Kahn's algorithm for Topological Sorting

Final Portfolio Piece

Haarika Sai Katlaparthi
NUID: 002964238

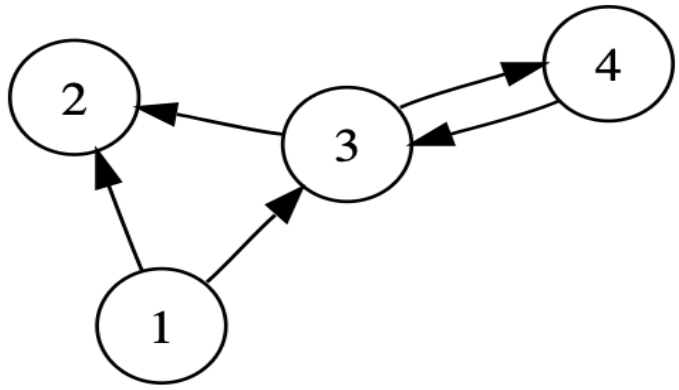
Introduction

Directed Graph:

In mathematics, and more specifically in graph theory, a directed graph (or digraph) is a graph that is made up of a set of vertices connected by directed edges, often called arcs.

In formal terms, a directed graph is an ordered pair G = (V, A) where:

- V is a set whose elements are called vertices, nodes, or points
- A is a set of ordered pairs of vertices, called arcs, directed edges (sometimes simply edges with the corresponding set named E instead of A), arrows, or directed lines.

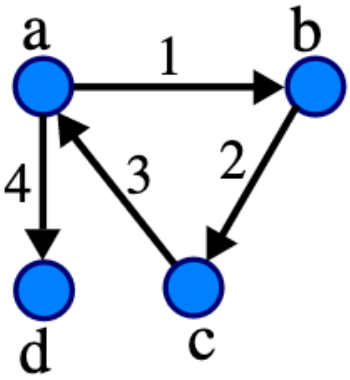


Basic Terminology

An arc (x, y) is considered to be directed from x to y; y is called the head, and x is called the tail of the arc; y is said to be a direct successor of x, and x is said to be a direct predecessor of y. If a path leads from x to y, then y is said to be a successor of x and reachable from x, and x is said to be a predecessor of y. The arc (y, x) is called the reversed arc of (x, y).

The adjacency matrix of a multidigraph with loops is the integer-valued matrix with rows and columns corresponding to the vertices, where a nondiagonal entry a_{ij} is the number of arcs from vertex i to vertex j, and the diagonal entry a_{ii} is the number of loops at vertex i. The adjacency matrix of a directed graph is a logical matrix and is unique up to the permutation of rows and columns.

Another matrix representation for a directed graph is its incidence matrix. Below is an example depicting a directed graph with its adjacency matrix.



	1	2	3	4
a	1	0	-1	1
b	-1	1	0	0
c	0	-1	1	0
d	0	0	0	-1

Indegree and Outdegree

For a vertex, the number of head ends adjacent to a vertex is called the indegree of the vertex and the number of the tail ends adjacent to a vertex is its outdegree (called branching factor in trees).

Let G = (V, A) and v ∈ V. The in-degree of v is denoted deg−(v) and its outdegree is denoted deg+(v).

A vertex with deg−(v) = 0 is called a source, as it is the origin of each of its outcoming arcs. Similarly, a vertex with deg+(v) = 0 is called a sink since it is the end of each of its incoming arcs.

The degree sum formula states that, for a directed graph,

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |A|.$$

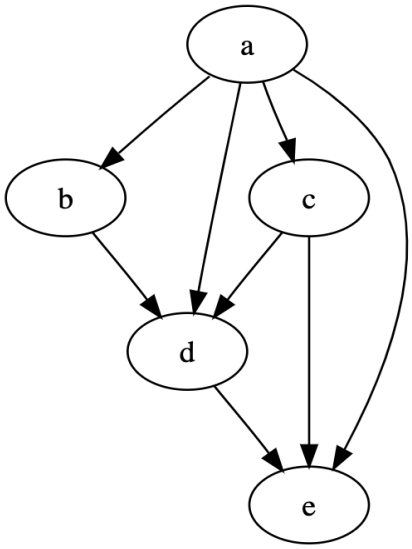
If for every vertex v ∈ V, deg+(v) = deg−(v), the graph is called a balanced directed graph.

Directed Acyclic Graph

In mathematics, particularly graph theory, and computer science, a **directed acyclic graph** (DAG) is a directed graph with no directed cycles. That is, it consists of vertices and edges (also called arcs), with each edge directed from one vertex to another, such that following those directions will never form a closed loop.

A directed graph is a DAG if and only if it can be topologically ordered, by arranging the vertices as a linear ordering that is consistent with all edge directions. DAGs have numerous scientific and computational applications, ranging from biology (evolution, family trees, epidemiology) to information science (citation networks) to computation (scheduling).

Directed acyclic graphs are sometimes instead called acyclic directed graphs or acyclic digraphs. Below is an example of a DAG.



A graph is formed by vertices and by edges connecting pairs of vertices, where the vertices can be any kind of object that is connected in pairs by edges. In the case of a directed graph, each edge has an orientation, from one vertex to another vertex.

A **path** in a directed graph is a sequence of edges having the property that the ending vertex of each edge in the sequence is the same as the starting vertex of the next edge in the sequence; a path forms a **cycle** if the starting vertex of its first edge equals the ending vertex of its last edge. A directed acyclic graph is a directed graph that has no cycles.

A vertex v of a directed graph is said to be **reachable** from another vertex u when there exists a path that starts at u and ends at v. As a special case, every vertex is considered to be reachable from itself (by a path with zero edges). If a vertex can reach itself via a nontrivial path (a path with one or more edges), then that path is a cycle, so another way to define directed acyclic graphs is that they are the graphs in which no vertex can reach itself via a nontrivial path.

Topological Sorting

In computer science, a topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks.

Precisely, a topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited. A topological ordering is possible if and only if the graph has no directed cycles, that is if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time.

Topological sorting has many applications, especially in ranking problems such as feedback arc sets. Topological sorting is possible even when the DAG has disconnected components.

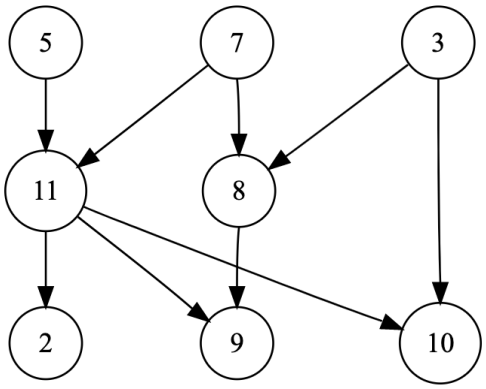
An example of topological sorting:

The canonical application of topological sorting is in scheduling a sequence of jobs or tasks based on their dependencies. The jobs are represented by vertices, and there is an edge from x to y if job x must be completed before job y can be started (for example, when washing clothes, the washing machine must finish before we put the clothes in the dryer).

Then, a topological sort gives an order in which to perform the jobs. A closely related application of topological sorting algorithms was first studied in the early 1960s in the context of the PERT technique for scheduling in project management. In this application, the vertices of a graph represent the milestones of a project, and the edges represent tasks that must be performed between one milestone and another.

Topological sorting forms the basis of linear-time algorithms for finding the critical path of the project, a sequence of milestones and tasks that controls the length of the overall project schedule.

In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers. It is also used to decide in which order to load tables with foreign keys in databases.



The graph shown to the left has many valid topological sorts, including:

- 5, 7, 3, 11, 8, 2, 9, 10 (visual top-to-bottom, left-to-right)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges, asymptotically, O(|V| + |E|)

Algorithms like Kahn's Algorithm, Depth first algorithm, and breadth-first algorithm are derived using Topological sorting methods.

Representing the graph

There are several ways to take the image above and represent it in code. We'll use an Adjacency list for clarity, by keeping track of a set of nodes (the circles), and for each node, a set of nodes it points to.

```
// A directed acyclic graph with a list of nodes
// and a mapping of connections between nodes
const dag = {
  nodes: new Set([5, 7, 3, 11, 8, 2, 9, 10]),
  adjacency: {
      5: new Set([11]),
      7: new Set([11, 8]),
      3: new Set([8, 10]),
      11: new Set([2, 9, 10]),
      8: new Set([9]),
 },
};
```

Kahn's algorithm

One of these algorithms, first described by Kahn (1962), works by choosing vertices in the same order as the eventual topological sort.[2] First, find a list of "start nodes" which have no incoming edges and insert them into a set S; at least one such node must exist in a non-empty acyclic graph.

Pseudocode:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge
while S is not empty do
        remove a node n from S
        add n to L
        for each node m with an edge e from n to m do
              remove edge e from the graph
              if m has no other incoming edges then
              insert m into S

if the graph has edges then
   return error (graph has at least one cycle)
else
   return L (a topologically sorted order)
```

Code:

```
# A Python program to print topological sorting of a graph
# using indegrees

#without cycle
from collections import defaultdict

#Class to represent a graph
class Graph:
   def __init__(self,vertices):
      self.graph = defaultdict(list) #dictionary containing adjacency List
      self.V = vertices #No. of vertices

   # function to add an edge to graph
   def addEdge(self,u,v):
      self.graph[u].append(v)


   # The function to do Topological Sort.
   def topologicalSort(self):
```

```
# Create a vector to store indegrees of all
# vertices. Initialize all indegrees as 0.
in_degree = [0]*(self.V)

# Traverse adjacency lists to fill indegrees of
  # vertices.  This step takes O(V+E) time
for i in self.graph:
   for j in self.graph[i]:
      in_degree[j] += 1

# Create an queue and enqueue all vertices with
# indegree 0
queue = []
for i in range(self.V):
   if in_degree[i] == 0:
      queue.append(i)

#Initialize count of visited vertices
cnt = 0

# Create a vector to store result (A topological
# ordering of the vertices)
top_order = []

# One by one dequeue vertices from queue and enqueue
# adjacents if indegree of adjacent becomes 0
while queue:

   # Extract front of queue (or perform dequeue)
   # and add it to topological order
   u = queue.pop(0)
   top_order.append(u)

   # Iterate through all neighbouring nodes
   # of dequeued node u and decrease their in-degree
   # by 1
   for i in self.graph[u]:
      in_degree[i] -= 1
      # If in-degree becomes zero, add it to queue
      if in_degree[i] == 0:
         queue.append(i)

   cnt += 1
```

```
        # Check if there was a cycle
        if cnt != self.V:
            print ("There exists a cycle in the graph")
        else :
            #Print topological order
            print (top_order)

g= Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);


print ("Following is a Topological Sort of the given graph")
g.topologicalSort()
```

Output:

Following is a Topological Sort of the given graph
[4, 5, 2, 0, 3, 1]

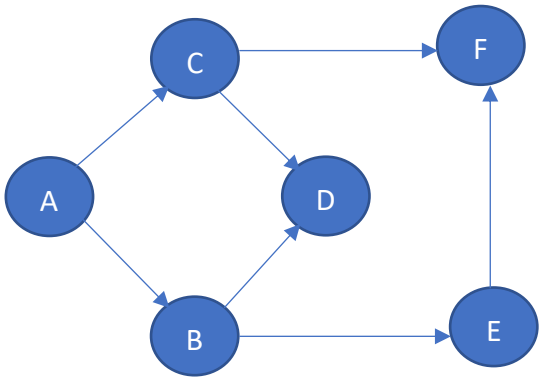
Complexity Analysis:

- **Time Complexity: O(V+E).**
  The outer for loop will be executed V number of times and the inner for loop will be executed E number of times.
- **Auxiliary Space: O(V).**
  The queue needs to store all the vertices of the graph. So, the space required is O(V).

Applications of Topological Sorting:

1. Finding deadlocks in OS
2. Scheduling jobs from the given dependencies among jobs.
3. Instruction scheduling
4. Ordering of formula cell evaluation when recomputing formula values in spreadsheets
5. Logic synthesis
6. Determining the order of compilation tasks to perform in makefiles
7. Data serialization
8. Resolving symbol dependencies in linkers
9. Finding cycles in a graph
10. Finding prerequisites of a task

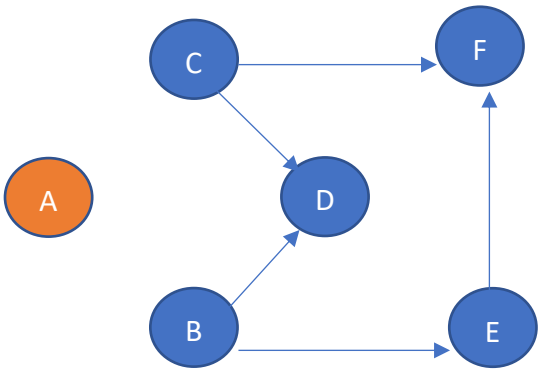
Kahn's Algorithm with an example:



Basically, the algorithm:

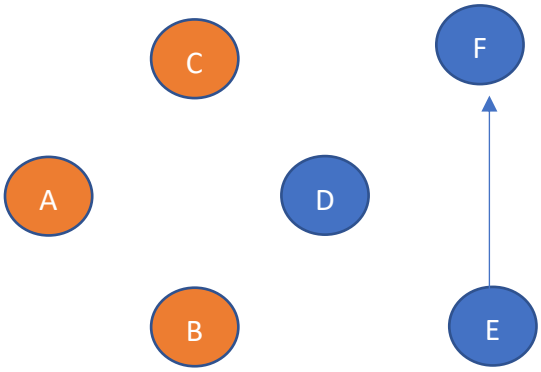
1. Finds the vertex that has no incoming edges.
2. Remove all outgoing edges from that vertex.
3. Add vertex to the topological order.
4. Repeat the process until all the vertices are processed.

STEP 1:



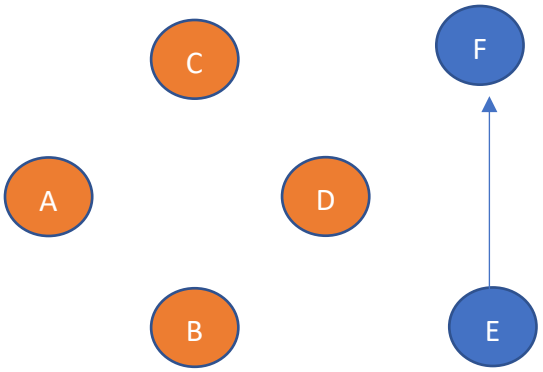
Order: A

STEP 2:



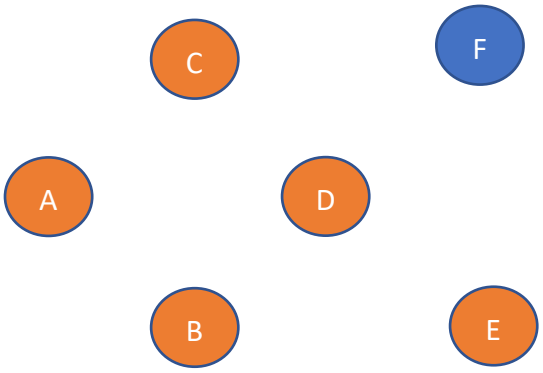
Order: A B C

STEP 3:



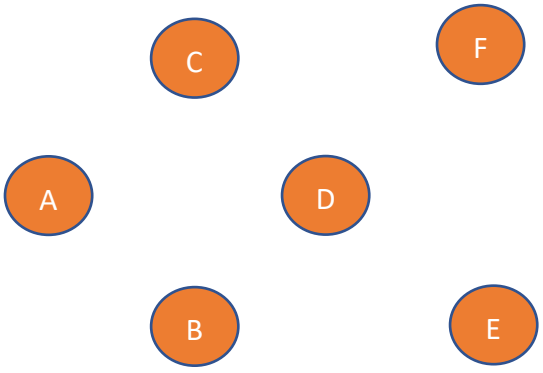
Order: A B C D

STEP 4:



Order: A B C D F

STEP 5:



Order: A B C D E F