JavaScript Course Schedule II

Challenge

There is a total of `numCourses` courses you must take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [$a_i$, $b_i$]` indicates that you must take course `$b_i$` first if you want to take course `$a_i$`.

For example, the pair `[0, 1]`, indicates that to take course `0` you must first take course `1`.

Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

1st Example

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take
             course 1 you should have finished course 0. So,
             the correct course order is [0,1].`
```

2nd Example

```
Input: numCourses    = 4,
       prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3]
```

Example continues on next page...

```
Explanation: There are a total of 4 courses to take. To take
             course 3 you should have finished both courses 1
             and 2. Both courses 1 and 2 should be taken
             after you finished course 0. So, one correct
             course order is [0,1,2,3]. Another correct
             ordering is [0,2,1,3].
```

3rd Example

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

Constraints

- `1 <= numCourses <= 2000`
- `0 <= prerequisites.length <= numCourses * (numCourses - 1)`
- `prerequisites[i].length == 2`
- `0 <= a_i, b_i < numCourses`
- `a_i != b_i`
- All the pairs `[a_i, b_i]` are distinct.

Solution

```
const findOrder = (numCourses, prerequisites) => {
    const visited = new Set(),
          adjList = new Map(),
          result  = [];
    let visiting  = new Set();
```

Solution continues on next page...

```
    for (let i = 0; i < numCourses; i++) {
        adjList.set(i, []);
    }

    for (const item of prerequisites) {
        const [course, prereq] = item;

        adjList.get(course).push(prereq);
    }

    const dfs = (node) => {
        if (visited.has(node)) {
            return true;
        }

        if (visiting.has(node)) {
            return false;
        }

        visiting.add(node);

        const children = adjList.get(node);

        for (const child of children) {
            if (!dfs(child)) {
                return false;
            }
        }

        visited.add(node);

        result.push(node);

        return true;
    };
```

Solution continues on next page...

```
    for (const node of adjList.keys()) {
        if (!dfs(node)) return [];
    }

    return result;
};
```

Explanation

I've written a function called `findOrder` that takes in the number of courses (`numCourses`) and an array of prerequisites (`prerequisites`). The function returns an array (`result`) that represents the order in which the courses can be taken.

The function begins by initializing three data structures: `visited`, a set to keep track of visited nodes; `adjList`, a map to represent the adjacency list of the courses and their prerequisites; and `result`, an empty array to store the order of courses.

Next, a set called `visiting` is created to keep track of nodes that are currently being visited.

A loop is used to initialize the adjacency list for each course. The key of the map is the course number, and the value is an empty array.

Another loop is used to populate the adjacency list based on the prerequisites array. Each item in the array represents a prerequisite relationship, where the first element is the course, and the second element is the prerequisite for that course. The prerequisite is pushed into the corresponding array in the adjacency list.

It then defines a recursive function called `dfs` (depth-first search).

It takes in a node as an argument and recursively explores its children.

Inside the `dfs` function, it first checks if the node has been visited before. If it has, it returns true.

Then, it checks if the node is currently being visited. If it is, it means there is a cycle in the graph, so it returns `false`.

If the node is neither visited nor currently being visited, it adds the node to the `visiting` set.

It retrieves the children of the current node from the adjacency list.

It iterates over each child and recursively calls the `dfs` function on them. If any of the recursive calls return `false`, it means there is a cycle, so it returns `false`.

If all the children have been visited and there is no cycle, the current node is added to the `visited` set and pushed to the `result` array.

Finally, the `dfs` function is called for each node in the adjacency list. If any of the calls return `false`, it means there is a cycle, so an empty array is returned.

If all the nodes have been visited without any cycles, the `result` array is returned, representing the order in which the courses can be taken.

Author: Trevor Morin