1145. Binary Tree Coloring Game

Medium Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

The problem presents a game played on a binary tree by two players. The game is turn-based, and each player gets to color nodes on the tree. The first player colors a node red and the second player colors a different node blue. Then, each player, on their turn, must color an uncolored neighboring node of their color. This game progresses until neither player can make a move. The winner is determined by who has colored more nodes. The twist here is that the binary tree's number of nodes (`n`) is odd, ensuring there can't be a tie, and all nodes have unique values from `1` to `n`.

As a player, your task is to determine if there exists a move (choosing `y`) that guarantees a win regardless of how the game progresses (assuming optimal moves from the opponent). Since the binary tree structure can significantly affect gameplay, the strategy is not straightforward. The problem requires you to think in terms of the binary tree's structure and subsets of nodes that can be controlled by each player.

Intuition

The intuition behind the solution is to consider the possible regions the second player can take over, ensuring that the number of nodes colored blue will be greater than those colored red. Since player two gets to pick any node to color blue, the strategy involves picking a node that maximizes player two's advantage.

The binary tree can be divided into three regions from the perspective of node `x` (the one first player colors red):

1. The left subtree of node `x`.
2. The right subtree of node `x`.
3. The remaining part of the tree above node `x`.

The second player will win if they can control more than half of the nodes in the tree by the end of the game. To guarantee a win, the second player should color a node that lets them take control over the largest region possible. Since the regions are distinct and cover the entire tree, player two should look to find out the size of these regions and color a node in the region with the most nodes.

Here's the approach in steps:

- Find the node `x` in the tree using a Depth-First Search (DFS).
- Compute the size of the left and right subtrees of node `x` (denoted `l` and `r`) using another recursive count function.
- The size of the third region can be determined by subtracting the sizes of left and right subtrees and node `x` itself from the total number of nodes (`n - l - r - 1`).
- Check if any of the three regions has more than half of the total nodes by comparing each of their sizes to `n // 2`.
- The second player should start by coloring a node in the largest region to ensure more than half of the nodes can be colored blue.

From a strategic standpoint, player two gains an advantage because they can freely choose their starting node in response to player one's choice, thereby selecting the region that gives them the best chance of winning.

Solution Approach

The solution to the problem uses a recursive depth-first search (DFS) algorithm to explore the binary tree, which is a typical approach when dealing with tree data structures. Here's how the solution is structured:

1. The `dfs` function is used to locate the node with the value `x`. It searches through the tree recursively until it finds the node with the given value or reaches a leaf. The search starts from the root of the tree and proceeds to the left and right children respectively.

2. Once the node `x` is found, the `count` function recursively computes the number of nodes in a subtree rooted at the given node. It sums up the count from the left child, the right child, and the current node by returning `1 + count(root.left) + count(root.right)`.

3. With both `l` and `r` obtained, which are the sizes of the left and right subtrees of node `x`, we can deduce the size of the third region by subtracting `l + r + 1` (the `+1` is for the node `x` itself) from the total number of nodes `n`. This gives the number of nodes in the rest of the tree, not part of `x`'s left or right subtrees.

4. The winning condition for the second player is to control more than half of the nodes. To satisfy this, we compare each region's size (left subtree size `l`, right subtree size `r`, and the third region `n - l - r - 1`) to half of the total nodes `n // 2`. If any of these regions contain more nodes than `n // 2`, the second player can choose their initial node `y` in that region and guarantee a victory.

5. Finally, the solution returns `True` if at least one of the regions has more nodes than `n // 2` and `False` otherwise. This is done using the `max` function to find the largest region size and comparing it to `n // 2` with `max(l, r, n - l - r - 1) > n // 2`.

In summary, the solution leverages tree traversal to find crucial information about the structure of the tree and then applies a comparison to determine if a winning move is possible. The efficiency of this approach stems from minimizing the number of nodes visited by only traversing the necessary parts of the tree and using simple arithmetic to assess the winning condition.

Example Walkthrough

Let's consider a small binary tree for an example to illustrate the solution approach. Suppose that our binary tree looks like this, with the unique values of the nodes labeled from `1` to `7` (since `n` is odd), and Player One starts by coloring node `4` red:

```
       4
     /   \
    2     6
   / \   / \
  1   3 5   7
```

Following the solution steps:

1. Player One colors node `4` red. Now we need to start a DFS to find this node `x` (which is already known in this scenario, but we'd DFS if we were only given a value).

2. We find node `4`, and now we compute the size of its left and right subtrees using a `count` function.

   - The left subtree of node `4` (`l`) contains nodes `2`, `1`, and `3`. So `l = 3`.
   - The right subtree of node `4` (`r`) contains nodes `6`, `5`, and `7`. So `r = 3`.

3. Then, we calculate the size of the third region, which is the size of the tree not including node `4` and its subtrees. Since there are `7` nodes in total (`n`), the third region size is `n - l - r - 1`, which equals `7 - 3 - 3 - 1 = 0`.

4. Next, we check if any of the regions (`l`, `r`, and the third region) have more than half of the total nodes (`n // 2`). Since `n` is `7`, `n // 2` is `3`.

   - Left subtree size `l`: `3` nodes
   - Right subtree size `r`: `3` nodes
   - The rest of the tree: `0` nodes

5. None of the sizes is strictly greater than `3` (`n // 2`), so in this particular example, there isn't a guaranteed winning move for Player Two. In this case, the output would be `False` because Player Two cannot color a starting node `y` that guarantees them a region with more than `3` nodes.

To conclude, applying the steps to our example, we see that neither the left nor the right subtree or the third region provides a guaranteed win for Player Two since all have fewer than or exactly `n // 2` nodes. Hence, there is no starting node `y` in this binary tree configuration that ensures Player Two's victory assuming optimal play by both players.

Python Solution

```python
class Solution:
    def btreeGameWinningMove(self, root: Optional[TreeNode], n: int, x: int) -> bool:
        # Helper function to find the node with value x.
        def find_node(current_root):
            # Return None if we reach a leaf node.
            if current_root is None:
                return None
            # Return the current node if its value matches x.
            if current_root.val == x:
                return current_root
            # Recursively search in the left and right subtrees.
            return find_node(current_root.left) or find_node(current_root.right)

        # Helper function to count the number of nodes in a subtree.
        def count_nodes(current_root):
            # Base case: if current node is None, return count as 0.
            if current_root is None:
                return 0
            # Count the current node and recursively count the nodes in the subtrees.
            return 1 + count_nodes(current_root.left) + count_nodes(current_root.right)

        # Find the node with value x.
        target_node = find_node(root)
        # Count the nodes in the left and right subtree of the target node.
        left_count = count_nodes(target_node.left)
        right_count = count_nodes(target_node.right)

        # Choose the maximum count from the left or right subtree of the target node,
        # or the rest of the tree excluding the target node's subtree.
        # Player two wins if their chosen part has more nodes than half of the total nodes.
        return max(left_count, right_count, n - left_count - right_count - 1) > n // 2

```

Java Solution

```java
class Solution {
    // Method to determine if a winning move is possible in a binary tree game.
    public boolean btreeGameWinningMove(TreeNode root, int n, int x) {
        // Find the node with value 'x'.
        TreeNode xNode = findNode(root, x);
        // Count the nodes in the left subtree of node 'x'.
        int leftSubtreeCount = countNodes(xNode.left);
        // Count the nodes in the right subtree of node 'x'.
        int rightSubtreeCount = countNodes(xNode.right);
        // The parent's subtree size is the total nodes minus the ones in the subtree of 'x'.
        int parentSubtreeCount = n - leftSubtreeCount - rightSubtreeCount - 1;
        // Check if any subtree has more than half of the nodes. If so, it's a winning move.
        return Math.max(Math.max(leftSubtreeCount, rightSubtreeCount), parentSubtreeCount) > n / 2;
    }

    // Helper method to find the node with value 'x' in the tree.
    private TreeNode findNode(TreeNode root, int x) {
        if (root == null || root.val == x) {
            return root;
        }
        // Search the left subtree.
        TreeNode leftFind = findNode(root.left, x);
        // If not found in the left subtree, search the right subtree.
        return leftFind != null ? leftFind : findNode(root.right, x);
    }

    // Helper method to count the number of nodes in the given subtree.
    private int countNodes(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // Count the current node plus the nodes in the left and right subtrees.
        return 1 + countNodes(root.left) + countNodes(root.right);
    }
}

```

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:
    // Determines if there is a winning move in the binary tree game
    bool btreeGameWinningMove(TreeNode* root, int nodeCount, int target) {
        // Find the target node in the tree
        TreeNode* targetNode = findTargetNode(root, target);
        // Count the nodes in the left and right subtrees of the target node
        int leftCount = countNodes(targetNode->left);
        int rightCount = countNodes(targetNode->right);
        // The parent partition's node count is the remainder when subtracting from the total
        int parentPartitionCount = nodeCount - leftCount - rightCount - 1;
        // A winning move is possible if any partition has more than half of the total nodes
        return max({leftCount, rightCount, parentPartitionCount}) > nodeCount / 2;
    }

private:
    // Helper function to find the node with value x using DFS
    TreeNode* findTargetNode(TreeNode* currentNode, int targetValue) {
        // If the current node is nullptr or the target node, return it
        if (!currentNode || currentNode->val == targetValue) {
            return currentNode;
        }
        // Search in the left subtree
        TreeNode* leftResult = findTargetNode(currentNode->left, targetValue);
        // If found in the left subtree, return it; otherwise, search in the right subtree
        return leftResult ? leftResult : findTargetNode(currentNode->right, targetValue);
    }

    // Helper function to count the nodes in the given subtree
    int countNodes(TreeNode* currentNode) {
        // If the current node is nullptr, return 0, as there are no nodes to count
        if (!currentNode) {
            return 0;
        }
        // Count the current node, then recursively count left and right subtrees
        return 1 + countNodes(currentNode->left) + countNodes(currentNode->right);
    }
};

```

Typescript Solution

```typescript
// TypeScript definition for a binary tree node.
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Determine if there is a winning move in a binary tree game.
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @param {number} n - The total number of nodes in the tree.
 * @param {number} x - The value of the node the first player chooses.
 * @returns {boolean} - True if there is a winning move, false otherwise.
 */
function btreeGameWinningMove(root: TreeNode | null, n: number, x: number): boolean {
    // Recursive function to find the node with value x.
    function findNodeWithValueX(node: TreeNode | null): TreeNode | null {
        if (!node || node.val === x) {
            return node;
        }

        // Traverse left and then right in search for the node.
        return findNodeWithValueX(node.left) || findNodeWithValueX(node.right);
    }

    // Function to count the nodes in the subtree rooted at the given node.
    function countNodes(node: TreeNode | null): number {
        if (!node) {
            return 0;
        }

        // Count 1 for the current node and then count the nodes in left and right subtree.
        return 1 + countNodes(node.left) + countNodes(node.right);
    }

    // Find the node with value x.
    const xNode = findNodeWithValueX(root);

    // Count the nodes in the left and right subtree of the node with value x.
    const leftSubtreeNodeCount = countNodes(xNode.left);
    const rightSubtreeNodeCount = countNodes(xNode.right);

    // The remaining nodes are those not in x's left subtree, right subtree, nor x itself.
    const remainingNodes = n - leftSubtreeNodeCount - rightSubtreeNodeCount - 1;

    // The second player wins if they can choose a subtree larger than n / 2 nodes after first move.
    return Math.max(leftSubtreeNodeCount, rightSubtreeNodeCount, remainingNodes) > n / 2;
}

```

Time and Space Complexity

Time Complexity

The provided code consists of two main functions: `dfs` and `count`.

1. `dfs` function: This is a pre-order traversal to find the node with value `x`. In the worst-case scenario, it visits each node once, resulting in a time complexity of `O(n)`, where `n` is the total number of nodes in the tree.

2. `count` function: This function is called for both the left and right subtrees of the node with value `x` to count the number of nodes therein, resulting in a worst-case complexity of `O(n)`, since a tree with a skewed structure could be a single branch.

3. After finding the node `x` and counting the left and right subtrees, we evaluate the maximum number of nodes in player 2's three choices (left subtree, right subtree, the rest of the tree) in constant time.

Therefore, the overall time complexity is `O(n)` for finding node `x` plus `O(n)` for counting the nodes in the left subtree and `O(n)` for counting the nodes in the right subtree.

Hence, the combined time complexity is `O(n) + O(n) + O(n)` which simplifies to `O(n)`.

Space Complexity

The space complexity is mainly due to the recursion stack used during the DFS and counting procedures.

1. `dfs` function: The recursive stack could in the worst case store `O(h)` frames, where `h` is the height of the tree, resulting in a space complexity of `O(h)`.

2. `count` function: After the `dfs` function has completed, the recursive `count` function is called separately for the left and right subtrees, each consuming space on the call stack. The space taken by both calls will not exceed `O(h)` in total at any time because they are called sequentially, not concurrently.

In the case of a balanced binary tree, the height `h` is `log(n)`, leading to a space complexity of `O(log(n))`. However, in the worst-case scenario of a skewed tree (similar to a linked list), where the tree's height is equal to the number of nodes, the space complexity becomes `O(n)`.

Therefore, the combined space complexity of the code is `O(n)` in the worst case.