Previous Next

207. Course Schedule

March 2, 2020 | 42.9K views

Average Rating: 4.10 (29 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.
```

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
             also have finished course 1. So it is impossible.
```

Constraints:

- The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
- You may assume that there are no duplicate edges in the input prerequisites.
- `1 <= numCourses <= 10^5`

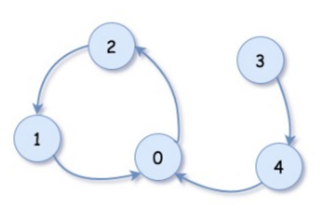
Solution

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 or simply DFS (depth-first search).

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

> As a reminder, backtracking is a general algorithm that is often applied to solve the constraint satisfaction problems, 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.

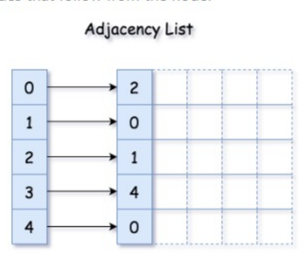
The general idea here is that we could enumerate each course (vertex), to check if it could form cyclic dependencies (*i.e.* a cyclic path) starting from this course.

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.

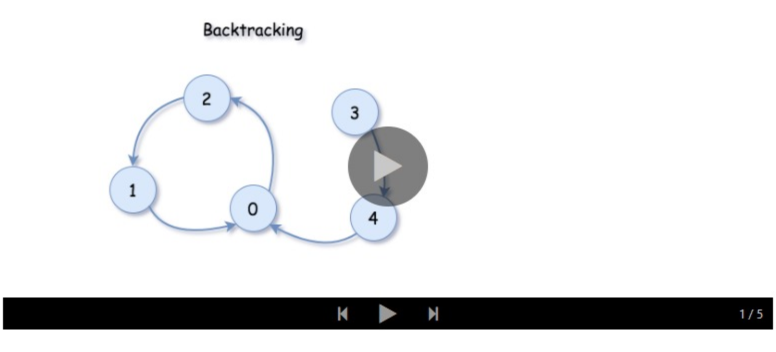
Algorithm

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.



- 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.

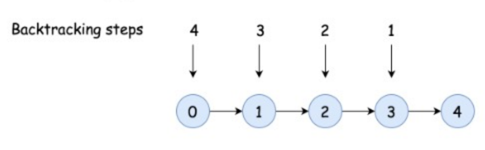


Java Python Copy

```python
class Solution(object):
    def canFinish(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: bool
        """
        from collections import defaultdict
        courseDict = defaultdict(list)

        for relation in prerequisites:
            nextCourse, prevCourse = relation[0], relation[1]
            courseDict[prevCourse].append(nextCourse)

        path = [False] * numCourses
        for currCourse in range(numCourses):
            if self.isCyclic(currCourse, courseDict, path):
                return False
        return True


    def isCyclic(self, currCourse, courseDict, path):
        """
        backtracking method to check that no cycle would be formed starting from currCourse
        """
        if path[currCourse]:
            # come across a previously visited node, i.e. detect the cycle
```

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.

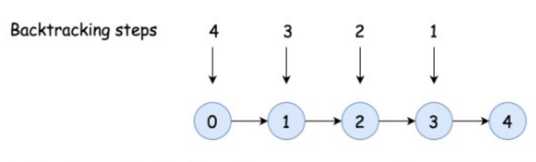


  - Again, follow the above worst scenario where all nodes are chained up in a line, it would take us in total $\sum_{i=1}^{|V|} i = \frac{(1+|V|)\cdot|V|}{2}$ steps to finish the check for all nodes.
  - As a result, the overall time complexity of the algorithm would be $\mathcal{O}(|E| + |V|^2)$.
- Space 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:

```python
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

```python
class Solution(object):
    def canFinish(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: bool
        """
        from collections import defaultdict
        courseDict = defaultdict(list)

        for relation in prerequisites:
            nextCourse, prevCourse = relation[0], relation[1]
            courseDict[prevCourse].append(nextCourse)

        checked = [False] * numCourses
        path = [False] * numCourses

        for currCourse in range(numCourses):
            if self.isCyclic(currCourse, courseDict, checked, path):
                return False
        return True


    def isCyclic(self, currCourse, courseDict, checked, path):
        """   """
        # 1). bottom-cases
        if checked[currCourse]:
```

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 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 in 1962, in his paper of "Topological order of large networks". 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:

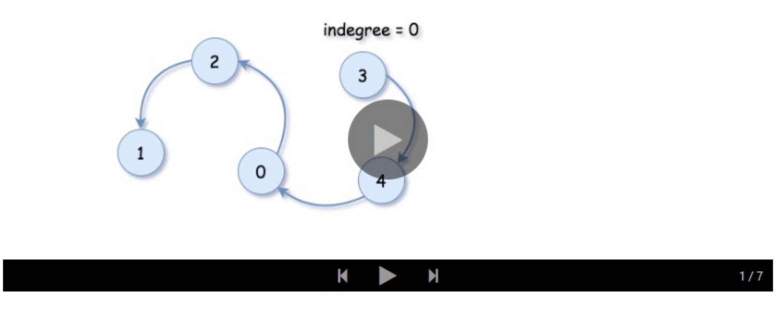
```
L = Empty list that will contain the sorted elements
S = Set of all nodes with no incoming edge

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

```python
class GNode(object):
    """  data structure represent a vertex in the graph."""
    def __init__(self):
        self.inDegrees = 0
        self.outNodes = []

class Solution(object):
    def canFinish(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: bool
        """
        from collections import defaultdict, deque
        # key: index of node; value: GNode
        graph = defaultdict(GNode)

        totalDeps = 0
        for relation in prerequisites:
            nextCourse, prevCourse = relation[0], relation[1]
            graph[prevCourse].outNodes.append(nextCourse)
            graph[nextCourse].inDegrees += 1
            totalDeps += 1

        # we start from courses that have no prerequisites.
        # we could use either set, stack or queue to keep track of courses with no dependence.
        nodepCourses = deque()
```

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.* `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 Next

Comments: 18 Sort By

Type comment here... (Markdown is supported)

Preview Post

spyHippo ★ 54 March 13, 2020 8:54 PM
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?
15 Share Reply
SHOW 3 REPLIES

ufdeveloper ★ 59 March 17, 2020 6:14 AM
This is a hard problem
36 Share Reply
SHOW 4 REPLIES

liaison STAFF ★ 5661 March 4, 2020 11:57 PM
hi @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.
Imagine in another extreme scenario, where we have many nodes but no edge, in this case, the DFS
Read More
4 Share Reply
SHOW 1 REPLY

chu_steven ★ 4 June 13, 2020 2:20 AM
Can you provide a visual example of Approach (3) for a graph that *does* have a cycle? That'd be helpful :)
3 Share Reply

Algorithm0110 ★ 3 March 4, 2020 2:21 PM
Approach 1 time complexity seems incorrect. DFS cycle check should be O(V+E), which invalidates the analysis.
2 Share Reply

_noexcuses ★ 235 April 27, 2020 1:09 AM
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 Share Reply
SHOW 3 REPLIES

monester ★ 75 March 16, 2020 5:32 AM
tbh approach 1 is the only answer I can come up with in an interview. xD
3 Share Reply

kushalmahajan ★ 32 July 1, 2020 11:25 PM
For topological sort the solution given in Approach 2 of https://leetcode.com/problems/course-schedule-ii/solution/ is better than the premium solution here. Maybe the article with an update to that approach and animation will make it a one stop to build intuition.
1 Share Reply

nostradamus29 ★ 1 June 28, 2020 1:25 PM
Hey coders.. when I was first solving this problem, I build a hashmap like this -
For prerequisites = [[1,0], [1,2], [2,3]], my hashmap is -
1 -> 0,2
2 -> 3
So, I go through dependencies of 1 which are 0,2.
Read More
1 Share Reply
SHOW 1 REPLY

nitkkr_gaurav ★ 1 March 21, 2020 11:55 AM
Approach 3 is flawed. You cannot move beyond node 4
0 Share Reply

1 2