Course Schedule

There are a total of *n* courses you have to take, labeled from `0` to `n - 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?

For example:

```
2, [[1,0]]
```

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

```
2, [[1,0],[0,1]]
```

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

Note:
The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.

click to show more hints.

Hints:

1. This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

Solution 1

```
public boolean canFinish(int numCourses, int[][] prerequisites) {
    int[][] matrix = new int[numCourses][numCourses]; // i -> j
    int[] indegree = new int[numCourses];

    for (int i=0; i<prerequisites.length; i++) {
        int ready = prerequisites[i][0];
        int pre = prerequisites[i][1];
        if (matrix[pre][ready] == 0)
            indegree[ready]++; //duplicate case
        matrix[pre][ready] = 1;
    }

    int count = 0;
    Queue<Integer> queue = new LinkedList();
    for (int i=0; i<indegree.length; i++) {
        if (indegree[i] == 0) queue.offer(i);
    }
    while (!queue.isEmpty()) {
        int course = queue.poll();
        count++;
        for (int i=0; i<numCourses; i++) {
            if (matrix[course][i] != 0) {
                if (--indegree[i] == 0)
                    queue.offer(i);
            }
        }
    }
    return count == numCourses;
}
```

written by jiayu.liu.961 original link here

Solution 2

1. BFS(Topological Sort)

```
bool canFinish(int numCourses, vector<vector<int>>& prerequisites)
{
    vector<unordered_set<int>> matrix(numCourses); // save this directed graph
    for(int i = 0; i < prerequisites.size(); ++ i)
        matrix[prerequisites[i][1]].insert(prerequisites[i][0]);

    vector<int> d(numCourses, 0); // in-degree
    for(int i = 0; i < numCourses; ++ i)
        for(auto it = matrix[i].begin(); it != matrix[i].end(); ++ it)
            ++ d[*it];

    for(int j = 0, i; j < numCourses; ++ j)
    {
        for(i = 0; i < numCourses && d[i] != 0; ++ i); // find a node whose in-de
gree is 0

        if(i == numCourses) // if not find
            return false;

        d[i] = -1;
        for(auto it = matrix[i].begin(); it != matrix[i].end(); ++ it)
            -- d[*it];
    }

    return true;
}
```

2. DFS(Finding cycle)

```
bool canFinish(int numCourses, vector<vector<int>>& prerequisites)
{
    vector<unordered_set<int>> matrix(numCourses); // save this directed graph
    for(int i = 0; i < prerequisites.size(); ++ i)
        matrix[prerequisites[i][1]].insert(prerequisites[i][0]);

    unordered_set<int> visited;
    vector<bool> flag(numCourses, false);
    for(int i = 0; i < numCourses; ++ i)
        if(!flag[i])
            if(DFS(matrix, visited, i, flag))
                return false;
    return true;
}
bool DFS(vector<unordered_set<int>> &matrix, unordered_set<int> &visited, int b,
vector<bool> &flag)
{
    flag[b] = true;
    visited.insert(b);
    for(auto it = matrix[b].begin(); it != matrix[b].end(); ++ it)
        if(visited.find(*it) != visited.end() || DFS(matrix, visited, *it, flag))
            return true;
    visited.erase(b);
    return false;
}
```

written by makuiyu original link here

Solution 3

As suggested by the hints, this problem is equivalent to detecting a cycle in the graph represented by `prerequisites`. Both BFS and DFS can be used to solve it using the idea of **topological sort**. If you find yourself unfamiliar with these concepts, you may refer to their wikipedia pages. Specifically, you may only need to refer to the link in the third hint to solve this problem.

Since `pair<int, int>` is inconvenient for the implementation of graph algorithms, we first transform it to a graph. If course `u` is a prerequisite of course `v`, we will add a directed edge from node `u` to node `v`.

BFS

BFS uses the indegrees of each node. We will first try to find a node with`0` indegree. If we fail to do so, there must be a cycle in the graph and we return `false`. Otherwise we have found one. We set its indegree to be `-1` to prevent from visiting it again and reduce the indegrees of all its neighbors by `1`. This process will be repeated for `n` (number of nodes) times. If we have not returned `false`, we will return `true`.

```
class Solution {
public:
    bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {
        vector<unordered_set<int>> graph = make_graph(numCourses, prerequisites);
        vector<int> degrees = compute_indegree(graph);
        for (int i = 0; i < numCourses; i++) {
            int j = 0;
            for (; j < numCourses; j++)
                if (!degrees[j]) break;
            if (j == numCourses) return false;
            degrees[j] = -1;
            for (int neigh : graph[j])
                degrees[neigh]--;
        }
        return true;
    }
private:
    vector<unordered_set<int>> make_graph(int numCourses, vector<pair<int, int>>&
prerequisites) {
        vector<unordered_set<int>> graph(numCourses);
        for (auto pre : prerequisites)
            graph[pre.second].insert(pre.first);
        return graph;
    }
    vector<int> compute_indegree(vector<unordered_set<int>>& graph) {
        vector<int> degrees(graph.size(), 0);
        for (auto neighbors : graph)
            for (int neigh : neighbors)
                degrees[neigh]++;
        return degrees;
    }
};
```

DFS

For DFS, it will first visit a node, then one neighbor of it, then one neighbor of this neighbor... and so on. If it meets a node which was visited in the current process of DFS visit, a cycle is detected and we will return `false`. Otherwise it will start from another unvisited node and repeat this process till all the nodes have been visited. Note that you should make two records: one is to record all the visited nodes and the other is to record the visited nodes in the current DFS visit.

The code is as follows. We use a `vector<bool> visited` to record all the visited nodes and another `vector<bool> onpath` to record the visited nodes of the current DFS visit. Once the current visit is finished, we reset the `onpath` value of the starting node to `false`.

```
class Solution {
public:
    bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {
        vector<unordered_set<int>> graph = make_graph(numCourses, prerequisites);
        vector<bool> onpath(numCourses, false), visited(numCourses, false);
        for (int i = 0; i < numCourses; i++)
            if (!visited[i] && dfs_cycle(graph, i, onpath, visited))
                return false;
        return true;
    }
private:
    vector<unordered_set<int>> make_graph(int numCourses, vector<pair<int, int>>&
prerequisites) {
        vector<unordered_set<int>> graph(numCourses);
        for (auto pre : prerequisites)
            graph[pre.second].insert(pre.first);
        return graph;
    }
    bool dfs_cycle(vector<unordered_set<int>>& graph, int node, vector<bool>& onp
ath, vector<bool>& visited) {
        if (visited[node]) return false;
        onpath[node] = visited[node] = true;
        for (int neigh : graph[node])
            if (onpath[neigh] || dfs_cycle(graph, neigh, onpath, visited))
                return true;
        return onpath[node] = false;
    }
};
```

written by jianchao.li.fighter original link here