Course Prerequisite Planning Using DFS-Based Topological Sorting

Bryan Ho - 13523029
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: bryanho67@gmail.com , 13523029@std.stei.itb.ac.id

***Abstract*—Effective academic planning is crucial for students to complete their studies efficiently, particularly in programs with structured course dependencies like Informatics Engineering at Institut Teknologi Bandung. Many courses have prerequisites that form a directed acyclic graph (DAG), where each node represents a course and each edge represents a prerequisite relationship. This paper presents a systematic approach to course scheduling using topological sorting based on Depth-First Search (DFS). By representing course dependencies as a graph and applying DFS-based topological sorting, an optimal order in which students can take courses while satisfying all prerequisite constraints is generated. The proposed method not only ensures prerequisite fulfillment but also helps identify potential bottlenecks and critical paths in the curriculum structure. A case study on the Informatics Engineering curriculum at ITB demonstrates the practicality and effectiveness of this approach in assisting students and academic advisors in course planning and progression tracking.**

***Keywords*—Course Scheduling, Depth-First Search, Directed Acyclic Graph, Topological Sorting**

I. Introduction

Academic planning plays a vital role in ensuring timely graduation and efficient course progression, especially in structured undergraduate programs such as Informatics Engineering at Institut Teknologi Bandung (ITB). In such programs, many courses are interdependent, requiring students to complete certain prerequisites before taking advanced subjects. This creates a complex web of course relationships that must be carefully considered when building a semester-by-semester academic plan.

Manually navigating these prerequisites can be challenging for students and academic advisors alike, particularly when the curriculum spans multiple years and includes strict prerequisite chains. Poor planning may lead to delays in graduation due to missed prerequisite fulfillment or inefficient course selection.

These prerequisite relationships can be naturally modeled as a directed acyclic graph (DAG), where each node represents a course and each edge indicates a prerequisite requirement. In computer science, topological sorting is a technique that produces a linear ordering of nodes in a DAG such that for every directed edge from node A to node B, A comes before B in the ordering. This concept can be applied to curriculum planning by identifying a valid sequence in which students can complete their courses without violating any prerequisite constraints.

This paper explores the application of Depth-First Search (DFS)-based topological sorting to determine valid course sequences within the Informatics Engineering curriculum at ITB. By constructing a course dependency graph and applying the DFS algorithm, this approach helps reveal the critical paths and potential bottlenecks in the curriculum. The result is a tool that aids both students and academic advisors in making more informed and strategic decisions regarding course planning and registration.

II. Thoretical Basis

A. Graph

A graph G is a mathematical structure used to represent a set of discrete objects and the connections between them. Formally, graph is defined as an ordered pair $G = (V, E)$, where V is a finite set of vertices (or nodes) and E is a finite set of edges. In the context of this study, each vertex represents a distinct course in the curriculum, while each edge denotes a dependency or prerequisite relationship between courses.

Graphs can be classified based on the directionality of their edges:

1. Undirected Graph
   An edge has no direction, indicating a mutual or symmetric relationship between two vertices.
2. Directed Graph
   Each edge has a direction, from vertex u to vertex v denoted as (u, v). In this study, a directed edge (u, v) implies that course u is a prerequisite for course v.

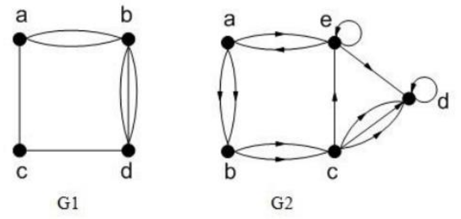

Figure 2.1 (G1) undirected graph, (G2) directed graph
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

Several key terms from graph theory are used throughout this paper:

1. Adjacent Vertices
   Two vertices are adjacent if there is a direct edge between them.
2. Path
   A sequence of vertices where each consecutive pair is connected by an edge.
3. Cycle
   A path that begins and ends at the same vertex, with all intermediate vertices distinct.
4. Connected Vertices
   Two vertices are connected if there exists a path between them. In directed graphs, this may refer to reachability in a specific direction.

A graph is said to be acyclic if it contains no cycles. When a directed graph is acyclic, it is called a Directed Acyclic Graph (DAG). DAGs are essential for modeling systems with one-way dependencies and no circular references, ensuring that no course indirectly depends on itself. This makes DAGs particularly suitable for representing course prerequisite structures.

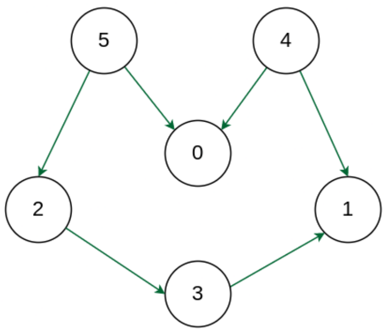

Figure 2.2 Directed acyclic graph (DAG)
Source: https://www.geeksforgeeks.org/dsa/topological-sorting/

B. Topological Sorting

Topological sorting is defined as a linear ordering of vertices in a Directed Acyclic Graph (DAG) such that for every directed edge (u, v), vertex u appears before vertex v in the ordering. This technique is fundamental for sequencing tasks or events with dependencies, making it directly applicable to course prerequisite planning. In the context of curriculum design, a topological sort produces a sequence of courses in which all prerequisite constraints are satisfied. If course u is a prerequisite for course v, then u will necessarily appear earlier than v in the resulting order.

It is important to note that topological sorting is only applicable to DAGs. This constraint arises for two primary reasons:

1. Undirected Edges
   Graphs with undirected edges imply mutual dependency between connected vertices. For instance, an undirected edge between u and v suggests that u depends on v and v depends on u. Such bidirectional dependency contradicts the requirement of a linear ordering, as neither node can be definitively placed before the other without violating the dependency rule.
2. Cycles
   The presence of a cycle in a directed graph, such as $u \rightarrow v \rightarrow w \rightarrow u$, also prevents topological sorting. Cycles represent circular dependencies where every vertex is indirectly dependent on itself, making it impossible to identify a valid starting point. Any attempt to sort such a graph would violate the prerequisite condition, as no vertex in the cycle can come before all others it depends on.

Another significant characteristic of topological sorting is that the resulting order may not be unique. A given DAG may have multiple valid topological orderings depending on the structure of its dependencies. This reflects real-world flexibility, where several different but valid sequences may satisfy all constraints.

For example, consider a graph with vertices $V = \{0, 1, 2, 3, 4, 5\}$ and edges $E = \{(2,3), (3,1), (4,0), (4,1), (5,0), (5,2)\}$ as shown in Figure 2.2. Both of the following are valid topological orderings:

- $5 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0$
- $4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0$

Each ordering respects the prerequisite relationships in the graph, even though the paths to the final outcome differ. This non-uniqueness underscores the versatility of topological sorting within constrained systems like academic course planning.

C. Depth-First Search (DFS)

Depth-First Search (DFS) is one of the fundamental graph traversal algorithms used to systematically explore the vertices and edges of a graph. The core principle of DFS is to start at a selected vertex and explore as far as possible along each branch before backtracking. This approach makes DFS particularly useful in applications that require exhaustive traversal, such as cycle detection, pathfinding, and most notably, topological sorting on Directed Acyclic Graphs (DAGs).

The basic steps of a DFS traversal from a given starting vertex v are as follows:

1. Visit vertex v.
2. Recursively visit each unvisited neighbor w of v.
3. If a vertex u is reached such that all its neighbors have already been visited, the algorithm backtracks to the previous vertex that still has unvisited neighbors.
4. This process continues until all reachable vertices from the starting vertex have been visited.
5. The search is complete when there are no more unvisited vertices that can be reached from any previously visited node.

DFS is commonly implemented using recursion or an explicit stack, which naturally maintains the traversal order and facilitates backtracking.

D. Topological Sorting using DFS

One of the most widely used methods to perform topological sorting is by leveraging Depth-First Search (DFS). In this approach, the graph is traversed recursively, exploring each vertex and its dependencies before recording it in the final order. A crucial component in this algorithm is the use of a stack as an auxiliary data structure to ensure that nodes are added in post-order, that is, a node is pushed onto the stack only after all of its adjacent (dependent) nodes have been fully visited.

The general algorithm for topological sorting using DFS is as follows:

1. Construct a directed acyclic graph (DAG) with n vertices and directed edges representing prerequisite relationships.
2. Initialize a visited array (or set) to keep track of visited nodes and an empty stack to store the topological order.
3. For each unvisited vertex in the graph, call a DFS function starting from that vertex and mark the current node as visited.
4. Recursively perform DFS on all unvisited neighbors (nodes that directly depend on the current node).
5. Once all neighbors are visited, push the current node onto the stack.
6. After all vertices have been visited, pop elements from the stack one by one to generate the topologically sorted order.

This method guarantees that each course is placed after all of its prerequisites in the final sequence. Because nodes are pushed only after their dependent nodes have been processed, the stack naturally captures the reverse of the desired ordering. Popping from the stack yields a valid course progression path that satisfies all prerequisite constraints.

III. Implementation

A. Research Limitation

On writing this paper, the author limits the implementation to a case study involving a single course from the Informatics Engineering undergraduate curriculum at Institut Teknologi Bandung. This simplification is intended to clearly demonstrate the application of DFS-based topological sorting in course prerequisite planning. The limitations of this study are:

1. The dataset is based solely on the official 2019 Informatics Engineering curriculum.
2. Only one target course is used as the endpoint in the prerequisite dependency chain.
3. The analysis includes only the direct and indirect prerequisite courses that lead to this target course.
4. Prerequisite relationships are assumed to form a directed acyclic graph (DAG) with no cycles.
5. Semester information, such as course availability in specific semesters, is not considered.

B. Research Site

This study uses the undergraduate 2019 Informatics Engineering curriculum from Institut Teknologi Bandung as its research site. The course selected as the endpoint for prerequisite planning is IF4073 – Image Interpretation and Processing, a higher-level subject typically taken in the last year of the program. This course serves as the target node in the prerequisite dependency chain used to demonstrate the proposed method.

The implementation focuses on all direct and indirect prerequisite courses that must be completed before enrolling in IF4073 – Image Interpretation and Processing, as outlined in the official academic guide. The prerequisite structure is as follows:

IF4073 – Image Interpretation and Processing

- IF3270 – Machine Learning
  - IF3170 – Artificial Intelligence
    - IF2121 – Computational Logic
    - IF2124 – Formal Language Theory and Automata
      - IF2120 – Discrete Mathematics
      - IF2110 – Algorithm and Data Structure
    - IF2220 – Probability and Statistics
      - MA1101 – Mathematics IA
      - MA1201 – Mathematics IIA
      - IF2120 – Discrete Mathematics
    - IF2211 – Algorithm Strategies
  - IF2110 – Algorithm and Data Structure

- IF3260 – Computer Graphics
    - IF2130 – Computer Organization and Architecture
    - IF2110 – Algorithm and Data Structure
    - IF2123 – Geometric and Linear Algebra
        - MA1101 – Mathematics IA

This hierarchical dependency is modeled as a directed acyclic graph (DAG), where each course is represented as a node, and each prerequisite relationship is represented by a directed edge. The complete prerequisite structure for the IF4073 – Image Interpretation and Processing course, serving as the case study for this paper, is illustrated in Figure 3.1.

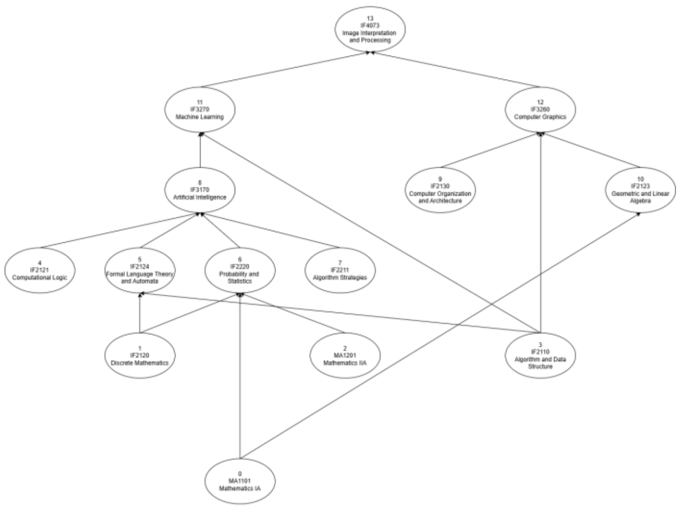
13 IF4073 Image Interpretation and Processing
11 IF3270 Machine Learning
12 IF3260 Computer Graphics
8 IF3170 Artificial Intelligence
9 IF2130 Computer Organization and Architecture
10 IF2123 Geometric and Linear Algebra
4 IF2121 Computational Logic
5 IF2124 Formal Language Theory and Automata
6 IF2220 Probability and Statistics
7 IF2211 Algorithm Strategies
1 IF2120 Discrete Mathematics
2 MA1201 Mathematics IIA
3 IF2110 Algorithm and Data Structure
0 MA1101 Mathematics IA

Figure 3.1 IF4073 – Image Interpretation and Processing course prerequisite directed acyclic graph (DAG)

Note that each course node in Figure 3.1 is labeled with a number, which corresponds to the course name as detailed in Table 3.1.

Table 3.1 Course node number mapping for IF4073 – Image Interpretation and Processing course prerequisite

Number	Course Code	Course Name
0	MA1101	Mathematics IA
1	IF2120	Discrete Mathematics
2	MA1201	Mathematics IIA
3	IF2110	Algorithm and Data Structure
4	IF2121	Computational Logic
5	IF2124	Formal Language Theory and Automata
6	IF2220	Probability and Statistics
7	IF2211	Algorithm Strategies
8	IF3170	Artificial Intelligence
9	IF2130	Computer Organization and Architecture
10	IF2123	Geometric and Linear Algebra
11	IF3270	Machine Learning
12	IF3260	Computer Graphics
13	IF4073	Image Interpretation and Processing

C. *Implementation of DFS-Based Topological Sorting for the IF4073 – Image Interpretation and Processing Course Prerequisites*

With the course prerequisite structure now represented as a directed acyclic graph (DAG), as shown in Figure 3.1, the next step is to determine a valid sequence of courses that satisfies all prerequisite constraints. The core of this implementation relies on a Depth-First Search (DFS) traversal modified to produce a topological ordering. The algorithm systematically explores each node (course) and its unvisited dependencies, ensuring that all prerequisites are fully processed before a course is added to the final sequence.

To achieve this, a stack is used as an auxiliary data structure. A stack plays a crucial role in maintaining this order. During the DFS traversal, courses are pushed onto this stack only after all of their dependent courses (courses for which the current node is a prerequisite) have been fully visited and processed. This ensures that foundational courses, which have fewer or no unmet prerequisites, are placed closer to the top of the stack. As a result, when elements are popped from the stack, they directly yield a valid topological order for course progression.

To demonstrate the algorithm (as detailed in Section II.D, "Topological Sorting using DFS," of the Theoretical Basis), DFS-based topological sorting is applied to the prerequisites of IF4073 – Image Interpretation and Processing. The implementation of this approach is as follows:

Iteration 0:
Initialize an array or set that marks all nodes (courses) as unvisited initially to track nodes and an empty stack to store the course in topological order.

Visited[] =

0	1	2	3	4	5	6
F	F	F	F	F	F	F
7	8	9	10	11	12	13
F	F	F	F	F	F	F

Stack[] =

top

bottom

Iteration 1:
Start DFS from an unvisited node, such as node 0, and mark it as visited.

Visited[] =

0	1	2	3	4	5	6
T	F	F	F	F	F	F
7	8	9	10	11	12	13
F	F	F	F	F	F	F

Stack[] =

top

bottom

Iteration 2:
Among the unvisited neighbors node 6 and node 10 of node 0, perform DFS on node 6 and mark it as visited.

Visited[] =

0	1	2	3	4	5	6
T	F	F	F	F	F	T
7	8	9	10	11	12	13
F	F	F	F	F	F	F

Stack[] =

top

bottom

Iteration 3:
Since node 8 is an unvisited neighbor of node 6, perform DFS on node 8 and mark it as visited.

Visited[] =

0	1	2	3	4	5	6
T	F	F	F	F	F	T
7	8	9	10	11	12	13
F	T	F	F	F	F	F

Stack[] =

top

bottom

Iteration 4:
Since node 11 is an unvisited neighbor of node 8, perform DFS on node 11 and mark it as visited.

Visited[] =

0	1	2	3	4	5	6
T	F	F	F	F	F	T
7	8	9	10	11	12	13
F	T	F	F	T	F	F

Stack[] =

top

bottom

Iteration 5:
Since node 13 is an unvisited neighbor of node 11, perform DFS on node 13 and mark it as visited.

Node 13 has no unvisited neighbors, so push it onto the stack. Backtrack and continue pushing nodes onto the stack once all their neighbors are processed.

Visited[] =

0	1	2	3	4	5	6
T	F	F	F	F	F	T
7	8	9	10	11	12	13
F	T	F	F	T	F	T

Stack[] =

top

6	8	11	13			

bottom

Iteration 6:
After completing the DFS traversal from node 6 and pushing it onto the stack, return to node 0 and continue DFS on its next unvisited neighbor, node 10. Mark it as visited.

Visited[] =

0	1	2	3	4	5	6
T	F	F	F	F	F	T
7	8	9	10	11	12	13
F	T	F	T	T	F	T

Stack[] =

top

6	8	11	13			

bottom

Iteration 7:
Since node 12 is an unvisited neighbor of node 10, perform DFS on node 12 and mark it as visited.

Node 12 has no unvisited neighbors, so push it onto the stack. Backtrack and continue pushing nodes onto the stack once all their neighbors are processed.

Since node 0 has no unvisited neighbors left, push it onto the stack.

Visited[] =

0	1	2	3	4	5	6
T	F	F	F	F	F	T

7	8	9	10	11	12	13
F	T	F	T	T	T	T

Stack[] =

top

0	10	12	6	8	11	13

bottom

Iteration 8:
Select an unvisited node, such as node 1, and perform DFS from it. Mark node 1 as visited.

Visited[] =

0	1	2	3	4	5	6
T	T	F	F	F	F	T
7	8	9	10	11	12	13
F	T	F	T	T	T	T

Stack[] =

top

0	10	12	6	8	11	13

bottom

Iteration 9:
Since node 5 is an unvisited neighbor of node 1, perform DFS on node 5 and mark it as visited.

Node 5 has no unvisited neighbors, so push it onto the stack. Backtrack and continue pushing nodes onto the stack once all their neighbors are processed.

Since node 1 has no unvisited neighbors left, push it onto the stack.

Visited[] =

0	1	2	3	4	5	6
T	T	F	F	F	T	T
7	8	9	10	11	12	13
F	T	F	T	T	T	T

Stack[] =

top

1	5	0	10	12	6	8

bottom

11	13					

Iteration 10:
Select an unvisited node, such as node 2, and perform DFS from it. Mark node 2 as visited.

Node 2 has no unvisited neighbors, so push it onto the stack.

Visited[] =

0	1	2	3	4	5	6
T	T	T	F	F	T	T
7	8	9	10	11	12	13
F	T	F	T	T	T	T

Stack[] =

top

2	1	5	0	10	12	6

bottom

8	11	13				

Iteration 11:
Select an unvisited node, such as node 3, and perform DFS from it. Mark node 3 as visited.

Node 3 has no unvisited neighbors, so push it onto the stack.

Visited[] =

0	1	2	3	4	5	6
T	T	T	T	F	T	T
7	8	9	10	11	12	13
F	T	F	T	T	T	T

Stack[] =

top

3	2	1	5	0	10	12

bottom

6	8	11	13			

Iteration 12:
Select an unvisited node, such as node 4, and perform DFS from it. Mark node 4 as visited.

Node 4 has no unvisited neighbors, so push it onto the stack.

Visited[] =

0	1	2	3	4	5	6
T	T	T	T	T	T	T
7	8	9	10	11	12	13
F	T	F	T	T	T	T

Stack[] =

top

4	3	2	1	5	0	10

bottom

12	6	8	11	13		

Iteration 13:
Select an unvisited node, such as node 7, and perform DFS from it. Mark node 7 as visited.

Node 7 has no unvisited neighbors, so push it onto the stack.

Visited[] =

0	1	2	3	4	5	6
T	T	T	T	T	T	T
7	8	9	10	11	12	13
T	T	F	T	T	T	T

Stack[] =

top

7	4	3	2	1	5	0

bottom

10	12	6	8	11	13	

Iteration 14:
Select an unvisited node, such as node 9, and perform DFS from it. Mark node 9 as visited.

Node 9 has no unvisited neighbors, so push it onto the stack.

Visited[] =

0	1	2	3	4	5	6
T	T	T	T	T	T	T
7	8	9	10	11	12	13
T	T	T	T	T	T	T

Stack[] =

top

9	7	4	3	2	1	5

bottom

0	10	12	6	8	11	13

All nodes have been marked as visited. The DFS traversal is complete, and the stack now contains the nodes in reverse topological order.

IV. Result

After completing the DFS traversal, the stack contains the courses in topological order. To obtain the correct sequence for course planning, the elements are popped from the stack one by one. This produces a valid topological order in which each course appears only after all of its prerequisites have been scheduled.

By applying this DFS-based topological sorting algorithm to the course prerequisite graph for IF4073 – Image Interpretation and Processing, the resulting topological order obtained by popping the stack is as follows:

9 → 7 → 4 → 3 → 2 → 1 → 5 → 0 → 10 → 12 → 6 → 8 → 11 → 13

Translating these numerical labels back to their respective course codes and names (refer to Table 3.1 for mapping), this sequence represents one of the possible valid progression paths for students aiming to complete IF4073:

IF2130 (Computer Organization and Architecture) → IF2211 (Algorithm Strategies) → IF2121 (Computational Logic) → IF2110 (Algorithm and Data Structure) → MA1201 (Mathematics IIA) → IF2120 (Discrete Mathematics) → IF2124 (Formal Language Theory and Automata) → MA1101 (Mathematics IA) → IF2123 (Geometric and Linear Algebra) → IF3260 (Computer Graphics) → IF2220 (Probability and Statistics) → IF3170 (Artificial Intelligence) → IF3270 (Machine Learning) → IF4073 (Image Interpretation and Processing)

V. Conclusion

Effective academic progression is a critical aspect of higher education, especially within structured curricula such as the Informatics Engineering program at Institut Teknologi Bandung (ITB). By modeling course dependencies as a directed acyclic graph (DAG) and applying a depth-first search (DFS)-based topological sorting algorithm, this approach successfully generates a valid course sequence that satisfies all prerequisite constraints.

Topological sorting ensures that each course is scheduled only after all of its prerequisites have been completed, making it highly suitable for resolving curriculum planning challenges. This method not only maintains prerequisite integrity but also helps uncover the hierarchical structure of course dependencies. As a result, students can follow a clear and feasible progression path, while academic advisors gain a practical tool to support data-driven course planning.

As curriculum requirements grow increasingly complex, graph-based techniques such as DFS-based topological sorting offer a robust foundation for the development of intelligent and personalized academic planning systems.

VI. Appendix

Video explanation of this paper: https://youtu.be/ZAe_lYKwx8Y

VII. Acknowledgment

This paper was completed by the grace, guidance, and protection of God which enabled the author to finish it on time. The author would like to extend heartfelt gratitude to all parties who supported the study and learning process in IF2211 Algorithm Strategies course, especially to Dr. Nur Ulfa Maulidevi, S.T, M.Sc., Dr. Ir. Rinaldi Munir, M.T., and Monterico Adrian, M.T. as the course lecturers for their invaluable guidance. Lastly, the author apologizes for any shortcomings that may remain in this paper. It is sincerely hoped that this paper can serve as a useful reference for future studies or research purposes.

References

[1] Munir, Rinaldi. 2024. "Graf (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf (accessed on 21 June 2025).

[2] Munir, R., & Maulidevi, N. U. 2025. "Breadth First Search (BFS) dan Depth First Search (DFS) - Bagian 1". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf (accessed on 21 June 2025).

[3] GeeksforGeeks. 2023. "Introduction to Directed Acyclic Graph". https://www.geeksforgeeks.org/dsa/introduction-to-directed-acyclic-graph/ (accessed on 22 June 2025).

[4] GeeksforGeeks. 2025. "Topological Sorting". https://www.geeksforgeeks.org/dsa/topological-sorting/ (accessed on 22 June 2025).

[5] https://youtu.be/n_yl2a6n7nM?si=VW-PbbiktnCePlJw (accessed on 22 June 2025).

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Juni 2025

[illegible]

Bryan Ho (13523029)