1448. Count Good Nodes in Binary Tree

Medium Tree Depth-First Search Breadth-First Search Binary Tree

Problem Description

The problem requires counting the "good" nodes in a binary tree. A "good" node is defined as a node where, along the path from the root of the tree to that node, there are no nodes with a value greater than the value of the node itself. To simplify, if you start at the root and walk towards the node, every ancestor's value you encounter must be less than or equal to the node's value for it to be considered good. We must traverse the tree and count how many such good nodes exist.

Intuition

The key to solving this problem is to maintain the max value encountered along the path from the root to the current node. We perform a depth-first search (DFS) traversal of the tree and carry the maximum value found so far to each node's children. At each node, we do two checks:

1. If the current node's value is greater than or equal to the max value we've seen so far on this path, it qualifies as a good node, and we increment our count of good nodes. We also update the max value to the current node's value, because it's now the highest value seen on the path for the subtree rooted at this node.

2. We continue to traverse the tree by going left and right, carrying forward this updated max value to be used for the node's children.

The code defines a recursive `dfs` function that takes the current node and the current max value as parameters. If the node is `None`, we've hit a leaf's child, and there's nothing more to do, so we return. If the node is good, we increment our global answer `ans` by 1 and update the max value if necessary. Then we call `dfs` on the left and right children, ensuring that we pass the potentially updated max value. The code starts with a max value initialized to a very small number to ensure that the root node is always considered good (since there's no value in the tree less than this initial value).

The global variable `ans` is used to retain the count of good nodes found during the DFS traversal. After the traversal is completed, `ans` will store the total number of good nodes, and it's returned by the `goodNodes` function as the final answer.

Solution Approach

The solution implements a Depth-First Search (DFS) algorithm to traverse the tree. DFS is a common tree traversal technique that explores as far as possible along each branch before backtracking. This allows the solution to keep track of the current path's maximum value and check for "good" nodes.

The given Python code defines a nested `dfs` function within the `goodNodes` method. The `dfs` function is responsible for traversing the tree. It is called recursively for the root node initially, with the lowest possible integer value `mx` set as the initial maximum value (`-1000000`) encountered along the path. This is to ensure that the root node is always considered as a “good” node, since its value will certainly be higher than this minimum value.

The data structure used here is the given binary tree structure with `TreeNode` objects. Each `TreeNode` object contains a value `val`, and two pointers, `left` and `right`, pointing to its child nodes.

Here's what happens in the recursive `dfs` function:

- The function receives a `TreeNode` object and the current path's max value `mx`.
- It first checks if the current node is `None`. If so, the recursion ends (base case).
- If the current node is not `None`, it checks if the node's value is greater than or equal to `mx`. If it is, it increments the counter variable `ans` by 1 since this node is "good".
- The maximum value `mx` may be updated to the current node's value if it is indeed higher.
- The `dfs` function is then recursively called with the left child of the current node and the updated max value, followed by the recursive call with the right child and the updated max value.

This process continues recursively, visiting all the nodes in the tree, checking each node's value against the maximum value seen so far along that path, and updating the count of good nodes in the global `ans` variable. After the DFS is completed, `ans` will hold the count of all good nodes, which is then returned by the `goodNodes` method.

Here's an example of how the `dfs` function is defined within the `goodNodes` method, and how it gets called initially:

```python
def goodNodes(self, root: TreeNode) -> int:
    def dfs(root: TreeNode, mx: int):
        # rest of the dfs implementation

    ans = 0
    dfs(root, -1000000)  # Start the DFS with the root node and a min initial value
    return ans
```

This DFS pattern, combined with the use of a recursive helper function and a global counter variable, encapsulates the desired logic in a clean and efficient manner to solve the task at hand.

Example Walkthrough

Let's run through a small example using a binary tree to illustrate the solution approach outlined. Suppose we have the following binary tree:

```
       3
      / \
     1   4
    /   / \
   3   1   5
```

We want to count the number of "good" nodes in this tree. A node is "good" if no value greater than that node's value is encountered from the root to that node itself.

1. We start the DFS with the root node (value 3) and the minimum initial value as `mx = -1000000`.

2. Since the root node's value (3) is greater than `mx`, we count it as a "good" node. The "good" nodes count `ans` is now 1.

3. Recursively call DFS on the left child (value 1) with `mx = 3` (the value of the root node, since it was larger).

4. The left child's value (1) is not greater than the current `mx = 3`, so we do not increment `ans`. The "good" nodes count remains 1.

5. Recursively call DFS on the left child's left child (value 3) with `mx = 3`.

6. This left child's left child's value (3) is equal to `mx`, so it's a "good" node. Increment `ans` to 2.

7. Since the left child (value 1) has no right child, we backtrack and continue the DFS on the right child of the root (value 4) with `mx = 3`.

8. The right child's value (4) is greater than `mx`, so it's "good". Increment `ans` to 3 and update `mx` to 4.

9. Recursively call DFS on the right child's left child (value 1) with `mx = 4`.

10. The right child's left child's value (1) is not greater than `mx = 4`. We do not increment `ans`.

11. Recursively call DFS on the right child's right child (value 5) with `mx = 4`.

12. The right child's right child's value (5) is greater than `mx`, so it's "good". Increment `ans` to 4 and update `mx` to 5.

Now that the entire tree has been traversed, we can conclude that there are 4 "good" nodes in this tree.

Hence, the `goodNodes` method will return 4, which is the total count of good nodes for this example binary tree.

Python Solution

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def goodNodes(self, root: TreeNode) -> int:
        # Inner function to perform depth-first search (DFS) on the tree.
        def dfs(node: TreeNode, max_val: int):
            # Base case: if the current node is None, return from the function.
            if node is None:
                return

            # Using nonlocal keyword to modify the 'good_nodes_count'
            # variable defined in the parent function's scope
            nonlocal good_nodes_count

            # If the current node's value is greater than or equal
            # to the max value encountered so far, it is a 'good' node.
            if max_val <= node.val:
                # Increment count of 'good' nodes.
                good_nodes_count += 1
                # Update max value to current node's value.
                max_val = node.val

            # Recursively call dfs for the left child with updated max value.
            dfs(node.left, max_val)
            # Recursively call dfs for the right child with updated max value.
            dfs(node.right, max_val)

        # Initialize count of 'good' nodes to 0.
        good_nodes_count = 0
        # Invoke dfs with the root of the tree and a very small initial max value.
        dfs(root, float('-inf'))
        # Return final count of 'good' nodes.
        return good_nodes_count
```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val;         // Value of the node
    TreeNode left;   // Reference to the left child
    TreeNode right;  // Reference to the right child

    // Constructor for a tree node with no children
    TreeNode() {}

    // Constructor for a tree node with a specific value
    TreeNode(int value) { this.val = value; }

    // Constructor for a tree node with a value and references to left and right children
    TreeNode(int value, TreeNode leftChild, TreeNode rightChild) {
        this.val = value;
        this.left = leftChild;
        this.right = rightChild;
    }
}

public class Solution {
    private int numGoodNodes = 0; // Variable to keep count of good nodes

    // Public method that starts the depth-first search and returns the number of good nodes
    public int goodNodes(TreeNode root) {
        dfsHelper(root, Integer.MIN_VALUE);
        return numGoodNodes;
    }

    // Helper method that performs a depth-first search on the tree
    private void dfsHelper(TreeNode node, int maxSoFar) {
        if (node == null) {
            return; // Base case: if the node is null, return
        }
        if (maxSoFar <= node.val) {
            // If the current value is greater than or equal to the maximum value so far,
            // it is a good node, so increment the counter and update the maximum value
            numGoodNodes++;
            maxSoFar = node.val;
        }
        dfsHelper(node.left, maxSoFar); // Recursively call helper for left subtree
        dfsHelper(node.right, maxSoFar); // Recursively call helper for right subtree
    }
}
```

C++ Solution

```cpp
#include <algorithm> // For using max function
#include <functional> // For std::function

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {} // default constructor
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} // constructor with value
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {} // constructor with value and left, righ
};

class Solution {
public:
    int goodNodes(TreeNode* root) {
        int countOfGoodNodes = 0; // This will keep track of the number of good nodes

        // Depth First Search function that traverses the tree.
        // It maintains the maximum value seen so far on the path from the root to the current node.
        std::function<void(TreeNode*, int)> dfs = [&](TreeNode* node, int maxValueSoFar) {
            if (!node) {
                return; // Base case: if the node is null, return
            }
            // If the current node's value is greater than or equal to the max value seen so far,
            // increment the count of good nodes and update the max value for the path.
            if (maxValueSoFar <= node->val) {
                ++countOfGoodNodes;
                maxValueSoFar = node->val;
            }
            // Continue the DFS traversal on the left and right children of the current node.
            dfs(node->left, maxValueSoFar);
            dfs(node->right, maxValueSoFar);
        };

        // Start the DFS with the initial max value as the minimum possible integer value.
        dfs(root, INT_MIN);

        return countOfGoodNodes; // Return the final count of good nodes.
    }
};
```

Typescript Solution

```typescript
// Global variable to keep track of the number of good nodes
let goodNodesCount: number = 0;

/**
 * Represents a node in a binary tree.
 */
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

/**
 * Calculate the number of good nodes in a binary tree.
 * A good node is a node that is the largest value from root to the node itself.
 *
 * @param {TreeNode | null} node - The node to start the deep-first search from.
 * @param {number} maxSoFar - The largest value encountered from the root to the current node.
 */
function traverseAndCountGoodNodes(node: TreeNode | null, maxSoFar: number): void {
    if (!node) {
        return;
    }
    if (maxSoFar <= node.val) {
        goodNodesCount++;
        maxSoFar = node.val; // Update maxSoFar if the current node has a higher value
    }
    // Traverse left and right subtrees
    traverseAndCountGoodNodes(node.left, maxSoFar);
    traverseAndCountGoodNodes(node.right, maxSoFar);
}

/**
 * Entry function to count the number of good nodes in a binary tree starting from the root.
 *
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @returns {number} - The count of good nodes.
 */
function goodNodes(root: TreeNode | null): number {
    goodNodesCount = 0; // Reset the global count for good nodes
    traverseAndCountGoodNodes(root, -Infinity); // Start DFS with the lowest possible value
    return goodNodesCount; // Return the count of good nodes
}
```

Time and Space Complexity

The provided Python code defines a function `goodNodes` that counts the number of "good" nodes in a binary tree. A "good" node is defined as a node whose value is greater than or equal to all the values in the nodes that lead to it from the root. The function implements a depth-first search (DFS) to traverse the tree and count these nodes.

Time Complexity

The time complexity of the code is determined by how many nodes are visited during the DFS traversal. The function visits every node exactly once. Therefore, the time complexity is `O(N)`, where `N` is the number of nodes in the tree.

Space Complexity

The space complexity is primarily determined by the call stack due to the recursive nature of the DFS. In the worst-case scenario (a skewed tree), the depth of the recursive call stack can become `O(N)` if the tree is a linear chain of `N` nodes. In the average case of a balanced tree, the height would be `O(logN)`, resulting in `O(logN)` recursive calls at any given time. However, since we need to consider the worst case, the space complexity of the code is `O(N)`.