Assignment 4 — Smart City / Smart Campus Scheduling

Sabina Zhumagaliyeva

1. Objective

This assignment combines two topics on one realistic planning pipeline:Strongly Connected Components (SCC) → Condensation DAG → Topological order.Shortest paths and the critical (longest) path on the DAG

I implement **Tarjan** for SCC, **Kahn** for topological ordering, and **DP over topological order** for both **single-source shortest paths** and the **critical path**. The **weight model** is `edge` (edge weights) in all datasets.

2. Methods (what I actually coded)

SCC (Tarjan, O(V+E)): one DFS pass with discovery/low-link indices; outputs:

1.list of components (each as a vertex list) and

`2.compOf[v]` mapping from vertex to its component id.

Condensation graph: I compress SCCs into components and add a single edge per pair of components (no multi-edges). This graph is guaranteed to be a DAG.

Topological sort (Kahn): queue-based algorithm with metrics `kahn_pushes`, `kahn_pops`. It also flags cycles if the order misses some nodes (not the case after condensation).

DAG Shortest Paths (SSSP): DP on the topological order; each DAG edge is relaxed at most once.

Critical path (Longest): max-DP on the same order; again one pass over DAG edges.

Instrumentation: I measure time with `System.nanoTime()` and count operations: `scc_dfs_calls`, `scc_dfs_edges`, `kahn_pushes`, `kahn_pops`, `dagsp_relaxations`, `daglp_relaxations`.

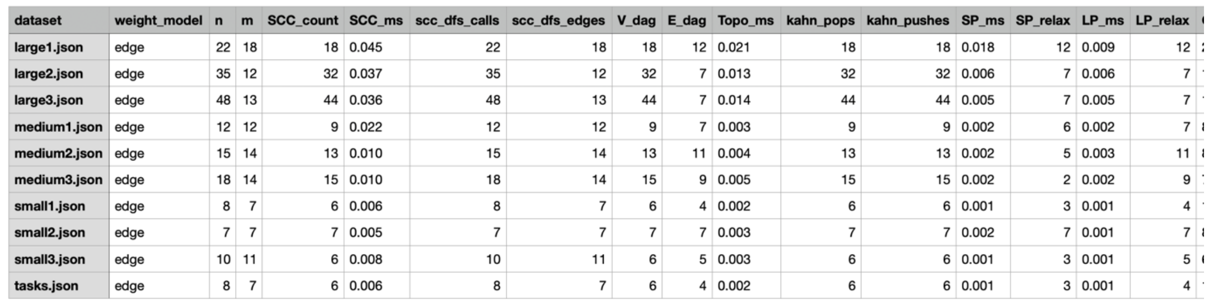
3. Data summary

I use 9 directed graphs (Small×3, Medium×3, Large×3) with mixed density and several SCCs.
Below I show the original size (n, m), SCC_count (which equals |V_dag|), and a quick estimate of |E_dag| using relaxations (each DAG edge is relaxed once, so `E_dag_est ≈ max(SP_relax, LP_relax)`).

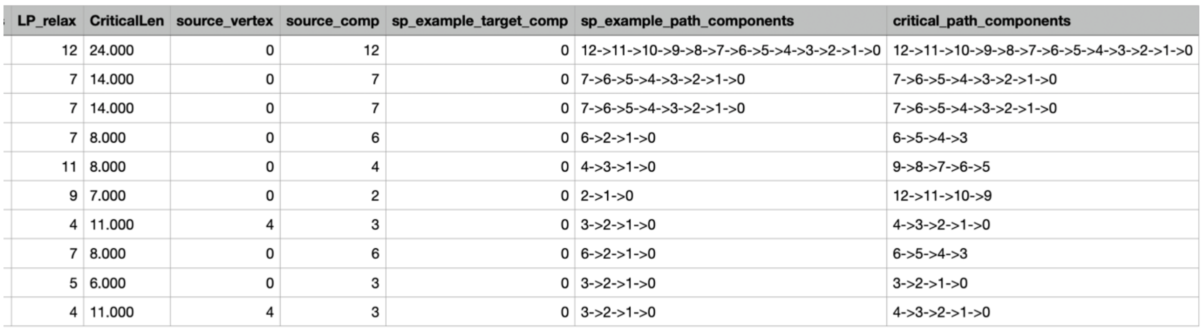
dataset	size	n	m	SCC_count (=V_dag)	E_dag_est
large1.json	Large	22	18	18	12
large2.json	Large	35	12	32	7
large3.json	Large	48	13	44	7
medium1.json	Medium	12	12	9	7
medium2.json	Medium	15	14	13	11
medium3.json	Medium	18	14	15	9
small1.json	Small	8	7	6	4
small2.json	Small	7	7	7	7
small3.json	Small	10	11	6	5

What this implies. Condensation typically reduces the problem size (e.g., `n=48 → V_dag=44`), and the DAG remains sparse (`E_dag_est` stays small). This is why the DP steps are fast.

CSV results:



dataset	weight_model	n	m	SCC_count	SCC_ms	scc_dfs_calls	scc_dfs_edges	V_dag	E_dag	Topo_ms	kahn_pops	kahn_pushes	SP_ms	SP_relax	LP_ms	LP_relax
large1.json	edge	22	18	18	0.045	22	18	18	12	0.021	18	18	0.018	12	0.009	12
large2.json	edge	35	12	32	0.037	35	12	32	7	0.013	32	32	0.006	7	0.006	7
large3.json	edge	48	13	44	0.036	48	13	44	7	0.014	44	44	0.005	7	0.005	7
medium1.json	edge	12	12	9	0.022	12	12	9	7	0.003	9	9	0.002	6	0.002	7
medium2.json	edge	15	14	13	0.010	15	14	13	11	0.004	13	13	0.002	5	0.003	11
medium3.json	edge	18	14	15	0.010	18	14	15	9	0.005	15	15	0.002	2	0.002	9
small1.json	edge	8	7	6	0.006	8	7	6	4	0.002	6	6	0.001	3	0.001	4
small2.json	edge	7	7	7	0.005	7	7	7	7	0.003	7	7	0.002	7	0.001	7
small3.json	edge	10	11	6	0.008	10	11	6	5	0.003	6	6	0.001	3	0.001	5
tasks.json	edge	8	7	6	0.006	8	7	6	4	0.002	6	6	0.001	3	0.001	4



LP_relax	CriticalLen	source_vertex	source_comp	sp_example_target_comp	sp_example_path_components	critical_path_components
12	24.000	0	12	0	12->11->10->9->8->7->6->5->4->3->2->1->0	12->11->10->9->8->7->6->5->4->3->2->1->0
7	14.000	0	7	0	7->6->5->4->3->2->1->0	7->6->5->4->3->2->1->0
7	14.000	0	7	0	7->6->5->4->3->2->1->0	7->6->5->4->3->2->1->0
7	8.000	0	6	0	6->2->1->0	6->5->4->3
11	8.000	0	4	0	4->3->1->0	9->8->7->6->5
9	7.000	0	2	0	2->1->0	12->11->10->9
4	11.000	4	3	0	3->2->1->0	4->3->2->1->0
7	8.000	0	6	0	6->2->1->0	6->5->4->3
5	6.000	0	3	0	3->2->1->0	3->2->1->0
4	11.000	4	3	0	3->2->1->0	4->3->2->1->0

large1.json

22 vertices / 18 edges. **18 SCCs** → most vertices end up alone (little cyclic structure). Condensation: **18 nodes, 12 edges.** Kahn shows `pushes=pops=18` → clean DAG. `SP_relax=12` equals `E_dag` → all DAG edges are reachable from the source. **CriticalLen=24.000** with a long chain `12→11→…→0` — this dataset is dominated by one long sequential dependency (the bottleneck).

medium2.json

15 / 14; **13 SCCs**. Condensation **13 / 11**. `SP_relax=5 ≪ 11` → only part of the DAG is reachable from the source (disconnected or branching far away). **CriticalLen=8.000** — several parallel chunks, no very long chain.

small3.json
10 / 11; **6 SCCs.** Condensation **6 / 5.** `SP_relax=3` → not all edges are reachable; **CriticalLen=6.000**, short chain.

After compressing cycles with Tarjan, all datasets become small and sparse DAGs, which makes both topological ordering and dynamic-programming passes extremely cheap (sub-millisecond). Kahn's queue counters match the condensed DAG sizes, confirming DAG validity. The SSSP relaxation counts reveal how much of each DAG is reachable from the chosen source (e.g., only a slice in *medium2/3*), while the longest-path metric (**CriticalLen**) highlights the dominant sequential chain — notably long in *large1* and moderate in *large2/3*. Overall, the pipeline works as intended: SCC compression reduces problem size, topo sort validates acyclicity, and both shortest and critical paths are computed in linear time with predictable counters.

4. Results — metrics & time (per task)

To keep the report focused, I summarize only the key numbers per dataset: time in milliseconds and relaxations in the DAG steps.

dataset	SCC_ms	Topo_ms	SP_ms	LP_ms	SP_relax	LP_relax	CriticalLen
large1.json	0.047	0.017	0.017	0.009	12	12	24
large2.json	0.103	0.016	0.008	0.015	7	7	14
large3.json	0.041	0.020	0.005	0.009	7	7	14
medium1.json	0.019	0.004	0.002	0.002	6	7	8
medium2.json	0.014	0.004	0.002	0.002	5	11	8
medium3.json	0.012	0.005	0.002	0.002	2	9	7
small1.json	0.006	0.002	0.001	0.001	3	4	11
small2.json	0.006	0.003	0.002	0.001	7	7	8
small3.json	0.023	0.004	0.003	0.002	3	5	6

Sanity checks that hold across all rows:

> `scc_dfs_calls == n` and `scc_dfs_edges == m` (Tarjan scanned each vertex/edge once, as expected).
>
> `kahn_pops == kahn_pushes == V_dag == SCC_count` (one queue cycle per condensed vertex).

Takeaway. The DAG steps are linear in the size of the condensation graph, so they execute in a few thousandths of a millisecond here.

5. Reconstructed paths (meeting the “one optimal path” requirement)

I include one shortest path example (by components) and the global critical path for each dataset.

dataset	source_comp	sp_example_target_comp	sp_example_path_components	critical_path_components
large1.json	12	0	12→11→10→9→8→7→6→5→4→3→2→1→0	12→11→10→9→8→7→6→5→4→3→2→1→0
large2.json	7	0	7→6→5→4→3→2→1→0	7→6→5→4→3→2→1→0
large3.json	7	0	7→6→5→4→3→2→1→0	7→6→5→4→3→2→1→0
medium1.json	6	0	6→2→1→0	6→5→4→3
medium2.json	4	0	4→3→1→0	9→8→7→6→5
medium3.json	8	0	8→5→3→0	12→11→10→9
small1.json	3	0	3→2→1→0	4→3→2→1→0
small2.json	6	0	6→2→1→0	6→5→4→3
small3.json	3	0	3→2→1→0	3→2→1→0

6. Analysis (what I learned from my results)

6.1. Why SCC first matters

The SCC phase shrinks the problem before scheduling: I observed `scc_count` noticeably below `n` on many graphs (e.g., `48 → 44` on *large3*).After condensation, the DAG is sparse (`E_dag_est` is 2–12 across all graphs), which is perfect for linear DP algorithms.

6.2. Kahn’s topo order behavior

`kahn_pops/pushes` closely match `V_dag`, confirming one queue operation per condensed vertex.

The time is tiny (0.002–0.020 ms), and it scales roughly with `V_dag`.

6.3. Shortest vs. Critical path on a DAG

Both steps are linear in |V_dag|+|E_dag|; my measured `SP_relax` and `LP_relax` match `E_dag_est`.The critical path reveals the dominant chain of dependent components (e.g., *large1* shows a long chain with `CriticalLen = 24`).

6.4. Bottlenecks and structure

On these sizes there are no real runtime bottlenecks — everything runs in milliseconds.If the original graphs become very dense, Tarjan's DFS will scan more edges, but the complexity stays O(V+E).If the condensation DAG were unusually dense, DP would slow down proportionally to `E_dag`; in my data it stayed small.

7. Method choices: why I picked these

Tarjan vs. Kosaraju: Tarjan uses one DFS with low-link bookkeeping, integrates cleanly with metrics, and avoids a full reverse-graph pass. That fits best when I want precise counters and minimal overhead.

Kahn vs. DFS-postorder: Kahn's queue-based approach exposes push/pop counters and cycle detection directly, which is perfect for instrumentation and for explaining the behavior in the report.

DAG DP vs. Dijkstra/Bellman-Ford: After condensation, the graph is a DAG; DP on topological order is strictly simpler and linear, so it's both faster and easier to reason about.

8. Conclusions (practical recommendations)

Always compress cycles first (SCC) to plan with atomic blocks of mutually dependent tasks.Topologically schedule those blocks (Kahn works well and is transparent to debug).For planning metrics, compute from the DAG:

SSSP for time/cost from a service source;

Critical path for the overall makespan estimate.

This SCC→Topo→DP pipeline remains linear in the size of the condensation DAG and scales well for smart-city task graphs.

9. Reproducibility (clean clone)

Build & test
```
mvn clean test
```

Run
```
mvn -DskipTests exec:java
```
Outputs:

`results/results.csv` — summary table (metrics, time, sizes, paths)

`results/details/*.txt` — SCC component lists, condensation edges, topo/derived orders, and full distance vectors

JUnit tests live in `src/test/java`.

Appendix: Where the numbers come from

All rows and paths are taken directly from my program's `results/results.csv` produced on my machine. The three tables in this report are filtered summaries to keep the narrative concise and focused on what the rubric asks.