1372. Longest ZigZag Path in a Binary Tree

Medium Tree Depth-First Search Dynamic Programming Binary Tree Leetcode Link

Problem Description

You are tasked with finding the longest ZigZag path in a binary tree. A ZigZag path is defined as follows:

- You first pick any node as a starting point and choose a direction to move in (either left or right).
- Based on the chosen direction, you move to the corresponding child node (move to the right child if the direction is right, or to the left child if the direction is left).
- After moving to a child node, you switch the direction for the next move (from right to left or left to right).
- You repeat the process of moving to the next child node and switching direction until there are no more nodes to visit.

The ZigZag length is the number of nodes you have visited on the path minus one, since a path with a single node has a length of zero. Your objective is to find the length of the longest ZigZag path in the given binary tree.

Intuition

To solve the problem, we employ depth-first search (DFS) to explore each possible path within the tree. The purpose of using DFS is to traverse the tree in a manner where we check every possible ZigZag path starting from each node, considering both the left and right directions.

Here is the intuition behind the DFS approach:

- We define a recursive DFS function that takes the current node and two integers, `l` and `r`, which represent the lengths of the ZigZag path when the last move was to the left (l) or to the right (r), respectively.
- At each call to the DFS function, if the current node is `None` (i.e., we can't move further), we return as we've reached the end of a path.
- We maintain a variable `ans` to keep track of the maximum length found so far. For each node, we update `ans` with the maximal value out of `ans`, `l`, and `r`.
- When we move to a left child (`root.left`), we increase the length of the ZigZag path for a move to the right (`r + 1`), and reset the left move length to zero. This is because moving to the left child is considered a right move for the parent node.
- Similarly, when moving to the right child (`root.right`), we increase the length of the ZigZag path for a move to the left (`l + 1`), and reset the right move length to zero.
- This recursive process continues until all nodes have been visited, and by then we will have recorded the maximum length of all ZigZag paths in the `ans` variable.

The solution uses a nonlocal declaration for the `ans` variable to ensure that its value is updated across different recursive calls. Each recursive call processes the current node and then makes two more recursive calls to the child nodes (if they exist), one for each possible direction (left or right) for the next move, continuing the ZigZag pattern.

Solution Approach

The solution uses a depth-first search (DFS) to explore each possible ZigZag path. Here's a breakdown of the implementation, referring to the Python code provided:

- We utilize the `TreeNode` class to represent the structure of the binary tree, where each node has a value (`val`), a pointer to the left child (`left`), and a pointer to the right child (`right`).

- The `Solution` class has the method `longestZigZag`, which initiates the recursive DFS process to find the longest ZigZag path. The method takes a `TreeNode` as an input, which is the `root` of the binary tree.

- Inside `longestZigZag`, we define a nested function `dfs` which recursively processes each node of the tree. The `dfs` function has parameters:
  - `root`: the current node being processed.
  - `l`: an integer representing the ZigZag path length when the last move was to the left.
  - `r`: an integer representing the ZigZag path length when the last move was to the right.
- The `dfs` function begins by checking if the current node is `None`. If it is, that branch of recursion stops because we can't move further down the tree from this node.

- To keep track of the maximum ZigZag path length found at any point, we use a variable `ans` which is declared as `nonlocal`. This ensures that `ans` can be updated across the different recursive calls and still maintains its most recent value.

- At each node, we update `ans` to be the maximum value between the current `ans` and the values of `l` and `r`. This ensures that as we move through the tree, any longer path lengths are captured and kept.

- When the DFS explores the left subtree by calling `dfs(root.left, r + 1, 0)`, it increments the ZigZag path length for a direction switch to the right (`r + 1`) and resets the left path length to zero (`0`). This is because a move to the left child is seen as a previous right move from the parent node's perspective.

- Conversely, when exploring the right subtree with `dfs(root.right, 0, l + 1)`, it increments the ZigZag path length for a direction switch to the left (`l + 1`) and resets the right path length to zero (`0`). A move to the right child is considered as a previous left move.

- The initial call to the `dfs` function is made with `dfs(root, 0, 0)` starting at the `root` with lengths of `0` for both left and right paths as no moves have been made yet.

- Once the DFS traversal is complete and all ZigZag paths have been computed, the longest path length is stored in `ans`, which is returned as the final result.

In summary, the solution effectively utilizes recursive DFS to explore each path, alternating moves and updating the path length while maintaining a global maximum. This is an example of a modified DFS where additional parameters (`l` and `r`) are used not only to keep track of the path length but also to encode the state of the ZigZag movement (direction change after each step). The complexity of the algorithm is O(n) where n is the number of nodes in the binary tree since every node is visited exactly once by the DFS traversal.

Example Walkthrough

To illustrate the solution approach, let us consider a small binary tree example:

```
    1
   / \
  3   2
 /   / \
4   5   6
```

We would like to find the longest ZigZag path in this binary tree.

Step 1: We start with the root of the tree which is the node with the value `1`. A recursive call `dfs(root, 0, 0)` is made with both `l` and `r` initialized to zero since no moves have been made yet.

Step 2: At the root node `1`, since it is not `None`, we have two options to continue the ZigZag path: move to the left child or right child. We make two recursive calls to explore both options:

- `dfs(root.left, r + 1, 0)` which is `dfs(node 3, 1, 0)`
- `dfs(root.right, 0, l + 1)` which is `dfs(node 2, 0, 1)`

In each of these calls, we are ZigZagging from the root now, so the appropriate path lengths `l` and `r` are updated.

Step 3: Now in the subtree rooted at `3`, we have `l = 1, r = 0`. Since node `3` has a left child `4` and no right child, we make another recursive call `dfs(node 3.left, r + 1, 0)` which becomes `dfs(node 4, 1, 0)`. At this point, since node `4` is a leaf node without children, the path ends here, and `ans` is potentially updated to the maximum of `ans`, `l`, or `r` which in this case would be `1`.

Similarly, in the subtree rooted at `2`, we have `l = 0, r = 1`. Node `2` has both a left and a right child, so two recursive calls are made:

- `dfs(node 2.left, r + 1, 0)` which is `dfs(node 5, 2, 0)`
- `dfs(node 2.right, 0, l + 1)` which is `dfs(node 6, 0, 1)`

At each leaf node (`5` and `6`), the same check for `None` occurs, and the `ans` is updated with the path lengths of `2` and `1`, respectively.

Step 4: Once all leaves have been reached, and `None` returned for each childless call, the recursion unwinds. The `ans` variable that was kept up to date in each step now contains the value `2`, which is the longest ZigZag path length in the binary tree (from `1` to `2` to `5`).

The longest ZigZag path in this binary tree is therefore of length `2`, which corresponds to two moves: one move from root (`1`) to the right child (`2`), and another move from the right child (`2`) to its left child (`5`). This illustrates how the DFS algorithm zigzags through the tree to find the longest possible path.

Python Solution

```python
class Solution:
    def longestZigZag(self, root: TreeNode) -> int:
        # Helper function to perform DFS (Depth-First Search) on the tree
        def dfs(node, left_length, right_length):
            # If the node is None, we've reached the end of a path
            if node is None:
                return
            # Update the global answer by taking the maximum value found so far
            nonlocal max_length
            max_length = max(max_length, left_length, right_length)
            # Recursively explore the left child, incrementing the "left_length" as we are
            # making a zig (left direction) from the right side of the current node
            dfs(node.left, right_length + 1, 0)
            # Recursively explore the right child, incrementing the "right_length" as we are
            # making a zag (right direction) from the left side of the current node
            dfs(node.right, 0, left_length + 1)

        # Initialize the maximum length to 0 before starting DFS
        max_length = 0
        # Start DFS with the root of the tree, initial lengths are 0 as starting point
        dfs(root, 0, 0)
        # Once DFS is complete, return the maximum zigzag length found
        return max_length
```

Java Solution

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

/**
 * Solution class contains methods to calculate the longest zigzag path in a binary tree.
 */
class Solution {
    // Variable to store the length of the longest zigzag path found so far.
    private int longestZigZagLength;

    /**
     * Method to find the length of the longest zigzag path starting from the root node.
     *
     * @param root The root node of the binary tree.
     * @return The length of the longest zigzag path in the tree.
     */
    public int longestZigZag(TreeNode root) {
        // Initialize with the assumption that there's no zigzag path.
        longestZigZagLength = 0;

        // Start Depth First Search traversal from the root.
        dfs(root, 0, 0);

        // Return the length of the longest zigzag path found.
        return longestZigZagLength;
    }

    /**
     * A recursive DFS method that traverses the tree to find the longest zigzag path.
     *
     * @param node        The current node being visited.
     * @param leftLength  The current length of the zigzag path when coming from a left child.
     * @param rightLength The current length of the zigzag path when coming from a right child.
     */
    private void dfs(TreeNode node, int leftLength, int rightLength) {
        // If the node is null, we have reached beyond leaf, so return.
        if (node == null) {
            return;
        }

        // Update the longestZigZagLength with the maximum path length seen so far at this node.
        longestZigZagLength = Math.max(longestZigZagLength, Math.max(leftLength, rightLength));

        // Traverse left - the zigzag is coming from the right so add 1 to rightLength.
        dfs(node.left, rightLength + 1, 0);

        // Traverse right - the zigzag is coming from the left so add 1 to leftLength.
        dfs(node.right, 0, leftLength + 1);
    }
}
```

C++ Solution

```cpp
#include <algorithm> // For using max()

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

    // Constructors
    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 longestPath = 0; // Variable to store the longest ZigZag path length found

    // Function to calculate the length of the longest ZigZag path in a binary tree.
    int longestZigZag(TreeNode* root) {
        traverse(root, 0, 0); // Start the traversal from the root node
        return longestPath;   // Return the longest ZigZag path length found
    }

    // Helper function to perform DFS and find the length of the longest ZigZag path
    void traverse(TreeNode* node, int leftSteps, int rightSteps) {
        if (!node) return; // If we reach a null node, stop the traversal

        // Update the maximum length of ZigZag path seen so far
        longestPath = std::max(longestPath, std::max(leftSteps, rightSteps));

        // If we take a left child, we can move right the next step; thus, increment rightSteps.
        // If we take a right child, we can move left the next step; thus, increment leftSteps.
        // Pass 0 for the opposite direction because ZigZag path resets on turning twice in the same direction.
        if (node->left) traverse(node->left, rightSteps + 1, 0);       // Traverse the left child
        if (node->right) traverse(node->right, 0, leftSteps + 1);      // Traverse the right child
    }
};
```

Typescript Solution

```typescript
// Interface for a binary tree node
interface TreeNode {
  val: number;        // Value of the node
  left: TreeNode | null;   // Reference to the left child
  right: TreeNode | null;  // Reference to the right child
}

// Variable to store the longest ZigZag path length found
let longestPath: number = 0;

// Function to calculate the length of the longest ZigZag path in a binary tree
function longestZigZag(root: TreeNode | null): number {
  traverse(root, 0, 0); // Start the traversal from the root node
  return longestPath;   // Return the longest ZigZag path length found
}

// Helper function to perform DFS and find the length of the longest ZigZag path
function traverse(node: TreeNode | null, leftSteps: number, rightSteps: number) {
  if (!node) return; // If we reach a null node, stop the traversal

  // Update the maximum length of the ZigZag path seen so far
  longestPath = Math.max(longestPath, leftSteps, rightSteps);

  // If we take a left child, we can move right on the next step; thus, increment rightSteps.
  // If we take a right child, we can move left on the next step; thus, increment leftSteps.
  // Pass 0 for the opposite direction because the ZigZag path resets on turning twice in the same direction.
  if (node.left) traverse(node.left, rightSteps + 1, 0); // Traverse the left child
  if (node.right) traverse(node.right, 0, leftSteps + 1); // Traverse the right child
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)` where `n` is the number of nodes in the binary tree. The reason for this is that the function `dfs` visits every node in the tree exactly once. During each call, it performs a constant amount of work, regardless of the size of the subtree it is working with.

Space Complexity

The space complexity of the code is `O(h)`, where `h` is the height of the binary tree. This space is used by the call stack due to recursion. In the worst case, when the binary tree becomes a skewed tree (like a linked list), the height `h` can be equal to `n`, which means the space complexity in the worst case would be `O(n)`. However, for a balanced tree, the height `h` will be `log(n)`, resulting in a space complexity of `O(log(n))`.