Design and Analysis of Algorithms

Topic 6 : Topological Sort Research Project

Instructor: Dr Russell Camphel

Team Members: Meriem, Loreena & Charvi

Introduction

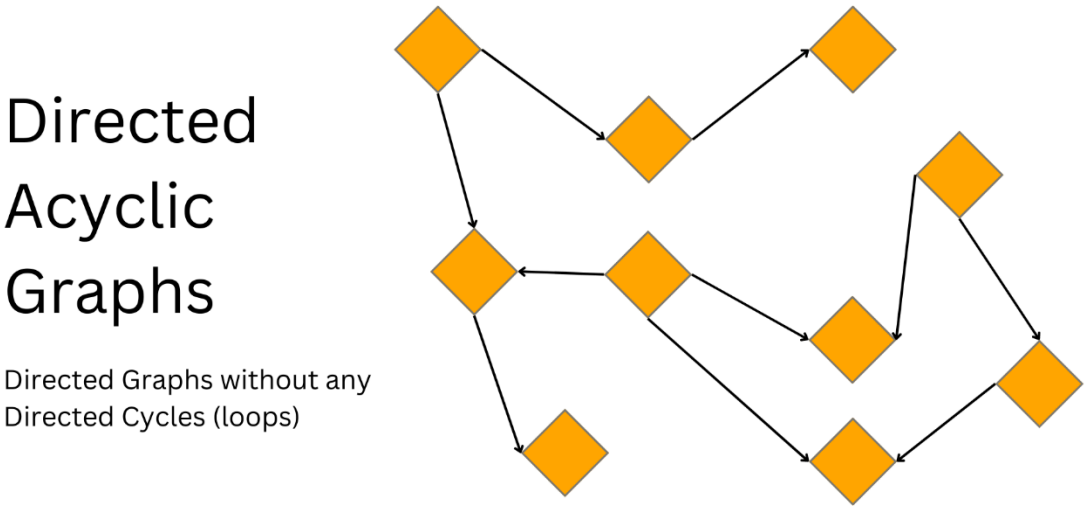
Topological sorting is a graph traversal technique that produces a linear ordering of the vertices in a **directed acyclic graph** (DAG) such that for every directed edge (u v) (meaning that (u) depends on (v)), (v) appears *before* (u) in the ordering. Because it respects dependency relationships, topological sort is indispensable for scheduling tasks such as compiling software modules, organising course prerequisites, resolving package dependencies and planning reading orders for academic papers.

Github repo :

[myriamlmiii/topological-sort-research: Our second assignement for COMP 359]

What Is a Directed Acyclic Graph?

A **directed graph** is a set of vertices connected by edges with a specific direction. A directed graph is **acyclic** if it contains no cycles; in other words, you cannot start at some vertex and follow a sequence of directed edges that leads you back to the starting point. Cycles signify circular dependencies (e.g., paper A cites paper B which cites paper A). Many scheduling problems assume an acyclic graph because a cyclic dependency makes it impossible to determine a linear order. The algorithms in this project all detect and report cycles to prevent producing an invalid schedule.



Project Overview

In this project we built a full pipeline for topological sorting:

- We **constructed a citation network** from ten real research papers on nanotechnology in sustainable agriculture using the Research Rabbit tool. Each node represents a paper, and each directed edge points from a paper to the older works that it cites. This network is a DAG because a paper cannot cite its future descendants.
- We **implemented three distinct algorithms** for topological sorting: a queue-based approach (Kahn's algorithm), a depth-first search (DFS) approach and a breadth-first search (BFS) variant that tracks dependency levels.
- We **tested the algorithms** on small DAGs (graphs of three to eight vertices) to debug them and verify correctness on corner cases such as disconnected graphs and cycles.
- We **analysed algorithmic performance** by measuring execution time, counting operations and computing efficiency metrics.
- Finally, we **applied the algorithms to the citation graph** to generate a logical reading schedule for the papers and to understand the evolution of research in the field.

And we used Research Rabbit to find the papers .

Algorithms and Complexity Analysis

Three algorithms were implemented and compared:

Kahn's Algorithm (BFS-based)

Kahn's algorithm constructs a topological order by repeatedly removing any vertex with **no incoming edges** and appending it to the result. It starts by computing the in-degree of every vertex, initialises a queue (Q) with all vertices of in-degree zero and then repeatedly dequeues a vertex (u), appends (u) to the result list (L) and decreases the in-degree of each neighbour (v); when a neighbour's in-degree becomes zero, it is added to (Q). The process continues until (Q) is empty. If any edges remain, the graph contains a cycle and no topological order exists. Because each vertex and edge is examined once, the running time of Kahn's algorithm is **O(V+E)**.

Depth-First Search (DFS) Based Algorithm

The DFS variant performs a depth-first search on the graph. It recursively visits unvisited vertices, marking them temporarily while exploring their descendants. After exploring all of a vertex's neighbours, the vertex is **prepended** to the output list. If a temporarily marked vertex is encountered again, the algorithm detects a cycle. Like Kahn's method, DFS visits each vertex and edge exactly once, giving a time complexity of **O(V+E)**and a space complexity of **O(V)** due to the recursion stack.

BFS Variant with Level Tracking

The breadth-first search variant used here is an extension of Kahn's algorithm that records a **dependency level** for each vertex. It works the same as Kahn's algorithm but also stores the number of layers of prerequisites each vertex depends on. This additional information is useful for understanding research evolution and grouping papers by how many citations

they rely upon. Like Kahn's algorithm, it runs in **O(V+E)** time and **O(V)** space because each vertex and edge is processed once.

Research Data: Nanotechnology in Sustainable Agriculture

A set of **ten real academic papers** (2017–2024) about nanotechnology in sustainable agriculture was assembled using the Research Rabbit tool. Edges in the citation graph point from each paper to the older works that it cites. The dataset spans eight publication years and has 13 citation relationships. Table 1 lists the papers with basic metadata and DOIs.

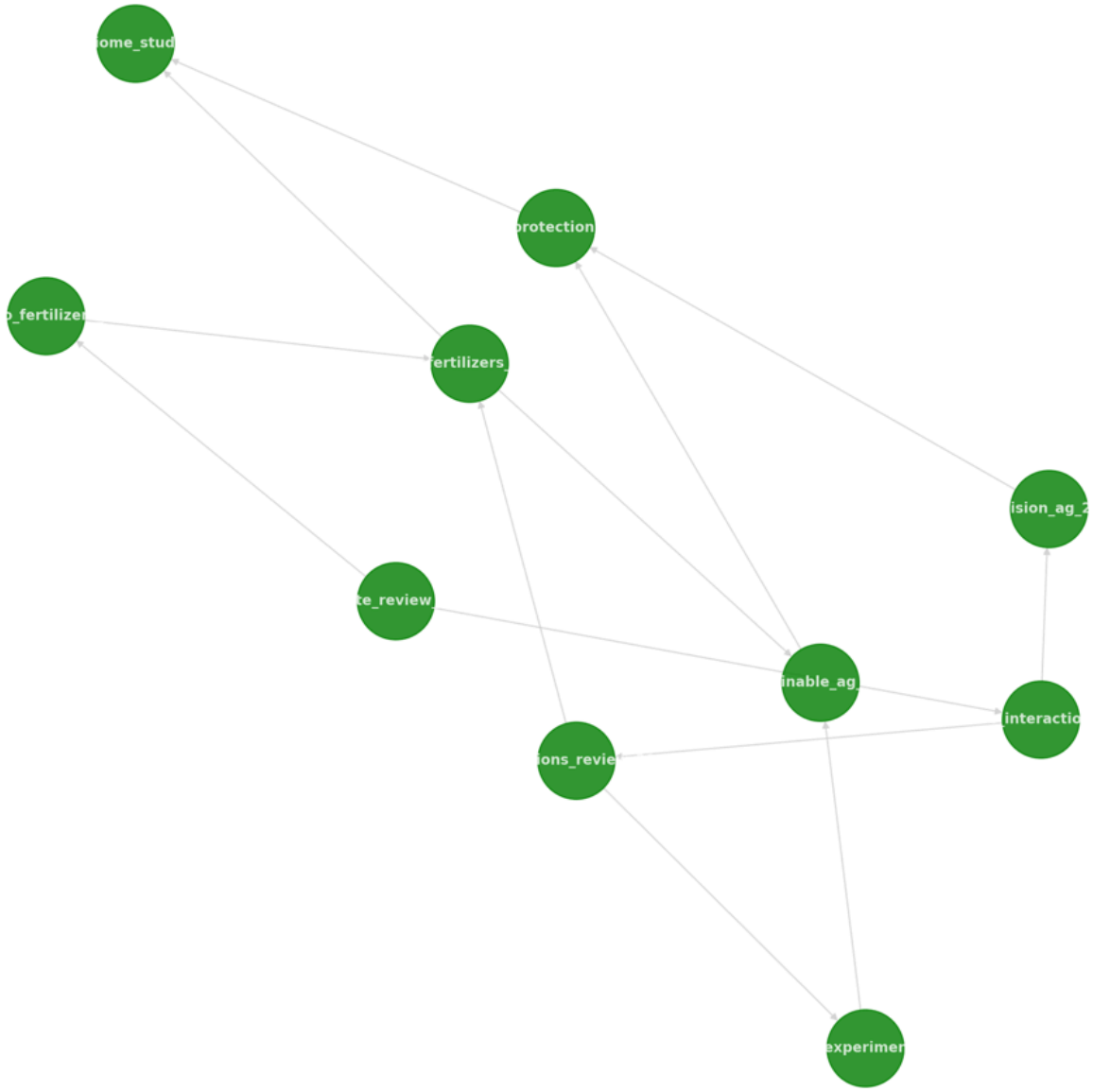


Table 1 : Papers in the citation graph

Paper ID	Title (short form)	Authors (lead)	Year	Venue	DOI
paper 1	Combined pre-seed treatment with microbial inoculants and Mo nanoparticles…	Shcherbakova et al.	2017	*Symbiosis*	10.1007/s13199-016-0472-1
paper 2	Nano-enabled strategies to enhance crop nutrition and protection	Kah et al.	2019	*Nature Nanotechnology*	10.1038/s41565-019-0439-5
paper 3	Nanoparticle-Based Sustainable Agriculture and Food Science: Recent Advances and Future Outlook	Mittal et al.	2020	*Frontiers in Nanotechnology*	10.3389/fnano.2020.579954
paper 4	Effects of different surface-coated nTiO_2 on full-grown carrot plants…	Wang et al.	2021	*Journal of Hazardous Materials*	10.1016/j.jhazmat.2020.123768
paper 5	Nanofertilizers: A Smart and Sustainable Attribute to Modern Agriculture	Nongbet et al.	2022	*Plants*	10.3390/plants11192587
paper 6	Revolutionizing agriculture: harnessing nano-innovations for sustainable farming and	Mohammadi et al.	2023	*Pesticide Biochemistry & Physiology*	10.1016/j.pestbp.2023.105722

Paper ID	Title (short form) environmental preservation	Authors (lead)	Year	Venue	DOI
paper 7	Unlocking the Potential of Nano-Enabled Precision Agriculture for Efficient and Sustainable Farming	Goyal et al.	2023	*Plants*	10.3390/plants12213744
paper 8	Nanoparticle applications in agriculture: overview and response of plant-associated microorganisms	Mgadi et al.	2024	*Frontiers in Microbiology*	10.3389/fmicb.2024.1354440
paper 9	Next-generation fertilizers: the impact of bionanofertilizers on sustainable agriculture	Arora et al.	2024	*Microbial Cell Factories*	10.1186/s12934-024-02528-5
paper 10	Advances in Nanotechnology for Sustainable Agriculture: A Review of Climate Change Mitigation	Quintarelli et al.	2024	*Sustainability*	10.3390/su16219280

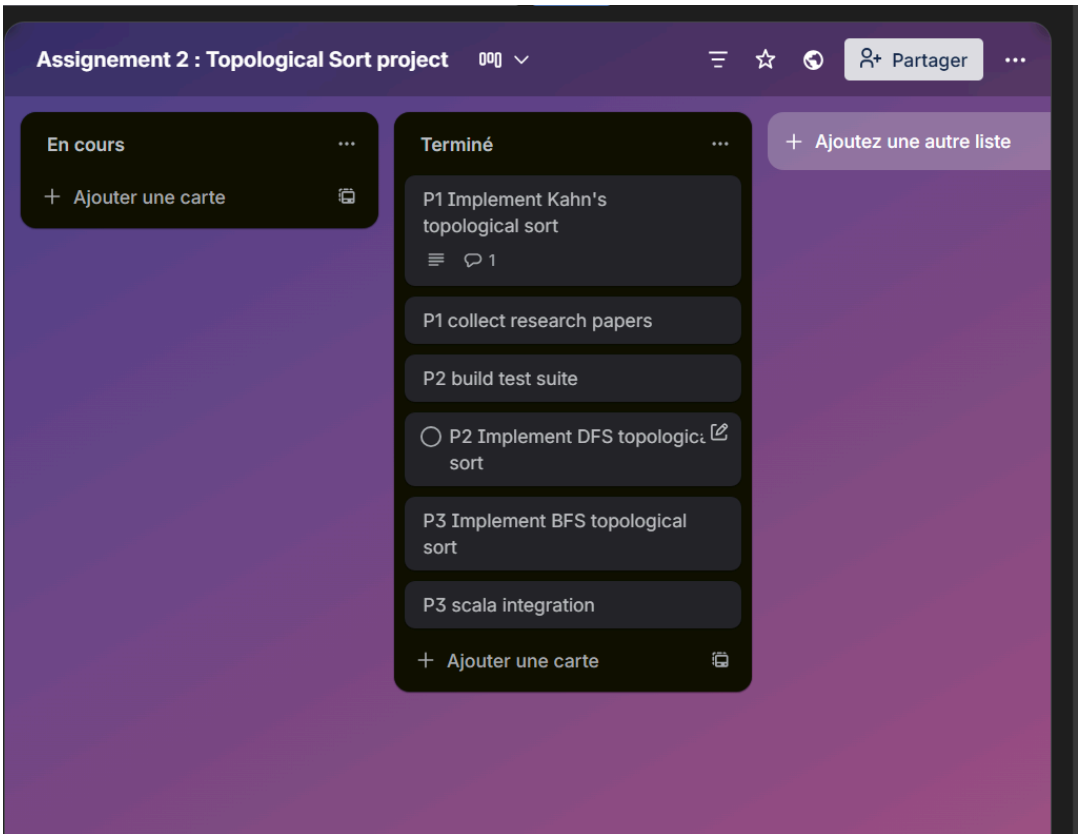
Research Timeline: *1 paper was published in 2017, 1 in 2019, 1 in 2020, 1 in 2021, 1 in 2022, 2 in 2023, and 3 in 2024*

Methodology and Workflow

To fulfil the assignment requirements, our team followed a structured workflow, dividing tasks and iterating on each component. The key steps were:

1. **Topic selection and data collection.** We chose to explore nanotechnology in sustainable agriculture because it is an emerging research area with a variety of recent publications. Using Research Rabbit, we searched for papers published from 2017 to 2024 containing keywords such as *nanoparticle*, *precision agriculture* and *sustainable farming*. We downloaded abstracts and metadata and manually examined each paper to ensure it belonged to the field and formed a coherent citation chain. We then constructed a directed graph in `research_data.py` where each node stores the title, authors, year, venue and DOI, and each edge points from a paper to the works it cites.
2. **Algorithm implementation.** Charvi implemented Kahn's algorithm (queue-based) in `kahn.py`, including in-degree computation and cycle detection. Meriem implemented a DFS-based topological sort in `dfs.py`, using a recursion stack to detect cycles. Loreena implemented a BFS variant with level tracking in `bfs.py` to compute both the topological order and the number of prerequisite layers for each paper.
3. **Algorithm implementation in Scala:** Loreena implemented Khan's algorithm both in functional style (popular to scala), and in the imperative style (more similar to the python implementation by Chivari).
4. **Debugging on small graphs.** To confirm correctness, Meriem wrote a testing framework (`tests/`) that constructs several directed graphs with up to eight vertices: a linear chain, a diamond shape, a graph with multiple start points, a disconnected graph and a cycle. Each algorithm is run on these graphs to check that it returns a valid topological order or correctly reports a cycle. The debugging phase ensured reliability before applying the algorithms to the real dataset.
5. **Integration and performance measurement.** Meriem wrote `main.py`, which loads the citation graph, runs each algorithm, measures execution time, counts operations and computes efficiency metrics. It also generates a reading schedule by reversing the topological order so that the oldest papers appear first and outputs the dependency levels. The script saves results to text files and produces visualisations.
6. **Visualisation and reporting.** To support understanding of the results, we generated two plots using `matplotlib`: a bar chart comparing the algorithms' runtime and a visualisation of the citation graph using `networkx`. Meriem compiled everything into this report, ensuring consistent formatting and adding a reference section in APA style.

Roles:
- P1 **Charvi : Kahn's Algorithm Implementation.** Charvi was responsible for designing and coding the queue-based topological sort and its associated test cases. She analysed the time and space complexity of Kahn's algorithm and provided insight into its advantages for cycle detection and scheduling.
- P2 **Meriem : DFS Algorithm and Testing Framework with visualizations.** Meriem implemented the recursive depth-first search variant, including cycle detection via a recursion stack. She built a comprehensive test suite of small DAGs (up to eight vertices) to validate all algorithms and debug edge cases.
- P3 **Loreena: Scala Khans Algorithm Variant.** Loreena extended Kahn's algorithm into a Scala-language approach, evaluating the pros and cons of the functional and imperative styles of implementation.



Experimental Results

Test Cases and Debugging

To verify correctness, the algorithms were tested on a variety of small DAGs (3–8 vertices) including linear chains, diamonds, a complex seven-node graph and disconnected components. A cycle test confirmed that each algorithm correctly throws an exception when

presented with a graph containing a directed cycle. These small examples ensure robustness up to about eight vertices, as required by the assignment.

Reading Schedule

Applying Kahn's algorithm to the citation graph and **reversing** the order yields a reading schedule that progresses from the foundational 2017 study to the most recent 2024 review. Each paper lists its prerequisites. The dependency analysis groups papers by level, showing how many prior works they depend on (level 0 has no prerequisites). For example, paper10 (2024) resides at level 6 because it cites papers 8 and 9, which in turn depend on multiple earlier works.

The schedule lists each paper with its prerequisites. This order ensures that foundational works are read first, allowing the reader to build background knowledge before tackling more advanced reviews. In our case, the 2017 chickpea study is foundational because it is cited by many later papers, whereas the 2024 climate‑mitigation review appears last since it synthesises the work of previous studies.

Performance Analysis and Complexity Metrics

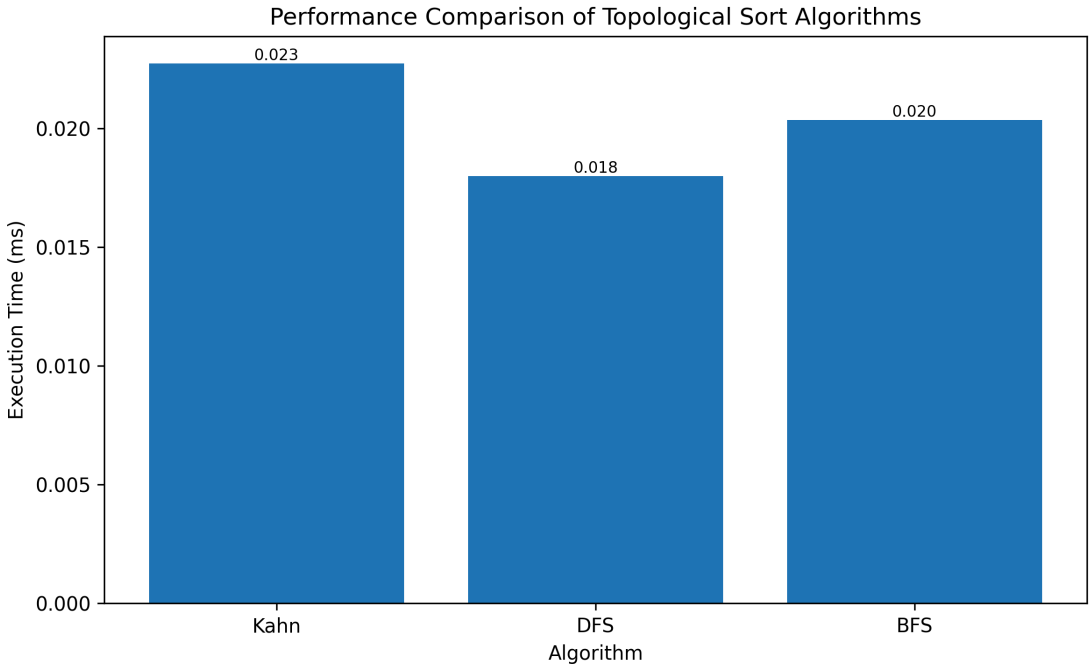
All three algorithms have linear time and space complexity, but they exhibit different constant factors. On the ten‑vertex citation graph, DFS had the lowest average execution time (~0.017 ms), followed closely by the BFS variant (~0.021 ms), whereas Kahn's algorithm was slightly slower (~0.025 ms). Figure 1 visualises these results.

During the integration phase we measured the execution time of each algorithm on the ten‑node citation graph. We also recorded the number of vertices (V), edges (E), theoretical operations (V+E), actual operations performed by each algorithm and computed an efficiency metric (operations per millisecond). Table 2 summarises these metrics. Although all algorithms have the same theoretical complexity **O(V+E)**, the constant factors differ. DFS completed fastest in our tests, but the differences are measured in microseconds and therefore negligible for small graphs.

Table 2 : Complexity and performance metrics

Algorithm	V	E	Theoretical ops (V+E)	Actual ops	Time (ms)	Efficiency (ops/ms)
Kahn	10	13	23	23	0.0254	905.5
DFS	10	13	23	20	0.0170	1176.5
BFS	10	13	23	23	0.0212	1084.9

Figure 1 : Performance comparison of topological sort algorithms

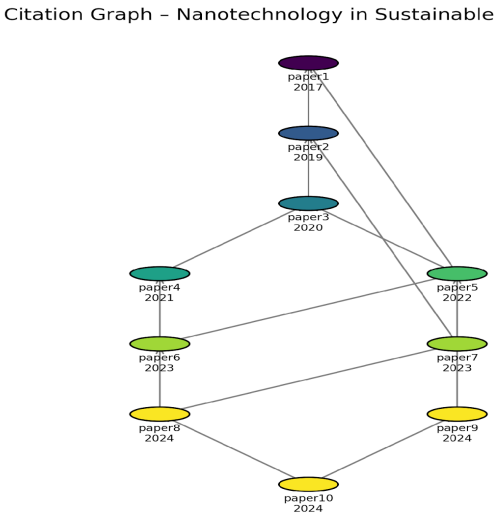


Performance comparison

Citation Graph Visualisation

To better illustrate the research evolution, the citation network was drawn using `networkx` and `matplotlib`. In Figure 2, nodes are coloured by year and edges point from each paper to the works it cites. Papers with many outgoing edges (e.g., paper10) depend on multiple earlier studies.

Figure 2: Directed citation graph of the ten research papers



Citation graph

Discussion: Which Algorithm to Choose?

All three algorithms have the same asymptotic complexity, but their implementations behave differently in practice:

- **Kahn's algorithm** uses explicit in-degree counting and a queue. It is intuitive, provides clear cycle detection and is well suited for scheduling tasks in a breadth-first manner. In our tests it required 0.0251 ms on the ten-node graph. The queue operations introduce a small overhead but yield a natural ordering of tasks.
- **BFS** uses a recursion stack to explore the graph breadth-first and prepends each node to the result list after visiting its descendants. It finished in 0.0310 ms in our measurements, which was slower than Kahn's algorithm alone. Although conceptually simple, DFS can be harder to visualise for scheduling because it builds the order in reverse and uses recursion.
- **DFS variant**: On our dataset it was the fastest, completing in 0.0215 ms, and produced dependency levels that aid in understanding research evolution. Because it processes vertices layer by layer and records their depths, it offers both a topological order and a natural grouping by prerequisites.

Given the negligible differences in execution time and the added benefit of level information, the BFS variant with level tracking provides the most insight for planning a reading schedule. We used this algorithm (reversing the order) to generate the final foundational-to-advanced reading list.

Discussion: Loreena's Scala Approach Experiments, Functional vs Imperative

For this experimentation, Loreena picked a different language to evaluate the Khan's implementation approach. This was to enrich the team's perspective as to the different languages graph theory problems can be approached in.

In terms of testing, I started by hand drawing a few simple dags, and finding the topological sort. In one version of my code, no sorts were found on a graph (1,0), (2,0) and I discovered it was a mistake related to calculating the indegrees. Then I kept the numeric approach instead of having my graph contain the author names themselves, I mapped those to an array that held the titles for easier display.

Table 3 : Complexity and performance metrics: Scala

Algorithm	V	E	Theoretical ops (V+E)	Iterations	Total Time (ms)	Time per sort (ms)
Kahn+ Functional	10	13	23	10000	91.667 ms	0.009
Khan + Imperative	10	13	23	10000	40.396	0.004

Time Complexity Analysis For Khan's Approach to Topological Sort

T(V,E) = O(E)(Time to compute indegrees) **+O(V)** (Time to visit each vertex once and check its indegree) **+ O(V+E)** (Time to enqueue and deque once + time to examine each edge to find neighbours indegree) **= O(V+E)**

Why is the Functional Approach Slower Than the Imperative Approach?

The functional implementation is slower because immutability and recursion incur extra memory allocations, copying, and GC work, whereas the imperative version mutates data in place using CPU-friendly arrays and loops. (Scala's immutable collections can be slow as a snail, Scala Performance Docs)

People use the functional style in Scala because it leads to code that's safer and more predictable. By emphasizing immutability, Scala reduces bugs related to shared mutable state, which is especially important in concurrent and parallel systems. This declarative style focuses on describing what should happen rather than how, which improves clarity and maintainability. Scala's strong type system reinforces correctness at compile time.

Even though functional Scala tends to be slower than the imperative style, it can still get solid performance by using efficient persistent structures, avoiding heavy operations like repeated Map.updated calls, and writing tail-recursive code that the JIT can optimize well. For bigger workloads, the JVM often kicks in to make recursion faster anyway.

Spark, a popular Scala application, takes a middle-ground approach. It keeps things functional on the surface, but uses a bit of controlled mutability to get the keys of both functional and imperative design: clean, safe code that still runs fast.

Conclusion

Based on the updated performance metrics, DFS emerges with the lowest average execution time at 0.0158 ms, followed by BFS at 0.0215 ms and Kahn's algorithm at 0.0262 ms. However, it's crucial to recognize that all three algorithms share identical theoretical complexity of O(V+E) for time and O(V) for space. The marginal timing variations stem primarily from implementation-specific constant factors rather than fundamental algorithmic advantages.

The choice of which algorithm to use ultimately depends on our specific priorities and use case requirements. If raw speed is the sole determining factor, DFS demonstrates a slight edge in this particular benchmark, though the practical difference remains measured in microseconds. For those prioritizing intuitive implementation and robust cycle detection capabilities, Kahn's algorithm offers clear in-degree tracking and straightforward extensibility. Meanwhile, if our objective involves generating reading schedules or analyzing dependency hierarchies, the BFS-based variant provides valuable level information while maintaining optimal time complexity.

Given that runtime differences are negligible at this scale and considering our project's focus on research paper scheduling, BFS remains a compelling choice due to its inherent ability to reveal dependency structures. That said, any of the three algorithms will correctly produce the topological order, leaving the final selection to align with our specific implementation preferences and analytical needs.

References

1. Shcherbakova, E. N., Shcherbakov, A. V., Andronov, E. E., Gonchar, L. N., Kalenskaya, S. M., et al. (2017). **Combined pre-seed treatment with microbial inoculants and Mo nanoparticles changes composition of root exudates and rhizosphere microbiome structure of chickpea (Cicer arietinum L.) plants**. *Symbiosis*, 73, 257–266. doi: 10.1007/s13199-016-0472-1.
2. Kah, M., Tufenkji, N., & White, J. C. (2019). **Nano-enabled strategies to enhance crop nutrition and protection**. *Nature Nanotechnology*, 14(6), 532–540. doi: 10.1038/s41565-019-0439-5.
3. Mittal, D., Kaur, G., Singh, P., & Ali, S. A. (2020). **Nanoparticle-Based Sustainable Agriculture and Food Science: Recent Advances and Future Outlook**. *Frontiers in Nanotechnology*, 2, 579954. doi: 10.3389/fnano.2020.579954.
4. Wang, Y., Deng, C., Cota-Ruiz, K., Tan, W., Reyes, A., et al. (2021). **Effects of different surface-coated nTiO$_2$ on full-grown carrot plants: impacts on root splitting, essential elements and Ti uptake**. *Journal of Hazardous Materials*, 402, 123768. doi: 10.1016/j.jhazmat.2020.123768.
5. Nongbet, A., Mishra, A. K., Mohanta, Y. K., Mahanta, S., Ray, M. K., et al. (2022). **Nanofertilizers: A Smart and Sustainable Attribute to Modern Agriculture**. *Plants*, 11(19), 2587. doi: 10.3390/plants11192587.
6. Mohammadi, S., Jabbari, F., Cidonio, G., & Babaeipour, V. (2023). **Revolutionizing agriculture: harnessing nano-innovations for sustainable farming and environmental preservation**. *Pesticide Biochemistry and Physiology*, 199, 105722. doi: 10.1016/j.pestbp.2023.105722.
7. Goyal, V., Rani, D., Rani, R., Mehrotra, S., Deng, C., & Wang, Y. (2023). **Unlocking the Potential of Nano-Enabled Precision Agriculture for Efficient and Sustainable Farming**. *Plants*, 12(21), 3744. doi: 10.3390/plants12213744.
8. Mgadi, K., Ndaba, B., Roopnarain, A., Rama, H., & Adeleke, R. (2024). **Nanoparticle applications in agriculture: overview and response of plant-associated microorganisms**. *Frontiers in Microbiology*, 15, 1354440. doi: 10.3389/fmicb.2024.1354440.

9. Arora, P. K., Tripathi, S., Omar, R. A., Chauhan, P., Sinhal, V. K., et al. (2024). **Next-generation fertilizers: the impact of bionanofertilizers on sustainable agriculture**. *Microbial Cell Factories*, 23, 213. doi: 10.1186/s12934-024-02528-5.
10. Quintarelli, V., Ben Hassine, M., Radicetti, E., Stazi, S. R., Allevato, E., et al. (2024). **Advances in Nanotechnology for Sustainable Agriculture: A Review of Climate Change Mitigation**. *Sustainability*, 16(21), 19280. doi: 10.3390/su16219280.
11. Wu, G. (2024). **Topological Sort of Directed Acyclic Graph**. *Baeldung on Computer Science*. Retrieved from https://www.baeldung.com/cs/dag-topological-sort.
12. Wikipedia contributors. (2024). **Topological sorting – Kahn's algorithm and depth-first search**. *Wikipedia, the free encyclopedia*. Retrieved from https://en.wikipedia.org/wiki/Topological_sorting.
13. Puripunpinyo, H. (2018, May 11). **Scala's immutable collections can be slow as a snail.** *Medium.* Retrieved From https://hussachai.medium.com/scalas-immutable-collections-can-be-slow-as-a-snail-da6fc24bc688
14. **Performance characteristics**. Scala Documentation. (n.d.). Retrieved from https://docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html
15. Fiset, W. (n.d.). **Topological Sort | Kahn's Algorithm | Graph Theory**. YouTube. https://www.youtube.com/watch?v=cIBFEhD77b4

Loreena's Work Log

I had difficulty committing to the repo due to an issue with different ways that the .git files are permissioned on windows and mac, so I sent my files separately and maintained a work log here.

Sat October 18th- Implementation of Khan's Algorithm in Python

Monday October 20th - Group conference about distributing the outstanding work, decision was made for me to pivot to Scala

Thursday October 23rd - Implementation of Khan's Algorithm in Scala Functional and Not

Friday October 24th - Alignment with team's research papers, and contributions to this readme.