1557. Minimum Number of Vertices to Reach All Nodes

Medium Graph Leetcode Link

Problem Description

The given problem presents a Directed Acyclic Graph (DAG) which means a graph that has directed edges and no cycles. It includes 'n' vertices which are labeled from `0` to `n-1`. We're also provided an array of `edges` where each edge is represented by a pair `[from_i, to_i]`, indicating a directed edge that goes from vertex `from_i` to vertex `to_i`.

The task is to find the smallest set of vertices with the property that starting from those vertices, one can reach all other vertices in the graph. The important aspect of this problem is to understand that these vertices should be the starting points for traversing the entire graph. It is also given that there is a unique solution for this problem, which means there's only one smallest set of such vertices and the vertices can be returned in any sequence.

Intuition

To solve this problem, we take advantage of a key characteristic of DAGs — a node that has no incoming edges can be a starting point to reach all other nodes because there are no other nodes that need to be visited before it. Conversely, nodes with incoming edges have at least one other node that must be processed first.

The solution approach is as follows:

1. We initialize a counter `cnt` to keep track of the number of incoming edges each vertex has.
2. To fill this counter, we iterate over all directed edges. For each edge, we increase the count for the target vertex `to_i` because this edge represents an incoming edge to `to_i`.
3. Once we have the count of incoming edges for each vertex, the vertices with a count of zero are exactly the vertices we're looking for. They have no incoming edges and can serve as starting points.
4. We iterate through all vertices from `0` to `n-1`, and those with a `cnt` value of zero are added to our results list as they represent the smallest set of vertices that can be used to reach all other vertices in the graph.

By following this process, we efficiently identify the vertices that are not dependent on any other vertices to be visited first, which fulfills the requirement of reaching every vertex in the graph starting from the smallest set of initial vertices.

Solution Approach

The solution is implemented in Python, and it uses the `Counter` class from the `collections` module to keep track of the number of incoming edges for each vertex in the graph.

Here's a step-by-step breakdown of the algorithm implemented in the solution code:

1. A `Counter` object named `cnt` is created to count the incoming edges of each vertex. This is constructed by iterating over the `edges` list and counting each target vertex `t` (the second element of each edge `[_, t]`).

2. A for-loop comprehensions construct:

```python
[i for i in range(n) if cnt[i] == 0]
```

   This is used to iterate over all vertex indices from `0` to `n-1`.

3. For each index `i`, the loop checks if `cnt[i] == 0`. This condition is true for the vertices that do not have any incoming edges (meaning they are not the target of any edge in the `edges` list).

4. The indices `i` that satisfy the condition are collected into a list. These form the smallest set of vertices from which all nodes in the graph are reachable.

5. The final list is returned as the output of the `findSmallestSetOfVertices` function.

The algorithm essentially uses a counting method to identify *source nodes* in the graph. A *source node* is a node with no incoming edges, which means that it can be a starting point, and you can reach every other node in the DAG via a path from these sources.

This algorithm runs in linear time relative to the number of edges, as it only makes a single pass through the `edges` list to build the counter and then another pass through the list of vertex indices to collect the result. Therefore, the time complexity is O(E + V), where E is the number of edges and V is the number of vertices.

Example Walkthrough

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

Suppose we have a graph with `n = 4` vertices and the following directed edges: `edges = [[0,1], [0,2], [1,3], [2,3]]`. The edges indicate that there are directed paths from vertex 0 to vertex 1, from vertex 0 to vertex 2, from vertex 1 to vertex 3, and from vertex 2 to vertex 3.

Let's walk through the approach:

1. We set up a counter `cnt` to track the incoming edges to each vertex. In our graph representation, the incoming edges are described by the second element in each edge pair. So initially, the counter `cnt` for all vertices is set to zero: `cnt = {0: 0, 1: 0, 2: 0, 3: 0}`.

2. Next, we iterate over the list of edges to update these counts. After iterating, we get:

   - For edge `[0,1]`, increase `cnt[1]` by 1;
   - For edge `[0,2]`, increase `cnt[2]` by 1;
   - For edge `[1,3]`, increase `cnt[3]` by 1;
   - For edge `[2,3]`, increase `cnt[3]` by 1 again.

   After processing all edges, the updated counts of incoming edges for the vertices are: `cnt = {0: 0, 1: 1, 2: 1, 3: 2}`.

3. With these counts, we now identify vertices with zero incoming edges, as they are potential starting points to reach any other vertex. From `cnt`, we see that only vertex `0` has a count of zero.

4. Using a list comprehension, we extract vertices with zero counts into our result list:

```python
result = [i for i in range(n) if cnt[i] == 0] # result will be [0]
```

5. The final `result` list, which contains the smallest set of vertices from which all other vertices are reachable, is `[0]`.

This is the end of the example walkthrough. By starting at vertex `0`, we can reach all other vertices in the graph, fulfilling the task. The solution approach works efficiently, regardless of the graph's size, by focusing on finding vertices with no dependencies (no incoming edges).

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def findSmallestSetOfVertices(self, n: int, edges: List[List[int]]) -> List[int]:
        # Create a counter to count how many times each node is the target of an edge
        target_counter = Counter(target for _, target in edges)

        # Systematically check each vertex from 0 to n-1
        # If the count of incoming edges is 0, then it is not a target of any edge
        # and must be included in the smallest set of vertices that reaches all nodes
        return [vertex for vertex in range(n) if target_counter[vertex] == 0]

```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<Integer> findSmallestSetOfVertices(int n, List<List<Integer>> edges) {
        // Create an array to count the in-degree of each vertex
        int[] inDegreeCount = new int[n];

        // Iterate over the edges to count the in-degree for each vertex
        for (List<Integer> edge : edges) {
            // Increment the in-degree count of the destination vertex
            inDegreeCount[edge.get(1)]++;
        }

        // Prepare a list to hold the answer vertices
        List<Integer> answerVertices = new ArrayList<>();

        // Go through the vertices
        for (int i = 0; i < n; i++) {
            // If a vertex has an in-degree of 0, it's not reachable from any other vertex
            if (inDegreeCount[i] == 0) {
                // Add the vertex to answerVertices as it is a candidate for the
                // smallest set of vertices from which all nodes are reachable
                answerVertices.add(i);
            }
        }

        // Return the list of vertices from which all other nodes are reachable
        return answerVertices;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This method finds the smallest set of vertices from which all nodes are reachable
    // in the given directed graph represented by 'n' nodes and 'edges'.
    vector<int> findSmallestSetOfVertices(int n, vector<vector<int>>& edges) {
        // Initialize a counter vector to track the incoming edges for each vertex.
        vector<int> incomingEdgesCounter(n, 0);

        // Iterate over all edges to increment the counter of the destination nodes.
        for (const auto& edge : edges) {
            ++incomingEdgesCounter[edge[1]]; // Increment the counter of incoming edges for the destination node.
        }

        // Prepare an vector to store the answer.
        vector<int> answer;

        // Iterate over all nodes to check which nodes have zero incoming edges.
        for (int i = 0; i < n; ++i) {
            // If the current node has zero incoming edges,
            // it means it cannot be reached by other nodes,
            // so we add it to the answer set.
            if (incomingEdgesCounter[i] == 0) {
                answer.push_back(i);
            }
        }

        // Return the vector containing all nodes with zero incoming edges.
        return answer;
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the smallest set of vertices from which all nodes in the graph are reachable.
 * @param {number} numVertices - The total number of vertices in the graph.
 * @param {number[][]} edges - The edges of the graph represented as an array of tuples [from, to].
 * @returns {number[]} The array of vertex indices that form a smallest set of vertices.
 */
function findSmallestSetOfVertices(numVertices: number, edges: number[][]): number[] {
    // Initialize an array to count the in-degree of each vertex.
    const inDegreeCount: number[] = new Array(numVertices).fill(0);

    // Iterate over edges to calculate the in-degree for each vertex.
    for (const [_, to] of edges) {
        inDegreeCount[to]++;
    }

    // Initialize an array to store the answer.
    const answer: number[] = [];

    // Iterate over the vertices.
    for (let i = 0; i < numVertices; ++i) {
        // If the in-degree of a vertex is 0, it means that it is not reachable from any other vertex.
        // Therefore, it must be included in the set.
        if (inDegreeCount[i] === 0) {
            answer.push(i);
        }
    }

    // Return the list of vertices that must be included in the smallest set.
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by two main operations. The first operation is the creation of the `Counter`, which involves iterating over all the edges. If `m` represents the number of edges, then this operation is `O(m)`.

The second operation is the list comprehension, which checks each vertex to see if it has a count of zero in the `Counter`. Since there are `n` vertices, this operation is `O(n)`.

Since these two operations happen sequentially and not nested, the overall time complexity is `O(m + n)`.

Space Complexity

The space complexity is primarily impacted by the `Counter` that stores occurrence counts for the target vertices of each edge. In the worst case, all `m` edges might be pointing to different target vertices, so the `Counter` would store `m` key-value pairs.

Therefore, the space complexity is `O(m)` for the `Counter`. However, the output list's size is at most `n` in the case when no vertices are targets. This `n` sized list is separate from the `Counter` storage.

Thus, when considering both the `Counter` and the final output list, the space complexity is `O(m + n)`.