cycle formation.

Thus, while traditional adjacency matrices or lists were not explicitly coded, the use of `networkx` provided both adjacency-list behavior and easy edge access for algorithmic operations, enabling clean and efficient implementations.

2.1 Prim's Algorithm

Prim's algorithm is a classic greedy approach for finding a Minimum Spanning Tree (MST) in a connected, undirected, weighted graph. It starts from an arbitrary vertex and grows the MST by repeatedly selecting the lightest edge that connects a node inside the tree to a node outside it. The algorithm ensures the final tree has the minimal total edge weight while covering all vertices.

Originally proposed by Czech mathematician Vojtěch Jarník in 1930 and independently rediscovered by Robert C. Prim in 1957, this algorithm is particularly well-suited for dense graphs when implemented with an efficient priority queue.

Core Concepts

- The MST is constructed incrementally, starting from a single vertex.
- A **visited set** tracks which nodes are already included in the MST.
- A **priority queue** is used to always select the next minimum-weight edge.
- Two variations are implemented: **lazy** (push all candidates) and **eager** (only push better connections).

Advantages

- Provides optimal MST for any connected, weighted, undirected graph.
- Can be efficiently implemented using heaps and adjacency structures.
- The eager version avoids redundant edge evaluations.

Limitations

- Inefficient for disconnected graphs (requires preprocessing).
- Less suitable for extremely sparse graphs compared to Kruskal.
- Requires additional care when edge weights are dynamic or updated in real time.

Lazy Prim's Implementation

In the lazy version, all eligible edges from the current MST boundary are pushed into the priority queue, including redundant ones that connect two already visited nodes. Only valid edges are accepted during extraction. While this version is conceptually simple, it suffers from extra insertions and checks, which slow down the algorithm, especially for large or dense graphs.

Python Implementation

```
def lazy_prim_mst(graph):
    n = len(graph.nodes)
    visited = [False] * n
    mst_edges = []
    priority_queue = []
    total_weight = 0

    def visit(u):
        visited[u] = True
        for v, data in graph[u].items():
            if not visited[v]:
                heapq.heappush(priority_queue, (data.get('weight', 1), u, v))

    visit(0)

    while priority_queue and len(mst_edges) < n - 1:
        weight, u, v = heapq.heappop(priority_queue)
        if visited[u] and visited[v]:
            continue
        mst_edges.append((u, v, weight))
        total_weight += weight
        if not visited[u]:
            visit(u)
        if not visited[v]:
            visit(v)
```

```python

    return total_weight
```

Listing 2.1: Lazy Prim's MST Algorithm

Eager Prim's Implementation

The eager version improves upon the lazy approach by tracking, for each unvisited node, the lightest known edge that could connect it to the MST. Instead of pushing all possible edges into the heap, it maintains and updates a "minimum edge map" as the tree grows. This significantly reduces heap operations and redundant edge processing.

Why it's better:

- Reduces the number of edges pushed into the priority queue.
- Achieves improved time complexity of $\mathcal{O}(E \log V)$ using a binary heap.
- Particularly advantageous for large and dense graphs where edge evaluations dominate runtime.
- Memory-efficient and cache-friendly due to fewer dynamic heap allocations.

Real-world use cases for this version include high-performance network design tools, graph-based machine learning models, and any setting where MST computation is frequent and time-critical.

Python Implementation

```python
def eager_prim_mst(graph):
    n = len(graph.nodes)
    visited = [False] * n
    min_edge = [(float('inf'), None)] * n
    min_edge[0] = (0, None)
    priority_queue = [(0, 0)]
    mst_edges = []
    total_weight = 0

    while priority_queue:
        weight, u = heapq.heappop(priority_queue)
        if visited[u]:
            continue
        visited[u] = True
        if min_edge[u][1] is not None:
            mst_edges.append((min_edge[u][1], u, weight))
            total_weight += weight
        for v, data in graph[u].items():
            edge_weight = data.get('weight', 1)
            if not visited[v] and edge_weight < min_edge[v][0]:
                min_edge[v] = (edge_weight, u)
                heapq.heappush(priority_queue, (edge_weight, v))

```

```python
    return total_weight
```

Listing 2.2: Eager Prim's MST Algorithm

2.2 Kruskal's Algorithm

Kruskal's algorithm is another greedy technique for constructing an MST. It operates by sorting all edges by weight and iteratively adding the next lightest edge that connects two previously unconnected components. To efficiently track connected components, it uses a **disjoint-set (union-find)** data structure.

This edge-centric approach makes Kruskal's algorithm particularly effective on sparse graphs or when the edges are known upfront.

Core Concepts

- All graph edges are sorted in ascending order of weight.
- A cycle detection mechanism (disjoint-set) is used to maintain acyclic structure.
- Each edge is considered for inclusion; only those connecting distinct components are added to the MST.

Advantages

- Efficient on sparse graphs with fewer edges.
- Can handle graphs that are initially disconnected (builds a forest).
- Straightforward to implement with edge sorting and union-find logic.

Limitations

- Requires pre-sorting of edges, which can be time-consuming on large dense graphs.
- The union-find structure must be carefully optimized for performance.
- Not naturally adaptive to incremental or real-time edge additions.

Basic Kruskal's Implementation

The basic version of Kruskal's algorithm uses a simple union-find structure for cycle detection. While it works correctly and demonstrates the core logic, it can become inefficient for large graphs due to the non-optimized path compression and union operations.

Python Implementation

```python
def kruskal_mst(graph):
    parent = {}
    def find(u):
        while parent[u] != u:
            parent[u] = parent[parent[u]]
            u = parent[u]
        return u

    def union(u, v):
        pu, pv = find(u), find(v)
        if pu == pv:
            return False
        parent[pu] = pv
        return True

    for node in graph.nodes:
        parent[node] = node

    edges = sorted(graph.edges(data=True), key=lambda x: x[2].get
        ('weight', 1))
    mst_weight = 0
    for u, v, data in edges:
        if union(u, v):
            mst_weight += data.get('weight', 1)

    return mst_weight
```

Listing 2.3: Basic Kruskal's MST Algorithm

Optimized Kruskal's Version with Disjoint Set

This enhanced version implements the disjoint-set data structure with both **path compression** and **union by rank**, significantly improving the performance of find and union operations.

Why it's better:

- Reduces the amortized complexity of each union/find operation to near-constant time: $\mathcal{O}(\alpha(n))$, where α is the inverse Ackermann function.
- Handles large-scale and batch-processing scenarios efficiently.
- Makes Kruskal's algorithm viable for real-time systems with massive edge sets or graphs with high dynamicity.

Applications include real-time communication routing, image segmentation in computer vision (e.g., graph-based clustering), and terrain modeling in GIS systems.

Python Implementation

```python
class DisjointSet:
    def __init__(self):
        self.parent = {}
        self.rank = {}

    def make_set(self, x):
        self.parent[x] = x
        self.rank[x] = 0

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        xroot = self.find(x)
        yroot = self.find(y)
        if xroot == yroot:
            return False
        if self.rank[xroot] < self.rank[yroot]:
            self.parent[xroot] = yroot
        else:
            self.parent[yroot] = xroot
            if self.rank[xroot] == self.rank[yroot]:
                self.rank[xroot] += 1
        return True

def kruskal_mst_optimized(graph):
    ds = DisjointSet()
    for node in graph.nodes:
        ds.make_set(node)

    edges = sorted(graph.edges(data=True), key=lambda x: x[2].get('weight', 1))
    mst_weight = 0
    for u, v, data in edges:
        if ds.union(u, v):
            mst_weight += data.get('weight', 1)
    return mst_weight
```

Listing 2.4: Optimized Kruskal's Algorithm with Disjoint Set

2.3 Performance Evaluation

This section presents the empirical results of applying minimum spanning tree algorithms—**Prim's** and **Kruskal's**—to various types of weighted, undirected graphs. The performance of each algorithm is evaluated using two primary metrics: **execution time** and **memory usage**.

The results are illustrated graphically to enable direct comparison and clear visualization of performance trends across different graph configurations, including sparse and dense graphs. These visualizations offer insight into how algorithmic design and underlying data structures (e.g., adjacency matrices vs. edge lists, priority queues vs. union-find) impact practical efficiency and scalability in constructing MSTs.

2.3.1 Sparse vs Dense Graphs – MST Algorithms

This experiment evaluates the performance of four Minimum Spanning Tree (MST) algorithms — Lazy Prim's, Eager Prim's, Kruskal's, and Optimized Kruskal's — across sparse and dense graph configurations. Each algorithm was tested on graphs ranging from 10 to 500 nodes. For each configuration, execution time and memory usage were recorded to assess the impact of edge density on performance and scalability.

Empirical Results Table

Table 2.1: Execution Time and Memory Usage – Sparse vs Dense Graphs (MST)

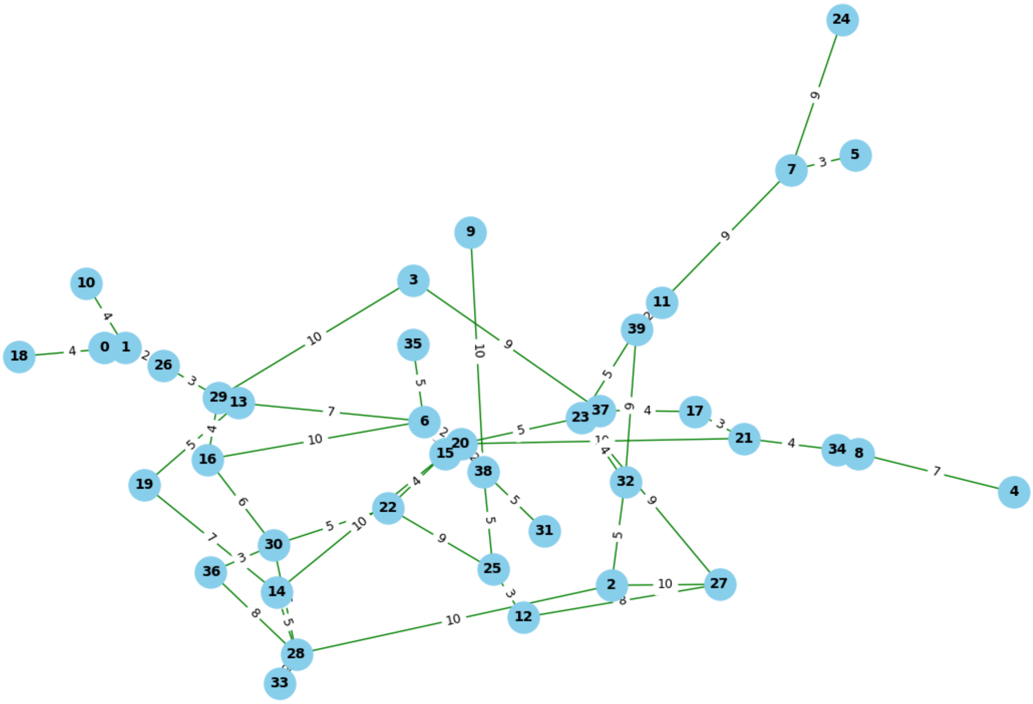
Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Lazy Prim's MST	10	Sparse	0.00041	832	32
Lazy Prim's MST	10	Dense	0.000206	1040	28
Lazy Prim's MST	15	Sparse	0.000306	1008	65
Lazy Prim's MST	15	Dense	0.000361	1144	39
Lazy Prim's MST	50	Sparse	0.00052	1472	167
Lazy Prim's MST	50	Dense	0.001203	2192	122
Lazy Prim's MST	75	Sparse	0.000356	1512	232
Lazy Prim's MST	75	Dense	0.000876	2704	204
Lazy Prim's MST	100	Sparse	0.000844	2168	437
Lazy Prim's MST	100	Dense	0.001809	3712	279
Lazy Prim's MST	250	Sparse	0.001977	152246	989
Lazy Prim's MST	250	Dense	0.004457	155852	731
Lazy Prim's MST	500	Sparse	0.003247	7908	2086
Lazy Prim's MST	500	Dense	0.009025	164106	1499
Eager Prim's MST	10	Sparse	7.8e-05	616	32
Eager Prim's MST	10	Dense	7.9e-05	776	28
Eager Prim's MST	15	Sparse	7.1e-05	776	65
Eager Prim's MST	15	Dense	0.000136	856	39
Eager Prim's MST	50	Sparse	0.000365	1448	167
Eager Prim's MST	50	Dense	0.000551	2088	122
Eager Prim's MST	75	Sparse	0.000283	1624	232
Eager Prim's MST	75	Dense	0.000659	2760	204
Eager Prim's MST	100	Sparse	0.000606	2520	437
Eager Prim's MST	100	Dense	0.000841	3720	279
Eager Prim's MST	250	Sparse	0.001305	5704	989
Eager Prim's MST	250	Dense	0.002626	8344	731
Eager Prim's MST	500	Sparse	0.003018	10548	2086
Eager Prim's MST	500	Dense	0.010737	16724	1499

Table 2.2: Execution Time and Memory Usage – Sparse vs Dense Graphs (MST, cont.)

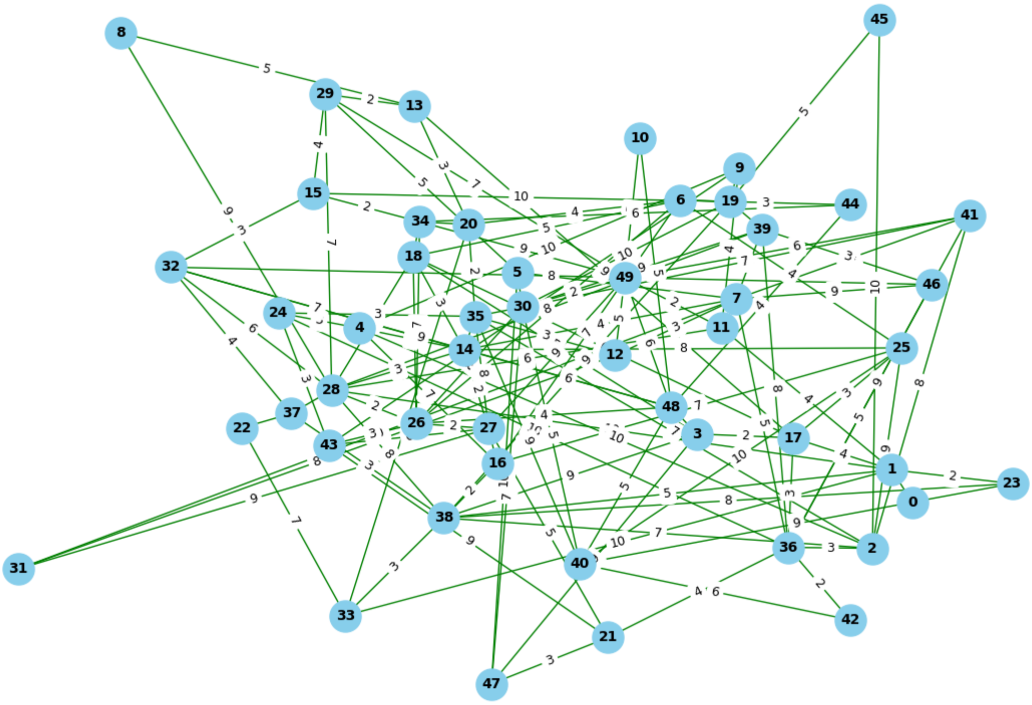
Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Kruskal's MST	10	Sparse	0.00027	2568	32
Kruskal's MST	10	Dense	0.000155	2568	28
Kruskal's MST	15	Sparse	0.000142	3256	65
Kruskal's MST	15	Dense	0.000234	3256	39
Kruskal's MST	50	Sparse	0.000296	4608	167
Kruskal's MST	50	Dense	0.000783	7808	122
Kruskal's MST	75	Sparse	0.000633	7336	232
Kruskal's MST	75	Dense	0.000582	8032	204
Kruskal's MST	100	Sparse	0.000402	7456	437
Kruskal's MST	100	Dense	0.001489	14760	279
Kruskal's MST	250	Sparse	0.000974	26256	989
Kruskal's MST	250	Dense	0.00523	29104	731
Kruskal's MST	500	Sparse	0.002486	51176	2086
Kruskal's MST	500	Dense	0.006417	206092	1499
Kruskal's MST (Optimized)	10	Sparse	0.000185	2536	32
Kruskal's MST (Optimized)	10	Dense	0.000203	2536	28
Kruskal's MST (Optimized)	15	Sparse	0.000172	3504	65
Kruskal's MST (Optimized)	15	Dense	0.000206	3504	39
Kruskal's MST (Optimized)	50	Sparse	0.00033	5392	167
Kruskal's MST (Optimized)	50	Dense	0.000508	9688	122
Kruskal's MST (Optimized)	75	Sparse	0.000254	9216	232
Kruskal's MST (Optimized)	75	Dense	0.000723	9912	204
Kruskal's MST (Optimized)	100	Sparse	0.000571	153423	437
Kruskal's MST (Optimized)	100	Dense	0.001746	19064	279
Kruskal's MST (Optimized)	250	Sparse	0.001444	35176	989
Kruskal's MST (Optimized)	250	Dense	0.006115	38024	731
Kruskal's MST (Optimized)	500	Sparse	0.003379	69304	2086
Kruskal's MST (Optimized)	500	Dense	0.006373	195312	1499

Visual Graph Examples

Figure 2.1 provides visual samples of both sparse and dense graphs used in the MST experiments, each consisting of 50 nodes.



(a) Sparse Graph with 50 Nodes



(b) Dense Graph with 50 Nodes

Figure 2.1: Examples of Sparse and Dense Graphs Used in MST Evaluation

Graphical Analysis of MST Performance

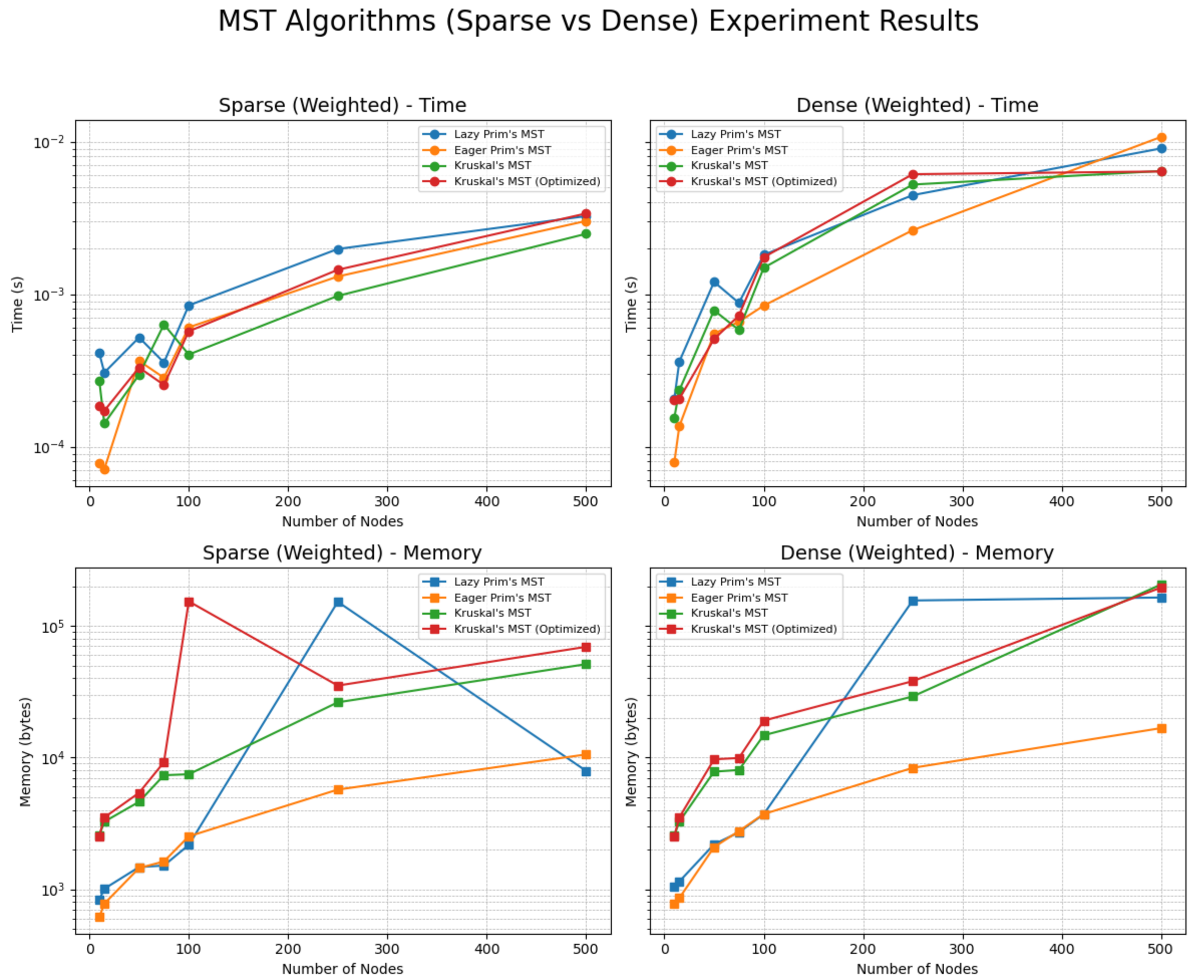


Figure 2.2: Performance of MST Algorithms: Time and Memory on Sparse vs Dense Graphs

Graph Density Impact on MST Performance: Graph density plays a critical role in the efficiency of MST algorithms. Sparse graphs allow faster execution and lower memory usage due to fewer edges, favoring heap-based implementations like Eager Prim's. Dense graphs increase computational demands, especially for Kruskal's algorithm, where edge sorting dominates. Optimized Kruskal consistently delivers reliable results across all densities.

Key Observations

- **Eager Prim's MST** was the most memory-efficient algorithm across all graph types and sizes.
- **Lazy Prim's MST** offered good speed on sparse graphs but was slower and more memory-intensive on dense graphs.
- **Kruskal's MST (Optimized)** balanced performance well and handled large, dense graphs efficiently.

- Memory usage scales faster in dense graphs, particularly for Kruskal's variants.

2.3.2 Connected Graphs – MST Algorithms

This experiment focuses on evaluating the performance of Minimum Spanning Tree (MST) algorithms — Lazy Prim's, Eager Prim's, Kruskal's, and Optimized Kruskal's — on connected graphs with weighted edges. All test cases ensured full connectivity across node sets ranging from 10 to 500 nodes. The key metrics assessed were execution time and memory usage under guaranteed connectivity.

Empirical Results Table

Table 2.3: Execution Time and Memory Usage – Connected Graphs (MST)

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Lazy Prim's MST	10	Connected	8.1e-05	912	43
Lazy Prim's MST	15	Connected	9.6e-05	1016	60
Lazy Prim's MST	50	Connected	0.000431	1712	208
Lazy Prim's MST	75	Connected	0.001321	2296	290
Lazy Prim's MST	100	Connected	0.001689	150743	430
Lazy Prim's MST	250	Connected	0.005915	153176	950
Lazy Prim's MST	500	Connected	0.010736	158644	2209
Eager Prim's MST	10	Connected	0.000139	712	43
Eager Prim's MST	15	Connected	0.000178	824	60
Eager Prim's MST	50	Connected	0.000481	1832	208
Eager Prim's MST	75	Connected	0.0007	2456	290
Eager Prim's MST	100	Connected	0.000941	3144	430
Eager Prim's MST	250	Connected	0.002736	7272	950
Eager Prim's MST	500	Connected	0.006981	162282	2209
Kruskal's MST	10	Connected	0.0001	2568	43
Kruskal's MST	15	Connected	0.000139	3256	60
Kruskal's MST	50	Connected	0.000423	7336	208
Kruskal's MST	75	Connected	0.000602	7616	290
Kruskal's MST	100	Connected	0.000943	13864	430
Kruskal's MST	250	Connected	0.001713	27312	950
Kruskal's MST	500	Connected	0.003876	53352	2209
Kruskal's MST (Optimized)	10	Connected	8.3e-05	2536	43
Kruskal's MST (Optimized)	15	Connected	0.000211	3504	60
Kruskal's MST (Optimized)	50	Connected	0.000451	9216	208
Kruskal's MST (Optimized)	75	Connected	0.00054	9496	290
Kruskal's MST (Optimized)	100	Connected	0.000678	18168	430
Kruskal's MST (Optimized)	250	Connected	0.001803	36232	950
Kruskal's MST (Optimized)	500	Connected	0.006293	219741	2209

Visual Graph Example

The following figure displays an example of a connected graph with 75 nodes, used during the performance evaluation.

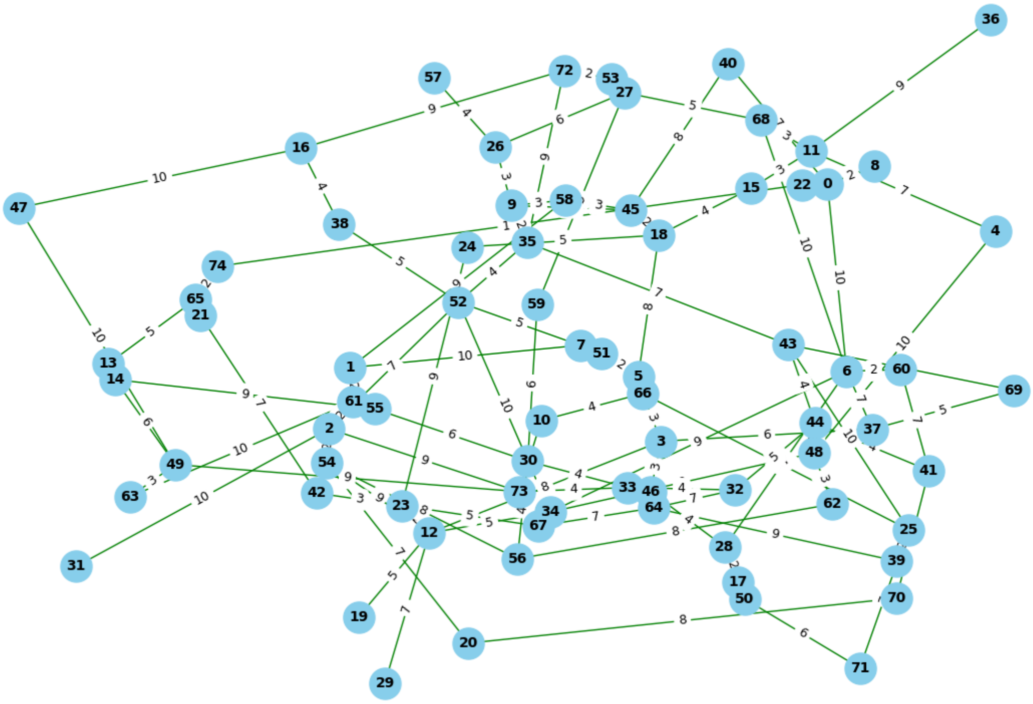


Figure 2.3: Connected Weighted Graph with 75 Nodes

Graphical Analysis of Algorithm Performance

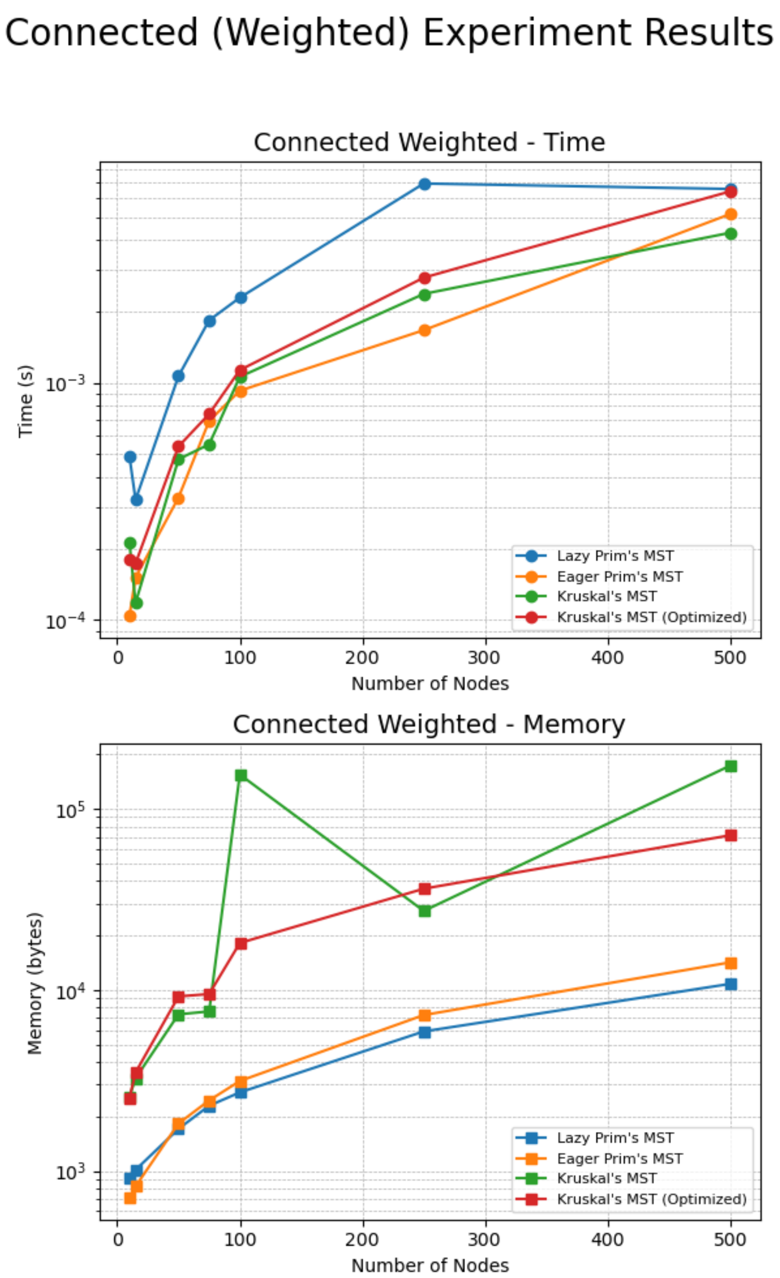


Figure 2.4: MST Algorithms: Time and Memory Analysis on Connected Graphs

Performance Patterns in Connected Graphs: All MST algorithms maintained reliable performance across connected graphs, but the variation in edge density and structural complexity influenced memory and time efficiency. Dense connectivity increased overhead for Lazy Prim's, while Eager Prim's consistently used the least memory. Optimized Kruskal's algorithm scaled well across all node counts.

Key Observations

- **Lazy Prim's MST** had higher memory usage at larger node scales due to internal heap overhead.

- **Eager Prim's MST** showed the most stable memory profile with minimal spikes across increasing sizes.
- **Kruskal's MST (Optimized)** maintained low runtime with acceptable memory, especially for 250 and 500 nodes.
- Memory usage became a limiting factor in large connected graphs, particularly in Lazy Prim and Kruskal variants.

2.3.3 Cyclic vs Acyclic Graphs – MST Algorithms

This experiment investigates the impact of graph structure — specifically whether a graph is cyclic or acyclic — on the performance of MST algorithms. The evaluation includes Lazy Prim's, Eager Prim's, Kruskal's, and Optimized Kruskal's algorithms. Each algorithm was tested on graphs of varying sizes (10 to 500 nodes), comparing execution time and memory usage between cyclic and acyclic configurations.

Empirical Results Table

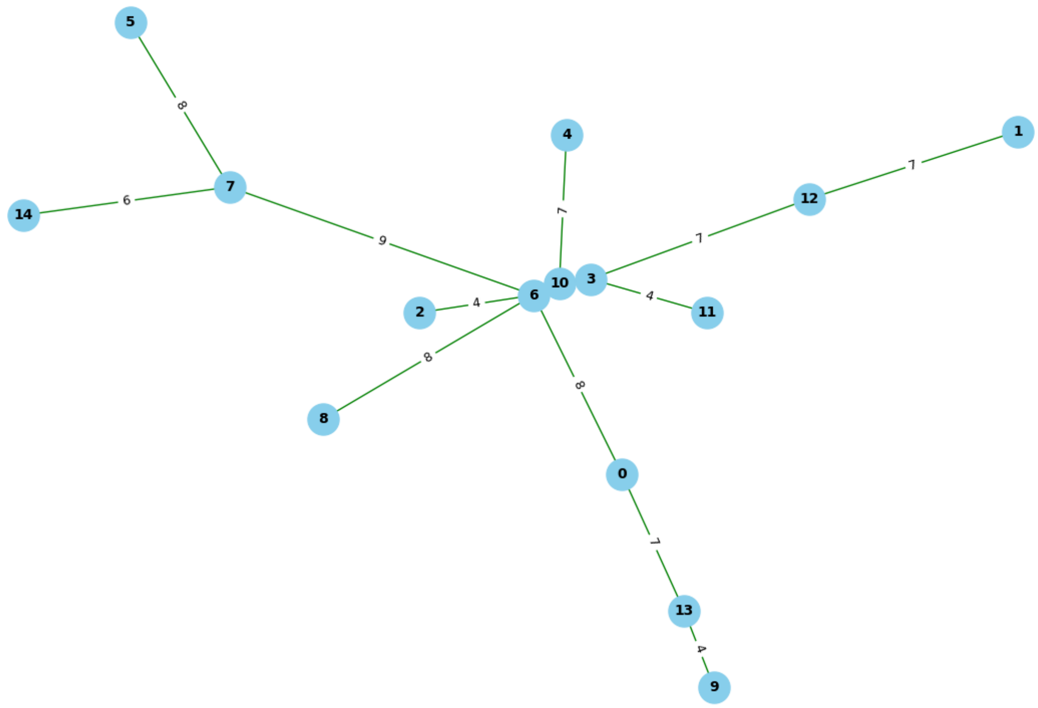
Table 2.4: Execution Time and Memory Usage – Cyclic vs Acyclic Graphs (MST)

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Lazy Prim's MST	10	Acyclic	5.1e-05	848	61
Lazy Prim's MST	10	Cyclic	0.000124	880	51
Lazy Prim's MST	15	Acyclic	0.000135	952	81
Lazy Prim's MST	15	Cyclic	0.000198	1080	60
Lazy Prim's MST	50	Acyclic	0.000328	1520	267
Lazy Prim's MST	50	Cyclic	0.001044	151639	82
Lazy Prim's MST	75	Acyclic	0.000524	1912	385
Lazy Prim's MST	75	Cyclic	0.001633	5496	112
Lazy Prim's MST	100	Acyclic	0.00122	2368	566
Lazy Prim's MST	100	Cyclic	0.004509	10080	116
Lazy Prim's MST	250	Acyclic	0.002639	4880	1408
Lazy Prim's MST	250	Cyclic	0.019354	460145	251
Lazy Prim's MST	500	Acyclic	0.003381	8988	2922
Lazy Prim's MST	500	Cyclic	0.07665	1645764	499
Eager Prim's MST	10	Acyclic	4.5e-05	648	61
Eager Prim's MST	10	Cyclic	7.3e-05	648	51
Eager Prim's MST	15	Acyclic	0.0001	792	81
Eager Prim's MST	15	Cyclic	0.000146	856	60
Eager Prim's MST	50	Acyclic	0.000282	1640	267
Eager Prim's MST	50	Cyclic	0.000837	2312	82

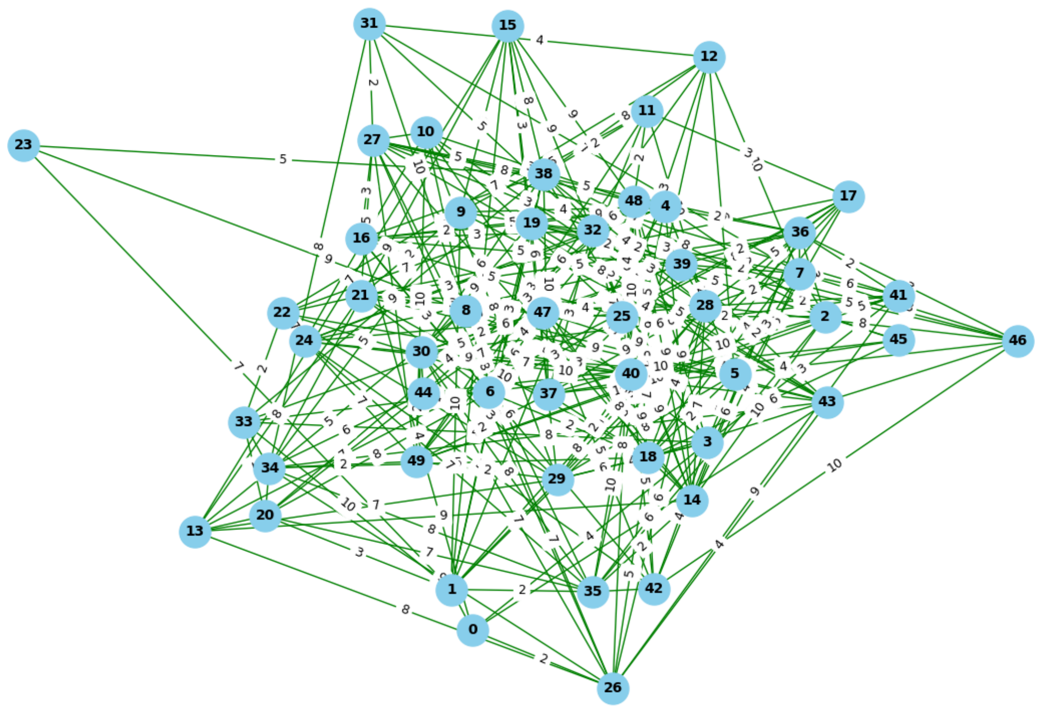
Table 2.5: Execution Time and Memory Usage – Cyclic vs Acyclic Graphs (MST)

Algorithm	**Nodes**	**Graph Type**	**Time (s)**	**Memory (B)**	**Result**
Eager Prim's MST	75	Acyclic	0.00032	2232	385
Eager Prim's MST	75	Cyclic	0.001107	3192	112
Eager Prim's MST	100	Acyclic	0.000642	2888	566
Eager Prim's MST	100	Cyclic	0.002012	4680	116
Eager Prim's MST	250	Acyclic	0.001298	154608	1408
Eager Prim's MST	250	Cyclic	0.013534	11880	251
Eager Prim's MST	500	Acyclic	0.003272	12708	2922
Eager Prim's MST	500	Cyclic	0.033659	171516	499
Kruskal's MST	10	Acyclic	6.4e-05	2568	61
Kruskal's MST	10	Cyclic	5.4e-05	2568	51
Kruskal's MST	15	Acyclic	6.3e-05	3256	81
Kruskal's MST	15	Cyclic	0.000187	3256	60
Kruskal's MST	50	Acyclic	0.000343	7336	267
Kruskal's MST	50	Cyclic	0.001216	8800	82
Kruskal's MST	75	Acyclic	0.000262	7336	385
Kruskal's MST	75	Cyclic	0.002362	15336	112
Kruskal's MST	100	Acyclic	0.000568	13464	566
Kruskal's MST	100	Cyclic	0.002974	28984	116
Kruskal's MST	250	Acyclic	0.001179	26320	1408
Kruskal's MST	250	Cyclic	0.029508	566744	251
Kruskal's MST	500	Acyclic	0.002022	51352	2922
Kruskal's MST	500	Cyclic	0.208314	2058176	499
Kruskal's MST (Optimized)	10	Acyclic	7.1e-05	2536	61
Kruskal's MST (Optimized)	10	Cyclic	0.000165	2536	51
Kruskal's MST (Optimized)	15	Acyclic	0.000171	3504	81
Kruskal's MST (Optimized)	15	Cyclic	0.000237	3504	60
Kruskal's MST (Optimized)	50	Acyclic	0.000613	9216	267
Kruskal's MST (Optimized)	50	Cyclic	0.001405	10680	82
Kruskal's MST (Optimized)	75	Acyclic	0.000561	9216	385
Kruskal's MST (Optimized)	75	Cyclic	0.002202	17216	112
Kruskal's MST (Optimized)	100	Acyclic	0.00083	17768	566
Kruskal's MST (Optimized)	100	Cyclic	0.01043	33288	116
Kruskal's MST (Optimized)	250	Acyclic	0.00196	35896	1408
Kruskal's MST (Optimized)	250	Cyclic	0.046033	576478	251
Kruskal's MST (Optimized)	500	Acyclic	0.003118	69480	2922
Kruskal's MST (Optimized)	500	Cyclic	0.208448	2076304	499

Visual Graph Examples



(a) Acyclic Weighted Graph with 15 Nodes



(b) Cyclic Weighted Graph with 50 Nodes

Figure 2.5: Examples of Acyclic and Cyclic Weighted Graphs

Graphical Analysis of Algorithm Performance

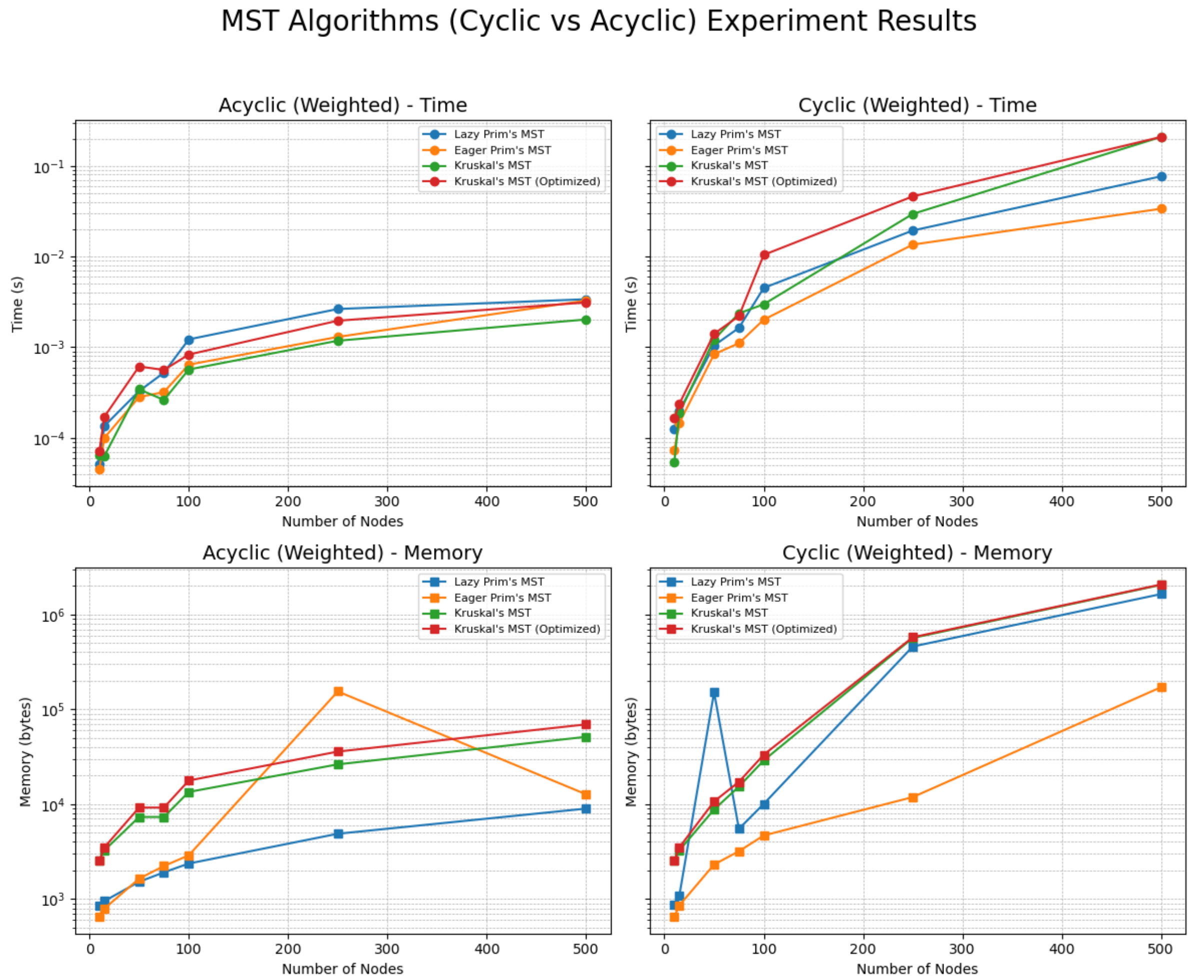


Figure 2.6: Performance of MST Algorithms: Time and Memory – Cyclic vs Acyclic Graphs

Structural Impact (Cyclic vs Acyclic): Graph structure significantly influences both performance and memory usage. Cyclic graphs typically contain a greater number of edges, which increases computational complexity and memory requirements. Acyclic graphs (trees or forests) minimize redundant connections, resulting in faster runtimes and lower memory consumption, particularly for Prim's algorithms.

Key Observations

- **Lazy Prim's MST** exhibited a steep rise in memory for cyclic graphs beyond 100 nodes, due to increasing edge density.
- **Eager Prim's MST** was most memory-efficient on both graph types and scaled better in cyclic scenarios.
- **Kruskal's MST (Optimized)** delivered consistent time performance but required significantly more memory for cyclic graphs.

- Cyclic graphs caused up to 10× increases in memory consumption for Kruskal's variants at larger node counts.

2.3.4 Tree Graphs – MST Algorithms

This experiment evaluates the performance of four MST algorithms — Lazy Prim's, Eager Prim's, Kruskal's, and Optimized Kruskal's — on tree-structured weighted graphs. Since a tree is an acyclic connected graph with exactly $n - 1$ edges, the MST of such graphs should match the input structure. The goal was to observe how each algorithm behaves in such minimal-edge environments, especially in terms of time and memory usage.

Empirical Results Table

Table 2.6: Execution Time and Memory Usage – Tree Graphs (MST)

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Lazy Prim's MST	10	Tree	5.6e-05	848	46
Lazy Prim's MST	15	Tree	9.7e-05	920	76
Lazy Prim's MST	50	Tree	0.000337	1520	258
Lazy Prim's MST	75	Tree	0.000489	1944	426
Lazy Prim's MST	100	Tree	0.000624	2400	498
Lazy Prim's MST	250	Tree	0.000921	4976	1379
Lazy Prim's MST	500	Tree	0.002893	157169	2718
Eager Prim's MST	10	Tree	6.4e-05	648	46
Eager Prim's MST	15	Tree	0.000101	760	76
Eager Prim's MST	50	Tree	0.000296	1640	258
Eager Prim's MST	75	Tree	0.000339	2264	426
Eager Prim's MST	100	Tree	0.000898	2888	498
Eager Prim's MST	250	Tree	0.002644	6632	1379
Eager Prim's MST	500	Tree	0.005036	160506	2718
Kruskal's MST	10	Tree	7.5e-05	2568	46
Kruskal's MST	15	Tree	8.4e-05	3256	76
Kruskal's MST	50	Tree	0.0003	7336	258
Kruskal's MST	75	Tree	0.000535	7336	426
Kruskal's MST	100	Tree	0.000671	13464	498
Kruskal's MST	250	Tree	0.001632	26320	1379
Kruskal's MST	500	Tree	0.004198	51352	2718
Kruskal's MST (Optimized)	10	Tree	0.000137	2536	46
Kruskal's MST (Optimized)	15	Tree	0.00017	3504	76
Kruskal's MST (Optimized)	50	Tree	0.00028	9216	258
Kruskal's MST (Optimized)	75	Tree	0.000552	9216	426
Kruskal's MST (Optimized)	100	Tree	0.000633	17768	498
Kruskal's MST (Optimized)	250	Tree	0.001626	35240	1379
Kruskal's MST (Optimized)	500	Tree	0.003562	69480	2718

Visual Graph Example

The graph below represents a tree structure with 50 nodes and 49 edges used in one of the benchmark scenarios:

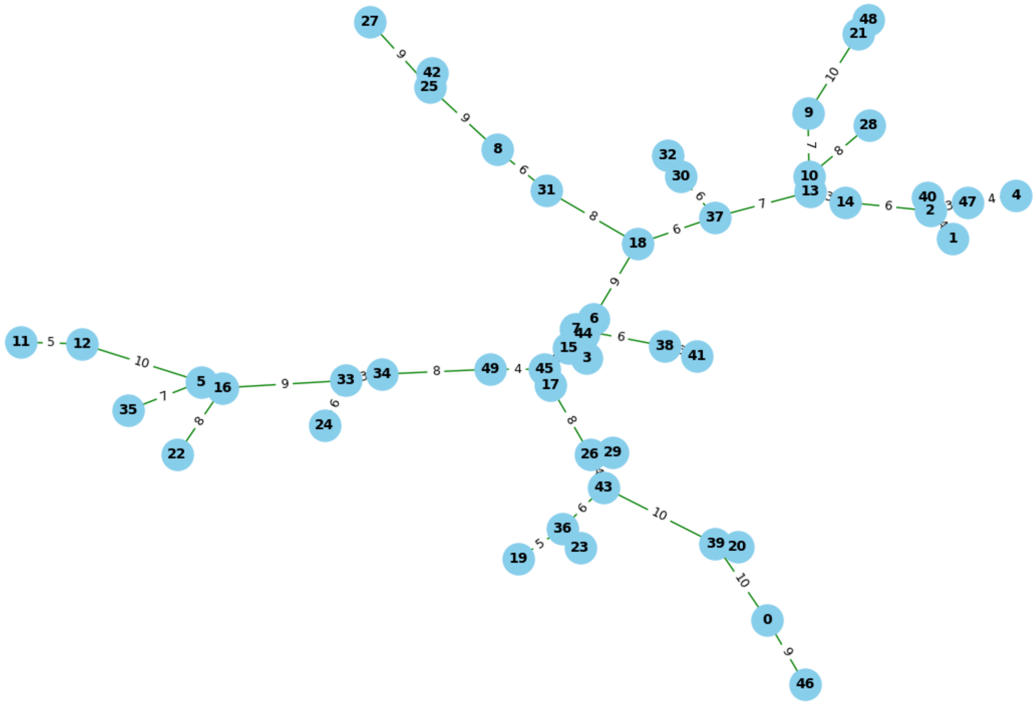


Figure 2.7: Tree Graph with 50 Nodes

Graphical Analysis of Algorithm Performance

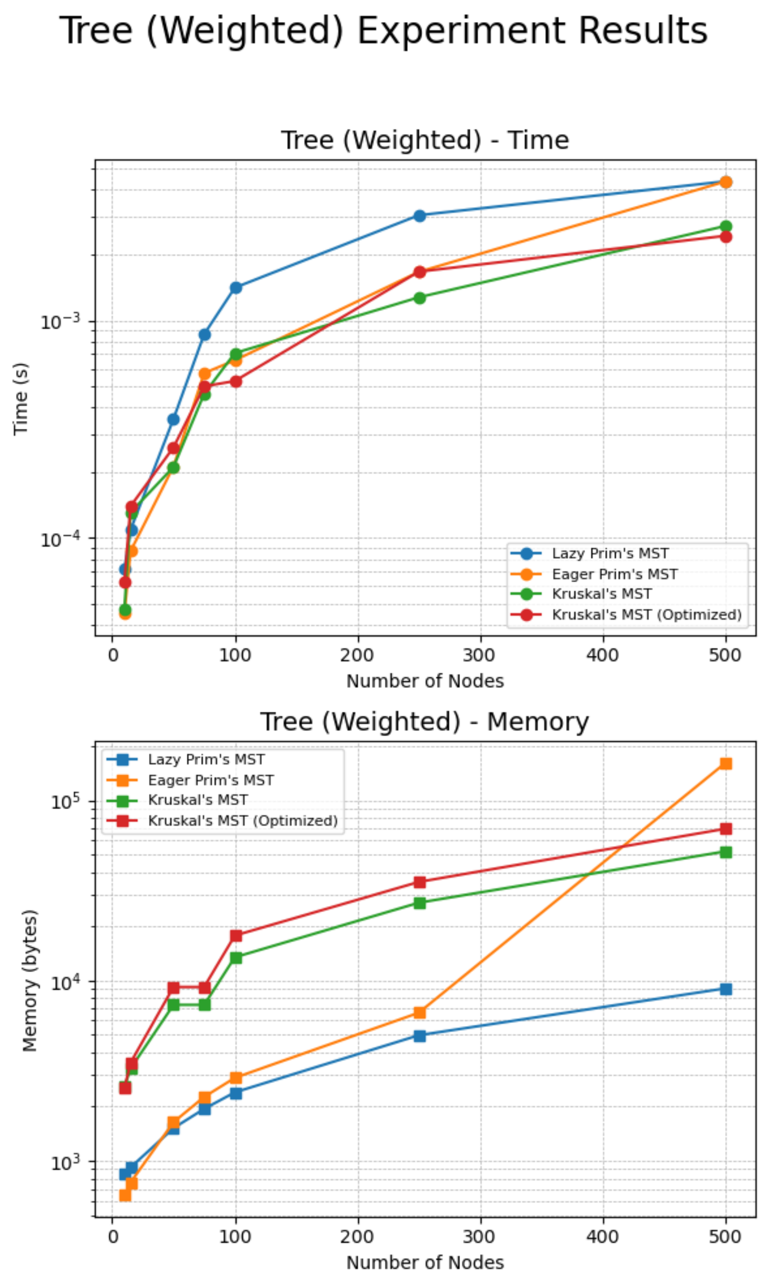


Figure 2.8: Time and Memory Analysis of MST Algorithms on Tree Graphs

Performance on Tree Graphs: Tree graphs inherently simplify MST computation due to their structure. All algorithms detected the pre-existing spanning tree effectively. The memory footprint was noticeably lower for Eager Prim's, while Kruskal's variants required more space, particularly in large node counts. Execution times were minimal for all approaches.

Key Observations

- All algorithms performed efficiently due to the minimal edge count (exactly $n-1$).

- **Eager Prim's MST** showed the lowest memory usage across all tree sizes.
- **Kruskal's MST (Optimized)** performed well, but memory grew with node count due to union-find structures.
- Lazy Prim's MST was fast and stable, though slightly less memory-efficient than the eager variant.

Chapter 3

Conclusion

The empirical analysis of Minimum Spanning Tree (MST) algorithms conducted in this report was executed on an **Asus ZenBook 14**, equipped with an **Intel Core i7 8th Gen processor (1.8 GHz, 4 cores, 8 threads)**, **16GB RAM**, and **512GB SSD**. All implementations were developed in **Python** using **Visual Studio Code** and executed within a **Jupyter Notebook** environment. Performance metrics, including execution time and memory usage, were measured with the `timeit` and `tracemalloc` modules, ensuring accurate benchmarking across a variety of graph structures.

The study focused on two core MST algorithms—**Prim's** and **Kruskal's**—each tested in both standard and optimized forms. Graph types ranged from sparse to dense, cyclic to acyclic, connected to tree-structured graphs. This allowed a well-rounded understanding of each algorithm's strengths and trade-offs under diverse conditions.

Key Insights

- **Eager Prim's Algorithm** consistently achieved the best performance in terms of memory efficiency and execution time, particularly on sparse and tree graphs. Its use of a priority queue and selective edge consideration reduced overhead.

- **Lazy Prim's Algorithm** was simpler to implement but less efficient on dense or cyclic graphs due to redundant edge insertions and checks. It showed good speed in smaller sparse configurations but consumed more memory overall.

- **Kruskal's Algorithm** performed well on sparse graphs due to its edge-centric nature. However, performance degraded with increased density as edge sorting became more computationally expensive.

- **Optimized Kruskal's Algorithm**, using path compression and union by rank in its disjoint-set structure, significantly improved runtime and scalability. It showed consistent behavior across all graph types, handling large and dense graphs more effectively than the basic variant.

- **Graph Topology Effects**:

  - *Sparse vs Dense:* All algorithms performed better on sparse graphs. Dense graphs increased memory usage and execution time, especially for Kruskal-based methods.

- *Connected vs Disconnected:* All tests used connected graphs to ensure MST validity; however, edge count and structure still influenced algorithm efficiency.
- *Cyclic vs Acyclic:* Cyclic graphs introduced more computational overhead due to additional edges, particularly in Kruskal's union-find logic.
- *Trees:* Tree graphs were the most efficient configuration for all algorithms, as they are minimal edge cases. Eager Prim's was again the most memory-efficient in this scenario.

Complexity Overview

The empirical findings aligned well with theoretical expectations. Prim's algorithm, implemented with a heap-based priority queue, achieved $O(E \log V)$ time complexity, making it favorable for dense graphs in its eager variant. Kruskal's algorithm, dependent on sorting edges, maintained its $O(E \log E)$ complexity, with its optimized version benefiting from near-constant-time union-find operations. These complexities were reflected in measured performance across test cases.

Final Remarks

This laboratory work reaffirmed the importance of choosing MST algorithms based on graph characteristics. Eager Prim's algorithm emerged as the most resource-efficient in the majority of scenarios, while Kruskal's optimized version proved highly scalable and robust for large, dense graphs. Through empirical analysis, theoretical knowledge was reinforced with practical insights, highlighting how data structures and implementation details directly impact algorithmic performance.

Source Code

GitHub Link: `https://github.com/PatriciaMoraru/AA_Laboratory_Works`