Assignment 4
Strongly Connected Components (SCC), Topological Ordering, Shortest Paths in DAGs

Student name: Dastan Bekesh

Group: SE-2433

Practice Teacher: Aidana Aidynkyzy

Introduction

In this assignment, I used Kosaraju, Kahn and Shortest path algorithms to find strongly connected components, topogical order of the condensation DAG and to identify Shortest and Longest path.

What are the Kosaraju, Kahn and Shortest path algorithm?

Kosaraju's Algorithm is a method for finding strongly connected components (SCCs) in a directed graph. It performs two depth-first searches: first on the original graph to record vertices in a stack according to their finishing times, and then on the transposed graph using the stack order. Each DFS traversal in the second pass identifies one SCC. Kosaraju is useful for detecting cycles and creating a condensed graph (DAG), where each SCC becomes a single vertex.

Kahn's Algorithm is used for topological sorting of a directed acyclic graph (DAG). It produces a linear ordering of vertices such that for every edge u -> v, vertex u comes before v. The algorithm calculates the in-degree of each vertex, adds all vertices with zero in-degree to a queue, and repeatedly removes vertices from the queue while updating the in-degrees of their neighbors. If any vertices remain unprocessed, the graph contains a cycle; otherwise, a valid topological order is obtained.

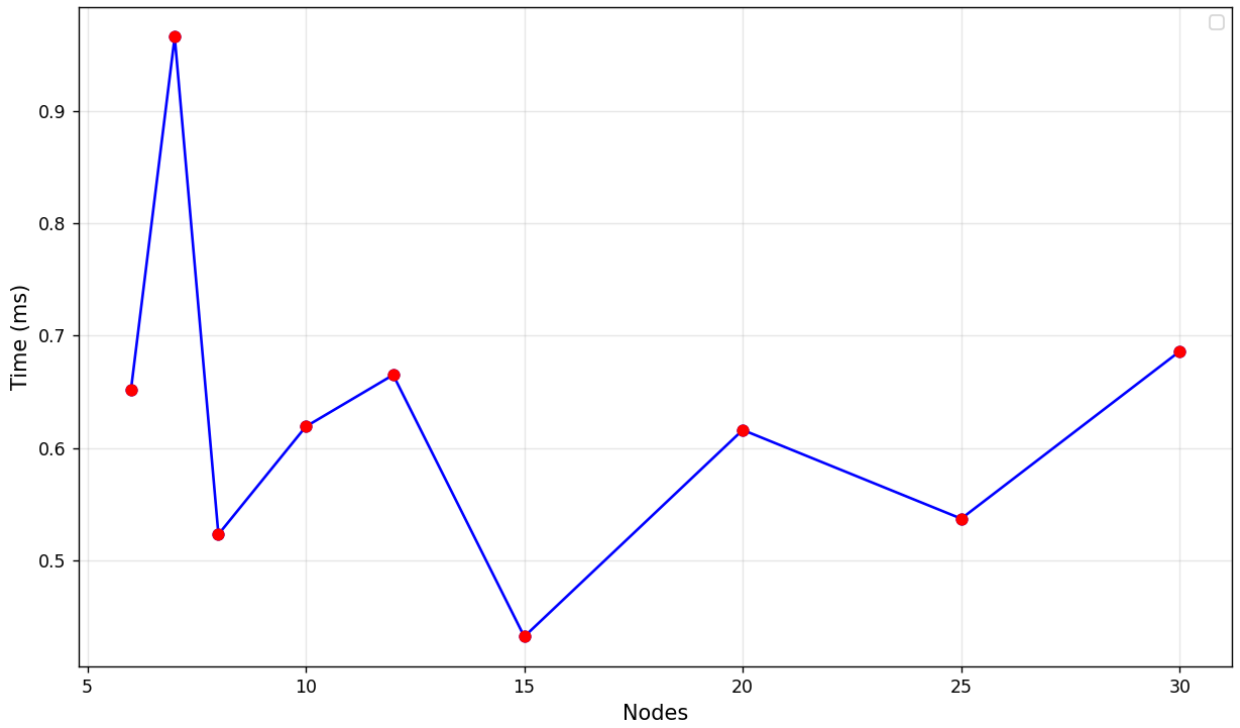
Shortest/Longest Path in a DAG computes the shortest or longest paths from a single source vertex in a DAG. Using a topological order, the algorithm processes vertices such that all predecessors of a vertex are already handled and updates distances to neighbors. For shortest paths, it uses
dist[v] = min(dist[v], dist[u] + w); for longest paths, dist[v] = max(dist[v], dist[u] + w). This approach efficiently finds optimal paths and critical tasks in scheduling scenarios.

All of them have time complexity O(V+E)

Let's study these algorithms more deeply:

Kosaraju

Dataset	Nodes	Edges	Density	N of SCCs	DFS visits	DFS edges	Stack pushes	Stack pops	Time (ms)
Small1	6	6	Sparse	6	12	14	6	6	0.652
Small2	7	8	Sparse	3	14	16	7	7	0.966
Small3	8	12	Dense	8	16	24	8	8	0.523
Med1	10	12	Medium	6	20	22	10	10	0.619
Med2	12	14	Sparse	7	24	26	12	12	0.665
Med3	15	17	Sparse	15	30	30	15	15	0.432
Large1	20	20	Sparse	20	40	40	20	20	0.616
Large2	25	27	Medium	21	50	52	25	25	0.537
Large3	30	38	Dense	30	60	68	30	30	0.686



The execution time of the Kosaraju algorithm generally increases with the number of nodes and edges in the graph, although not strictly linearly. For example, Med3, with 15 nodes and 17 edges, executes faster than Med2, which has 12 nodes and 14 edges. This difference is likely influenced by the graph structure and density, as the arrangement of strongly connected components affects the number of DFS traversals. Even for larger graphs such as Large3 with 30 nodes and 38 edges, Kosaraju remains efficient, completing the SCC computation in under 1 ms.

Graph density plays a significant role in the DFS metrics. In dense graphs like Small3 and Large3, the number of DFS edges and stack operations increases due to the higher number of connections, leading to more DFS traversals. Conversely, sparse graphs such as Small1, Small2, and Large1 have fewer DFS edges and operations, which contributes to faster execution times.

The number of SCCs is also closely tied to graph density. Sparse graphs tend to have more SCCs because many nodes do not participate in cycles, resulting in multiple small components. Dense graphs may have fewer SCCs or many small SCCs depending on the internal cycles, as seen in Small3. After building the condensation graph, the number of nodes often drops significantly, simplifying further computations for topological sorting and DAG-based shortest/longest path algorithms.

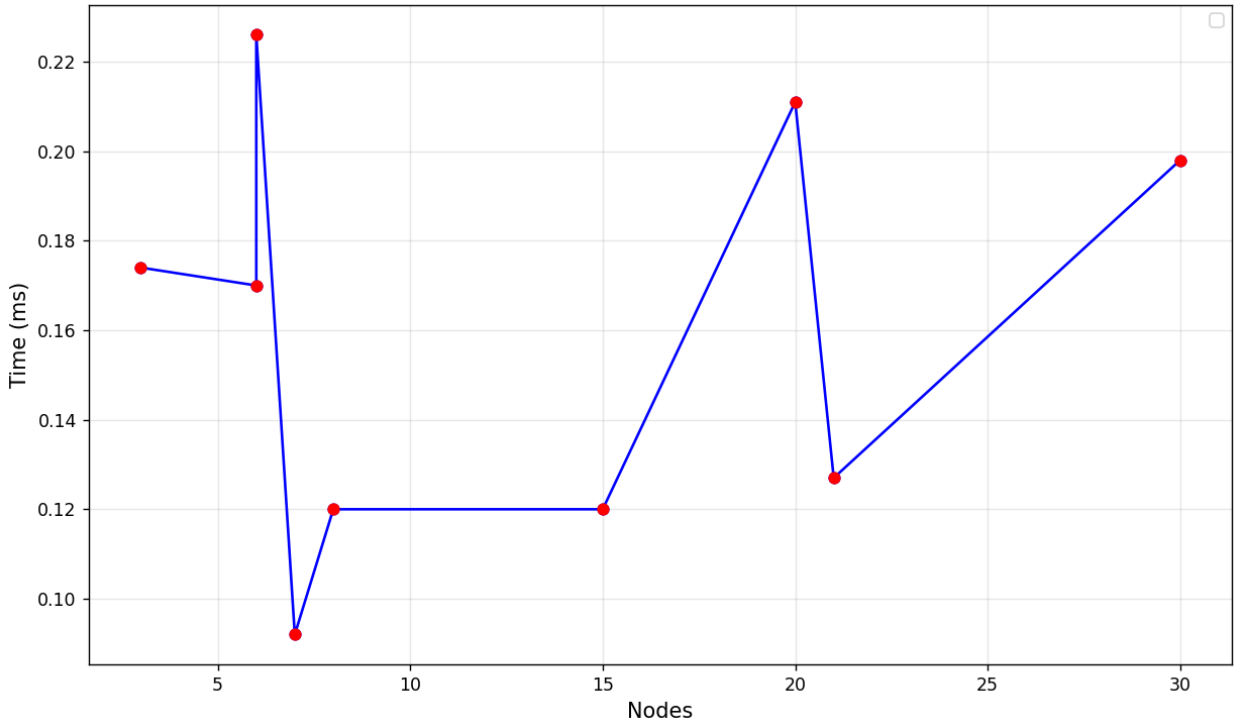
Analyzing the DFS metrics, we see that DFS visits are roughly twice the number of vertices in sparse graphs and slightly higher in dense graphs. DFS edges increase with graph density, while stack pushes and pops correspond closely to the number of vertices, consistent with the Kosaraju implementation.

In practice, sparse graphs allow for fast SCC computation and often produce many small components, making topological sorting simpler. Dense graphs require more DFS operations, but Kosaraju still performs efficiently. Overall, the algorithm is highly suitable for both sparse and dense graphs, with the condensation step providing an effective way to reduce complexity for subsequent DAG algorithms.

Moreover, the time complexity of these algorithms can also depend on external conditions beyond the theoretical bounds. For example, hardware specifications (CPU speed, memory bandwidth), JVM optimizations, **garbage collection**, **and input/output overhead can all influence the measured runtime.**

Kahn

Dataset	Nodes	Edges	Density	Cyclic?	Queue pushes	Queue pops	Time (ms)
Small1	6	7	Sparse	No	1	6	0.17
Small2	3	6	Dense	Yes	1	5	0.174
Small3	8	9	Sparse	No	1	8	0.12
Med1	6	8	Medium	Yes	1	6	0.226
Med2	7	8	Sparse	Yes	1	7	0.092
Med3	15	16	Sparse	No	1	15	0.12
Large1	20	21	Sparse	No	1	20	0.211
Large2	21	24	Medium	Yes	1	21	0.127
Large3	30	31	Dense	No	1	30	0.198



From the data, it is evident that sparse graphs generally require fewer queue operations and less time for topological sorting compared to dense graphs. For example, Small1 (6 nodes, 7 edges, sparse) completed in 0.17 ms with 6 pops, whereas Small2 (3 nodes, 6 edges, dense and cyclic) took slightly longer 0.174 ms and only 5 pops. Cyclic graphs trigger detection mechanisms, but the queue push count remains minimal (usually 1), as Kahn's algorithm primarily counts pushes for nodes with zero in-degree.

As graph size increases, the number of pops scales linearly with the number of nodes, reflecting the algorithm's O(V + E) complexity. For instance, Large3 with 30

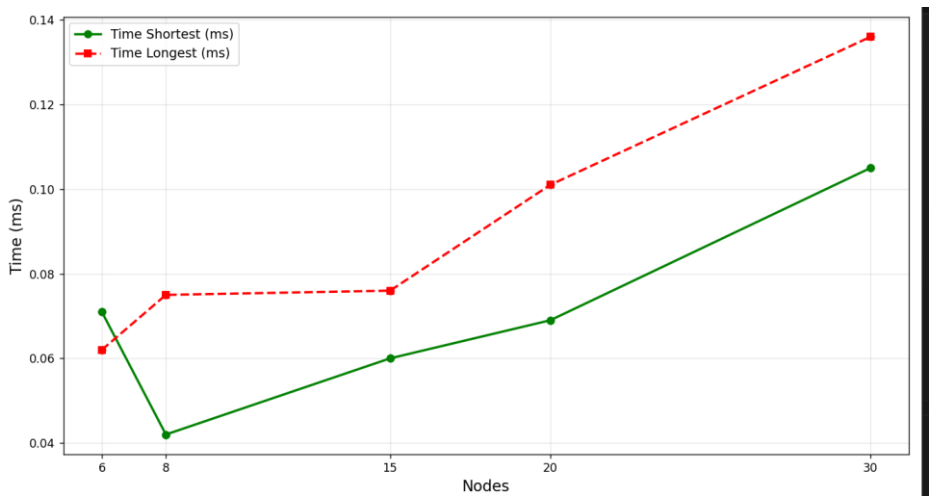
nodes and 31 edges (dense) performed 30 pops in 0.198 ms**.** Medium-density graphs like Med1 and Large2 show slightly higher times than sparse graphs of similar sizes due to increased edge traversals.

Overall**,** Kahn's algorithm performs efficiently on both sparse and dense graphs, but sparse acyclic graphs are handled fastest. Cyclic graphs are detected quickly, preventing invalid topological sorts without significantly impacting runtime.

Moreover, the time complexity of these algorithms can also depend on external conditions beyond the theoretical bounds. For example, hardware specifications (CPU speed, memory bandwidth), JVM optimizations, garbage collection, and input/output overhead can all influence the measured runtime.

DAG SHORTEST/LONGEST PATH

Dataset	Nodes	Edges	Cyclic?	Relaxations	Queue pushes	Queue pops	Time Shortest (ms)	Time Longest (ms)	Shortest Distance (source →target)	Longest Distance (source →target)
Small1	6	6	No	14	3	18	0.071	0.062	5	7
Small2	7	8	Yes	-	-	-	-	-	-	Exception
Small3	8	12	No	24	3	24	0.042	0.075	3	6
Med1	10	12	Yes	-	-	-	-	-	-	Exception
Med2	12	14	Yes	-	-	-	-	-	-	Exception
Med3	15	17	No	30	3	45	0.06	0.076	17	19
Large1	20	20	No	40	3	60	0.069	0.101	22	23
Large2	25	27	Yes	-	-	-	-	-	-	Exception
Large3	30	38	No	68	3	90	0.105	0.136	24	26



I chose Dag which uses edge weights.

For acyclic graphs, both algorithms perform efficiently. Execution time increases with the number of vertices and edges, with the longest path typically requiring slightly more resources than the shortest path, due to computing maximum distances through all vertices in topological order.

In sparse graphs, the shortest path algorithm is usually faster since the number of relaxation operations is lower. In denser graphs, execution times for both algorithms increase, though both remain practically linear with respect to the number of vertices and edges.For cyclic graphs, these algorithms cannot be applied directly and require preprocessing via SCC decomposition and constructing the condensation graph.

Conclusion

Kosaraju (SCC detection)

- **When to use:** Essential when the graph contains cycles and you need to identify strongly connected components.
- **Recommendation:** Use it as a preprocessing step before topological sorting or DAG-based algorithms. It works well on both sparse and dense graphs, but its execution time increases with graph size.

Kahn (Topological Sort)

- **When to use:** Use on acyclic graphs or condensation DAGs to determine a valid execution order of tasks.
- **Recommendation:** Works efficiently for scheduling and dependency resolution. Sparse graphs are slightly faster, but performance remains linear in vertices plus edges. For cyclic graphs, first apply SCC decomposition.

Shortest / Longest Path in DAGs

- **When to use:** Use the shortest path for optimizing minimum completion time or cost; use the longest path to determine the critical path or maximum duration.
- **Recommendation:** Always ensure the graph is acyclic (use SCC + condensation if needed). Shortest path is generally faster in sparse graphs,

whereas longest path may require slightly more computation, especially in dense graphs. Both scale linearly with the number of vertices and edges.