2924. Find Champion II

Medium Graph

Problem Description

In this problem, we are dealing with a tournament involving `n` teams, where each team is represented as a node in a Directed Acyclic Graph (DAG). The 2D integer array `edges` defines the relationships between the teams in terms of their strength; if there is a directed edge from team `a` to team `b` (represented as `[a, b]`), it implies that team `a` is stronger than team `b`. The goal is to find which team is the champion of the tournament. A team is considered the champion if there is no other team that is stronger than it, meaning no edges are directed towards it. We want to return the index of that champion team. However, if there's no unique champion (either because there are multiple strongest teams or the graph structure doesn't allow for a single strongest team), we should return `-1`.

Intuition

The problem can be solved by analyzing the in-degree of each node (team) in the graph. The in-degree of a node is the number of edges directed towards it. If a team is the strongest, it means no other team can beat it, which implies that it should have an in-degree of `0`—no edges pointing to it. Therefore:

1. We first initialize an array `indeg` to keep track of in-degrees for all nodes, with the size equal to the number of teams `n`.
2. We iterate over the `edges` array. For each directed edge `[u, v]`, we increment the in-degree count for team `v` since the edge indicates that team `u` (index `u`) is stronger than team `v` (index `v`).
3. After populating the `indeg` array, we check for the following scenario:
   - If there is exactly one team with an in-degree of `0`, that team is the champion as it is not outmatched by any other team.
   - If there is more than one team or no teams with an in-degree of `0`, it means there is no unique champion, and we return `-1`.

The solution approach is straightforward and efficient because it relies on counting and analyzing in-degrees, which can be done in linear time with respect to the number of edges in the graph.

Solution Approach

To implement our solution, we utilize a simple yet effective approach that revolves around the concept of in-degrees for nodes in a Directed Acyclic Graph (DAG). Our algorithm follows these steps:

1. We first create an array `indeg` which has the same length as the number of teams `n`. This array is used to keep track of the in-degrees for each node (team), and it is initialized with zeros since initially, we assume that no edges are directed towards any of the nodes.
2. We then iterate through the provided list of edges. For each edge `[u, v]` in the `edges` array, we know that the edge represents a match where team `u` is stronger than team `v`. Therefore, we increment the in-degree count of node `v` by 1, as this indicates another team is stronger than team `v`.
3. After processing all edges, we check the `indeg` array for the champion team. In a DAG, a node with an in-degree of `0` means there are no incoming edges, therefore no team is stronger than it. The champion team is then the unique node with an in-degree of `0`.
4. The final step is to return the correct index. If there is exactly one team with an in-degree of `0`, we return the index of this team, as it represents the champion. If there are multiple teams with an in-degree of `0` or if every team has an in-degree greater than `0`, we return `-1` to indicate there is no unique champion.

The solution uses a counting array to keep track of in-degrees, which is a common pattern when working with graph data structures that can easily represent dependencies or hierarchies.

The provided reference solution implements this strategy concisely in Python:

```
class Solution:
    def findChampion(self, n: int, edges: List[List[int]]) -> int:
        indeg = [0] * n
        for _, v in edges:
            indeg[v] += 1
        return -1 if indeg.count(0) != 1 else indeg.index(0)
```

In this implementation:

- `indeg = [0] * n` initializes the in-degree count array.
- The `for _, v in edges:` loop goes through the edges list and increments the in-degree count for the end node of every edge. The underscore `_` is used to ignore the first element `u` of each edge since we only care about the in-degrees in this context.
- The final line uses a conditional expression to check the `indeg` array: `indeg.count(0) != 1` checks if there's precisely one team with an in-degree of `0`, and `indeg.index(0)` finds the index of that team. If the count is not 1, we return `-1`.

The algorithm's time complexity is O(m + n), where `m` is the number of edges, because we need to process each edge once, and `n` is the number of teams, as we need to check each team's in-degree and also return the index of the champion.

Example Walkthrough

Let's consider a simple example where we have `n = 4` teams and the following set of edges that represent the outcome of the matches:

```
edges = [[0, 1], [2, 3], [2, 1]]
```

Using these edges, let's walk through the solution:

Step 1: We create an array `indeg` with size `n` and initialize it with zeros.

```
indeg = [0, 0, 0, 0]
```

Each index corresponds to a team (0 through 3), and the value at that index will represent the team's in-degree.

Step 2: We iterate through the edges:

- For the first edge `[0, 1]`, we increment the in-degree of team 1.

```
indeg = [0, 1, 0, 0]
```

- For the second edge `[2, 3]`, we increment the in-degree of team 3.

```
indeg = [0, 1, 0, 1]
```

- For the third edge `[2, 1]`, we increment the in-degree of team 1 again.

```
indeg = [0, 2, 0, 1]
```

After processing all edges, the `indeg` array is `[0, 2, 0, 1]`.

Step 3: We check for a unique team with an in-degree of `0`. In our example, we have two teams (team 0 and team 2) with an in-degree of `0`.

Step 4: We analyze the result. Since there are two teams with no incoming edges, there is no unique champion team. Therefore, according to the rules, we should return `-1`.

Using the provided Python implementation:

```
class Solution:
    def findChampion(self, n: int, edges: List[List[int]]) -> int:
        indeg = [0] * n
        for _, v in edges:
            indeg[v] += 1
        return -1 if indeg.count(0) != 1 else indeg.index(0)
```

If we call `findChampion(4, [[0, 1], [2, 3], [2, 1]])`, it will initialize the `indeg` array, increment the in-degrees accordingly, and then return `-1` as there are multiple teams with an in-degree of `0`.

Solution Implementation

Python

```
from typing import List

class Solution:
    def findChampion(self, n: int, edges: List[List[int]]) -> int:
        # Create a list to keep track of in-degrees for all the nodes
        in_degree = [0] * n

        # Iterate over each edge in the graph
        for _, destination in edges:
            # Increment the in-degree for the destination node of each edge
            in_degree[destination] += 1

        # Find if there is exactly one node with an in-degree of 0
        # This node is the potential champion as it has no incoming edges
        if in_degree.count(0) == 1:
            # Return the index of the node with an in-degree of 0
            return in_degree.index(0)
        else:
            # If there is not exactly one node with in-degree 0, return -1
            return -1
```

Java

```
class Solution {
    // Method to find the "champion" node.
    // The champion is defined as the unique node with no incoming edges.
    public int findChampion(int n, int[][] edges) {
        // Array to store the incoming degree count for each node
        int[] incomingDegreeCounts = new int[n];

        // Loop through all edges to calculate the incoming degree counts
        for (int[] edge : edges) {
            // Increment the count of incoming edges for the destination node
            incomingDegreeCounts[edge[1]]++;
        }

        // Variable to store the champion node index, initialized as -1 (not found)
        int champion = -1;
        // Counter to keep track of how many nodes have zero incoming edges
        int zeroIncomingCount = 0;

        // Loop through all nodes to find the champion node
        for (int i = 0; i < n; ++i) {
            // Check if the current node has zero incoming edges
            if (incomingDegreeCounts[i] == 0) {
                // Increment the counter for nodes with zero incoming edges
                zeroIncomingCount++;
                // Set the current node as the potential champion
                champion = i;
            }
        }

        // Check if there is exactly one node with zero incoming edges
        // If so, return the index of the champion node; otherwise, return -1
        return zeroIncomingCount == 1 ? champion : -1;
    }
}
```

C++

```
class Solution {
public:
    int findChampion(int n, vector<vector<int>>& edges) {
        // Create an array to track the in-degree (number of incoming edges) of each node
        int inDegree[n];
        memset(inDegree, 0, sizeof(inDegree)); // Initialize all in-degree values to 0

        // Iterate through the edges and increment the in-degree of the destination nodes
        for (auto& edge : edges) {
            ++inDegree[edge[1]];
        }

        int champion = -1; // Variable to store the candidate node for champion
        int zeroInDegreeCount = 0; // Counter to keep track of the number of nodes with 0 in-degrees

        // Iterate through all nodes to find the node with 0 in-degree, if any
        for (int i = 0; i < n; ++i) {
            if (inDegree[i] == 0) { // A node with 0 in-degree can be a candidate for champion
                ++zeroInDegreeCount;
                champion = i; // Update candidate node
            }
        }

        // If there is exactly one node with 0 in-degree, it is the champion
        // Otherwise, there is no champion (return -1)
        return zeroInDegreeCount == 1 ? champion : -1;
    }
};
```

TypeScript

```
// Function to find the "champion" node in a directed graph.
// The champion node is one that has no incoming edges.
// Only one such node must exist for it to be the champion.
// If more than one such node exists or if none exists, return -1.
function findChampion(n: number, edges: number[][]): number {
    // Initialize an array to track the number of incoming edges (indegree) for each node.
    const indegrees: number[] = Array(n).fill(0);

    // Populate the indegrees array by incrementing the indegree for every destination node
    // in the edges.
    for (const [, destination] of edges) {
        ++indegrees[destination];
    }

    // Initialize variables to store the answer (the champion's index) and a count of nodes with 0 indegree.
    let answer: number = -1;
    let zeroIndegreeCount: number = 0;

    // Iterate over each node and find nodes with an indegree of 0 (no incoming edges).
    for (let i = 0; i < n; ++i) {
        if (indegrees[i] === 0) {
            // If a node with no incoming edges is found, increment the count and update the answer.
            ++zeroIndegreeCount;
            answer = i;
        }
    }

    // If there is exactly one node with no incoming edges, return that node's index.
    // Otherwise, return -1 to indicate that no champion was found.
    return zeroIndegreeCount === 1 ? answer : -1;
}
```

```
from typing import List

class Solution:
    def findChampion(self, n: int, edges: List[List[int]]) -> int:
        # Create a list to keep track of in-degrees for all the nodes
        in_degree = [0] * n

        # Iterate over each edge in the graph
        for _, destination in edges:
            # Increment the in-degree for the destination node of each edge
            in_degree[destination] += 1

        # Find if there is exactly one node with an in-degree of 0
        # This node is the potential champion as it has no incoming edges
        if in_degree.count(0) == 1:
            # Return the index of the node with an in-degree of 0
            return in_degree.index(0)
        else:
            # If there is not exactly one node with in-degree 0, return -1
            return -1
```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the number of nodes. This time complexity comes from having to iterate through all the edges once to calculate the in-degrees and then through all the nodes to find the one with an in-degree of 0.

The space complexity is also `O(n)`, originating from the storage requirements of the `indeg` list that has to store an in-degree value for each of the `n` nodes.