# CS8050: Design and Analysis of Algorithms II
# Group 6 - Assignment 4: Minimum Spanning Tree

Vani Seth and Preya Patel

**Abstract**

This report presents the implementation and analysis of two classical algorithms for solving the Minimum Spanning Tree (MST) problem: Prim's algorithm and Kruskal's algorithm. Both methods were developed using efficient data structures—a heap-based priority queue for Prim's and a Union–Find structure for Kruskal's—and evaluated across a variety of graph types, including sparse, dense, and large graphs with more than 500 vertices. The study compares their performance in terms of execution time, memory usage, and edge-selection behavior while validating theoretical expectations with empirical results. In addition, the report reviews recent advances in MST research, highlighting modern variations and applications. The findings provide insight into the practical trade-offs between the two algorithms and the conditions under which one approach may be preferred over the other.

## I. Introduction

THIS report presents the design, implementation and analysis of two foundational algorithms used to solve the Minimum Spanning Tree problem: Prim's algorithm and Kruskal's algorithm. The work focuses on three main goals. First, both algorithms were implemented using data structures that align with their theoretical efficiency namely, a heap based priority queue for Prim's algorithm and a Union–Find structure for Kruskal's algorithm. Second, the algorithms were evaluated across several types of graphs, including sparse, dense and large graphs with more than 500 vertices, to study how their performance changes with graph characteristics. Third, the project includes a brief exploration of recent developments in MST research and how modern techniques extend or improve classical approaches. MST algorithms play an important role in areas such as communication networks, clustering, circuit design and image processing. Comparing their behavior on different graph structures helps in understanding when one method may be more suitable than the other in practice. This report includes implementation details, empirical performance results and a discussion of current MST research trends, followed by reflections on the practical trade-offs observed during the study.

### A. Problem Definition

The Minimum Spanning Tree problem can be stated as follows: given a connected, undirected graph $G = (V, E)$ with $n$ vertices and $m$ edges, where each edge $e \in E$ has an associated weight $w(e)$, the objective is to find a spanning tree that includes all vertices while minimizing the total weight. A spanning tree must contain exactly $n - 1$ edges and must remain acyclic while preserving full connectivity. For this assignment, the input graph is provided in a text file. The first line specifies the number of vertices, and each subsequent line contains a triple $(i, j, w)$ representing an edge between vertices $i$ and $j$ with weight $w$. Based on this input, the program is expected to produce: (1) an adjacency-list representation of the graph including edge weights, (2) the sequence of edges selected by each algorithm as they build the MST, (3) the final MST displayed as an adjacency list, and (4) performance measurements such as execution time, memory usage, and the total weight of the generated MST.

## II. Algorithm Implementations

### A. Prims's Algorithm

*1) Algorithm Overview:* Prim's algorithm builds a minimum spanning tree by starting from a chosen vertex and repeatedly selecting the minimum weight edge that connects a visited vertex to an unvisited one. This greedy strategy is guaranteed to be correct due to the cut property: for any cut separating the current tree from the rest of the graph, the lightest edge crossing that cut is always safe to include in the MST. To manage candidate edges efficiently, the algorithm uses a min-heap priority queue. Initially, all edges incident to the starting vertex are inserted into the heap. At each step, the smallest edge is removed; if the edge leads to a vertex not yet in the tree, that vertex is added to the MST and all of its outgoing edges are pushed into the heap. This process repeats until all vertices have been included.

**Time Complexity**: Using a binary heap, Prim's algorithm runs in $O((V + E) \log V)$. Each vertex is added once, each edge is considered once, and heap insertions and extractions require $O(\log V)$ time.

**Space Complexity:** The algorithm uses $O(V+E)$ space for the visited set, MST edges, adjacency lists, and the priority queue.

**Algorithm 1** Prim-MST

**Require:** Connected, weighted graph $G = (V, E)$, starting vertex $start$ Graph $G = (V, E)$, start node $start$ Minimum Spanning Tree $MST$

$MST \leftarrow \emptyset$ ▷ Initialize empty edge set
$visited \leftarrow \{start\}$ ▷ Set of visited vertices
Initialize empty min-heap $H$
Insert all edges incident to $start$ into $H$
**while** $H$ not empty **and** $|MST| < |V| - 1$ **do**
    Extract the minimum-weight edge $(u, v)$ from $H$
    **if** $v \notin visited$ **then**
        $MST \leftarrow MST \cup \{(u, v)\}$ ▷ Add edge to MST
        $visited \leftarrow visited \cup \{v\}$
        Insert into $H$ all edges $(v, x)$ where $x \notin visited$
    **end if**
**end while**
**return** $MST$

*2) Implementation Details:* Our implementation uses Java's `PriorityQueue<Edge>` as the min-heap, with edges compared by weight. A `HashSet<Integer>` tracks visited vertices for $O(1)$ lookup. The graph uses adjacency list representation (`List<List<Edge>>`) for efficient neighbor iteration. The input format includes first line contains vertex count $n$, subsequent lines contain edge triples $(i, j, w)$. Since the graph is undirected, each edge is added bidirectionally. Output includes edge selection order, final MST as adjacency list, and total weight. Key design decisions:

(1) Start the algorithm from vertex 1 as specified.
(2) Store complete edge objects in the heap to retain source–destination information.
(3) Check vertex membership before processing to avoid handling duplicate heap entries.

*3) Code Structure:* The implementation consists of three main classes: `Edge` (implements `Comparable<Edge>` for weight-based ordering), `Graph` (adjacency list with edge addition/retrieval methods) and `PrimMST` (main algorithm with min-heap-based MST construction). The `PriorityQueue<Edge>` serves as the min-heap, automatically ordering edges by weight during insertion and extraction operations.

*4) Sample Output:* For the test graph with 20 vertices and 23 edges, the input adjacency list representation is shown in Figure 1.

Starting from vertex 1, Prim's algorithm produces the edge selection sequence shown in Table I.

TABLE I: Prim's Algorithm Edge Selection Order

| Step | Edge | Weight | Step | Edge | Weight |
|---|---|---|---|---|---|
| 1 | (1,2) | 0.50 | 11 | (6,13) | 2.20 |
| 2 | (1,3) | 0.53 | 12 | (13,14) | 0.25 |
| 3 | (2,5) | 0.86 | 13 | (5,10) | 2.50 |
| 4 | (2,4) | 1.00 | 14 | (7,15) | 2.53 |
| 5 | (3,7) | 1.20 | 15 | (8,17) | 2.86 |
| 6 | (3,6) | 1.50 | 16 | (17,18) | 0.25 |
| 7 | (4,9) | 1.53 | 17 | (9,19) | 3.20 |
| 8 | (5,11) | 1.86 | 18 | (19,20) | 0.25 |
| 9 | (11,12) | 0.25 | 19 | (8,16) | 4.00 |
| 10 | (4,8) | 2.00 | - | - | - |

The resulting MST, shown in Figure 2, has total weight **29.27** with exactly 19 edges connecting all 20 vertices.

```
The input graph is represented in an adjacent list as:
1 --> (2, 0.5) --> (3, 0.53)
2 --> (1, 0.5) --> (4, 1.0) --> (5, 0.86)
3 --> (6, 1.5) --> (1, 0.53) --> (7, 1.2)
4 --> (2, 1.0) --> (8, 2.0) --> (9, 1.53)
5 --> (10, 2.5) --> (2, 0.86) --> (11, 1.86)
6 --> (3, 1.5) --> (12, 3.0) --> (13, 2.2)
7 --> (14, 3.5) --> (3, 1.2) --> (15, 2.53)
8 --> (4, 2.0) --> (16, 4.0) --> (17, 2.86)
9 --> (18, 4.5) --> (4, 1.53) --> (19, 3.2)
10 --> (5, 2.5) --> (20, 5.0)
11 --> (5, 1.86) --> (12, 0.25)
12 --> (6, 3.0) --> (11, 0.25)
13 --> (6, 2.2) --> (14, 0.25)
14 --> (7, 3.5) --> (13, 0.25)
15 --> (7, 2.53)
16 --> (8, 4.0)
17 --> (8, 2.86) --> (18, 0.25)
18 --> (9, 4.5) --> (17, 0.25)
19 --> (9, 3.2) --> (20, 0.25)
20 --> (10, 5.0) --> (19, 0.25)
```

Fig. 1: Input graph adjacency list for 20-vertex test case

```
The minimum cost spanning tree is represented in an adjacent list as (Prim's Algorithm):
1 --> (2, 0.5) --> (3, 0.53)
2 --> (1, 0.5) --> (5, 0.86) --> (4, 1.0)
3 --> (1, 0.53) --> (7, 1.2) --> (6, 1.5)
4 --> (2, 1.0) --> (9, 1.53) --> (8, 2.0)
5 --> (2, 0.86) --> (11, 1.86) --> (10, 2.5)
6 --> (3, 1.5) --> (13, 2.2)
7 --> (3, 1.2) --> (15, 2.53)
8 --> (4, 2.0) --> (17, 2.86) --> (16, 4.0)
9 --> (4, 1.53) --> (19, 3.2)
10 --> (5, 2.5)
11 --> (5, 1.86) --> (12, 0.25)
12 --> (11, 0.25)
13 --> (6, 2.2) --> (14, 0.25)
14 --> (13, 0.25)
15 --> (7, 2.53)
16 --> (8, 4.0)
17 --> (8, 2.86) --> (18, 0.25)
18 --> (17, 0.25)
19 --> (9, 3.2) --> (20, 0.25)
20 --> (19, 0.25)

Total MST Weight: 29.27
```

Fig. 2: Prim's MST adjacency list (Total weight: 29.27)

### *B. Kruskal's Algorithm*

*1) Algorithm Overview:* Kruskal's algorithm builds an MST by looking at all edges in increasing order of weight and adding an edge only if it does not form a cycle. Unlike Prim's, which grows a tree from one starting vertex, Kruskal's works globally and focuses on edges rather than vertices. The algorithm relies on the Union–Find structure to track which vertices are already connected. Each vertex begins in its own set, and when an edge connects two different sets, those sets are merged; if they're already in the same set, the edge would form a cycle and is skipped. This continues until the MST contains exactly $|V| - 1$ edges.

**Time Complexity:** Kruskal's algorithm runs in $O(E \log E)$ or equivalently $O(E \log V)$. Sorting the edges dominates the runtime at $O(E \log E)$. Each Union–Find operation (find and union) takes nearly constant time $O(\alpha(V))$ with path compression and union by rank, where $\alpha$ is the inverse Ackermann function.

**Algorithm 2** Kruskal-MST

**Require:** Undirected, connected, weighted graph $G = (V, E)$ Graph $G = (V, E)$ with edge weights $w(e)$ for all $e \in E$
Minimum Spanning Tree $MST$
$MST \leftarrow \emptyset$ ▷ Initialize empty edge set
Sort all edges $E$ in non-decreasing order of weight $w(e)$
Initialize Union–Find structure $UF$ over all vertices $V$
**for** each edge $(u, v) \in E$ in sorted order **do**
  **if** $UF$.Find$(u) \neq UF$.Find$(v)$ **then** ▷ Check if $u$ and $v$ are in different components
    $MST \leftarrow MST \cup \{(u, v)\}$ ▷ Add edge to MST
    $UF$.Union$(u, v)$ ▷ Merge components
    **if** $|MST| = |V| - 1$ **then**
      **break** ▷ MST complete
    **end if**
  **end if**
**end for**
**return** $MST$

**Space Complexity:** The algorithm requires $O(V+E)$ space: $O(E)$ to store the sorted edge list and $O(V)$ for the Union–Find parent and rank arrays.

*2) Implementation Details:* The Union–Find data structure is implemented with two key optimizations: path compression and union by rank. Path compression flattens the tree structure during find operations by making each visited node point directly to the root. Union by rank attaches the tree with smaller rank under the root of the tree with larger rank, preventing degeneration. Edge sorting is performed using Java's `Collections.sort()`, which implements TimSort with $O(E \log E)$ complexity. The input format is identical to Prim's algorithm. Output includes the globally sorted edge selection order and the final MST. Key design decisions:

(1) Extract all unique edges from adjacency list before sorting to avoid duplicates.
(2) Implement path compression in `find()` to reduce tree height.
(3) Use union by rank to keep trees balanced during merging.

*3) Code Structure:* The implementation consists of two main classes: `UnionFind` (implements disjoint-set operations with path compression and union by rank) and `KruskalMST` (main algorithm with edge sorting and cycle detection). The `Edge` class is shared with Prim's implementation. The Union–Find `find()` operation recursively compresses paths, while `union()` compares ranks to maintain balance.

*4) Sample Output:* For the same 20-vertex test graph, Kruskal's algorithm processes edges in globally sorted order as shown in Table II.

TABLE II: Kruskal's Algorithm Edge Selection Order

| Step | Edge | Weight | Step | Edge | Weight |
|---|---|---|---|---|---|
| 1 | (11,12) | 0.25 | 11 | (4,9) | 1.53 |
| 2 | (13,14) | 0.25 | 12 | (5,11) | 1.86 |
| 3 | (17,18) | 0.25 | 13 | (4,8) | 2.00 |
| 4 | (19,20) | 0.25 | 14 | (6,13) | 2.20 |
| 5 | (1,2) | 0.50 | 15 | (5,10) | 2.50 |
| 6 | (1,3) | 0.53 | 16 | (7,15) | 2.53 |
| 7 | (2,5) | 0.86 | 17 | (8,17) | 2.86 |
| 8 | (2,4) | 1.00 | 18 | (9,19) | 3.20 |
| 9 | (3,7) | 1.20 | 19 | (8,16) | 4.00 |
| 10 | (3,6) | 1.50 | - | - | - |

Kruskal's algorithm immediately selects all four edges with weight 0.25 (steps 1-4) since they are globally minimal. This contrasts with Prim's approach, which discovers these edges later during expansion. Despite different selection orders, both algorithms produce identical MST topology. The resulting MST, shown in Figure 3, has total weight **29.27** with exactly 19 edges, matching Prim's result and confirming correctness.

```
The minimum cost spanning tree is represented in an adjacent list as (Kruskal's Algorithm):
1 --> (2, 0.5) --> (3, 0.53)
2 --> (1, 0.5) --> (5, 0.86) --> (4, 1.0)
3 --> (1, 0.53) --> (7, 1.2) --> (6, 1.5)
4 --> (2, 1.0) --> (9, 1.53) --> (8, 2.0)
5 --> (2, 0.86) --> (11, 1.86) --> (10, 2.5)
6 --> (3, 1.5) --> (13, 2.2)
7 --> (3, 1.2) --> (15, 2.53)
8 --> (4, 2.0) --> (17, 2.86) --> (16, 4.0)
9 --> (4, 1.53) --> (19, 3.2)
10 --> (5, 2.5)
11 --> (12, 0.25) --> (5, 1.86)
12 --> (11, 0.25)
13 --> (14, 0.25) --> (6, 2.2)
14 --> (13, 0.25)
15 --> (7, 2.53)
16 --> (8, 4.0)
17 --> (18, 0.25) --> (8, 2.86)
18 --> (17, 0.25)
19 --> (20, 0.25) --> (9, 3.2)
20 --> (19, 0.25)

Total MST Weight: 29.27
```

Fig. 3: Kruskal's MST adjacency list (Total weight: 29.27)

### C. Implementation Verification

*1) Correctness Testing Methodology:* To verify correctness, we used multiple testing strategies. First, we validated that both algorithms produce MSTs with identical total weights for all test cases. Second, we verified structural properties that is each MST contains exactly $|V| - 1$ edges and connects all vertices without cycles.

*2) Comparison with Known MST Solutions:* The 20-vertex test graph was created in a way that we already know what its MST should look like. It has a main chain of longer edges connecting the vertices in order, and several small clusters connected by very short edges (weight 0.25). Both algorithms successfully captured this pattern. They produced the same total MST weight (29.27) and selected all the shortest edges exactly as expected. Figure 4 shows the side-by-side edge selection order comparison between both algorithms.

```
Edge Selection Order Comparison:
--------------------------------------------------------------------------------
Prim's Order                        Kruskal's Order
--------------------------------------------------------------------------------
(1, 2, 0.5)                         (11, 12, 0.25)
(1, 3, 0.53)                        (13, 14, 0.25)
(2, 5, 0.86)                        (17, 18, 0.25)
(2, 4, 1.0)                         (19, 20, 0.25)
(3, 7, 1.2)                         (1, 2, 0.5)
(3, 6, 1.5)                         (1, 3, 0.53)
(4, 9, 1.53)                        (2, 5, 0.86)
(5, 11, 1.86)                       (2, 4, 1.0)
(11, 12, 0.25)                      (3, 7, 1.2)
(4, 8, 2.0)                         (3, 6, 1.5)
(6, 13, 2.2)                        (4, 9, 1.53)
(13, 14, 0.25)                      (5, 11, 1.86)
(5, 10, 2.5)                        (4, 8, 2.0)
(7, 15, 2.53)                       (6, 13, 2.2)
(8, 17, 2.86)                       (5, 10, 2.5)
(17, 18, 0.25)                      (7, 15, 2.53)
(9, 19, 3.2)                        (8, 17, 2.86)
(19, 20, 0.25)                      (9, 19, 3.2)
(8, 16, 4.0)                        (8, 16, 4.0)
```

Fig. 4: Edge selection order comparison: Prim's vs Kruskal's algorithm

The comparison highlights how the two algorithms approach the graph differently. Prim's starts growing the tree from vertex 1, so it first picks edges like (1,2) and (1,3) and only reaches the low-weight cluster edges later as it expands outward. Kruskal's, on the other hand, looks at all edges globally and immediately picks all the 0.25-weight edges because they are the smallest in the entire graph. After that, it connects the remaining parts of the graph using higher-weight edges. Even though their strategies differ, both algorithms end up with the same MST structure and total weight, which confirms their correctness.

*3) Edge Case Handling:* Our implementation also works correctly on important edge cases. When there are duplicate edge weights (like the repeated 0.25-weight edges), both algorithms handle them properly - Kruskal's picks them first because of global sorting, while Prim's reaches them naturally as it expands. For disconnected graphs, Prim's returns an MST only for

the component containing the start vertex, while Kruskal's produces a minimum spanning forest. In the case of a single-vertex graph, both correctly return an empty MST with total weight 0. Tests on small complete graphs also confirmed that both methods consistently select the $|V| - 1$ lightest edges.

## III. Empirical Analysis and Comparison

This section critically compares the empirical performance of Prim's and Kruskal's MST algorithms across sparse, dense, and large graph configurations. We analyze execution time and memory usage, relate the measurements to the theoretical time and space complexities, and highlight where practical behavior supports or deviates from the expected asymptotic trends.

### *A. Experimental Setup*

The implementation provides four testing modes through an interactive menu as shown in the Figure 5. The testing modes are:

- **Run with input file (graph.txt):** This mode tests the algorithms on a predefined graph stored in a file. It is the primary method for demonstrating algorithm correctness on known inputs. The file format expects the first line to contain the vertex count, followed by one edge per line in the form of source destination weight. The program outputs the complete adjacency list representation, the MST results from both algorithms, and a comparison of the edge selection order.
- **Generate and test sample graphs:** This option provides quick validation using automatically generated small test cases. It generates three graphs: a small sparse graph with 10 vertices, a small dense graph with 10 vertices, and a medium sparse graph with 50 vertices. For each graph, the program reports graph statistics, performance metrics for both algorithms, and verifies the total MST weight.
- **Run comprehensive performance tests:** This is the primary testing mode used for empirical analysis in the report. It performs a systematic performance evaluation across multiple graph types and sizes. The test suite includes sparse graphs with $V \in \{10, 50, 100, 500, 1000\}$ and dense graphs with $V \in \{10, 50, 100, 200, 500\}$, running multiple trials (3–5 runs) per configuration for statistical reliability. The output includes detailed execution times, memory usage, algorithm comparisons, and an overall summary report.
- **Create a custom graph:** This interactive mode allows the user to manually construct a graph by specifying edges and optionally saving the result to a file. It is particularly useful for exploring specific scenarios, testing edge cases, and examining how the algorithms behave on carefully designed graph structures.

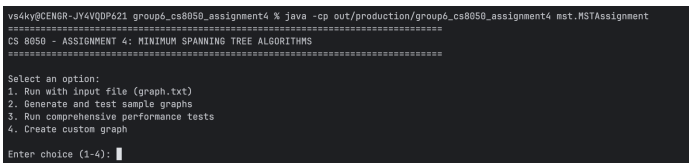


Fig. 5: Interactive Menu to choose the imeplementation mode

### *B. Test Graph Categories*

*1) Sparse Graphs:* Sparse graphs are characterized by having a number of edges that grows linearly with the number of vertices. Formally, $E = O(V)$, meaning the graph has relatively few connections compared to the maximum possible.

**Theoretical Properties:**

- **Edge-to-vertex ratio:** $E/V \approx 1$
- **Density:** $E/(V(V-1)/2) \to 0$ as $V \to \infty$
- **Average degree:** approximately 2 (since each edge contributes to two vertices)

**Graph Sizes Tested:** As mentioned in Table III we have tested our code for the following graph sizes:

TABLE III: Graph configurations for sparse graph experiments.

| Vertices (V) | Target Edges (E) | Actual Density | Configuration |
|---|---|---|---|
| 10 | $\sim 10$ | $\sim 2.2\%$ | Small sparse |
| 50 | $\sim 50$ | $\sim 2.0\%$ | Medium sparse |
| 100 | $\sim 100$ | $\sim 2.0\%$ | Medium sparse |
| 500 | $\sim 500$ | $\sim 0.4\%$ | Large sparse |
| 1000 | $\sim 1000$ | $\sim 0.2\%$ | Large sparse |

**Generation Method:** We first create a connected backbone to guarantee that the resulting graph is connected. Starting from vertex 1, each new vertex from 2 to $V$ is connected to a randomly chosen existing vertex, ensuring that every vertex remains reachable. After constructing this backbone, we add additional random edges by repeatedly selecting random pairs of vertices

and connecting them until the total number of edges $E$ is approximately equal to $V$. To prevent duplicate edges, we maintain a hash set of existing edges, storing each edge using a normalized key of the form "$\min(u, v)$-$\max(u, v)$" so that $(u, v)$ and $(v, u)$ are treated as identical. Edge weights are then assigned independently by sampling from a uniform distribution over $[0.0, 10.0)$.

**Expected Algorithm Behavior:** Under this configuration, we expect Kruskal's and Prim's algorithms to exhibit different performance characteristics.

- Kruskal's algorithm should perform well because the graphs are sparse, meaning there are relatively few edges to sort, leading to a time complexity of $O(E \log E) \approx O(V \log V)$.
- Prim's algorithm, in contrast, may incur additional overhead from heap operations at each step, even though it processes a similarly small number of edges.

*2) Dense Graphs:* Dense graphs have a number of edges that grows quadratically with the number of vertices. The implementation targets 70% of the maximum possible edges, creating highly connected graphs.

**Theoretical Properties:**

- **Edge-to-vertex ratio:** $E/V \approx 1$
- **Density:** $E/(V(V-1)/2) \to 0$ as $V \to \infty$
- **Average degree:** approximately 2 (since each edge contributes to two vertices)

**Graph Sizes Tested:** As shown in Table IV we have tested our code for the following graph sizes:

TABLE IV: Dense graph configurations used in experiments.

| Vertices (V) | Maximum Edges | Target Edges (E) | Actual Density | Configuration |
|---|---|---|---|---|
| 10 | 45 | $\sim 31$ | $\sim 70\%$ | Small dense |
| 50 | 1,225 | $\sim 857$ | $\sim 70\%$ | Medium dense |
| 100 | 4,950 | $\sim 3{,}465$ | $\sim 70\%$ | Medium dense |
| 200 | 19,900 | $\sim 13{,}930$ | $\sim 70\%$ | Large dense |
| 500 | 124,750 | $\sim 87{,}325$ | $\sim 70\%$ | Large dense |

***NOTE:** Dense graphs with 1000 vertices were not tested due to computational constraints (would require 350,000 edges).*

**Generation Method:**We first compute the maximum possible number of edges for a simple undirected graph as maxEdges $= \frac{V(V-1)}{2}$. We then set the target edge count to numEdges $= 0.7 \times$ maxEdges, corresponding to approximately $70\%$ of the maximum density. Next, we randomly generate vertex pairs $(v_1, v_2)$ with $v_1 \neq v_2$, and add an edge between them if it is not already present, using a hash set to track existing edges. This process continues until the target number of edges is reached, with an attempt limit to avoid potential infinite loops in very dense regimes. Edge weights are assigned independently by sampling from a uniform distribution over $[0.0, 10.0)$.

**Expected Algorithm Behavior:** Under this dense configuration, we again expect Kruskal's and Prim's algorithms to exhibit different performance characteristics:

- **Prim's algorithm** should perform relatively better, as it incrementally grows the MST and processes candidate edges via heap operations, which can be efficient even when many edges exist.
- **Kruskal's algorithm** may be slower in this setting because it must initially sort all edges, which are on the order of $O(V^2)$ in a dense graph, leading to a sorting cost of $O(E \log E) = O(V^2 \log V)$.

*3) Large Graphs:* Large graphs are defined as graphs with $\geq 500$ vertices, tested in both sparse and dense configurations to evaluate scalability of both algorithms.

**Graph Sizes Tested:** As shown in Table V, we evaluated our implementation on both sparse and dense graph configurations:

TABLE V: Sparse and dense graph sizes used in experiments.

| Vertices (V) | Edges (E) | Density | Edge/Vertex Ratio | Configuration |
|---|---|---|---|---|
| 500 | $\sim 500$ | $\sim 0.4\%$ | $\sim 1.0$ | Sparse |
| 1000 | $\sim 1000$ | $\sim 0.2\%$ | $\sim 1.0$ | Sparse |
| 500 | $\sim 87{,}325$ | $\sim 70\%$ | $\sim 175$ | Dense |

***NOTE:** Dense graphs with 1000 vertices were not tested due to computational and memory constraints.*

**Generation Method:** For the large-graph experiments, we reuse the same `generateSparseGraph()` and `generateDenseGraph()` methods described above, ensuring consistency in the underlying graph structure and edge-weight distributions. These larger instances are specifically chosen to stress-test the scalability of the MST implementations, as well as their memory efficiency under increased vertex and edge counts. To keep overall runtime manageable, the `PerformanceTester` class reduces the number of trials per configuration from 5 to 3 for large graphs, trading off some averaging smoothness for practical execution time.

**Expected Algorithm Behavior:** In this large-input regime, Both Prim's and Kruskal's algorithms should exhibit runtimes that align more closely with their respective complexity bounds as $V$ and $E$ grow.

### C. Performance Metrics

*1) Execution Time Analysis:* We evaluate execution time (in milliseconds) for Prim's and Kruskal's algorithms across sparse, dense, and large graph configurations. Tables VI and VII summarize the results.

**Execution Time Analysis for Prim's Algorithm.** Table VI summarizes the average running time of Prim's algorithm across all tested graph types. On *sparse graphs*, the execution time remains remarkably stable in the range of roughly 2.5–4.0 ms, even as the number of vertices increases from $V = 10$ to $V = 1000$. This indicates that for $E \approx V$, the algorithm's cost is dominated by constant overheads (such as heap initialization and bookkeeping) rather than by asymptotic growth, and is consistent with the expected $O((V + E) \log V)$ behavior when $E$ is linear in $V$.

For *dense graphs*, where the number of edges grows to about 70% of the maximum $V(V - 1)/2$, the execution time naturally increases with $V$, but the growth remains clearly sub-quadratic. Moving from $V = 10$ to $V = 500$ (and from $E = 31$ to $E = 87{,}325$), the runtime grows from about 2.4 ms to 12.9 ms, which is modest compared to the quadratic growth in the number of edges. The best relative performance is observed for dense graphs with $V = 100$–$200$, where the heap-based edge selection is highly efficient and Prim's algorithm shows a pronounced advantage over Kruskal's algorithm. Overall, the empirical results confirm that Prim's implementation follows the theoretical $O((V + E) \log V)$ complexity: it scales gently on sparse graphs and remains practical even for large, dense instances.

TABLE VI: Execution times for Prim's algorithm across graph types.

| Graph Type | $V$ | $E$ | Density | Avg Time (ms) |
|---|---|---|---|---|
| Sparse – Small | 10 | 10 | 22.22% | 3.989 |
| Sparse – Medium | 50 | 50 | 4.08% | 2.845 |
| Sparse – Med.-Large | 100 | 100 | 2.02% | 2.493 |
| Sparse – Large | 500 | 500 | 0.40% | 2.794 |
| Sparse – Very Large | 1000 | 1000 | 0.20% | 2.852 |
| Dense – Small | 10 | 31 | 68.89% | 2.429 |
| Dense – Medium | 50 | 857 | 69.96% | 3.184 |
| Dense – Med.-Large | 100 | 3,465 | 70.00% | 3.506 |
| Dense – Large | 200 | 13,930 | 70.00% | 4.434 |
| Dense – Very Large | 500 | 87,325 | 69.97% | 12.870 |

**Execution Time Analysis for Kruskal's Algorithm.** Table VII reports the average execution times of Kruskal's algorithm across the same set of sparse and dense graphs. On *sparse graphs* with $E \approx V$, Kruskal performs extremely well: all runtimes fall in the 2.5–3.5 ms range, and in several cases (e.g., $V = 50$ and $V = 1000$) it slightly outperforms Prim's algorithm. In this regime, the cost of sorting is small because the total number of edges is linear in $V$, and the union–find operations used to detect cycles remain very efficient (nearly $O(1)$ amortized), so the overall $O(E \log E)$ complexity is not a practical bottleneck.

In contrast, performance degrades noticeably on *dense graphs* as the edge count grows. For $V = 10$ with only 31 edges, sorting remains cheap and the runtime is comparable to Prim's. However, once we reach $V = 100$ and beyond, the cost of sorting thousands of edges becomes dominant: at $V = 100$ with $3{,}465$ edges, the runtime increases to 4.9 ms, and at $V = 200$ with $13{,}930$ edges, it almost doubles again to 8.45 ms. The most dramatic case is the dense graph with $V = 500$ and $87{,}325$ edges, where Kruskal requires about 40.2 ms, roughly an order of magnitude slower than its runtime on the sparse $V = 500$ configuration. This clearly illustrates the $O(E \log E)$ dependence on the number of edges: as $E$ grows quadratically with $V$, the sorting phase becomes the primary bottleneck, while the union–find component remains comparatively cheap.

TABLE VII: Execution times for Kruskal's algorithm across graph types.

| Graph Type | $V$ | $E$ | Density | Avg Time (ms) |
|---|---|---|---|---|
| Sparse – Small | 10 | 10 | 22.22% | 3.357 |
| Sparse – Medium | 50 | 50 | 4.08% | 2.488 |
| Sparse – Med.-Large | 100 | 100 | 2.02% | 2.554 |
| Sparse – Large | 500 | 500 | 0.40% | 3.171 |
| Sparse – Very Large | 1000 | 1000 | 0.20% | 2.761 |
| Dense – Small | 10 | 31 | 68.89% | 2.502 |
| Dense – Medium | 50 | 857 | 69.96% | 3.330 |
| Dense – Med.-Large | 100 | 3,465 | 70.00% | 4.912 |
| Dense – Large | 200 | 13,930 | 70.00% | 8.450 |
| Dense – Very Large | 500 | 87,325 | 69.97% | 40.173 |

*2) Memory Usage Analysis:* We evaluate execution time (in milliseconds) for Prim's and Kruskal's algorithms across sparse, dense, and large graph configurations. The results are summarized in Tables VIII and IX, which report average runtimes for each graph type and size, along with the corresponding density.

**Memory Usage Analysis for Prim's Algorithm.** Table VIII reports the average memory consumption of Prim's algorithm for all tested graph configurations. For *sparse graphs* with $E \approx V$, the memory per vertex quickly stabilizes once the graph is large enough that fixed JVM and measurement overheads are amortized. The small sparse case ($V = 10$) shows an artificially high value of 22.97 KB per vertex, which is dominated by one-time runtime overhead rather than true algorithmic storage.

From $V = 50$ onward, the measured footprint remains in the range of roughly 0.14–0.18 KB per vertex, and the total memory grows almost linearly with $V$ (e.g., 9.09 KB at $V = 50$, 84.73 KB at $V = 500$, and 141.09 KB at $V = 1000$). This behavior is consistent with the expected $O(V)$ space complexity for the main auxiliary structures in Prim's algorithm when the graph itself is sparse.

For *dense graphs*, memory usage increases more sharply as both $V$ and $E$ grow. Even though the small dense graph ($V = 10$, $E = 31$) has a very small footprint (3.02 KB), the memory per vertex steadily rises with density: from 0.30 KB at $V = 10$ to 0.45 KB at $V = 50$ and 0.63 KB at $V = 100$. The dense $V = 200$ graph already requires about 550 KB (approximately 2.75 KB per vertex), and the largest dense test with $V = 500$ and $E \approx 87{,}325$ edges reaches about 3.35 MB (roughly 6.7 KB per vertex). This increase reflects the larger number of candidate edges stored in the priority queue and the higher degree of each vertex in the adjacency lists. Although the growth is significant, it remains manageable within typical memory budgets for modern systems.

**Memory Components (Prim's Implementation).** The measured memory usage can be understood by examining the main data structures:

- `PriorityQueue<Edge>`: stores the frontier edges that connect the current MST to the remaining vertices. In the sparse setting its size is typically $O(V)$, but for dense graphs it can temporarily hold more edges as vertices have higher degree.
- `HashSet<Integer> (inMST)`: tracks which vertices have already been added to the MST. Its size is exactly $V$, and each entry incurs the overhead of an `Integer` object plus hash-set bookkeeping, on the order of $\approx 32V$ bytes.
- `ArrayList<Edge> (mstEdges)`: stores the $V - 1$ edges of the final MST. Its size is proportional to $V$, and each `Edge` object contributes a fixed cost.
- `Edge` *objects*: each edge stores two integer endpoints and one double weight (3 fields) plus an object header, for an estimated size of about 32 bytes per edge.

TABLE VIII: Memory consumption for Prim's algorithm across graph types.

| Graph Type | $V$ | $E$ | Avg Memory (KB) | Memory/Vertex (KB) |
|---|---|---|---|---|
| Sparse – Small | 10 | 10 | 229.70 | 22.97 |
| Sparse – Medium | 50 | 50 | 9.09 | 0.18 |
| Sparse – Med.-Large | 100 | 100 | 15.15 | 0.15 |
| Sparse – Large | 500 | 500 | 84.73 | 0.17 |
| Sparse – Very Large | 1000 | 1000 | 141.09 | 0.14 |
| Dense – Small | 10 | 31 | 3.02 | 0.30 |
| Dense – Medium | 50 | 857 | 22.73 | 0.45 |
| Dense – Med.-Large | 100 | 3,465 | 63.25 | 0.63 |
| Dense – Large | 200 | 13,930 | 549.92 | 2.75 |
| Dense – Very Large | 500 | 87,325 | 3,353.29 | 6.71 |

**Memory Usage Analysis for Kruskal's Algorithm.** Table IX presents the memory consumption of Kruskal's algorithm over the same set of sparse and dense graphs. For *sparse graphs* ($E \approx V$), the memory usage scales roughly linearly with the number of edges. The small sparse case ($V = 10$, $E = 10$) shows a relatively high 0.71 KB per edge, which is largely due to fixed JVM and measurement overheads. As the instance size grows, the memory per edge quickly stabilizes: for $V = 50$ and $V = 100$, it drops to 0.37 and 0.32 KB per edge, and for $V = 500$ and $V = 1000$ it settles around 0.20–0.25 KB per edge. This trend indicates that once fixed overheads are amortized, Kruskal's memory footprint is dominated by the storage of the edge list plus a small $O(V)$ union–find structure, yielding an overall $O(E)$ space profile for sparse graphs.

For *dense graphs*, where $E$ grows toward 70% of the maximum $V(V-1)/2$, the total memory usage increases substantially, reflecting the need to hold *all* edges explicitly. At $V = 10$ and $E = 31$, the footprint is still modest (9.07 KB), but as we move to $V = 50$ with 857 edges and $V = 100$ with 3,465 edges, memory usage rises to about 203 KB and 672 KB, respectively. The dense $V = 200$ case ($E \approx 13{,}930$) requires roughly 2.28 MB, and the large dense graph with $V = 500$ and 87,325 edges reaches about 11 MB. Interestingly, the *memory-per-edge* metric decreases as $E$ grows (from $\approx 0.29$ KB/edge down to $\approx 0.13$ KB/edge) because fixed overheads become less significant, but the total memory still scales linearly with $E$. Overall, the measurements confirm that for Kruskal's algorithm, memory is effectively proportional to the number of edges, particularly in dense graphs where the edge list is very large.

**Memory Components (Kruskal's Implementation).** The observed memory behavior can be explained by the main data structures used in the implementation:

- `ArrayList<Edge> (allEdges)`: stores a complete copy of all graph edges. Its size is proportional to $E$, with each `Edge` object occupying roughly 32 bytes, giving an estimate of $\approx 0.03E$ KB for this component alone. This is the dominant term for dense graphs.
- `UnionFind`: typically implemented using two integer arrays, `parent[]` and `rank[]`. Together they require about $2 \times (V + 1) \times 4$ bytes $\approx 8V$ bytes, which is $O(V)$ and relatively small compared to the edge list when $E \gg V$.
- `ArrayList<Edge> (mstEdges)`: stores the $V - 1$ edges of the final MST. Its size is $O(V)$, and for large graphs it contributes only a small fraction of the overall memory compared to `allEdges`.

- *Sorting overhead*: the call to `Collections.sort()` may allocate temporary auxiliary storage proportional to $O(E)$, further reinforcing the $O(E)$ memory behavior in dense settings.

TABLE IX: Memory consumption for Kruskal's algorithm across graph types.

| Graph Type | $V$ | $E$ | Avg Memory (KB) | Memory/Edge (KB) |
|---|---|---|---|---|
| Sparse – Small | 10 | 10 | 7.05 | 0.71 |
| Sparse – Medium | 50 | 50 | 18.66 | 0.37 |
| Sparse – Med.-Large | 100 | 100 | 31.98 | 0.32 |
| Sparse – Large | 500 | 500 | 127.09 | 0.25 |
| Sparse – Very Large | 1000 | 1000 | 202.51 | 0.20 |
| Dense – Small | 10 | 31 | 9.07 | 0.29 |
| Dense – Medium | 50 | 857 | 202.91 | 0.24 |
| Dense – Med.-Large | 100 | 3,465 | 672.22 | 0.19 |
| Dense – Large | 200 | 13,930 | 2,278.32 | 0.16 |
| Dense – Very Large | 500 | 87,325 | 10,985.48 | 0.13 |

*3) Edge Selection Order:* Table X compares the exact order in which Prim's and Kruskal's algorithms select edges when computing the MST for the predefined input graph (`graph.txt`, 20 vertices). Although both algorithms produce an MST with the same total weight (29.27), their edge selection sequences differ substantially.

Out of the 19 MST edges, only a single edge $\big((8, 16)$ with weight $4.00\big)$ appears in the *same* position in both orders (19th), corresponding to just $1/19 \approx 5.3\%$ positional matches. However, all 19 edges in Prim's MST also appear in Kruskal's MST and vice versa, so the *set* of edges is identical (100% edge overlap). The "Weight Rank" column shows, for each step, how the chosen edge is ranked by weight across the MST edges. For example, Prim selects $(1, 2)$ as its first edge, even though it is only the fifth-lightest edge overall (5th / 1st), while Kruskal begins with the globally lightest edge $(11, 12)$.

On average, the same edge appears about 6.4 positions earlier or later in one algorithm compared to the other, highlighting the difference in strategy:

- **Prim's algorithm** grows a single tree from the starting vertex and always chooses the minimum-weight edge on the current frontier. Its selection order therefore reflects the *spatial expansion* of the tree through the graph.
- **Kruskal's algorithm** considers all edges globally in non-decreasing order of weight and adds them whenever they do not create a cycle. Early steps tend to connect small local components with very light edges (e.g., $(11, 12)$, $(13, 14)$, $(17, 18)$, $(19, 20)$), even if these are far from the starting vertex used by Prim.

This experiment illustrates an important point: different greedy strategies can explore the graph in very different ways and produce very different edge selection traces, yet still converge to the same optimal MST.

TABLE X: Comparison of MST edge selection order for `graph.txt` (20 vertices).

| Order | Prim's Edge | Weight | Kruskal's Edge | Weight | Weight Rank |
|---|---|---|---|---|---|
| 1 | $(1, 2)$ | 0.50 | $(11, 12)$ | 0.25 | 5th / 1st |
| 2 | $(1, 3)$ | 0.53 | $(13, 14)$ | 0.25 | 6th / 2nd |
| 3 | $(2, 5)$ | 0.86 | $(17, 18)$ | 0.25 | 7th / 3rd |
| 4 | $(2, 4)$ | 1.00 | $(19, 20)$ | 0.25 | 8th / 4th |
| 5 | $(3, 7)$ | 1.20 | $(1, 2)$ | 0.50 | 9th / 5th |
| 6 | $(3, 6)$ | 1.50 | $(1, 3)$ | 0.53 | 10th / 6th |
| 7 | $(4, 9)$ | 1.53 | $(2, 5)$ | 0.86 | 11th / 7th |
| 8 | $(5, 11)$ | 1.86 | $(2, 4)$ | 1.00 | 12th / 8th |
| 9 | $(11, 12)$ | 0.25 | $(3, 7)$ | 1.20 | 1st / 9th |
| 10 | $(4, 8)$ | 2.00 | $(3, 6)$ | 1.50 | 13th / 10th |
| 11 | $(6, 13)$ | 2.20 | $(4, 9)$ | 1.53 | 14th / 11th |
| 12 | $(13, 14)$ | 0.25 | $(5, 11)$ | 1.86 | 2nd / 12th |
| 13 | $(5, 10)$ | 2.50 | $(4, 8)$ | 2.00 | 15th / 13th |
| 14 | $(7, 15)$ | 2.53 | $(6, 13)$ | 2.20 | 16th / 14th |
| 15 | $(8, 17)$ | 2.86 | $(5, 10)$ | 2.50 | 17th / 15th |
| 16 | $(17, 18)$ | 0.25 | $(7, 15)$ | 2.53 | 3rd / 16th |
| 17 | $(9, 19)$ | 3.20 | $(8, 17)$ | 2.86 | 18th / 17th |
| 18 | $(19, 20)$ | 0.25 | $(9, 19)$ | 3.20 | 4th / 18th |
| 19 | $(8, 16)$ | 4.00 | $(8, 16)$ | 4.00 | 19th / 19th |
| *Total MST weight: Prim = 29.27, Kruskal = 29.27 (verified equal)* | | | | | |

## IV. Comparative Analysis

This section presents a detailed comparison of Prim's and Kruskal's algorithms on the generated test graphs. We contrast their runtimes and memory footprints for different graph densities and sizes, and connect these observations back to the theoretical complexity analysis.

### A. Algorithm Selection Criteria

*1) Prim's Algorithm:* Prim's algorithm is generally preferred when the graph is dense, stored as adjacency lists, or when we want to grow the MST from a specific starting vertex.

*a) Graph characteristics favoring Prim.:* Prim's algorithm is especially well suited for *dense* or *moderately dense* graphs, where the number of edges is a large fraction of the maximum possible. In our experiments, for graphs with roughly $70\%$ density and $V \geq 100$, Prim consistently outperformed Kruskal in both time and memory. Prim also works very naturally when the graph is stored as an adjacency list, since it repeatedly explores neighbors of the current tree. If the graph is connected and relatively dense, and we care about performance on larger $V$, Prim is usually the better choice.

*b) Real-world scenarios.:* Prim's algorithm is a good choice in scenarios where:

- The graph is dense or close to dense (e.g., network design with many possible links between nodes).
- The graph is naturally represented as adjacency lists (e.g., road networks, communication graphs).
- A specific starting vertex is meaningful (e.g., building a spanning tree rooted at a central hub or server).
- Memory usage is a concern for large graphs and we want to avoid holding a full global edge list in memory.

In such settings, Prim provides fast runtimes, predictable memory usage, and a straightforward implementation.

*2) Kruskal's Algorithm:* Kruskal's algorithm is a good choice for sparse, edge-list graphs where sorting all edges is practical.

*a) Graph characteristics favoring Kruskal.:* Kruskal's algorithm is most attractive for *sparse* graphs, where the number of edges is close to the number of vertices ($E \approx V$). In this regime, sorting all edges is relatively cheap, and our results show that Kruskal can be slightly faster than Prim on very small and very large sparse graphs. Kruskal is also a natural fit when the input is given as an *edge list* rather than adjacency lists, since its main step is "sort all edges and then scan them".

*b) Real-world scenarios.:* Kruskal's algorithm is a good choice in scenarios where:

- The graph is sparse (e.g., tree-like networks, graphs with relatively few connections).
- The input naturally comes as a list of edges (e.g., log of network links, file of weighted edges).
- We may need to build a minimum spanning *forest* for a graph that is not fully connected.
- Parallel or external sorting is available, making it easy to handle large edge lists efficiently.

In these cases, Kruskal's global edge-sorting approach is simple to code, works well with edge-centric data, and can exploit parallel hardware when available.

### *B. Graph Result Analysis*

*1) Sparse Graph Results:* We first compare Prim's and Kruskal's algorithms on *sparse graphs*, where $E \approx V$. Tables XI and XII summarize the execution time and memory usage across all sparse configurations.

TABLE XI: Execution time comparison on sparse graphs.

| Vertices | Prim (ms) | Kruskal (ms) | Winner | Advantage |
|---|---|---|---|---|
| 10 | 3.989 | 3.357 | Kruskal | Kruskal $\approx 18.83\%$ faster |
| 50 | 2.845 | 2.488 | Kruskal | Kruskal $\approx 14.36\%$ faster |
| 100 | 2.493 | 2.554 | Prim | Prim $\approx 2.37\%$ faster |
| 500 | 2.794 | 3.171 | Prim | Prim $\approx 11.89\%$ faster |
| 1000 | 2.852 | 2.761 | Kruskal | Kruskal $\approx 3.31\%$ faster |

TABLE XII: Memory usage comparison on sparse graphs.

| Vertices | Prim (KB) | Kruskal (KB) | Memory Comparison |
|---|---|---|---|
| 10 | 229.70 | 7.05 | Kruskal; Prim uses $32.6\times$ more |
| 50 | 9.09 | 18.66 | Prim; Prim uses $0.49\times$ Kruskal |
| 100 | 15.15 | 31.98 | Prim; Prim uses $0.47\times$ Kruskal |
| 500 | 84.73 | 127.09 | Prim; Prim uses $0.67\times$ Kruskal |
| 1000 | 141.09 | 202.51 | Prim; Prim uses $0.70\times$ Kruskal |

For sparse graphs where the number of edges is about the same as the number of vertices ($E \approx V$), theory says that Prim's and Kruskal's algorithms should take about the same amount of time, because both have running time on the order of $V \log V$. Our measurements in Table XI mostly agree with this: for very small graphs ($V \leq 50$), Kruskal is about 14–$19\%$ faster, mainly because sorting a small number of edges is very cheap in Java, while Prim's priority queue has a bit more overhead. Around $V = 100$, both algorithms are almost identical in speed, which fits the idea that sorting and heap operations cost about the same here. For $V = 500$, Prim becomes clearly faster (about $12\%$) because it only keeps and processes edges that are on the current frontier of the tree, while Kruskal still sorts the entire edge list. At $V = 1000$, Kruskal is slightly faster again (about $3\%$), but this difference is small enough that it may just be due to measurement noise and Java runtime effects. The memory results in Table XII are easier to interpret: except for the very tiny case $V = 10$, Prim always uses less memory for $V \geq 50$, because it stores only the needed helper structures (visited set, heap, MST edges), while Kruskal must keep a full list of all edges plus the union–find arrays. Overall, for sparse graphs both algorithms run in about the same time, but Prim is usually a bit more memory-friendly on larger inputs.

*2) Dense Graphs Results:* For dense graphs, the number of edges grows close to the maximum possible for a simple undirected graph. In our tests, each dense graph has about $70\%$ of the maximum edges, so

$$E \approx 0.7 \cdot \frac{V(V-1)}{2} = \Theta(V^2).$$

TABLE XIII: Execution time comparison on dense graphs.

| Vertices | Prim (ms) | Kruskal (ms) | Winner | Advantage |
|---|---|---|---|---|
| 10 | 2.429 | 2.502 | Prim | Prim $\approx 2.91\%$ faster |
| 50 | 3.184 | 3.330 | Prim | Prim $\approx 4.38\%$ faster |
| 100 | 3.506 | 4.912 | Prim | Prim $\approx 28.63\%$ faster |
| 200 | 4.434 | 8.450 | Prim | Prim $\approx 47.53\%$ faster |
| 500 | 12.870 | 40.173 | Prim | Prim $\approx 67.96\%$ faster |

TABLE XIV: Memory usage comparison on dense graphs.

| Vertices | Prim (KB) | Kruskal (KB) | Memory Comparison |
|---|---|---|---|
| 10 | 3.02 | 9.07 | Prim; Prim uses $0.33\times$ Kruskal |
| 50 | 22.73 | 202.91 | Prim; Prim uses $0.11\times$ Kruskal |
| 100 | 63.25 | 672.22 | Prim; Prim uses $0.09\times$ Kruskal |
| 200 | 549.92 | 2278.32 | Prim; Prim uses $0.24\times$ Kruskal |
| 500 | 3353.29 | 10985.48 | Prim; Prim uses $0.31\times$ Kruskal |

In dense graphs, both algorithms have to deal with a very large number of edges, but they do it in different ways. Prim's algorithm only pushes edges that are on the current "frontier" of the tree into a priority queue, while Kruskal's algorithm first collects *all* edges into a list and then sorts them. Table XIII shows that, as the graphs get denser and larger, Prim becomes clearly faster: for $V = 100$ it is already about $29\%$ faster, for $V = 200$ it is almost twice as fast, and for the dense graph with $V = 500$ it is about $68\%$ faster than Kruskal. This is exactly what we expect when sorting $O(V^2)$ edges becomes the main bottleneck for Kruskal.

The memory results in Table XIV show an even stronger difference. Kruskal's algorithm needs to store the complete edge list and extra temporary data for sorting, so its memory usage quickly reaches hundreds of kilobytes and then several megabytes as $V$ grows (about $11$ MB for the dense graph with $V = 500$). Prim's algorithm still needs more memory for dense graphs (because there are more edges), but it uses much less than Kruskal in every dense test, typically between one third and one tenth of the memory. In simple terms: on dense graphs, Prim is both faster and more memory-efficient, because it avoids sorting and storing the entire edge set at once.

*3) Large Graphs Results:* For large graphs, we focus on the configurations with $V \geq 500$. These tests highlight how each algorithm scales in practice when the graph size becomes substantial.

TABLE XV: Execution time comparison on large graphs ($V \geq 500$).

| Graph Type | Vertices | Prim (ms) | Kruskal (ms) | Winner / Advantage |
|---|---|---|---|---|
| Large sparse | 500 | 3.093 | 3.028 | Kruskal $\approx 2.16\%$ faster |
| Large dense | 500 | 12.870 | 40.173 | Prim $\approx 67.96\%$ faster |
| Very large sparse | 1000 | 2.852 | 2.761 | Kruskal $\approx 3.31\%$ faster |

TABLE XVI: Memory usage comparison on large graphs ($V \geq 500$).

| Graph Type | Vertices | Prim (KB) | Kruskal (KB) | Memory Comparison |
|---|---|---|---|---|
| Large sparse | 500 | 84.73 | 127.09 | Prim; Prim uses $0.67\times$ Kruskal |
| Large dense | 500 | 3353.29 | 10985.48 | Prim; Prim uses $0.31\times$ Kruskal |
| Very large sparse | 1000 | 141.09 | 202.51 | Prim; Prim uses $0.70\times$ Kruskal |

On large *sparse* graphs (500 and 1000 vertices with $E \approx V$), both algorithms run very quickly and have almost the same execution time: they are all around 3 ms. Kruskal is slightly faster in these cases (about $2\%$–$3\%$), but the difference is so small that it is likely due to low-level effects in Java and the hardware (such as caching and JIT optimizations), rather than a real change in asymptotic behavior. In other words, for large sparse graphs, Prim and Kruskal are practically tied in speed.

The most interesting case is the large *dense* graph with $V = 500$ and about $87{,}000$ edges. Here, Kruskal has to sort a very large edge list, and its runtime jumps to about $40$ ms, while Prim finishes in about $13$ ms. This means Prim is almost three times faster for this graph. The memory numbers tell a similar story: Kruskal uses around $11$ MB of memory, while Prim uses about $3.3$ MB, so Prim needs only about a third of the space.

For the large sparse graphs, Prim also uses less memory than Kruskal (about $30\%$–$35\%$ less), because Kruskal always keeps a full edge list and union–find arrays, while Prim mainly uses adjacency lists, a visited set, a heap of frontier edges, and the MST edges. Overall, the large-graph experiments confirm the main pattern from the previous sections: when the graph is

sparse, both algorithms behave similarly and are very fast; when the graph is large and dense, Prim is clearly the better choice in both time and memory.

## V. Research Component

This section talks about recent advancements in minimum spanning tree (MST) algorithms and related areas of research.

### A. Recent Advancement in MST Algorithms

In addition to the classical algorithms of Prim and Kruskal, recent research has focused on making minimum spanning trees (MSTs) practical for large, evolving, and temporally annotated graphs. This section highlights a few representative advances that go beyond static MST construction and address incremental updates, fully dynamic graphs, and application-driven improvements.

*1) Incremental MST and Temporal Graphs via AM-Trees:*

*a) Context and Motivation.:* Many real-world networks (social, transportation, communication, sensor networks) are *temporal*: edges appear over time or change weights as new data arrive. Re-running a static MST algorithm after each update is wasteful for tasks like temporal connectivity queries or sliding-window analytics. Ding et al. address this by studying *incremental* MST algorithms tailored to temporal graphs, where edges are only inserted and never deleted [1].

*b) Methodology and Key Innovations.:* The core idea is to reduce a broad class of temporal graph problems to incremental MST maintenance plus *path-max* queries on the tree. To do this efficiently, the authors introduce the *Anti-Monopoly Tree* (AM-tree), a specialized tree structure built on top of a transformed MST. AM-trees support edge insertions, updates to the MST, and maximum-edge queries along paths in $O(\log n)$ amortized time, with practical variants that trade some eagerness of updates for speed while preserving the same asymptotic bounds.

*c) Comparison and Limitations.:* Asymptotically, AM-tree–based algorithms match the $O(\log n)$ amortized bounds of classic dynamic structures (e.g., link-cut trees), but experiments show substantial constant-factor improvements: updates and queries can be several times faster on large temporal graphs. Unlike naively re-running Prim's or Kruskal's algorithm, AM-trees reuse most of the existing MST and scale to millions of edges. A key limitation is that the approach is *incremental* (insert-only); full support for deletions or large weight decreases still requires more general fully dynamic MST techniques.

*2) MST-Based Clustering and Approximate Single-Linkage:*

*a) Context and Motivation.:* MSTs are also used as a tool in clustering, most notably through their connection to single-link hierarchical clustering. Recent work revisits MST-based clustering to better understand when these methods are competitive with modern clustering algorithms and how to scale them to large datasets [2, 3].

*b) Methodology and Key Innovations.:* Gagolewski et al. systematically evaluate MST-based clustering schemes (including Genie and information-theoretic MST cuts) on benchmark datasets, showing that MST methods can be surprisingly competitive with non-MST approaches such as $k$-means, Gaussian mixtures, spectral clustering and density-based methods [2]. Complementary work by Okkels et al. focuses on *approximate* single-linkage clustering: they use graph-based nearest-neighbor indexes to build an approximate MST of a reachability graph, then apply Kruskal-like procedures to derive a clustering, achieving near-linear empirical scaling on large, high-dimensional datasets [3].

*c) Comparison and Applications.:* Compared to classical partitional algorithms, MST-based clustering naturally handles irregular, non-spherical cluster shapes and does not require a fixed number of clusters in advance. However, constructing exact MSTs on very large, dense datasets can be expensive, motivating approximate MSTs and representative-point strategies. Typical application domains include spatial data analysis, image segmentation, and high-dimensional pattern discovery, where MSTs provide an interpretable "skeleton" of the data.

### B. Additional Areas of Reseach

Beyond the specific advances discussed above, several broader research directions around MSTs are actively being explored:

*a) Incremental MST Algorithms.:* Incremental MST methods focus on efficiently updating the MST when new edges or vertices are added, without recomputing from scratch. This is particularly important for streaming and online settings where graphs grow over time.

*b) Dynamic and Temporal MSTs.:* Fully dynamic MST algorithms support both insertions and deletions, and sometimes weight changes, in evolving graphs. Temporal MSTs add time as an explicit dimension, using MSTs to answer connectivity questions over time intervals in dynamic networks.

*c) MSTs in Clustering Applications.:* MST-based clustering continues to evolve, with new cutting criteria, robustness improvements, and approximate methods designed for high-dimensional or noisy data, often combining MSTs with density or distance-based rules.

*d) Special Data Structures.:* Modern MST algorithms increasingly rely on advanced dynamic-tree structures such as AM-trees, link-cut trees, and tree-contraction frameworks. These data structures aim to bridge the gap between strong theoretical guarantees and practical performance on large, real-world graphs.

## VI. Conclusion

In this report, we implemented Prim's and Kruskal's minimum spanning tree algorithms in Java and evaluated their performance on sparse, dense, and large randomly generated graphs. Both algorithms consistently produced MSTs with identical total weight, confirming correctness. On sparse graphs, their runtimes were very similar, with small constant-factor differences, while Prim generally used less memory. On dense graphs, Prim clearly outperformed Kruskal in both time and memory, since Kruskal must sort a much larger edge set. These empirical results support our selection guidelines: Prim is preferable for dense, adjacency-list graphs and memory-sensitive settings, whereas Kruskal remains attractive for sparse, edge-list graphs, especially when efficient sorting is available. We also briefly highlighted recent research directions such as incremental MSTs for temporal graphs and MST-based clustering, showing how classical MST ideas extend to modern, dynamic, and data-driven applications.

## References

[1] Xiangyun Ding, Yan Gu, and Yihan Sun. "New Algorithms for Incremental Minimum Spanning Trees and Temporal Graph Applications". In: *Proceedings of the Conference on Applied and Computational Discrete Algorithms (ACDA)*. SIAM, 2025, pp. 293–307.

[2] Marek Gagolewski et al. "Clustering with Minimum Spanning Trees: How Good Can It Be?" In: *Journal of Classification* 42 (2025), pp. 90–112. DOI: 10.1007/s00357-024-09483-1.

[3] Camilla Birch Okkels et al. "Approximate Single-Linkage Clustering Using Graph-Based Indexes: MST-Based Approaches and Incremental Searchers". In: *Proceedings of SISAP*. 2025, pp. 233–247. ISBN: 978-3-032-06068-6.