★ ★ ★ ★ ★ Average Rating: 5.00 (29 votes)

Premium

Solution

Overview

In this problem, we need to learn all the courses as fast as possible. The number of courses we can learn in one semester is unlimited, and the only limitation is the prerequisite relationship: we can only learn those courses whose prerequisite(s) is fulfilled.

This problem is an application of Topological sorting, and there are mainly two different kinds of solutions: BFS (Breadth-First Search) and DFS (Depth-First Search).

In this article, three approaches are introduced:

1. *Breadth-First Search (Kahn's Algorithm)*
2. *Depth-First Search: Depth-First Search: Check for Cycles + Find Longest Path*
3. *Depth-First Search: Combine*

Generally, we recommend *Approach 1* and *Approach 3* since they are efficient and easy to implement. We include *Approach 2* for a better understanding to *Approach 3*. (Therefore, it is recommended to read *Approach 2* before *Approach 3*.)

Once you've finished this problem, you can try challenging the follow-up 1494. Parallel Courses II.

Approach 1: Breadth-First Search (Kahn's Algorithm)

Intuition

We can treat the problem as a directed **graph** problem (the courses are nodes and the prerequisites are egdes). What we need to do is somehow iterate over all the nodes in the graph.

For iteration, we can do BFS or DFS. We introduce BFS in this approach and DFS in the following approaches.

To achieve the fastest learning speed, our strategy is:

> Learn **all courses** available in each semester.

This is intuitive. Even if we deliberately choose not to learn one available course, we still need to learn it in the following semesters. There is no harm to learn it now. Also, if we learn it later, then we have to postpone all courses whose prerequisite is that course.

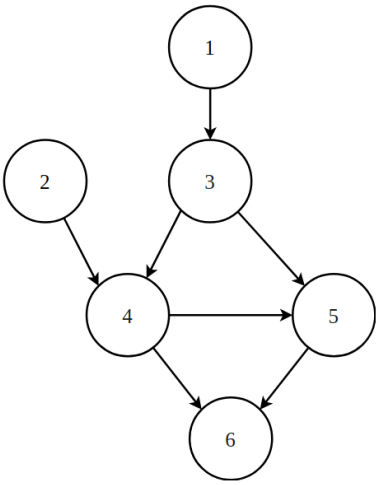
Now, the first question is:

> Where to start? (Which courses are available?)

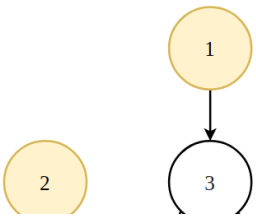
We can not start from courses with prerequisites.

> We start from nodes with **no prerequisites**.

For example, in this graph, which courses can we learn in the first semester?



Yes, those courses marked with yellow can be learned in the first semester.



Testcase Run Code Result Debugger

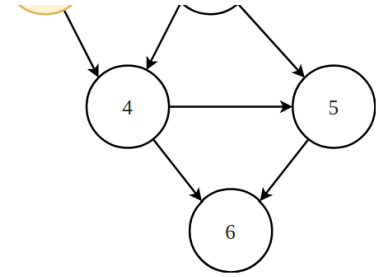
Accepted Runtime: 0 ms

Your input
3
[[1,3],[2,3]]

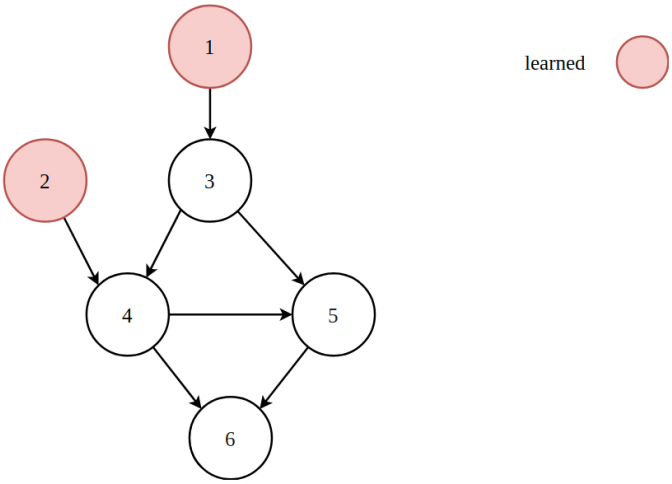
Output 2 Diff

Expected 2

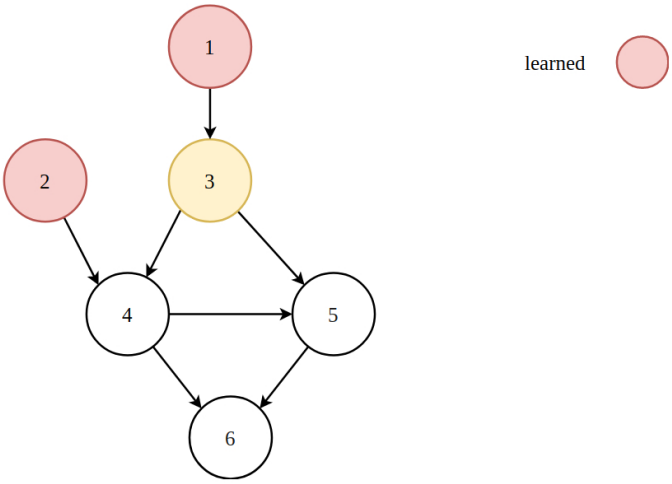
Console Use Example Testcases



Now, we have learned those courses, what should we learn next?



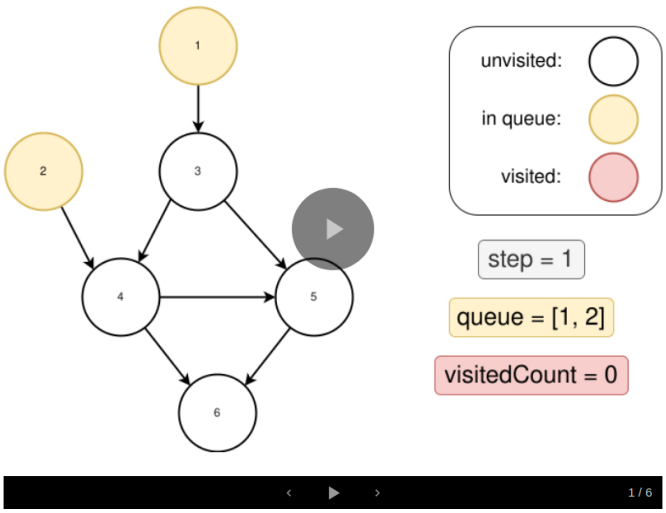
Yes, the new yellow courses can be learned, since their prerequisites are fulfilled:



Keep going until no available courses to learn.

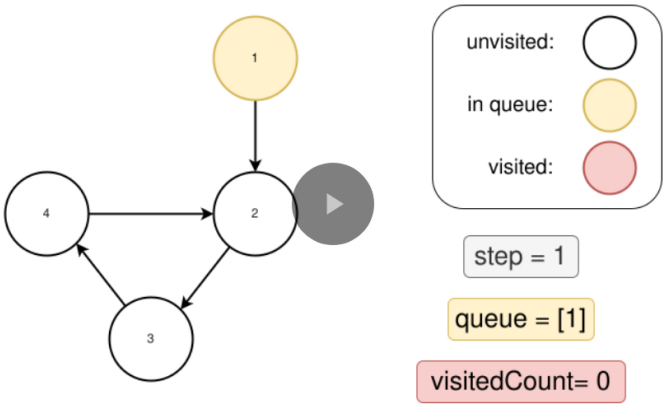
By using this strategy to allocate courses to semesters, we are guaranteed to minimize the number of semesters needed. This is because in each semester, we're learning every course that isn't "locked" by a prerequisite, and so there is no possible way to be faster.

Let's finish this example with Breadth-First Search:



In some other cases, we can not learn all nodes. If the number of nodes we visited is strictly less than the number of total nodes, then there is not way to learn all the courses and we can do nothing but return `-1`.

For example, in this graph with a cycle, we can not learn all the courses:



> This approach is also called Kahn's algorithm (with some modifications to adapt to the problem).

Algorithm

Step 1: Build a directed graph from `relations`.

Step 2: Record the in-degree of each node. (i.e., the number of edges towards the node)

Step 3: Initialize a queue, `queue`. Put nodes with an in-degree of `0` into `queue`. Initialize `step = 0`, `visited_count = 0`.

Step 4: Start BFS: Loop until `queue` is empty:

1. Initialize a queue `next_queue` to record the nodes needed in the next iteration.
2. Increment `step`.
3. For each `node` in `queue`:
    1. Increment `visitedCount`
    2. For each `end_node` reachable from `node`:
        1. Decrement the in-degree of `end_node`
        2. If the in-degree of `end_node` reaches 0, push it into `next_queue`
4. Assign `queue` to `next_queue`

Step 5: If `visited_count == N`, return `step`. Otherwise, return `-1`.

Implementation

C++ | Java | Python3 — Copy

```
    vector<int> inCount(N + 1, 0);  // or indegree
    vector<vector<int>> graph(N + 1);
    for (auto& relation : relations) {
        graph[relation[0]].push_back(relation[1]);
        inCount[relation[1]]++;
    }
    int step = 0;
    int studiedCount = 0;
    vector<int> bfsQueue;
    for (int node = 1; node < N + 1; node++) {
        if (inCount[node] == 0) {
            bfsQueue.push_back(node);
        }
    }
    // start learning with BFS
    while (!bfsQueue.empty()) {
        // start new semester
        step++;
        vector<int> nextQueue;
        for (auto& node : bfsQueue) {
            studiedCount++;
            for (auto& endNode : graph[node]) {
                inCount[endNode]--;
                // if all prerequisite courses learned
                if (inCount[endNode] == 0) {
                    nextQueue.push_back(endNode);
                }
```

Complexity Analysis

Let E be the length of `relations`. N is the number of courses, as explained in the problem description.

- Time Complexity: $\mathcal{O}(N + E)$. For building the graph, we spend $\mathcal{O}(N)$ to initialize the graph, and spend $\mathcal{O}(E)$ to add egdes since we iterate `relations` once. For BFS, we spend $\mathcal{O}(N + E)$ since we need to visit every node and edge once in BFS in the worst case.
- Space Complexity: $\mathcal{O}(N + E)$. For the graph, we spend $\mathcal{O}(N + E)$ since we have $\mathcal{O}(N)$ keys and $\mathcal{O}(E)$ values. For BFS, we spend $\mathcal{O}(N)$ since in the worst case, we need to add all nodes to the queue in the same time.

Approach 2: Depth-First Search: Check for Cycles + Find Longest Path

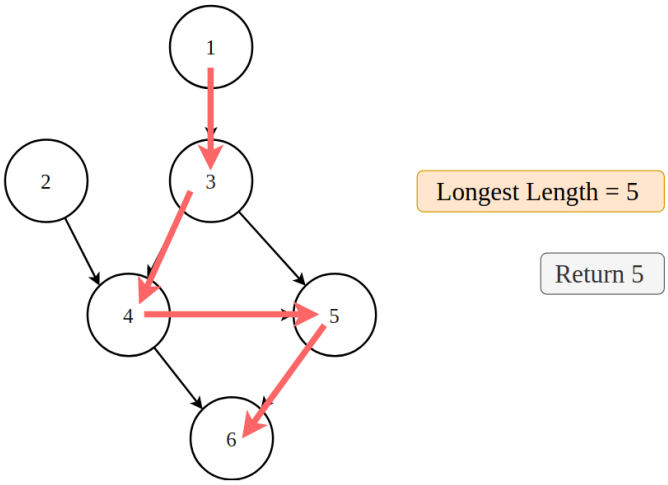
Intuition

Intuition

There is an important insight:

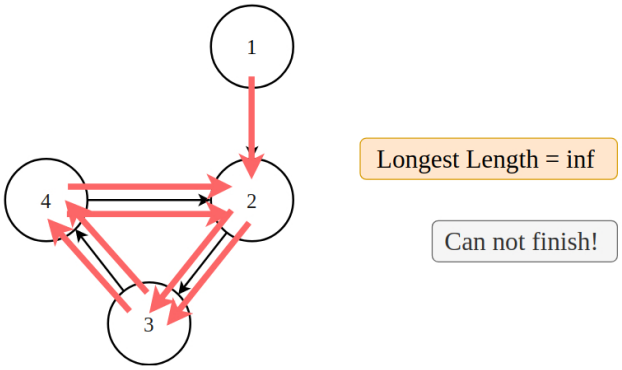
> The number of semesters needed is equal to the **length of the longest path** in the graph.

For example, the longest path in the graph below is `5`, so the number of semesters needed is `5`:



Why? Treat the path as a sequence of prerequisites, and for each prerequisite, we need to spend one semester to advance to the next node.

But there is a problem: if the graph has a cycle, then the longest path would be infinite.



So firstly, we need to check if the graph has a cycle. If it does, we can directly return `-1` since we can never finish all courses.

Now we break the problem into two parts:

1. Check if the graph has a cycle
2. Calculate the length of the longest path

Each of the two parts can be done with DFS. In *Approach 3*, we will show how to achieve those two-part simultaneously in one single DFS. However, in this approach, for a better understanding, we separate them into two separate DFS traverals.

Check If the Graph Has A Cycle

Each node has one of the three states: unvisited, visiting, and visited.

Before the DFS, we initialize all nodes in the graph to unvisited.

When performing a DFS, we mark the current node as *visiting* until we search all paths out of the node from the node. If we meet a node marked with processing, it must come from the upstream path and therefore, we've detected a cycle.

If DFS finishes, and all node are marked as visited, then the graph contains no cycle.

Calculate the Length of the Longest Path

The DFS function should return the maximum out of the recursive calls for its child nodes, plus one (the node itself).

In order to prevent redundant calculations, we need to store the calculated results. This is an example of dynamic programming, as we're storing the result of subproblems.

Algorithm

Step 1: Build a directed graph from `relations`.

Step 2: Implement a function `dfsCheckCycle` to check whether the graph has a cycle.

Step 3: Implement a function `dfsMaxPath` to calculate the length of the longest path in the graph.

Step 4: Call `dfsCheckCycle`, return `-1` if the graph has a cycle.

Step 5: Otherwise, call `dfsMaxPath`. Return the length of the longest path in the graph.

Implementation

C++ | Java | Python3 — Copy

```
        for (int node = 1; node < N + 1; node++) {
            // if has cycle, return -1
            if (dfsCheckCycle(node, graph, visited) == -1) {
                return -1;
            }
        }
```

```
        }

        // if no cycle, return the longest path
        vector<int> visitedLength(N + 1, 0);
        int maxLength = 1;
        for (int node = 1; node < N + 1; node++) {
            int length = dfsMaxPath(node, graph, visitedLength);
            maxLength = max(length, maxLength);
        }
        return maxLength;
    }

private:
    int dfsCheckCycle(int node, vector<vector<int>>& graph,
                      vector<int>& visited) {
        // return -1 if has a cycle
        // return 1 if does not have any cycle
        if (visited[node] != 0) {
            return visited[node];
        } else {
            // mark as visiting
            visited[node] = -1;
        }
```

Complexity Analysis

Let E be the length of `relations`.

- Time Complexity: $\mathcal{O}(N + E)$. For building the graph, we spend $\mathcal{O}(N)$ to initialize the graph, and spend $\mathcal{O}(E)$ to add egdes since we iterate `relations` once. For DFS, we spend $\mathcal{O}(N + E)$ since we need to visit every node and edge once in DFS in the worst case.
- Space Complexity: $\mathcal{O}(N + E)$. For the graph, we spend $\mathcal{O}(N + E)$ since we have $\mathcal{O}(N)$ keys and $\mathcal{O}(E)$ values. For DFS, we spend $\mathcal{O}(N)$ since in the worst case, we need to add all nodes to the stack to recursively call DFS. Also, we run DFS twice.

Approach 3: Depth-First Search: Combine

Intuition

> This approach is an improvement of *Approach 2*. It is recommended to ensure that you fully understood *Approach 2* before continuing onto this final approach.

Here, we combine the two functions in *Approach 2*, `dfsCheckCycle` and `dfsMaxPath`, into one single function, `dfs`.

> The new `dfs` should return `-1` if a cycle is detected, and return the longest length otherwise.

Just simple modifications on `dfsCheckCycle` will do:

Recall in `dfsCheckCycle`, each node has three states: unvisited, visiting, and visited.

We can change the **visited** state to the **longest length** starting from the current node, and let the dfs return the longest length starting from the current node.

The pseudo-code is as below:

```
set states of all nodes to unvisited

def dfs(node):
    if the state of node is visiting:
        # detects cycles
        return -1
    else if the state of node is visited:
        return the state of node # the longest length

    set the state of node to visiting

    max_length = -1
    for child_node in child_nodes:
        child_answer = dfs(child_node)
        # if detects cycles in child_node
        if child_answer == -1:
            return -1
        else:
            max_length = max(max_length, child_answer)

    set the state of node to max_length
    return max_length
```

Algorithm

Step 1: Build a directed graph from `relations`.

Step 2: Implement a function `dfs` to check whether the graph has a cycle and calculate the length of the longest path in the graph.

Step 3: Call `dfs`; return `-1` if the graph has a cycle. Otherwise, return the length of the longest path in the graph.

Implementation

C++ | Java | Python3 — Copy

```
        }
        return maxLength;
    }

private:
    int dfs(int node, vector<vector<int>>& graph, vector<int>& visited) {
        // return the longest path (inclusive)
        if (visited[node] != 0) {
            return visited[node];
        } else {
            // mark as visiting
```

```
            visited[node] = -1;
        }
        int maxLength = 1;
        for (auto& endNode : graph[node]) {
            int length = dfs(endNode, graph, visited);
            // we meet a cycle!
            if (length == -1) {
                return -1;
            }
            maxLength = max(length + 1, maxLength);
        }
        // mark as visited
        visited[node] = maxLength;
        return maxLength;
    }
};
```

Complexity Analysis

Let E be the length of `relations`.

- Time Complexity: $\mathcal{O}(N + E)$. For building the graph, we spend $\mathcal{O}(N)$ to initialize the graph, and spend $\mathcal{O}(E)$ to add egdes since we iterate `relations` once. For DFS, we spend $\mathcal{O}(N + E)$ since we need to visit every node and edge once in DFS in the worst case.
- Space Complexity: $\mathcal{O}(N + E)$. For the graph, we spend $\mathcal{O}(N + E)$ since we have $\mathcal{O}(N)$ keys and $\mathcal{O}(E)$ values. For DFS, we spend $\mathcal{O}(N)$ since in the worst case, we need to add all nodes to the stack to recursively call DFS.

Report Article Issue

Comments: 8 | Best | Most Votes | Newest to Oldest | Oldest to Newest

Type comment here... (Markdown is supported)

Post

kiljaeden ★ 504 March 3, 2021 6:01 AM

actually I think this is an easy hard, or hard medium

36 Show 1 reply Reply

harerama ★ 10 February 28, 2021 11:47 PM

cool

3 Reply

krishnaramb ★ 2 July 19, 2021 11:01 PM

Using Kahn algorithm here, seems like much easy,

class Solution {
public:

```
/* Use the Kahn Algorithm for finding out how many of indegree of zero nodes label are there

*/
int minimumSemesters(int n, vector<vector<int>>& relations) {
    vector<vector<int>> adj(n+1); //node 0 is not used
    vector<int> indegree(n+1, 0);
```

Read More

0 Reply

wddwycc ★ 82 March 29, 2021 5:53 AM

How about use two hashmaps: https://leetcode.com/problems/parallel-courses/discuss/1132696/Rust-hashmap-solution

0 Reply

foymula ★ 2 Last Edit: March 21, 2021 9:09 PM

Can someone help explain why the DFS calls in the main method have to be in a for loop? Is this because we need to check the graph from each angle since we don't necessarily know the starting point that we need to look at?

0 Show 2 replies Reply

RiccardT ★ 0 Last Edit: April 3, 2021 10:47 AM

I would argue that the first solution isn't using breadth first search since we aren't using a queue, but rather a list of all available courses, to traverse. In fact, we don't even have a visited structure. We aren't even popping! Here is my version of Approach 1 with OOP and a semester, schedule, classes naming style so that the solution is (hopefully) a bit more intuitive.

https://leetcode.com/problems/parallel-courses/discuss/1140784/Object-Oriented-Solution-with-Modified-Khan's-Algorithm-READABLE-AF

-1 Show 2 replies Reply

sharabiania ★ -1 March 29, 2021 6:44 PM

This is wrong: `The number of semesters needed is equal to the length of the longest path in the graph.`

counter example: `[[1,2],[3,4],[4,5],[5,2]]`

longest path here is "3" but the answer is "4".

-9 Show 2 replies Reply

tuanzai ★ -31 May 24, 2021 4:37 PM

should be easy not medium

-34 Reply