Topological Sort (medium)

We'll cover the following

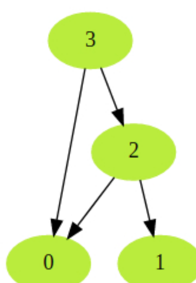
Problem Statement

Topological Sort of a directed graph (a graph with unidirectional edges) is a linear ordering of its vertices such that for every directed edge (U, V) from vertex `U` to vertex `V`, `U` comes before `V` in the ordering.

Given a directed graph, find the topological ordering of its vertices.

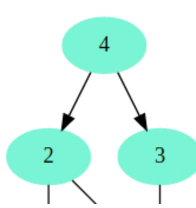
Example 1:

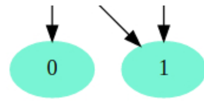
```
Input: Vertices=4, Edges=[3, 2], [3, 0], [2, 0], [2, 1]
Output: Following are the two valid topological sorts for the given graph:
1) 3, 2, 0, 1
2) 3, 2, 1, 0
```



Example 2:

```
Input: Vertices=5, Edges=[4, 2], [4, 3], [2, 0], [2, 1], [3, 1]
Output: Following are all valid topological sorts for the given graph:
1) 4, 2, 3, 0, 1
2) 4, 3, 2, 0, 1
3) 4, 3, 2, 1, 0
4) 4, 2, 3, 1, 0
5) 4, 2, 0, 3, 1
```

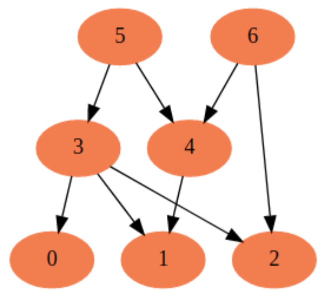




Example 3:

```
Input: Vertices=7, Edges=[6, 4], [6, 2], [5, 3], [5, 4], [3, 0], [3, 1], [3, 2], [4, 1]
Output: Following are all valid topological sorts for the given graph:
1) 5, 6, 3, 4, 0, 1, 2
2) 6, 5, 3, 4, 0, 1, 2
3) 5, 6, 4, 3, 0, 2, 1
4) 6, 5, 4, 3, 0, 1, 2
5) 5, 6, 3, 4, 0, 2, 1
6) 5, 6, 3, 4, 1, 2, 0

There are other valid topological ordering of the graph too.
```



Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```python
def topological_sort(vertices, edges):
  sortedOrder = []
  # TODO: Write your code here
  return sortedOrder


def main():
  print("Topological sort: " +
        str(topological_sort(4, [[3, 2], [3, 0], [2, 0], [2, 1]])))
  print("Topological sort: " +
        str(topological_sort(5, [[4, 2], [4, 3], [2, 0], [2, 1], [3, 1]])))
  print("Topological sort: " +
        str(topological_sort(7, [[6, 4], [6, 2], [5, 3], [5, 4], [3, 0], [3, 1], [3, 2], [4, 1]])))


main()
```

Run | Save | Reset

Solution

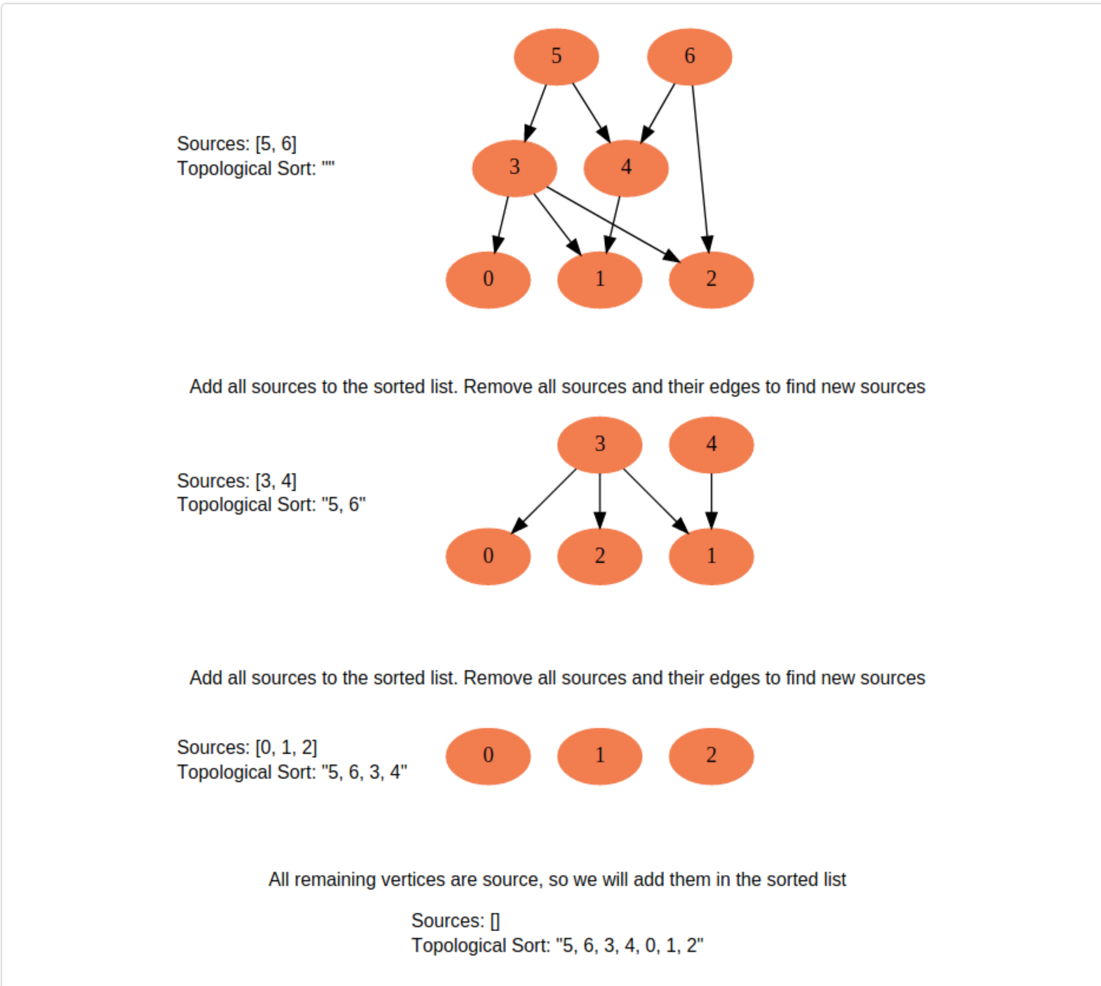
The basic idea behind the topological sort is to provide a partial ordering among the vertices of the graph such that if there is an edge from `U` to `V` then U≤V i.e., `U` comes before `V` in the ordering. Here are a few fundamental concepts related to topological sort:

1. **Source:** Any node that has no incoming edge and has only outgoing edges is called a source.

2. **Sink:** Any node that has only incoming edges and no outgoing edge is called a sink.
3. So, we can say that a topological ordering starts with one of the sources and ends at one of the sinks.
4. A topological ordering is possible only when the graph has no directed cycles, i.e. if the graph is a **Directed Acyclic Graph (DAG)**. If the graph has a cycle, some vertices will have cyclic dependencies which makes it impossible to find a linear ordering among vertices.

To find the topological sort of a graph we can traverse the graph in a **Breadth First Search (BFS)** way. We will start with all the sources, and in a stepwise fashion, save all sources to a sorted list. We will then remove all sources and their edges from the graph. After the removal of the edges, we will have new sources, so we will repeat the above process until all vertices are visited.

Here is the visual representation of this algorithm for Example-3:



This is how we can implement this algorithm:

a. Initialization

1. We will store the graph in **Adjacency Lists**, which means each parent vertex will have a list containing all of its children. We will do this using a **HashMap** where the ‘key’ will be the parent vertex number and the value will be a **List** containing children vertices.
2. To find the sources, we will keep a **HashMap** to count the in-degrees i.e., count of incoming edges of each vertex. Any vertex with ‘0’ in-degree will be a source.

b. Build the graph and find in-degrees of all vertices

1. We will build the graph from the input and populate the in-degrees **HashMap**.

c. Find all sources

1. All vertices with '0' in-degrees will be our sources and we will store them in a **Queue**.

d. Sort

1. For each source, do the following things:
   - Add it to the sorted list.
   - Get all of its children from the graph.
   - Decrement the in-degree of each child by 1.
   - If a child's in-degree becomes '0', add it to the sources **Queue**.
2. Repeat step 1, until the source **Queue** is empty.

Code

Here is what our algorithm will look like:

Java Python3 C++ JS

```python
from collections import deque


def topological_sort(vertices, edges):
  sortedOrder = []
  if vertices <= 0:
    return sortedOrder

  # a. Initialize the graph
  inDegree = {i: 0 for i in range(vertices)}  # count of incoming edges
  graph = {i: [] for i in range(vertices)}  # adjacency list graph

  # b. Build the graph
  for edge in edges:
    parent, child = edge[0], edge[1]
    graph[parent].append(child)  # put the child into it's parent's list
    inDegree[child] += 1  # increment child's inDegree

  # c. Find all sources i.e., all vertices with 0 in-degrees
  sources = deque()
  for key in inDegree:
    if inDegree[key] == 0:
      sources.append(key)

  # d. For each source, add it to the sortedOrder and subtract one from all of its children's in-degrees
  # if a child's in-degree becomes zero, add it to the sources queue
  while sources:
    vertex = sources.popleft()
    sortedOrder.append(vertex)
    for child in graph[vertex]:  # get the node's children to decrement their in-degrees
      inDegree[child] -= 1
      if inDegree[child] == 0:
        sources.append(child)

  # topological sort is not possible as the graph has a cycle
  if len(sortedOrder) != vertices:
    return []

  return sortedOrder


def main():
  print("Topological sort: " +
        str(topological_sort(4, [[3, 2], [3, 0], [2, 0], [2, 1]])))
  print("Topological sort: " +
        str(topological_sort(5, [[4, 2], [4, 3], [2, 0], [2, 1], [3, 1]])))
  print("Topological sort: " +
        str(topological_sort(7, [[6, 4], [6, 2], [5, 3], [5, 4], [3, 0], [3, 1], [3, 2], [4, 1]])))


main()
```

Run Save Reset

Time Complexity

In step 'd', each vertex will become a source only once and each edge will be accessed and removed once. Therefore, the time complexity of the above algorithm will be $O(V + E)$, where 'V' is the total number of vertices and 'E' is the total number of edges in the graph.

Space Complexity

The space complexity will be $O(V + E)$, since we are storing all of the edges for each vertex in an adjacency list.

Similar Problems

Problem 1: Find if a given **Directed Graph** has a cycle in it or not.

Solution: If we can't determine the topological ordering of all the vertices of a directed graph, the graph has a cycle in it. This was also referred to in the above code:

```
    if (sortedOrder.size() != vertices) // topological sort is not possible as the graph has a cycle
      return new ArrayList<>();
```

← Back

Introduction

Next →

Tasks Scheduling (medium)

Completed

Report an Issue Ask a Question