1457. Pseudo-Palindromic Paths in a Binary Tree

Medium | Bit Manipulation | Tree | Depth-First Search | Breadth-First Search | Binary Tree

Leetcode Link

Problem Description

In this problem, we are given a binary tree where each node contains a digit value from 1 to 9. A path in the binary tree is considered from the root node to any leaf node (i.e., a node without children). A pseudo-palindromic path is defined as a path where the node values can be rearranged to form a palindrome. Remember, a palindrome is a sequence of numbers that reads the same backward as forward.

To solve this problem, we need to determine the count of pseudo-palindromic paths that exist from the root node to all leaf nodes.

Intuition

The intuition here is to perform a Depth-First Search (DFS) traversal of the tree, using a counter to keep track of the frequency of each digit along the current path. Because we're interested in node values, which range from 1 to 9, an array of size 10 is enough to serve as our counter (index zero is unused).

As we traverse the binary tree, we increment the counter for the current node's value. Once we reach a leaf node, we check the counter for the palindromic property. A sequence can form a palindrome if at most one number has an odd frequency (for the middle element in the palindrome), while all others should have even frequencies.

So, for each leaf node, if the count of numbers with odd frequency is less than two, we know that the current path is a pseudo-palindromic path. If this condition is satisfied, we increase our answer count. While backtracking (going back to the parent node on the DFS traversal), we decrement the counter for the current node's value to make sure the counter only represents the frequency of digits along the current path.

Here's the step-by-step process:

1. Initialize the answer count `ans` to zero.
2. Initialize the digit frequency counter, a list `counter` of size 10, with zeroes.
3. Start the DFS from the root node, keeping track of the current path's digit frequencies.
4. Each time we visit a node:
   - Increment the frequency counter for that node's value.
   - If we reach a leaf node, check if at most one digit has an odd frequency count.
   - If the above check is true, increment the pseudo-palindromic path count `ans`.
5. Before backtracking, decrement the frequency counter for the current node's value to ensure it reflects only the active path's frequencies.
6. Continue the DFS until all paths are checked.
7. Finally, return the count `ans` which represents the number of pseudo-palindromic paths.

This recursive approach elegantly checks all root-to-leaf paths and uses the properties of palindromes to count those that can form a pseudo-palindrome.

Solution Approach

The solution uses Depth-First Search (DFS), a common algorithm for traversing or searching tree or graph data structures. Here's how the algorithm is applied in this solution:

1. A recursive function `dfs` is defined within the `pseudoPalindromicPaths` method. The function takes a single parameter `root`, representing the current node in the tree.
2. On each call, the function checks if the `root` is `None`. If so, it returns immediately because we have reached beyond a leaf node.
3. Two variables, `ans` and `counter`, are used, leveraging the `nonlocal` keyword. `ans` captures the number of pseudo-palindromic paths, and `counter` is an integer list of size 10, keeping track of the occurrences of each digit along the current path.
4. As the DFS descends down the tree, it increments `counter[root.val]` for the current node's value. This adjustment is part of keeping track of the digits' frequency in the current path.
5. If the current node is a leaf (both `root.left` and `root.right` are `None`), the algorithm checks if the path represented by `counter` can be a pseudo-palindrome. This check involves summing the number of digits that appear an odd number of times using a generator expression `sum(1 for i in range(1, 10) if counter[i] % 2 == 1)` and comparing it with `< 2`. This condition is based on the definition of a palindrome, which allows for at most one character (digit, in this case) to appear an odd number of times.
6. If the condition for a pseudo-palindromic path is met at a leaf node, `ans` is incremented.
7. The DFS continues recursively for the left and right children of the current node with `dfs(root.left)` and `dfs(root.right)`, exploring all possible paths to leaf nodes.
8. Before each recursive call returns, the counter for the current node's value is decremented with `counter[root.val] -= 1`. This backtracking step ensures that once a path has been checked, the counter reflects the correct frequency for the "active" path.
9. Once all possible paths have been checked, the `dfs` function ends, and the `pseudoPalindromicPaths` method returns `ans`; the total number of pseudo-palindromic paths from the root node to leaf nodes.

Overall, this approach effectively uses DFS to explore all paths while maintaining a frequency counter to check for the palindromic property at each leaf node. It demonstrates a combination of DFS for tree traversal and frequency counting as a method to validate the palindromic property efficiently.

Example Walkthrough

Let's consider a simple binary tree to illustrate the solution approach:

```
1
   / \
  2   3
 / \
3   1
```

Here is how the solution applies Depth-First Search (DFS):

1. Start the DFS traversal from the root node which contains the value `2`. The `counter` array is initialized to zero and looks like this: `[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]` after visiting the root. The `ans` count starts at 0.
2. The traversal moves to the left child with the value `3`. The `counter` array is updated: `[0, 0, 1, 1, 0, 0, 0, 0, 0, 0]`.
3. The traversal then moves to its left child with the value `3`. The `counter` array becomes `[0, 0, 1, 1, 0, 0, 0, 0, 0, 0]`. The current node is a leaf. The sum of digits appearing an odd number of times is 3 (digits 1, 2, and 3 have odd counts), which is more than one, so this path does not form a pseudo-palindrome. The `ans` count remains 0.
4. The traversal then backtracks to the node with the value `3` and decrements the `counter` for the node with value `3`: `[0, 0, 1, 1, 0, 0, 0, 0, 0, 0]`.
5. As the `3` node does not have a right child, it backtracks to the root node and travels to the right child, which has value `1`. The `counter` is updated to `[0, 1, 1, 1, 0, 0, 0, 0, 0, 0]`.
6. The current node is a leaf node, and again the sum of digits appearing an odd number of times is 3 (digits 1, 2, and 3). So this path is not a pseudo-palindrome either. The `ans` count remains 0.
7. The DFS has explored all the paths from the root to the leaf nodes. Since we did not find any pseudo-palindromic paths, the final `ans` returned would be 0 for this binary tree.

The algorithm has successfully checked all root-to-leaf paths for the pseudo-palindromic property by maintaining a frequency counter during traversal and has concluded that there are no paths that can form a pseudo-palindrome in the 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 pseudoPalindromicPaths(self, root: TreeNode) -> int:
        # Helper function to perform a depth-first search (DFS) on the tree.
        def dfs(node):
            if node is None:  # Base case: if the current node is None, return.
                return

            # Access the variables outside of the nested function's scope.
            nonlocal pseudo_palindrome_path_count, value_counter

            # Increment the count of the current node's value.
            value_counter[node.val] += 1

            # Check if it's a leaf node (i.e., has no child nodes).
            if node.left == node.right == None:
                # Count the number of digits that appear an odd number of times.
                odd_count = sum(1 for i in range(1, 10) if value_counter[i] % 2 == 1)
                # If there's at most one value that appears an odd number of times,
                # it's a pseudo-palindromic path.
                if odd_count < 2:
                    pseudo_palindrome_path_count += 1

            else:
                # If not a leaf, continue DFS on child nodes.
                dfs(node.left)
                dfs(node.right)

            # After visiting a node, decrement the node's value count.
            value_counter[node.val] -= 1

        # Initialize the count of pseudo-palindromic paths as 0.
        pseudo_palindrome_path_count = 0
        # Initialize a list to count the occurrences of each value (1 through 9).
        value_counter = [0] * 10
        # Start the DFS from the root.
        dfs(root)
        # Return the total number of pseudo-palindromic paths in the tree.
        return pseudo_palindrome_path_count
```

Java Solution

```java
class Solution {
    // We use a variable to keep the count of pseudo-palindromic paths
    private int pseudoPalindromicPathCount;

    // Counter array to keep track of the frequency of each digit from 1-9
    private int[] digitFrequencyCounter;

    // This method starts the DFS traversal from the root to calculate the number of pseudo-palindromic paths in the tree
    public int pseudoPalindromicPaths(TreeNode root) {
        pseudoPalindromicPathCount = 0;
        digitFrequencyCounter = new int[10];
        depthFirstSearch(root);
        return pseudoPalindromicPathCount;
    }

    // Helper method for deep-first search
    private void depthFirstSearch(TreeNode node) {
        // Base case: if the node is null, we've reached a leaf, there is nothing to do further
        if (node == null) {
            return;
        }

        // Increment the frequency of the current node's value
        digitFrequencyCounter[node.val]++;

        // If it's a leaf node, check if the path constitutes a pseudo-palindrome
        if (node.left == null && node.right == null) {
            if (isPseudoPalindromicPath()) {
                pseudoPalindromicPathCount++; // Increment the count if it is a pseudo-palindrome
            }
        } else {
            // Continue the traversal left and right in the tree if not a leaf node
            depthFirstSearch(node.left);
            depthFirstSearch(node.right);
        }

        // Backtrack: Decrement the frequency of the current node's value before returning to the parent
        digitFrequencyCounter[node.val]--;
    }

    // Check if the path represents a pseudo-palindrome
    private boolean isPseudoPalindromic() {
        int oddCount = 0; // This variable counts the number of digits which appear an odd number of times

        // A palindrome has at most one digit with an odd frequency
        for (int i = 1; i <= 9; ++i) {
            if (digitFrequencyCounter[i] % 2 == 1) {
                oddCount++;
            }
        }

        // If there is at most one odd frequency count, then we can form a pseudo-palindrome
        return oddCount < 2;
    }
}
```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    int pseudoPalindromicPaths(TreeNode* root) {
        vector<int> valueCounter(10, 0); // Variable to store the number of pseudo-palindromic paths
        dfs(root, valueCounter); // Start the DFS traversal from the root node
        return pseudoPalindromicPathCount; // Return the final count of paths
    }

private:
    int pseudoPalindromicPathCount = 0; // Counter for the count of pseudo-palindromic paths in the binary tree

    // Helper method for depth-first search
    void dfs(TreeNode* node, vector<int>& valueCounter) {
        if (node == nullptr) return; // If the current node is null, there is nothing to do, return immediately

        ++valueCounter[node->val]; // Increment the counter for the current node's value

        // If the current node is a leaf node (no left or right children)
        if (!node->left && !node->right) {
            int oddCount = 0; // Variable to count the number of odd occurrences of values
            for (int i = 1; i <= 9; ++i) // Check each value in the counter
                if (valueCounter[i] % 2 == 1) // If the count is odd,
                    ++oddCount; // increment the number of odd value counts

            // If there is at most one odd value count, the path can form a pseudo-palindrome
            if (oddCount < 2) ++pseudoPalindromicPathCount;
        } else {
            // If the current node is not a leaf, continue depth-first search on the children
            dfs(node->left);
            dfs(node->right);
        }

        --valueCounter[node->val]; // Decrement the counter for the current node's value before backtracking
    }
};
```

Typescript Solution

```typescript
/**
 * Definition for a binary tree node.
 */
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;
    }
}

// Variable to store the number of pseudo-palindromic paths
let pseudoPathCount: number = 0;

// Counter for the values in the binary tree nodes
let valueCounter: number[] = new Array(10).fill(0);

/**
 * Initializes the count of pseudo-palindromic paths in the binary tree.
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @return {number} - The count of pseudo-palindromic paths.
 */
function pseudoPalindromicPaths(root: TreeNode | null): number {
    pseudoPathCount = 0; // Initialize the palindrome path count
    valueCounter.fill(0); // Reset the counter values to zero
    dfs(root); // Start depth-first search from the root node
    return pseudoPathCount; // Return the final count of paths
}

/**
 * Helper function for performing depth-first search on the tree.
 * @param {TreeNode | null} node - The current node being visited.
 */
function dfs(node: TreeNode | null): void {
    if (!node) return; // If the current node is `null`, return immediately

    valueCounter[node.val]++; // Increment the counter for the current node's value

    // Check if the current node is a leaf (no left or right children)
    if (!node.left && !node.right) {
        let oddCount = 0; // Variable to count the number of odd occurrences of values

        // Check each value in the counter
        for (let i = 1; i <= 9; ++i) {
            // If the count is odd, increment the number of odd value counts
            if (valueCounter[i] % 2 !== 0) {
                oddCount++;
            }
        }

        // If there is at most one odd value count, the path can form a pseudo-palindrome
        if (oddCount < 2) {
            pseudoPathCount++;
        }
    } else {
        // If the current node is not a leaf, continue depth-first search on its children
        dfs(node.left);
        dfs(node.right);
    }

    valueCounter[node.val]--; // Decrement the counter for the current node's value before backtracking
}
```

Time and Space Complexity

The time complexity of the provided code can be analyzed by looking at the functions it performs:

1. The `dfs` function is called on every node of the binary tree exactly once. If we let n be the total number of nodes in the tree, the total number of calls is $O(n)$.
2. For each node of the tree, the counter updates and the palindromic condition check are performed. The counter update is a constant-time operation, $O(1)$.
3. The palindromic condition check iterates from 1 to 9, so it always performs 9 constant-time checks, which is also $O(1)$.

Adding these together, the overall time complexity of the code is $O(n)$.

The space complexity analysis takes into account:

1. The counter list, which uses space for 10 integers, so it is $O(1)$.
2. The recursive depth of the `dfs` function, which in the worst case can be $O(h)$ where h is the height of the tree. In a balanced tree, h would be $\log n$, but in the worst case (completely unbalanced tree), h can be $O(n)$.

Therefore, the space complexity of the algorithm is $O(h)$, which ranges from $O(\log n)$ in the best case (balanced tree) to $O(n)$ in the worst case (unbalanced tree).