2876. Count Visited Nodes in a Directed Graph

Hard Graph Memoization Dynamic Programming Leetcode Link

Problem Description

In this problem, we are given a directed graph which is represented by a 0-indexed array `edges`. Each element `edges[i]` indicates that there is a directed edge from node `i` to node `edges[i]`. The graph consists of `n` nodes labeled from `0` to `n - 1`, and `n` directed edges.

The task is to determine, for each starting node `i`, how many distinct nodes can be visited if we follow the directed edges starting from node `i` until we return to any node visited earlier in the *same* process. This essentially means we're calculating how many unique nodes we encounter before we enter a cycle or revisit any node during traversal which implies the start of a cycle.

Intuition

To solve this problem, we need to keep track of two key pieces of information:

- The number of nodes visited in the current traversal, starting from each node `i`.
- Whether or not we've entered a cycle, and if so, the length of that cycle.

To do this, we can start at each unvisited node and follow edges to traverse the graph, using an array `vis` to record the order in which we visited nodes. During the traversal, we do one of two things:

- If we find a node that has already been visited in this traversal (`vis[j]` is nonzero), we then know that we have entered a cycle. We now know the number of nodes outside the cycle that have been visited and also the length of the cycle. For each unvisited node, its answer is then the number of unique nodes visited — which is the sum of the length of the cycle and the path length to the cycle for nodes outside the cycle, or simply the cycle length for nodes inside the cycle.

- If we find a node for which we've already computed an answer, then, for each visited node, its answer is the distance from the current node to this node plus the answer of this node we've already computed.

The algorithm involves iterating over each of the nodes of the graph and performing a depth-first traversal. During the traversal, we update the `vis` array to reflect the visitation order and calculate the necessary counts for the cycle and the path leading to the cycle. Once a traversal from a starting node is completed or we encounter a previously computed answer, we backtrack while updating the `ans` array with the number of nodes that can be visited for each start node.

This method allows us to efficiently compute the number of nodes that can be visited from each starting node by leveraging the information gained during each traversal and avoiding redundant computations.

Solution Approach

The solution implements a form of depth-first traversal. Here's a breakdown of the approach based on the provided Reference Solution Approach and Python solution:

1. **Initialization:** Create two arrays, `ans` and `vis`. `ans` will store the final answer for each node (i.e., the number of different nodes visited), and `vis` will store the visitation order for each node. Both are initially filled with zeros.

2. **Outer Loop:** Iterate through all the nodes with a `for` loop, using each node as a starting point for the process.

3. **Traversal Initiation:** For each starting node, if `ans[i]` is zero, meaning that we have not computed the answer for this node, we start traversing from this node `i`.

4. **Inner Loop - Traversing Nodes:** In a `while` loop, continue to follow the directed edges until we encounter a node that we have already visited in this traversal (which implies a cycle) or a node that we have already computed the answer for in a different traversal. For each node `j` we visit:
   - Increment the count (`cnt`) which keeps track of how many nodes we have visited in this traversal.
   - Update `vis[j]` with the visitation order `cnt`.
   - Move to the next node using `j = edges[j]`.
5. **Cycle and Path Detection:** Once we've either hit an already visited node within this traversal or an already computed node, calculate the cycle length if necessary and the total number of nodes visited. If a cycle is detected, the length is determined by the difference between the current `cnt` value and the visitation order of the node that's part of the cycle (`vis[j]`).

6. **Backtracking and Answer Update:** Once we've hit a node for which an answer was already computed or the start of a cycle has been determined, loop back through the nodes we just visited to update their answers (`ans[j]`). The nodes inside the cycle all have the same answer (the cycle length).

7. **Population of Answers:** Loop through the previously visited nodes and update their answers based on whether they are inside or outside the cycle.

The implementation uses the `max(total, cycle)` to ensure the answer for each node is either the total number of nodes visited (for those outside the cycle) or the length of the cycle (for those inside the cycle).

Data Structures Used:

- One-dimensional lists (`ans` to store answers for each node, `vis` to store visitation order).

Algorithm Patterns Used:

- Depth-First Search (DFS) traversal to explore the graph.
- Cycle detection and cycle length calculation within a directed graph.
- Backtracking to propagate the calculated answers to all starting points included in a traversal.

This approach minimizes redundant computations and efficiently calculates the answers for all the nodes in a directed graph that potentially contains cycles.

Example Walkthrough

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

Suppose we have the following directed graph represented by `edges = [1, 2, 0, 4, 5, 3]`.

This means we have six nodes, and the directed edges are as follows:

- Edge from node `0` to node `1`
- Edge from node `1` to node `2`
- Edge from node `2` to node `0` (forming a cycle with nodes 0, 1, 2)
- Edge from node `3` to node `4`
- Edge from node `4` to node `5`
- Edge from node `5` to node `3` (forming a cycle with nodes 3, 4, 5)

We can visualize the graph like this:

```
0 → 1 → 2
↑       ↓
└───────┘

3 → 4 → 5
↑         ↓
└─────────┘
```

Now, let's walk through the solution steps:

1. **Initialization:** Create two arrays `ans` and `vis` with length 6 (the number of nodes), initializing all values to zero.

2. **Outer Loop:** We start with node `0` and since `ans[0]` is zero, we begin traversal.

3. **Traversal Initiation:** Set `cnt = 1` since we are starting traversal from node `0`. Then we set `vis[0] = 1` to indicate node `0` is visited first.

4. **Inner Loop - Traversing Nodes:** We follow the directed edge to node `1`. Increase `cnt` to `2` and mark `vis[1] = 2`. Next, visit node `2`, increment `cnt` to `3`, and mark `vis[2] = 3`.

5. **Cycle Detection:** When we attempt to move from node `2` to node `0`, we notice that `vis[0]` is already marked (indicating a cycle since we've returned to an already visited node within this traversal). The cycle includes nodes 0, 1, and 2.

6. **Backtracking and Answer Update:** The length of the cycle is `cnt - vis[0] = 3 - 1 = 2`. We then backtrack, updating `ans[0]`, `ans[1]`, and `ans[2]` to `2` as they are part of the cycle.

7. **Population of Answers for Nodes Outside the Cycle:** Since there are no nodes outside the cycle for this starting node, we don't update any additional answers.

Next, we would proceed with node `3` as the starting node and repeat the steps. Eventually, we would identify the cycle among nodes 3, 4, and 5, and after backtracking and updating, we'd have `ans[3]`, `ans[4]`, and `ans[5]` all set to `3`.

After processing all nodes, the final `ans` array is `[2, 2, 2, 3, 3, 3]`, indicating the number of nodes we can visit from each starting node before encountering a cycle or revisiting a node.

During the entire process, we only traverse the graph starting from each node one time, ensuring an efficient solution that avoids redundant calculations. The `vis` array helps to quickly determine the presence of a cycle and the nodes involved in the cycle for accurate answer updates.

Python Solution

```python
from typing import List

class Solution:
    def countVisitedNodes(self, edges: List[int]) -> List[int]:
        num_nodes = len(edges)  # Determine the total number of nodes
        visit_counts = [0] * num_nodes  # Initialize list to store visit counts for each node
        visited = [0] * num_nodes  # Keep track of visited nodes with a timestamp

        # Traverse all nodes to calculate visit counts
        for node in range(num_nodes):
            # Check if we haven't already calculated the visit count for this node
            if not visit_counts[node]:
                count = 0  # Initialize count for counting the nodes visited in this traversal
                current_node = node  # Start traversing from the current node

                # Traverse nodes until we reach a node we've already visited
                while not visited[current_node]:
                    count += 1  # Increase visit count
                    visited[current_node] = count  # Mark current node as visited with a timestamp
                    current_node = edges[current_node]  # Move to the next node in the sequence

                # Analyze the traversal to distinguish between cycle and path lengths
                cycle_length, total_length = 0, count + visit_counts[current_node]
                if not visit_counts[current_node]:
                    # If the target node hasn't been calculated, we found a cycle
                    cycle_length = count - visited[current_node] + 1
                    total_length = count

                # Now backfill the visit_counts array with appropriate visit counts
                current_node = node  # Start again from the initial node
                while not visit_counts[current_node]:
                    # The visit count at each node is the maximum of total length and cycle length
                    visit_counts[current_node] = max(total_length, cycle_length)
                    total_length -= 1  # Reduce total_length as we move back
                    current_node = edges[current_node]  # Move to the next node

        return visit_counts  # Return the computed visit counts for each node

```

Java Solution

```java
class Solution {
    public int[] countVisitedNodes(List<Integer> edges) {
        int n = edges.size(); // The size of the graph (number of nodes)
        int[] visitedCount = new int[n]; // Array to store the count of visits for each node
        int[] visitedState = new int[n]; // Array to track the state of each node during traversal

        // Loop through each node
        for (int currentNode = 0; currentNode < n; ++currentNode) {
            // Only process nodes that have not been counted yet
            if (visitedCount[currentNode] == 0) {
                int count = 0; // Initialize visit count
                int nextNode = currentNode; // Start traversal from the current node

                // Traverse the graph until a previously visited node is encountered
                while (visitedState[nextNode] == 0) {
                    visitedState[nextNode] = ++count; // Increment visit count
                    nextNode = edges.get(nextNode); // Move to the next node in the edge list
                }

                // Variables to determine the length of the cycle and total visits
                int cycleLength = 0;
                int totalVisits = count + visitedCount[nextNode];

                // If the node encountered is the start of the cycle
                if (visitedCount[nextNode] == 0) {
                    cycleLength = count - visitedState[nextNode] + 1; // Calculate cycle length
                }

                nextNode = currentNode; // Reset traversal starting node
                // Assign the maximum of total visits or cycle length to each node in the path
                while (visitedCount[nextNode] == 0) {
                    visitedCount[nextNode] = Math.max(totalVisits--, cycleLength);
                    nextNode = edges.get(nextNode); // Move to the next node in the edge list
                }
            }
        }

        return visitedCount;  // Return the count array for all nodes
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to count the number of visited nodes in a directed graph.
    // edges: An array where the i-th element is the node visited after node i.
    std::vector<int> countVisitedNodes(std::vector<int>& edges) {
        int numNodes = edges.size();
        std::vector<int> visitCount(numNodes);
        std::vector<int> visited(numNodes);

        // Iterate through all nodes in the graph.
        for (int i = 0; i < numNodes; ++i) {
            // If the current node already has a determined visit count, skip it.
            if (!visitCount[i]) {
                int currentCount = 0, currentNode = i;
                // Traverse the graph until a visited node or end of path is found.
                while (visited[currentNode] == 0) {
                    visited[currentNode] = ++currentCount;
                    currentNode = edges[currentNode];
                }

                int cycleSize = 0, totalVisits = currentCount + visitCount[currentNode];
                // If visitCount at the current node is zero, it means we've found a cycle.
                if (visitCount[currentNode] == 0) {
                    cycleSize = currentCount - visited[currentNode] + 1;
                }

                // Backtrack and assign maximum between total visits and cycle size.
                currentNode = i;
                while (visitCount[currentNode] == 0) {
                    visitCount[currentNode] = std::max(totalVisits--, cycleSize);
                    currentNode = edges[currentNode];
                }
            }
        }

        // Return the visit count for each node.
        return visitCount;
    }
};

```

Typescript Solution

```typescript
function countVisitedNodes(edges: number[]): number[] {
    // Get the total number of nodes (length of the edges array)
    const nodeCount = edges.length;
    // Initialize the answer array with zeros
    const answer: number[] = new Array(nodeCount).fill(0);
    // Initialize the visited array with zeros to track if a node has been visited
    const visited: number[] = new Array(nodeCount).fill(0);

    // Iterate over each node
    for (let currentNode = 0; currentNode < nodeCount; ++currentNode) {
        // Only process nodes that have not determined the visited count yet
        if (answer[currentNode] === 0) {
            let visitCount = 0; // Counter for number of nodes visited in the current path
            let nodeIndex = currentNode; // Current node index being visited

            // Traverse the nodes starting from the current node until a visited one is encountered
            while (visited[nodeIndex] === 0) {
                visitCount++;
                visited[nodeIndex] = visitCount; // Mark the node as visited with the count
                nodeIndex = edges[nodeIndex]; // Move to the next connected node
            }

            // Calculate the total visits including the current path and potential prior answers
            let cycleLength = 0;
            let totalVisits = visitCount + answer[nodeIndex];

            // If the next node doesn't have an answer yet, it means we found a cycle
            if (answer[nodeIndex] === 0) {
                cycleLength = visitCount - visited[nodeIndex] + 1; // Calculate the cycle length
            }

            // Reset nodeIndex to the start of the current path
            nodeIndex = currentNode;

            // Update the answer array with the maximum visits for each node in the current path
            while (answer[nodeIndex] === 0) {
                answer[nodeIndex] = Math.max(totalVisits, cycleLength);
                totalVisits--;
                nodeIndex = edges[nodeIndex]; // Move to the next node in the path
            }
        }
    }

    // Return the filled answer array with the visit count for each node
    return answer;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`. This is because each node is visited exactly once in a single pass to count the number of nodes visited before reaching a previously visited node (or itself, forming a cycle). The outer loop runs for each node (`n` iterations), and the inner while loops process each node only once, without revisits due to the `vis` array marking visited nodes.

The space complexity of the given code is also `O(n)`. This is due to the allocation of two arrays of size `n`: `ans` array to store the answer for each node, and the `vis` array to keep track of the visitation status of each node. There are no additional data structures or recursive calls that would increase the space usage, so it remains linear with respect to the number of nodes `n`.