Assignment 4 – Smart City / Smart Campus Scheduling

1. Objective

The purpose of this assignment was to integrate two main graph-algorithm topics in one practical case study related to Smart City / Smart Campus scheduling:

1. Detection of **Strongly Connected Components (SCCs)** and construction of a **topological order** of condensed components.
2. Computation of **Shortest and Longest Paths** in Directed Acyclic Graphs (DAGs).

The scenario models real city-service dependencies (street cleaning, maintenance, sensors, etc.), where some tasks form cycles and must be grouped, while others remain acyclic and can be scheduled optimally.

2. Implementation Overview

Language and Tools: Java 17, Maven, JUnit 5, Gson

Project structure

- graph.scc – TarjanSCC and CondensationBuilder
- graph.topo – KahnTopo (topological sort)
- graph.dagsp – DagShortestPaths (shortest and longest paths)
- metrics – Metrics interface and SimpleMetrics implementation
- io, model – data structures and JSON parser
- Main – main program executing the entire pipeline

Processing pipeline

1. Load graph data from a JSON file.
2. Apply Tarjan's algorithm to detect SCCs and their sizes.
3. Build a condensation graph (DAG of components).
4. Perform topological sorting using Kahn's algorithm.
5. Compute single-source shortest paths and the critical (longest) path.
6. Measure execution time and operation counts for every step.

Metrics recorded

- SCC phase: DFS visits and edges
- Topological sort: queue pushes and pops
- DAG paths: number of edge relaxations
- Timing via System.nanoTime()

Weight model: edge-based weights ("weight_model": "edge")

3. Datasets

Nine datasets were prepared to test algorithm behavior on graphs of different sizes and densities.

Category	File	n	Edges	Type	Description
Small	small1_dag.json	7	7	DAG	Simple chain of tasks
Small	small2_one_cycle.json	8	8	Cyclic	One three-vertex cycle
Small	small3_two_cycles.json	9	9	Cyclic	Two independent cycles
Medium	medium1_mixed.json	14	14	Mixed	One SCC and long DAG chain
Medium	medium2_many_scc.json	16	19	Mixed	Several two-vertex SCCs
Medium	medium3_dense_dag.json	12	18	DAG	High edge density
Large	large1_sparse_mixed.json	24	24	Mixed	Sparse graph with two SCCs
Large	large2_dense_mixed.json	28	29	Mixed	Dense graph after compression
Large	large3_dag_perf.json	40	39	DAG	Performance and scaling test

Each dataset was automatically processed by Main, and results were saved in the /out folder.

4. Experimental Results

4.1 SCC and Topological Ordering

Dataset	SCC count	SCC time (ns)	DFS visits	Topo time (ns)	Push/Pops
small1	7	**28 600**	7	**13 000**	7 / 7
small2	6	**57 600**	8	**16 900**	6 / 6
small3	6	**66 200**	9	**40 300**	6 / 6
medium1	12	**93 200**	14	**20 500**	12 / 12
medium2	12	**49 800**	16	**21 800**	12 / 12
medium3	12	**57 400**	12	**38 100**	12 / 12
large1	20	**347 500**	24	**119 900**	20 / 20
large2	26	**122 200**	28	**64 700**	26 / 26
large3	40	**122 700**	40	**273 200**	40 / 40

Observations
SCC execution time increases proportionally to the number of vertices and edges.
Dense graphs lead to more DFS edges and slightly higher runtime.
Kahn’s algorithm remains consistently fast and scales linearly with graph size.

4.2 Shortest and Longest Paths on DAG

Dataset	SP relax	SP time (ns)	LP relax	LP time (ns)	Critical Path Length
small1	6	7 300	6	6 100	4
small2	5	8 900	5	8 600	5
small3	4	13 000	4	8 400	4
medium1	11	10 100	11	8 700	7
medium2	11	9 800	11	8 900	11
medium3	11	17 100	11	15 200	4
large1	18	85 600	18	106 300	18
large2	25	31 100	25	28 900	16
large3	14	33 900	14	28 100	7

Observations
Shortest-path and longest-path algorithms show linear scaling with the number of edges.

Relaxation counts and times correspond closely to the number of vertices in the DAG. Critical-path lengths reflect graph depth and confirm correct computation.

5. Analysis

5.1 Algorithm Behavior

The experimental data in Tables 4.1 and 4.2 confirm that all implemented algorithms—Tarjan's SCC detection, Kahn's topological sorting, and DAG shortest/longest path computations—behaved in accordance with their theoretical time complexities.

Strongly Connected Components (SCC).
Tarjan's algorithm demonstrated a clear linear relationship between input size and runtime. For instance, the smallest graph (*small1_dag.json*, n = 7, E = 7) completed in **28 600 ns**, whereas the largest dataset (*large3_dag_perf.json*, n = 40, E = 39) required **122 700 ns**.
This 5.7× increase in vertices resulted in approximately 4.3× higher runtime, which aligns with the expected complexity of **O(V + E)**.
Moreover, the number of DFS visits exactly matched the number of vertices in every acyclic graph, confirming that no redundant traversals occurred.
In cyclic graphs such as *small2_one_cycle.json* and *small3_two_cycles.json*, SCC time rose to **57 600–66 200 ns**, reflecting the additional stack operations required to process back-edges within cycles.

Topological sorting.
Kahn's algorithm produced consistent and predictable results.
Even for larger DAGs (*large3_dag_perf.json*, 40 vertices),
topological ordering completed in **273 200 ns**, only about twice as long as
large1_sparse_mixed.json (24 vertices, **119 900 ns**).
Operation counts—pushes and pops—always equaled the number of vertices, confirming linear complexity **O(V + E)** and the absence of redundant queue operations.
The successful generation of a valid topological order for each condensation graph also proves that all condensed graphs were acyclic.

Shortest and longest path algorithms.
Both DAG-SP algorithms exhibited linear scaling.
For example, *small1_dag.json* performed **6 relaxations** in **7 300 ns**,
while *large1_sparse_mixed.json* required **18 relaxations** and **85 600 ns**.
The runtime increase closely follows the edge count, as each edge is relaxed once.
The longest-path computation (critical path) consistently mirrored shortest-path performance in both relaxation count and total time, verifying that both traversed the DAG in identical topological order as expected.

5.2 Effect of Structure and Density

Graph structure—especially the presence of cycles and edge density—had a measurable effect on performance.

- **Acyclic graphs** (e.g., *small1_dag.json*, *medium3_dense_dag.json*, *large3_dag_perf.json*) exhibited the smallest SCC times, because each vertex formed its own component and DFS visited every node exactly once.

In *small1*, SCC = 7 and time = 28 600 ns, compared to *small2* with cycles (SCC = 6, time = 57 600 ns).

- **Cyclic or mixed graphs** required additional internal DFS work.
  For example, *small3_two_cycles.json* (two independent cycles) completed in **66 200 ns**, and *medium2_many_scc.json*—which contained multiple 2-vertex cycles—required **49 800 ns**, despite a relatively small size (n = 16).
  These results confirm that cycles increase recursion depth and stack operations.
- **Dense DAGs** showed a moderate rise in both SCC and SP times due to higher edge counts.
  Medium3_dense_dag.json (E = 18) required **57 400 ns** for SCC, higher than *medium1_mixed.json* (E = 14, 93 200 ns) despite fewer vertices, since additional edges increased DFS edge traversals.
  Similar behavior appeared in DAG-SP results, where *medium3* had **17 100 ns** versus **9 800 ns** for *medium2*.

In summary, SCC time depends not only on the number of vertices but also directly on **edge density** and **presence of cycles**.
Graphs with multiple back-edges cause more recursive updates and thus higher execution time.

5.3 Performance Scaling and Bottlenecks

Across all datasets, **SCC detection** consistently consumed the majority of total runtime.
In *large1_sparse_mixed.json*, SCC required **347 500 ns**, while the combined Topo + SP stages totaled around **205 000 ns**.
This demonstrates that the initial DFS phase dominates computation, as it processes every vertex and edge exactly once.

Topological sort and **DAG-SP** stages, on the other hand, remained lightweight and linearly proportional to V + E.
Even for the largest graphs, the total number of relaxations never exceeded the number of edges, confirming the correctness of implementation.

The **critical-path lengths** reported in Table 4.2 also validate algorithmic accuracy:
short DAGs (e.g., *small1*, *small3*) produced critical lengths of 4–5,
while deeper graphs (*medium2_many_scc.json*) reached 11 and *large1_sparse_mixed.json* reached 18.
This directly matches the logical depth of dependencies encoded in the datasets.
Therefore, the computed longest paths accurately represent real scheduling bottlenecks.

5.4 Summary of Findings

- Tarjan's SCC runtime scales linearly with (V + E) and increases in dense or cyclic graphs.
- Kahn's topological sort is fast and stable, confirming acyclicity of all condensation graphs.
- DAG shortest- and longest-path algorithms show linear runtime with respect to edge count.
- SCC detection is the dominant computational bottleneck in the full scheduling pipeline.
- Critical-path lengths correlate precisely with task-dependency depth, validating correctness of both SCC compression and DAG-SP computations.

6. Conclusions

- SCC compression is essential for handling cyclic dependencies in scheduling tasks.
- Topological ordering defines a safe and efficient execution order for planning systems.
- Shortest-path analysis provides earliest start times; longest-path analysis identifies critical chains.
- All implemented algorithms demonstrated linear complexity and stable performance for graphs up to 40–50 vertices.
- In practical scheduling problems:
    - For mixed graphs, use the full sequence **SCC → Condensation → Topo → DAG-SP**.
    - For pure DAGs, the pipeline can start from the topological step.
- Edge-based weights were sufficient for representing task durations; the model can be extended with node-based costs if required.

7. Reproducibility

To reproduce the results:

1. Place JSON datasets in the /data folder.
2. Run the project with

mvn clean compile exec:java -Dexec.mainClass="com.meaemb.Main"

The project can be executed either using the Maven command above or simply by running the Main class from IntelliJ IDEA.

or directly run the Main class from the IDE.
3. Reports will be generated in /out.
4. JUnit tests under src/test/java verify correctness for SCC, topological sort, and DAG path algorithms.