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 the vertices.

And the problem to determine if we could build a valid schedule of courses that satisfy all the dependencies can be equivalent to determine if there is a cycle in the graph. For example, in the example 1 (DEMO) of the LeetCode's page, it 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 problems that exhibit the characteristic of a sequence of choices contributing to the solutions, and incrementally explore such choices to backtrack, as soon as it determines that the candidate choice would not lead to a valid solution.

The general idea here is that we could enumerate each course (vertex), to check if it would lead to a cycle in the graph, and if it would, we would backtrack.

The check of cycle dependency can be conducted by the idea of **backtracking** where we incrementally follow the dependencies until either there is no more dependency or we arrive at a course that we have visited before.

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, where each vertex represents a course, and each directed edge leads to a prerequisite course, which can be implemented as hashmap or dictionary. Each entry in the adjacency list represents a node which consists of a node index and a list of its neighbors (prerequisite courses).

- Step 2). we then enumerate each node (course) in the constructed graph, to check if we could arrive at a cycle starting from the node. Since we would need to keep track of the nodes that we have visited, we would also put in such the nodes we visited to detect if we come across a cycle, which would lead to a dead-end. Meanwhile, we also remove the visited nodes for each iteration.

- Step 3). we then enumerate each node (course) in the constructed graph, to check if we could arrive at a cycle starting from the node. Since we would need to keep track of the nodes that we have visited, we would also put in such the nodes we visited to detect if we come across a cycle, which would lead to a dead-end. Meanwhile, we also remove the visited nodes for each iteration.

Backtracking

```
class Solution:
    def canFinish(self, numCourses, prerequisites):
        # Step 1). build a graph data structure
        graph = {}
        for course in range(numCourses):
            graph[course] = []
        for [c, p] in prerequisites:
            graph[c].append(p)
        # Step 2). check if there is a cycle starting from each node
        visited = set()
        for course in range(numCourses):
            if self.isCycle(course, graph, visited):
                return False
        # Step 3). check if there is a cycle starting from each node
        visited = set()
        for course in range(numCourses):
            if self.isCycle(course, graph, visited):
                return False
        return True

    def isCycle(self, curCourse, graph, visited):
        # backtracking method to check that no cycle exists in the current
        # starting from curCourse
        if curCourse in visited:
            return True
        visited.add(curCourse)
        for neighbor in graph[curCourse]:
            if self.isCycle(neighbor, graph, visited):
                return True
        visited.remove(curCourse)
        return False
```

Complexity

- Time Complexity: $\mathcal{O}(E + V)$ where $\mathcal{O}(E)$ is the number of dependencies, $\mathcal{O}(V)$ is the number of courses and $\mathcal{O}$ is the maximum length of all paths in the graph.
  - First of all, it would take $\mathcal{O}$ steps to build a graph in the first step. In the second step, for each node, we would check if there is a cycle starting from it. In the worst case, it would take a maximum $\mathcal{O}$ steps to terminate the backtracking.

  - Again, follow the above search pattern and we would arrive at the shortest path in a tree, it would take no more than $\sum_{i=1}^{V-1} \mathcal{O}^i$ steps to finish the backtracking in the worst case.
  - As a result, the overall time complexity of the algorithm would be $\mathcal{O}(E + \mathcal{O}^V)$.
- Space Complexity: $\mathcal{O}(E + \mathcal{O}^V)$ with the same denotation as in the above time complexity.
  - We built a graph data structure in the algorithm, which would consume $\mathcal{O}(E + V)$ space.
  - In the second step of the backtracking process, we employed a set of hashmap / path to keep track of the visited nodes, which consume $\mathcal{O}^V$ space.
  - Finally, since we implement the function in recursion, which would incur additional memory consumption on the stack in the worst case, we would arrive at nodes as shallow as in a tree, the recursion would pile up $\mathcal{O}^V$ stacks.
  - Hence the overall space complexity of the algorithm would be $\mathcal{O}(E + V) + \mathcal{O}^V + \mathcal{O}^V$, which is $\mathcal{O}(E + \mathcal{O}^V)$.

Approach 2: Postorder DFS (Depth-First Search)

Intuition

As we might notice that, with the above backtracking algorithm, we would visit certain nodes multiple times, which is the main drawback.

For instance, in the above graph where the nodes are charted up in a tree, the leaf nodes would be visited multiple times. Since we would visit the leaf nodes, which are most valuable in the following graph code.

```
for i in range(0, numNodes):
    # start from the current node to check if a cycle might be form
    curNode = i
    if self.isCycle(curNode, graph, visited):
        return False
```

You might wonder that there is a better approach that also each nodes once and only once. And the answer is yes.

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

The rationale is that given a node, if the subgraph formed by all descendant nodes (children) of the node, then setting this node to be visitable 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 only if all its descendant nodes before the node visited.

Algorithm

We could implement the postorder DFS based on the above backtracking algorithm, by simply adding another interval (i.e. checkedNodes) set / hashmap which indicates whether we have form the cycle, check starting from a particular node.

Here, we would modify the step 2). and step 3). of the algorithm, where the same as in the previous backtracking algorithm.

- Step 2). We build a graph data structure from the given list of course dependencies.
- Step 3). Enumerate each node (course) in the graph, to check if we could form a dependency cycle starting from the node. Since we would need to keep track of the nodes that we have visited, we would also put in such the nodes we visited to detect if we come across a cycle, which would lead to a dead-end. Meanwhile, we also remove the visited nodes for each iteration.
- Step 3.2). Enumerate each of the child nodes (i.e. postorder), we mark the current node as checked.

```
class Solution:
    def canFinish(self, numCourses, prerequisites):
        # Step 1). build a graph
        graph = {}
        for course in range(numCourses):
            graph[course] = []
        for [c, p] in prerequisites:
            graph[c].append(p)
        # Step 2). check if there is a cycle starting from each node
        visited = set()
        for course in range(numCourses):
            if self.isCycle(course, graph, visited):
                return False
        # Step 3). check if there is a cycle starting from each node
        visited = set()
        for course in range(numCourses):
            if self.isCycle(course, graph, visited):
                return False
        return True

    def isCycle(self, curCourse, graph, checked):
        # Step 3). DFS
        # Step 3.2). postorder
        if curCourse in checked:
            return False
        for neighbor in graph[curCourse]:
            if self.isCycle(neighbor, graph, checked):
                return True
        checked.add(curCourse)
        return False
```

Note, one could also use a single hmap with 2 states such as not_visited, checking_path, and checked, instead of two separate hashmaps. The rationale behind this approach is that it might be faster to hash two separate hmaps.

Complexity

- Time Complexity: $\mathcal{O}(E + V)$ where $\mathcal{O}(E)$ is the number of dependencies, and $\mathcal{O}$ is the number of dependencies.
  - As in the previous algorithm, it would take $\mathcal{O}$ time complexity to build a graph in the first step.
  - In the second step of the backtracking process in the graph, we visit each vertex and each edge once and only once in the worst case, i.e. $\mathcal{O}$ + $\mathcal{O}$.
- Space Complexity: $\mathcal{O}(E + \mathcal{O})$ with the same denotation as in the above time complexity.
  - We built a graph data structure in the algorithm, which would consume $\mathcal{O}(E + V)$ space.
  - In the second step of the backtracking process, we employed two hashmaps / path and checked to keep track of the visited path and the state of check respectively, which consumes $\mathcal{O}$ space.
  - Finally, since we implement the function in recursion, which would incur additional memory consumption on the stack, in the worst case where all nodes charted up in a line, the recursion would pile up $\mathcal{O}$ stacks.
  - Hence the overall space complexity of the algorithm would be $\mathcal{O}(E + V) + \mathcal{O} + \mathcal{O}$.

Approach 3: Topological Sort

Intuition

Actually, the problem is also known as topological sort problem, which is by far a popular one for all vendors in LeetCode (Standard Apache Graph with Spring Framework is used).

The topological algorithm was first proposed by Arthur Kahn in 1962, in his paper of "Topological order of large networks". The algorithm returns a topological order of the given graph, 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 view, we can start from those nodes with no dependencies, and then we can remove the edges from the graph, and then we can add any nodes to the global order following the dependencies (edges).
- Step 1). Initialize the global order with an empty list.
- With the removal of edges, there would more nodes appearing with no prerequisite dependencies, in addition to the initial list in the first step.
- Step 2). Enumerate each node in the current list, and add to remove edges from the graph. There are two possible outcomes:
  - 1). If there are still some edges left in the graph, then there edges must be forming a cycle, which means that there is no valid topological ordering. It is due to those edges dependencies that we cannot enumerate all nodes during the above process.
  - 2). Otherwise, if we have removed all the edges from the graph, and we get somehow a topological order of the graph.

Algorithm

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


```
class DAGVertex:
    # any structure required a vertex in the graph.
    # for example:
    # self.inDegree = 0
    # self.neighbors = []
    pass

class DAGGraph(object):
    def __init__(self, numCourses, prerequisites):
        # type numCourses: int
        # type prerequisites: List[List[int]]
        # type self: None
        # type self.graph: dict[int]
        # type self.inDegreeMap: dict[int]
        self.graph = {}
        self.inDegreeMap = {}
        for i in range(0, numCourses):
            self.graph[i] = []
            self.inDegreeMap[i] = 0
        for [c, p] in prerequisites:
            # if c not in self.graph:
            #.....
            self.graph[p].append(c)
            self.inDegreeMap[c] = self.inDegreeMap[c] + 1
        # if graph has cycle, then have no prerequisites.
        # if graph has no cycle, then at least one course has no
        # prerequisites.
        # type self.inDegreeMap: dict[int]
        # type self.graph: dict[int]
        # type self.numNodes: int
        self.numNodes = 0
        for key in self.graph:
            self.numNodes = self.numNodes + len(self.graph[key])
        self.numNodes = self.numNodes + len(self.graph)
        # type self.numNodes: int
        pass

    # an start from course that have no prerequisites.
    # return True if successfully, and False if there is cycle.
    # type self: DAGGraph
    # type return: bool
    def startFrom(self):
        if self.graph:
            return False
        else:
            return True
        pass

    # return True if succ
    # type self: DAGGraph
    # type return: bool
    def isSuccessful(self):
        if self.inDegreeMap:
            return False
        else:
            return True
        pass

    # start from course
    # type self: DAGGraph
    # type course: int
    # type return: None
    def startFrom(self, course):
        if course:
            return True
        else:
            return False
```