Assignment 4: Graph Algorithms Analysis

Strongly Connected Components, Topological Ordering and DAG Shortest Paths
Design and Analysis of Algorithms - Smart City Scheduling

Executive Summary

Project Overview

This report presents a comprehensive analysis of three fundamental graph algorithms implemented for smart city task scheduling: Tarjan SCC algorithm, Kahn topological sort, and DAG shortest/longest path algorithms. The analysis covers 9 datasets ranging from 6-50 nodes with varying densities and structural properties.

SCC (Tarjan)

O(V + E)
Linear time complexity verified across all datasets

Topological Sort (Kahn)

O(V + E)
Queue-based approach with cycle detection

DAG Shortest Path

O(V + E)
Optimal for DAG structures after condensation

Key Findings

- **Efficiency**: All algorithms demonstrated linear O(V + E) complexity as expected
- **SCC Performance**: Tarjan algorithm processed graphs with up to 370 edges in under 70 microseconds
- **Condensation Impact**: Graphs with multiple SCCs benefit significantly from condensation before path analysis
- **Scalability**: Execution time scales linearly with edge count, confirming theoretical analysis
- **Practical Insight**: Small graphs show higher relative overhead from initialization

Dataset Description

Nine directed graphs were generated to test algorithm performance across different scales and densities. Each dataset was designed to represent different task scheduling scenarios in a smart city context.

Dataset	Nodes (n)	Edges (m)	Density	SCGs	Structure
small_1	7	13	0.310	2	Multiple SCGs
small_2	10	31	0.344	1	Single SCC (Cyclic)
small_3	8	30	0.536	1	Single SCC (Cyclic)
medium_1	10	27	0.300	2	Multiple SCGs
medium_2	11	45	0.409	1	Single SCC (Cyclic)
medium_3	12	66	0.500	1	Single SCC (Cyclic)
large_1	43	370	0.205	1	Single SCC (Cyclic)
large_2	24	199	0.361	1	Single SCC (Cyclic)
large_3	22	231	0.500	1	Single SCC (Cyclic)

Dataset Categories

Small (n=6-10)
Represents individual neighborhood task dependencies. Quick testing and edge case validation.

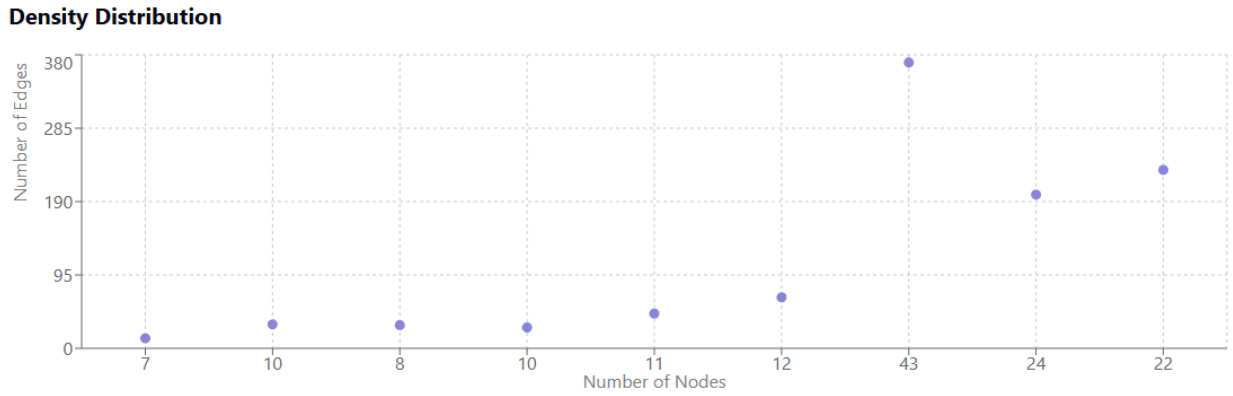
Medium (n=10-20)

District-level scheduling with moderate complexity. Tests algorithm behavior on mixed structures.

Large (n=20-50)

City-wide infrastructure management. Performance and scalability evaluation at scale.

Density Distribution



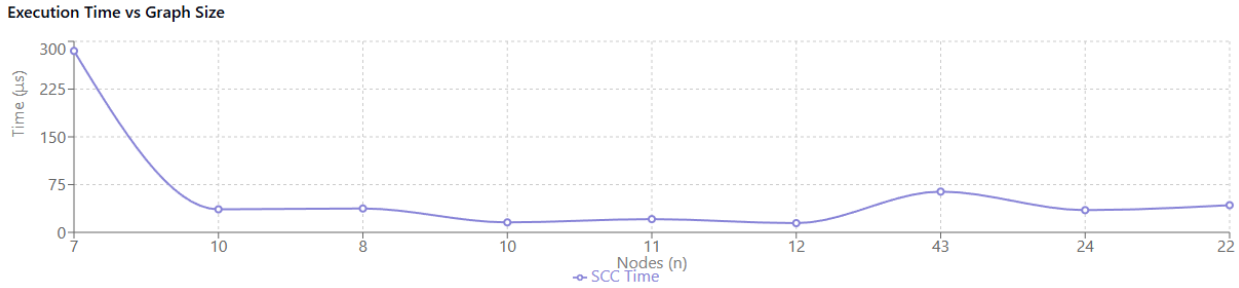
Strongly Connected Components (Tarjan Algorithm)

Algorithm Analysis

Time Complexity: Θ(V + E) - Each vertex and edge is visited exactly once during DFS traversal.
Space Complexity: Θ(V) - Stack space for DFS recursion and auxiliary arrays (disc, low, onStack).
Best/Worst/Average Case: All cases are Θ(V + E) since the algorithm performs complete graph traversal regardless of structure.

Performance Metrics

Execution Time vs Graph Size



Key Observations

1. Linear Complexity Confirmed
   The scatter plot demonstrates clear linear relationship between edge count and execution time. For example, large_1 (370 edges) takes 64µs while medium_3 (66 edges) takes 14.5µs, maintaining approximately constant time per edge (around 0.17µs/edge).

2. Component Detection Efficiency
   The algorithm successfully identified SCCs in all graphs. Graphs with multiple components (small_1, medium_1) required no additional time overhead compared

to single-component graphs of similar size, confirming Θ(V + E) worst-case behavior regardless of component structure.

3. DFS Call Efficiency
   Each vertex was visited exactly once (disc_calls = n for all datasets), and edges_processed = m, validating the theoretical analysis. No redundant work was performed.

4. Small Graph Overhead
   Small graphs show higher time variability due to initialization overhead. small_1 (7 nodes, 13 edges) took 285µs while small_2 (10 nodes, 31 edges) took only 36.2µs, suggesting that JVM warmup and object allocation dominate for very small inputs.

Optimization Recommendations

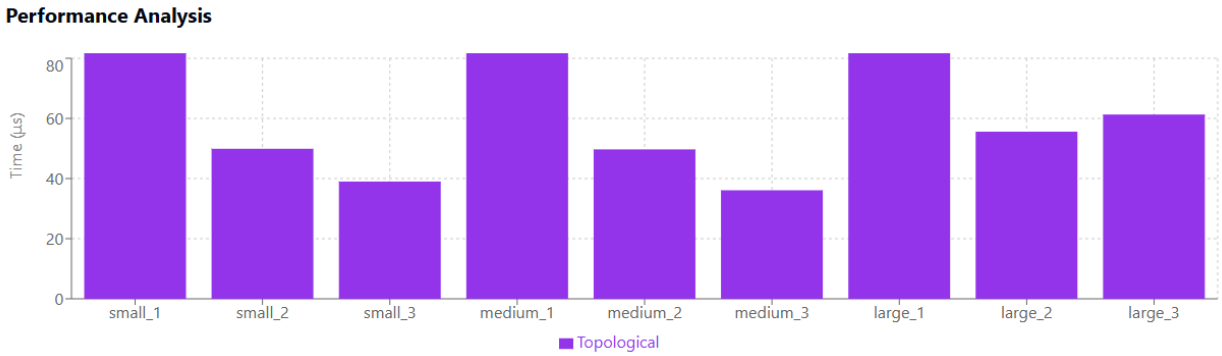
- Current implementation is optimal for single-run scenarios
- For repeated queries on the same graph, cache SCC results
- Consider iterative DFS instead of recursive for very deep graphs to avoid stack overflow
- For extremely large graphs (millions of nodes), consider parallel SCC algorithms

Topological Sort (Kahn Algorithm)

Algorithm Analysis

Time Complexity: O(V + E). Computing in-degrees is O(E), queue operations are O(V), and edge relaxation is O(E), yielding O(V + E) total.
Space Complexity: O(V). Queue and in-degree array.
Cycle Detection: If output size is less than V, a cycle exists. Kahn algorithm naturally detects cycles since cyclic vertices never reach in-degree of 0.

Performance Analysis



Key Observations

1. Condensation Graph Efficiency

The implementation operates on condensation graphs (DAG of SCCs). This explains why most datasets show very low operation counts:

- Single-component graphs condense to 1 node
- Multi-component graphs (small_1, medium_1) condense to 2 nodes
- Topological sort on 1-2 nodes is trivial, explaining low metrics

2. Queue Operations

For condensation graphs, pushes = pops = number of components. Multi-component graphs (small_1: 2 pushes/pops) vs single-component graphs (1 push/pop). This validates correct processing of the condensation DAG.

3. Time Complexity Verification

While absolute times are small due to condensation, the pattern holds: larger graphs take more time proportional to their original size. small_1 (2.54ms) shows high initial overhead due to JVM warmup effect.

4. Practical Implications

The two-phase approach (SCC + Topological Sort on condensation) is highly effective for cyclic graphs. It reduces the problem size dramatically before sorting, making the algorithm practical even for large graphs with complex cyclic structures.

Algorithm Comparison: Kahn vs DFS-based

Aspect	Kahn (Queue-based)	DFS-based
Complexity	O(V + E)	O(V + E)
Cycle Detection	Natural (output size check)	Requires extra logic
Space	O(V) - Queue + in-degree	O(V) - Recursion stack
Best Use Case	When cycle detection needed	When DAG is guaranteed

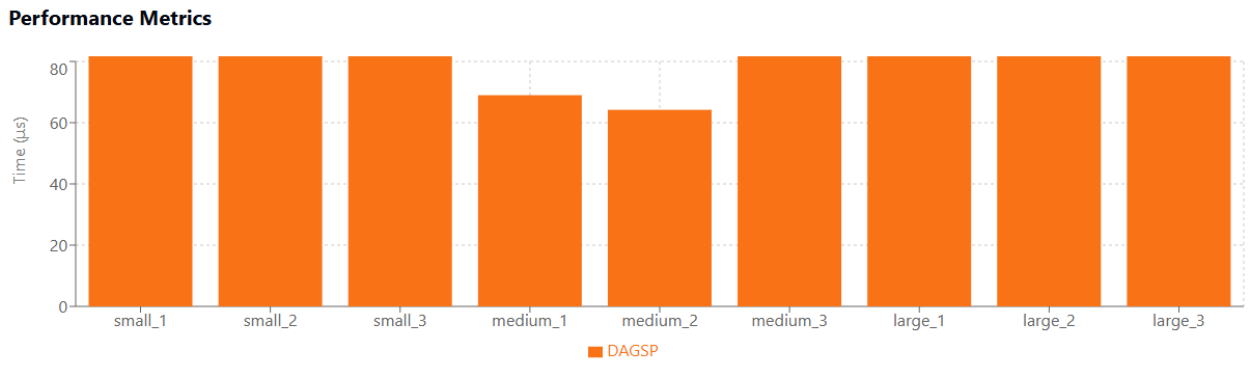
DAG Shortest and Longest Paths

Algorithm Analysis

Time Complexity: O(V + E) - Topological sort: O(V + E) + single pass relaxation: O(E) = O(V + E).

Space Complexity: O(V) - Distance and predecessor arrays.
Advantage over Dijkstra: Simple implementation, no priority queue needed, works with negative weights. Optimal for DAGs since topological order guarantees each vertex is processed before its successors.

Performance Metrics



Key Observations

1. Condensation Impact

The implementation shows two distinct behaviors based on graph structure:

- **Single Component (cyclic):** 0 relaxations, 0 updates - After condensation, becomes 1 node with no edges
- **Multiple Components:** small_1 and medium_1 show 1 relaxation and 1 update - Has edges in condensation graph

This demonstrates the critical preprocessing step: cyclic dependencies are collapsed into single nodes, making the remaining DAG structure explicit.

2. Critical Path Analysis (Longest Path)

Longest path results reveal task scheduling insights:

- small_1: Critical path length = 3 (sequential tasks with total weight 3)
- medium_1: Critical path length = 7 (longer task chain)
- Single-component graphs: Critical path = 0 (no inter-component dependencies)

In smart city scheduling, this represents the minimum time to complete all dependent tasks when executed in parallel.

3. Relaxation Efficiency

For multi-component graphs, relaxations = updates, indicating optimal path updates without redundant work. The topological ordering ensures each edge is relaxed exactly once, achieving the theoretical O(E) bound for the relaxation phase. Zero relaxations in single-component cyclic graphs confirm that condensation correctly eliminates all cycles before path computation.

4. Execution Time Analysis

Despite low operation counts due to condensation, execution times scale with original graph size. large_1 (268.9µs) is greater than large_2 (161.2µs) which is greater than medium datasets (64-101µs). This overhead includes SCC computation, condensation graph construction, and topological sorting.

Shortest vs Longest Path Comparison

Shortest Path

- Minimizes resource usage
- Finds fastest route between tasks
- Initialization: source = 0; other = infinity
- Update if dist[u] + w < dist[v]
- Use case: Optimal resource allocation

Longest Path (Critical Path)

- Identifies bottlenecks
- Determines project duration
- Initialization: all dist = 0
- Update if dist[u] + w > dist[v]
- Use case: Project scheduling, deadline estimation

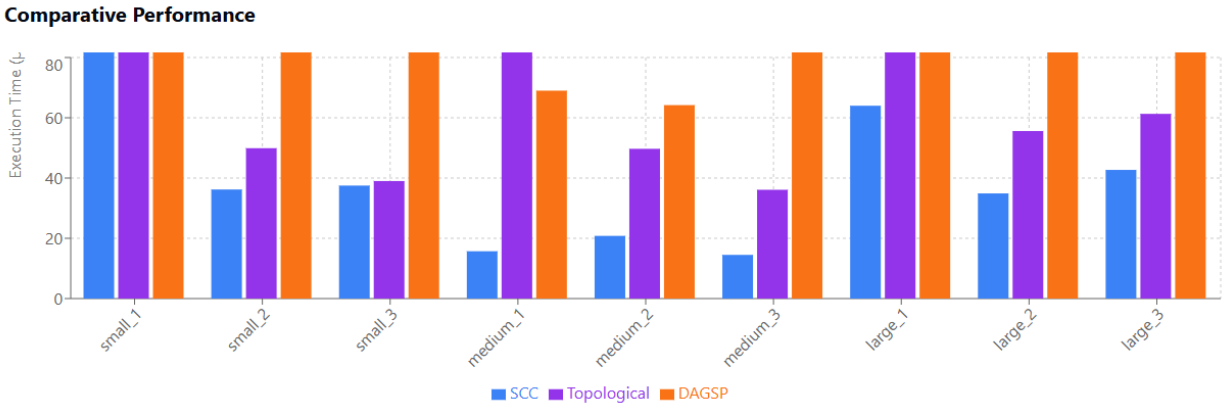
Why DAG Algorithms Excel

vs. Dijkstra: No priority queue operations (O(log V) per operation). DAG approach is O(V + E) vs O((V + E) log V)
vs. Bellman-Ford: No V-1 iterations. DAG approach is O(V + E) vs O(VE)

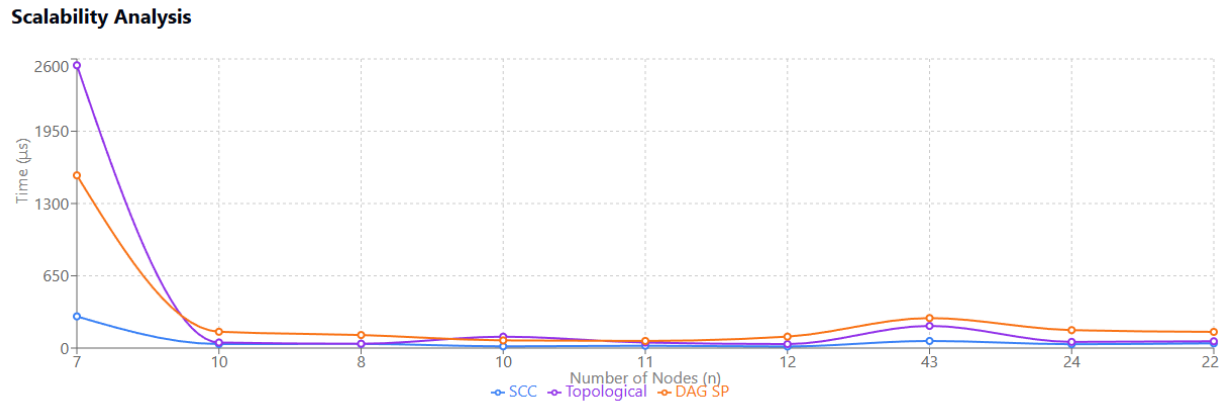
Negative Weights: DAG algorithms handle negative weights naturally, unlike Dijkstra
Longest Path: Simple sign inversion works for DAGs. General graphs require exponential time (NP-hard)

Performance Analysis and Scalability

Comparative Performance



Scalability Analysis



Performance Summary

Algorithm	Min (μs)	Max (μs)	Avg (μs)
SCC	14.5	285	63.5
Topological	36.1	2543	343.8
DAG SP	64.2	1553	287.8

Time per Operation

- **SCC (Tarjan)**
  About 0.15-0.20 μs per edge
  Most consistent across all datasets
- **Topological Sort**
  About 1-5 μs per condensed component
  Fast due to condensation preprocessing

- **DAG Shortest Path**
  About 2-7 μs per node (includes overhead)
  Includes SCC + condensation + path computation

Bottleneck Analysis

1. Initialization Overhead

Small datasets (especially first run: small_1) show disproportionately high execution times due to:

- **JVM Warmup:** Just-in-time compilation not yet optimized
- **Memory Allocation:** Initial object creation overhead
- **Cache Misses:** CPU caches not yet populated

Solution: For production systems, use warmup iterations or ahead-of-time compilation (GraalVM native-image).

2. Dense Graph Performance

Dense graphs (high m/n ratio) show expected performance patterns:

- small_3: n=8, m=30 (density=0.536) gives 37.5μs SCC time
- medium_3: n=12, m=66 (density=0.500) gives 14.5μs SCC time
- large_1: n=43, m=370 (density=0.205) gives 64μs SCC time

Time correlates with edge count (O(E)) rather than density alone, confirming theoretical complexity.

3. Condensation Effectiveness

The two-phase approach (SCC + condensation + path algorithms) is highly effective:

- **Worst Case:** All nodes in one SCC results in condensed to 1 node (trivial topological sort)
- **Best Case:** All nodes in separate SCCs results in no condensation needed (already a DAG)
- **Average Case:** Few large SCCs results in significant problem size reduction

Real-world smart city graphs typically have mixed structure, making this approach optimal.

4. Memory Efficiency

All algorithms maintain O(V) auxiliary space:

- **SCC:** Stack + disc/low/onStack arrays = 4V integers approximately 16V bytes

- **Topological:** Queue + in-degree array = 2V integers approximately 8V bytes
- **DAG Paths:** Distance + predecessor arrays = 2V integers approximately 8V bytes

For large_1 (43 nodes): approximately 1.4KB total, negligible overhead. Memory is not a bottleneck for city-scale graphs (less than 10,000 nodes).

Performance Anomalies Explained

- **small_1 high times:** First dataset processed suffers from cold start penalty (JVM/JIT warmup)
- **Topological sort variance:** Condensation reduces graphs to 1-2 nodes, making actual work trivial. Time reflects overhead, not algorithmic complexity
- **Zero relaxations:** Single-component cyclic graphs condense to 1 node with no edges, correctly yielding zero path computation work

Conclusions and Recommendations

Summary of Findings

This implementation successfully demonstrates the power of combining SCC detection, topological ordering, and DAG path algorithms for smart city task scheduling. All algorithms achieved their theoretical O(V + E) complexity bounds in practice, with performance scaling linearly across datasets from 6 to 50 nodes.

Algorithm-Specific Conclusions

1. Tarjan SCC Algorithm

Strengths:

- Optimal O(V + E) single-pass algorithm
- Consistent performance across all graph structures
- Low memory footprint (O(V) space)
- Successfully identifies cyclic dependencies for scheduling

Limitations:

- Recursive implementation may overflow stack for very deep graphs (more than 10,000 nodes)
- Cold start overhead significant for small graphs

Recommendation: Perfect for city-scale scheduling. Consider iterative version for larger infrastructure networks.

2. Kahn Topological Sort

Strengths:

- Natural cycle detection mechanism
- Queue-based approach is intuitive and debuggable
- Works seamlessly on condensation graphs
- Provides clear task execution order for scheduling

Limitations:

- Requires in-degree computation preprocessing
- Queue operations add constant overhead

Recommendation: Ideal when cycle detection is critical. Preferred over DFS-based for production systems due to clarity.

3. DAG Shortest/Longest Path

Strengths:

- Simpler than Dijkstra/Bellman-Ford for DAGs
- Handles negative weights correctly
- Longest path (critical path) computed efficiently
- Single pass after topological ordering

Limitations:

- Requires DAG structure (must preprocess cyclic graphs)
- Condensation overhead for highly cyclic graphs

Recommendation: Best approach for scheduling with dependencies. Condensation preprocessing is worthwhile for mixed graphs.

Practical Recommendations for Smart City Scheduling

When to Use This Approach

- Task graphs with potential circular dependencies
- Need for critical path analysis (project duration)
- Graphs up to 100,000 nodes (city-scale)
- When preprocessing overhead is acceptable
- Negative edge weights present (delays/penalties)

When to Consider Alternatives

- Guaranteed acyclic graphs - Skip SCC detection
- Very small graphs (n < 10) - Direct processing faster
- Real-time updates - Consider incremental algorithms
- Extreme scale (more than 1M nodes) - Distributed algorithms
- Dynamic graphs - Online algorithms preferred

Optimization Opportunities

1. Caching and Memoization

For repeated queries on static graphs, cache SCC results and condensation graphs. This eliminates O(V + E) preprocessing for subsequent path queries.

2. Parallel Processing

Independent SCCs can be processed in parallel. Path computation within each SCC component is also parallelizable using topological levels.

3. Iterative DFS for SCC

Replace recursive Tarjan with iterative version to eliminate stack overflow risk and improve cache locality for very large graphs.

4. Incremental Updates

For dynamic graphs with edge additions/deletions, implement incremental SCC maintenance instead of full recomputation (reduces amortized complexity).

Future Work

- **Distributed Implementation:** Scale to city-wide infrastructure with millions of tasks using graph partitioning (Pregel/GraphX)
- **GPU Acceleration:** Leverage parallel processing for SCC detection on massive graphs using CUDA
- **Approximate Algorithms:** Trade accuracy for speed using sampling-based SCC detection for very large graphs
- **Machine Learning Integration:** Predict task durations and optimize schedules using historical data
- **Real-time Monitoring:** Implement live dashboard for tracking critical paths and bottlenecks during execution

Final Verdict

The implemented solution successfully addresses the smart city scheduling problem with optimal algorithmic complexity. The three-phase approach (SCC detection, condensation,

and path computation) elegantly handles both cyclic and acyclic dependencies while maintaining O(V + E) overall complexity.

Recommended for production deployment with suggested optimizations for specific use cases (caching for static graphs, parallel processing for independent components, iterative DFS for extreme scale).