Table of Contents

[Topological Sort 1](#_Toc108090537)

[**Definition** 1](#_Toc108090538)

[**Examples** 1](#_Toc108090539)

[**Cycles Are Not Possible in Topological Ordering** 3](#_Toc108090540)

[**Algorithm** 3](#_Toc108090541)

[**Implementation** 6](#_Toc108090542)

[**Can We Do Better?** 7](#_Toc108090543)

[**Improved Algorithm** 7](#_Toc108090544)

# **Topological Sort**

## **Definition**

* A **topological sort** is an ordering of vertices in a ***directed acyclic graph (DAG)***.
* Given digraph *G* = (*V*, *E*), find a linear ordering of vertices such that for all edges (*v*, *w*) in *E*, *v* precedes *w* in the ordering
* Precisely, a topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited.

## **Examples**

**Example 1**

* The following graph in represents the course prerequisite structure at a Miami University.

Diagram

Description automatically generated

* A directed edge (*v*, *w*) indicates that course *v* must be completed before course *w* may be attempted. This means that *w* is a prerequisite to *v*.
* For example, MAC3311 and MAD2104 are prerequisites to and must be completed before MAD3512.
* A **topological ordering** of these courses is ***any course* *sequence*** that does not violate the prerequisite requirement.
  + This means that there may be more than one possible ordering.
  + Any legal ordering will do

**Example 2**

* Let’s say we want to find a topological ordering for *v6* the following graph
  + *v*1, *v*2, *v*5, *v*4, *v*3, *v*7, *v*6 is a possible topological ordering

and

* + *v*1, *v*2, *v*5, *v*4, *v*7, *v*3, *v*6 is a possible topological ordering

A picture containing old, watch

Description automatically generated

**Example 3**

* Let’s say we want to find a topological ordering for *plating food* in following graph
* Notice how any *w* comes after a *v*
  + Butter Toast comes after Toast Bread.
  + Plate Food transitively comes after Butter Toast.
  + etc.

Diagram

Description automatically generated

**Diagram

Description automatically generated**

## **Cycles Are Not Possible in Topological Ordering**

* A **topological ordering is not possible if the graph has a cycle**, since for two vertices *v* and *w* on the cycle, *v* precedes *w* and *w* precedes *v*.
* These two courses would be in a deadlock since you would need one to complete the other and is therefore impossible.

## **Algorithm**

* Here we describe a simple algorithm to find a topological ordering:

1. **Steph 1:** Identify vertices that have **no incoming edges**.

We define the **indegree** of a vertex *v* as the **number of incoming edges** (*u*, *v*).

A picture containing text, clock

Description automatically generated

The indegree of A and F are **zero**

Diagram

Description automatically generated  
If all vertices have incoming edges (no vertices with indegree 0), then the graph has at least one **cycle** (cyclic graph).

A picture containing text, clock

Description automatically generated

Compute the indegrees of all vertices in the graph.

Select any vertex with indegree 0.

1. **Step 2:** **Remove** **this vertex** of indegree 0 and all its **outgoing edges** from the graph.

A picture containing text, clock, watch

Description automatically generated

1. Place this vertex in the **output** or print it.
2. **Reduce** the indegree of all **adjacent vertices**.
3. **Step 3:** Repeat Step 1 and Step 2 **until the graph is empty**

A picture containing text, clock

Description automatically generated

Repeat Step 1, we identify that B and F have an indegree of 0

Repeat Step 2, we delete B and its outgoing edges, then place it in the output

A picture containing text, clock

Description automatically generated

Repeat Step 1, we identify that C and F have an indegree of 0

Repeat Step 2, we delete F and its outgoing edges, then place it in the output

Diagram

Description automatically generated

Repeat Step 1, we identify that C has an indegree of 0

Repeat Step 2, we delete C and its outgoing edges, then place it in the output

A picture containing clipart

Description automatically generated

Repeat for D and E

## **Implementation**

* **Variables**
  + **Indegree Array:** We store each vertex’s indegree (# of incoming edges) in an array.
  + **Adjacency List:** Read graph into an adjacency list, and each vertex is the head of a list which holds all edges adjacent to the vertex.
* **Algorithm**
  + Initialize the indegree array by storing each vertex’s indegree (# of incoming edges)
    - O(|E|)
  + Find a vertex with indegree 0 from the indegree array and output it
    - O(|V|)
  + Mark and output vertex
    - O(|V|)
  + Reduce indegree of all vertices adjacent to vertex
    - O(|E|)

A picture containing text, clock, clipart

Description automatically generated Diagram

Description automatically generated

Text

Description automatically generated with medium confidence

1. The function **findNewVertexOfIndegreeZero( )** isa simple sequential scan of the indegree array of *v* vertices.
   1. It looks for a vertex with indegree 0 that has not already been assigned a topological number.
   2. It returns null if no such vertex exists; this indicates that the graph has a cycle.
2. We assign the returned vertex a number according to its topological order.
3. We then reduce indegree of all vertices adjacent to vertex

* **Runtime**
  + Each call to findNewVertexOfIndegreeZero( )takes *O*(|*V*|) time.
  + Since there are |*V*| such calls to this function, the running time of the algorithm is *O*(|*V*|2).

## **Can We Do Better?**

* By paying more careful attention to the data structures, it is possible to do better.
* The cause of the poor running time is the sequential scan through the array of vertices with each call to findNewVertexOfIndegreeZero( )taking *O*(|*V*|) time.
* If the graph is sparse, we would expect that only a few vertices have their indegrees updated during each iteration.
* However, in the search for a vertex of indegree 0, we look at (potentially) all the vertices, even though only a few have changed.
* We can remove this inefficiency by keeping all the (unassigned) vertices of indegree 0 in a special *box.*
* The findNewVertexOfIndegreeZero method then returns (and removes) any vertex in the box.
* When we decrement the indegrees of the adjacent vertices, we check each vertex and place it in the box if its indegree falls to 0.

## **Improved Algorithm**

* As before, we will assume that the graph is already read into an adjacency list and that the indegrees are computed and stored with the vertices.
* We also assume each vertex has a field named topNum, in which to place its topological numbering.
* To implement the box, we can use either a stack or a queue; we will use a queue.

**Pseudocode**

1. First, the indegree is computed for every vertex and placed into an array.
2. All vertices of indegree 0 are placed on an initially empty queue.
3. If the queue is empty, then there is a cycle in the graph.
4. While the queue is not empty, a vertex *v* is removed, and all vertices adjacent to *v* have their indegrees decremented.
5. A vertex is put on the queue as soon as its indegree falls to 0.
6. **The topological ordering then is the order in which the vertices dequeue.**

A picture containing table

Description automatically generated

**Example**

* Figure 9.6 shows the status after each phase of the loop for Figure 9.4.

A picture containing old, watch

Description automatically generated

Table

Description automatically generated with medium confidence

**Runtime**

* The time to perform this algorithm is *O*(|*E*| + |*V*|) if adjacency lists are used.
* This is apparent when one realizes that the body of the for loop is executed at most once per edge.