124. Binary Tree Maximum Path Sum

Hard · Java · Depth-First Search · Dynamic Programming · Binary Tree

Leetcode Link

Problem Description

In this problem, we are given the root of a binary tree and need to find the maximum path sum from any sequence of nodes in the tree. Here, a path is defined as a sequence of nodes where consecutive nodes are connected by an edge, and a node can only be used once within a single path. The path sum is the total of the values of all nodes in that path. Notably, the path does not have to go through the root of the tree. Our goal is to calculate the maximum sum from all such possible paths in the tree.

Intuition

When dealing with trees, recursion is often a natural approach. Since we are looking for the maximum path sum, at every node we have a choice: include that node on a path extending left or right, or start a new path through that node. We can recursively find the maximum path sums going through the left child and the right child. However, when combining these sums at a node, we must realize that we cannot include both child paths since that would create a loop, not a valid path.

The solution's intuition hinges on realizing that for any node, the maximum sum wherein that node is the highest point (i.e., the 'root' of that path) is its value plus the maximum sums of its left and right subtrees. We only add the subtree sums if they are positive, since any path would only include a subtree if it contributed positively to the total sum.

We define the `dfs` function that does the following:

- When the node is `None`, return 0 because an empty node contributes nothing to the path sum.
- Recursively calculate the maximum path sum for the left and right subtrees. If the sum is negative, we reset it to 0, as described earlier.
- Calculate the potential maximum path sum at the current node by adding the node's value to the maximum path sums from both subtrees. This represents the largest value that could be achieved when passing through that node and potentially including both left and right paths (but not combining them).
- Update a global variable `ans` that tracks the overall maximum path sum found anywhere in the tree with this new potential maximum.
- Finally, for the recursion to continue upwards, return the node's value plus the greater of the two maximum sums from the left or right subtree. This represents the best contribution that node can make towards a higher path.

The `maxPathSum` function initiates this recursive process, starting from the root, and returns the maximum path sum found.

Solution Approach

The solution uses a Depth-First Search (DFS) algorithm to recursively traverse the binary tree and calculate the path sums. This approach ensures that every node is visited, and that the maximum path sums for the subtrees of each node are considered. Here are the steps and ideas involved:

1. The core function is `dfs`, which is a recursive function that takes a node of the tree as an argument and returns the maximum path sum obtained by including the current node and extending to either its left or right child (not both).
2. In the base case, if the current node is `None`, the function returns `0`, meaning there is no contribution to the path sum.
3. When `dfs` is called on a non-null node, it first recursively calculates the maximum path sums of the left and right subtrees with `left = max(0, dfs(root.left))` and `right = max(0, dfs(root.right))`. The `max(0, ...)` pattern is used to ignore negative path sums since including a negative path would decrease the overall sum, which is not optimal.
4. After obtaining the maximum non-negative path sums from the left and right subtrees, it calculates the maximum path sum that includes the current node and both children: `root.val + left + right`. This is not the value returned by the function because a path should only extend in one direction; however, this value is crucial because it is the highest path sum for the subtree for which `root` is the highest point.
5. The global variable `ans` is updated with the maximum of its current value and the newly calculated sum (`ans = max(ans, root.val + left + right)`). This step is essential because `ans` keeps track of the maximum path sum found in the entire tree, including paths that do not extend to the root of the entire tree.
6. Finally, `dfs` returns `root.val + max(left, right)`, taking into account the current node's value and the maximum contribution from either the left or right child. This allows the function to relay upward through the recursion the best path sum contribution of the current node to its parent.

The overall maximum path sum is found by setting the initial value of `ans` to `-inf` to ensure that any path sum in the tree will be larger (since the tree is non-empty), and then calling `dfs(root)` which triggers the recursive process. The final value of `ans` after the recursion completes gives us the result we are seeking – the maximum path sum of any non-empty path in the given binary tree.

Example Walkthrough

Let's illustrate the solution approach with a small binary tree example:

Consider the following binary tree:

```
1         5
2        / \
3       4   8
4      /   / \
5     11  13  4
6    /  \      \
7   7    2      1
```

We want to find the maximum path sum in this tree using the solution approach described.

1. Start with the function `dfs` at the root node with the value `5`.
2. The `dfs` function is called recursively on the left child `4` and the function call sequence continues down to `11`.
3. At `11`, `dfs` goes to `7` (left child), which returns `7` because it is a leaf node.
4. Then `dfs` goes to the `2` (right child), which returns `2`.
5. At node `11`, the function now has left and right values: the `dfs` result from the left is `7` and the right is `2`. Node `11` itself has a value of `11`, so the local maximum we can generate at this node (including its value and both children) is `11 + 7 + 2 = 20`. However, for the purposes of propagation up the tree, we return `11 + max(7, 2) = 11 + 7 = 18`.
6. The global maximum `ans` is updated if `20` is greater than the current value of `ans`.
7. The recursion unfolds, and a similar process occurs for node `4` on the left and for `8`, `13`, `4`, and `1` on the right side of the tree.
8. At node `5`, the maximum path sums from left and right sides are collected (already calculated by `dfs` for child nodes).
9. The overall maximum path that can be formed including node `5` and its children is compared against the global maximum `ans`. The value returned is `5 + max(left, right)` to propagate up the tree.
10. In each step, whenever a higher path sum is found at a node, the global maximum `ans` is updated.
11. The process continues until the topside of the recursion, where the `dfs` function was initially called with the root of the tree.
12. The final value of `ans` after recursion completes represents the maximum path sum from any sequence of nodes in the given binary tree.

For our example, let's assume that the left side (with nodes `5`, `4`, and `11`) produces a path sum of `18`, and the right side (with nodes `5`, `8`, `4`, and `1`) produces a path sum of `18` as well. The highest path sum that includes the root would be `5 + 18 + 18 = 41`. But suppose there's an even higher path sum that doesn't include the root, say `13 + 8 + 4`, which equals `25` (node `13` is the root of this path). Hence, the final answer for this example would depend on which sum is larger, and `ans` would be updated with the maximum sum found.

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 maxPathSum(self, root: TreeNode) -> int:
        # Helper function to perform depth-first search
        def dfs(node: TreeNode) -> int:
            # Base case: if the current node is None, return 0
            if not node:
                return 0

            # Recursively calculate the maximum path sum on the left subtree
            # If negative, we take 0 to avoid decreasing the overall path sum
            left_max = max(0, dfs(node.left))

            # Similarly, do the same for the right subtree
            right_max = max(0, dfs(node.right))

            # Update the overall maximum path sum
            # This includes the node value and the maximum paths from both subtrees
            nonlocal max_path_sum
            max_path_sum = max(max_path_sum, node.val + left_max + right_max)

            # Return the maximum path sum without splitting
            # The current node's value plus the greater of its left or right subtree paths
            return node.val + max(left_max, right_max)

        # Initialize the overall maximum path sum to negative infinity
        # To account for potentially all negative-valued trees
        max_path_sum = float('-inf')

        # Start DFS with the root of the tree
        dfs(root)

        # After DFS is done, max_path_sum holds the maximum path sum for the tree
        return max_path_sum
```

Java Solution

```java
class Solution {
    private int maxSum = Integer.MIN_VALUE;  // Initialize maxSum with the smallest possible integer value.

    // Returns the maximum path sum of any path that goes through the nodes of the given binary tree.
    public int maxPathSum(TreeNode root) {
        calculateMaxPathFromNode(root);
        return maxSum;
    }

    // A helper method that computes the maximum path sum from a given node and updates the overall maxSum.
    private int calculateMaxPathFromNode(TreeNode node) {
        if (node == null) {
            // If the current node is null, we return 0 since null contributes nothing to the path sum.
            return 0;
        }
        // Compute and get the maximum sum of paths from the left child.
        // If the value is negative, we ignore the left child's contribution by taking 0.
        int leftMaxSum = Math.max(0, calculateMaxPathFromNode(node.left));
        // Compute and get the maximum sum of paths from the right child.
        // If the value is negative, we ignore the right child's contribution by taking 0.
        int rightMaxSum = Math.max(0, calculateMaxPathFromNode(node.right));
        // Update maxSum with the greater of the current maxSum or the sum of the current node value plus leftMaxSum and rightMaxSum.
        // This accounts for the scenario where the path involving the current node and its left and right children yields the max pa
        maxSum = Math.max(maxSum, node.val + leftMaxSum + rightMaxSum);
        // This call must return the maximum path sum including the currently evaluated node and one of its subtrees
        // to ensure a path cannot have branches, and must go straight through the parents or children nodes.
        return node.val + Math.max(leftMaxSum, rightMaxSum);
    }
}
/**
 * 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;
    }
}
```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;        // Node's value
    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 maxPathSum(TreeNode* root) {
        int maxValue = INT_MIN; // Initialize the maximum path sum as the smallest possible integer.

        // Depth-first search function to explore nodes and calculate max path sum.
        std::function<int(TreeNode*)> depthFirstSearch = [&](TreeNode* node) {
            if (!node) {
                return 0; // Base case: If the node is null, return 0 as it adds nothing to the path sum.
            }
            // Recursively calculate the max path sum for the left and right children, ignoring negative sums.
            int leftMax = std::max(0, depthFirstSearch(node->left));
            int rightMax = std::max(0, depthFirstSearch(node->right));

            // Update the global max path sum by considering the current node value and its maximum left and right path sums.
            maxValue = std::max(maxValue, leftMax + rightMax + node->val);

            // Return the maximum path sum of either the left or right subtree plus the current node value.
            return node->val + std::max(leftMax, rightMax);
        };

        // Start DFS from root.
        depthFirstSearch(root);

        return maxValue; // Return the global maximum path sum found.
    }
};
```

Typescript Solution

```typescript
// Definition for a binary tree node.
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);
    }
}

// Initialize a global variable to hold the maximum path sum
let maxPathSumValue = -1001;

/**
 * Helper function to calculate the maximum path sum
 * @param {TreeNode | null} node - The current node of the binary tree
 * @return {number} The maximum sum from the current node to any leaf
 */
function maxPathSumDfs(node: TreeNode | null): number {
    // Base case: If the node is null, return 0
    if (!node) {
        return 0;
    }

    // Calculate maximum sum starting from left child
    const leftMax = Math.max(0, maxPathSumDfs(node.left));

    // Calculate maximum sum starting from right child
    const rightMax = Math.max(0, maxPathSumDfs(node.right));

    // Update global maxPathSumValue with the maximum sum of the current subtree
    // which includes the current node and both left and right maximum sums
    maxPathSumValue = Math.max(maxPathSumValue, leftMax + rightMax + node.val);

    // Return the maximum sum from current node to any leaf
    return Math.max(leftMax, rightMax) + node.val;
}

/**
 * Function to find the maximum path sum of a binary tree
 * @param {TreeNode | null} root - The root of the binary tree
 * @return {number} The maximum path sum in the binary tree
 */
function maxPathSum(root: TreeNode | null): number {
    maxPathSumDfs(root); // Call the helper function starting from the root
    return maxPathSumValue; // Return the global maximum path sum
}
```

Time and Space Complexity

The time complexity of the code is `O(n)` because the `dfs` function visits each node in the binary tree exactly once, where `n` is the number of nodes in the binary tree.

The space complexity of the code is `O(h)`, where `h` is the height of the tree. This is due to the recursion stack during the depth-first search (DFS). In the worst case of a skewed tree, the space complexity becomes `O(n)`, where `n` is the number of nodes, because the height of the tree can be `n` in the case of a completely unbalanced tree. For a balanced tree, the space complexity would be `O(log n)`, because the height `h` would be proportional to `log n`.