Topological Sorting

Khritish Kumar Behera

July 18, 2024

Introduction

Topological sorting addresses a common scenario where tasks have dependencies. It involves creating a linear ordering of these tasks such that any prerequisite task is completed before its dependent task. This ordering helps ensure a proper execution flow.

Definition

Topological sorting refers to a linear ordering of vertices in a Directed Acyclic Graph (DAG). In this ordering:

- A vertex u precedes vertex v only if there exists a directed edge from u to v (denoted as $u \rightarrow v$)

Crucially, the graph must be both directed and acyclic:

- **Directed:** Bidirectional edges (cycles) render a linear ordering meaningless, as there's no inherent order of execution for connected nodes
- **Acyclic:** Cycles (e.g., $u \rightarrow v \rightarrow c \rightarrow u$) create an impossible ordering. In such a cycle, u precedes v, both u and v precede c, and all three precede u again, leading to a logical contradiction.

Algorithm (DFS-based)

While topological sorting might appear complex, it can be achieved using Depth-First Search (DFS). Here's the algorithm breakdown:

Initialization:

1. Create an empty visited array to track visited nodes
2. Create an empty stack

Processing:

1. Iterate through each node in the graph
2. If the node is not visited (unprocessed), perform a DFS traversal starting from that node
3. *During DFS traversal:* Recursively visit each unvisited child/neighbor of the current node
4. Once all child nodes have been visited, push the current node onto the stack

Output:

1. After processing all nodes, the stack will contain the topological sorting in Last-In-First-Out (LIFO) order. Popping elements from the stack provides the desired ordering

This approach leverages the inherent property of DFS: nodes are explored completely before backtracking and visiting their children. This ensures that all prerequisites of a node are processed before the node itself is pushed onto the stack, resulting in a valid topological ordering.

Psudocode (DFS based)

Assume there is a DAG, with n node (0 to $n-1$), where the directed edges are stored in an adjacency list (ADJ_LIST)

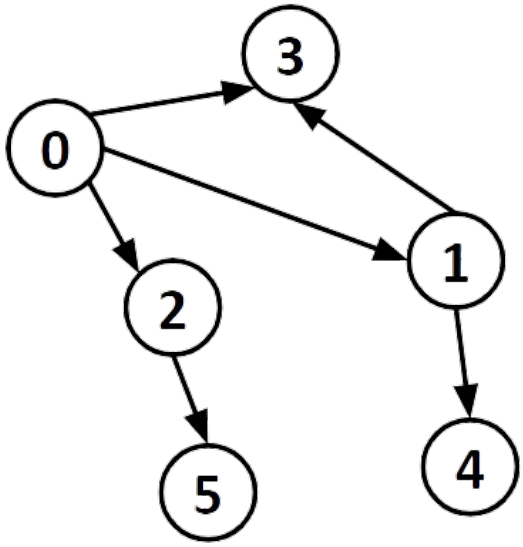
```
Code:
visited = []
stack = []
DFS(node):
    if node not in visited:
        add node to visited
        next_nodes = ADJ_LIST(node)
        for next_node in next_nodes:
            DFS(next_node)
        stack.push(node)
for node from 0 to n - 1:
    DFS(node)

topo_sort = []
while stack:
     topo_sort += stack.pop()
return topo_sort
```

Explanation of Pseudocode

- The 'DFS' function takes a node as input and performs a DFS traversal
- It marks the current node as visited and recursively visits all its unvisited neighbors
- Once all neighbors are visited, the current node is pushed onto the stack
- The 'TopologicalSort' function iterates through all nodes in the graph

- If a node is unvisited, it initiates a DFS traversal from that node
- After all nodes are processed, the stack will contain the topological sorting in reverse order (due to LIFO)
- The elements are popped from the stack and appended to the 'topological_sort' list, providing the desired ordering

Example: Dry run




```
DFS(0) -> DFS(1) -> DFS(4) -> x
                              stack.push(4)
                    DFS(3) -> x
                    stack.push(3)
                    stack.push(1)
       -> DFS(2) -> DFS(5) -> x
                              stack.push(5)
                    stack.push(2)
       -> DFS(3) -> already visited
       stack.push(0)
stack = [4, 3, 1, 5, 20] (head)
(performing LIFO)
topo_sort = [0, 2, 5, 1, 3, 4]
```

Algorithm (BFS-based): Kahn's Algorithm

This approach leverages the concept of in-degree, which refers to the number of incoming directed edges to a node. A node with an in-degree of 0 has no dependencies on other nodes and can be processed immediately.

Note:
In-degree is the number of incoming directed edge to a node.

Algorithm Breakdown:
Initialization

- Create an empty queue (FIFO) to store nodes for processing.
- Create a hash set or dictionary to store the in-degree of each node.

Processing

1. Iterate through all edges in the graph and update the in-degree of the destination node for each edge (increment by 1)
2. Add all nodes with an in-degree of 0 to the queue. These nodes have no dependencies and can be processed first
3. While the queue is not empty:
   - Dequeue a node from the queue
   - Remove all outgoing edges from the dequeued node (edges pointing to other nodes)
   - For each neighbor (destination node) of the dequeued node:
     - Decrement the in-degree of the neighbor in the in-degree hash set.
     - If the neighbor's in-degree becomes 0, add it to the queue.
   - Add the dequeued node to the final topological sort array.

Psudocode (BFS based)

```
COMPUTE_IN_DEGREE(ADJ_LIST):
    indegree = {0}
    For node, connected_nodes in ADJ_LIST.items()
        For child_node in connected_nodes:
            indegree[child_node] += 1
    return indegree

BFS(indegree, ADJ_LIST):
    queue = []
    topo_sort = []
    For node, in_deg in indegree.items():
        if in_deg == 0:
            queue.push(node)
    while queue not empty:
        popped_node = queue.pop()
        connected_nodes = ADJ_LIST(popped_node)
        For child_node in connected_nodes:
            indegree[child_node] -= 1
            if indegree[child_node] == 0:
                queue.push(child_node)
        topo_sort.append(popped_node)
    return topo_sort

KAHNS_ALGORITHM(ADJ_LIST):
    indegree =  COMPUTE_IN_DEGREE(ADJ_LIST)
    return BFS(indegree, ADJ_LIST)
```

Cycle Detection in Directed Graphs (Using Kahn's Algorithm)

Kahn's algorithm can be effectively used to detect cycles in directed acyclic graphs (DAGs). However, it doesn't directly search for cycles. Instead, it leverages the property that a DAG has a topological ordering where each node's predecessors (nodes with incoming edges) appear before it in the ordering.

Detection via Topological Ordering Length

Here's the logic:

1. **Run Kahn's Algorithm:** Apply Kahn's algorithm to compute a topological ordering of the graph
2. **Check Ordering Length:** If the topological ordering list's length (‘| L|‘) is equal to the total number of nodes (‘|V|‘) in the graph, it implies no cycles exist. All nodes were processed and included in the ordering, signifying no remaining dependencies
3. **Cycle Detected:** However, if the topological ordering list's length is less than the total number of nodes ($|L| < |V|$), it indicates the presence of one or more cycles. Some nodes in the cycle have dependencies that prevent them from reaching an in-degree of zero, thus never being processed by Kahn's algorithm and not appearing in the topological ordering

This approach efficiently detects cycles by leveraging the topological properties of DAGs.

Alternative Approaches

- **Depth-First Search (DFS):** DFS specifically searches for back edges (edges pointing from a visited node back to an ancestor), which indicate cycles
- **Union-Find Data Structure:** This approach can be used to identify cycles by checking if merging nodes connected by an edge leads to a cycle (same root)

While Kahn's algorithm with topological ordering length check is a good technique, alternative methods like DFS or Union-Find may be more suitable depending on the specific application and performance requirements.