1080. Insufficient Nodes in Root to Leaf Paths

Medium Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

The given LeetCode problem deals with a binary tree and a specified limit value. The objective is to examine all paths from the root to the leaves, and for each node, determine if it is part of any path where the sum of the node values meets or exceeds the limit. If a node does not lie on such a path, it is considered 'insufficient' and should be deleted. After deleting all insufficient nodes, the modified binary tree is returned.

The key point is to understand that a node is 'insufficient' if *every* path through that node has a sum of values less than the limit – meaning it doesn't support any path that would satisfy the threshold. Keep in mind, leaf nodes are the ones with no children.

Intuition

The intuition behind the solution revolves around recursively checking each node, starting from the root and going down to the leaves. As we traverse path by path, we subtract the node's value from the limit, effectively calculating the sum of the path as we progress. If we reach a leaf node (a node with no children), we check if the updated limit is greater than 0. If it is, then the path sum up to this node was not sufficient, and this leaf node is deleted (return `None`).

For non-leaf nodes, we apply this process recursively to both the left and right children. After assessing both subtrees, we have to determine if the current node becomes a leaf node as a result (i.e., both children are `None` after potentially deleting insufficient nodes). If after deletion of child nodes the current node is a leaf and it was insufficient, we delete the current node as well (again, return `None`); otherwise, we keep it.

The process continues until all nodes are visited. If the root itself turns out to be insufficient, the result will be an empty tree (i.e., the root would also be returned as `None`).

The solution elegantly side-steps the need to keep track of all paths from the root to each node by updating the `limit` on the go and utilizing the recursive stack to backtrack once a leaf has been reached or a subtree has been pruned.

Solution Approach

To solve this problem, a depth-first search (DFS) algorithm is employed. This recursive approach allows us to traverse the tree from the root node to the leaves, checking each node along the way to see if it is sufficient with respect to the `limit`.

Here's a step-by-step explanation of the code:

- The `sufficientSubset` function is defined with `root` (the current node) and `limit` (the remaining path sum before reaching insufficiency) as parameters.
- The base case for the recursion is checking if the current `root` node is `None`. If it is, we return `None`, effectively ending that path.
- For each node, we subtract the node's value from the `limit`. This step accumulates the sum of node values on the current path.
- If the current node is a leaf (`root.left is None and root.right is None`), we check if the reduced limit is positive. If it is positive, then this path does not satisfy the sum requirement, so we return `None` to delete this leaf. Otherwise, we return the current node itself as this node is sufficient and should remain.
- If the current node is not a leaf, we recursively call `sufficientSubset` on both the left and right children of the node. Here, we pass the updated limit after subtracting the current node's value.
- After the recursive calls, we need to determine if the current node should be deleted. This is based on whether both of its children are `None` after the potential deletion of insufficient nodes. If both children are `None`, we return `None`, deleting the current node. If at least one child remains, we return the current node itself, as it supports a sufficient path.
- This recursive process will prune all insufficient nodes from the tree and, once the recursion stack unwinds back to the root, will return the new tree rooted at the (potentially new) 'root'.

This elegant approach ensures we only traverse each node once, giving us an efficient time complexity proportional to the number of nodes in the tree, which is O(n).

Example Walkthrough

Let's consider a simple binary tree alongside a `limit` to see how the algorithm works:

```
   Node values:       Limit:

       5                10
      / \
     4   8
    /   / \
   11  13  4
  / \       \
 7   2       1
```

Following the described approach:

1. Start at the root: `Node(5)`. Since its value is less than the limit (10), continue the recursion. Update the limit to `limit - node.value` (`10 - 5 = 5`).
2. Recurse left to `Node(4)` and right to `Node(8)` with the new limit (5).
3. At `Node(4)`:
   - Subtract its value from the limit (`5 - 4 = 1`), and recurse left to `Node(11)` (it has no right child).
   - At `Node(11)`, subtract its value from the limit (`1 - 11 = -10`), which means we have surpassed our limit when we reach the children of `Node(11)`.
   - Both children of `Node(11)`, `Node(7)` and `Node(2)`, when evaluated, would result in a negative limit after subtracting their values. Thus, they will both be pruned and `Node(11)` will be left with no children.
   - Since `Node(11)` is now effectively a leaf and has no sufficient subpath (its children were pruned), it is also pruned. `Node(4)` is left with no children.
   - `Node(4)` becomes a leaf and is insufficient, so it is pruned.
4. Back at the root, recurse to the right to `Node(8)`:
   - Update the limit (`5 - 8 = -3`). The condition is satisfied for `Node(8)` since we haven't encountered a leaf. Hence, continue.
   - Recurse to `Node(13)` and `Node(4)` with the new limit.
   - `Node(13)` is a leaf. Check `(-3 - 13)`. This is negative, which means the path of `Node(8) -> Node(13)` is sufficient, so we keep `Node(13)`.
   - `Node(4)` isn't a leaf. Update limit (`-3 - 4 = -7`).
     - Recurse to the right to `Node(1)`. `Node(4)` has no left child.
     - `Node(1)` is a leaf. Check `(-7 - 1)`, which is still negative, so the path of `Node(8) -> Node(4) -> Node(1)` is sufficient, and we keep `Node(1)`.
5. After all recursions, the insufficient nodes have been pruned. At the end, our tree will look like:

```
       5
        \
         8
        / \
       13  4
            \
             1
```

This example demonstrates the principle of the depth-first search (DFS) algorithm in action, recursively pruning nodes that do not support paths with sums meeting or exceeding the given limit. When the recursion unwinds, the resultant tree is bereft of all insufficient nodes, and the sufficient structure is returned.

Python Solution

```python
# Class definition for a binary tree node.
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right


class Solution:
    def sufficientSubset(self, root: Optional[TreeNode], limit: int) -> Optional[TreeNode]:
        """
        Removes subtrees that are not sufficient, i.e., the total sum of any path from the root
        to any leaf node is less than the given limit.

        :param root: TreeNode - the root of the binary tree
        :param limit: int - the threshold limit for the sum of the path values
        :return: TreeNode - the modified tree with insufficient subtrees removed
        """

        # If the root node is None, just return None.
        if root is None:
            return None

        # Reduce the remaining limit by the value of the current node.
        limit -= root.value

        # If it's a leaf node and the limit is not reached, prune this node.
        if root.left is None and root.right is None:
            return None if limit > 0 else root

        # Recursively prune the left and right subtrees.
        root.left = self.sufficientSubset(root.left, limit)
        root.right = self.sufficientSubset(root.right, limit)

        # If both subtrees are pruned, prune this node too.
        return None if root.left is None and root.right is None else root

```

Java Solution

```java
/* Class definition for a binary tree node. */
class TreeNode {
    int val; // Value of the node
    TreeNode left; // Pointer to the left child
    TreeNode right; // Pointer to the right child

    /* Constructor for creating a tree node without children. */
    TreeNode(int val) {
        this.val = val;
    }

    /* Constructor for creating a tree node with given value, left child, and right child. */
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

/* Solution class contains the method 'sufficientSubset' to prune the tree. */
class Solution {

    /**
     * Prunes the tree such that the sum of values from root to . any leaf node is at least 'limit'.
     *
     * @param root The root of the binary tree.
     * @param limit The minimum sum from root to leaf required.
     * @return The pruned binary tree.
     */
    public TreeNode sufficientSubset(TreeNode root, int limit) {
        // Base case: if the current node is null, return null as there's nothing to check or prune.
        if (root == null) {
            return null;
        }

        // Subtract the current node's value from the remaining limit.
        limit -= root.val;

        // Check if the current node is a leaf node.
        if (root.left == null && root.right == null) {
            // If the remaining limit is still greater than 0 after considering current node's value,
            // it means the path sum of this leaf is insufficient, hence return null (prune it).
            // Otherwise, return the current leaf as it satisfies the condition.
            return limit > 0 ? null : root;
        }

        // Recursive call for the left subtree, potentially prune the left child.
        root.left = sufficientSubset(root.left, limit);
        // Recursive call for the right subtree, potentially prune the right child.
        root.right = sufficientSubset(root.right, limit);

        // After the recursive calls, if both children are null, it means they were pruned, then the
        // current node becomes a leaf and we'll need to check whether it should also be pruned or not.
        // If at least one child remains, the current node should also remain.
        return (root.left == null && root.right == null) ? null : root;
    }
}

```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    // Constructor with default value initialization
    TreeNode(int x = 0, TreeNode *left = nullptr, TreeNode *right = nullptr)
        : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Function to prune the tree based on the limit provided.
    TreeNode* sufficientSubset(TreeNode* root, int limit) {
        // Base case: If the node is null, return null.
        if (!root) {
            return nullptr;
        }

        // Subtract the value of the current node from the limit.
        limit -= root->val;

        // Check if it's a leaf node.
        if (!root->left && !root->right) {
            // If the updated limit is greater than 0, the path sum is insufficient; prune this node.
            return limit > 0 ? nullptr : root;
        }

        // Recursive case: Traverse down to the left and right subtrees.
        root->left = sufficientSubset(root->left, limit);
        root->right = sufficientSubset(root->right, limit);

        // If after the pruning, the current node becomes a leaf node (i.e., both left and right are null),
        // and the path doesn't meet the criteria, return null, else return the node itself.
        return root->left == nullptr && root->right == nullptr ? nullptr : root;
    }
};

```

Typescript Solution

```typescript
// Definition for the binary tree node
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Removes the subtrees where the total sum of the path from the root node to any leaf is less than the given limit.
 * @param root - The current node of the binary tree.
 * @param limit - The minimum required sum from the root to leaf path.
 * @returns The modified subtree, or null if the subtree's sum is insufficient.
 */
function sufficientSubset(root: TreeNode | null, limit: number): TreeNode | null {
    // Base case: if the current node is null, return null
    if (root === null) {
        return null;
    }

    // Subtract the value of the current node from the remaining limit
    limit -= root.val;

    // If the current node is a leaf, check if the sum of the path meets the limit
    if (root.left === null && root.right === null) {
        // Return null if the sum is insufficient; otherwise return the current node
        return limit > 0 ? null : root;
    }

    // Recursively call sufficientSubset on the left and right subtrees
    root.left = sufficientSubset(root.left, limit);
    root.right = sufficientSubset(root.right, limit);

    // If both children are removed, remove the current node as well
    if (root.left === null && root.right === null) {
        return null;
    }

    // Otherwise, return the current node with its potentially pruned children
    return root;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(N)`, where `N` is the number of nodes in the binary tree. This is because the function visits each node exactly once in a depth-first search manner. It performs a constant amount of work at each node by subtracting the node's value from the `limit` and deciding whether to keep or discard the node based on the updated limit.

Space Complexity

The space complexity of the provided code is `O(H)`, where `H` is the height of the binary tree. This complexity arises due to the recursive call stack that can grow up to `H` levels deep in the case of a skewed tree (where `H` can be equal to `N` in the worst case). For a balanced binary tree, the height `H` would be `log(N)`, resulting in a space complexity of `O(log(N))`. However, in the worst case (such as a skewed tree), the space complexity is `O(N)`.