# Assignment 4 – Smart City Scheduling


## Pangerey Aiym

## SE-2422


### Algorithms Implemented:

SCC: Tarjan's algorithm

Condensation Graph: components compressed into a DAG. Parallel edges collapsed by min weight.

Topological Sort: Kahn's algorithm.

Shortest Paths on DAG: dynamic programming over topological order from a source component.

Critical Path :max-DP over topological order

# 1.Data Summary

| Graph ID | n (vertices) | m (edges) | Weight Model | Description |
|---|---|---|---|---|
| 1 | 8 | 9 | Edge weights | Small directed graph with several small SCCs and one larger cycle |
| 2 | 9 | 9 | Edge weights | Sparse DAG with no cycles, all vertices are singletons |
| 3 | 10 | 10 | Edge weights | Contains one tri-cycle component 1–2–3 |
| 4 | 12 | 14 | Edge weights | Fully acyclic structure |
| 5 | 16 | 21 | Edge weights | Moderately dense DAG |
| 6 | 18 | 21 | Edge weights | Similar density, larger graph size |
| 7 | 20 | 29 | Edge weights | Irregular structure with multiple branches |
| 8 | 25 | 36 | Edge weights | Large acyclic DAG |
| 9 | 40 | 46 | Edge weights | Extra-large DAG showcasing scalability |

Graphs 1–3: Small to medium graphs, some containing small cycles or tri-components.

Graphs 4–6: Fully acyclic DAGs with moderate density.

Graphs 7–9: Larger DAGs up to 40 nodes and 46 edges, demonstrating scalability of the algorithms.

Iplemented graphs by using EXCEL. All datasets used edge-weighted models to evaluate shortest and longest paths across components

CSV results:

| graph_id | n | m | scc_count | scc_components |
|---|---|---|---|---|
| 1 | 8 | 9 | 6 | (1) 7 ; (1) 6 ; (1) 5 ; (1) 4 ; (1) 3 ; (3) 0 1 2 |
| 2 | 9 | 9 | 9 | (1) 8 ; (1) 7 ; (1) 6 ; (1) 5 ; (1) 4 ; (1) 3 ; (1) 1 ; (1) 2 ; (1) 0 |
| 3 | 10 | 10 | 8 | (1) 9 ; (1) 8 ; (1) 7 ; (1) 6 ; (1) 5 ; (1) 4 ; (3) 1 2 3 ; (1) 0 |
| 4 | 12 | 14 | 12 | (1) 11 ; (1) 10 ; (1) 9 ; (1) 8 ; (1) 7 ; (1) 6 ; (1) 5 ; (1) 4 ; (1) 3 ; (1) 2 ; (1) 1 ; (1) 0 |
| 5 | 16 | 21 | 16 | (1) 15 ; (1) 14 ; (1) 13 ; (1) 12 ; (1) 11 ; (1) 10 ; (1) 9 ; (1) 8 ; (1) 7 ; (1) 6 ; (1) 5 ; (1) 4 ; (1) 3 ; (1) 2 ; (1) 1 ; (1) 0 |
| 6 | 18 | 21 | 18 | (1) 17 ; (1) 16 ; (1) 15 ; (1) 14 ; (1) 13 ; (1) 12 ; (1) 11 ; (1) 10 ; (1) 9 ; (1) 8 ; (1) 7 ; (1) 6 ; (1) 5 ; (1) 4 ; (1) 3 ; (1) 2 ; (1) 1 ; (1) 0 |
| 7 | 20 | 29 | 20 | (1) 17 ; (1) 18 ; (1) 13 ; (1) 19 ; (1) 14 ; (1) 9 ; (1) 15 ; (1) 10 ; (1) 5 ; (1) 1 ; (1) 16 ; (1) 11 ; (1) 6 ; (1) 2 ; (1) 7 ; (1) 3 ; (1) 12 ; (1) 8 ; (1) 4 ; (1) 0 |
| 8 | 25 | 36 | 25 | (1) 24 ; (1) 23 ; (1) 22 ; (1) 21 ; (1) 20 ; (1) 19 ; (1) 18 ; (1) 17 ; (1) 16 ; (1) 15 ; (1) 14 ; (1) 13 ; (1) 12 ; (1) 11 ; (1) 10 ; (1) 9 ; (1) 8 ; (1) 7 ; (1) 6 ; (1) 5 ; (1) 4 ; (1) 3 ; (1) 2 ; (1) 1 ; (1) 0 |
| 9 | 40 | 46 | 40 | (1) 39 ; (1) 38 ; (1) 37 ; (1) 36 ; (1) 35 ; (1) 34 ; (1) 33 ; (1) 32 ; (1) 31 ; (1) 30 ; (1) 29 ; (1) 28 ; (1) 27 ; (1) 26 ; (1) 25 ; (1) 24 ; (1) 23 ; (1) 22 ; (1) 21 ; (1) 20 ; (1) 19 ; (1) 18 ; (1) 17 ; (1) 16 ; (1) 15 ; (1) 14 ; (1) 13 ; (1) 12 ; (1) 11 ; (1) 10 ; (1 |

This table shows the results of the Tarjan SCC algorithm, which finds strongly connected components in each directed graph.Each row represents one graph from tasks.json. For example, in Graph 1 with 8 vertices and 9 edges, we have 6 SCCs:
(1) 7 ; (1) 6 ; (1) 5 ; (1) 4 ; (1) 3 ; (3) 0 1 2.That means vertices 0, 1, and 2 form a cycle they depend on each other, while others are single, independent nodes. This table shows the results of the Tarjan SCC algorithm, which finds strongly connected components in each directed graph.Each row represents one graph from tasks.json.

| topo_order | derived_order |
|---|---|
| [5, 4, 3, 2, 1, 0] | {0 1 2} {3} {4} {5} {6} {7} |
| [8, 6, 7, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} |
| [7, 6, 5, 4, 3, 2, 1, 0] | {0} {1 2 3} {4} {5} {6} {7} {8} {9} |
| [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} |
| [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} |
| [17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} |
| [19, 9, 13, 15, 18, 8, 12, 14, 17, 5, 7, 11, 16, 2, 4, 6, 10, 0, 1, 3] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} |
| [24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21} {22} {23} {24} |
| [39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21} {22} {23} {24} {25} {26} {27} {28} {29} {30} {31} {32} {33} {34} {35} {36} {37} {38} {39} |

This table shows the Topological Order and the Derived Order of each graph after SCC compression.The Topological Order lists components in the correct sequence, meaning every directed edge goes from an earlier component to a later one no backward edges.For example, in Graph 1, the topological order [5, 4, 3, 2, 1, 0] means component 5 has no incoming edges, while 0 comes last.The Derived Order expands each component back to its original nodes.If one SCC had multiple vertices (like {0 1 2}), they appear together inside braces.So {0 1 2} {3} {4} {5} {6} {7} means tasks 0–2 are connected, and others can be scheduled after them.

| longest_path_len | dfs_calls | edges_visited_dfs | queue_pushes | queue_pops | relaxations | time_ms | weight_model | source_comp | target_comp | dag_n | dag_m | remarks |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 8 | 9 | 6 | 6 | 5 | 0.102 | edge | 5 | 4 | 6 | 6 | ok |
| 20 | 9 | 9 | 9 | 9 | 8 | 0.103 | edge | 8 | 6 | 9 | 9 | ok |
| 19 | 10 | 10 | 8 | 8 | 7 | 0.078 | edge | 7 | 6 | 8 | 7 | ok |
| 26 | 12 | 14 | 12 | 12 | 12 | 0.098 | edge | 11 | 10 | 12 | 14 | ok |
| 44 | 16 | 21 | 16 | 16 | 18 | 0.131 | edge | 15 | 14 | 16 | 21 | ok |
| 45 | 18 | 21 | 18 | 18 | 19 | 0.765 | edge | 17 | 16 | 18 | 21 | ok |
| 16 | 20 | 29 | 20 | 20 | 21 | 0.162 | edge | 19 | 9 | 20 | 29 | ok |
| 48 | 25 | 36 | 25 | 25 | 24 | 0.231 | edge | 24 | 19 | 25 | 36 | ok |
| 96 | 40 | 46 | 40 | 40 | 39 | 0.367 | edge | 39 | 38 | 40 | 46 | ok |

This table shows the performance metrics and path results for each graph after running Tarjan SCC, topological sort, and shortest/longest path algorithms.The longest_path_len column represents the critical path length  the maximum total weight from the source to the last task, showing project duration or dependency depth.dfs_calls and edges_visited_dfs measure how many recursive steps Tarjan made higher values mean more traversal due to larger or denser graphsqueue_pushes and queue_pops show operations performed during Kahn’s topological

sorting, reflecting how many nodes were added or removed from the queue.relaxations refers to updates in the shortest-path algorithm more relaxations happen in denser graphs.The time_ms shows runtime efficiency: from 0.07 ms on small graphs up to 0.36 ms for the largest 40 vertices.All graphs use the edge weight model and completed successfully (ok), meaning the system scaled linearly with graph size.

| topo_order | derived_order | shortest_path | shortest_dist_to_target | s |
|---|---|---|---|---|
| [5, 4, 3, 2, 1, 0] | {0 1 2} {3} {4} {5} {6} {7} | {0 1 2}->{3} | 1 | [ |
| [8, 6, 7, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} | {0}->{1} | 1 | [ |
| [7, 6, 5, 4, 3, 2, 1, 0] | {0} {1 2 3} {4} {5} {6} {7} {8} {9} | {0}->{1 2 3} | 2 | [ |
| [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} | {0}->{1} | 1 | [ |
| [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} | {0}->{1} | 3 | [ |
| [17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} | {0}->{1} | 2 | [ |
| [19, 9, 13, 15, 18, 8, 12, 14, 17, 5, 7, 11, 16, 2, 4, 6, 10, 0, 1, 3] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} | {0}->{1} | 1 | [ |
| [24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21} {22} {23} {24} | {0}->{5} | 2 | [ |
| [39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] | {0} {1} {2} {3} {4} {5} {6} {7} {8} {9} {10} {11} {12} {13} {14} {15} {16} {17} {18} {19} {20} {21} {22} {23} {24} {25} {26} {27} {28} {29} {30} {31} {32} {33} {34} {35} {36} {37} {38} {39} | {0}->{1} | 1 | [ |

The topo_order column shows the order in which components SCCs are processed — earlier components have no incoming edges, ensuring dependency correctness.The derived_order expands those components into their original vertex groups.For example, {0 1 2} {3} {4} {5} {6} {7} means vertices 0–2 form a single connected group, followed by independent vertices.The shortest_path column identifies the minimal connection between components (e.g., {0}->{1} or 0 1 2 to 3, representing the most efficient dependency flow.shortest_dist_to_target gives the total weight of that minimal path (1–3 units in all cases).In practice, this table demonstrates how DAG-based scheduling identifies both a valid execution order and the fastest route through tasks  crucial for optimizing processing time or task scheduling in Smart City systems.

| all_distances_from_source | critical_path |
|---|---|
| [11, 7, 4, 2, 1, 0] | {0 1 2}->{3}->{4}->{5}->{6}->{7} |
| [16, 12, 11, 9, 4, 3, 1, 4, 0] | {0}->{2}->{3}->{4}->{5}->{6}->{7}->{8} |
| [19, 17, 16, 13, 8, 6, 2, 0] | {0}->{1 2 3}->{4}->{5}->{6}->{7}->{8}->{9} |
| [13, 11, 7, 5, 9, 15, 12, 8, 6, 4, 1, 0] | {0}->{1}->{2}->{3}->{4}->{5}->{6}->{7}->{8}->{9}->{10}->{11} |
| [19, 14, 10, 7, 5, 6, 4, 8, 20, 15, 11, 9, 6, 5, 3, 0] | {0}->{1}->{2}->{3}->{4}->{5}->{6}->{7}->{8}->{9}->{10}->{11}->{12}->{13}->{14}->{15} |
| [23, 20, 18, 17, 14, 10, 8, 5, 4, 7, 20, 17, 15, 10, 6, 3, 2, 0] | {0}->{1}->{2}->{3}->{4}->{5}->{6}->{7}->{8}->{9}->{10}->{11}->{12}->{13}->{14}->{15}->{16}->{17} |
| [11, 8, 7, 7, 6, 4, 6, 5, 3, 1, 10, 7, 5, 2, 7, 3, 9, 5, 4, 0] | {0}->{3}->{7}->{11}->{15}->{19} |
| [16, 14, 12, 10, 8, 14, 12, 10, 8, 6, 12, 10, 8, 6, 4, 10, 8, 6, 4, 2, 8, 6, 4, 2, 0] | {0}->{1}->{2}->{3}->{4}->{5}->{6}->{7}->{8}->{9}->{10}->{11}->{12}->{13}->{14}->{15}->{16}->{17}->{18}->{19}->{20}->{21}->{22}->{23}->{24} |
| [10, 25, 23, 22, 18, 19, 17, 16, 12, 9, 26, 25, 21, 18, 16, 16, 12, 9, 7, 6, 24, 21, 19, 18, 14, 12, 10, 9, 5, 2, 21, 20, 16, 13, 11, 10, 6, 3, 1, 0] | {0}->{1}->{2}->{3}->{4}->{5}->{6}->{7}->{8}->{9}->{10}->{11}->{12}->{13}->{14}->{15}->{16}->{17}->{18}->{19}->{20}->{21}->{22}->{23}->{24}->{25}->{26}->{27}->{28}->{29}->{30}->{31}->{32}->{33}->{34}->{35}->{36}->{37}-> |

This table presents the distance results and the critical (longest) paths for each graph.The column all_distances_from_source shows the computed shortest distance from the source vertex (or component) to every other vertex in the DAG. For example, in smaller graphs, distances like [11, 7, 4, 2, 1, 0] indicate how far each node is from the source smaller values show closer tasks in the dependency chain.The critical_path column shows the longest route across the graph — the sequence of tasks that takes the most total time to complete.For instance, {0}->{1}->{2}->{3}->{4}->{5}->{6}->{7} means every task depends on the completion of the previous one, forming a linear chain.In practice, identifying this longest path is crucial for project scheduling and performance analysis, as it highlights potential bottlenecks or delays in the overall system flow.

In csv:

SCC: scc_count, scc_components + condensation build (dag_n, dag_m).

Topological sort: topo_order and derived_order.

Shortest/Longest paths: Chosen weight model documented (weight_model=edge).

SSSP: source_comp, all_distances_from_source, shortest_path, shortest_dist_to_target.

Critical path: critical_path, longest_path_len.

Metrics: dfs_calls, edges_visited_dfs, queue_, relaxations, time_ms.

Validity: remarks.

**Complexity:**

Tarjan SCC: O(n + m)

Condensation build: O(n + m)

Kahn topo: O(dag_n + dag_m)

SSSP in DAG: O(dag_n + dag_m)

Critical path: O(dag_n + dag_m)

# 2. Results Overview

| Metric | Observation |
|---|---|
| Average SCC count | Most graphs were fully acyclic; only small graphs 1 and 3 had non-trivial cycles |
| Topo sort validity | All 9 graphs produced valid topological orders like size = SCC count |
| Shortest path cost | Ranged from 1 to 3 for typical cases; reflects minimal edge weights |
| Longest path length | Grows linearly with graph size |
| Runtime (ms) | 0.07 – 0.37 ms for large graphs; small graphs under 0.1 ms |

In this table we can see average css count, topological sort validity and so on. For instance runtime through my tables were 0.07 to 0.37 in 9th graph.

# 3. Detailed Analysis

### 3.1 Tarjan's SCC

Bottleneck: recursive DFS stack depth and edge scans on cyclic subgraphs.

Structure effect: in denser graphs Graphs 5–6 back-edges form multi-node SCCs, in sparse DAGs Graphs 2, 4, 8–9 scc_count =n

Evidence: Graph 5 dfs_calls=44 and Graph 6 dfs_calls=45 vs Graph 2 dfs_calls=20)

### 3.2 Kahn's Topological Sort

Bottleneck: queue operations scale with in-degree after SCC compression.

Structure effect: many small SCCs near-linear queue traffic, higher dag_m increases pushes/pops.

Evidence: Graph 5 queue_pushes=21 vs Graph 2 queue_pushes=9 , both validated as ok.

### 3.3 DAG Shortest Paths

Bottleneck: number of relaxations dag_m.

Structure effect: sparse condensation Graphs 8–9 have fewer relaxations and fast completion less than 0.37 ms.

Evidence: Graph 5 relaxations=18, time_ms=0.131 vs Graph 6 relaxations=19, time_ms=0.765.

### 3.4 Critical Path or Longest Path

Computed via max-DP over topo_order from source_comp.

Paths are reported as groups of original nodes per component,lengths align with the aggregated edge weights.

4. Structural Observations

| Property | Low Density DAGs | Higher Density / Cyclic Graphs |
|---|---|---|
| SCC Count | High ,1 per node | Fewer, larger SCCs 2–3 nodes |
| DFS Calls (Tarjan) | Linear with n | Increases slightly with cycles |
| Queue Ops (Topo) | Minimal | Higher with dense dependencies |
| Relaxations (DAG-SP) | < 20 | Up to 40 for denser graphs |
| Runtime | 0.07–0.37 ms | Slightly higher for dense graphs |

density=m/n(n−1)

| Graph | n | m | density | scc_count (=dag_n) | dag_m | weight_model |
|---|---|---|---|---|---|---|
| 1 | 8 | 9 | 0.161 | 6 | 6 | edge |
| 2 | 9 | 9 | 0.125 | 9 | 9 | edge |
| 3 | 10 | 10 | 0.111 | 8 | 7 | edge |
| 4 | 12 | 14 | 0.106 | 12 | 14 | edge |
| 5 | 16 | 21 | 0.088 | 16 | 21 | edge |
| 6 | 18 | 21 | 0.065 | 18 | 21 | edge |
| 7 | 20 | 29 | 0.076 | 20 | 29 | edge |
| 8 | 25 | 36 | 0.060 | 25 | 36 | edge |
| 9 | 40 | 46 | 0.029 | 40 | 46 | edge |

Summary: Tarjan's SCC handles cycles; DFS recursion is its main bottleneck. Kahn's algorithm scales linearly after SCC compression, bottlenecked by queue ops. DAG Shortest Paths depends on edge density; more edges = more relaxations. Structural density and SCC sizes directly affect all three phases denser graphs yield fewer SCCs but more work per edge.

# 4. Performance Insights

| Graph | n | m | Time (ms) | Remarks |
|---|---|---|---|---|
| 1 | 8 | 9 | 0.102 | Higher due to SCC compression and reconstruction overhead |
| 2–4 | 9–12 | 9–14 | 0.07 – 0.10 | Pure DAGs, nearly instant processing |
| 5–7 | 16–20 | 21–29 | 0.13 – 0.16 | Larger but stable scaling |
| 8–9 | 25–40 | 36–46 | 0.23 – 0.37 | Linear growth with size (O(V + E)) |

In this table we can see brief description why timings are high and so on. For instance in first graph with n=8 and m=9 time is 0.102, due to scc compression and reconstruction overhead.Time of graphs 2-4 with lengths 9-12 is from 0.07-0.10.

# 5. Conclusions & Recommendations

| Algorithm | Use When | Notes |
|---|---|---|
| Tarjan SCC | Graphs with possible cycles | Efficient O(V + E); enables safe compression to DAG before scheduling |
| Kahn Topo | Pure DAG tasks | Simple queue-based logic; ideal for real-time task ordering |
| DAG Shortest Path | When weights represent time or cost | Linear complexity; best choice for Smart City task scheduling |
| Longest Path (critical | Project timing and dependency analysis | Highlights bottleneck tasks in scheduling |

**Practical Recommendation:**

Use Tarjan + Kahn as a pre-processing pipeline for any real-world workflow with dependencies.

Apply DAG-SP to minimize total cost or duration after cycles are removed.

For dense graphs, use adjacency lists to keep time = O(V + E).

Use Metrics to monitor scaling in future city-scale datasets.

## Summary

All algorithms performed correctly and efficiently.No invalid orders or cycles remained after SCC compression.Time complexity remained linear, confirming the approach is feasible for large real-world task graphs.