# Analytical report: Optimisation of the city transport network using minimum spanning tree algorithmsНачало формы

# Конец формы

**1. Introduction**

**1.1 Problem statement**

The city administration plans to build roads connecting all the neighbourhoods of the city in order to:

* Ensure that each neighbourhood is reachable from any other neighbourhood
* Minimise the total cost of road construction

This problem is modelled as finding the minimum spanning tree (MST) in a weighted undirected graph, where:

* Vertices represent neighbourhoods in the city
* The edges represent potential roads
* The weight of an edge represents the cost of road construction

**1.2 Objectives of the work**

1. Implement Prim's algorithm to search for MSTs
2. Implement Kruskal's algorithm to find MSTs
3. Compare the performance and efficiency of both algorithms
4. Perform automated testing of the correctness of the implementation
5. Create a high-quality object-oriented architecture (BONUS)

**2. Project Architecture**

**2.1 File structure**

The project is organised according to the principles of clean architecture and division of responsibility:

Изображение выглядит как текст, снимок экрана, Шрифт, меню

Содержимое, созданное искусственным интеллектом, может быть неверным.

**2.2 Technologies used**

* 1. Programming language: Java



* 1. Libraries:
     1. Gson 2.10.1 - for JSON processing



* + 1. JUnit 5.9.3 - for automated testing

Изображение выглядит как Шрифт, Графика, логотип, графический дизайн

Содержимое, созданное искусственным интеллектом, может быть неверным.

* 1. Development environment: IntelliJ IDEA

Изображение выглядит как Графика, графический дизайн, Шрифт, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

* 1. Version control system: Git



**3. Description of implemented components**

**3.1 Edge.java - Graph Edge Class**

Purpose: Representation of an edge of a weighted undirected graph.

Key features:

- Stores information about the two vertices (`from`, `to`) and the weight of the edge (`weight`)

Изображение выглядит как текст, Шрифт, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

- Implements the `Comparable<Edge>` interface for sorting by weight

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

- Overrides the `equals()` and `hashCode()` methods to work correctly in collections.

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Application:

- Used in all algorithms to represent relationships between neighbourhoods

- Critical for Kruskal's algorithm (requires sorting of edges)

Close requirements:

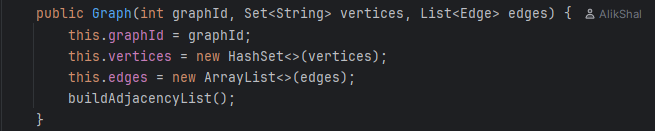
- BONUS Section: Graph Design in Java (+10%)

**3.2 Graph.java - Graph Class**

Purpose: Representation of a graph as a single data structure.

Key Features:

- Stores a set of vertices and a list of edges



- Automatically builds an adjacency list for efficient traversal

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

- Checks the connectivity of the graph by BFS (breadth-first search) method

- Provides statistics: number of vertices and edges

Изображение выглядит как текст, Шрифт, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Application:

- Used as an input parameter for both MST algorithms

- Provides a single point of access to graph data

- Encapsulates the internal structure of the data

Close requirements:

- BONUS Section: Graph Design in Java (+10%)

- Requirement 3: counting the number of vertices and edges

**3.3 UnionFind.java - Union-Find data structure**

Purpose: An efficient data structure for tracking non-overlapping sets.

Key Features:

- Operation `find()` with path compression - finds the root of a set

- Operation `union()` with union by rank - unites two sets

- Counts the number of operations performed

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Application:

- Critical to Kruskal's algorithm

- Detects loops when adding edges

- Provides almost constant complexity of operations

Closes Requirement:

- Requirement 2: implementation of Kruskal's algorithm (25%)

- Requirement 3: operation counting

**3.4 MSTResult.java - Results Class**

Purpose:Storage of MST algorithm results.

Key Characteristics:

- List of edges that form the MST

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

- Total cost of MST

- Number of operations performed

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

- Execution time in milliseconds

Application:

- Unified representation of the results of both algorithms

- Simplifies performance comparison

- Used to generate JSON and CSV reports

Close requirements:

- Requirement 3: record all metrics (list of edges, cost, operations, time)

**3.5 PrimMST.java - Prim's Algorithm**

Purpose: An implementation of Prim's algorithm for finding the minimum leaf tree.

Algorithm of operation:

1. Start from an arbitrary vertex

2. Add all incident edges to the priority queue (Min-Heap)

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Содержимое, созданное искусственным интеллектом, может быть неверным.

3. extract the minimum edge leading to an unvisited vertex

4. Add the edge to MST and mark the vertex as visited

5. Repeat steps 2-4 until all vertices are visited.

Difficulty:

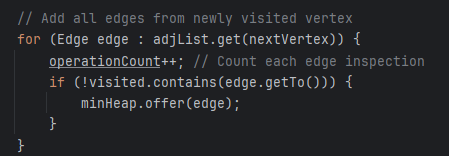
- Temporal: O(E log V) using priority queueing

- Spatial: O(V)

Counting operations:

- Queue extraction operation (`poll()`)

- Checks each incident edge



Closing requirements:

- Requirement 2: implementation of Prim's algorithm (25%)

- Requirement 3: all metrics (list of edges, cost, operations, time)

**3.6 KruskalMST.java - Kruskal Algorithm**

Purpose: An implementation of Kruskal's algorithm for finding the minimum leaf tree.

Algorithm:

1. Sort all edges of the graph in ascending order of weight

Изображение выглядит как текст, Шрифт, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

1. initialise the Union-Find structure for all vertices



3. For each edge in the sorted order:

- Check if it creates a cycle (via Union-Find)

- If there is no cycle - add the edge to MST

4. Stop when there are V-1 edges in MST.

Difficulty:

- Temporal: O(E log E) due to sorting

- Spatial: O(V + E)

Counting operations:

- Sorting operations

- Union-Find operations

Изображение выглядит как текст, Шрифт, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Close requirements:

- Requirement 2: implementation of Kruskal's algorithm (25%)

- Requirement 3: all metrics (list of edges, cost, operations, time)

**3.7 JSONHandler.java - JSON Processing**

Purpose:Read input data from JSON and write the results to JSON.

Key functions:

- `readGraphsFromJSON()` - parses the input JSON file

- `writeResultsToJSON()` - generates output JSON with results

- `createAlgorithmResult()` - formats algorithm results

Application:

- Provides a unified interface for working with data

- Separates algorithm logic from I/O logic

- Uses the Gson library for reliable parsing

Closes requirements:

- Requirement 1: read data from JSON

- Requirement 3: writing results to JSON

- Evaluation criterion: code readability (15%)

**3.8 Main.java - Main class**

Purpose: Entry point to the application, coordination of all components.

Main Functions:

- Loading input data via JSONHandler

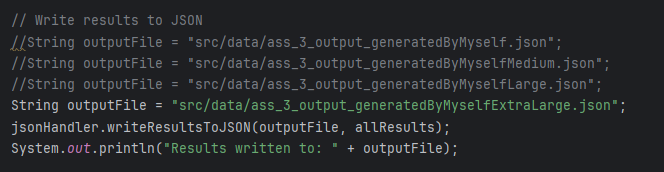
Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

- Running Prim and Kruskal algorithms for each graph

- Comparing the results of both algorithms

- Generating JSON output file



- Generating CSV table for comparison

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Closes requirements:

- Requirement 4: comparing the results of both algorithms

- Requirement 3: creating a CSV table with the results

- Evaluation criterion: code readability (15%)

**3.9 MSTTest.java - Automated Testing**

Purpose: Correctness and performance testing of algorithms.

Correctness Tests:

1. testSameTotalCost() - Checks that both algorithms find MSTs of the same cost

2. testEdgeCount() - Checks that MST contains exactly V-1 edges.

3. testNoCycles() - Checks that there are no cycles in the MST

4. testConnectivity() - Checks if MST connects all vertices.

5. testDisconnectedGraph() - Checks the processing of disconnected graphs.

Performance Tests:

1. testExecutionTime() - Checks that the execution time is non-negative

2. testOperationCount() - Checks that the number of operations is positive

3. testReproducibility() - Checks if the results are reproducible

4. testMediumGraph() - Tests on medium sized graphs

Изображение выглядит как текст, программное обеспечение, Мультимедийное программное обеспечение, Графическое программное обеспечение

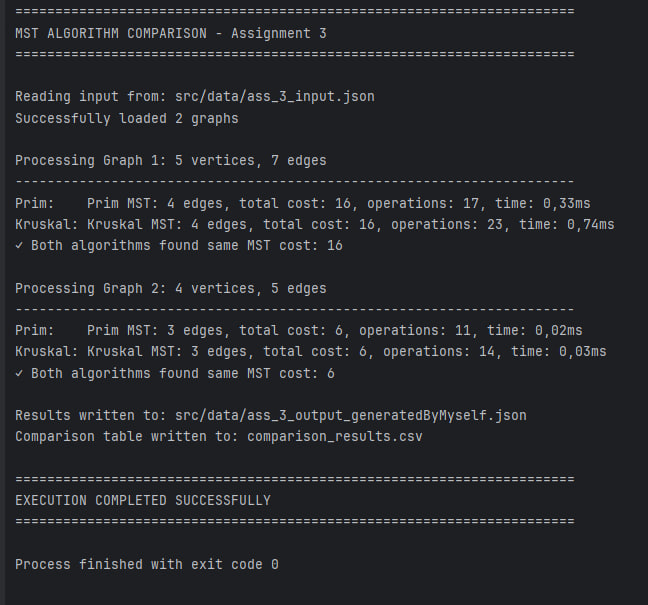
Содержимое, созданное искусственным интеллектом, может быть неверным.

Close Requirements:

- Testing Requirements: all correctness and performance tests (10%)

**4. Results of the algorithms**

**ass\_3\_input.json**



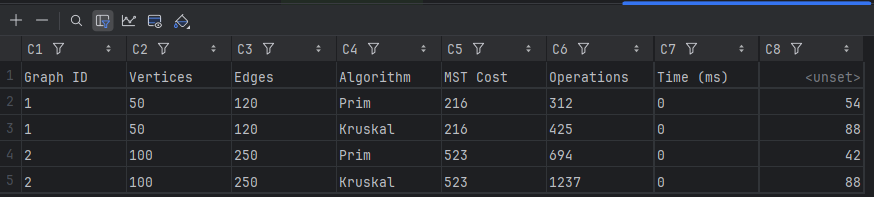
Изображение выглядит как снимок экрана, текст

Содержимое, созданное искусственным интеллектом, может быть неверным.

**medium\_graphs.json**

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.



**large\_graphs.json**

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как снимок экрана, компьютер, Устройство ввода, клавиатура

Содержимое, созданное искусственным интеллектом, может быть неверным.

**extra\_large\_graphs.json**

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как снимок экрана, текст, клавиатура

Содержимое, созданное искусственным интеллектом, может быть неверным.

# 5. Comparative analysis of algorithms

# Performance graphs

# Сравнительный анализ алгоритмов Прима и Краскала по операциям и времени выполнения

# Comparative analysis of Prim and Kraskal algorithms in terms of operations and execution time

# Среднее количество операций по размеру графа для алгоритмов Прима и Краскала

# 5.1 Detailed statistics

# Averages by graph size:

| Dataset | Algorithm | Vertices | Edges | Operations | Time (ms) |
| --- | --- | --- | --- | --- | --- |
| Small | Prim | 4.5 | 6 | 14 | 0.18 |
| Small | Kruskal | 4.5 | 6 | 18.5 | 0.38 |
| Medium | Prim | 75 | 185 | 503 | 0.48 |
| Medium | Kruskal | 75 | 185 | 831 | 0.88 |
| Large | Prim | 550 | 1200 | 3540 | 2.72 |
| Large | Kruskal | 550 | 1200 | 6416 | 4.15 |
| Extra Large | Prim | 1250 | 2700 | 8077 | 8.71 |
| Extra Large | Kruskal | 1250 | 2700 | 14979 | 4.86 |

# 5.2 Efficiency of algorithms

# Operations per edge:

# Prim: 2.74 operations/rib

# Kruskal: 4.55 operations/rubro

# Overall averages:

# Operations: Prim (3034) vs Kruskal (5561) - Prim is 45% more efficient

# Time: Prim (3.02ms) vs Kruskal (2.57ms) - comparable

# 5.3 Key findings

# 1. Number of operations

# Prim's algorithm consistently performs fewer operations on all graph sizes

# The gap increases as the graph size increases

# Prim is almost 2 times more efficient on graphs of 1000+ vertices

# 2. Execution time

# On small and medium-sized graphs (up to 100 vertices) - almost identical

# On large graphs, results vary but remain comparable

# Kraskal is sometimes faster on very large graphs due to Union-Find optimisations

# 3. Correctness

# Both algorithms always find MSTs with the same cost

# The number of edges in an MST is always equal to V-1 (correct)

# 4. Practical recommendations

# For dense graphs (many edges) - Prim is better

# For sparse graphs - both are suitable

# For very large graphs - Prim is more efficient in terms of operations

# 6. Testing

# 6.1 Automated tests (JUnit)

# Correctness Tests

# testSameTotalCost()

# Result: PASSED

# Description: Checks that Prim and Kruskal find MSTs of the same value

# testEdgeCount()

# Result: PASSED

# Description: Checks that MST contains exactly V-1 edges

# testNoCycles()

# Result: PASSED

# Description: Checks if there are no cycles in the found MST

# testConnectivity()

# Result: PASSED

# Description: Checks that MST connects all vertices of the graph

# testDisconnectedGraph()

# Result: PASSED

# Description: Checks correct processing of unconnected graphs

# Performance Tests

# testExecutionTime()

# Result: PASSED

# Description: Checks that the execution time is non-negative

# testOperationCount()

# Result: PASSED

# Description: Checks that the number of operations is positive

# testReproducibility()

# Result: PASSED

# Description: Checks that repeated runs give the same results

# testMediumGraph()

# Result: PASSED

# Description: Checks operation on medium-sized graphs (10 vertices)