1026. Maximum Difference Between Node and Ancestor

Medium Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

Given a binary tree, where each node contains an integer, we are asked to find the largest absolute difference in value between two nodes where one node is an ancestor of the other. In other words, if we pick any node `a` as an ancestor, and any node `b` as a descendant of `a`, what is the maximum absolute difference between `a.val` and `b.val` (`|a.val - b.val|`) that we can find in the tree?

An important detail to note is that a node can be considered an ancestor of itself, leading to a minimum absolute difference of 0 in such a scenario. The problem is focusing on finding the maximum difference, hence we need to look for pairs of ancestor and descendant nodes where this difference is the largest.

Intuition

The intuition behind the solution is that we can find the maximum difference by thoroughly searching through the tree. We do this using a depth-first search (DFS) algorithm, which will allow us to explore each branch of the tree to its fullest extent before moving on to the next branch.

During the traversal, we keep track of the minimum (`mi`) and maximum (`mx`) values encountered along the path from the root to the current node. At each node, we calculate the absolute difference between `root.val` and both the `mi` and `mx` values, updating the global maximum `ans` if we find a larger difference.

The core idea is to track the range of values (minimum and maximum) on the path from the root to the current node because this range will allow us to compute the required maximum absolute difference at each step. By the time we complete our traversal, we will have examined all possible pairs of ancestor and descendant nodes and thus found the maximum difference.

To implement this, we use a recursive helper function `dfs(root, mi, mx)` that performs a depth-first search on the binary tree. The `mi` and `mx` parameters keep track of the minimum and maximum values respectively, seen from the root to the current node. The function also updates a nonlocal variable `ans`, which keeps track of the maximum difference found so far.

Finally, we initiate our DFS with the root node and its value as both the initial minimum and maximum, and after completing the traversal, we return the value stored in `ans`, which will be the maximum ancestor-difference that we were tasked to find.

Solution Approach

The solution to this problem involves a recursive depth-first search (DFS) algorithm to traverse the binary tree. The critical aspect of the approach is to maintain two variables, `mi` and `mx`, to record the minimum and maximum values found along the path from the root node to the current node.

Here is a step-by-step breakdown of the implementation details:

1. Define a recursive helper function `dfs(root, mi, mx)` that will be used for DFS traversal of the tree.
2. If the current `root` is `None`, which means we've reached a leaf node's child, we return, as there are no more nodes to process in this path.
3. The helper function is designed to continuously update a nonlocal variable `ans`, which holds the maximum absolute difference found.
4. At each node, we compare and update `ans` with the absolute difference of the current node's value `root.val` with both the minimum (`mi`) and maximum (`mx`) values seen so far along the path from the root.
5. We perform this comparison using `max(ans, abs(mi - root.val), abs(mx - root.val))`.
6. After updating `ans`, we also update `mi` and `mx` for the recursive calls on the children nodes, setting `mi` to `min(mi, root.val)` and `mx` to `max(mx, root.val)`. This ensures that as we go deeper into the tree, our range `[mi, mx]` remains updated with the smallest and largest values seen along the path.
7. Recursive calls are then made to continue the DFS traversal on the left child `dfs(root.left, mi, mx)` and the right child `dfs(root.right, mi, mx)` of the current node.

The main function initializes the variable `ans` to 0 and then calls `dfs(root, root.val, root.val)`. We start with both `mi` and `mx` as the root's value, since initially, the root is the only node in the path. The implementation leverages the default argument-passing mechanism in Python, where every child node receives the current path's minimum and maximum values to keep the comparison going.

After the completion of the DFS traversal, the `ans` variable, which was kept up-to-date during the traversal, will contain the final result—the maximum difference. The function finally returns `ans`.

The primary data structure used in this implementation is the binary tree itself. No additional data structures are needed because the recursion stack implicitly manages the traversal, and the updating of minimum and maximum values is done using integer variables. This efficient use of space and recursive traversal makes it a neat and effective solution.

Example Walkthrough

Let's consider a small binary tree to illustrate the solution approach. Our binary tree is as follows:

```
        8
       / \
      3   10
     / \    \
    1   6    14
       / \   /
      4   7 13
```

We want to find the maximum absolute difference between the values of any two nodes where one is an ancestor of the other.

We begin by calling the recursive function `dfs` on the root node with value `8`. We start with `mi = mx = 8` since the root is both the minimum and maximum of the path consisting of just itself.

1. The `dfs` function is first called with `root.val = 8`, `mi = 8`, `mx = 8`. We are at the root.
2. Explore left child (`3`). Call `dfs(3, min(8,3), max(8,3))`:
   - Now `mi = 3`, `mx = 8`.
   - Update potential answer compare with previous `ans`:
     - `max(0, abs(3 - 8), abs(8 - 3))`
     - `ans = 5`
3. Go down to the left child of `3`, node `1`. Call `dfs(1, min(3,1), max(8,1))`:
   - Here, `mi = 1`, `mx = 8`.
   - Update answer:
     - `max(5, abs(1 - 8), abs(8 - 1))`
     - `ans = 7`

   Node `1` is a leaf; the traversal will go back up.
4. The other child of `3` is `6`. Call `dfs(6, min(3,6), max(8,6))`:
   - `mi = 3`, `mx = 8`.
   - Update answer:
     - `max(7, abs(3 - 6), abs(8 - 6))`
     - `ans` remains `7`.
5. Node `6` has a left child `4`. Call `dfs(4, min(3,4), max(8,4))`:
   - `mi = 3`, `mx = 8`.
   - Update answer:
     - `max(7, abs(3 - 4), abs(8 - 4))`
     - `ans` remains `7`.

   Since `4` is a leaf node, we go back up.
6. Node `6` also has right child `7`. Call `dfs(7, min(3,7), max(8,7))`:
   - `mi = 3`, `mx = 8`.
   - Update answer:
     - `max(7, abs(3 - 7), abs(8 - 7))`
     - `ans` remains `7`.

   Node `7` is also a leaf; traverse back up to `3`, then to `8`.
7. Now explore right child (`10`) of the root. Call `dfs(10, min(8,10), max(8,10))`:
   - `mi = 8`, `mx = 10`.
   - Update answer:
     - `max(7, abs(8 - 10), abs(10 - 8))`
     - `ans` remains `7`.
8. Node `10` has right child `14`. Call `dfs(14, min(8,14), max(10,14))`:
   - `mi = 8`, `mx = 14`.
   - Update answer:
     - `max(7, abs(8 - 14), abs(14 - 8))`
     - `ans` becomes `14 - 8 = 6` but since our current `ans = 7`, there is no update.
9. Node `14` has a left child `13`. Call `dfs(13, min(8,13), max(14,13))`:
   - `mi = 8`, `mx = 14`.
   - Update answer:
     - `max(7, abs(8 - 13), abs(14 - 13))`
     - `ans` remains `7` because the differences `5` and `1` are smaller than the current `ans`.

After the recursive depth-first search completes, we find that the maximum absolute difference is `7`, which comes from the difference between nodes `8` (ancestor) and `1` (descendant).

Python Solution

```python
class TreeNode:
    # A class for a binary tree node
    def __init__(self, val=0, left=None, right=None):
        self.val = val       # Node value
        self.left = left     # Left child
        self.right = right   # Right child

class Solution:
    def maxAncestorDiff(self, root: Optional[TreeNode]) -> int:
        # Helper function to perform Depth-First Search (DFS)
        def dfs(node, min_value, max_value):
            # Base case: if the current node is None, return
            if node is None:
                return

            # Calculate the maximum difference for the current node
            nonlocal max_difference
            current_diff = max(abs(min_value - node.val), abs(max_value - node.val))
            max_difference = max(max_difference, current_diff)

            # Update the minimum and maximum values with the value of the current node
            new_min = min(min_value, node.val)
            new_max = max(max_value, node.val)

            # Recursively call dfs for the left and right subtrees
            dfs(node.left, new_min, new_max)
            dfs(node.right, new_min, new_max)

        # Initialize max_difference which will hold the result
        max_difference = 0

        # Start DFS from root with its value as both initial min and max
        dfs(root, root.val, root.val)

        # Return the maximum difference found
        return max_difference

```

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;
    }
}

class Solution {
    private int maxDifference;

    /**
     * Calculates the maximum difference between values of any two connected nodes in the binary tree.
     * @param root The root of the binary tree.
     * @return The maximum difference calculated.
     */
    public int maxAncestorDiff(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // Start DFS with the initial value of the root for both minimum and maximum.
        depthFirstSearch(root, root.val, root.val);
        return maxDifference;
    }

    /**
     * A recursive DFS function that traverses the tree to find the maximum difference.
     * @param node The current node being visited.
     * @param minVal The minimum value seen so far in the path from root to the current node.
     * @param maxVal The maximum value seen so far in the path from root to the current node.
     */
    private void depthFirstSearch(TreeNode node, int minVal, int maxVal) {
        if (node == null) {
            return;
        }

        // Calculate the potential differences between the current node value and the observed min and max values.
        int currentMaxDifference = Math.max(Math.abs(minVal - node.val), Math.abs(maxVal - node.val));

        // Update the maxDifference if the current one is greater.
        maxDifference = Math.max(maxDifference, currentMaxDifference);

        // Update the min and max values to carry them forward in the DFS.
        minVal = Math.min(minVal, node.val);
        maxVal = Math.max(maxVal, node.val);

        // Recur for both the left and right subtrees.
        depthFirstSearch(node.left, minVal, maxVal);
        depthFirstSearch(node.right, minVal, maxVal);
    }
}

```

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:
    /* Function to calculate max difference between any ancestor and node value. */
    int maxAncestorDiff(TreeNode* root) {
        int maxDifference = 0; // to store the maximum difference

        // Lambda function for depth-first search starting from 'node'
        // It carries the current minimum and maximum values as 'currentMin' and 'currentMax'
        function<void(TreeNode*, int, int)> dfs = [&](TreeNode* node, int currentMin, int currentMax) {
            if (node == nullptr) {
                // If the node is null, return as there is nothing to process
                return;
            }
            // Update maxDifference with the maximum of the current difference
            // and the differences with the current node's value
            maxDifference = max({
                maxDifference,
                abs(currentMin - node->val),
                abs(currentMax - node->val)
            });

            // Update currentMin and currentMax with respective minimum and maximum values
            currentMin = min(currentMin, node->val);
            currentMax = max(currentMax, node->val);

            // Continue depth-first search on left and right subtrees
            dfs(node->left, currentMin, currentMax);
            dfs(node->right, currentMin, currentMax);
        };

        // Initialize DFS with the value of the root for both min and max
        dfs(root, root->val, root->val);

        // Return the maximum difference found
        return maxDifference;
    }
};

```

Typescript Solution

```typescript
// Global variable to track the maximum difference between an ancestor and a node value
let maxDifference: number = 0;

// Recursive function traverses the tree to find the maximum difference between an ancestor and a node value
function dfs(node: TreeNode | null, minVal: number, maxVal: number): void {
    if (!node) {
        // If node is null, return because we've reached a leaf node's child.
        return;
    }

    // Calculate the potential new max differences with the current node
    const potentialMaxDiff = Math.max(Math.abs(node.val - minVal), Math.abs(node.val - maxVal));

    // Update the global maxDifference if the new potential difference is greater
    maxDifference = Math.max(maxDifference, potentialMaxDiff);

    // Update the min and max values seen so far after considering the current node's value
    const newMinVal = Math.min(minVal, node.val);
    const newMaxVal = Math.max(maxVal, node.val);

    // Continue the DFS traversal for left and right children
    dfs(node.left, newMinVal, newMaxVal);
    dfs(node.right, newMinVal, newMaxVal);
}

// Primary function to initiate the maxAncestorDiff calculation, given the root of a binary tree
function maxAncestorDiff(root: TreeNode | null): number {
    if (root === null) {
        // If the tree is empty, the maximum difference is 0 by definition
        return 0;
    }

    // Since we start at the root, the starting min and max values are the root's value
    dfs(root, root.val, root.val);

    // After traversing the tree, return the global maxDifference found
    return maxDifference;
}

```

Time and Space Complexity

The given Python function `maxAncestorDiff` computes the maximum difference between the values of any two nodes with an ancestor/descendant relationship in a binary tree.

Time Complexity:

The time complexity of the function is `O(N)`, where `N` is the number of nodes in the binary tree. This is because the function performs a depth-first search (DFS), visiting each node exactly once. During each visit, it performs a constant amount of work by updating the minimum and maximum values encountered so far and comparing them to the current node's value.

Space Complexity:

The space complexity of the function is `O(H)`, where `H` is the height of the binary tree. This complexity comes from the call stack used for recursion during DFS. In the worst case of a skewed tree, where the tree takes the form of a linked list (either every node has only a right child or only a left child), the height `H` would be equal to `N`, leading to a worst-case space complexity of `O(N)`. For a balanced tree, the space complexity would be `O(log N)`, as the height `H` would be logarithmic in the number of nodes `N`.