Optimization of a City Transportation Network — Project Report

This concise report summarizes the implementation and results for the Minimum Spanning Tree (MST) assignment. The project implements Prim’s and Kruskal’s algorithms, measures operation counts and execution time, and produces JSON output for each input graph.

1. Input

Input file: ass\_3\_input.json.

Format (example):

{  
 "graphs": [  
 {  
 "id": 1,  
 "nodes": ["A", "B", "C", "D", "E"],  
 "edges": [  
 {"from":"A","to":"B","weight":4},  
 {"from":"A","to":"C","weight":3},  
 ...  
 ]  
 }  
 ]  
}

Each graph is an undirected weighted graph; edges are given once per undirected edge.

2. Implementation overview

Languages & files: - Java (no external libraries required) - src/Main.java — program entry: reads input, runs algorithms, writes ass\_3\_output.json. - src/JsonSimpleParser.java — small parser targeted at the provided input format. - src/Graph.java, src/Edge.java — graph models. - src/MST.java — implementations of Prim and Kruskal and result statistics. - src/UnionFind.java — disjoint set union used by Kruskal.

Design notes: - The parser is intentionally small and tailored to the expected JSON structure. It is not a general-purpose JSON parser but keeps the project dependency-free. - Operation counting is explicit and intended to reflect meaningful algorithmic actions (PQ pushes/polls, edge examinations for Prim; edge examinations and union-find operations for Kruskal). This matches the assignment requirement to record operation counts. - Execution time is measured using System.nanoTime() and reported as milliseconds (double with two decimals).

3. Algorithms (short descriptions and important code excerpts)

Prim’s algorithm (high level): - Start from an arbitrary node. - Use a min-heap (priority queue) of candidate edges crossing the visited set. - Repeatedly extract minimum-weight edge that connects to an unvisited node.

Key code excerpt from MST.primMST:

PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(e -> e.weight));  
visited.add(startNode);  
for (Edge e : adj.get(startNode)) pq.add(e); // push  
while (!pq.isEmpty() && res.mstEdges.size() < nodes.size()-1) {  
 Edge e = pq.poll(); // pop  
 if (visited.contains(e.to)) continue;  
 visited.add(e.to);  
 res.mstEdges.add(e);  
 // push neighbors of new node  
}

Kruskal’s algorithm (high level): - Sort edges by weight. - Iterate edges in increasing order. - Use union-find to add an edge if it connects two different components.

Key code excerpt from MST.kruskalMST:

edges.sort(Comparator.comparingInt(e -> e.weight));  
UnionFind uf = new UnionFind(n);  
for (Edge e : edges) {  
 int a = idx.get(e.from), b = idx.get(e.to);  
 int ra = uf.find(a), rb = uf.find(b);  
 if (ra != rb) {  
 uf.union(ra, rb);  
 res.mstEdges.add(e);  
 }  
}

4. Results (example)

When run on the provided ass\_3\_input.json, the program produces ass\_3\_output.json. Example per-graph output (excerpt):

{  
 "results": [  
 {  
 "graph\_id": 1,  
 "input\_stats": { "vertices": 5, "edges": 7 },  
 "prim": { "mst\_edges": [...], "total\_cost": 16, "operations\_count": 42, "execution\_time\_ms": 1.52 },  
 "kruskal": { "mst\_edges": [...], "total\_cost": 16, "operations\_count": 37, "execution\_time\_ms": 1.28 }  
 }  
 ]  
}

Notes: - Both algorithms report identical MST total cost (a correctness requirement). The actual chosen edges can be the same or differ in order/structure but must have equal total weight. - Operation counts and execution times will vary by implementation details and the runtime environment.

5. Comparison & Analysis

Correctness: - Both implementations produce MSTs with identical total cost on valid connected graphs.

Efficiency (practical summary): - Prim (using binary heap): O(E log V) — good for dense and sparse graphs; adjacency representation important. - Kruskal (with sorting + union-find): O(E log E) dominated by sorting; performs well when edges are the focus. For sparse graphs E ≈ V, Kruskal’s cost is similar to Prim’s.

When to prefer: - Use Prim when you have an adjacency representation and want better performance on dense graphs (with optimized heap or Fibonacci heap improvements for theoretical bounds). - Use Kruskal when edges are already listed or when you want a simple union-find-based implementation; it’s often simpler to implement when input is an edge list.

Operation-count considerations: - Operation counts capture algorithmic work but are implementation-dependent. The project counts PQ operations and union/find calls as primary actions.

6. Conclusions

* Both algorithms are implemented correctly and produce MSTs with equal total weights.
* Choice between Prim and Kruskal depends on graph density and input representation.

7. How to run (Windows PowerShell)

From the project root (where src and ass\_3\_input.json are located):

# compile  
javac -d out src\\*.java -encoding UTF-8  
# run  
if ($LASTEXITCODE -eq 0) { java -cp out Main }

After running, ass\_3\_output.json will be created in the project root.

If javac is not found, install JDK and ensure javac and java are on PATH.