

207. Course Schedule (/problems/course-schedule/)

March 2, 2020 | 26.5K views

Average Rating: 4.05 (20 votes)

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses-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**, is it possible for you to finish all courses?

Example 1:

Input: `numCourses = 2, prerequisites = [[1,0]]`

Output: `true`

Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0. So it is possible to finish all courses.

Example 2:

Input: `numCourses = 2, prerequisites = [[1,0],[0,1]]`

Output: `false`

Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0, and to take course 0 you also have finished course 1. So it is impossible to finish all courses.

Constraints:

- The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented (<https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs>).
- You may assume that there are no duplicate edges in the input prerequisites.

- $1 \leq \text{numCourses} \leq 10^5$

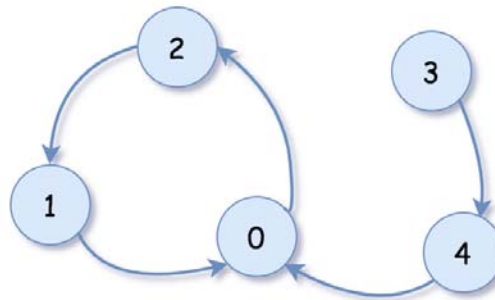
Solution

Approach 1: Backtracking

Intuition

The problem could be modeled as yet another **graph traversal** problem, where each course can be represented as a *vertex* in a graph and the dependency between the courses can be modeled as a directed edge between two vertex.

And the problem to determine if one could build a valid schedule of courses that satisfies all the dependencies (*i.e.* **constraints**) would be equivalent to determine if the corresponding graph is a **DAG** (*Directed Acyclic Graph*), *i.e.* there is no cycle existed in the graph.



A typical strategy for graph traversal problems would be backtracking (<https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/>) or simply DFS (depth-first search).

Here let us start with the backtracking algorithm, which arguably might be more intuitive.

As a reminder, backtracking (<https://en.wikipedia.org/wiki/Backtracking>) is a general algorithm that is often applied to solve the constraint satisfaction problems (https://en.wikipedia.org/wiki/Constraint_satisfaction_problem), which incrementally builds candidates to the solutions, and abandons a candidate (*i.e.* backtracks) as soon as it determines that the candidate would not lead to a valid solution.

The general idea here is that we could enumerate each course (vertex), to check if it could form cyclic

dependencies (*i.e.* a cyclic path) starting from this course.

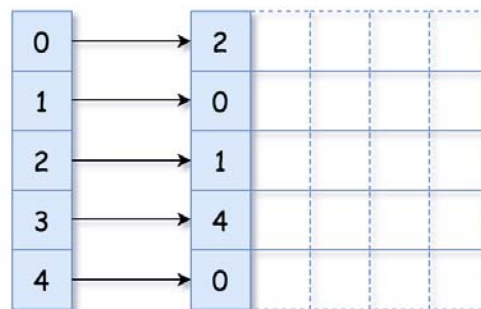
The check of cyclic dependencies for each course could be done via **backtracking**, where we incrementally follow the dependencies until either there is no more dependency or we come across a previously visited course along the path.

Algorithm

The overall structure of the algorithm is simple, which consists of three main steps:

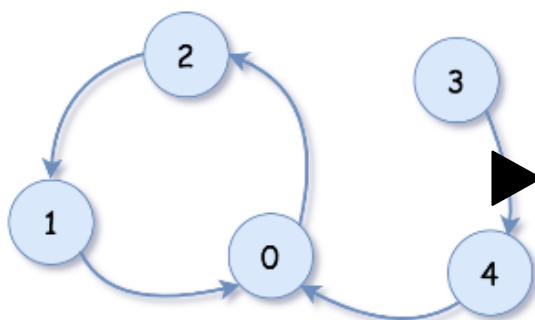
- Step 1). we build a graph data structure from the given list of course dependencies. Here we adopt the adjacency list data structure as shown below to represent the graph, which can be implemented via hashmap or dictionary. Each entry in the adjacency list represents a node which consists of a node index and a list of neighbors nodes that follow from the node.

Adjacency List




- Step 2). we then enumerate each node (course) in the constructed graph, to check if we could form a dependency cycle starting from the node.
- Step 3). we perform the cyclic check via backtracking, where we **breadcrumb** our path (*i.e.* mark the nodes we visited) to detect if we come across a previously visited node (hence a cycle detected). We also remove the breadcrumbs for each iteration.

Backtracking



Java

Python

 Copy

```

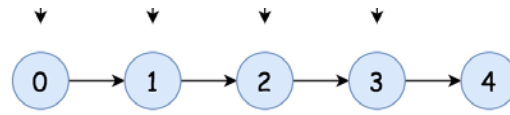
1 class Solution(object):
2     def canFinish(self, numCourses, prerequisites):
3         """
4         :type numCourses: int
5         :type prerequisites: List[List[int]]
6         :rtype: bool
7         """
8         from collections import defaultdict
9         courseDict = defaultdict(list)
10
11        for relation in prerequisites:
12            nextCourse, prevCourse = relation[0], relation[1]
13            courseDict[prevCourse].append(nextCourse)
14
15        path = [False] * numCourses
16        for currCourse in range(numCourses):
17            if self.isCyclic(currCourse, courseDict, path):
18                return False
19        return True
20
21
22    def isCyclic(self, currCourse, courseDict, path):
23        """
24        backtracking method to check that no cycle would be formed starting from currCourse
25        """
26        if path[currCourse]:
27            # come across a previously visited node, i.e. detect the cycle
28            return True
29
30        # before backtracking, mark the node in the path
31        path[currCourse] = True
32
33        # backtracking
34        ret = False
35        for child in courseDict[currCourse]:
36            ret = self.isCyclic(child, courseDict, path)
37            if ret: break
38
39        # after backtracking, remove the node from the path
40        path[currCourse] = False
41        return ret

```

Complexity

- Time Complexity: $\mathcal{O}(|E| + |V|^2)$ where $|E|$ is the number of dependencies, $|V|$ is the number of courses and d is the maximum length of acyclic paths in the graph.
 - First of all, it would take us $|E|$ steps to build a graph in the first step.
 - For a single round of backtracking, in the worst case where all the nodes chained up in a line, it would take us maximum $|V|$ steps to terminate the backtracking.

Backtracking steps	4	3	2	1

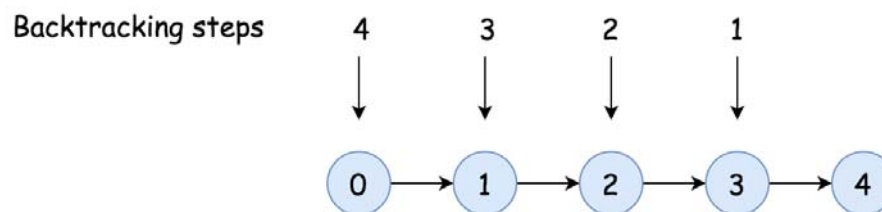


- Again, follow the above worst scenario where all nodes are chained up in a line, it would take us in total $\sum_{i=1}^{|V|} i = \frac{(1+|V|) \cdot |V|}{2}$ steps to finish the check for all nodes.
- As a result, the overall time complexity of the algorithm would be $\mathcal{O}(|E| + |V|^2)$.
- Space Complexity: $\mathcal{O}(|E| + |V|)$, with the same denotation as in the above time complexity.
 - We built a graph data structure in the algorithm, which would consume $|E| + |V|$ space.
 - In addition, during the backtracking process, we employed a sort of bitmap (path) to keep track of all visited nodes, which consumes $|V|$ space.
 - Finally, since we implement the function in recursion, which would incur additional memory consumption on call stack. In the worst case where all nodes are chained up in a line, the recursion would pile up $|V|$ times.
 - Hence the overall space complexity of the algorithm would be $\mathcal{O}(|E| + 3 \cdot |V|) = \mathcal{O}(|E| + |V|)$.

Approach 2: Postorder DFS (Depth-First Search)

Intuition

As one might notice that, with the above backtracking algorithm, we would visit certain nodes multiple times, which is not the most efficient way.



For instance, in the above graph where the nodes are chained up in a line, the backtracking algorithm would end up of being a nested two-level iteration over the nodes, which we could rewrite as the following pseudo code:

```
for i in range(0, len(nodes)):
    # start from the current node to check if a cycle might be formed.
    for j in range(i, len(nodes)):
        isCyclic(nodes[j], courseDict, path)
```

One might wonder that if there is a better algorithm that visits each node once and only once. And the answer is yes.

In the above example, for the first node in the chain, once we've done the check that there would be no cycle formed starting from this node, we don't have to do the same check for all the nodes in the downstream.

The rationale is that given a node, if the subgraph formed by all descendant nodes from this node has no cycle, then adding this node to the subgraph would not form a cycle either.

From the perspective of graph traversal, the above rationale could be implemented with the strategy of **postorder DFS** (depth-first search), in which strategy we visit a node's descendant nodes before the node itself.

Algorithm

We could implement the postorder DFS based on the above backtracking algorithm, by simply adding another bitmap (*i.e.* `checked[node_index]`) which indicates whether we have done the cyclic check starting from a particular node.

Here are the breakdowns of the algorithm, where the first 2 steps are the same as in the previous backtracking algorithm.

- Step 1). We build a graph data structure from the given list of course dependencies.
- Step 2). We then enumerate each node (course) in the constructed graph, to check if we could form a dependency cycle starting from the node.
- Step 3.1). We check if the current node has been *checked* before, otherwise we enumerate through its child nodes via backtracking, where we **breadcrumb** our path (*i.e.* mark the nodes we visited) to detect if we come across a previously visited node (hence a cycle detected). We also remove the breadcrumbs for each iteration.
- Step 3.2). Once we visited all the child nodes (*i.e.* postorder), we mark the current node as `checked` .

Java

Python

 Copy

```
1 class Solution(object):
2     def canFinish(self, numCourses, prerequisites):
3         """
4         :type numCourses: int
5         :type prerequisites: List[List[int]]
6         :rtype: bool
7         """
8         from collections import defaultdict
9         courseDict = defaultdict(list)
10
11        for relation in prerequisites:
12            nextCourse, prevCourse = relation[0], relation[1]
13            courseDict[prevCourse].append(nextCourse)
14
15        checked = [False] * numCourses
16        path = [False] * numCourses
17
18        for currCourse in range(numCourses):
19            if self.isCyclic(currCourse, courseDict, checked, path):
20                return False
21        return True
22
23
24    def isCyclic(self, currCourse, courseDict, checked, path):
25        """ """
26        # 1). bottom-cases
27        if checked[currCourse]:
28            # this node has been checked, no cycle would be formed with this node.
29            return False
30        if path[currCourse]:
31            # came across a marked node in the path, cyclic !
32            return True
33
34        # 2). postorder DFS on the children nodes
35        # mark the node in the path
36        path[currCourse] = True
37
38        ret = False
39        # postorder DFS, to visit all its children first.
40        for child in courseDict[currCourse]:
41            ret = self.isCyclic(child, courseDict, checked, path)
42            if ret: break
43
44        # 3). after the visits of children, we come back to process the node itself
45        # remove the node from the path
46        path[currCourse] = False
47
48        # Now that we've visited the nodes in the downstream,
49        # we complete the check of this node.
50        checked[currCourse] = True
51        return ret
```

Note, one could also use a single bitmap with 3 states such as `not_visited`, `visited`, `checked`,

rather than having two bitmaps as we did in the algorithm, though we argue that it might be clearer to have two separated bitmaps.

Complexity

- Time Complexity: $\mathcal{O}(|E| + |V|)$ where $|V|$ is the number of courses, and $|E|$ is the number of dependencies.
 - As in the previous algorithm, it would take us $|E|$ time complexity to build a graph in the first step.
 - Since we perform a postorder DFS traversal in the graph, we visit each vertex and each edge once and only once in the worst case, i.e. $|E| + |V|$.
- Space Complexity: $\mathcal{O}(|E| + |V|)$, with the same denotation as in the above time complexity.
 - We built a graph data structure in the algorithm, which would consume $|E| + |V|$ space.
 - In addition, during the backtracking process, we employed two bitmaps (`path` and `visited`) to keep track of the visited path and the status of check respectively, which consumes $2 \cdot |V|$ space.
 - Finally, since we implement the function in recursion, which would incur additional memory consumption on call stack. In the worst case where all nodes chained up in a line, the recursion would pile up $|V|$ times.
 - Hence the overall space complexity of the algorithm would be $\mathcal{O}(|E| + 4 \cdot |V|) = \mathcal{O}(|E| + |V|)$.

Approach 3: Topological Sort

Intuition

Actually, the problem is also known as topological sort (https://en.wikipedia.org/wiki/Topological_sorting) problem, which is to find a global order for all nodes in a DAG (*Directed Acyclic Graph*) with regarding to their dependencies.

A linear algorithm was first proposed by Arthur Kahn (https://en.wikipedia.org/wiki/Topological_sorting) in 1962, in his paper of "Topological order of large networks" (<https://dl.acm.org/doi/10.1145/368996.369025>). The algorithm returns a topological order if there is any. Here we quote the pseudo code of the Kahn's algorithm from wikipedia as follows:

L = Empty list that will contain the sorted elements

S = Set of all nodes **with** no incoming edge

while S **is** non-empty **do**

 remove a node n **from** S

 add n to tail of 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 graph has edges **then**

return error (graph has at least one cycle)

else

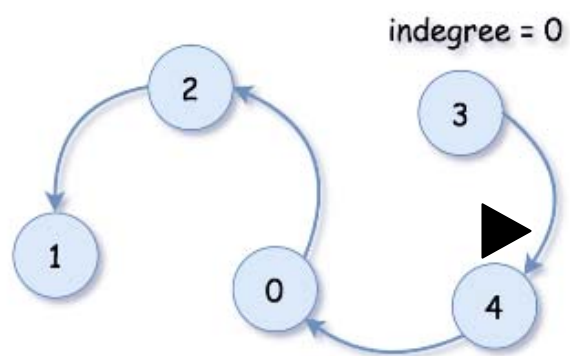
return L (a topologically sorted order)

To better understand the above algorithm, we summarize a few points here:

- In order to find a global order, we can start from those nodes which do not have any prerequisites (*i.e.* indegree of node is zero), we then incrementally add new nodes to the global order, following the dependencies (edges).
- Once we follow an edge, we then remove it from the graph.
- With the removal of edges, there would more nodes appearing without any prerequisite dependency, in addition to the initial list in the first step.
- The algorithm would terminate when we can no longer remove edges from the graph. There are two possible outcomes:
 - 1). If there are still some edges left in the graph, then these edges must have formed certain cycles, which is similar to the deadlock situation. It is due to these cyclic dependencies that we cannot remove them during the above processes.
 - 2). Otherwise, *i.e.* we have removed all the edges from the graph, and we got ourselves a topological order of the graph.


Algorithm

Following the above intuition and pseudo code, here we list some sample implementations.



Java

Python

 Copy

```

1 class GNode(object):
2     """ data structure represent a vertex in the graph."""
3     def __init__(self):
4         self.inDegrees = 0
5         self.outNodes = []
6
7 class Solution(object):
8     def canFinish(self, numCourses, prerequisites):
9         """
10         :type numCourses: int
11         :type prerequisites: List[List[int]]
12         :rtype: bool
13         """
14         from collections import defaultdict, deque
15         # key: index of node; value: GNode
16         graph = defaultdict(GNode)
17
18         totalDeps = 0
19         for relation in prerequisites:
20             nextCourse, prevCourse = relation[0], relation[1]
21             graph[prevCourse].outNodes.append(nextCourse)
22             graph[nextCourse].inDegrees += 1
23             totalDeps += 1
24
25         # we start from courses that have no prerequisites.
26         # we could use either set, stack or queue to keep track of courses with no dependence.
27         nodepCourses = deque()
28         for index, node in graph.items():
29             if node.inDegrees == 0:
30                 nodepCourses.append(index)
31
32         removedEdges = 0
33         while nodepCourses:
34             # pop out course without dependency
35             course = nodepCourses.pop()
36
37             # remove its outgoing edges one by one
38             for nextCourse in graph[course].outNodes:
39                 graph[nextCourse].inDegrees -= 1
40                 removedEdges += 1
41             # while removing edges, we might discover new courses with prerequisites
42             removed, i.e. new courses without prerequisites.
43             if graph[nextCourse].inDegrees == 0:
44                 nodepCourses.append(nextCourse)
45
46         if removedEdges == totalDeps:
47             return True
48         else:
49             # if there are still some edges left, then there exist some cycles
50             # Due to the dead-lock (dependencies), we cannot remove the cyclic edges
51             return False

```

Note that we could use different types of containers, such as Queue, Stack or Set, to keep track of the nodes that have no incoming dependency, *i.e.* $\text{indegree} = 0$. Depending on the type of container, the resulting topological order would be different, though they are all valid.

Complexity

- Time Complexity: $\mathcal{O}(|E| + |V|)$ where $|V|$ is the number of courses, and $|E|$ is the number of dependencies.
 - As in the previous algorithm, it would take us $|E|$ time complexity to build a graph in the first step.
 - Similar with the above postorder DFS traversal, we would visit each vertex and each edge once and only once in the worst case, i.e. $|E| + |V|$.
 - As a result, the overall time complexity of the algorithm would be $\mathcal{O}(2 \cdot |E| + |V|) = \mathcal{O}(|E| + |V|)$.
- Space Complexity: $\mathcal{O}(|E| + |V|)$, with the same denotation as in the above time complexity.
 - We built a graph data structure in the algorithm, which would consume $|E| + |V|$ space.
 - In addition, we use a container to keep track of the courses that have no prerequisite, and the size of the container would be bounded by $|V|$.
 - As a result, the overall space complexity of the algorithm would be $\mathcal{O}(|E| + 2 \cdot |V|) = \mathcal{O}(|E| + |V|)$.

Rate this article:

◀ Previous (/articles/decode-ways/)

Next ▶ (/articles/roman-to-integer/)

Comments: **13**

Sort By ▼



Type comment here... (Markdown is supported)

👁 Preview

Post



(/spyhippo)

spyHippo (/spyhippo) ★ 52 🕒 March 13, 2020 11:24 AM

Am I tripping or the illustration in Approach 3 is ... WRONG? After 3 and 4 are done, 0 still has indegree of 1 from node 1 ... How could there be a toposort here?

14 ▲ ▼ | 🔄 Share | ↩ Reply

SHOW 3 REPLIES



(/ufdeveloper)

ufdeveloper (/ufdeveloper) ★ 36 🕒 March 16, 2020 8:44 PM

This is a hard problem

20 ▲ ▼ | 🔄 Share | ↩ Reply

SHOW 1 REPLY



(/_noexcuses)

_noexcuses (/_noexcuses) ★ 200 🕒 April 26, 2020 3:39 PM

There is no such thing as $O(2E + V)$. It's $O(E + V)$. You **drop the constants**. How come the solution writers not know that?

3 ^ v | Share | Reply

SHOW 2 REPLIES



(/liaison)

liaison (/liaison) 🛡 STAFF ★ 5270 🕒 March 4, 2020 1:27 PM

hi @Algorithm0110 (<https://leetcode.com/algorithm0110>), for a SINGLE DFS cycle check in the approach 1, the maximal number of steps it requires would be $O(E)$, which is bounded by the number of edges in the graph, as I mentioned about the scenario where all nodes are chained up in a line. Since in a single round of DFS, for a DAG, normally we would not iterate all nodes in a graph.

Read More

3 ^ v | Share | Reply



(/algorithm0110)

Algorithm0110 (/algorithm0110) ★ 3 🕒 March 4, 2020 3:51 AM

Approach 1 time complexity seems incorrect. DFS cycle check should be $O(V+E)$, which invalidates the analysis.

2 ^ v | Share | Reply



(/monester)

monester (/monester) ★ 66 🕒 March 15, 2020 8:02 PM

tbh approach 1 is the only answer I can come up with in an interview. xD

3 ^ v | Share | Reply



(/nitkk_r_gaurav)

nitkk_r_gaurav (/nitkk_r_gaurav) ★ 1 🕒 March 21, 2020 2:25 AM

Approach 3 is flawed. You cannot move beyond node 4

0 ^ v | Share | Reply



(/mxsokal)

mxsokal (/mxsokal) ★ 0 🕒 4 hours ago

Basically, the problem is just to find out whether the graph is acyclic. That's it.

0 ^ v | Share | Reply



(/sergiitk)

sergiitk (/sergiitk) ★ 14 🕒 8 hours ago

Approach 3 in Python. No need to track `totalDeps`. Just compare it with the len of edges. Lines 45-50 can be replaced with simple

```
return removedEdges == len(prerequisites)
```

Read More

0 ^ v | Share | Reply



(/carlos6125)

carlos6125 (/carlos6125) ★ 0 🕒 May 20, 2020 12:25 AM

When calculating the time complexity for approach 1 - How did we get $(1 + V)(V) / 2$? What does the plus 1 and the divide by 2 represent?

0 ^ v | Share | Reply

Copyright © 2020 LeetCode

[Help Center \(/support/\)](/support/) | [Terms \(/terms/\)](/terms/) | [Privacy \(/privacy/\)](/privacy/)

[United States \(/region/\)](/region/)