1120. Maximum Average Subtree

Medium Tree Depth-First Search Binary Tree

Leetcode Link

Problem Description

The problem provides us with the root node of a binary tree and asks us to calculate the maximum average value of any subtree within that binary tree. Here, a subtree can be any node and all of its descendants—it could be the entire tree, a leaf node, or any other set of connected nodes within the tree.

The average value of a subtree is the sum of all node values in that subtree divided by the number of nodes it contains. Since we are looking for the maximum average, we are concerned with finding the subtree that has the highest average value compared to all other subtrees in the binary tree.

Intuition

The intuitive approach to solving this problem is to explore each subtree, calculate its sum and node count, and then use these to compute the average value for that particular subtree. To achieve this, we'll need to perform a traversal of the binary tree.

A depth-first search (DFS) traversal is suitable for this task because it allows us to go deep into the tree to examine individual subtrees before moving on to the next subtree. This way, we can use a postorder traversal (left-right-root) to compute the sum and node count of each subtree.

While traversing, for each node, we calculate:

1. The total sum of node values (`ts`) in the current subtree (includes the current node, its left subtree, and its right subtree).
2. The total number of nodes (`ns`) in the current subtree.

With both these values at hand for each subtree, we then calculate the average by dividing `ts` by `ns`. This average is then compared to the current maximum average we've found (`ans`). If the current average is greater, we update `ans` with this new value.

The purpose of the `nonlocal ans` in the solution code is to allow the nested `dfs` function to modify the `ans` variable defined in the enclosing scope, which maintains the current maximum average.

As we recursively return the sum and node count for each subtree, we can easily compute the average for each parent subtree, inherently checking every subtree in the binary tree. The function ultimately returns the maximum average found during the traversal.

Solution Approach

The solution uses a recursive function to traverse the tree using depth-first search (DFS), specifically a postorder traversal pattern. In postorder traversal, we visit the left and right subtrees before processing the current node. This allows us to gather information from the subtrees before deciding on the parent node's information, which is essential for this problem as we need to compute subtrees' sums and node counts.

Here's a breakdown of the solution implementation:

1. Define a recursive function `dfs` that takes a node as an argument and returns a tuple `(s, n)` where:
   - `s` is the sum of all nodes within the current subtree.
   - `n` is the count of the nodes within the current subtree.
2. When the `dfs` function is called with `None` (i.e., the subtree is empty), it returns `(0, 0)`, indicating that the sum and count are both 0 for an empty subtree.
3. The function calculates `ls` and `le` by recursively calling `dfs` on the left child of the current node, and similarly `rs` and `rn` for the right child.
4. Once we have the sums (`ts` and `rs`) and counts (`ln` and `rn`) for both child subtrees, compute the sum `s` for the current node, which is the value of the current node plus `ls` plus `rs`.
5. Compute the count `n` for the current node as 1 (to count the current node) plus `ln` plus `rn`.
6. Use a `nonlocal` variable `ans` to keep track of the current maximum average. This variable is defined outside the `dfs` function so that it retains its value across different calls to `dfs` and isn't reinitialized in each recursive call.
7. Update `ans` with the maximum value between the current `ans` and the average for the current subtree (`s / n`).
8. After processing the root node, the `dfs` traversal ensures that each node's subtree has been considered. The `ans` variable will contain the maximum average.
9. The main function of the class `Solution` will initialize `ans` to 0 and call the `dfs` function on the binary tree's root.
10. Lastly, return the `ans` variable as the final result, which represents the maximum average value of a subtree of the binary tree.

This approach is efficient because it traverses each node of the tree exactly once, resulting in an `O(n)` time complexity, where `n` is the number of nodes in the binary tree.

The code for calculating sum and number of nodes in each subtree looks like this:

```
1  if root is None:
2      return 0, 0
3  ls, ln = dfs(root.left) # Postorder traversal - Left
4  rs, rn = dfs(root.right) # Postorder traversal - Right
5  s = root.val + ls + rs # Sum of current subtree
6  n = 1 + ln + rn # Number of nodes in current subtree
```

And updating the maximum average found so far is as simple as:

```
1 nonlocal ans
2 ans = max(ans, s / n)
```

By carefully updating the `ans` variable during the traversal, we ensure that we have the maximum average value of any subtree by the end of the traversal.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach. Consider a binary tree:

```
1       5
2      / \
3     6   8
4    / \ / \
5   1  7 3  9
```

We want to find the maximum average value of any subtree.

- Start by calling the `dfs` function on the root. We initialize `ans = 0`.
- At Node 5 - the root:
  1. First, call `dfs` on the left subtree (Node 6).
     - At Node 6:
       - Call `dfs` on the left subtree (Node 1), which returns `(1, 1)` - its own value and count since it is a leaf node.
       - Call `dfs` on the right subtree (None), which returns `(0, 0)`.
       - Now, the sum at Node 6 is 6 (itself) + 1 (left subtree) = 7 and the count is 1 (itself) + 1 (left subtree) = 2.
       - Update the maximum average `ans` if the average here (7 / 2 = 3.5) is greater than the current `ans`.
  2. Then, call `dfs` on the right subtree (Node 8).
     - At Node 8:
       - Call `dfs` on the left subtree (Node 7), returning `(7, 1)`.
       - Call `dfs` on the right subtree (Node 3), returning `(3, 1)`.
       - Sum at Node 8 is 8 + 7 + 3 = 16 and count is 1 + 1 + 1 = 3.
       - Update the maximum average `ans` if the average here (16 / 3 < 6) is greater than the current `ans`.
  3. Now, the sum at Node 5 is 5 (itself) + 7 (left subtree) + 18 (right subtree) = 30 and the count is 1 (itself) + 2 (left subtree) + 3 (right subtree) = 6.
  4. Update the maximum average `ans` if the average here (30 / 6 = 5) is greater than the current `ans`.
- After traversing the entire tree, the `ans` variable holds the maximum average found, which is 6 in this case (the average value of the subtree rooted with Node 8).

This example illustrates the process of traversing the tree using postorder traversal, where we calculate the sum and count of the nodes for each subtree, updating the maximum average found (`ans`) along the way. The tree is only traversed once, making the solution efficient with O(n) complexity, where n is the number of nodes in the binary tree.

Python Solution

```
1  # Definition for a binary tree node.
2  class TreeNode:
3      def __init__(self, val=0, left=None, right=None):
4          self.val = val
5          self.left = left
6          self.right = right
7
8  class Solution:
9      def maximumAverageSubtree(self, root: Optional[TreeNode]) -> float:
10         # Helper function to perform DFS (Depth-First Search) on the tree.
11         # This function returns the sum of subtree values and the count of nodes in the subtree.
12         def dfs(node):
13             if node is None:
14                 # If the node is null, return sum = 0 and count = 0
15                 return 0, 0
16             # Recurse on the left child and get left subtree sum and node count
17             left_sum, left_count = dfs(node.left)
18             # Recurse on the right child and get right subtree sum and node count
19             right_sum, right_count = dfs(node.right)
20             # Calculate the total sum and total count of nodes including this node
21             total_sum = node.val + left_sum + right_sum
22             total_count = 1 + left_count + right_count
23             # Update the maximum average if needed
24             nonlocal max_average
25             max_average = max(max_average, total_sum / total_count)
26             # Return the sum and count to parent node's calculation
27             return total_sum, total_count
28
29         # Initialize the maximum average variable
30         max_average = 0
31         # Call the helper function and start the DFS from the root
32         dfs(root)
33         # Return the final maximum average value after traversing all subtrees
34         return max_average
35
```

Java Solution

```
1  class Solution {
2      // A member variable to store the maximum average found so far.
3      private double maxAverage;
4
5      // This method starts the process by calling the DFS method on the root.
6      public double maximumAverageSubtree(TreeNode root) {
7          dfs(root);
8          return maxAverage;
9      }
10
11     // Perform depth-first search to calculate sum and count of nodes for each subtree.
12     private int[] dfs(TreeNode node) {
13         // If the node is null, return an array containing 0 sum and 0 node count.
14         if (node == null) {
15             return new int[2]; // Default initialization to {0, 0}.
16         }
17
18         // Recursive case: Calculate the sum of values and count of nodes in the left subtree.
19         int[] leftSubtreeResult = dfs(node.left);
20         // Recursive case: Calculate the sum of values and count of nodes in the right subtree.
21         int[] rightSubtreeResult = dfs(node.right);
22
23         // Computing the sum and count for the current subtree including the current node's value.
24         int subSubtree = node.val + leftSubtreeResult[0] + rightSubtreeResult[0];
25         int countSubtree = 1 + leftSubtreeResult[1] + rightSubtreeResult[1];
26
27         // Update the maximum average if the average of the current subtree is greater.
28         maxAverage = Math.max(maxAverage, (double) subSubtree / countSubtree);
29
30         // Return an array containing the sum and count for use in parent node calculations.
31         return new int[] {subSubtree, countSubtree};
32     }
33 }
34
35 /**
36  * Definition for a binary tree node.
37  * public class TreeNode {
38  *     int val; // The value of the node.
39  *     TreeNode left; // Reference to the left child node.
40  *     TreeNode right; // Reference to the right child node.
41  *     TreeNode() {} // Constructor with no arguments.
42  *     TreeNode(int val) { this.val = val; } // Constructor with initial value.
43  *     TreeNode(int val, TreeNode left, TreeNode right) { // Constructor with initial value and left/right children.
44  *         this.val = val;
45  *         this.left = left;
46  *         this.right = right;
47  *     }
48  * }
49  */
50
```

C++ Solution

```
1  #include <algorithm> // For std::max
2  #include <functional> // For std::function
3  #include <utility> // For std::pair
4
5  // Definition for a binary tree node.
6  struct TreeNode {
7      int val;
8      TreeNode *left;
9      TreeNode *right;
10     TreeNode() : val(0), left(nullptr), right(nullptr) {}
11     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
12     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
13 };
14
15 class Solution {
16 public:
17     double maximumAverageSubtree(TreeNode* root) {
18         double maxAverage = 0; // This will hold the maximum average found.
19
20         // Recursive function to perform DFS (Depth-First Search).
21         std::function<std::pair<int, int>(TreeNode*)> dfs = [&](TreeNode* node) -> std::pair<int, int> {
22             if (!node) {
23                 return {0, 0}; // If the node is null, return 0 sum and 0 count.
24             }
25             // Compute the sum and count for left subtree.
26             std::pair<int, int> leftSubtree = dfs(node->left);
27             // Compute the sum and count for right subtree.
28             std::pair<int, int> rightSubtree = dfs(node->right);
29
30             // Current sum is the value of the node plus sum of left and right subtrees.
31             int currentSum = node->val + leftSubtree.first + rightSubtree.first;
32             // Current count is 1 (for the current node) plus count of left and right subtrees.
33             int currentCount = 1 + leftSubtree.second + rightSubtree.second;
34
35             // Update the maximum average if the current average is greater.
36             maxAverage = std::max(maxAverage, static_cast<double>(currentSum) / currentCount);
37
38             // Return the current sum and count for further use.
39             return {currentSum, currentCount};
40         };
41
42         // Start DFS from the root of the tree.
43         dfs(root);
44
45         // After the DFS is complete, maxAverage holds the maximum average value.
46         return maxAverage;
47     }
48 };
49
```

Typescript Solution

```
1  // Definition for a binary tree node.
2  class TreeNode {
3      val: number;
4      left: TreeNode | null;
5      right: TreeNode | null;
6
7      constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
8          this.val = val;
9          this.left = left;
10         this.right = right;
11     }
12 }
13
14 let maximumAverage: number = 0.0; // This will hold the maximum average found.
15
16 // A helper function that takes a TreeNode and returns the sum and number of nodes
17 // in the subtree rooted at the given node using depth-first search.
18 function dfs(node: TreeNode | null): [number, number] {
19     if (!node) {
20         return [0, 0]; // If the node is null, return 0 sum and 0 count.
21     }
22
23     // Compute the sum and count for the left and right subtrees.
24     const [leftSum, leftCount] = dfs(node.left);
25     const [rightSum, rightCount] = dfs(node.right);
26
27     // Calculate current sum by adding the node's value to the sum of its subtrees.
28     const currentSum = node.val + leftSum + rightSum;
29     // Calculate current count by adding 1 (for the node)
30     // plus the count of left and right subtrees.
31     const currentCount = 1 + leftCount + rightCount;
32
33     // Update the maximum average if the current node's average is greater.
34     maximumAverage = Math.max(maximumAverage, currentSum / currentCount);
35
36     // Return the current sum and count for the subtree.
37     return [currentSum, currentCount];
38 }
39
40 // The entry function that initiates the depth-first search to find the maximum
41 // average of any subtree in the binary tree, starting from the root.
42 function maximumAverageSubtree(root: TreeNode | null): number {
43     dfs(root); // Start DFS from the root of the tree.
44     return maximumAverage;
45 }
```

Time and Space Complexity

The given Python code defines a function `maximumAverageSubtree` that calculates the maximum average value of all subtrees in a binary tree. The `dfs` function, a helper function, is used to perform a depth-first search on the tree.

- **Time Complexity:**

The time complexity for this code is determined by the need to visit each node of the binary tree exactly once during the DFS traversal. Since every node is visited exactly once, and at every node we perform a constant amount of work (calculating sums and the number of nodes, and updating the maximum average), the time complexity is $O(N)$, where N is the number of nodes in the binary tree.

- **Space Complexity:**

The space complexity is a bit more tricky to analyze due to the recursive nature of the DFS. In the worst case, the space complexity is defined by the height of the tree, which in the case of a skewed tree (where every node has only one child) could be $O(N)$. However, for a balanced tree, the height (thus the maximum depth of the recursive call stack) would be $O(\log N)$. Thus, the space complexity of the code is $O(H)$, where H is the height of the tree. In the average case for a reasonably balanced tree, this can be expressed as $O(\log N)$, but in the worst-case scenario (a completely unbalanced tree), the space complexity would be $O(N)$.

To summarize:

- Time Complexity: $O(N)$
- Space Complexity: $O(H)$, which would be $O(\log N)$ in a balanced tree and $O(N)$ in the worst case.