1457. Pseudo-Palindromic Paths in a Binary Tree

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 only 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 counter 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 that node's path 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 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)
2       /    \
3     (3)   (1)
4     /
5   (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 1. The counter array becomes [0, 1, 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 1: [0, 0, 1, 1, 0, 0, 0, 0, 0, 0].

5. As the previous node does not have a right child, it backtracks to the root node and travels to the right child, which have value 1. The counter is updated to [0, 1, 1, 0, 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
1  # Definition for a binary tree node.
2  class TreeNode:
3      def __init__(self, val=0, left=None, right=None):
4          self.val = val
5          self.left = left
6          self.right = right
7
8  class Solution:
9      def pseudoPalindromicPaths(self, root: TreeNode) -> int:
10         # Helper function to perform a depth-first search (DFS) on the tree.
11         def dfs(node):
12             if node is None:  # Base case: if the current node is None, return.
13                 return
14
15             # Access the variables outside of the nested function's scope.
16             nonlocal palindrome_path_count, value_counter
17
18             # Increment the count of the current node's value.
19             value_counter[node.val] += 1
20
21             # Check if it's a leaf node (i.e., has no child nodes).
22             if node.left is None and node.right is None:
23                 # Count the values that appear odd number of times.
24                 odd_count = sum(1 for i in range(1, 10) if value_counter[i] % 2 == 1)
25                 # If there is at most one value that appears an odd number of times,
26                 # it it's a pseudo-palindromic path.
27                 if odd_count < 2:
28                     palindrome_path_count += 1
29             else:
30                 # If not a leaf, continue DFS on child nodes.
31                 dfs(node.left)
32                 dfs(node.right)
33
34             # After visiting a node, decrement the node's value count.
35             value_counter[node.val] -= 1
36
37         # Initialize the count of pseudo-palindromic paths as 0.
38         palindrome_path_count = 0
39         # Initialize a list to count the occurrences of each value (1 through 9).
40         value_counter = [0] * 10
41         # Start the DFS from the root.
42         dfs(root)
43         # Return the total number of pseudo-palindromic paths in the tree.
44         return palindrome_path_count
```

Java Solution

```java
1  class Solution {
2      // We use a variable to keep the count of pseudo-palindromic paths
3      private int pseudoPalindromicPathCount;
4
5      // Counter array to keep track of the frequency of each digit from 1-9
6      private int[] digitFrequencyCounter;
7
8      // This method starts the DFS traversal from the root to calculate the number of pseudo-palindromic paths in the tree
9      public int pseudoPalindromicPaths(TreeNode root) {
10         pseudoPalindromicPathCount = 0;
11         digitFrequencyCounter = new int[10];
12         depthFirstSearch(root);
13         return pseudoPalindromicPathCount;
14     }
15
16     // Helper method for deep-first search
17     private void depthFirstSearch(TreeNode node) {
18         if (node == null) {
19             return; // If the node is null, we backtrack as there is nothing to do further
20         }
21
22         // Increment the frequency of the current node's value
23         digitFrequencyCounter[node.val]++;
24
25         // If it's a leaf node, check if the path constitutes a pseudo-palindrome
26         if (node.left == null && node.right == null) {
27             if (isPseudoPalindrome()) {
28                 pseudoPalindromicPathCount++; // Increment the count if it is a pseudo-palindrome
29             }
30         } else {
31             // Continue the traversal left and right in the tree if not a leaf node
32             depthFirstSearch(node.left);
33             depthFirstSearch(node.right);
34         }
35
36         // Backtrack: Decrement the frequency of the current node's value before returning to the parent
37         digitFrequencyCounter[node.val]--;
38     }
39
40     // Check if the path represents a pseudo-palindrome
41     private boolean isPseudoPalindrome() {
42         int oddCount = 0; // This variable counts the number of digits which appear an odd number of times
43         for (int i = 1; i < 10; i++) {
44             if (digitFrequencyCounter[i] % 2 == 1) {
45                 oddCount++;
46             }
47         }
48         // If there is at most one odd frequency count, then we can form a pseudo-palindrome
49         return oddCount < 2;
50     }
51 }
```

C++ Solution

```cpp
1  /**
2   * Definition for a binary tree node.
3   */
4  class TreeNode {
5  public:
6      int val;
7      TreeNode *left;
8      TreeNode *right;
9      TreeNode() : val(0), left(nullptr), right(nullptr) {}
10     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
11     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
12 };
13
14 class Solution {
15 public:
16     int palindromePathCount; // Variable to store the number of pseudo-palindromic paths
17     vector<int> valueCounter; // Counter for the values in the binary tree nodes
18
19     // Public method to initiate the search for pseudo-palindromic paths in the binary tree
20     int pseudoPalindromicPaths(TreeNode* root) {
21         palindromePathCount = 0; // Initialize the palindrome path count
22         valueCounter.resize(10); // Initialize the counter with a size to include values from 0-9
23         dfs(root); // Start depth-first search from the root node
24         return palindromePathCount; // Return the final count of paths
25     }
26
27     // Helper method for depth-first search
28     void dfs(TreeNode* node) {
29         if (!node) return; // If the current node is null, there is nothing to do, return immediately
30
31         ++valueCounter[node->val]; // Increment the counter for the current node's value
32
33         // If the current node is a leaf node (no left or right children)
34         if (node->left && node->right) {
35             int oddCount = 0; // Variable to count the number of odd occurrences of values
36             for (int i = 1; i < 10; ++i) // Check each value in the counter
37                 if (valueCounter[i] % 2 == 1) // If the count is odd,
38                     ++oddCount; // increment the number of odd value counts
39
40             // If there is at most one odd value count, the path can form a pseudo-palindrome
41             if (oddCount < 2) ++palindromePathCount;
42         } else {
43             // If the current node is not a leaf, continue depth-first search on the children
44             dfs(node->left);
45             dfs(node->right);
46         }
47
48         --valueCounter[node->val]; // Decrement the counter for the current node's value before backtracking
49     }
50 };
```

Typescript Solution

```typescript
1  /**
2   * Definition for a binary tree node.
3   */
4  class TreeNode {
5      val: number;
6      left: TreeNode | null;
7      right: TreeNode | null;
8      constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
9          this.val = val;
10         this.left = left;
11         this.right = right;
12     }
13 }
14
15 // Variable to store the number of pseudo-palindromic paths
16 let palindromePathCount: number = 0;
17
18 // Counter for the values in the binary tree nodes
19 let valueCounter: number[] = new Array(10).fill(0);
20
21 // Initiates the count of pseudo-palindromic paths in the binary tree.
22 // @param {TreeNode | null} root - The root node of the binary tree.
23 // @return {number} - The count of pseudo-palindromic paths.
24 function pseudoPalindromicPaths(root: TreeNode | null): number {
25     palindromePathCount = 0; // Initialize the palindrome path count
26     valueCounter.fill(0); // Reset the counter values to zero
27     dfs(root); // Start depth-first search from the root node
28     return palindromePathCount; // Return the final count of paths
29 }
30
31 // Helper function for performing depth-first search on the tree.
32 // @param {TreeNode | null} node - The current node being visited.
33 function dfs(node: TreeNode | null): void {
34     if (!node) return; // If the current node is 'null', return immediately
35
36     valueCounter[node.val]++; // Increment the counter for the current node's value
37
38     // Check if the current node is a leaf (no left or right children)
39     if (!node.left && !node.right) {
40         let oddCount = 0; // Variable to count the number of odd occurrences of values
41
42         // Check each value in the counter
43         for (let i = 1; i < 10; i++) {
44             // If the count is odd, increment the number of odd value counts
45             if (valueCounter[i] % 2 == 1) {
46                 oddCount++;
47             }
48         }
49
50         // If there is at most one odd value count, the path can form a pseudo-palindrome
51         if (oddCount < 2) palindromePathCount++;
52     } else {
53         // If the current node is not a leaf, continue depth-first search on its children
54         dfs(node.left);
55         dfs(node.right);
56     }
57
58     valueCounter[node.val]--; // Decrement the counter for the current node's value before backtracking
59 }
```

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 O(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).