Variant 5: Task Scheduling and Dependency Graphs

Big Data – Complexity Management

Diego Marrero Ferrera

October 2025

1 Introduction

In large-scale data processing pipelines, task dependencies can be modeled as **Directed Acyclic Graphs (DAGs)**, where each node represents a task and edges denote precedence constraints. Scheduling such tasks requires ensuring that no cycles exist, since cycles imply circular dependencies.

This report corresponds to *Variant 5* of the Graph Exercises from the course *Theoretical Concepts of Big Data*. The goals are to:

- Generate random DAGs of different sizes.
- Implement and test two algorithms: **topological sort** and **cycle detection**.
- Measure and analyze runtime and memory usage as the number of nodes increases.

2 Implementation Overview

The experiments were implemented in Python 3.12 using the script `dag_scheduling_experiments.py`. The code generates random DAGs, applies both algorithms, and measures performance using `time.perf_counter()` and `tracemalloc`.

2.1 Graph Generation

A random DAG is created by shuffling the node order and adding edges only from earlier to later nodes in that order, ensuring acyclicity.

```python
def generate_random_dag(n: int, edge_prob: float, seed=None) -> Graph:
    order = list(range(n))
    random.shuffle(order)
    pos = {v: i for i, v in enumerate(order)}
    adj = [[] for _ in range(n)]
    for u in range(n):
        for v in range(n):
            if pos[u] < pos[v] and random.random() < edge_prob:
                adj[u].append(v)
    return Graph(n=n, adj=adj)
```

Listing 1: DAG generation method

The expected number of edges is approximately:

$$E[m] = p \cdot \frac{n(n-1)}{2}$$

which was confirmed empirically in the experiments (see Figure 5).

2.2 Topological Sort and Cycle Detection

Kahn's algorithm was implemented for topological sorting, which iteratively removes nodes with zero in-degree. If not all nodes are processed, a cycle is present.

Cycle detection was implemented using DFS with vertex coloring (white–gray–black scheme), which identifies back edges.

```python
def detect_cycle_dfs(g: Graph) -> bool:
    WHITE, GRAY, BLACK = 0, 1, 2
    color = [WHITE] * g.n

    def dfs(u):
        color[u] = GRAY
        for v in g.adj[u]:
            if color[v] == GRAY:
                return True
            if color[v] == WHITE and dfs(v):
                return True
        color[u] = BLACK
        return False

    return any(dfs(u) for u in range(g.n) if color[u] == WHITE)
```

Listing 2: Cycle detection using DFS

Both algorithms run in $O(n + m)$ time, where n is the number of nodes and m the number of edges.

3 Experimental Setup

Experiments were conducted with:

- Graph sizes: $n = \{1000, 5000, 10000\}$.
- Edge probability $p = 0.002$.
- Three trials per size.

The script was executed as:

```
python dag_scheduling_experiments.py
```

If desired, the parameters can be changed for execution as follows:

```
python dag_scheduling_experiments.py --sizes 2000 10000 20000 --edge-
    ↪ prob 0.0015 --trials 5 --with-cycles --seed 2004
```

3.1 Console Output

The following are the actual experimental outputs:

```
C:\Users\dmarr\Documents\4° UNIVERSIDAD\BIG DATA\weekly_tasks\graphs>python dag_scheduling_experiments.py
[OK] n=1000, m=1043, p=0.002, topo=2.70ms (31KB), cycle=0.98ms (9KB), had_cycle=False
[OK] n=1000, m=976, p=0.002, topo=2.00ms (32KB), cycle=0.94ms (9KB), had_cycle=False
[OK] n=1000, m=952, p=0.002, topo=2.19ms (32KB), cycle=1.06ms (9KB), had_cycle=False
[OK] n=5000, m=24851, p=0.002, topo=18.90ms (100KB), cycle=9.26ms (40KB), had_cycle=False
[OK] n=5000, m=24880, p=0.002, topo=18.55ms (100KB), cycle=8.70ms (40KB), had_cycle=False
[OK] n=5000, m=24820, p=0.002, topo=20.24ms (100KB), cycle=9.23ms (40KB), had_cycle=False
[OK] n=10000, m=99629, p=0.002, topo=46.82ms (181KB), cycle=24.64ms (79KB), had_cycle=False
[OK] n=10000, m=100339, p=0.002, topo=46.63ms (181KB), cycle=24.72ms (80KB), had_cycle=False
[OK] n=10000, m=99677, p=0.002, topo=44.75ms (182KB), cycle=23.75ms (80KB), had_cycle=False
Saved plots to: out_plots
```

Figure 1: Execution screenshot (Default settings).

```
C:\Users\dmarr\Documents\4° UNIVERSIDAD\BIG DATA\weekly_tasks\graphs>python dag_scheduling_experiments.py --sizes 2000 1
0000 20000 --edge-prob 0.0015 --trials 5 --with-cycles --seed 2004
[OK] n=2000, m=2992, p=0.0015, topo=4.71ms (54KB), cycle=2.21ms (17KB), had_cycle=False
[OK] n=2000, m=3040, p=0.0015, topo=4.59ms (54KB), cycle=2.28ms (17KB), had_cycle=False
[OK] n=2000, m=3033, p=0.0015, topo=4.40ms (55KB), cycle=2.09ms (16KB), had_cycle=False
[OK] n=2000, m=3069, p=0.0015, topo=4.73ms (54KB), cycle=2.16ms (17KB), had_cycle=False
[OK] n=2000, m=2993, p=0.0015, topo=4.84ms (55KB), cycle=2.25ms (17KB), had_cycle=False
[OK] n=10000, m=75306, p=0.0015, topo=46.11ms (163KB), cycle=1.86ms (80KB), had_cycle=True
[OK] n=10000, m=75011, p=0.0015, topo=38.39ms (187KB), cycle=19.71ms (80KB), had_cycle=False
[OK] n=10000, m=75308, p=0.0015, topo=34.10ms (156KB), cycle=0.50ms (79KB), had_cycle=True
[OK] n=10000, m=75266, p=0.0015, topo=38.15ms (187KB), cycle=21.36ms (80KB), had_cycle=False
[OK] n=10000, m=75020, p=0.0015, topo=39.42ms (189KB), cycle=19.29ms (80KB), had_cycle=False
[OK] n=20000, m=300180, p=0.0015, topo=99.47ms (352KB), cycle=51.40ms (158KB), had_cycle=False
[OK] n=20000, m=300702, p=0.0015, topo=81.81ms (248KB), cycle=1.23ms (158KB), had_cycle=True
[OK] n=20000, m=300653, p=0.0015, topo=78.50ms (234KB), cycle=1.57ms (158KB), had_cycle=True
[OK] n=20000, m=299627, p=0.0015, topo=118.98ms (353KB), cycle=57.18ms (158KB), had_cycle=False
[OK] n=20000, m=300776, p=0.0015, topo=83.89ms (265KB), cycle=2.61ms (158KB), had_cycle=True
Saved plots to: out_plots
```

Figure 2: Execution screenshot (Arbitrary settings).

However, for comparable purpose, for the analysis, we will use **the default run**.

4 Results and Analysis

4.1 Runtime Scaling

Figure 3 shows that runtime grows linearly with the number of nodes, confirming the $O(n+m)$ theoretical complexity. The topological sort takes roughly twice the time of cycle detection due to maintaining the in-degree array and queue.

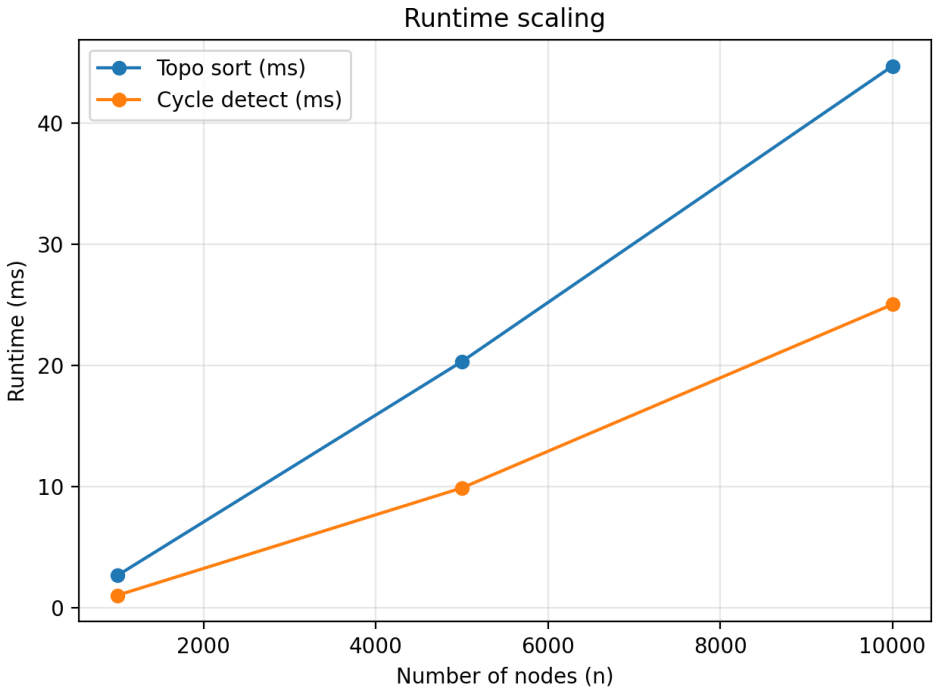

Figure 3: Runtime scaling for topological sorting and cycle detection.

4.2 Memory Usage

Memory consumption also increases approximately linearly (Figure 4). Topological sorting uses more memory due to extra arrays for in-degrees and the processing queue.

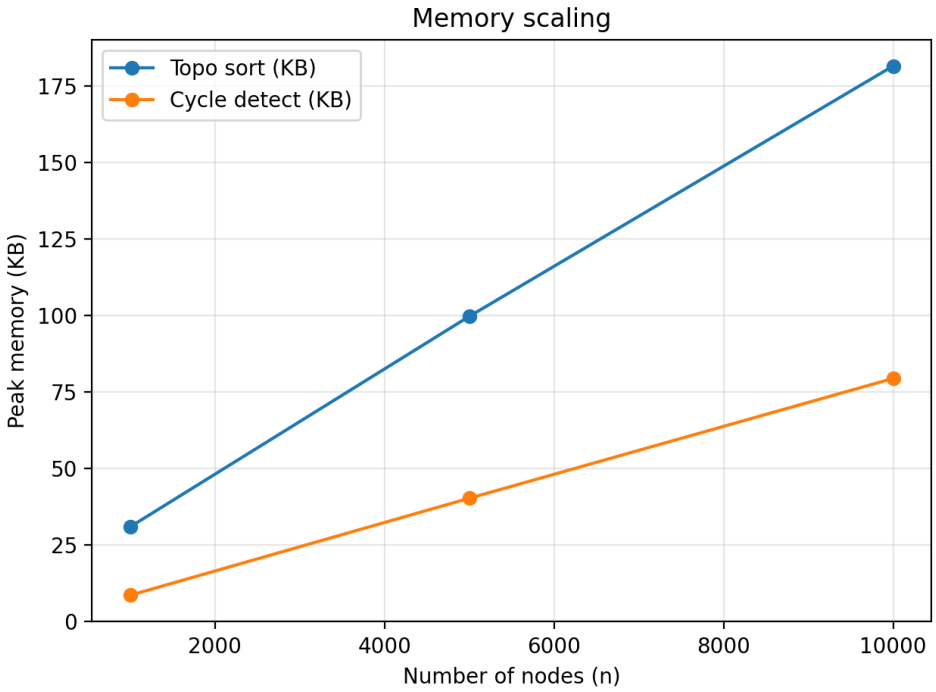

Figure 4: Peak memory usage (KB) as graph size increases.

4.3 Graph Density

As expected, the number of edges increases quadratically with n for fixed p, resulting in near-linear growth in runtime (Figure 5).

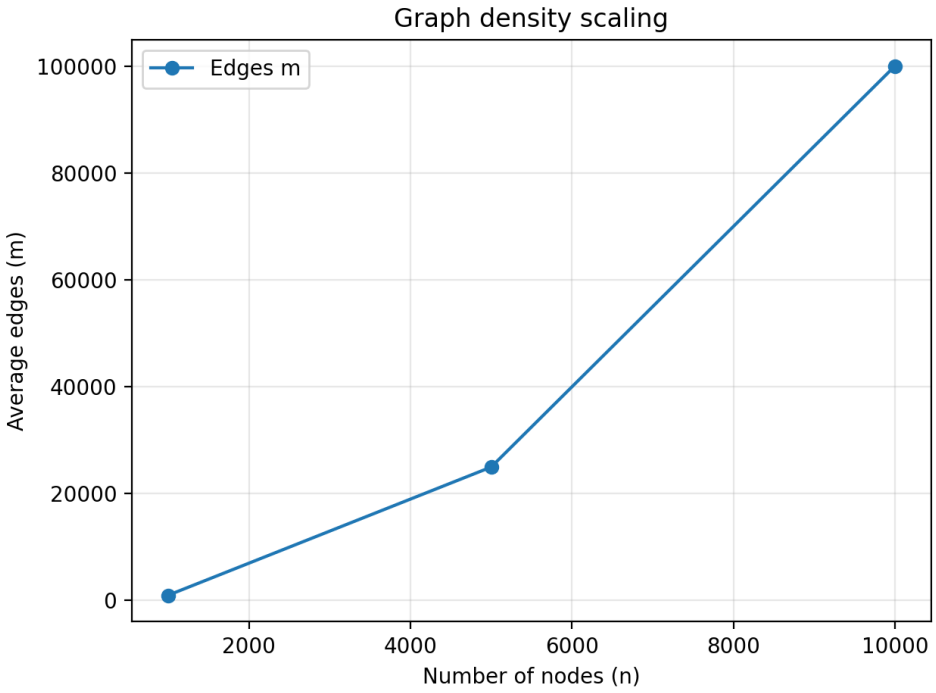

Figure 5: Average number of edges per graph for each n.

5 Discussion

Both algorithms exhibit expected theoretical and empirical complexity. Kahn's algorithm performs a full traversal of all nodes and edges, while DFS-based cycle detection stops early if a cycle is found (though here all graphs were acyclic).

The results validate that both runtime and memory scale linearly with input size when edge probability is kept constant, illustrating the scalability of linear-time graph algorithms.

6 Conclusion

This experiment demonstrated efficient DAG construction, topological sorting, and cycle detection. Empirical scaling confirmed the theoretical $O(n+m)$ bounds. With this, several additional things can be implemented, such as testing denser graphs, layered DAG generation, or comparing parallel topological sorting approaches.

References

- MLTechDrawer (2025). *Graphs Exercises – Block 1: Complexity Management.* `https://mltechdrawer.github.io/BIGDATA/Block1_Theoretical_Concepts_of_Big_Data/Complexity_Management/graphs_exercises`
- Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM.*
- Repository including the Task and Images for a better visualization. `https://github.com/DieGodMF4/Weekly-tasks-BD`