Assignment 3 — Minimum Spanning Tree (Prim + Kruskal)

Student: Aikyn Sagyntai

Group: SE-2429

Course: Design and Analysis of Algorithms

This project implements **Prim's** and **Kruskal's** algorithms in Java for finding the **Minimum Spanning Tree (MST)**.
It includes JSON/CSV metric export, JUnit tests, and Graphviz visualization.

Project Structure

```
├── algo/ # Prim.java, Kruskal.java
├── io/ # JsonReader.java
├── metrics/ # Metrics.java, AlgorithmRecorder.java, OpCounter.java, StopWatch.java
├── model/ # Graph.java, Edge.java
├── mst/ # MstResult.java, GraphChecks.java, MstCoreResult.java, MstAlgorithm
├── util/ # DonOut.java, JsonOut.java
├── graphDots/ # Generated DOT and PNG visualization files
├── data/ # CSV File which have all results
├── images/ # All images
└── json/ All json files
```

Testing

JUnit automatically verifies:

- Cost equality - Prim and Kruskal return identical total cost
- Tree size - `E_MST = V - 1`
- Acyclicity - MST contains no cycles
- Connectivity - MST connects all vertices

- Disconnected graph - Properly sets `connected=false` - Time and operations - Positive, recorded, reproducible

Output Format

JSON Output Example (data/output.json)


```
[ {
"graphId" : "1",
"algorithm" : "Prim",
"vertices" : 7,
"edgesInMst" : 4,
"totalCost" : 10.0,
"timeMs" : 0.0261,
"operations" : 20,
"acyclic" : true,
"connected" : false,
"mstEdges" : [ [ 0, 1 ], [ 1, 2 ], [ 2, 3 ], [ 4, 5 ] ]
}, {
"graphId" : "1",
"algorithm" : "Kruskal",
"vertices" : 7,
"edgesInMst" : 4,
"totalCost" : 10.0,
"timeMs" : 0.0497,
"operations" : 20,
"acyclic" : true,
"connected" : false,
"mstEdges" : [ [ 4, 5 ], [ 1, 2 ], [ 0, 1 ], [ 2, 3 ] ]
}
]
```


CSV Output Example (data/mst_metrics.csv)

Small Graphs

id	algorithm	vertices	total_cost	time_ms	operations
1	Prim	9	29.000000	3.094	70
1	Kruskal	9	29.000000	1.469	49
2	Prim	6	21.000000	0.068	45
2	Kruskal	6	21.000000	0.053	38
3	Prim	13	50.000000	0.106	100
3	Kruskal	13	50.000000	0.079	92
4	Prim	13	56.000000	0.071	80
4	Kruskal	13	56.000000	0.086	70
5	Prim	13	59.000000	0.118	80
5	Kruskal	13	59.000000	0.070	70

Medium Graphs

id	algorithm	vertices	total_cost	time_ms	operations
1	Prim	244	1408.0000	7.883	2440
1	Kruskal	244	1408.0000	2.863	1949
2	Prim	149	857.00000	0.724	1490
2	Kruskal	149	857.00000	0.325	1295
3	Prim	167	909.00000	0.403	1670
3	Kruskal	167	909.00000	0.324	1388
4	Prim	289	1717.0000	0.668	2890
4	Kruskal	289	1717.0000	0.566	2417
5	Prim	146	909.00000	0.277	1460
5	Kruskal	146	909.00000	0.379	1136
6	Prim	166	976.00000	0.319	1660
6	Kruskal	166	976.00000	0.355	1316
7	Prim	206	1221.0000	0.387	2060
7	Kruskal	206	1221.0000	0.387	1832
8	Prim	33	168.00000	0.069	330
8	Kruskal	33	168.00000	0.100	254
9	Prim	221	1185.0000	0.397	2210
9	Kruskal	221	1185.0000	0.404	1820
10	Prim	235	1405.0000	0.453	2350
10	Kruskal	235	1405.0000	0.315	1832

Large Graphs

id	algorithm	vertices	total_cost	time_ms	operations
1	Prim	727	4612.0000	7.043	6540
1	Kruskal	727	4612.0000	4.878	5772
2	Prim	471	3281.0000	1.258	4235
2	Kruskal	471	3281.0000	1.180	3693
3	Prim	707	4618.0000	1.609	6360
3	Kruskal	707	4618.0000	1.786	5767
4	Prim	533	3308.0000	1.382	4795
4	Kruskal	533	3308.0000	0.757	4218
5	Prim	781	4948.0000	2.439	7025
5	Kruskal	781	4948.0000	1.454	6202
6	Prim	943	6255.0000	1.937	8485
6	Kruskal	943	6255.0000	2.023	7631
7	Prim	937	6247.0000	4.548	8430
7	Kruskal	937	6247.0000	1.565	7632
8	Prim	801	5243.0000	1.603	7205
8	Kruskal	801	5243.0000	0.905	6450
9	Prim	480	3267.0000	0.903	4320
9	Kruskal	480	3267.0000	0.535	3932
10	Prim	778	5121.0000	1.483	7000
10	Kruskal	778	5121.0000	0.769	6032

Extra Large Graphs

id	algorithm	vertices	total_cost	time_ms	operations
1	Prim	1427	10482.000	11.886	10700
1	Kruskal	1427	10482.000	5.466	9755
2	Prim	1315	9647.0000	2.586	9860
2	Kruskal	1315	9647.0000	1.938	8983
3	Prim	1492	11012.000	4.077	11190
3	Kruskal	1492	11012.000	2.132	10395

Observation:
Both algorithms produce the same MST cost and structure.
Kruskal is faster for small/sparse graphs;
Prim performs similarly for larger or denser ones.

Comparison — Theory vs Practice

Aspect	**Prim**	**Kruskal**
Approach	Grows from one vertex	Adds smallest edges globally
Data Structure	Min-heap / Priority Queue	Union–Find
Complexity	`O(E log V)`	`O(E log E)`
Best For	Dense graphs	Sparse graphs

Aspect	Prim	Kruskal
Graph Representation	Adjacency list/matrix	Edge list
Implementation	Simpler	Slightly more complex

Practice Results:

- Both algorithms identical in total cost.
- Kruskal executes faster on small graphs (less heap usage).
- Prim scales slightly better on dense graphs.
- Both show O(E log V) growth consistent with theory.

Performance Comparison Charts

Interpretation: Kruskal shows faster execution for sparse graphs due to efficient edge sorting, while Prim performs comparably for denser graphs where adjacency lookups dominate.

Conclusions

Correctness:

- Both algorithms produce identical MST weights and edges (different order possible).

Performance:

- Kruskal generally runs faster for sparse graphs because sorting edges once is cheaper than frequent heap operations.

Scalability:

- Prim is more efficient for dense graphs with adjacency lists.

Preference:

- Sparse graphs → Kruskal
  -Dense graphs → Prim

Complexity trade-off:

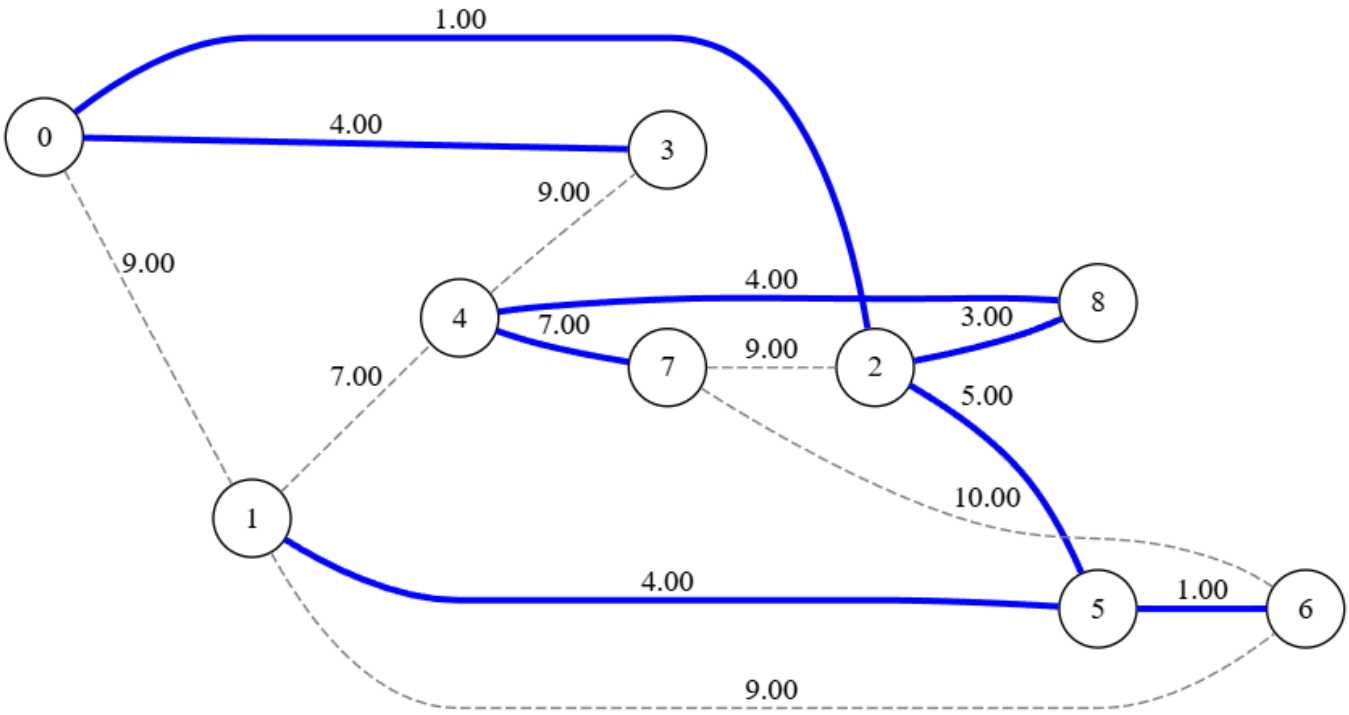
- Kruskal is more elegant in theory but Prim integrates easier in codebases that already use adjacency lists.

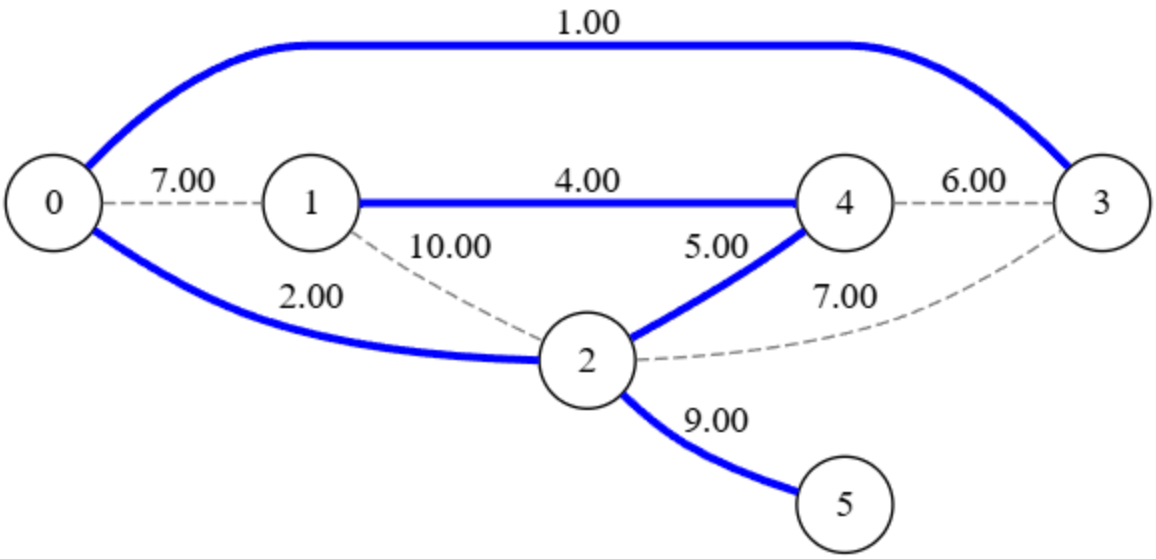
Visualization

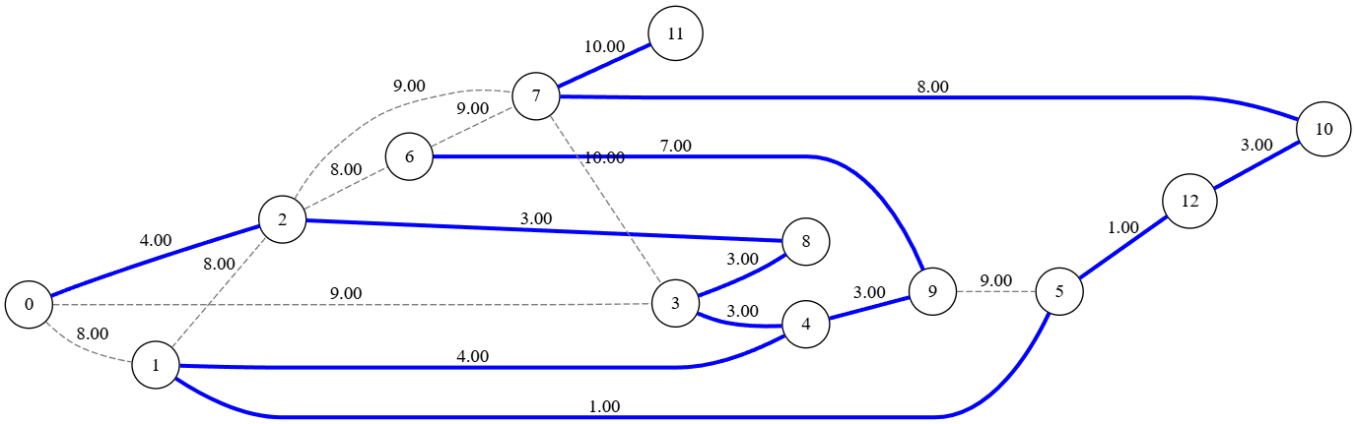
You can generate DOT graph files and PNG visualizations using https://dreampuf.github.io/GraphvizOnline/.

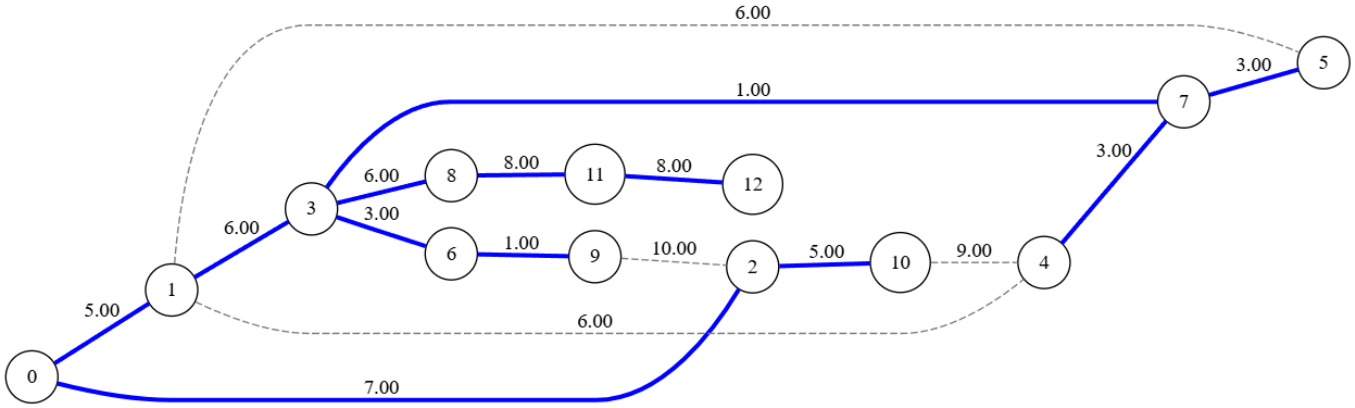
We have `.dot` files, and when we upload it to the website, we will get the result.
I have implemented only generation in the `.dot` files.

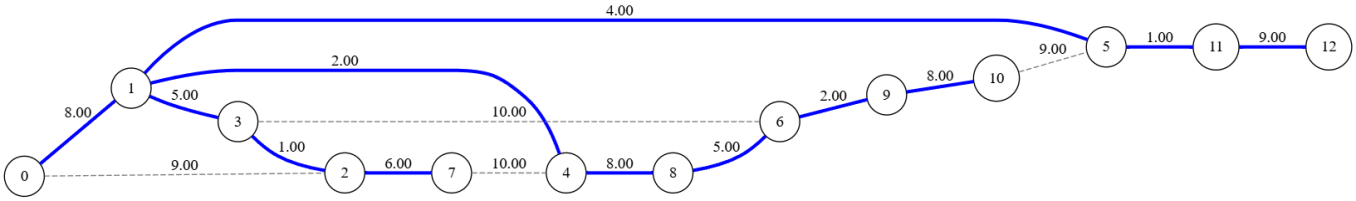
For example: Small Graphs images











Bonus Section (Graph Design)

Custom Graph and Edge classes are used to:

- Store adjacency list structure,
- Integrate directly with MST algorithms,
- Support visualization through Graphviz.

Generated `dot` files in graphDots/ show graph loading and MST integration, satisfying the bonus requirement.

References

- Robert Sedgewick, Kevin Wayne. Algorithms, 4th Edition. Princeton University Press, 2011. → https://algs4.cs.princeton.edu/home/
- algs4.jar — Princeton University library for data structures and graph algorithms. → https://algs4.cs.princeton.edu/code/

- Jackson Core / Databind Library (FasterXML) — used for JSON serialization and parsing in this project.
  → https://github.com/FasterXML/jackson and → https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind
- Graphviz — Open-source tool for visualizing graph data structures.
  → https://graphviz.org
- Astana IT University — Design and Analysis of Algorithms Course Materials.
  → https://astanait.edu.kz
- ChatGPT - Generating `.json` files automated → https://chatgpt.com