2313. Minimum Flips in Binary Tree to Get Result

Hard | Tree | Depth-First Search | Dynamic Programming | Binary Tree | Leetcode Link

Problem Description

In this problem, we're given the root of a binary tree that represents a boolean expression. The leaves of this tree have values of either `0` or `1`, which stand for the boolean values `false` and `true`, respectively. The non-leaf nodes can have one of four values: `2` (OR operation), `3` (AND operation), `4` (XOR operation), or `5` (NOT operation). The goal is to perform a minimum number of flips on the leaf nodes, where a flip changes a `0` to a `1` or a `1` to a `0`, so that when the expression represented by the whole tree is evaluated, it yields the boolean value given by the variable `result`.

It's interesting to note that for the NOT operation, which is represented by the value `5`, the tree node will have only one child. For all other operations, there will be two children.

The problem assures us that it is always possible to achieve the desired `result` through a series of flips. However, the challenge is to find the minimum number of flips needed to do so.

Intuition

To solve the problem, we must carefully navigate the tree and consider the cost (in terms of flips) of getting to the desired `result`. We'll need to perform a depth-first traversal of the tree. For each node, we need to consider two scenarios:

- The number of flips required if the current node's evaluation needs to be `true`
- The number of flips required if the current node's evaluation needs to be `false`

Because we are dealing with a boolean operation, we must consider the operation represented by each non-leaf node when computing the number of flips.

Here's the intuition behind the solution:

- If we encounter a leaf node, the cost to make it `true` or `false` is straightforward: if the leaf node's current value is `0`, it would take 0 flips to make it `false` and 1 flip to make it `true`, and vice versa for a leaf with a value of `1`.

- For non-leaf nodes, the situation is more complex, and we need to consider the operation represented by the node and the costs associated with flipping each of its children to make it either `true` or `false`.

The core of the solution is a recursive depth-first search that, for each node, returns a tuple `(cost_to_true, cost_to_false)` where:

- `cost_to_true` is the minimum number of flips to make the current subtree evaluate to `true`.
- `cost_to_false` is the minimum number of flips to make the current subtree evaluate to `false`.

Depending on the node's operation, we combine the costs of its children according to the boolean logic of OR, AND, XOR, and NOT.

Finally, the answer will be the cost associated with the root node to make its evaluation equal to the `result`. In the provided solution, this is implemented as `dfs(root)[int(result)]`.

Solution Approach

The solution implements a recursive approach to traverse the tree and calculate the minimum number of flips required to achieve the desired result for each subtree. The algorithm uses Depth-First Search (DFS) which is a common tree traversal technique.

Here's how we implement the solution:

- We define a recursive function called `dfs` which takes a node as input and returns a tuple `(cost_to_true, cost_to_false)`. This tuple represents the minimum number of flips required to make the subtree rooted at the given node evaluate to `true` and `false`, respectively.

- For a leaf node with value `0` or `1`, the cost to make it `true` is its value (since `0` means `false` and `1` means `true`), and the cost to make it `false` is its complement (which is calculated by `x ^ 1`, where `x` is the value of the node).

- For a non-leaf node, we recursively get the costs for its left and right children. The combined cost depends on the operation the non-leaf node represents:
    - OR (`2`): The minimum number of flips required to make the result `true` is simply the sum of flips for both children to be `true`. To make the result `false`, we need to have at least one child as `false`, so we take the minimum of either child being `false` or both being `false`.
    - AND (`3`): The flips to make the result `true` are the minimum required to get one or both children `true`. To make the result `false`, both children must be `false`, so we add both children's `cost_to_false`.
    - XOR (`4`): To maintain the `true` result, either one child should be `true` and the other `false`, so we take the two minimums accordingly. To make the result `false`, both children need to have the same evaluation, so we consider either both `true` or both `false`.
    - NOT (`5`): This operation flips the outcome, so we swap the costs for `true` and `false` of the single child node.
- After calculating the costs for a node's children, we take the appropriate combination of costs as per the node's operation. Importantly, the solution ensures that results are mathematically minimized, taking advantage of the boolean nature of the problem to calculate the minimum number of steps.

- The base case for the recursion is when we encounter a `None` type for a child node; we return `(inf, inf)` since we don't perform operations on non-existent children.

- The function `dfs(root)[int(result)]` is used to extract the cost corresponding to the desired boolean `result`.

The solution leverages tuple unpacking, bitwise operations, and the recursive structure of the DFS algorithm, which makes it efficient and covers all edge cases due to its minimization logic at each step of the recursion.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the following binary tree representing a boolean expression:

```
        2
       / \
      /   \
     5     3
    /     / \
   1     1   0
```

In this tree:

- The root node represents an OR operation (`2`).
- The left child of the root is a NOT operation (`5`), with a leaf child of `1` (true).
- The right child of the root is an AND operation (`3`), with two leaf children `1` (true) and `0` (false).

Let's say we want the final evaluated result of the tree to be `true` (`1`).

Here's how we walk through the solution approach:

1. At the root (OR operation), we need to evaluate both the left and right subtrees.
2. The left subtree is a NOT operation with a child leaf node of `1`:
    - To make the NOT node `true`, since it's currently true, it would need to be flipped to false first, then NOT applied. That's 1 flip.
    - To make it `false`, we do nothing since NOT `1` is `false`. That's 0 flips.
3. The right subtree is an AND operation with two leaf children `1` and `0`:
    - To make the AND node `true`, we need both children to be `true`. In this case, we need 1 flip for the child `0` to become `1`.
    - To make it `false`, we can choose either of the children to be `false`, which in this case requires 0 flips since the right child is already `0`.
4. Now, back at the root OR node, to make it `true`:
    - We can make either the left or the right subtree `true`. As per the calculated costs, making the left subtree `true` costs 1 flip, and making the right subtree `true` costs 1 flip as well. So, we would choose the minimum, which is 1 flip in this case.
    - We don't need to calculate the cost to make it `false` since the desired result is `true`.
5. The minimum number of flips to get to the result is hence `1` which corresponds to flipping the leaf node from `0` to `1` in the right AND subtree.

By following the approach outlined in the solution, we recursively calculate the minimum number of flips and choose the optimal path at each step to achieve the desired boolean value with the least number of flips.

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 minimumFlips(self, root: Optional[TreeNode], result: bool) -> int:
        # Helper function to perform a depth-first search on the binary tree.
        # This function will return a tuple with the minimum number of flips required
        # to get the current subtree to evaluate to 0 and 1, respectively.
        def dfs(node: Optional[TreeNode]) -> (int, int):
            if node is None:
                return float('inf'), float('inf')  # No flips can be done on a non-existent node

            # If node value is 0 or 1, no need to flip to obtain the same value,
            # but one flip is needed to change it to the opposite value.
            if node.val in (0, 1):
                return node.val, node.val ^ 1

            # Recurse on the left and right subtrees.
            left_flips, right_flips = dfs(node.left), dfs(node.right)

            # Calculate the minimum number of flips depending on the operation
            # represented by the current node's value (AND, OR, XOR, NAND).
            if node.val == 2:  # AND
                return (left_flips[0] + right_flips[0],  # 0 AND 0
                        min(left_flips[0] + right_flips[1], left_flips[1] + right_flips[0], left_flips[1] + right_flips[1]))  # 1 A
            if node.val == 3:  # OR
                return (min(left_flips[0] + right_flips[0], left_flips[0] + right_flips[1], left_flips[1] + right_flips[0]),  # 0 O
                        left_flips[1] + right_flips[1])  # 1 OR 1
            if node.val == 4:  # XOR
                return (min(left_flips[0] + right_flips[0], left_flips[1] + right_flips[1]),  # 0 XOR 0
                        min(left_flips[0] + right_flips[1], left_flips[1] + right_flips[0]))  # 1 XOR 1
            # NAND operation (not standard, but assuming it's the opposite of AND)
            return (min(left_flips[1], right_flips[1]),   # 0 NAND 0
                    min(left_flips[0], right_flips[0]))   # 1 NAND 1

        # Call the dfs helper function starting with the root of the tree.
        # Convert the boolean result to an integer (0 or 1) to get the corresponding flips.
        return dfs(root)[int(result)]

# Note: It assumes that Optional[TreeNode] and float('inf') are already valid as per the Python 3 syntax.
```

Java Solution

```java
class Solution {
    // Function to calculate the minimum flips to make the tree evaluate to the desired result
    public int minimumFlips(TreeNode root, boolean result) {
        // Call the DFS traversal and return the flips needed for the desired result (false: 0, true: 1)
        return dfs(root)[result ? 1 : 0];
    }

    // DFS traversal to calculate minimum flips at each subtree
    private int[] dfs(TreeNode node) {
        // Base case: if the node is null, we return large numbers because there is nothing to flip
        if (node == null) {
            return new int[] {Integer.MAX_VALUE / 2, Integer.MAX_VALUE / 2};
        }

        // Leaf node condition, where value directly determines the number of flips (0 for no change, 1 for a flip)
        int value = node.val;
        if (value < 2) {
            return new int[] {value, value ^ 1};
        }

        // Recurse on the left and right subtrees
        int[] leftFlips = dfs(node.left);
        int[] rightFlips = dfs(node.right);

        // Initialize the minimum flips for making the current subtree true or false
        int minFlipsFalse, minFlipsTrue;

        // Check the type of operator at the current node and calculate flips accordingly
        if (value == 2) { // AND operator
            minFlipsFalse = leftFlips[0] + rightFlips[0]; // Both subtrees should be false
            minFlipsTrue = Math.min(Math.min(leftFlips[1] + rightFlips[0], leftFlips[0] + rightFlips[1]), leftFlips[1] + rightFlips
        } else if (value == 3) { // OR operator
            minFlipsFalse = Math.min(Math.min(leftFlips[0] + rightFlips[1], leftFlips[1] + rightFlips[0]), leftFlips[0] + rightFlip
            minFlipsTrue = leftFlips[1] + rightFlips[1]; // Both subtrees should be true
        } else if (value == 4) { // XOR operator
            minFlipsFalse = Math.min(leftFlips[0] + rightFlips[0], leftFlips[1] + rightFlips[1]); // Subtrees should have same trut
            minFlipsTrue = Math.min(leftFlips[0] + rightFlips[1], leftFlips[1] + rightFlips[0]); // Subtrees should have opposite t
        } else { // NOT operator (Assuming it's a unary operator on left subtree only)
            minFlipsFalse = Math.min(leftFlips[1], rightFlips[1]); // The subtree should be true to propagate false upwards
            minFlipsTrue = Math.min(leftFlips[0], rightFlips[0]); // The subtree should be false to propagate true upwards
        }

        // Return the minimum flips required to make the subtree true and false
        return new int[] {minFlipsFalse, minFlipsTrue};
    }
}
```

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 minimumFlips(TreeNode* root, bool result) {
        // Utility lambda function to perform a depth-first search on the tree.
        // It returns a pair of integers representing the minimum flips needed to
        // make the subtree rooted at 'node' evaluate to false (first) and true (second).
        function<pair<int, int>(TreeNode*)> dfs = [&](TreeNode* node) -> pair<int, int> {
            // Base case: if the node is null, return maximum possible values
            // since we can't flip a null node.
            if (!node) {
                return {INT_MAX, INT_MAX};
            }
            int value = node->val;
            // If the value of the node is 0 or 1, it's a leaf node and we return
            // the cost to flip it to 0 (first) and to 1 (second), which is just one flip.
            if (value < 2) {
                return {value, value ^ 1};
            }
            // Recursively get the minimum flips for left and right subtrees.
            auto [leftFlipsFalse, leftFlipsTrue] = dfs(node->left);
            auto [rightFlipsFalse, rightFlipsTrue] = dfs(node->right);
            int flipsToFalse = 0, flipsToTrue = 0;

            // Based on the value of the node (operation type), combine the results from left
            // and right subtrees to determine the minimum flips for current subtree.
            if (value == 2) { // AND operation
                flipsToFalse = leftFlipsFalse + rightFlipsFalse;
                flipsToTrue = min({leftFlipsFalse + rightFlipsTrue, leftFlipsTrue + rightFlipsFalse, leftFlipsTrue + rightFlipsTrue
            } else if (value == 3) { // OR operation
                flipsToFalse = min({leftFlipsFalse + rightFlipsFalse, leftFlipsFalse + rightFlipsTrue, leftFlipsTrue + rightFlipsFa
                flipsToTrue = leftFlipsTrue + rightFlipsTrue;
            } else if (value == 4) { // XOR operation
                flipsToFalse = min(leftFlipsFalse + rightFlipsFalse, leftFlipsTrue + rightFlipsTrue);
                flipsToTrue = min(leftFlipsFalse + rightFlipsTrue, leftFlipsTrue + rightFlipsFalse);
            } else { // NOT operation (we assume that 'value' is either 0, 1, 2, 3, 4, or 5)
                flipsToFalse = min(leftFlipsTrue, rightFlipsTrue);
                flipsToTrue = min(leftFlipsFalse, rightFlipsFalse);
            }
            // Return the pair of flips for false and true.
            return {flipsToFalse, flipsToTrue};
        };

        // Call the DFS function and get the minimum flips needed for false and true.
        auto [flipsForFalse, flipsForTrue] = dfs(root);
        // Return the corresponding flips based on the 'result' parameter.
        return result ? flipsForTrue : flipsForFalse;
    }
};
```

Typescript Solution

```typescript
// The TreeNode class represents each node in the binary tree.
class TreeNode {
    val: number
    left: TreeNode | null
    right: TreeNode | null
    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = (val === undefined ? 0 : val)
        this.left = (left === undefined ? null : left)
        this.right = (right === undefined ? null : right)
    }
}

// The minimumFlips function calculates the minimum number of flips required
// to make the output of the binary tree equal to the result boolean.
// @param root - The root of the binary tree.
// @param result - The expected result of the binary tree after performing flips.
// @returns - the minimum number of flips needed.
function minimumFlips(root: TreeNode | null, result: boolean): number {
    // The dfs (Depth-First Search) function computes two values for each node:
    // The minimum flips required to make the value of the subtree rooted at this node 0 and 1, respectively.
    const dfs = (node: TreeNode | null): [number, number] => {
        if (!node) {
            // If the node is null, return maximum values as it contributes
            // nothing to the count of flips.
            return [Number.MAX_SAFE_INTEGER, Number.MAX_SAFE_INTEGER];
        }

        const value = node.val;

        // If the node is a leaf node or holds a value of 0 or 1, return the values
        // itself and flip required to change it (0 becomes 1 and 1 becomes 0).
        if (value < 2) {
            return [value, value ^ 1];
        }

        // Otherwise, compute the flips for left and right subtrees.
        const [leftFlipsToZero, leftFlipsToOne] = dfs(node.left);
        const [rightFlipsToZero, rightFlipsToOne] = dfs(node.right);

        // Calculate the flips based on the logical operation represented by the current node.
        if (value === 2) {
            // Logical AND
            return [leftFlipsToZero + rightFlipsToZero, Math.min(leftFlipsToZero + rightFlipsToOne, leftFlipsToOne + rightFlipsToZe
        }

        if (value === 3) {
            // Logical OR
            return [Math.min(leftFlipsToZero + rightFlipsToZero, leftFlipsToZero + rightFlipsToOne, leftFlipsToOne + rightFlipsToZe
        }

        if (value === 4) {
            // Logical XOR
            return [Math.min(leftFlipsToZero + rightFlipsToZero, leftFlipsToOne + rightFlipsToOne), Math.min(leftFlipsToZero + righ
        }

        // Logical NAND
        return [Math.min(leftFlipsToOne, rightFlipsToOne), Math.min(leftFlipsToZero, rightFlipsToZero)];
    };

    // Initial call to dfs function starting at the root,
    // and return the required minimum flips based on the desired result.
    return dfs(root)[result ? 1 : 0];
}
```

Time and Space Complexity

The given code implements an algorithm to calculate the minimum number of flips necessary in a binary tree such that the value of the root node is equal to the desired result.

Time Complexity:

The time complexity of the algorithm is `O(N)`, where `N` is the number of nodes in the binary tree. This is because we visit each node exactly once in a depth-first search (DFS) manner. For each node, we perform a constant-time operation regardless of its value, which either directly returns the values for leaves or calculates the minimum based on the values returned from the children for internal nodes.

Space Complexity:

The space complexity of the algorithm can be analyzed by considering the call stack for the recursive function `dfs`. In the worst-case scenario, the recursion depth is equal to the height of the tree. For a balanced binary tree, the height is `O(log N)`, which would make the space complexity `O(log N)` due to the call stack. However, for a completely unbalanced tree (e.g., a linked list), the recursion could be as deep as `O(N)`. Thus, the space complexity of the algorithm is `O(H)`, where `H` is the height of the tree, and in the worst case, it is `O(N)`.