Task Scheduling Using Topological Sorting Method

Vinu K C, H S Saraswathi, Dr. Latha B M, Jeevan B G, Arpit S B, Asha N S

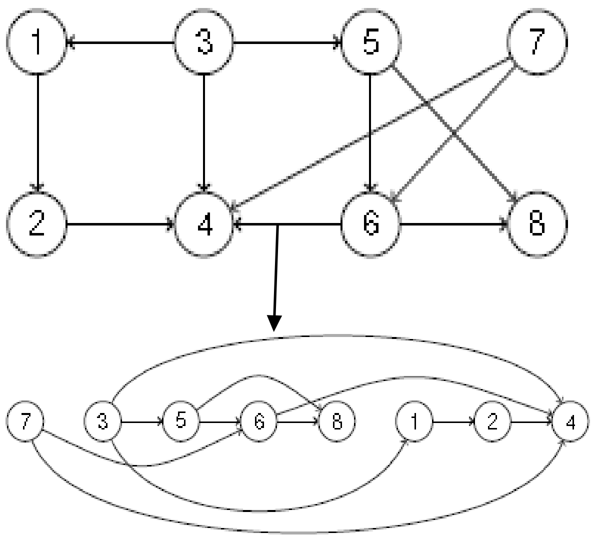
Department of IS&E, Jain Institute of Technology

Abstract

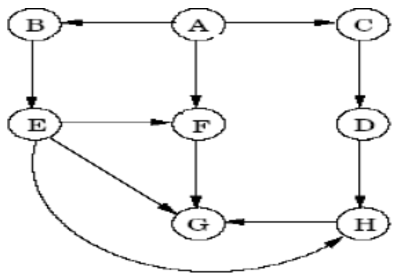
The paper discusses topological sorting, beginning with a fundamental definition and various implementation examples. It provides an in-depth explanation of the algorithm and demonstrates it through different datasets. Various scenarios showcasing the application of these algorithms are examined, followed by a thorough analysis to highlight differences in the results achieved. The paper evaluates three distinct algorithms for performing topological sorts, detailing their respective advantages and disadvantages. It includes examples from other research to illustrate the differences in sorting techniques. Lastly, the paper reviews the concept of topological sorting and explores its practical uses.

Keywords– DFS (Depth for Search),SRL (Source Removal Algorithm)

1 INTRODUCTION



Topological sorting is a method used to order the vertices of a graph in a linear sequence that reflects their dependencies. Specifically, if there is a directed edge from vertex a to vertex b, then a must appear before b in the ordering. This concept is akin to scheduling tasks where some tasks need to be completed before others, similar to the requirement of completing prerequisite courses before enrolling in advanced ones. To implement topological sorting, the graph must be a Directed Acyclic Graph (DAG), meaning it should be directed and contain no cycles. A DAG is characterized by the absence of cycles, ensuring that such a linear ordering is feasible.



DAGs are useful in modeling scenarios that involve prerequisite constraints.

Properties of DAGs:

- Source and Sink Vertices: In any Directed Acyclic Graph (DAG), there must be at least one vertex with no incoming edges (in-degree zero), known as a source vertex, and at least one vertex with no outgoing edges (out-degree zero), known as a sink vertex.
- Strongly Connected Components: A directed graph GGG is a DAG if and only if each vertex in GGG forms its own strongly connected component. This implies that no vertex in a DAG can be reached from any other vertex in a cyclic manner.
- DFS Finishing Times: In a DAG, for any edge

(v,w)(v, w)(v,w), the finishing time of vertex www is always less than the finishing time of vertex v in a Depth-First Search (DFS) traversal. This ordering is crucial for topological sorting.

- Topological Sort Definition: A topological sort of a directed graph G=(V,E)G = (V, E)G=(V,E) is an ordering of the vertices such that for every directed edge (u,v)(u, v)(u,v) in the graph, vertex uuu appears before vertex v in the ordering.

2 LITERATURE REVIEW

Topological sorting is most commonly applied to DAGs, which are graphs with directed edges and no cycles. The classic definition involves a linear ordering of vertices that respects the direction of edges.

In[1]Kahn, A.B. (1962). "Topological Sorting of Large Networks." *Communications of the ACM*, 5(11), 558-562.Kahn introduced the concept of topological sorting and an algorithm for achieving it. His method is based on repeatedly removing nodes with no incoming edges.

In[2]Tarjan, R.E. (1972). "Depth-First Search and Linear Graph Algorithms.". *SIAM Journal on Computing*, 1(2), 146-160.Tarjan extended Kahn's work and presented an efficient algorithm for topological sorting using depth-first search (DFS).

In[3]Topological sorting is crucial for scheduling tasks where certain tasks must precede others.Reference: Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press

In[4]In compiler design, topological sorting is used to determine the order of code generation and instruction scheduling.Reference: Aho, A.V., Lam, M.S., Sethi, R., & Ullman, J.D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

In[5]Communications of the *ACM*, 5(11), 558-562.Researchers have explored various improvements to these algorithms, including methods for handling specific types of graphs and optimizing performance for practical applications.Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

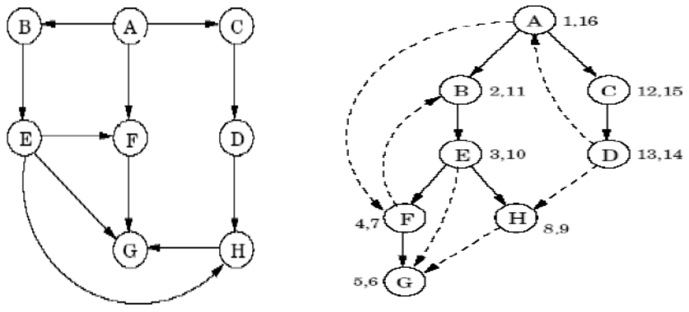
In[6]Topological sorting is crucial for scheduling tasks where certain tasks must precede others.Reference: Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

In[7] compiler design, topological sorting is used to determine the order of code generation and instruction scheduling.Reference: Aho, A.V., Lam, M.S., Sethi, R., & Ullman, J.D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

Dependency Resolution

- Topological sorting helps in resolving dependencies in software systems and package management.Reference: S. H. & S. H. (2022). "A Survey of Dependency Resolution Techniques." *Software: Practice and Experience*, 52(2), 150-174.

- While the basic algorithms are efficient, large-scale applications may face issues related to memory and processing constraints.
- Reference: Cohen, J., & Goldstein, S. (2015). "Efficient Algorithms for Large-Scale Topological Sorting." *Journal of Computer and System Sciences*, 81(4), 683-702.



3.RESEARCH METHODOLOGY

3.1 DEPTH-FIRST SEARCH

Depth-first search (DFS) is a traversal technique where the algorithm explores a graph by first visiting a node's child before proceeding further down the graph until it reaches a terminal node. This approach, as described by Cormen and colleagues, involves returning to the most recently visited node that has not yet been fully explored. In this process, nodes that have been visited are pushed onto a stack for subsequent examination. DFS focuses on nodes that have edges directed towards them, rather than away. As the algorithm progresses, it follows a pattern that ensures nodes already explored are not revisited, thus facilitating the topological sorting process.

```
Topological-Sort(G)[1]
{
• CalldfsAllVerticesonGto compute
  f[v]foreachvertexv
  • IfGcontainsabackedge(v,w)(i.e.,iff[w]>f[v]),r
  eporterror ;
  • e
  lse,aseachvertexisfinishedprependittoalist; //orpushinstack
• Returnthelist; //listisavalidtopologicalsort
}
```

RunningtimeisO(V+E), whichistherunningtimeforDFS.

Topologicalorder :A C D BEH F G
Inthisalgorithmeachnodeandedgearevisitedonly once,thereforethe algorithmexistsinlinear time.
Advantages**:**

- Memory requirement is linear with respect to searchgraph.
- Time complexity for this algorithm is O(b^d).
- DFS istimelimited.
- IfDFSfindsasolutionwithoutexploringmuchduring its path then it would consume very less timeand space.

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. The algorithm uses a stack to keep track of the nodes to visit next.

Here's a step-by-step breakdown of the code:

Initialization:

- graph[MAX_NODES][MAX_NODES]: This is a 2D array used to represent the adjacency matrix of the graph, where graph[i][j] is 1 if there is an edge between node i and node j, and 0 otherwise.
- visited[MAX_NODES]: This is a boolean array used to keep track of which nodes have been visited during the DFS traversal.
- numNodes = 6: This specifies the total number of nodes in the graph.

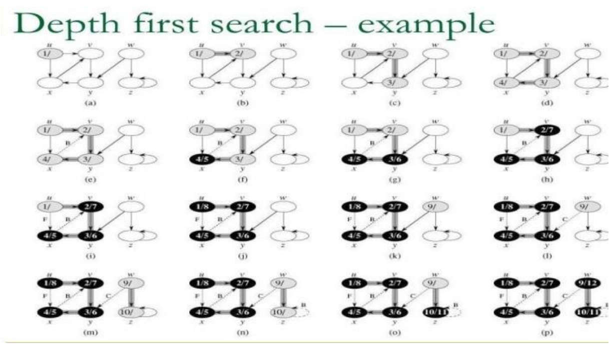
DFS Function:

- void dfs(int v): This function performs a depth-first search starting from the node v.
- Mark the node as visited: visited[v] = true;
- print the current node: printf("%d ", v);
- Explore adjacent nodes: For each node next from 0 to num Nodes - 1, if graph[v][next] is 1 (indicating an edge from node v to next) and next has not been visited (!visited[next]), recursively call dfs(next) to continue the traversal.

Main Function:

- Initialize the visited array: Set all elements to false, indicating that no nodes have been visited yet.
- Start DFS: Call dfs(0); to begin the depth-first search starting from node 0.

This DFS implementation assumes the graph is represented as an adjacency matrix and performs a recursive depth-first search starting from a given node (in this case, node 0). It prints the nodes in the order they are visited.



3.2 SOURCE REMOVAL ALGORITHM

Themaintechniqueusedinthisalgorithmistorepeatedlyidentifyandremoveasourceandallthecorrespondingedgesassociated to it..
The algorithm follows three simple steps:

- Pick a vertex and output it
- Remove the source and all the edges associated with it
- Keep repeating till the graph is empty

The algorithm is implemented in such a manner that it visits all the vertices in a topological sort manner. This algorithm usually uses an array to record the in-degree vertices. As a result there is no explicit need to delete vertices and edges.The algorithm uses a priority queue to record vertices with-in degree zero that hasn't been visited yet.

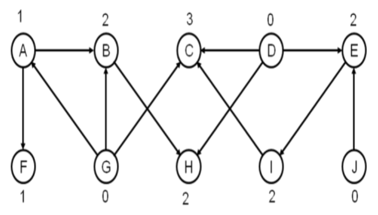
Algorithm goes as follows:

1 Initialize:

- Set num Visited Vertices to 0.

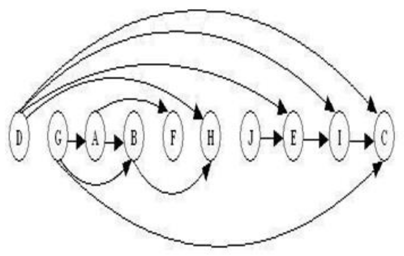
2 Repeat until there are no more vertices to visit:

- Check for vertices: If there are no vertices with an in-degree of 0, exit the loop.
- Select and Process: Choose a vertex v with an in-degree of 0.
- Visit the Vertex: Perform any required operations for visiting v.
- Update Count: Increment num Visited Vertices by 1.
- Remove Vertex: Delete vertex v and all its outgoing edges from the graph.



D	G	A	B	F	H	J	E	I	C

2 Return: The total number of visited vertices, num Visited Vertices.



Topological Sort Algorithm
Efficiency
This algorithm operates with a time complexity of \(O(|V| + |E|) \), where \(|V|\) is the number of vertices and \(|E|\) is the number of edges.
Description
The algorithm processes vertices in an order that respects their dependencies, eventually producing a topologically sorted list. It works as follows:
1. Initialize:
- Create an empty list `L` to store the topological order.
- Create a set `S` to hold vertices with no incoming edges.
2. Setup:
- Add all vertices with an in-degree of 0 to the set `S`.
3. Processing Loop:
- While the set `S` is not empty:
- Remove a vertex `n` from `S`.
- Insert `n` into the list `L`.
- For each vertex `m` that has an edge directed from `n` to `m`:
Remove the edge from the graph.
- If `m` now has no other incoming edges, add `m` to the set `S`.
4. Cycle Detection:
- After processing, if there are still edges remaining in the graph, return an error indicating the presence of a cycle (topological sorting is not possible).- Otherwise, return the list `L` as the topologically sorted order of vertices.
Notes
- This algorithm is particularly suited for Directed Acyclic Graphs (DAGs).
- The final ordering of vertices in `L` might vary depending on the order of vertex removal from set `S`.
- If the graph contains cycles, topological sorting cannot be completed.

By following these steps, the algorithm efficiently generates a valid topological sort or identifies if sorting is not feasible due to cycles in the graph.

Setup

1. Initialization**:**

- Nodes**:** Suppose we have the following nodes: 7, 5, 3, 11, 8, 2, 9, 10.
- Identify Nodes with No Incoming Edges: First, find all nodes with no incoming edges and add them to a set S.
- List Initialization**:** Create an empty list L that will eventually hold the sorted nodes.

Strategies for Node Ordering

Depending on the criteria used to choose the next node from set S, different topological sorts may result:

1. Smallest Numbered Node First**:**
   - Order nodes based on the smallest numerical value available.
   - Example Result: 3, 5, 7, 8, 11, 2, 9, 10.
2. Smallest Numbered Node First with a Modified Order**:**
   - Select nodes based on the smallest available number, but the sequence may differ based on edge removal order.
   - Example Result: 3, 7, 8, 5, 11, 10, 2, 9.
3. Fewest Edges First**:**
   - Prefer nodes with the fewest outgoing edges when multiple choices are available.
   - Example Result: 5, 7, 3, 8, 11, 10, 9, 2.
4. Largest Numbered Node First**:**
   - Choose nodes with the highest numerical value available.
   - Example Result: 7, 5, 11, 3, 10, 8, 9, 2.
5. Alternative Order with a Different Priority**:**
   - Choose nodes based on a different set of criteria or priority.
   - Example Result: 7, 5, 11, 2, 3, 8, 9, 10.

Notes

Graph Requirement: Ensure that the graph is acyclic for a valid topological sort.

*Variation**:** The final order in list L can differ depending on the node selection strategy used and the initial graph structure.*

ANALYZING TOPOLOGICALSORT

A topological sort of a Directed Acyclic Graph (DAG) has a noteworthy characteristic: if every pair of consecutive vertices in the sorted sequence is connected by a directed edge, the sequence often represents a directed Hamiltonian path. This characteristic implies:

1. Uniqueness of Topological Sort:
   - When the sorted sequence of vertices forms a directed Hamiltonian path (i.e., each consecutive vertex pair in the sort has a direct edge between them), it indicates that the topological sorting is unique.
   - The directed Hamiltonian path ensures there is only one way to order the vertices while respecting all dependency constraints, leading to a single, unique topological ordering.

2. Multiple Topological Orders:
   - If the topological sort does not create a directed Hamiltonian path (i.e., not every pair of consecutive vertices in the sort is connected by an edge), this suggests that the graph may have more than one valid topological ordering.
   - The absence of a Hamiltonian path means there are alternative ways to arrange the vertices while still satisfying the graph's ordering requirements.

APPLICATIONS

 Topological Sorting in Planning and Scheduling:

- Topological sorting is employed in various planning and scheduling tasks to determine a feasible order of operations or events based on their dependencies.

 Application in Defect Identification:

- With some adjustments, topological sorting can be utilized to uncover defects or issues within processes or systems.

 Cycle Detection in Graphs:

This method is a powerful tool for detecting cycles within a graph, which can help determine whether a graph is acyclic or not.

 Error Detection in DNA Sequencing:

Topological sorting is frequently applied to identify errors in the assembly of DNA fragments, aiding in the accurate reconstruction of genetic sequences.

CONCLUSION

This paper provides a comprehensive overview of topological sorting, starting with its definition and moving on to explore various implementations. It discusses the advantages of different algorithms used for topological sorting and provides detailed examples to illustrate their application. The paper contrasts cyclic and acyclic graphs, examining how the presence of cycles affects the analysis of each algorithm. Additionally, it addresses the time and space complexities associated with topological sorting. Towards the end, the paper draws a comparison between sequences of consecutively connected vertices and Hamiltonian paths. It concludes by discussing the diverse applications and implementations of topological sorting across various domains.

ACKNOWLEDGMENT

This research paper was completed with the assistance and support of many individuals. I would like to extend my sincere gratitude to Assistant Professor Sarswati for her unwavering support and encouragement, as well as her insightful guidance on the organization and presentation of the paper.

REFERENCES

1.http://www.cs.sunysb.edu/~algorith/files/topological-sorting.shtml

2.http://en.wikipedia.org/wiki/Topological_sorting

3.http://www.cs.nott.ac.uk/~nza/G5BADS06/lecture18.pdf

4.http://www.cs.nott.ac.uk/~nza/G5BADS06/lecture18.pdf

5.http://www.cse.cuhk.edu.hk/~taoyf/course/2100sum11/lec14.pdf