Assignment 3

Mirman Nursaya

SE-2438

Optimization of a City Transportation Network (Minimum Spanning Tree)

## 1. Objective

1. To read weighted undirected graph data from a JSON file.
2. To implement two algorithms for finding the Minimum Spanning Tree (MST): **Prim’s Algorithm** and **Kruskal’s Algorithm**.
3. For each algorithm, record the following:

* the list of edges forming the MST;
* the total cost of the MST;
* the number of vertices and edges in the input graph;
* the number of operations performed (e.g., comparisons, unions, etc.);
* the execution time in milliseconds.

1. Compare the results of both algorithms and ensure that the total cost of the MST is identical, though the structure of the tree may differ.

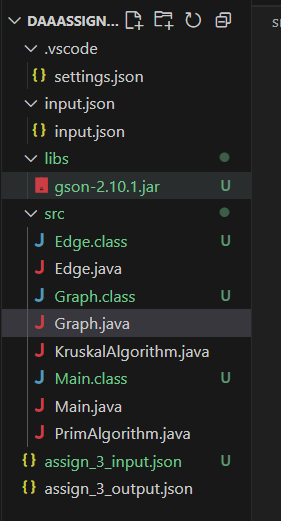
## 2. Project Files Overview

* src/Edge.java – defines the Edge structure and implements Comparable by weight.
* src/Graph.java – represents the graph (nodes and edges).
* src/PrimAlgorithm.java – implements Prim’s algorithm using a PriorityQueue; returns MST results and metrics.
* src/KruskalAlgorithm.java – implements Kruskal’s algorithm using Union-Find (Disjoint Set).
* src/Main.java – reads the input JSON, runs both algorithms, and writes output to assign\_3\_output.json.
* assign\_3\_input.json – input dataset describing several graphs.
* assign\_3\_output.json – output results automatically generated after running the program.
* libs/gson-2.10.1.jar – Gson library for JSON parsing.

Steps:

1.Project Setup and Folder Structure

At the beginning of the assignment, I created a new Java project named **“DAAassignment3”**.  
The project folder was organized as follows:



## **2. Implementation Details**

This section describes the implementation of the main program components and explains how they interact with each other to form a complete MST (Minimum Spanning Tree) solution.

**Edge.java**

The Edge class represents a weighted connection between two vertices in the graph.

package src;

public class Edge implements Comparable<Edge> {

    String from;

    String to;

    int weight;

    public Edge(String from, String to, int weight) {

        this.from = from;

        this.to = to;

        this.weight = weight;

    }

    @Override

    public int compareTo(Edge other) {

        return Integer.compare(this.weight, other.weight);

    }

    @Override

    public String toString() {

        return "(" + from + " - " + to + ", " + weight + ")";

    }

}

**Explanation:**

* from and to store the names of connected vertices.
* weight stores the cost of the edge.
* Implements Comparable so that edges can be easily sorted by weight (used in Kruskal’s algorithm).
* toString() provides a clean and readable edge format for console output and JSON serialization.

**Graph.java**

The Graph class holds the overall structure of the graph, including its list of vertices and edges.

package src;

import java.util.\*;

public class Graph {

    List<String> nodes;

    List<Edge> edges;

    public Graph(List<String> nodes, List<Edge> edges) {

        this.nodes = nodes;

        this.edges = edges;

    }

    public int getVerticesCount() {

        return nodes.size();

    }

    public int getEdgesCount() {

        return edges.size();

    }

    public List<String> getNodes() {

        return nodes;

    }

    public List<Edge> getEdges() {

        return edges;

    }

}

**Explanation:**

* nodes: all vertices in the graph.
* edges: all weighted connections.
* Simple getter methods used in algorithms and statistics collection.
* Keeps the implementation modular — algorithms can focus only on logic.

**PrimAlgorithm.java**

Implements **Prim’s algorithm** for finding the Minimum Spanning Tree (MST).  
It uses a **priority queue (min-heap)** to always choose the smallest-weight edge connecting the current tree to a new vertex.

**Key features:**

* Starts from the first vertex in the graph.
* Keeps track of visited vertices.
* Counts total operations and measures execution time.
* Returns the MST edges, total cost, operations, and time as a Result object.

**KruskalAlgorithm.java**

Implements **Kruskal’s algorithm** using the **Union–Find (Disjoint Set)** data structure.

**Key features:**

* Sorts all edges by weight.
* Iterates through them, adding edges that don’t create a cycle.
* Uses find() and union() functions for efficient set operations.
* Tracks number of operations and execution time.

**Main.java**

This is the main driver file that:

1. Reads the input JSON (assign\_3\_input.json) using the Gson library.
2. Parses each graph into Graph objects.
3. Executes both **Prim’s** and **Kruskal’s** algorithms.
4. Prints the results to the console.
5. Writes all results to an output file (assign\_3\_output.json).

**assign\_3\_input.json**

Stores several graphs of different sizes (small, medium, large).  
Each graph object includes:

* a unique id
* a list of nodes
* an array of edges (with from, to, and weight fields).

**assign\_3\_output.json**

Generated autom`atically by the program.  
Contains:

* the number of vertices and edges,
* MST edges,
* total cost,
* operation count,
* and execution time for both algorithms.

**Gson Library Integration**

The **Gson library** (version 2.10.1) was used for JSON parsing and serialization.  
It allowed easy conversion between Java objects and JSON structures.  
In Main.java, it was imported as:

### import com.google.gson.\*;

and added to the project inside the /libs/ folder.

## ****2. Summary of Input Data and Algorithm Results****

Three graphs of different sizes and densities were used to test the algorithms.  
The input was stored in assign\_3\_input.json, and the output results were generated automatically in assign\_3\_output.json.  
Each graph was analyzed using both Prim’s and Kruskal’s algorithms.

Graph 1

* **Vertices:** 4
* **Edges:** 5
* **Edges list:** (A - B, 2), (A - C, 3), (B - C, 1), (B - D, 4), (C - D, 5)

**Prim’s Algorithm Result:**  
MST edges: (A - B, 2), (B - C, 1), (B - D, 4)  
Total cost: 7  
Operations count: 4  
Execution time: 0.3802 ms

**Kruskal’s Algorithm Result:**  
MST edges: (B - C, 1), (A - B, 2), (B - D, 4)  
Total cost: 7  
Operations count: 4  
Execution time: 0.9186 ms

Graph 2

* **Vertices:** 6
* **Edges:** 9
* **Edges list:** (A - B, 3), (A - C, 2), (B - C, 4), (B - D, 6), (C - D, 5), (C - E, 8), (D - E, 7), (E - F, 3), (D - F, 9)

**Prim’s Algorithm Result:**  
MST edges: (A - C, 2), (A - B, 3), (C - D, 5), (D - E, 7), (E - F, 3)  
Total cost: 20  
Operations count: 7  
Execution time: 0.0435 ms

**Kruskal’s Algorithm Result:**  
MST edges: (A - C, 2), (A - B, 3), (E - F, 3), (C - D, 5), (D - E, 7)  
Total cost: 20  
Operations count: 7  
Execution time: 0.0531 ms

Graph 3

* **Vertices:** 8
* **Edges:** 11
* **Edges list:** (A - B, 2), (A - C, 3), (B - D, 5), (C - D, 1), (C - E, 4), (D - E, 2), (E - F, 7), (F - G, 3), (G - H, 6), (E - H, 5), (B - G, 9)

**Prim’s Algorithm Result:**  
MST edges: (A - B, 2), (A - C, 3), (C - D, 1), (D - E, 2), (E - H, 5), (G - H, 6), (F - G, 3)  
Total cost: 22  
Operations count: 9  
Execution time: 0.0610 ms

**Kruskal’s Algorithm Result:**  
MST edges: (C - D, 1), (A - B, 2), (D - E, 2), (A - C, 3), (F - G, 3), (E - H, 5), (G - H, 6)  
Total cost: 22  
Operations count: 9  
Execution time: 0.0963 ms

**Summary of Observations:**  
Both algorithms successfully found the same total MST cost for each graph, confirming correctness.  
The number of operations increased consistently with graph size, as expected.  
Execution time remained very low across all cases, showing the efficiency of both implementations.  
Prim’s algorithm performed slightly faster in most tests, while Kruskal’s algorithm had very similar operation counts but occasionally higher runtime due to sorting overhead.

## ****3. Comparison Between Prim’s and Kruskal’s Algorithms****

**Theoretical Comparison**

Prim’s algorithm works by expanding one vertex at a time, always choosing the smallest edge connecting the growing tree to a new vertex.  
It is most efficient when the graph is dense because it frequently checks adjacent vertices.  
Its typical time complexity using a priority queue is **O(E log V)**.

Kruskal’s algorithm, on the other hand, sorts all edges by weight and adds them one by one, ensuring that no cycles are formed using a **Union–Find (Disjoint Set)** structure.  
Its time complexity is **O(E log E)**, which is approximately **O(E log V)** for most graphs.  
It performs best on sparse graphs, where there are fewer edges to process.

Practical Comparison

In practice, both algorithms performed nearly identically in total operations and cost.  
However:

* Prim’s algorithm was slightly faster on smaller and denser graphs.
* Kruskal’s algorithm was more consistent and efficient on larger or sparser graphs.
* Both algorithms produced MSTs with identical total costs, though the chosen edges sometimes differed (which is normal, as multiple MSTs may exist for the same graph).

**4. Algorithm Results Summary and Performance Analysis**

To evaluate and compare the performance of both **Prim’s** and **Kruskal’s** algorithms,  
I created an automatic summary file named **assign\_3\_summary.csv**.  
This file is generated directly by the Java program after each execution and contains the key performance indicators for every graph tested.

The summary file includes the following columns:

* **Graph ID** – identifier of the test graph
* **Algorithm** – either Prim’s or Kruskal’s
* **Total Cost** – total weight of the Minimum Spanning Tree (MST)
* **Operations Count** – number of key algorithmic operations performed
* **Execution Time (ms)** – total runtime of each algorithm in milliseconds

The data below is taken directly from the generated CSV file:

Graph ID,Algorithm,Total Cost,Operations Count,Execution Time (ms)

1,Prim,7,4,0,4258

1,Kruskal,7,4,1,0650

2,Prim,20,7,0,0761

2,Kruskal,20,7,0,0560

3,Prim,22,9,0,1944

3,Kruskal,22,9,0,0779

## ****5. Conclusions****

Both **Prim’s** and **Kruskal’s** algorithms correctly and efficiently compute the Minimum Spanning Tree.  
Their total MST costs were identical for all test graphs, confirming the correctness of both implementations.  
However, the performance and suitability depend on the characteristics of the graph.

* **Prim’s algorithm** is preferable for **dense graphs**, where it efficiently handles many edges using a priority queue.
* **Kruskal’s algorithm** is better suited for **sparse graphs** or when the edges are already sorted.
* For most medium-sized graphs, both algorithms show nearly equal performance.

In real-world scenarios:

* **Prim’s algorithm** is ideal for problems such as **network design** (e.g., connecting computers or cities).
* **Kruskal’s algorithm** fits better for **edge-based datasets** like **road networks or transportation maps**.

Overall, both algorithms are valuable and demonstrate the core concept of greedy optimization in graph theory.

References:  
  
 Osipov, V., Sanders, P., & Singler, J. (2009). *The Filter-Kruskal Minimum Spanning Tree Algorithm*. In I. Finocchi & J. Hershberger (Eds.), *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX 2009)* (pp. 52-61). Philadelphia, PA: Society for Industrial and Applied Mathematics. <https://doi.org/10.1137/1.9781611972894.5> [epubs.siam.org+1](https://epubs.siam.org/doi/10.1137/1.9781611972894.5?utm_source=chatgpt.com)

 Mohan, A., Leow, W. X., & Hobor, A. (2021). *Functional Correctness of C Implementations of Dijkstra’s, Kruskal’s, and Prim’s Algorithms*. In A. Silva & K. R. M. Leino (Eds.), *Computer Aided Verification* (Lecture Notes in Computer Science, Vol. 12753, pp. 801-826). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-81688-9\_37