Graph Problems

207. Course Schedule (https://leetcode.com/problems/course-schedule/)

Medium

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?

Example 1:

```
Input: 2, [[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: 2, [[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 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.
- You may assume that there are no duplicate edges in the input prerequisites.

Performance

- Runtime: 44 ms, faster than 98.96% of Python3 online submissions for Course Schedule.
- Memory Usage: 14.9 MB, less than 56.70% of Python3 online submissions for Course Schedule.

Complexity Analysis

```
O(n + m) in time
O(1) in space
```

In [ ]:
```
from collections import defaultdict

class Graph:
    def __init__(self, number_nodes=0, edge_list=[]):
        self.number_nodes = number_nodes
        self.adj_dict = defaultdict(list)
        for edge in edge_list:
            self.adj_dict[edge[0]].append(edge[1])

    def hasCycle(self):
        visited = [False]*self.number_nodes
        node_on_path = [False]*self.number_nodes
        for v in range(self.number_nodes):
            if not(visited[v]):
                if self.hasCycleRec(v, visited, node_on_path):
                    return True
        return False

    def hasCycleRec(self, v0, visited, node_on_path):
        visited[v0] = True
        node_on_path[v0] = True
        for v1 in self.adj_dict[v0]:
            if node_on_path[v1]:
                return True
            if visited[v1] == False:
                if self.hasCycleRec(v1, visited, node_on_path):
                    return True
        node_on_path[v0] = False
        return False


class Solution:
    def canFinish(self, numCourses: 'int', prerequisites: 'List[List[int]]') -> 'bool':
        graph = Graph(numCourses, prerequisites)
        if graph.hasCycle():
            return False
        return True

my_sol = Solution()
print('Should print True:', my_sol.canFinish(2, [[1,0]]))
print('Should print False:', my_sol.canFinish(2, [[1,0],[0,1]]))
```

399. Evaluate Division (https://leetcode.com/problems/evaluate-division/)

Medium

Equations are given in the format A / B = k, where A and B are variables represented as strings, and k is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return -1.0.

Example:

```
Given a / b = 2.0, b / c = 3.0.
queries are: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .
return [6.0, 0.5, -1.0, 1.0, -1.0 ].
```

The input is:

```
vector<pair<string, string>> equations,
vector<double>&              values,
vector<pair<string, string>> queries,
```

where

```
equations.size() == values.size(),
```

and the values are positive. This represents the equations. Return vector.

According to the example above:

```
equations = [ ["a", "b"], ["b", "c"] ],
values = [2.0, 3.0],
queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].
```

The input is always valid. You may assume that evaluating the queries will result in no division by zero and there is no contradiction.

Performance

- Runtime: 36 ms, faster than 43.61% of Python3 online submissions for Evaluate Division.
- Memory Usage: 13.1 MB, less than 5.35% of Python3 online submissions for Evaluate Division.

Complexity Analysis

```
O(n) in time
O(n) in space
```

In [1]:

```
from collections import defaultdict

class Graph:
    def __init__(self, edges, weights):
        self.adj_list = defaultdict(list)
        self.edge_weights = defaultdict(float)
        for edge, weight in zip(edges, weights):
            self.adj_list[edge[0]].append(edge[1])
            self.edge_weights[(edge[0], edge[1])] = weight
            self.adj_list[edge[1]].append(edge[0])
            self.edge_weights[(edge[1], edge[0])] = 1.0/weight
        self.nodes = self. adj_list.keys()

    def nodesInGraph(self, nodes):
        for node in nodes:
            if not(node in self.nodes):
                return False
        return True

    def findPathValue(self, v1, v2):
        visited = {}
        for v in self.nodes:
            visited[v] = False
        path = []
        final_path = []
        self.findPathValueRec(v1, v2, visited, path, final_path)
        if len(final_path) > 0:
            return self.getPathValue(final_path[0])
        return -1.0

    def findPathValueRec(self, v1, v2, visited, path, final_path):
        visited[v1] = True
        path.append(v1)
        if v1 == v2:
            final_path.append(path[:])
            for k in visited:
                visited[k] = True
        for v in self.adj_list[v1]:
            if not(visited[v]):
                self.findPathValueRec(v, v2, visited, path, final_path)
        path.pop()
        visited[v1] = False

    def getPathValue(self, path):
        value = 1.0
        for i in range(len(path)-1):
            value *= self.edge_weights[(path[i], path[i+1])]
        return value

class Solution:
    def calcEquation(self, equations: 'List[List[str]]', values: 'List[float]', queries
        graph  = Graph(equations, values)
        query_values = []
        for query in queries:
            if graph.nodesInGraph(query):
                query_values.append(graph.findPathValue(query[0], query[1]))
            else:
                query_values.append(-1.0)
        return query_values

my_sol = Solution()
print('Should print [6.0, 0.5, -1.0, 1.0, -1.0]:', my_sol.calcEquation([ ["a", "b"], ["
[ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ]))
print('Should print [3.0]:', my_sol.calcEquation([ ["a","b"],["b","c"] ], [2.0,3.0], [[
```

```
Should print [6.0, 0.5, -1.0, 1.0, -1.0]: [6.0, 0.5, -1.0, 1.0, -1.0]
Should print [3.0]: [3.0]
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: