1273. Delete Tree Nodes

Medium Tree Depth-First Search Breadth-First Search Leetcode Link

Problem Description

This problem presents us with a tree where each node contains a specific value. The tree is rooted at node 0, and consists of `nodes` number of nodes. Each node `i` has a value `value[i]` and a parent node indexed by `parent[i]`. The task is to remove all subtrees which have a total sum of node values equal to zero. A subtree here is defined as a node, all its descendants, and all the connections between them. The objective is to determine the count of nodes remaining in the tree after all such zero-sum subtrees are removed.

Intuition

The intuition behind the solution comes from a postorder depth-first search (DFS) traversal of the tree. In postorder traversal, we visit a node's children before the node itself. This gives us a way to calculate the sum of the subtree rooted at each node, starting from the leaves up to the root.

The core idea is to:

1. Create graph `g` where each node keeps a list of its children.
2. Perform a DFS starting from the root node (indexed as 0) and recursively call DFS on its children to calculate the total sum of the subtree rooted at that node as well as the number of nodes in this subtree.
3. During the DFS, when we calculate the sum of the values in a subtree, if the sum is zero, we set the count of nodes in that subtree to zero, effectively removing it.
4. Return the sum and count of nodes from this subtree.
5. Upon completion of DFS, the result from the root node will give us the number of remaining nodes after all the zero-sum subtrees have been removed.

This approach capitalizes on the fact that a zero-sum subtree higher in the tree will include any zero-sum subtrees within it, meaning we will correctly "remove" those zero-sum subtrees in the process.

Solution Approach

The implementation of the solution approach involves two main parts: building a representation of the tree and performing a depth-first search to process and remove zero-sum subtrees.

The following data structures and algorithms are used:

- A `defaultdict` of lists to represent the graph (`g`). This is used to store the tree structure where each key is a node and the values are the list of its children.
- A recursive depth-first search (`dfs`) function that takes a node index (`i`) as an argument and returns a tuple containing the sum of values of the subtree rooted at that node and the number of remaining nodes in that subtree.

Here is a step-by-step breakdown of how the implementation works:

1. **Graph construction**: We iterate through the `parent` list starting from the first node (ignoring the root node at index 0, since it does not have a parent) and create an adjacency list representation of the tree. For each child node, it is appended to the list of its parent node in the graph. This is done by the code snippet:

   ```python
   g = defaultdict(list)
   for i in range(1, nodes):
       g[parent[i]].append(i)
   ```

2. **Postorder DFS traversal**: The `dfs` function works recursively and is called for the root node to start the process. It performs postorder traversal:
   - Initially, for each node, we assign `s` to its own value and set `m` (the node count) to 1.
   - The function then iterates over all children of the current node, stored in `g`, and calls `dfs` for each child node.
   - The sum of values `s` and the count `m` are updated with the results from children nodes. This is where the sum of the subtree is accumulated.
   - If the sum of the subtree up to this point is zero (`s == 0`), we set `m` to 0, effectively discarding this subtree.
   - The function returns a tuple `(s, m)`, representing the sum and the count of nodes in the current subtree.
3. **Cleaning and final result**: After the `dfs(0)` is called, it cascades up the recursion stack and sums up child nodes, checking for sums of zero. It uses the return value of `m` to determine if the subtree should be discarded. The final number of nodes of the whole tree is returned using `dfs(0)[1]` which is the second element of the tuple indicating the count of nodes remaining.

The entire function and postorder DFS ensure we remove all zero-sum subtrees and count the number of nodes left effectively. After running this algorithm, we get the answer to our problem—how many nodes are remaining in the tree after zero-sum subtrees are removed.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose the input is such that `nodes = 6`, the `values` array that represents each node's value is `[1, -2, -3, 4, 3, -3]`, and the `parent` array that tells us each node's parent is `[-1, 0, 1, 0, 1, 1]`. This means the tree structure is the following:

```
     0(1)
    /   \
  1(-2) 3(4)
  / |
2(-3) 4(3)
|
5(-3)
```

The numbers in parentheses are the node values.

1. **Graph construction**: Based on the `parent` array, our graph `g` will be:
   ```
   0: [1, 3]
   1: [2, 4, 5]
   ```

2. **Postorder DFS traversal**:
   - When the `dfs` is called on node 0, it will then call `dfs` on its children 1 and 3.
   - The `dfs(3)` call will immediately return `(4, 1)` because node 3 has no children.
   - `dfs(1)` on the other hand calls `dfs` on nodes 2, 4, and 5.
   - `dfs(2)` returns `(-3, 1)`, `dfs(4)` returns `(3, 1)`, and `dfs(5)` returns `(-3, 1)`.
   - `dfs(1)` accumulates the values and counts coming from its children (`-3 + 3 + -3 = -3`) and its own value to find the sum is `-2 - 3 = -5`, which is not zero. Thus, it returns `(-5, 4)` because it has 3 children plus itself, but the subtree at node 1 is not a zero-sum subtree.
   - Back to `dfs(0)`, we combine the results from its children `1` and `3`. The sum is `1 + 4 + -5 = 0` and the count would normally be 6. However, since this subtree sums to zero, we now set the count to `0`.

So, as per the `dfs` on node 0, the entire tree sums to zero, meaning all nodes would be removed. Hence, the final result would be `0`, since the zero-sum subtree in this case is the entire tree itself.

The output for our example would be that there are `0` nodes remaining after removing zero-sum subtrees. This is because eliminating the zero-sum subtree rooted at node 0 (the entire tree) leaves us with no nodes.

Python Solution

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

class Solution:
    def deleteTreeNodes(self, nodes: int, parents: List[int], values: List[int]) -> int:
        # Helper function to perform Depth-First Search (DFS)
        def dfs(node_index):
            # Initialize the sum of subtree values and count of nodes as the node's own value and 1, respectively.
            subtree_sum, node_count = values[node_index], 1

            # Recur for all children of the current node.
            for child_index in graph[node_index]:
                child_sum, child_count = dfs(child_index)
                subtree_sum += child_sum
                node_count += child_count

            # If the total sum of the subtree including the current node is zero,
            # the entire subtree is deleted, so set the node count to zero.
            if subtree_sum == 0:
                node_count = 0

            # Return tuple of subtree sum and number of nodes after deletions (if any).
            return (subtree_sum, node_count)

        # Generating the graph from the parent array, representing the tree structure.
        graph = defaultdict(list)
        for i in range(1, nodes):
            graph[parents[i]].append(i)

        # Starting DFS from the root node (node index 0) and returning the count of remaining nodes.
        return dfs(0)[1]

# Example usage:
# sol = Solution()
# print(sol.deleteTreeNodes(nodes=7, parents=[-1,0,0,1,2,2,2], values=[1,-2,4,0,-2,-1,-1]))
# The above call would return 2, as two nodes remain after deleting nodes with a subtree sum of 0.

```

Java Solution

```java
class Solution {
    private List<Integer>[] graph; // Adjacency list to represent the tree
    private int[] nodeValues; // Values corresponding to each node

    // Computes the number of nodes remaining after deleting nodes with a subtree sum of zero
    public int deleteTreeNodes(int nodes, int[] parent, int[] value) {
        // Initialize graph representation of nodes
        graph = new List[nodes];
        for (int i = 0; i < nodes; i++) {
            graph[i] = new ArrayList<>();
        }
        // Build the tree structure in the adjacency list format
        for (int i = 1; i < nodes; i++) {
            graph[parent[i]].add(i); // Add child to parent's list
        }
        // Assign node values
        this.nodeValues = value;
        // Perform depth-first search and determine the result
        return dfs(0)[1]; // The second value in the returned array is the count of remaining nodes
    }

    // Performs a depth-first search and returns an array with subtree sum and number of nodes
    private int[] dfs(int nodeIndex) {
        // Initialize with the current node's value and count (1)
        int[] result = new int[]{nodeValues[nodeIndex], 1};
        // Process all the children of the current node
        for (int childIndex : graph[nodeIndex]) {
            int[] subtreeResult = dfs(childIndex);
            // Add child's values to current sum and count
            result[0] += subtreeResult[0];
            result[1] += subtreeResult[1];
        }
        // If the subtree sum equals 0, the entire subtree is deleted
        if (result[0] == 0) {
            result[1] = 0; // Set the count to 0 because the subtree will be deleted
        }
        return result; // Return the sum and count of the subtree rooted at the current node
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int deleteTreeNodes(int totalNodes, vector<int>& parent, vector<int>& value) {
        // Build the graph as an adjacency list where each node points to its children
        vector<vector<int>> graph(totalNodes);
        for (int i = 1; i < totalNodes; ++i) {
            graph[parent[i]].emplace_back(i);
        }

        // Recursive depth-first search function that returns the sum of values and count of nodes in a subtree
        // including the root of that subtree
        function<pair<int, int>(int)> depthFirstSearch = [&](int node) -> pair<int, int> {
            int sum = value[node]; // Start with the value of current node
            int count = 1; // Count of nodes starts at 1 for the current node
            // Iterate over the children of the current node
            for (int child : graph[node]) {
                auto [childSum, childCount] = depthFirstSearch(child); // Recursively get sum and count of child subtree
                sum += childSum; // Add child subtree sum to current sum
                count += childCount; // Add child subtree count to current count
            }
            // If the sum of the subtree including the current node is 0, discard the subtree
            if (sum == 0) {
                count = 0;
            }
            return pair<int, int>{sum, count}; // Return the resulting sum and count of the subtree
        };

        // Start the DFS at the root node (0) and return the count of the remaining nodes in the tree
        return depthFirstSearch(0).second;
    }
};

```

Typescript Solution

```typescript
type TreeNode = { sum: number; count: number };

// The input arrays represent the parents and the respective value of each tree node
let totalNodes: number;
let parent: number[];
let value: number[];

// The adjacency list will represent the graph where each node points to its children
let graph: number[][] = [];

// Function to build the graph as an adjacency list
function buildGraph(): void {
    graph = Array.from({ length: totalNodes }, () => []);
    for (let i = 1; i < totalNodes; ++i) {
        graph[parent[i]].push(i);
    }
}

// Recursive depth-first search to calculate the sum of node values in the subtree and count of nodes
function depthFirstSearch(node: number): TreeNode {
    let sum: number = value[node]; // Initialize sum with the value of the current node
    let count: number = 1; // Initialize count as 1 for the current node

    // Process all children of the current node
    for (let child of graph[node]) {
        let { sum: childSum, count: childCount } = depthFirstSearch(child); // Recursive call
        sum += childSum; // Increment the sum by the child's subtree sum
        count += childCount; // Increment count by the child's subtree count
    }

    // If the subtree sum is zero, the entire subtree is discarded
    if (sum === 0) count = 0;

    return { sum, count }; // Return the sum and count of the subtree
}

// Main function to delete tree nodes based on the given rules
function deleteTreeNodes(totalNodes: number, parent: number[], value: number[]): number {
    // Initialize variables with provided inputs
    this.totalNodes = totalNodes;
    this.parent = parent;
    this.value = value;

    buildGraph(); // Construct the graph from input arrays

    // Start DFS from the root node (0) and return the count of remaining nodes after deletions
    return depthFirstSearch(0).count;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(N)`, where `N` is the number of nodes in the tree.

The reasoning for this complexity is as follows:

1. The code performs a single depth-first search (DFS) traversal of the tree.
2. During the traversal, for each node, it aggregates the sum of values (`s`) and counts the number of nodes (`m`) in its subtree including itself, unless the subtree sum is zero, in which case the count is set to zero.
3. Each node is visited exactly once during the DFS, and the work done at each node is proportional to the number of its children, which, across all nodes, sums to `N - 1` (total edges in a tree with `N` nodes).

Therefore, the entire operation is linear with respect to the number of nodes.

Space Complexity

The space complexity of the given code is `O(N)`.

The reasoning for this complexity is as follows:

1. The `g` variable is a dictionary representation of the tree, and in the worst case, it might need to store `N` lists (one for each node). Each list might contain a separate child, and therefore, the total space taken up by `g` is proportional to `N`.
2. The recursion stack during the DFS traversal will, in the worst case, go as deep as the height of the tree. In the worst case of a skewed tree, where the tree takes the form of a list, the height could also be `N`.
3. No other significant space-consuming structures or variables are used.

Combining the space requirements for the dictionary and the recursion stack gives us a combined space complexity of `O(N)`.