2050. Parallel Courses III

Hard · Graph · Topological Sort · Array · Dynamic Programming · Leetcode Link

Problem Description

In this problem, you are tasked with finding the minimum number of months needed to complete a set of courses with certain prerequisites. There are `n` courses labeled from `1` to `n`. A 2D integer array `relations` is given, where each pair `[prevCourse, nextCourse]` signifies that the `prevCourse` must be completed before the `nextCourse` can be taken. You also have an array `time` representing the number of months required to complete each course, indexed at `0`. There are two key rules you must follow: you can start any course at any time as long as its prerequisites are satisfied, and you can take as many courses concurrently as you desire. The task is to calculate the minimum total time needed to complete all courses, under the constraint that a direct acyclic graph (DAG) represents the course structure.

Intuition

To solve this problem, we need to process courses in a way that respects prerequisite relations and keeps track of the total time needed to complete each course individually, as well as the overall time to finish all courses. This lends itself to a topological sort approach, where we:

1. Determine the prerequisites (in-degree) for each course.
2. Create a graph representing the course dependencies.
3. Use a queue to process courses that have no prerequisites.
4. Keep an array (`f`) to store the finishing time for each course, based on its prerequisites' finishing times.
5. Use a variable (`ans`) to maintain the maximum finishing time across all courses.

The provided solution iterates through the courses, initializing a queue with those that have no prerequisites and setting their finishing time in the array `f` using the `time` array. We then go through the courses in the queue one by one, updating the finishing time for their dependent courses and ensuring we take the maximum time between the time already recorded for that course and the finishing time of the current course plus the time required for the dependent course. We decrement the in-degree of each dependent course and add it to the queue if its in-degree drops to zero.

Using these steps, we guarantee that we process the courses such that prerequisites are always completed before dependent courses and that concurrency is maximized, as there are no restrictions on the number of courses that can be taken simultaneously. The final answer (`ans`) represents the minimum number of months needed to complete all courses, as it accounts for the longest path through the graph, which effectively is the critical path that determines the overall time to completion.

Solution Approach

The solution to this problem efficiently makes use of topological sorting, a common algorithm for processing Directed Acyclic Graphs (DAGs), alongside dynamic programming to calculate minimum times.

Data Structures:

- A graph `g` is represented as a dictionary of lists, mapping each course to its list of next courses.
- An in-degree array `indeg`, which keeps track of how many prerequisites each course has.
- A queue `q`, implemented as a deque, used for the topological sort.
- An array `f`, used to store the finishing time for each course, or the minimum months needed to reach that course.

Algorithm Steps:

1. **Graph and In-Degree Initialization:** For each relation in `relations`, the graph `g` and the in-degree array `indeg` are populated. Every relation `[a, b]` indicates `b` is dependent on `a`, so `a` is added to the list of `b`'s prerequisites in the graph, and `b`'s in-degree is incremented since it has one more course that precedes it.

2. **Identifying Start Courses:** We iterate over all courses. Those without prerequisites (`indeg[i] == 0`) are added to the queue `q`, and their finishing time `f[i]` is set to their respective time from the `time` array.

3. **Topological Sort and Finishing Time Calculation:**
   - While the queue is not empty:
     - Dequeue an element `i`.
     - For each course `j` that `i` is a prerequisite for:
       - Update the finishing time for course `j` (`f[j]`) to be the maximum of the current stored time and the sum of `f[i]` and the time to complete course `j`. This ensures `f[j]` reflects the total time taken to reach course `j` after completing all its prerequisites.
       - Decrement the in-degree of course `j`. If `j`'s in-degree becomes 0, it means all its prerequisites are met, and it's added to the queue for processing.
     - Update the overall answer `ans` to be the maximum of its current value and `f[j]`.
4. **Result:** After processing all courses, `ans` holds the minimum number of months required to complete all courses. This is because `ans` tracks the longest path (in months) through the graph, which corresponds to the time it would take to finish the courses if they were taken sequentially following their dependencies.

Conclusion:

The algorithm successfully computes the minimum time to finish all courses by leveraging topological sort to respect course prerequisites and dynamic programming to calculate the minimum finish times, taking advantage of the ability to take multiple courses concurrently. This solution ensures that all courses are processed in the correct order and that the time tracking reflects the concurrent nature of course enrollment.

Example Walkthrough

Let's illustrate the solution approach using a small example:

Imagine a scenario with 4 courses, where the `time` array is given as `time = [1, 2, 3, 1]`, which indicates that course 1 takes 1 month to complete, course 2 takes 2 months, and so on. The `relations` array is given as `relations = [[1, 3], [2, 3], [3, 4]]`, meaning course 1 and 2 are prerequisites for course 3, and course 3 is a prerequisite for course 4.

1. **Graph and In-Degree Initialization:**
   - Initialize the graph `g` and in-degree array `indeg`.
   - For `relations = [[1, 3], [2, 3], [3, 4]]`, we build the graph `g = {1: [3], 2: [3], 3: [4]}`. The in-degree array `indeg = [0, 0, 2, 1]` will have the count of prerequisites for each course.
2. **Identifying Start Courses:**
   - We see that course 1 and course 2 have zero prerequisites (`indeg[0] == 0` and `indeg[1] == 0`).
   - Initialize the queue `q` with these courses and set `f = [1, 2, 0, 0]`, where `f[0] = 1` and `f[1] = 2`.
3. **Topological Sort and Finishing Time Calculation:**
   - Process the courses in the queue:
     - Dequeue course 1, with finishing time of 1 month. Course 3 is a dependent, so update `f[2]` to `max(f[2], f[0] + time[2]) = max(0, 1 + 3) = 4`.
     - Decrease in-degree of course 3 (`indeg[2]` becomes 1).
     - Since in-degree of course 3 is not 0, it remains in the graph.
     - Next, dequeue course 2, with a finishing time of 2 months. Again, course 3 is a dependent, so update `f[2]` to `max(f[2], f[1] + time[2]) = max(4, 2 + 3) = 5`.
     - Decrease in-degree of course 3 (`indeg[2]` becomes 0), and add course 3 to the queue.
     - Dequeue course 3, with finishing time of 5 months. Course 4 is a dependent, so update `f[3]` to `max(f[3], f[2] + time[3]) = max(0, 5 + 1) = 6`.
     - Decrease in-degree of course 4 (`indeg[3]` becomes 0), and add course 4 to the queue.
   - The queue is now empty, and we can calculate the maximum time from the array `f`, which is `ans = 6` months.
4. **Result:** We conclude that it will take a minimum of 6 months to complete all courses, given the ability to take multiple courses concurrently, and respecting the prerequisite requirements.

This walk-through illustrates the process of topological sorting, dynamic programming, and concurrent course enrollment utilized by the proposed solution to compute the minimum total time required to complete all the courses.

Python Solution

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

class Solution:
    def minimumTime(self, total_tasks: int, prerequisites: List[List[int]], task_durations: List[int]) -> int:
        # Create a graph using a dictionary and initialize in-degrees for all nodes.
        graph = defaultdict(list)
        in_degree = [0] * total_tasks

        # Build the graph and update in-degrees based on prerequisites.
        for start, end in prerequisites:
            graph[start - 1].append(end - 1)  # Adjusting indices to be 0-based.
            in_degree[end - 1] += 1

        # Initialize a queue for processing tasks with no prerequisites.
        queue = deque()
        # Initialize an array to keep track of the finish times of each task.
        finish_times = [0] * total_tasks
        # This will keep track of the global maximum time.
        max_time = 0

        # Enqueue tasks with no prerequisites and set their finish times.
        for i, (in_deg, duration) in enumerate(zip(in_degree, task_durations)):
            if in_deg == 0:
                queue.append(i)
                finish_times[i] = duration  # The finish time is the task's own duration.
                max_time = max(max_time, duration)  # Check if this is the new max time.

        # Process tasks in the queue.
        while queue:
            current_task = queue.popleft()
            # Iterate through the current task's dependent tasks.
            for dependent in graph[current_task]:
                # Calculate the finish time of the dependent task.
                finish_times[dependent] = max(finish_times[dependent], finish_times[current_task] + task_durations[dependent])
                # Update the global maximum time.
                max_time = max(max_time, finish_times[dependent])
                # Decrease the in-degree of the dependent task.
                in_degree[dependent] -= 1
                # If the dependent task has no more prerequisites, add it to the queue.
                if in_degree[dependent] == 0:
                    queue.append(dependent)

        # Return the global maximum time, which is the minimum time to complete all tasks.
        return max_time
```

Java Solution

```java
class Solution {

    public int minimumTime(int totalCourses, int[][] prerequisites, int[] timeToComplete) {
        // Create a graph represented as an adjacency list
        List<Integer>[] graph = new List[totalCourses];
        Arrays.setAll(graph, k -> new ArrayList<>());

        // Create an array to track the number of prerequisites for each course
        int[] incomingEdges = new int[totalCourses];

        // Build the graph and count incoming edges for each node (course)
        for (int[] relation : prerequisites) {
            // Courses are 1-indexed, but our array/graph is 0-indexed
            int prerequisite = relation[0] - 1;
            int course = relation[1] - 1;
            graph[prerequisite].add(course);
            ++incomingEdges[course];
        }

        // Initialize a queue for courses that have no prerequisites
        Deque<Integer> queue = new ArrayDeque<>();
        // Array to store the time to finish each course including prerequisites
        int[] finishTime = new int[totalCourses];
        // Variable to store the maximum time to finish all courses
        int maxTime = 0;

        // Enqueue courses without prerequisites and set their finish times
        for (int i = 0; i < totalCourses; ++i) {
            if (incomingEdges[i] == 0) {
                queue.offer(i);
                finishTime[i] = timeToComplete[i];
                maxTime = Math.max(maxTime, finishTime[i]);
            }
        }

        // Process the courses in topological order
        while (!queue.isEmpty()) {
            int currentCourse = queue.pollFirst();
            // For each course that follows the current one
            for (int nextCourse : graph[currentCourse]) {
                // Update the finish time to the maximum of the current known time and the time required after finishing the current
                finishTime[nextCourse] = Math.max(finishTime[nextCourse], finishTime[currentCourse] + timeToComplete[nextCourse]);
                maxTime = Math.max(maxTime, finishTime[nextCourse]);
                // If the course has no remaining prerequisites, add it to the queue
                if (--incomingEdges[nextCourse] == 0) {
                    queue.offer(nextCourse);
                }
            }
        }

        // Return the maximum time to finish all courses
        return maxTime;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minimumTime(int n, vector<vector<int>>& relations, vector<vector<int>>& time) {
        // Create a graph data structure to represent courses and their prerequisites
        vector<vector<int>> graph(n);
        // A vector to keep track of the number of prerequisites (in-degree) for each course
        vector<int> inDegree(n, 0);

        // Construct the graph and fill in-degree data
        for (const auto& relation : relations) {
            // Courses are numbered starting 1, so subtract 1 to get a 0-based index
            int prevCourse = relation[0] - 1;
            int nextCourse = relation[1] - 1;
            // Create an edge from 'prevCourse' to 'nextCourse'
            graph[prevCourse].push_back(nextCourse);
            // Increment in-degree for the 'nextCourse'
            ++inDegree[nextCourse];
        }

        // Queue for BFS traversal
        queue<int> courseQueue;
        // Vector to keep track of the total time to complete each course
        vector<int> finishTime(n, 0);
        // Initialize the final answer to time completion
        int totalTime = 0;

        // Enqueue courses with no prerequisites
        for (int i = 0; i < n; ++i) {
            if (inDegree[i] == 0) {
                courseQueue.push(i);
                finishTime[i] = time[i];
                // Update total time with the time taken to complete this course
                totalTime = max(totalTime, time[i]);
            }
        }

        // Process courses in the order they can be taken
        while (!courseQueue.empty()) {
            int currentCourse = courseQueue.front();
            courseQueue.pop();

            // Check all the next courses that can be taken after the current course
            for (int nextCourse : graph[currentCourse]) {
                // Decrease the in-degree of the next course
                --inDegree[nextCourse];

                // If all prerequisites of the next course have been taken, add to queue
                if (inDegree[nextCourse] == 0) {
                    courseQueue.push(nextCourse);
                }

                // Update the finish time of the next course
                // It will be the maximum of its current finish time and
                // the finish time of its prerequisite (current course) plus its own duration
                finishTime[nextCourse] = max(finishTime[nextCourse], finishTime[currentCourse] + time[nextCourse]);
                // Update the total time completion with the updated finish time of the next course
                totalTime = max(totalTime, finishTime[nextCourse]);
            }
        }

        // Return the maximum time required to finish all courses
        return totalTime;
    }
};
```

Typescript Solution

```typescript
function minimumTime(numCourses: number, prerequisites: number[][], courseDurations: number[]): number {
    // Graph represented as an adjacency list
    const graph: number[][] = Array(numCourses)
        .fill(0)
        .map(() => []);

    // Array to keep track of incoming edges (in-degree) for each node
    const inDegrees: number[] = Array(numCourses).fill(0);

    // Build graph and populate inDegrees array
    for(const [source, target] of prerequisites) {
        graph[source - 1].push(target - 1);
        ++inDegrees[target - 1];
    }

    // Queue for processing courses that have no prerequisites or whose prerequisites have been completed
    const queue: number[] = [];

    // Array to keep the completion time for each course
    const finishTimes: number[] = Array(numCourses).fill(0);

    // Variable to keep track of the maximum time required to finish all courses
    let maxTime: number = 0;

    // Initialize the queue with courses that have no prerequisites and calculate their finish times
    for(let i = 0; i < numCourses; ++i) {
        if(inDegrees[i] === 0) {
            queue.push(i);
            finishTimes[i] = courseDurations[i];
            maxTime = Math.max(maxTime, finishTimes[i]);
        }
    }

    // Process courses in topological order
    while(queue.length > 0) {
        const currentCourse = queue.shift()!;
        for(const nextCourse of graph[currentCourse]) {
            finishTimes[nextCourse] = Math.max(finishTimes[nextCourse], finishTimes[currentCourse] + courseDurations[nextCourse]);
            maxTime = Math.max(maxTime, finishTimes[nextCourse]);
            // When a prerequisite course is completed, decrease in-degree of the next course
            if (--inDegrees[nextCourse] === 0) {
                queue.push(nextCourse);
            }
        }
    }

    // Return the maximum time to complete all courses
    return maxTime;
}
```

Time and Space Complexity

The time complexity of the given code is `O(V + E)`, where `V` is the number of courses (nodes) and `E` is the number of dependencies (edges). This complexity arises because each course is enqueued and dequeued exactly once, which is `O(V)`. Every dependency is looked at exactly once when iterating through the adjacency list, which gives `O(E)`. Therefore, the total is `O(V + E)`.

The space complexity is also `O(V + E)` because we need to store the graph `g`, which contains `E` edges, the `indeg` array of size `V`, and the `f` array of size `V`. Considering the queue's space, in the worst case, it could hold all vertices, which adds up to `O(V)`. However, since this does not change the overall space complexity, it remains `O(V + E)`.