1136. Parallel Courses

Medium Graph Topological Sort

Leetcode Link

Problem Description

The problem presents a scenario where you have `n` courses numbered from `1` to `n`. You are also given a list of pairs, `relations`, representing the prerequisite relationships between the courses. Each pair `[prevCourse, nextCourse]` indicates that `prevCourse` must be completed before `nextCourse` can be taken.

The goal is to determine the minimum number of semesters required to complete all `n` courses, given that you can take any number of courses in a single semester, but you must have completed their prerequisites in the previous semester. If it's not possible to complete all courses due to a circular dependency or any other reason, the result should be `-1`.

Intuition

To solve this problem efficiently, we use a graph-based approach known as Topological Sorting, which is usually applied to scheduling tasks or finding a valid order in which to perform tasks with dependencies.

Here's a step-by-step breakdown of the intuition behind the solution:

1. We consider each course as a node in a directed graph where an edge from node `prevCourse` to node `nextCourse` indicates that `prevCourse` is a prerequisite for `nextCourse`.
2. We construct an in-degree list to track the number of prerequisite courses for each course. A course with an in-degree of `0` means it has no prerequisite and can be taken immediately.
3. We then use a queue to keep track of all courses that have no prerequisites (in-degree of `0`). This set represents the courses we can take in the current semester.
4. For each semester, we dequeue all available courses (i.e., those with no prerequisites remaining) and decrease the in-degree of their corresponding next courses by `1`, representing that we have taken one of their prerequisites.
5. If the in-degree of these next courses drops to `0`, it means they have no other prerequisites left, so we add them to the queue to be available for the next semester.
6. We continue this process, semester by semester, keeping a counter to track the number of semesters necessary until there are no more courses left to take.
7. If at the end of this process, there are still courses left (i.e., `n` is not zero), it implies that there is a cycle or unsatisfiable dependency, and we return `-1`. Otherwise, we return the counter, which now holds the minimum number of semesters needed to take all courses.

Solution Approach

The implementation follows the topological sorting approach using the Kahn's algorithm pattern. Here's how the algorithm unfolds:

1. **Graph Representation**: The graph is represented using an adjacency list, `g`, which maps each course to a list of courses that depend on it (i.e., the courses that have it as a prerequisite). Additionally, an in-degree list, `indeg`, is maintained to keep track of the number of prerequisites for each course.
2. **Initialization**: An initial scan of the `relations` array is performed to populate the adjacency list and in-degree list. Each relation `[prev, next]` increments the in-degree of the `next` course to indicate an additional prerequisite.
3. **Queue of Available Courses**: A queue, `q`, is initialized with all the courses that have an in-degree of `0`. These are the courses that can be taken in the first semester as they have no prerequisites.
4. **Iterating Over Semesters**: We use a while loop to simulate passing semesters. In each iteration, we process all courses in the queue:
   - Increment the `ans` variable, which represents the number of semesters.
   - For each course taken (dequeued), decrement `n` to reflect that one course has been completed.
   - Update the in-degree of all "child" courses (i.e., courses that the current course is a prerequisite for) by decrementing their respective in-degree count.
   - If any "child" course's in-degree becomes `0`, enqueue it, meaning it can be taken in the next semester.
5. **Detecting Cycles or Infeasible Conditions**: After processing all possible courses, if the `n` counter is not zero, it suggests that not all courses can be taken due to cycles (circular prerequisites) or other unsolvable dependencies. In such a case, we return `-1`.
6. **Returning the Result**: If the loop completes successfully and `n` is zero, the `ans` variable contains the minimum number of semesters needed to take all courses, which is then returned.

The data structures employed in this solution are:

- A `defaultdict` for constructing the adjacency list graph representation.
- A list to maintain in-degree counts.
- A `deque` to efficiently add and remove available courses as we iterate over semesters.

The algorithms and patterns used are part of graph theory and include:

- Topological sorting to determine a valid order to take the courses.
- Kahn's algorithm for detecting cycles and performing the sort iteratively by removing nodes with zero in-degree.

Throughout the process, we maintain a balance between the courses taken and the updates to the graph's state, ensuring that courses are only considered available once all their prerequisites have been fulfilled.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have `n = 4` courses and the `relations` list as `[[1,2], [2,3], [3,4]]`. According to this list, you must complete course 1 before taking course 2, course 2 before course 3, and course 3 before course 4.

Using the approach described above, let's proceed step by step:

- **Graph Representation**: We construct a graph such that `g = {1: [2], 2: [3], 3: [4]}`, and the in-degree list `indeg = [0, 1, 1, 1]` (since the numbering starts at 1, you might need to adjust the indices accordingly).
- **Initialization**: We perform an initial scan and find that course 1 has no prerequisites (`in-degree = 0`), so it can be taken immediately.
- **Queue of Available Courses**: We initialize a queue `q` with course 1 in it, ready to be taken in the first semester.
- **Iterating Over Semesters**:
  - First Semester: We dequeue course 1 and decrement `n` to 3 (as we have one less course to complete). We then decrease the in-degree of course 2 to 0 and enqueue it for the next semester.
  - Second Semester: We dequeue course 2 from `q` and decrement `n` to 2. We update the in-degree of course 3 to 0 and enqueue it for the next semester.
  - Third Semester: Course 3 is dequeued, `n` is decremented to 1, the in-degree of course 4 is now 0, and it gets enqueued for the next semester.
  - Fourth Semester: Finally, we dequeue course 4, and `n` is now 0, which means all courses have been accounted for.
- **Detecting Cycles or Infeasible Conditions**: Through our iterations, we didn't encounter any cycles, and all indegree counts could be reduced to 0 properly, indicating that we can plan courses without conflict.
- **Returning the Result**: We've successfully queued and completed all courses, one per semester. Therefore, the minimum number of semesters required is 4, which is the value of our counter after the last course is taken.

The result is `4` semesters to complete the given courses following the pre-defined prerequisites-order.

Python Solution

```python
from collections import defaultdict, deque
from typing import List

class Solution:
    def minimumSemesters(self, total_courses: int, prerequisites: List[List[int]]) -> int:
        # Create a default dictionary to hold the adjacency list representation of the graph
        graph = defaultdict(list)
        # Initialize a list for incoming degree count of each node
        incoming_degree = [0] * total_courses

        # Populate the graph and update the incoming degree counts
        for previous_course, course in prerequisites:
            # Adjust indices to be 0-based instead of 1-based
            previous_course -= 1
            course -= 1
            graph[previous_course].append(course)
            incoming_degree[course] += 1

        # Initialize a queue with all courses having 0 incoming degree
        queue = deque([i for i, degree in enumerate(incoming_degree) if degree == 0])
        # Initialize a variable to count the number of semesters
        semesters = 0

        # Process nodes level by level (BFS)
        while queue:
            # Increment the semester count
            semesters += 1
            # Process all nodes in the current level
            for _ in range(len(queue)):
                course = queue.popleft()
                total_courses -= 1  # Decrement the number of courses remaining
                # Decrease the incoming degree of all adjacent courses
                for next_course in graph[course]:
                    incoming_degree[next_course] -= 1
                    # If all prerequisites are now satisfied, add it to the queue
                    if incoming_degree[next_course] == 0:
                        queue.append(next_course)

        # If all courses are covered, return the number of semesters
        # If there are still courses remaining with nonzero incoming degree,
        # it means there's a cycle, and we return -1 to indicate it's not possible to finish
        return semesters if total_courses == 0 else -1
```

Java Solution

```java
class Solution {
    public int minimumSemesters(int N, int[][] relations) {
        // Create an adjacency list to represent the graph structure.
        List<Integer>[] graph = new List[N];
        // Array to keep track of the in-degrees of nodes.
        int[] inDegree = new int[N];
        // Initialize each list in the graph array.
        Arrays.setAll(graph, i -> new ArrayList<>());
        // Array to keep track of in-degrees (number of prerequisites).
        int[] indegree = new int[N];

        // Build the graph and count the in-degrees of each course.
        for (int[] relation : relations) {
            int prevCourse = relation[0] - 1; // Convert to 0-based index.
            int nextCourse = relation[1] - 1; // Convert to 0-based index.
            graph[prevCourse].add(nextCourse);
            indegree[nextCourse]++;
        }

        // Queue to store courses with no prerequisites (in-degree of 0).
        Deque<Integer> queue = new ArrayDeque<>();
        // Enqueue all courses with no prerequisites.
        for (int i = 0; i < N; ++i) {
            if (indegree[i] == 0) {
                queue.offer(i);
            }
        }

        // Variable to keep count of the minimum number of semesters.
        int semesters = 0;
        // Perform BFS to process courses.
        while (!queue.isEmpty()) {
            semesters++; // Increase semester count.
            // Process courses at the current semester.
            for (int k = queue.size(); k > 0; --k) {
                int course = queue.poll(); // Dequeue a course.
                --N; // Decrease the count of the remaining courses.
                // Decrease in-degree for the subsequent courses and check if they become ready to take.
                for (int nextCourse : graph[course]) {
                    if (--indegree[nextCourse] == 0) {
                        queue.offer(nextCourse);
                    }
                }
            }
        }

        // If there are no more courses to take, return semesters, otherwise return -1 to indicate the graph is not acyclic.
        return N == 0 ? semesters : -1;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minimumSemesters(int N, vector<vector<int>>& relations) {
        // Graph representation where each node points to its successors
        vector<vector<int>> graph(N);

        // In-degree array to track the number of prerequisites for each course
        vector<int> indegree(N);

        // Build the graph and fill in the in-degree array
        for (auto& relation : relations) {
            int prevCourse = relation[0] - 1; // Convert to 0-indexed
            int nextCourse = relation[1] - 1; // Convert to 0-indexed

            // Add edge from prevCourse to nextCourse
            graph[prevCourse].push_back(nextCourse);

            // Increment the in-degree of the nextCourse
            ++indegree[nextCourse];
        }

        // Queue to perform topological sorting
        queue<int> queue;

        // Enqueue all the courses that have no prerequisites (in-degree of 0)
        for (int i = 0; i < N; ++i) {
            if (!indegree[i]) {
                queue.push(i);
            }
        }

        int semesters = 0; // Number of semesters needed to complete all the courses

        // Perform the topological sort using BFS
        while (!queue.empty()) {
            // Increment the semester count as all the nodes in the queue can be taken in the current semester
            ++semesters;

            // Process all nodes in the current level of the queue
            for (int levelSize = queue.size(); levelSize > 0; --levelSize) {
                int course = queue.front();
                queue.pop();
                --N; // One less course to consider

                // Check all the next courses that current course points to
                for (int nextCourse : graph[course]) {
                    // Decrement their in-degree and if it drops to 0, add it to the queue
                    if (--indegree[nextCourse] == 0) {
                        queue.push(nextCourse);
                    }
                }
            }
        }

        // If all courses are processed, we return the number of semesters, otherwise it's impossible (-1)
        return N == 0 ? semesters : -1;
    }
};
```

Typescript Solution

```typescript
function minimumSemesters(totalCourses: number, prerequisitePairs: number[][]): number {
    // Create an adjacency list to represent the course graph, initialized with empty arrays
    const courseGraph = Array.from({ length: totalCourses }, () => []);

    // Initialize an array to keep track of how many prerequisites each course has
    const indegree = new Array(totalCourses).fill(0);

    // Build the graph and fill in the indegree array by iterating over the prerequisitePairs
    for (const [prerequisite, course] of prerequisitePairs) {
        courseGraph[prerequisite - 1].push(course - 1);
        indegree[course - 1]++;
    }

    // Queue for courses that have no prerequisites, hence can be taken
    const queue: number[] = [];

    // Find the initial courses with no prerequisites and add them to the queue
    for (let i = 0; i < indegree.length; i++) {
        if (indegree[i] === 0) {
            queue.push(i);
        }
    }

    // Perform a breadth-first search (BFS) on the course graph
    while (queue.length > 0) {
        semesters++; // Start a new semester
        // Process all courses that are available to be taken in this semester
        for (let count = queue.length; count > 0; --count) {
            const currentCourse = queue.shift(); // Take the course (remove from queue)
            totalCourses--; // Decrement the count of courses remaining to be taken
            for (const nextCourse of courseGraph[currentCourse]) {
                // Decrement the in-degree of the adjacent courses
                if (--indegree[nextCourse] === 0) {
                    queue.push(nextCourse); // If no more prerequisites, add to the queue
                }
            }
        }
    }

    // If all courses have been taken, return the total number of semesters needed, otherwise return -1
    return totalCourses === 0 ? semesters : -1;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed step by step:

1. Creating the graph `g` and the indegree array `indeg`: This involves iterating over each relation once. If there are `E` relations, this operation has a complexity of $O(E)$.
2. Building the initial queue `q` with vertices of indegree 0: This involves a single scan through the `indeg` array of `n` elements, which gives a complexity of $O(n)$.
3. The while loop to reduce the indegrees and find the order of courses: Within the loop, each vertex is processed exactly once, as it gets removed from the queue, and each edge is considered exactly once, as it gets decremented when the previous vertex is processed. So, the complexity related to processing all vertices and edges is $O(V + E)$, where `V` is the number of vertices and `E` is the number of edges.

Thus, the overall time complexity is $O(E) + O(n) + O(V + E)$. Since $V = n$, we can simplify this to $O(n + E)$.

Space Complexity

The space complexity of the code:

1. The graph `g` has a space complexity of $O(E)$, as it stores all the edges.
2. The indegree array `indeg` has a space complexity of $O(n)$ as it stores an entry for each vertex.
3. The queue `q` could potentially store all `n` vertices in the worst case, which gives a space complexity of $O(n)$.

As such, the overall space complexity is the sum of the complexities of `g`, `indeg`, and `q`, which gives $O(E + n)$. Note that even though `q` could be as large as $n - 1$ in a dense graph, we do not multiply by `n` in the space complexity as we consider distinct elements within the data structure.