637. Average of Levels in Binary Tree

Easy Tree Depth-First Search Breadth-First Search Binary Tree

Leetcode Link

Problem Description

The task is to calculate the average value of all the nodes on each level of a binary tree. A binary tree is a data structure where each node has at most two children, referred to as the left and the right child. In this context, a level refers to a set of nodes that are the same number of steps from the root. For example, the root itself is at level 0, its children are at level 1, the children's children are at level 2, and so on.

The input will be the root node of a binary tree, and the output should be a list of floating-point numbers, where each number represents the average value of the nodes at a corresponding level.

The problem states that the answer will be considered correct if it's within 10^{-5} of the actual average. This means our solution does not need to be exact down to the last decimal place, allowing for a small margin of error, potentially due to floating-point precision issues.

Intuition

To find the average value of nodes at each level, we can use a technique called breadth-first search (BFS). This approach involves traversing the tree level by level from top to bottom and left to right and computing the average of the nodes' values at each level before moving on to the next one.

Here's how we can arrive at the solution:

1. We start by initializing a queue data structure and put the root node in it.
2. The queue will help us process all nodes level by level. For each level:
   - We compute the total number of nodes at that level (this is the same as the number of elements currently in the queue).
   - We then iterate over these elements, summing up their values and adding their children (if any) to the back of the queue to be processed in the next round.
   - After processing all nodes in the current level, we calculate the average value by dividing the sum of the nodes' values by the total number of nodes.
   - We append this average to our answer list.
3. Once the queue is empty, all levels have been processed, and our answer list contains the average of each level.
4. We return the answer list.

Using a queue in this manner ensures that we process all nodes at a given level before moving on to the next, enabling us to calculate the average per level efficiently.

A note on the implementation: In Python, we use `deque` from the `collections` module, as it provides an efficient way to pop elements from the left of the queue and to append elements to the right of the queue with $O(1)$ time complexity for each operation, which is essential for maintaining the overall efficiency of the BFS algorithm.

Solution Approach

The implementation follows the BFS pattern, using a queue to track nodes at each level. Here are the steps taken, matched with the code provided:

1. **Queue Initialization:** The queue is initialized with `q = deque([root])`, which will contain all nodes from the current level that we need to process.
2. **Level Traversal:** We use a while-loop `while q:` to continue processing nodes level by level until there are no more nodes left to process.
3. **Level Processing:** Within the while-loop, we perform these actions:
   - `n, s = 0, len(q)` We start by storing the current level size `n` (which represents the number of nodes at the current level) and initialize a sum variable `s` to collect the sum of the nodes' values.
   - A for-loop is used to iterate over each node in the current level: `for _ in range(n):`. For each node, we use `q.popleft()` to pop the leftmost node from the queue (representing the next node in the current level).
4. **Node Processing and Child Enqueueing:** Within the for-loop:
   - We increment the sum with the current node's value: `s += root.val`.
   - We check for children of the current node and enqueue them:
     - If a left child exists: `if root.left:`, we append it to the queue `q.append(root.left)`.
     - Similarly, for a right child: `if root.right:`, we append it to the queue `q.append(root.right)`.

Each iteration of the for-loop processes one node from the current level until all nodes at that level have been handled.

5. **Average Calculation and Storage:** After all nodes of the current level are processed, we calculate the average by dividing the sum `s` by the number of nodes `n`: `ans.append(s / n)`. The result is then appended to the answer list `ans`.
6. **Continuation and Completion:** Once the for-loop finishes, the while-loop iterates again if there are any nodes left in the queue (these would be nodes from the next level that were enqueued during the for-loop). This process repeats until all levels are processed.
7. **Returning the Result:** After all levels have been iterated over and the while-loop exits, the answer list `ans`, now containing the averages of each level, is returned.

The solution uses a simple but effective linear-time algorithm that processes each node exactly once, giving an overall time complexity of $O(m)$, where m is the number of nodes in the tree. The space complexity is $O(m)$, where m is the maximum number of nodes at any level in the input tree (in the worst case, n could be up to $n/2$ for a full level in a perfectly balanced tree).

Example Walkthrough

Let's illustrate the solution approach with a concrete example. Consider the following binary tree:

```
    3
   / \
  9   20
     /  \
    15   7
```

We want to calculate the average value of all the nodes on each level of this tree.

1. **Queue Initialization:** Initialize the queue with the root node (which is 3).

   `q = deque([3])`

2. **Level Traversal:** The while-loop begins since the queue is not empty.

   `while q:`

3. **Level Processing:** For the first level (level 0 with the root node), we store the current queue size, which is 1, and the sum variable is initialized to 0.

   `s, n = 0, len(q)` → `s, n = 0, 1`

   We then enter a for-loop to process this level's nodes.

4. **Node Processing and Child Enqueueing:** Pop the leftmost node (which is 3) and add its value to the sum.

   `s += 3`

   Since node 3 has two children (9 and 20), we enqueue those children.

   ```
   q.append(9)
   q.append(20)
   ```

5. **Average Calculation and Storage:** At the end of the level, we calculate the average by dividing the sum by the number of nodes, which in this case is just 3 / 1.

   `ans.append(3.0)` → `ans = [3.0]`

6. **Continuation and Completion:** The queue now contains two nodes for the next level (9 and 20). The while-loop continues for the next iteration.

   For the second level (level 1), we again calculate the total nodes and sum.

   `s, n = 0, len(q)` → `s, n = 0, 2`

   Pop each node, add its value, and enqueue any children (15 and 7 are enqueued).

   ```
   s += 9
   s += 20
   q.append(15)
   q.append(7)
   ```

   Calculate and store the average for this level (29 / 2).

   `ans.append(14.5)` → `ans = [3.0, 14.5]`

   The queue now contains the nodes for the next level (15 and 7), so the while-loop iterates again.

   For the third level (level 2), we execute a similar process.

   `s, n = 0, len(q)` → `s, n = 0, 2`

   No more children are enqueued at this level because neither 15 nor 7 have children.

   Calculate and store the average (22 / 2).

   `ans.append(11.0)` → `ans = [3.0, 14.5, 11.0]`

7. **Returning the Result:** Now the queue is empty, and the while-loop exits. We return the list `ans`, which contains `[3.0, 14.5, 11.0]`. This list represents the average value of the nodes at each level of the given binary tree.

Python Solution

```python
from collections import deque

class TreeNode:
    # Definition for a binary tree node.
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def averageOfLevels(self, root: TreeNode) -> List[float]:
        # Initialize a queue for breadth-first traversal with the root node
        queue = deque([root])
        # List to store the averages of each level
        averages = []

        # Loop while there are nodes in the queue
        while queue:
            # Initialize the sum and get the number of nodes at the current level
            level_sum, num_nodes = 0, len(queue)
            # Iterate over all nodes in the current level
            for _ in range(num_nodes):
                # Pop the first node from the queue
                node = queue.popleft()
                # Add the node's value to the level sum
                level_sum += node.val
                # If there is a left child, add it to the queue
                if node.left:
                    queue.append(node.left)
                # If there is a right child, add it to the queue
                if node.right:
                    queue.append(node.right)
            # Calculate the average for the level and add it to the result list
            averages.append(level_sum / num_nodes)

        return averages
```

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 {

    /**
     * Computes the average of each level of a binary tree.
     * @param root Root of the binary tree.
     * @return List of averages of each level.
     */
    public List<Double> averageOfLevels(TreeNode root) {
        List<Double> averages = new ArrayList<>(); // Stores the averages of each level
        Deque<TreeNode> queue = new ArrayDeque<>(); // Queue for traversing the tree level by level
        queue.offer(root); // Start with the root

        while (!queue.isEmpty()) {
            int levelSize = queue.size(); // Number of nodes in the current level
            long sum = 0; // Sum of values of nodes in the current level
            // Iterate over all nodes in the current level
            for (int i = 0; i < levelSize; ++i) {
                TreeNode currentNode = queue.poll(); // Get and remove the node from the queue
                sum += currentNode.val; // Add the node's value to the sum
                // Enqueue child nodes for the next level
                if (currentNode.left != null) {
                    queue.offer(currentNode.left);
                }
                if (currentNode.right != null) {
                    queue.offer(currentNode.right);
                }
            }
            averages.add(sum / levelSize); // Compute and add the average for this level to the result
        }
        return averages; // Return the list of averages
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <queue>

// 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:
    vector<double> averageOfLevels(TreeNode* root) {
        queue<TreeNode*> nodeQueue; // Queue to hold the nodes at the current level.
        nodeQueue.push(root); // Initialize the queue with the root node.
        vector<double> averages; // Vector to store the average values of each level.

        while (!nodeQueue.empty()) {
            int levelSize = nodeQueue.size(); // Number of nodes at the current level.
            long long levelSum = 0; // Sum of the values of nodes at the current level. Use long long to avoid integer overflow.

            // Iterate over all nodes at the current level.
            for (int i = 0; i < levelSize; ++i) {
                TreeNode* currentNode = nodeQueue.front(); // Retrieve the front node in the queue.
                nodeQueue.pop(); // Remove the node from the queue.
                levelSum += currentNode->val; // Add the node value to the level sum.

                // If the left child exists, add it to the queue for the next level.
                if (currentNode->left) {
                    nodeQueue.push(currentNode->left);
                }
                // If the right child exists, add it to the queue for the next level.
                if (currentNode->right) {
                    nodeQueue.push(currentNode->right);
                }
            }

            // Calculate the average for the current level and add it to the result vector.
            // Casting levelSum to double to get floating point division.
            averages.push_back(static_cast<double>(levelSum) / levelSize);
        }

        return averages; // Return the vector containing averages of each level.
    }
};
```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

/**
 * Computes the average of the node values at each level of a binary tree.
 * @param {TreeNode | null} root - The root of the binary tree.
 * @return {number[]} - An array of averages, one for each level of the tree.
 */
function averageOfLevels(root: TreeNode | null): number[] {
    const nodeQueue: TreeNode[] = [root]; // Initialize the queue with the root node.
    let average: number[] = []; // Initialize the array to hold averages of each level.

    while (nodeQueue.length > 0) {
        const levelSize: number = nodeQueue.length; // The number of nodes at current level.
        let sum: number = 0; // Initialize sum of values at current level.

        // Process all nodes at the current level.
        for (let i = 0; i < levelSize; i++) {
            const currentNode = nodeQueue.shift(); // Remove the next node from queue.
            if (currentNode) {
                sum += currentNode.val; // Add the value of the current node to the level sum.

                // If the current node has a left child, add it to the queue.
                if (currentNode.left) {
                    nodeQueue.push(currentNode.left);
                }

                // If the current node has a right child, add it to the queue.
                if (currentNode.right) {
                    nodeQueue.push(currentNode.right);
                }
            }
        }

        // Calculate and store the average for this level and add it to the result array.
        averages.push(sum / levelSize); // Compute and add the average to the result array.
    }

    return averages; // Return the array of averages.
}

// Example usage:
// let tree = new TreeNode(3, new TreeNode(9), new TreeNode(20, new TreeNode(15), new TreeNode(7)));
// let result = averageOfLevels(tree);
// console.log(result); // Output: [3, 14.5, 11]
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is $O(m)$, where m is the number of nodes in the binary tree. This is because the algorithm uses a queue to traverse each node exactly once. During the traversal, each node's value is accessed and added to a sum, and its children are potentially added to the queue.

Space Complexity

The space complexity of the code can be considered as $O(w)$, where w is the maximum width of the tree or the maximum number of nodes at any level of the tree. This occurs because the queue stores a level of the tree at most, which, in the worst case, can be all the leaf nodes of a full binary tree at the last level.