687. Longest Univalue Path

Medium Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

The given problem requires us to find the length of the longest path in a binary tree, where all the nodes along the path have the same value. A path in a binary tree is a sequence of nodes where there is an edge between any two consecutive nodes in the sequence, and the length of the path is represented by the number of edges in that sequence. This specific path can start and end at any node, not necessarily including the root.

Intuition

To solve this problem, we employ a depth-first search (DFS) strategy to traverse the tree. The key intuition here is to use recursion to explore each subtree to find the longest path with equal node values that start from the current node and extend downwards. Once a node is reached by the recursion:

1. If the node is null, we cannot extend the path, so we return zero.
2. We recursively obtain the lengths of the paths extending from its left and right child nodes that have the same value as the current node.
3. To build the answer for the current node, we consider these cases:
   - If the left child's value equals the current node's value, we increment the left path length by one to include the current node; otherwise, the longest path starting from the current node towards the left is zero, since the values don't match.
   - Similarly, if the right child's value equals the current node's value, we increment the right path length by one; if not, the right path's length is zero.
4. The longest path that goes through the current node (but not necessarily includes it) is the sum of the left and right path lengths we just calculated. We update our global answer, `ans`, if this sum is larger than the current answer.
5. Each call needs to return the maximum length from either the left or the right that includes the current node. We can't return both as we are considering paths, not splits; so we return the larger of the two.

In the provided solution, `dfs` is our recursive function, and `ans` is the variable that keeps track of the longest path found so far. After calling `dfs` on our root node, `ans` will contain the maximum length of the path we are looking for, and we return it as the solution.

Solution Approach

The solution implements a recursive depth-first search approach, similar to finding the diameter of a binary tree (as suggested in the reference solution approach). Here's a more detailed breakdown of the implementation:

1. **Recursive Depth-First Search (DFS):**
   - We define a nested recursive helper function `dfs` inside our `longestUnivaluePath` method. This function is responsible for exploring the tree depth-first, starting from the `root`.
   - The purpose of `dfs` is to traverse the tree and calculate the longest univalue path for each node recursively.
2. **Global Variable for the Answer:**
   - We use a nonlocal variable `ans` which is declared before the nested `dfs` function and is used to keep track of the longest path we have found during the recursion.
3. **Tracking the Current Node's Path:**
   - Inside `dfs`, we initially check if the current `root` node is `None`. If it is, the function returns `0`, as there is no path extending from a nonexistent node.
   - We then recursively call `dfs` for both the `left` and `right` child nodes of the current `root`.
4. **Path Extension Logic:**
   - We extend the path only if the child node has the same value as the current node. If `root.left` exists and its value is the same as `root.val`, we add `1` to the value returned by `dfs(root.left)`; otherwise, we reset it to `0`.
   - Similarly, we do the same for `root.right`.
5. **Updating the Longest Path (`ans`):**
   - At each step, after computing the value of left and right univalue paths, we update our global variable `ans` with the sum of left and right paths if this sum represents a new maximum. It's crucial to recognize that the longest univalue path that passes through the current node is the sum of the longest univalue paths going down through its left and right children (provided the values match).
   - This step corresponds to potentially joining two paths together at the current `root` node to form a longer path.
6. **Returning the Best Unidirectional Path:**
   - Since we cannot include both the left and right paths in our final answer (as that would constitute a split, not a single path), we return the maximum of the two, again incremented by `1` for the current node if applicable.
7. **End of Recursion and Result:**
   - The recursion bottoms out when we hit `None` nodes, and it builds the solution as it unwinds the recursive calls.
   - Finally, the `longestUnivaluePath` method returns the value of `ans`.

Note that by defining the `dfs` function within the `longestUnivaluePath` method, we are able to use the `nonlocal` keyword which allows us to modify the `ans` variable inside the nested function. This is an example of using closure in Python to maintain the state between function calls.

The key algorithmic pattern here is recursion in tandem with a global variable to keep track of the optimal solution. The code uses a straightforward recursive DFS strategy tailored to track a specific condition—the nodes along a path have the same value.

By breaking down the problem into a series of recursive steps, each handling a local part of the tree around a single node, we are able to build the overall solution globally by gathering and combining the local answers.

Example Walkthrough

Let us consider a small binary tree to illustrate the solution approach:

```
      4
     / \
    4   5
   / \   \
  4   4   5
```

Now, let's walk through the algorithm on this example:

1. We initiate the `longestUnivaluePath` with the root of the tree (value 4).
2. In our first call to the recursive `dfs` function, we examine the root having the value `4`.
3. We recursively call `dfs` on the left and right children, which will return the longest univalue paths with the same value as these child nodes. The left child (value 4) and right child (value 5) calls are made.

For the left child (value 4):

4. The `dfs` function is called with the left child, which is a node with value `4`.
5. Similarly, `dfs` is called on its left and right children. Both have the value `4`, so each child will contribute a value of `1` to the path length (after the `dfs` call).
6. No further calls to `dfs` are necessary down the leftmost branch as the children are `None`. The call for the left child of our current node (value 4) returns `1` as the maximum from either side.
7. The call for the right child (also value 4) will also return `1` because its children are `None`.
8. Both children had the same value as the current node, so the total path length for this node is the sum of both children plus the current node, which is `1 + 1 = 2`. This value `2` will be considered further up the tree.

For the right child (value 5):

9. The `dfs` function is called with the right child, which is a node with value `5`.
10. Its left child is `None`, and its right child has the same value, `5`, so it will contribute `1`.
11. No further `dfs` calls down this path are necessary because the right child of the right node is `None`. The returned path length is `1`.

Back to the root:

12. Now we return to the root node with value `4`. The left path provided a length of `2` (obtained from the left child calls), and the right path provided a length of `1` (obtained from the right child call).
13. We take the sum of both lengths since both children's values match with the root's value, so we get `2 + 1`. We update the global answer `ans` if `3` is greater than the current value of `ans`.
14. The `dfs` call at the root then needs to return the longest univalue path that includes the root node. So it chooses the maximum between the left and right path lengths, which is `2` from the left, and returns `2`.

After the root node's `dfs` function is processed, our answer (`ans`) holds the longest univalue path, which is `3`. This is the sum of the lengths from the left path and the right path passing through the root. Since we start counting from one node and end at another, considering the number of edges between the nodes, we have two edges, which represent the longest path, where all nodes share the same value.

The visualization of the longest univalue path is as follows:

```
      4
     /
    4
   /
  4
```

So, the function `longestUnivaluePath` finally returns the value `2` which is the length of the longest path with equal node values in the tree.

Python Solution

```python
class TreeNode:
    # Initializer for the TreeNode class
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def longestUnivaluePath(self, root: TreeNode) -> int:
        # This helper function performs depth-first search on the binary tree
        # to find the longest univalue path from the root node.
        def dfs(node):
            if node is None:
                return 0

            # Recursively find the longest path in the left and right subtree
            left_path_length = dfs(node.left)
            right_path_length = dfs(node.right)

            # If the current node's value matches that of its left child,
            # we can extend the univalue path by 1; otherwise, we reset to 0.
            left_path = left_path_length + 1 if node.left and node.left.val == node.val else 0

            # Similarly for the right child.
            right_path = right_path_length + 1 if node.right and node.right.val == node.val else 0

            # We use a nonlocal variable 'longest_path' to keep track of the
            # longest univalue path found during the DFS.
            nonlocal longest_path
            longest_path = max(longest_path, left_path + right_path)

            # Return the longest path from this node to its children that continues
            # the univalue path (only the longer one of left or right can be part of the path).
            return max(left_path, right_path)

        longest_path = 0
        dfs(root)
        return longest_path

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
class TreeNode {
    int value; // Node value
    TreeNode left; // Left child
    TreeNode right; // Right child

    // Default constructor
    TreeNode() {}

    // Constructor with value
    TreeNode(int value) { this.value = value; }

    // Constructor with value, left child, and right child
    TreeNode(int value, TreeNode left, TreeNode right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    private int longestPath; // To store the result of the longest univalue path

    public int longestUnivaluePath(TreeNode root) {
        longestPath = 0; // Initialize the longest path length to 0
        depthFirstSearch(root); // Begin DFS from the root
        return longestPath; // Return the maximum length found
    }

    // Helper method to perform DFS on each node
    private int depthFirstSearch(TreeNode node) {
        if (node == null) {
            // If the node is null, there is no path
            return 0;
        }
        // Recursively find the longest path in the left subtree
        int leftPathLength = depthFirstSearch(node.left);
        // Recursively find the longest path in the right subtree
        int rightPathLength = depthFirstSearch(node.right);

        // If the left child exists and has the same value, extend the left path
        leftPathLength = (node.left != null && node.left.value == node.value) ? leftPathLength + 1 : 0;
        // If the right child exists and has the same value, extend the right path
        rightPathLength = (node.right != null && node.right.value == node.value) ? rightPathLength + 1 : 0;

        // Update the longest path found so far considering both leftPath and rightPath
        longestPath = Math.max(longestPath, leftPathLength + rightPathLength);

        // The returned value should be the longest single side path (not combined) from this node
        return Math.max(leftPathLength, rightPathLength);
    }
}

```

C++ Solution

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

    // Constructor to initialize a node with no children.
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

    // Constructor to initialize a node with a value and no children.
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    // Constructor to initialize a node with a value and left and right children.
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    int longestPath; // Stores the length of the longest univalue path.

    // Public method to initiate the process and return the longest univalue path.
    int longestUnivaluePath(TreeNode* root) {
        longestPath = 0;
        dfs(root);
        return longestPath;
    }

private:
    // Helper DFS method to compute the univalue path length for each subtree.
    int dfs(TreeNode* node) {
        if (!node) return 0; // Base case: if the node is null, return 0.

        // Recursively find the longest univalue path in the left and right subtrees.
        int leftPathLength = dfs(node->left);
        int rightPathLength = dfs(node->right);

        // If the left child exists and has the same value, increment leftPathLength.
        leftPathLength = (node->left && node->left->val == node->val) ? leftPathLength + 1 : 0;

        // If the right child exists and has the same value, increment rightPathLength.
        rightPathLength = (node->right && node->right->val == node->val) ? rightPathLength + 1 : 0;

        // Update the longestPath with the maximum value of itself and the sum of left and right paths.
        longestPath = std::max(longestPath, leftPathLength + rightPathLength);

        // Return the longer path of the two: either leftPathLength or rightPathLength.
        return std::max(leftPathLength, rightPathLength);
    }
};

```

Typescript Solution

```typescript
// Define the structure for a binary tree node with numeric values
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Computes the length of the longest path where each node in the path has the same value.
 * This path may or may not pass through the root.
 *
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @return {number} The length of the longest univalue path.
 */
function longestUnivaluePath(root: TreeNode | null): number {
  // If the tree is empty, return zero as there are no paths.
  if (root === null) {
    return 0;
  }

  // Initialize the result variable to store the maximum path length found so far.
  let result = 0;

  /**
   * Performs depth-first search on the binary tree to find the longest univalue path.
   *
   * @param {TreeNode | null} node - The current node being explored.
   * @param {number} targetValue - The target value nodes in the path should have.
   * @return {number} The length of the longest univalue path through the current node.
   */
  function dfs(node: TreeNode | null, targetValue: number): number {
    // Base case: if the node is null, return zero as there is no path.
    if (node === null) {
      return 0;
    }

    // Recursively find the longest univalue path in the left subtree.
    let leftPathLength = dfs(node.left, node.val);
    // Recursively find the longest univalue path in the right subtree.
    let rightPathLength = dfs(node.right, node.val);

    // Update the result with the maximum path length by summing left and right paths if they are univalue.
    result = Math.max(result, leftPathLength + rightPathLength);

    // If the current node's value matches the target value,
    // return the longest path length including the current node;
    // otherwise, return zero as the path breaks here.
    if (node.val === targetValue) {
      return 1 + Math.max(leftPathLength, rightPathLength);
    }
    return 0;
  }

  // Start the depth-first search from the root.
  dfs(root, root.val);

  // The result contains the longest univalue path length.
  return result;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `longestUnivaluePath` function is `O(n)` where `n` is the number of nodes in the binary tree. This is because the `dfs` function is called once for every node in the tree, and each call to the `dfs` function processes the current node independently of the other nodes. Since the algorithm must visit all nodes to determine the longest univalue path, it results in a linear time complexity relative to the number of nodes.

Space Complexity

The space complexity of the `longestUnivaluePath` function is `O(h)` where `h` is the height of the tree. This is due to the recursive nature of the `dfs` function, which results in a call stack proportional to the height of the tree in the worst case (e.g., the tree is skewed). For a balanced tree, the height `h` would be `log(n)`, leading to a space complexity of `O(log(n))`. However, in the worst case (a skewed tree), the space complexity would be `O(n)`, where `n` is the total number of nodes.