257. Binary Tree Paths

Easy Tree Depth-First Search String Backtracking Binary Tree Leetcode Link

Problem Description

The problem presents us with a binary tree, where each node contains an integer value and can have a left and/or right child. Our goal is to find all the unique paths from the root of the binary tree to its leaves. A leaf is defined as a node that has no children, which means neither a left nor a right child. The paths should be represented as strings, with each node's value on the path concatenated by "→". For instance, if the path goes through nodes with values 1, 2, and 3 in that order, the path string should be "1→2→3". We are required to return these path strings in any order.

Intuition

To solve this problem, we can utilize a technique called Depth-First Search (DFS). This strategy explores as far as possible along each branch before backtracking. Here's how we arrive at the solution approach:

- We can start at the root and perform a DFS traversal on the binary tree.
- As we traverse, we keep track of the current path by noting the nodes visited so far in the sequence.
- Whenever we reach a leaf node (a node without children), we record the path we've taken to get there. This is a complete root-to-leaf path, so we add this to our list of answers.
- The key part of this process is backtracking. When we visit a node and explore all of its children, we backtrack, which means we remove the node from our current path and return to the previous node to explore other paths.
- We continue this process until all paths are explored and we have visited all the leaves.

The recursive function `dfs()` in the solution code is where the DFS takes place. It takes the current node as a parameter, and as the function is called recursively, a local variable `t` keeps track of the current path as a list of node values. If a leaf node is reached, the current path is converted to the path string format required and added to the list `ans`, which contains all the full path strings. After exploring a node's left and right children, the node's value is popped from the path list to allow backtracking to the previous node's state.

Solution Approach

The solution approach for the given problem utilizes a typical pattern for tree traversal problems, which is the Depth-First Search (DFS) algorithm. Here's a step-by-step explanation of how the solution is implemented:

1. **Define a Recursive Function (`dfs`)**: The recursive function `dfs` is defined within the class `Solution`. It is invoked with the current node being visited. This function does not return any value but instead updates the `ans` list with the path strings.

2. **Base Case for Recursion**: In the function `dfs`, before going further into recursion, we check if the given `root` node is `None`, indicating that we've reached beyond a leaf node. In this case, the function simply returns without performing any further action.

3. **Track the Current Path**: A local variable `t` in the class scope is used to keep track of the current path. It's a stack (implemented using a list in Python) that we update as we go down the tree. For each node, we convert its value to a string and append it to `t`.

4. **Check for Leaf Node**: In the DFS process, if the current node is a leaf (both `left` and `right` child nodes are `None`), we convert the current path into the required string format, which is obtained by joining the sequence of node values in `t` with `"->"`. This complete path string is added to the `ans` list.

5. **Recursive DFS Calls**: If the current node has children, we perform DFS for both the left child (if not `None`) and right child (if not `None`). The function calls itself with `root.left` and `root.right` correspondingly.

6. **Backtracking**: After exploring both children from the current node, we need to backtrack. This ensures that when we return from the recursive call, the current path `t` reflects the state as if the current node was never visited. We achieve this by popping the last value from the `t`.

7. **Return Paths**: Once the DFS is complete, the `ans` list will be populated with all the path strings from root to all the leaf nodes. This list is returned as the final result of the `binaryTreePaths` method.

The use of a stack for tracking the paths and the pattern of adding to the list only at leaf nodes are critical parts of the algorithm. The recursive DFS makes sure all potential paths are explored, while backtracking ensures that every unique path is properly captured without duplication.

Example Walkthrough

Let's illustrate the solution approach using a small example binary tree:

Consider the following binary tree:

```
    1
   / \
  2   3
 /     \
5       4
```

Here's how we would apply the solution approach:

1. We start with the root node which is `1`.
2. We initialize `t` with the value of the current node: `t = ["1"]`.
3. The node `1` is not a leaf, so we continue. We have two recursive DFS calls to make because both left and right children exist: `root.left` (which is `2`) and `root.right` (which is `3`).
4. **First DFS call**: We move to the left child which is `2` and update `t` to `["1", "2"]`.
5. Node `2` has a left child but no right child. We make a recursive DFS call to the left child which is `5`.
6. On reaching the node `5`, we update `t` to `["1", "2", "5"]`.
7. Node `5` is a leaf node, so we join the elements of `t` with `"->"` to form the path string "1→2→5". We add this to our `ans` list.
8. **Backtracking**: We finished visiting `5`, so we pop it from `t` making `t = ["1", "2"]`.
9. There are no more children for `2` to visit, so we pop it from `t` as well. Now `t = ["1"]`.
10. **Second DFS call**: Now, we explore the right child of `1`, which is `3`. We update `t` to `["1", "3"]`.
11. Node `3` has a right child `4` but no left child. We make a recursive DFS call to the right child which is `4`.
12. On reaching `4`, we update `t` to `["1", "3", "4"]`.
13. Node `4` is a leaf node, so we join the elements of `t` with `"->"` to form the path string "1→3→4". We add this to our `ans` list.
14. **Backtracking**: We've visited `4`, so we pop it from `t`, reverting `t` back to `["1", "3"]`.
15. After exploring node `3`'s right child, we backtrack one more time, popping `3` from `t` and `t` is now back to `["1"]`.

Finally, since all paths have been explored, our `ans` list contains the root-to-leaf path strings: `["1->2->5", "1->3->4"]`. We'd return this list as the output of our function.

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 binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:

        # Perform Depth-First Search (DFS) to find all paths
        def dfs(node: Optional[TreeNode]):
            # If the current node is None, return without doing anything
            if node is None:
                return

            # Append the current node's value to the path
            path.append(str(node.val))

            # If the current node is a leaf node, add the path to the results
            if node.left is None and node.right is None:
                all_paths.append("->".join(path))
            else:
                # Otherwise, continue the DFS traversal
                dfs(node.left)
                dfs(node.right)

            # After traversing, remove the last node value from the path
            path.pop()

        # Initialize a list to store all paths as strings
        all_paths = []

        # Initialize a temporary list to store the current path
        path = []

        # Start the DFS with the root node
        dfs(root)

        # Return all the root-to-leaf paths found
        return all_paths

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    // A list to store all path strings
    private List<String> allPaths = new ArrayList<>();
    // A temporary list to keep the current path nodes
    private List<String> currentPath = new ArrayList<>();

    /**
     * Finds all paths from root to leaf in a binary tree.
     *
     * @param root The root of the binary tree.
     * @return A list of all root-to-leaf paths in string format.
     */
    public List<String> binaryTreePaths(TreeNode root) {
        // Perform a depth-first search starting from the root
        depthFirstSearch(root);
        return allPaths;
    }

    /**
     * Performs a recursive depth-first search to find all paths.
     *
     * @param node The current node in the binary tree.
     */
    private void depthFirstSearch(TreeNode node) {
        if (node == null) {
            // Base case: if node is null, do nothing
            return;
        }

        // Append current node's value to the path
        currentPath.add(String.valueOf(node.val));

        // If node is a leaf, add the path to the list of all paths
        if (node.left == null && node.right == null) {
            allPaths.add(String.join("->", currentPath));
        } else {
            // Recur for left and right children
            depthFirstSearch(node.left);
            depthFirstSearch(node.right);
        }

        // Backtrack: remove the last node from the path before returning
        currentPath.remove(currentPath.size() - 1);
    }
}

```

C++ Solution

```cpp
// Definition of the binary tree node.
struct TreeNode {
    int val; // Node value.
    TreeNode *left; // Pointer to left child.
    TreeNode *right; // Pointer to right child.

    // Constructor for creating a leaf node.
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

    // Constructor for creating a node with a specific value.
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    // Constructor for creating a node with specific value, left child, and right child.
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Finds all root-to-leaf paths in the binary tree.
    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> paths;              // Stores all the paths from root to leaf.
        vector<string> currentPathNodes;   // Tracks nodes of the current path.

        // Depth-first search to traverse the tree and build paths.
        function<void(TreeNode*)> dfs = [&](TreeNode* node) {
            if (!node) {
                return; // Base case: if the current node is null, just return.
            }

            // Add the current node value to the path.
            currentPathNodes.push_back(to_string(node->val));

            // If the current node is a leaf, join the current path and add to paths.
            if (!node->left && !node->right) {
                paths.push_back(join(currentPathNodes));
            } else {
                // Recursively call DFS on the non-null children.
                dfs(node->left);
                dfs(node->right);
            }
            // Backtracking step: remove the current node value from the path.
            currentPathNodes.pop_back();
        };

        // Start DFS from the root node.
        dfs(root);
        return paths;
    }

    // Helper function to create a string from vector of strings representing nodes.
    string join(const vector<string>& tokens, const string& separator = "->") {
        string path;
        for (size_t i = 0; i < tokens.size(); ++i) {
            if (i > 0) {
                path += separator;
            }
            path += tokens[i];
        }
        return path; // Return the joined path as a string.
    }
};

```

Typescript Solution

```typescript
// Represents a tree node with a numerical value and optional left and right children.
type TreeNode = {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
};

/**
 * Helper function that performs a Depth-First Search on the binary tree to find all paths from the
 * root to leaf nodes.
 *
 * @param node The current node being visited in the DFS traversal.
 * @param path The list of values collected so far from the root to the current node.
 * @param paths The list of all paths from the root to leaf nodes represented as strings.
 */
function traverseAndRecordPaths(node: TreeNode | null, path: number[], paths: string[]): void {
    // Base case: if the current node is null, return to avoid further processing.
    if (!node) {
        return;
    }

    // Append the current node's value to the path.
    path.push(node.val);

    // Check if the current node is a leaf node (no left and right children).
    if (!node.left && !node.right) {
        // If it's a leaf, add the path to the list of paths as a string.
        paths.push(path.join('->'));
    } else {
        // If not a leaf, recursively search the left and right subtrees.
        traverseAndRecordPaths(node.left, path, paths);
        traverseAndRecordPaths(node.right, path, paths);
    }

    // Pop the last node's value from the path after exploring all paths from here.
    path.pop();
}

/**
 * Encapsulates the process of collecting all paths in a binary tree from root to leaf nodes.
 *
 * @param root The root of the binary tree.
 * @return The list of all root-to-leaf paths as strings.
 */
function binaryTreePaths(root: TreeNode | null): string[] {
    // Initialize an array to store all root-to-leaf paths.
    const allPaths: string[] = [];
    // Temporary variable to store the current path in the recursion.
    const currentPath: number[] = [];

    // Perform the DFS traversal starting from the root to find all paths.
    traverseAndRecordPaths(root, currentPath, allPaths);

    // Return the collected paths.
    return allPaths;
}

```

Time and Space Complexity

The given Python function `binaryTreePaths` traverses a binary tree to find all root-to-leaf paths. The primary operation is a Depth-First Search (DFS) defined in a nested function. Here's the analysis of its complexities:

Time Complexity

The time complexity is `O(N)`, where `N` is the number of nodes in the tree. This is because each node in the binary tree is visited exactly once during the depth-first search. Therefore, the function's time complexity is directly proportional to the number of nodes.

Space Complexity

The space complexity of the function is also `O(N)`. The major factors contributing to the space complexity are:

1. The recursive call stack of the DFS, which in the worst case could be `O(N)` when the binary tree degrades to a linked list.
2. The list `t` that holds the current path. In the worst case, when the binary tree is completely skewed (e.g., each parent has only one child), this list can also take up to `O(N)` space.
3. The list `ans` that contains all the paths. In the best case, where each node has two children, there will be `N/2` leaf nodes resulting in `N/2` paths. Although each path could be of varying length, the concatenation of all paths will still take `O(N)` space since each node's value is only included in one path string.

Considering the stack space for recursive calls and the space for storing the paths, the space complexity in the worst case scenario will be linear with respect to the number of nodes in the tree.