

AN  
ASSIGNMENT  
ON  
**TOPOLOGICAL SORT**

*Submitted in partial fulfillment of the requirements for the degree of*

**Bachelor of Technology**  
In  
**Information Technology**

*By*

Atharva Mundke (2254491246027)

Under the guidance  
of  
**Prof. Rubi Mandal**



SHRI VILE PARLE KELAWANI MANDAL'S

**INSTITUTE OF TECHNOLOGY, DHULE**

Survey No. 499, Plot No. 02, Behind Gurudwara, Mumbai-Agra National Highway, Dhule-  
424001, Maharashtra, India.

**Academic Year 2023-24**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

# Topological Sort

Topological sorting is a crucial concept in graph theory, particularly applicable to directed acyclic graphs (DAGs). It involves arranging the vertices of a graph in a linear order such that for every directed edge  $u \rightarrow v$ , vertex  $u$  precedes vertex  $v$  in the ordering. This ensures that all dependencies are satisfied, making it indispensable in fields such as task scheduling and dependency resolution.

The significance of topological sorting is evident in its wide range of applications. In project management, it facilitates scheduling tasks with dependencies, ensuring that dependent tasks are completed before their dependents can start. Similarly, in software development, dependency resolution systems, such as package managers, rely on topological sorting to resolve dependencies efficiently.

An algorithmic approach is commonly employed to perform topological sorting. The algorithm operates greedily, starting by identifying vertices with zero indegree (no predecessors) and systematically removing them from the graph. This process continues iteratively until no vertices with zero indegree remain or until it's not feasible to remove any more vertices.

Implementation of the algorithm typically involves tracking the indegrees of vertices and utilizing a queue to store vertices with zero indegree. Vertices are removed from the graph, their successors' indegrees are updated, and the process repeats until all vertices are sorted. Termination conditions, such as the presence of cycles in the graph, are identified to ensure the algorithm's correctness.

An illustrative example demonstrates the step-by-step process of topological sorting, highlighting the computation of indegrees, removal of vertices, and the identification of cycles. Through the below solved example, the practical utility and effectiveness of topological sorting in resolving dependencies and scheduling tasks become apparent, solidifying its foundational importance in computer science and beyond.

### Algorithm:

In Topology Sort Queue is used.

Find Indegrees of all the vertices.

**Step 1:** The Vertices with zero (minimum) indegree can be inserted into the queue in any order.

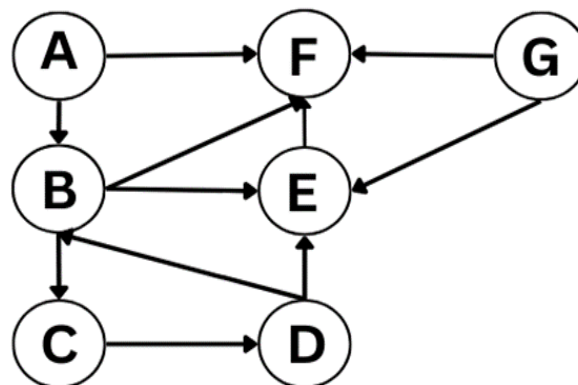
**Step 2:** Delete the front vertex from the queue. Delete that vertex and edges going from that vertex in graph. Update the indegree of all the vertices.

**Step 3:** Repeat step 2 until the queue becomes empty. Now look for the vertices whose updated indegree is zero (minimum). Insert such vertices into the queue.

**Step 4:** Repeat step 1, 2 and 3 until the graph becomes null. i.e. no vertices left in the graph.

### Example:

Sort the following graph using topology sort.



**Ans:** Indegree of all vertices are as follows:

$$\text{In}(A) = 0$$

$$\text{In}(B) = 3$$

$$\text{In}(C) = 0$$

$$\text{In}(D) = 1$$

$$\text{In}(E) = 3$$

$$\text{In}(F) = 4$$

$$\text{In}(G) = 0$$

From all the vertices, vertex A, C and G have zero indegree Hence, insert them into the queue.

Queue: A, C, G.

topo\_order : Empty.

**Step 1:** Delete the vertex A and the edges going from the vertex A.

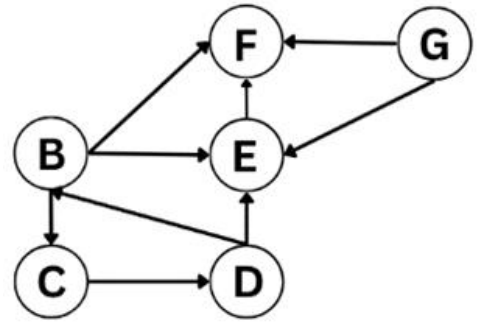
Queue: C, G.

topo\_order: A

Updated Indegree of vertices:  $\text{In}(B) = 2$ ,

$\text{In}(C) = 0$ ,  $\text{In}(D) = 1$ ,  $\text{In}(E) = 3$ ,  $\text{In}(F) = 3$ ,

$\text{In}(G) = 0$ .



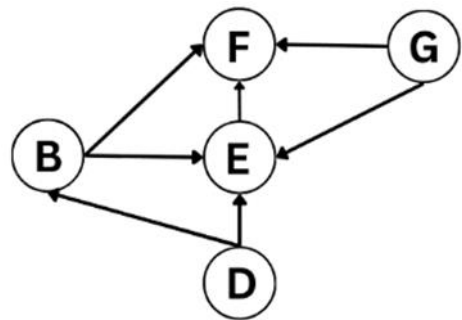
**Step 2:** Delete the vertex C and the edges going from the vertex C.

Queue: G.

topo\_order: A, C

Updated Indegree of vertices:  $\text{In}(B) = 1$ ,

$\text{In}(D) = 0$ ,  $\text{In}(E) = 3$ ,  $\text{In}(F) = 3$ ,  $\text{In}(G) = 0$ .



**Step 3:** Delete the vertex G and the edges going from the vertex G.

Queue: Empty.

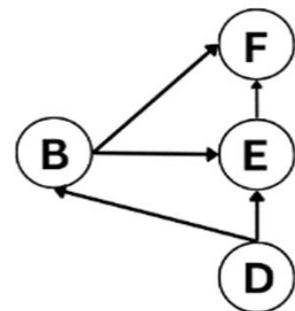
topo\_order: A, C, G

Updated Indegree of vertices:  $\text{In}(B) = 1$ ,  $\text{In}(D) = 0$ ,

$\text{In}(E) = 2$ ,  $\text{In}(F) = 2$ .

Insert the vertices with minimum indegree into queue.

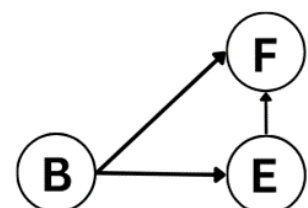
Queue: D



**Step 4:** Delete the vertex D and the edges going from the vertex D.

Queue: Empty.

topo\_order: A, C, G, D



Updated Indegree of vertices:  $\text{In}(B) = 0$ ,  $\text{In}(E) = 1$ ,  $\text{In}(F) = 2$ .

Insert the vertices with minimum indegree into queue.

Queue: B

**Step 5:** Delete the vertex B and the edges going from the vertex B.

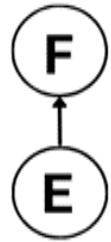
Queue: Empty.

topo\_order: A, C, G, D, B

Updated Indegree of vertices:  $\text{In}(E) = 0$ ,  $\text{In}(F) = 1$ .

Insert the vertices with minimum indegree into queue.

Queue: E



**Step 6:** Delete the vertex E and the edges going from the vertex E.

Queue: Empty.

topo\_order: A, C, G, D, B, E

Updated Indegree of vertices:  $\text{In}(F) = 0$ .

Insert the vertices with minimum indegree into queue.

Queue: F



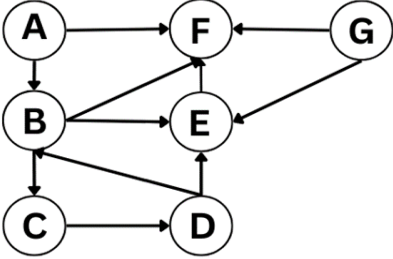
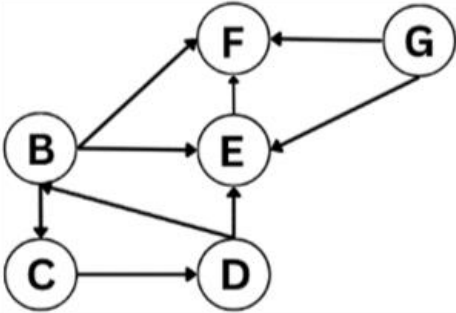
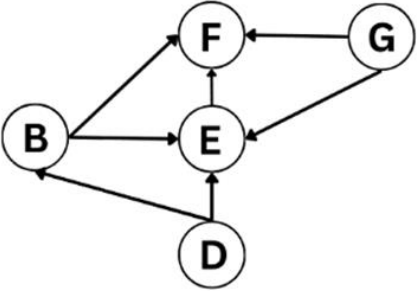
**Step 7:** Delete the vertex F and the edges going from the vertex F.

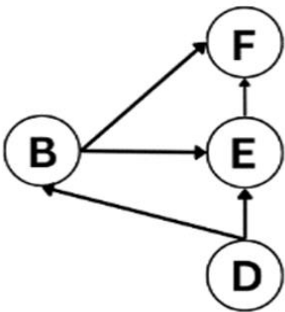
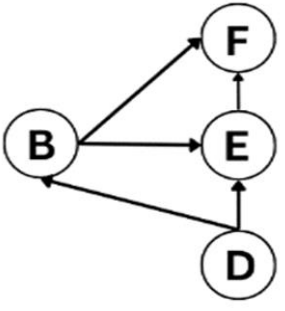
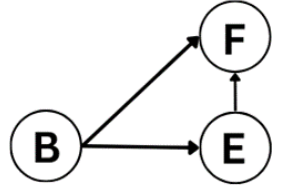
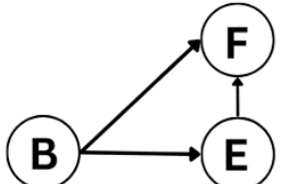
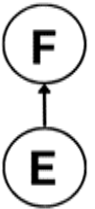
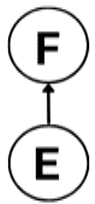
Queue: Empty.

topo\_order: A, C, G, D, B, E, F

Updated Indegree of vertices: Empty

Table 1. Solving Topological Sort

Topological order	Deleted vertex	Updated graph	Updated indegree	Queue
Empty	Empty		$\text{In}(A)=0$ $\text{In}(B)=3$ $\text{In}(C)=0$ $\text{In}(D)=1$ $\text{In}(E)=3$ $\text{In}(F)=4$ $\text{In}(G)=0$	A, C, G
Empty	A		$\text{In}(B)=2$ , $\text{In}(C)=0$ , $\text{In}(D)=1$ , $\text{In}(E)=3$ , $\text{In}(F)=3$ , $\text{In}(G)=0$	C, G
A	C		$\text{In}(B)=1$ , $\text{In}(D)=0$ , $\text{In}(E)=3$ , $\text{In}(F)=3$ , $\text{In}(G)=0$	G

A, C	G		$\text{In}(B) = 1,$ $\text{In}(D) = 0,$ $\text{In}(E) = 2,$ $\text{In}(F) = 2.$	Empty
A, C, G	Empty		$\text{In}(B) = 1,$ $\text{In}(D) = 0,$ $\text{In}(E) = 2,$ $\text{In}(F) = 2$	D
A, C, G	D		$\text{In}(B) = 0,$ $\text{In}(E) = 1,$ $\text{In}(F) = 2.$	Empty
A, C, G, D	Empty		$\text{In}(B) = 0,$ $\text{In}(E) = 1,$ $\text{In}(F) = 2.$	B
A, C, G, D	B		$\text{In}(E) = 0,$ $\text{In}(F) = 1.$	Empty
A, C, G, D, B	Empty		$\text{In}(E) = 0,$ $\text{In}(F) = 1.$	E

A, C, G, D, B	E	<b>F</b>	$\text{In}(F) = 0.$	Empty
A, C, G, D, B, E	Empty	<b>F</b>	$\text{In}(F) = 0.$	F
A, C, G, D, B, E	F	Empty	Empty	Empty
A, C, G, D, B, E, F	Empty	Empty	Empty	Empty

**We have no more vertices in the graph, so the topological sorting of graph will be A, C, G, D, B, E, F.**



## Program for Topological Sorting:

```
#include <iostream>
using namespace std;

#define MAX 100 // Maximum number of vertices in the graph

// Global variables
int n; // Number of vertices in the graph
int adj[MAX][MAX]; // Adjacency Matrix to represent the graph
int front = -1, rear = -1; // Initialize front and rear pointers for queue
int queue[MAX]; // Queue to perform topological sorting

// Function prototypes
void create_graph(); // Function to create the graph
void insert_queue(int vertex); // Function to insert a vertex into the queue
int delete_queue(); // Function to delete a vertex from the queue
int isEmpty_queue(); // Function to check if the queue is empty
int indegree(int v); // Function to calculate the indegree of a vertex

int main() {
    int i, v, count, topo_order[MAX], indeg[MAX];
    create_graph(); // Create the graph by taking input from the user

    // Find the indegree of each vertex
    for (i = 0; i < n; ++i) {
        indeg[i] = indegree(i); // Calculate the indegree of vertex i
        if (indeg[i] == 0)
            insert_queue(i); // If indegree is 0, insert vertex into the queue
    }

    count = 0; // Initialize count of processed vertices
    // Perform topological sorting using queue
    while (!isEmpty_queue() && count < n) {
        v = delete_queue(); // Delete a vertex from the queue
        topo_order[++count] = v; // Add vertex v to topo_order array

        // Delete all edges going from vertex v
        for (i = 0; i < n; ++i) {
            if (adj[v][i] == 1) { // If there is an edge from v to i
                adj[v][i] = 0; // Remove the edge
                indeg[i] = indeg[i] - 1; // Decrement the indegree of vertex i
                if (indeg[i] == 0)
                    insert_queue(i); // If indegree becomes 0, insert i into
the queue
            }
        }
    }
}
```

```

    }

    // If count is less than n, graph contains cycle
    if (count < n) {
        cout << "No topological ordering possible, graph contains cycle" <<
endl;
        exit(1); // Exit the program with error code 1
    }

    // Print the vertices in topological order
    cout << "Vertices in topological order are: ";
    for (i = 1; i <= count; ++i)
        cout << topo_order[i] << " "; // Print each vertex in topo_order array
    cout << endl;

    return 0; // Exit the program successfully
}

// Function to insert a vertex into the queue
void insert_queue(int vertex) {
    if (rear == MAX - 1) { // If queue is full
        cout << "Queue Overflow" << endl;
        return; // Exit the function
    }
    if (front == -1) { // If queue is initially empty
        front = 0; // Set front to 0
    }
    rear = rear + 1; // Increment rear pointer
    queue[rear] = vertex; // Insert vertex into the queue
}

// Function to check if the queue is empty
int isEmpty_queue() {
    if (front == -1 || front > rear) { // If front pointer is invalid or
greater than rear pointer
        return 1; // Queue is empty
    } else {
        return 0; // Queue is not empty
    }
}

// Function to delete a vertex from the queue
int delete_queue() {
    if (front == -1 || front > rear) { // If queue is empty
        cout << "Queue Underflow" << endl;
        exit(1); // Exit the program with error code 1
    }
}

```

```

    int del_item = queue[front]; // Get the vertex to be deleted
    front = front + 1; // Move front pointer
    return del_item; // Return the deleted vertex
}

// Function to calculate the indegree of a vertex
int indegree(int v) {
    int in_deg = 0; // Initialize indegree counter
    for (int i = 0; i < n; ++i) {
        if (adj[i][v] == 1) { // If there is an edge from i to v
            in_deg++; // Increment indegree counter
        }
    }
    return in_deg; // Return the calculated indegree
}

// Function to create the graph
void create_graph() {
    int max_edges, origin, destin;
    cout << "Enter number of nodes: ";
    cin >> n; // Input number of vertices
    max_edges = n * (n - 1); // Maximum number of edges in a directed graph

    // Input the edges of the graph
    for (int i = 1; i <= max_edges; ++i) {
        cout << "Enter edge " << i << " (-1 -1 to quit): ";
        cin >> origin >> destin; // Input origin and destination of edge

        if (origin == -1 && destin == -1) { // If user wants to stop entering
edges
            break; // Exit the loop
        }

        // Check if the input edge is valid
        if (origin >= n || destin >= n || origin < 0 || destin < 0) {
            cout << "Invalid edge!" << endl;
        } else {
            adj[origin][destin] = 1; // Set the edge in the adjacency matrix
        }
    }
}

```

This C++ program implements a topological sorting algorithm for a directed graph using Kahn's algorithm. Let's break down the code step by step:

### **Libraries:**

- `#include<iostream>`: This header file allows input and output operations.
- `using namespace std;` : This line enables the usage of names from the standard C++ library such as 'cout' and 'cin', without explicitly qualifying them.

### **Macros and Global Variables:**

- `#define MAX 100`: This macro defines the maximum size for the adjacency matrix and the queue.
- `int n;` : This variable holds the number of nodes in the graph.
- `int adj[MAX][MAX];` : This is the adjacency matrix representing the directed graph.
- `int front = -1, rear = -1;` : These variables represent the front and rear pointers for the queue.
- `int queue[MAX];` : This array represents the queue used in the algorithm.

### **Function Declarations:**

- `void create_graph();` : Function prototype to create the directed graph by taking input from the user.
- `void insert_queue(int vertex);` : Function prototype to insert a vertex into the queue.
- `int delete_queue();` : Function prototype to delete a vertex from the queue.
- `int isEmpty_queue();` : Function prototype to check if the queue is empty.
- `int indegree(int v);` : Function prototype to calculate the indegree of a vertex.
- `int main()` : The main function where the program execution begins.

### **Main Function:**

1. `create_graph()`: Calls the function to create the directed graph.
2. Initializes arrays 'indeg', 'topo\_order', and a counter 'count'.
3. Calculates indegrees of all vertices and adds vertices with indegree 0 to the queue.
4. While the queue is not empty, and 'count' is less than the number of vertices:
  - Dequeues a vertex 'v'.
  - Stores 'v' in 'topo\_order'.
  - Decrements indegree of adjacent vertices of 'v' and enqueues them if their indegree becomes 0.

5. If 'count' is less than 'n', prints a message indicating the presence of a cycle.
6. Prints the vertices in topological order.

**Explanation:**

- The program first creates a directed graph by taking input from the user, representing it using an adjacency matrix.
- Then, it calculates the indegree of each vertex and initializes a queue with vertices having indegree 0.
- The program processes vertices in a loop, dequeuing a vertex, updating the indegree of its adjacent vertices, and enqueueing them if their indegree becomes 0.
- If a topological ordering is not possible due to the presence of a cycle, it prints a message.
- Otherwise, it prints the vertices in topological order.

**Conclusion:**

The program demonstrates the implementation of Kahn's algorithm for topological sorting, a crucial operation in graph theory used in various applications like task scheduling and dependency resolution.

**Q. What is Kahn's algorithm, and how is it utilized in the context of topological sorting for directed graphs?**

**Ans:**

A topological sorting algorithm for a directed graph using Kahn's algorithm refers to the method of arranging the vertices of a directed graph in a linear order based on their dependencies, employing Kahn's algorithm. This algorithm provides a straightforward and efficient way to perform topological sorting, particularly suited for graphs with acyclic dependencies.

Here's an explanation of how Kahn's algorithm works:

1. Initialize:

- Begin by initializing an empty list to store the topologically sorted vertices.
- Compute the indegree (the number of incoming edges) for each vertex in the graph.

2. Identify Start Vertices:

- Enqueue all vertices with an indegree of zero into a queue.

3. Process Vertices:

- While the queue is not empty:
- Dequeue a vertex from the queue.
- Append the dequeued vertex to the sorted list.
- Reduce the indegree of its adjacent vertices by one.
- If any adjacent vertex's indegree becomes zero, enqueue it.

4. Cycle Detection:

- If the sorted list does not contain all vertices in the graph, it indicates the presence of a cycle, making topological sorting impossible.

5. Output:

- The sorted list obtained after processing all vertices represents the topological order of the graph.

Kahn's algorithm provides a linear-time complexity solution for topological sorting, making it efficient for large graphs. It effectively handles graphs with complex dependencies, ensuring that tasks or elements are ordered based on their prerequisites. This algorithm is widely used in various applications, including task scheduling, dependency resolution, and compiler optimizations. Kahn's algorithm for topological sorting is a widely-used method due to its simplicity and efficiency.

## **Analysis of time and space complexity of Topological Sort**

Time Complexity:

- The time complexity for constructing the graph is  $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges.
- The time complexity for performing topological sorting using BFS is also  $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges. This is because each vertex and each edge is visited once during the BFS traversal.

Space Complexity:

- The space complexity for storing the graph using an adjacency list is  $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges.
- Additional space is used for storing the in-degree of vertices, which requires  $O(V)$  space.
- A queue is used for BFS traversal, which can contain at most  $V$  vertices. Thus, the space complexity for the queue is  $O(V)$ .
- Overall, the space complexity of the algorithm is  $O(V + E)$  due to the storage of the graph, in-degree array, and the queue.
- In summary, the time complexity of the provided implementation is  $O(V + E)$ , and the space complexity is also  $O(V + E)$ .
- Note: Here, we can also use a array instead of the stack. If the array is used then print the elements in reverse order to get the topological sorting.

Advantages of Topological Sort:

- Helps in scheduling tasks or events based on dependencies.
- Detects cycles in a directed graph.

- Efficient for solving problems with precedence constraints.

#### Disadvantages of Topological Sort:

- Only applicable to directed acyclic graphs (DAGs), not suitable for cyclic graphs.
- May not be unique, multiple valid topological orderings can exist.
- Inefficient for large graphs with many nodes and edges.

#### Applications of Topological Sort:

- Task scheduling and project management.
- Dependency resolution in package management systems.
- Determining the order of compilation in software build systems.
- Deadlock detection in operating systems.
- Course scheduling in universities.