Solution: Course Schedule II

Let's solve the Course Schedule II problem using the Topological Sort pattern.

We'll cover the following

Statement

Let's assume that there are a total of n courses labeled from 0 to $n-1$. Some courses may have prerequisites. A list of prerequisites is specified such that if $Prerequisites_i = a, b$, you must take course b before course a.

Given the total number of courses n and a list of the prerequisite pairs, return the course order a student should take to finish all of the courses. If there are multiple valid orderings of courses, then the return any one of them.

> **Note:** There can be a course in the 0 to $n-1$ range with no prerequisites.

Constraints:

Let n be the number of courses.

- $1 \leq n \leq 2000$
- $0 \leq$ `prerequisites.length` $\leq n * (n-1)$
- `prerequisites[i].length` $== 2$
- $0 \leq a,\ b < n$
- $a \neq b$
- All the pairs $[a,\ b]$ are distinct.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

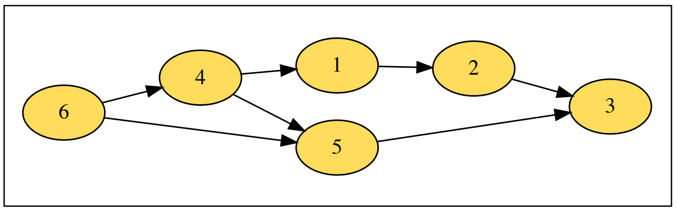
The naive approach is to use nested loops to iterate over the prerequisites list, find the dependency of one course on another, and store them in a separate array.

The time complexity of this naive approach is $O(n^2)$, where n is the number of courses. However, the space complexity of this naive approach is $O(n)$. Let's see if we can use the topological sort pattern to reduce the time complexity of our solution.

Optimized approach using topological sort

This problem can be solved using the topological sort pattern. Topological sort is used to find a linear ordering of elements that depend on or prioritize each other. For example, if A depends on B or if B has priority over A, B is listed before A in topological order.

For this problem, we find the topological order of the given number of courses using the list of the courses' prerequisites. Using the given list of prerequisites, we can build the graph representing the dependency of one course on another. To traverse a graph, we can use breadth-first search to find the courses' order.



Sample example

The vertices in the graph represent the courses, and the directed edge represents the dependency relationship. We get the order from the above example:

$$C_6 \rightarrow C_4 \rightarrow C_1 \rightarrow C_5 \rightarrow C_2 \rightarrow C_3$$

Another possible ordering of the above courses can be the following:

$$C_6 \rightarrow C_4 \rightarrow C_5 \rightarrow C_1 \rightarrow C_2 \rightarrow C_3$$

This order of graph vertices is known as a **topological sorted order**.

The basic idea behind the topological sort is to provide a partial ordering of the graph's vertices so that if there's an edge from U to V, then $U \leq V$. This means U comes before V in the ordering. Let's review some basic concepts that will help us understand topological sorting:

1. **Source:** Any vertex with no incoming edge and only outgoing edges is called a source.
2. **Sink:** Any vertex that has only incoming edges and no outgoing edge is called a sink.

So, we can say that a topological ordering starts with one of the sources and ends at one of the sinks.

A topological ordering is possible only when the graph has no directed cycles, that is, if the graph is a **Directed Acyclic Graph (DAG)**. No linear ordering among the vertices is possible if the graph has one or more cycles.

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

Here is how we will implement this algorithm:

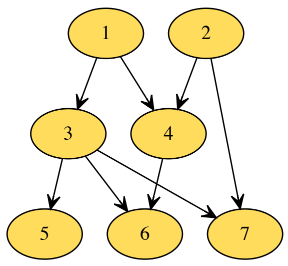
1. Initialization:
   - We store the graph in adjacency lists, where each parent vertex has a list containing all of its children. We do this using a hash map, where the key is the parent vertex number, and the value is a

list containing the children’s vertex numbers.
    - To find the sources, we keep a hash map to count the in-degrees, which is the count of incoming vertices’ edges. Any vertex with a 0 in-degree is a source.
2. Build the graph and find in-degrees of all vertices:
    - We build the graph from the input and populate the in-degrees hash map.
3. Find all sources:
    - Our sources are all the vertices with 0 in degrees, and we store them in a queue.
4. Sort:
    - For each source, we do the following:
        - Add it to the sorted list.
        - Retrieve 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 source queue.
    - Repeat the first step until the source queue is’nt empty.

The following illustration might clarify this process.

Initial structure

Sources:
Correct Order:



1 of 7

Let’s look at the code for this solution below:

Java

```java
class CourseSchedule{

   public static List <Integer> findOrder(int n, int[][] prerequisites) {
     List <Integer> sortedOrder = new ArrayList < > ();
     // if n is smaller than or equal to zero we will return the empty array
     if (n <= 0)
         return sortedOrder;

     // Store the count of incoming prerequisites in a hashmap
     HashMap <Integer, Integer> inDegree = new HashMap < > ();
     // a. Initialize the graph
     HashMap < Integer, List < Integer >> graph = new HashMap < > ();
     for (int i = 0; i < n; i++) {
         inDegree.put(i, 0);
         graph.put(i, new ArrayList < Integer > ());
     }

     // b. Build the graph
     for (int i = 0; i < prerequisites.length; i++) {
         int parent = prerequisites[i][1], child = prerequisites[i][0];
         graph.get(parent).add(child); // put the child into it's parent's list
         inDegree.put(child, inDegree.get(child) + 1); // increment child's inDegree
     }

     // c. Find all sources i.e., all n with 0 in-degrees
     Queue < Integer > sources = new LinkedList < > ();
     // traverse in inDegree using key
     for (Map.Entry < Integer, Integer > entry: inDegree.entrySet()) {
```

Course Schedule II

Solution summary

1. Make a graph by initializing the hash map with the vertices and their children.
2. Keep the number of in-degrees of each of the vertex in the separate hash map.
3. Find the source vertex.
4. Add the source to the sorted list.
5. Retrieve all the source children and add them to the queue.
6. Decrement the in-degrees of the retrieved children.
7. If the in-degree of the child vertex becomes equal to zero, add it to the sorted list.
8. Repeat the process until the queue isn't empty.

Time complexity

In step four, each vertex becomes a source only once, and each edge is accessed and removed once. Therefore, the above algorithm's time complexity is $O(v + e)$, where v is the total number of vertices and e is the total number of edges in the graph.

Space complexity

The space complexity is $O(v + e)$ because we store all edges for each vertex in an adjacency list.

Back
Course Schedule II

Next
Course Schedule

Mark as Completed