802. Find Eventual Safe States

Medium | Depth-First Search | Breadth-First Search | Graph | Topological Sort | Leetcode Link

Problem Description

In this problem, we are given a directed graph with `n` nodes labeled from `0` to `n-1`. The graph is defined by a 2-dimensional array `graph` where `graph[i]` contains a list of nodes that have directed edges from node `i`. If a node doesn't have any outgoing edges, we label it as a *terminal node*. Our task is to find all the *safe nodes* in this graph. A *safe node* is one from which every possible path leads to a terminal node or to another safe node (which will eventually lead to a terminal node).

Our goal is to return an array of all the safe nodes in sorted ascending order.

Intuition

The concept of *safe nodes* can be thought of in terms of a topological sort—the nodes that can eventually reach terminal nodes without falling into a cycle are considered safe. We can represent each node with different states to help us in our search for safe nodes: unvisited (color 0), visiting (color 1), and safe (color 2).

The depth-first search (DFS) algorithm can help us perform this search. If we are visiting a node and encounter a node that is already being visited, this means we have a cycle, and hence, the starting node cannot be a safe node. On the contrary, if all paths from a node lead to nodes that are already known to be safe or terminal, the node in question is also safe.

We make use of a coloring system to mark the nodes:

- Color 0: The node has not been visited yet.
- Color 1: The node is currently being visited (we're in the process of exploring its edges).
- Color 2: The node and all its children can safely lead to a terminal node.

We start our DFS with the unvisited nodes and explore their children. If during the exploration, we encounter a node that is currently being visited (color 1), this means there is a cycle, and we return False, marking the node as unsafe. We continue this process for all nodes, and those that end up marked as color 2 (safe), are added to our list of safe nodes.

This approach ensures that we are only considering nodes that lead to terminal nodes as safe, effectively implementing a topological sort, which is suitable for such dependency-based problems.

Solution Approach

The solution approach is centered on using the concepts of Depth-First Search (DFS) and coloring to determine which nodes are safe in the graph.

The key part of the implementation is the `dfs` function, which is recursive and performs the following steps:

1. Check if the current node `i` has already been visited:
   - If so, return whether it's a safe node (`color[i] == 2`).
2. Mark the node as currently being visited (color it with `1`).
3. Iterate through all the adjacent nodes (those found in `graph[i]`):
   - For each adjacent node, call the `dfs` function on it. If the `dfs` on any child returns `False`, marking it as part of a cycle or path to an unsafe node, the current node `i` cannot be a safe node, so return `False`.
4. If all adjacent nodes are safe or lead to safe nodes, mark the current node `i` as safe (color it with `2`) and return `True`.

The driver code does the following:

- Initializes an array `color` with `n` elements as `0`, to store the state (unvisited, visiting, or safe) of each node.
- It then iterates over each node and applies the `dfs` function. If the `dfs` function returns `True`, it indicates the node is safe, and it's added to the final list of safe nodes.
- The list of safe nodes is then returned, which are the nodes from which every path leads to a terminal node or eventually reaches another safe node.

This approach is efficient as it marks nodes that are part of cycles as unsafe early on through the depth-first search and avoids reprocessing nodes that have already been determined to be safe or unsafe. This minimizes the exploration we need to do when determining the safety of subsequent nodes.

Example Walkthrough

Let's consider a simple directed graph with four nodes (0 to 3) defined by the 2-dimensional array `graph`:

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

Here, we have edges from node 0 to nodes 1 and 2, from node 1 to node 2, and from node 2 to node 3. Node 3 doesn't have any outgoing edges, so it's a terminal node.

We apply our DFS-based solution approach to identify safe nodes.

1. Start with node 0, which is unvisited. We mark it as visiting (color it with `1`).
2. We see that node 0 has two adjacent nodes: node 1 and node 2. We explore node 1 first:
   - Node 1 is unvisited; mark it as visiting (color `1`).
   - Node 1 has one adjacent node, node 2. We explore node 2:
     - Node 2 is unvisited; mark it as visiting (color `1`).
     - Node 2 has one adjacent node, node 3. We explore node 3:
       - Node 3 is unvisited and has no outgoing edges, so it's a terminal node. Mark it as safe (color `2`).
     - Since node 3 is safe, we mark node 2 as safe (color `2`) and return `True`.
   - Since node 2 is safe, we mark node 1 as safe (color `2`) and return `True`.
3. Then we resume exploring the adjacent nodes of node 0 and move on to node 2.
   - Node 2 is already marked safe (color `2`), so we return `True`.
4. All paths from node 0 lead to safe nodes, so we mark node 0 as safe (color `2`).

Since we've visited all nodes and no cycles were detected, and all nodes lead to a terminal node, all nodes (0, 1, 2, 3) are marked as safe.

In the end, the driver code collects all nodes colored as `2`, sorts them (which is not necessary in this case, as the array is already sorted), and returns the list of safe nodes: `[0, 1, 2, 3]`.

Python Solution

```python
from typing import List

class Solution:
    def eventualSafeNodes(self, graph: List[List[int]]) -> List[int]:
        # Helper function to perform a depth-first-search (dfs)
        # to determine if a node leads to a cycle (not safe) or not.
        def dfs(node_index):
            # If the node is already visited, return True if it is safe (color 2)
            if node_colors[node_index]:
                return node_colors[node_index] == 2
            # Mark this node as visited (color 1)
            node_colors[node_index] = 1
            # Traverse all connected nodes to see if they lead to a cycle
            for next_node_index in graph[node_index]:
                # If a connected node is not safe, then this node is also not safe
                if not dfs(next_node_index):
                    return False
            # If all connected nodes are safe, mark this node as safe (color 2)
            node_colors[node_index] = 2
            return True

        # Get the number of nodes in the graph
        total_nodes = len(graph)
        # Initialize a list to store the status of the nodes
        # Color 0 means unvisited, 1 means visiting, 2 means safe
        node_colors = [0] * total_nodes
        # Use list comprehension to gather all nodes that are safe after DFS
        # These are the eventual safe nodes that do not lead to any cycles
        safe_nodes = [node_index for node_index in range(total_nodes) if dfs(node_index)]
        return safe_nodes

```

Java Solution

```java
class Solution {
    private int[] nodeColors;
    private int[][] graph;

    // Method to determine all the eventual safe nodes in a graph
    public List<Integer> eventualSafeNodes(int[][] graph) {
        int n = graph.length;
        nodeColors = new int[n]; // Initialize array to store the state of each node
        this.graph = graph; // Assign graph to class variable for easy access
        List<Integer> safeNodes = new ArrayList<>(); // List to store eventual safe nodes

        // Iterate over each node to determine if it's a safe node
        for (int i = 0; i < n; ++i) {
            if (isNodeSafe(i)) { // If the current node is safe, add it to the list
                safeNodes.add(i);
            }
        }
        return safeNodes; // Return the final list of safe nodes
    }

    // Helper method - Conducts a depth-first search to determine if a node is safe.
    // A node is considered safe if all its possible paths lead to a terminal node.
    private boolean isNodeSafe(int node) {
        if (nodeColors[node] > 0) { // If the node is already visited, return its state
            return nodeColors[node] == 2; // Return true if the node leads to a terminal node, i.e., is safe (color coded as 2)
        }
        nodeColors[node] = 1; // Mark the node as visited (color coded as 1)

        // Explore all connected nodes recursively
        for (int neighbor : graph[node]) {
            if (!isNodeSafe(neighbor)) { // If any connected node is not safe, then this node is not safe either
                return false;
            }
        }

        nodeColors[node] = 2; // Since all connected nodes are safe, mark this node as safe (color coded as 2)
        return true; // Return true indicating the node is safe
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    vector<int> colors; // Color array to mark the states of nodes: 0 = unvisited, 1 = visiting, 2 = safe

    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
        int n = graph.size();
        colors.assign(n, 0); // Initialize all nodes as unvisited
        vector<int> safeNodes; // List to hold the eventual safe nodes

        // Check each node to see if it's eventually safe
        for (int i = 0; i < n; ++i) {
            if (dfs(i, graph)) {
                safeNodes.push_back(i); // If it is safe, add it to the list
            }
        }

        return safeNodes; // Return the list of safe nodes
    }

    // Depth-first search to determine if a node is safe
    bool dfs(int nodeIndex, vector<vector<int>>& graph) {
        if (colors[nodeIndex]) {
            // If the node has been visited already, return true only if it's marked as safe
            return colors[nodeIndex] == 2;
        }

        colors[nodeIndex] = 1; // Mark the node as visiting
        for (int neighbor : graph[nodeIndex]) {
            // Explore all the neighbors of the current node
            if (!dfs(neighbor, graph)) {
                // If any neighbor is not safe, the current node is not safe either
                return false;
            }
        }

        colors[nodeIndex] = 2; // Mark the node as safe
        return true; // Return true as the node is safe
    }
};

```

Typescript Solution

```typescript
// Function to determine the eventual safe nodes in a graph
// @param graph - The adjacency list representation of a directed graph
// @returns An array of indices representing eventual safe nodes
const eventualSafeNodes = (graph: number[][]): number[] => {
    const n: number = graph.length; // Number of nodes in the graph
    const color: number[] = new Array(n).fill(0); // Array to mark the state of each node: 0 = unvisited, 1 = visiting, 2 = safe

    // Depth-first search function to determine if a node leads to a cycle or not
    // @param nodeIndex - The current node index being visited
    // @returns A boolean indicating if the node is safe (does not lead to a cycle)
    const dfs = (nodeIndex: number): boolean => {
        if (color[nodeIndex]) {
            // If the node has been visited, return true if it's marked as safe, false otherwise
            return color[nodeIndex] === 2;
        }
        color[nodeIndex] = 1; // Mark the node as visiting
        for (const neighbor of graph[nodeIndex]) {
            // Visit all neighbors to see if any lead to a cycle
            if (!dfs(neighbor)) {
                // If any neighbor is not safe, the current node is not safe either
                return false;
            }
        }
        color[nodeIndex] = 2; // Mark the node as safe since no cycles were found from it
        return true; // The node is safe
    };

    let ans: number[] = []; // Initialize an array to keep track of safe nodes
    for (let i = 0; i < n; ++i) {
        // Iterate over each node in the graph
        if (dfs(i)) {
            // If the node is safe, add it to the result
            ans.push(i);
        }
    }
    return ans; // Return the list of safe nodes
};

```

Time and Space Complexity

Time Complexity

The time complexity is `O(N + E)`, where `N` is the number of nodes and `E` is the number of edges in the graph. This is because each node is processed exactly once, and we perform a Depth-First Search (DFS) that in total will explore each edge once. When a node is painted grey (`color[i] = 1`), it represents that the node is being processed. If it is painted black (`color[i] = 2`), the node and all nodes leading from it are safe. The DFS will terminate early if a loop is found (thus encountering a grey node during DFS), ensuring each edge is only fully explored once in the case of a safe node.

Space Complexity

The space complexity is `O(N)`. This is due to the `color` array, which keeps track of the state of each node, using space proportional to the number of nodes `N`. Additionally, the system stack space used by the recursive DFS calls must also be considered; in the worst case, where the graph is a single long path, the recursion depth could potentially be `N`, adding to the space complexity. However, since space used by the call stack during recursion and the `color` array are both linear with respect to the number of nodes, the overall space complexity remains `O(N)`.