210. Course Schedule II

Dec. 16, 2018 | 100.6K views

Average Rating: 4.79 (124 votes)

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

Example 1:

```
Input: 2, [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should have
             course 0. So the correct course order is [0,1] .
```

Example 2:

```
Input: 4, [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3] or [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should have
             courses 1 and 2. Both courses 1 and 2 should be taken after you finished
             So one correct course order is [0,1,2,
3]. Another correct ordering is [0,2,1,3] .
```

Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.

Solution

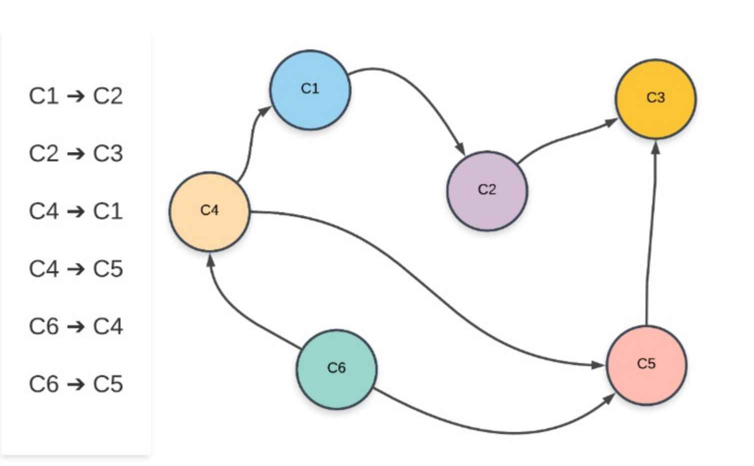
This is a very common problem that some of us might face during college. We might want to take up a certain set of courses that interest us. However, as we all know, most of the courses do tend to have a lot of prerequisites associated with them. Some of these would be hard requirements whereas others would be simply `suggested` prerequisites which you may or may not take. However, for us to be able to have an all round learning experience, we should follow the suggested set of prerequisites. How does one decide what order of courses they should follow so as not to miss out on any subjects?

As mentioned in the problem statement, such a problem is a natural fit for graph based algorithms and we can easily model the elements in the problem statement as a graph. First of all, let's look at the graphical representation of the problem and it's components and then we will move onto the solutions.

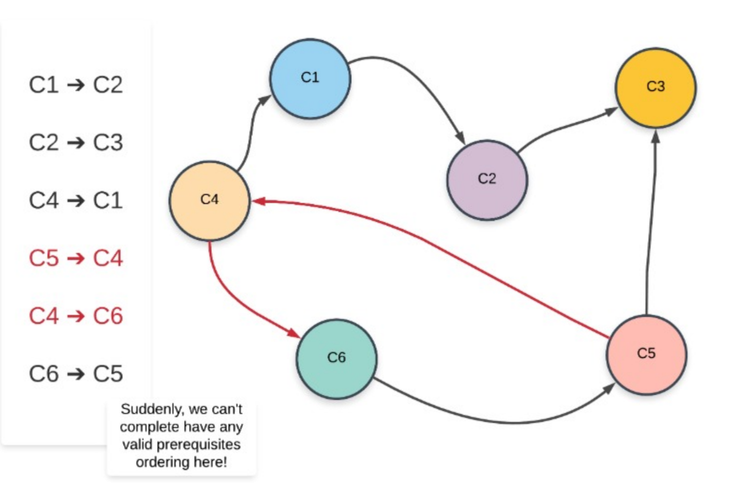
We can represent the information provided in the question in the form of a graph.

- Let $G(V, E)$ represent a `directed, unweighted` graph.
- Each course would represent a vertex in the graph.
- The edges are modeled after the prerequisite relationship between courses. So, we are given, that a pair such as $[a, b]$ in the question means the course `b` is a prerequisite for the course `a`. This can be represented as a `directed edge b ➔ a` in the graph.
- The graph is a cyclic graph because there is a possibility of a cycle in the graph. If the graph would be acyclic, then an ordering of subjects as required in the question would `always` be possible. Since it's mentioned that such an ordering may not always be possible, that means we have a cyclic graph.

Let's look at a sample graph representing a set of courses where such an ordering is possible and one where such an ordering is not possible. It will be easier to explain the approaches once we look at two sample graphs.



For the sample graph shown above, one of the possible ordering of courses is: `C6 ➔ C4 ➔ C1 ➔ C5 ➔ C2 ➔ C3` and another possible ordering of subjects is `C6 ➔ C4 ➔ C5 ➔ C1 ➔ C2 ➔ C3`. Now let's look at a graph where no such ordering of courses is possible.



Note that the edges that have changed from the previous graph have been highlighted in red.

> Clearly, the presence of a cycle in the graph shows us that a proper ordering of prerequisites is not possible at all. Intuitively, it is not possible to have e.g. two subjects S1 and S2 prerequisites of each other. Similar ideology applies to a larger cycle in the graph like we have above.

Such an ordering of subjects is referred to as a `Topological Sorted Order` and this is a common algorithmic problem in the graph domain. There are two approaches that we will be looking at in this article to solve this problem.

Approach 1: Using Depth First Search

Intuition

Suppose we are at a node in our graph during the depth first traversal. Let's call this node `A`.

> The way DFS would work is that we would consider all possible paths stemming from A before finishing up the recursion for A and moving onto other nodes. All the nodes in the paths stemming from the node A would have A as an ancestor. The way this fits in our problem is, all the courses in the paths stemming from the course A would have A as a prerequisite.

Now we know how to get all the courses that have a particular course as a prerequisite. If a valid ordering of courses is possible, the course `A` would come before all the other set of courses that have it as a prerequisite. This idea for solving the problem can be explored using depth first search. Let's look at the pseudo-code before looking at the formal algorithm.

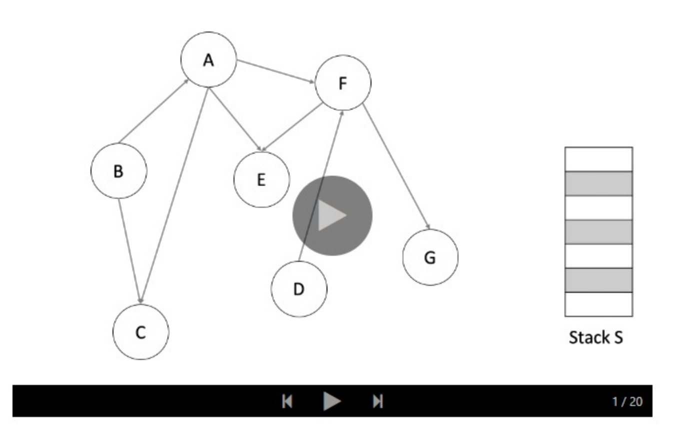
```
➔ let S be a stack of courses
➔ function dfs(node)
➔     for each neighbor in adjacency list of node
➔         dfs(neighbor)
➔     add node to S
```

Let's now look at the formal algorithm based on this idea.

Algorithm

1. Initialize a stack `S` that will contain the topologically sorted order of the courses in our graph.
2. Construct the adjacency list using the edge pairs given in the input. An important thing to note about the input for the problem is that a pair such as `[a, b]` represents that the course `b` needs to be taken in order to do the course `a`. This implies an edge of the form `b ➔ a`. Please take note of this when implementing the algorithm.
3. For each of the nodes in our graph, we will run a depth first search in case that node was not already visited in some other node's DFS traversal.
4. Suppose we are executing the depth first search for a node `N`. We will recursively traverse all of the neighbors of node `N` which have not been processed before.
5. Once the processing of all the neighbors is done, we will add the node `N` to the stack. We are making use of a stack to simulate the ordering we need. When we add the node `N` to the stack, all the nodes that require the node `N` as a prerequisites (among others) will already be in the stack.
6. Once all the nodes have been processed, we will simply return the nodes as they are present in the stack from top to bottom.

Let's look at an animated dry run of the algorithm on a sample graph before moving onto the formal implementations.



> An important thing to note about topologically sorted order is that there won't be just one ordering of nodes (courses). There can be multiple. For e.g. in the above graph, we could have processed the node "D" before we did "B" and hence have a different ordering.

Java Python Copy

```python
from collections import defaultdict
class Solution:

    WHITE = 1
    GRAY = 2
    BLACK = 3

    def findOrder(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: List[int]
        """

        # Create the adjacency list representation of the graph
        adj_list = defaultdict(list)

        # A pair [a, b] in the input represents edge from b --> a
        for dest, src in prerequisites:
            adj_list[src].append(dest)

        topological_sorted_order = []
        is_possible = True

        # By default all vertces are WHITE
        color = {k: Solution.WHITE for k in range(numCourses)}
        def dfs(node):
```

Complexity Analysis

- Time Complexity: $O(N)$ considering there are N courses in all. We essentially perform a complete depth first search covering all the nodes in the forest. It's a forest and not a graph because not all nodes will be connected together. There can be disjoint components as well.
- Space Complexity: $O(N)$, the space utilized by the recursion stack (not the stack we used to maintain the topologically sorted order)

Approach 2: Using Node Indegree

Intuition

This approach is much easier to think about intuitively as will be clear from the following point/fact about topological ordering.

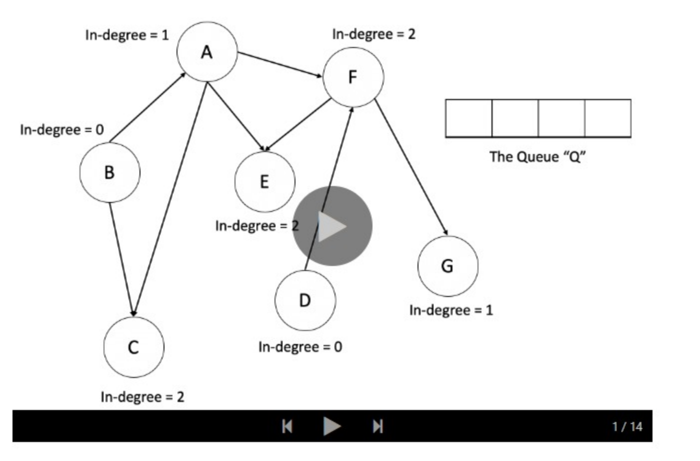
> The first node in the topological ordering will be the node that doesn't have any incoming edges. Essentially, any node that has an in-degree of 0 can start the topologically sorted order. If there are multiple such nodes, their relative order doesn't matter and they can appear in any order.

Our current algorithm is based on this idea. We first process all the nodes/course with 0 in-degree implying no prerequisite courses required. If we remove all these courses from the graph, along with their outgoing edges, we can find out the courses/nodes that should be processed next. These would again be the nodes with 0 in-degree. We can continuously do this until all the courses have been accounted for.

Algorithm

1. Initialize a queue, `Q` to keep a track of all the nodes in the graph with 0 in-degree.
2. Iterate over all the edges in the input and create an adjacency list and also a map of node v/s in-degree.
3. Add all the nodes with 0 in-degree to `Q`.
4. The following steps are to be done until the `Q` becomes empty.
   1. Pop a node from the `Q`. Let's call this node, `N`.
   2. For all the neighbors of this node, `N`, reduce their in-degree by 1. If any of the nodes' in-degree reaches 0, add it to the `Q`.
   3. Add the node `N` to the list maintaining topologically sorted order.
   4. Continue from step 4.1.

Let us now look at an animation depicting this algorithm and then we will get to the implementations.



An important thing to note here is, using a queue is not a hard requirement for this algorithm. We can make use of a stack. That however, will give us a different ordering than what we might get from the queue because of the difference in access patterns between the two data-structures.

Java Python Copy

```python
from collections import defaultdict, deque
class Solution:

    def findOrder(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: List[int]
        """

        # Prepare the graph
        adj_list = defaultdict(list)
        indegree = {}
        for dest, src in prerequisites:
            adj_list[src].append(dest)

            # Record each node's in-degree
            indegree[dest] = indegree.get(dest, 0) + 1

        # Queue for maintainig list of nodes that have 0 in-degree
        zero_indegree_queue = deque([k for k in range(numCourses) if k not in indegree])

        topological_sorted_order = []

        # Until there are nodes in the Q
        while zero_indegree_queue:
```

Complexity Analysis

- Time Complexity: $O(V + E)$ where V represents the number of vertices and E represents the number of edges. We pop each node exactly once from the zero in-degree queue and that gives us V. Also, for each vertex, we iterate over its adjacency list and in totality, we iterate over all the edges in the graph which gives us E. Hence, $O(V + E)$
- Space Complexity: $O(V + E)$. We use an intermediate queue data structure to keep all the nodes with 0 in-degree. In the worst case, there won't be any prerequisite relationship and the queue will contain all the vertices initially since all of them will have 0 in-degree. That gives us $O(V)$. Additionally, we also use the adjacency list to represent our graph initially. The space occupied is defined by the number of edges because for each node as the key, we have all its adjacent nodes in the form of a list as the value. Hence, $O(E)$. So, the overall space complexity is $O(V + E)$.

Kudos to people in the comments section and @himanshujain1](https://leetcode.com/himanshujain1/) for bringing up a grave oversight on my part in the complexity analysis for the second approach!

Rate this article: ★ ★ ★ ★ ★

Comments: 41 Sort By

Type comment here... (Markdown is supported)

Preview Post

Marlon2102win ★ 93 March 29, 2019 6:08 AM

I think the time complexity for solution 2 should be O(E + V) since for every node, we have to expand all of its neighbors. As for space, I think it's also O(max(E + V)), for adjacent List which use HashMap here, there should be O(E), and for queue, the worst case would be O(v), so all the course has no prerequisite.

53 Share Reply

SHOW 5 REPLIES

liuweilin17 ★ 18 February 13, 2019 4:13 AM

Elaborate general solution to topological sort

11 Share Reply

SHOW 2 REPLIES

durgaganesh ★ 20 August 26, 2019 3:13 AM

Nice article. But isn't the time complexity for the 1st approach O(V+E)? Because of DFS of the forest.

9 Share Reply

entergmode ★ 33 February 14, 2019 10:54 PM

very well written article!

8 Share Reply

SHOW 1 REPLY

Username1604 ★ 39 April 24, 2020 3:28 AM

Solution 2 is Kahn's algorithm, it could be useful to have proper references to additional materials to study.

7 Share Reply

rytaus ★ 6 March 3, 2019 6:14 AM

For the second method are you sure the time complexity would be O(N)? It looks like we touch every node once O(N), but the work we have to do there (update the indegree of other nodes) would be bounded by the number of nodes as well, leading to N work there. Would that not make this O(N*2)?

4 Share Reply

SHOW 3 REPLIES

1337c0d3r ADMIN ★ 2883 December 23, 2018 7:21 AM

Nice article Sachin!

6 Share Reply

SHOW 1 REPLY

hxuanhung ★ 162 December 1, 2019 9:21 PM

The step to build the adjacency list take `O(E)` where E is the number of edges. Plus `O(V)` to process all nodes. In total, time complexity is `O(V+E)`.