662. Maximum Width of Binary Tree

Medium | Tree | Depth-First Search | Breadth-First Search | Binary Tree | Leetcode Link

Problem Description

The problem presents a binary tree and requires finding the maximum width of the tree. The *maximum width* refers to the largest number of nodes between the leftmost and rightmost non-null nodes across all levels of the tree, including the null nodes that would be present if the tree were a complete binary tree down to that level.

Imagine looking at each level of the binary tree and stretching a rubber band from the first (leftmost) to the last (rightmost) node that exists at that level, specifying to also include the places where a node would be if the tree were complete. The width is the length of this rubber band, and our task is to find the longest one.

The problem guarantees that the maximum width can be stored within a 32-bit signed integer, which specifies an upper limit to the size of our answer.

Intuition

To solve this problem, we can use either Breadth-First Search (BFS) or Depth-First Search (DFS). The solution provided here uses BFS, which is often a natural choice for problems involving levels of a tree.

The intuition behind using BFS is that we process nodes level by level. At each level, we can measure the width by keeping track of the position of each node. We can assign an index to each node as if it were in a complete binary tree. This is done by numbering the root as 1, the left child as 2*i, and the right child as 2*i+1 (where i is the index of the current node).

We utilize a queue to keep track of nodes along with their indices. For each level, we calculate the width as the difference between the indices of the first and last nodes in the queue plus 1 (to account for the first node). We then update our answer to be the maximum width found so far.

Note that these indices can become very large since they double at each level, but we are only interested in the width (the difference between indices), not the indices themselves. Thus, we can maintain relative indices at each level to avoid overflow issues and still correctly calculate the width.

Solution Approach

The problem is solved using a breadth-first search (BFS), which is a common algorithm for traversing or searching tree or graph data structures. This approach works level by level, visiting all the nodes at the current depth before moving to the next.

Here is how the BFS solution is implemented:

1. **Queue Initialization**: A queue data structure is used to keep track of the nodes along with their indices. The queue is initialized by adding the root node with an index of `1`. This index assumes that the root node is at position `1` in a hypothetical complete binary tree.
2. **Level-wise Traversal**: A `while` loop is used to iterate over the tree level by level as long as the queue is not empty.
3. **Finding Width**: At the start of each level, the current width is calculated by subtracting the index of the first node in the queue (`q[0][1]`) from the index of the last node in the queue (`q[-1][1]`) and adding `1` (to include the first node in the count). The calculation `ans = max(ans, q[-1][1] - q[0][1] + 1)` updates the maximum width found so far.
4. **Index Assignment**: Inside the loop, another loop iterates over the nodes present at the current level (determined by the current size of the queue). For each node, if it has a left child, the left child is given an index `i << 1` (equivalent to `2*i`), and if it has a right child, the right child is given an index `i << 1 | 1` (equivalent to `2*i+1`), aligning with the indexing for a complete binary tree.
5. **Queue Updates**: After calculating the width and assigning new indices, nodes are dequeued (`popleft`), and their children are enqueued with their respective calculated indices.
6. **Returning the Result**: Once the BFS traversal is complete, and the queue is emptied, the algorithm returns `ans`, the maximum width found across all levels of the input binary tree.

Throughout the BFS, deque from the Python collections module is used as an efficient queue implementation allowing constant time complexity for adding and removing nodes. Also, the tree nodes are presumed to be defined as a class with attributes `val`, `left`, and `right`. The solution handles tree nodes containing integer values and may have left and right children.

The bit manipulation used to assign indices (`i << 1` and `i << 1 | 1`) is a performance optimization, as bit shifts are generally faster than multiplication or addition. This logic also ensures correctness when assigning positions to child nodes while keeping track of the overall structure for the width computation.

By structuring the BFS to consider indices as if they are part of a complete binary tree, the maximum width can be determined even if the actual tree is not complete, accounting for the non-null nodes and the gaps between them.

Example Walkthrough

Let's consider a binary tree for our example:

```
     1
    / \
   3   2
  /     \
 5       9
```

We will apply the BFS approach as described in the solution approach to find the maximum width of this tree.

1. **Queue Initialization**: We start by initializing a queue with the root node (`1`) and its index. The root is indexed `1`. So the queue will have one element: `[(1, 1)]`.
2. **Level-wise Traversal**: Now, we begin the level-by-level traversal. The queue is not empty, so we enter the `while` loop.
3. **Finding Width**: For this level, which is just the root, the width will be `q[-1][1] - q[0][1] + 1`, meaning `1 - 1 + 1 = 1`. This is now our current maximum width, `ans = 1`.
4. **Index Assignment**: There are no other nodes at this level, so we move to the children. Node `3` is given index `2*1 = 2`, and node `2` is given index `2*1+1 = 3`.
5. **Queue Updates**: We remove the root node from the queue and add its children with their indexes, so the queue now has `[(3, 2), (2, 3)]`.
6. **Traverse Second Level**: Now we process the second level. The width at this level would be `3 - 2 + 1 = 2`. We update our maximum width `ans = max(1, 2) = 2`.
7. **Index Assignment & Queue Updates for Second Level**: Node `5` gets an index `2*2 = 4`, and node `9` gets an index `2*3+1 = 7`. Our queue is updated to contain `[(5, 4), (9, 7)]`.
8. **Traverse Third Level**: At this level, we calculate the width using the indexes in the queue, `7 - 4 + 1 = 4`. We update `ans = max(2, 4) = 4`.

Since nodes `5` and `9` do not have children, the loop ends, as the queue is now empty.

9. **Returning the Result**: The maximum width found during traversal is `4`, which is the final answer for the maximum width of our example binary tree.

With this process, we have measured the maximum width of each level of the example tree and kept track of the largest value found, which is our answer to the problem. The width of the tree is the length of the longest rubber band we could stretch across any level, including both present and 'imaginary' (null) nodes, and for our example tree, that width is `4`.

Python Solution

```python
from collections import deque

# 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 widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        # Initialize the answer to zero, which will hold the maximum width
        max_width = 0

        # Queue will hold tuples of tree node and its index in the tree
        queue = deque([(root, 1)])

        # Continue the loop until the queue is empty
        while queue:
            # Calculate the current width using the first and last node's indices
            current_width = queue[-1][1] - queue[0][1] + 1
            max_width = max(max_width, current_width)  # Update maximum width

            # Iterate over the current level of the tree
            for _ in range(len(queue)):
                # Pop the node and its index from the queue
                node, index = queue.popleft()

                # If the left child exists, add it to the queue
                if node.left:
                    queue.append((node.left, index << 1))

                # If the right child exists, add it to the queue
                if node.right:
                    queue.append((node.right, (index << 1) | 1))

        # Return the maximum width found
        return max_width

```

Java Solution

```java
import javafx.util.Pair; // Ensure Pair class is imported, JavaFX or Apache Commons Lang (or define your own Pair class if needed)

/**
 * 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 {
    public int widthOfBinaryTree(TreeNode root) {
        // Queue to hold nodes and their respective indices in the tree
        Deque<Pair<TreeNode, Integer>> queue = new ArrayDeque<>();
        // Start with the root node and index 1
        queue.offer(new Pair<>(root, 1));
        // Initialize the maximum width
        int maxWidth = 0;

        // Loop until there are no more nodes to process
        while (!queue.isEmpty()) {
            // Calculate the width of the current level
            // The width is the difference in indices between the first and last nodes + 1
            maxWidth = Math.max(maxWidth,
                                queue.peekLast().getValue() - queue.peekFirst().getValue() + 1);

            // Iterate over all nodes at the current level
            int levelSize = queue.size();
            for (int i = 0; i < levelSize; ++i) {
                // Poll the current node and its index
                Pair<TreeNode, Integer> current = queue.pollFirst();
                TreeNode currentNode = current.getKey();
                int index = current.getValue();

                // Offer left child to queue if it exists
                if (currentNode.left != null) {
                    queue.offer(new Pair<>(currentNode.left, 2 * index));
                }
                // Offer right child to queue if it exists
                if (currentNode.right != null) {
                    queue.offer(new Pair<>(currentNode.right, 2 * index + 1));
                }
            }
        }
        // Return the maximum width found
        return maxWidth;
    }
}

```

C++ Solution

```cpp
#include <queue>
#include <algorithm>

// 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:
    int widthOfBinaryTree(TreeNode* root) {
        if (!root) return 0; // Guard clause for an empty tree

        std::queue<std::pair<TreeNode*, unsigned long>> queue; // Queue to perform level order traversal, holding nodes and their p
        queue.push({root, 1}); // Initialize with the root node with position 1
        int maxWidth = 0; // Variable to store the maximum width found

        while (!queue.empty()) {
            int levelSize = queue.size(); // Number of nodes at the current level
            unsigned long leftMost = queue.front().second; // Position of the leftmost node at the current level, used as an offset
            unsigned long rightMost = queue.back().second; // Position of the rightmost node at the current level, used to calculat
            maxWidth = std::max(maxWidth, static_cast<int>(rightMost - leftMost) + 1); // Update maxWidth if necessary

            // Iterate through the nodes of the current level
            for (int i = 0; i < levelSize; ++i) {
                auto nodePair = queue.front();
                queue.pop();
                TreeNode* currentNode = nodePair.first;
                unsigned long position = nodePair.second;

                // Subtract leftMost to avoid high values and overflow due to increasing node positions
                position -= leftMost;

                // Add children to the queue with their respective positions
                if (currentNode->left) {
                    queue.push({currentNode->left, 2 * position}); // Left child position
                }
                if (currentNode->right) {
                    queue.push({currentNode->right, 2 * position + 1}); // Right child position
                }
            }
        }
        return maxWidth; // Return the maximum width found
    }
};

```

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;
    }
}

// Function to compute the width of a binary tree.
function widthOfBinaryTree(root: TreeNode | null): number {
    // If the tree is empty, return width 0.
    if (!root) return 0;

    // Queue for level order traversal, holding nodes and their positions.
    const queue: Array<{ node: TreeNode; position: number }> = [];
    queue.push({ node: root, position: 1 });
    let maxWidth = 0; // Variable to store the maximum width.

    // While there are nodes in the queue to process...
    while (queue.length > 0) {
        const levelSize = queue.length; // Number of nodes at the current level.
        const leftMost = queue[0].position; // Position of the leftmost node at the current level (used as an offset).

        // Iterate through the nodes of the current level.
        for (let i = 0; i < levelSize; i++) {
            const { node: currentNode, position: currentPosition } = queue.shift()!;

            // On the last iteration, update maxWidth using currentPosition as the rightMost position.
            if (i === levelSize - 1) {
                maxWidth = Math.max(maxWidth, currentPosition - leftMost + 1);
            }

            // Subtract leftMost to avoid high values and potential overflow.
            const adjustedPosition = currentPosition - leftMost;

            // Add children to the queue with their respective positions.
            if (currentNode.left) {
                queue.push({ node: currentNode.left, position: 2 * adjustedPosition });
            }
            if (currentNode.right) {
                queue.push({ node: currentNode.right, position: 2 * adjustedPosition + 1 });
            }
        }
    }

    return maxWidth; // Return the calculated maximum width of the binary tree.
}

// Example usage:
// const root = new TreeNode(1, new TreeNode(3, new TreeNode(5), new TreeNode(3)), new TreeNode(2, null, new TreeNode(9)));
// console.log(widthOfBinaryTree(root)); // Outputs the width of the tree.

```

Time and Space Complexity

Time Complexity

The time complexity of the function is determined by how many times we visit each node in the binary tree. In this code, every node is visited exactly once during the breadth-first search, so the time complexity is `O(N)`, where `N` is the number of nodes in the binary tree.

Space Complexity

The space complexity depends on the maximum size of the queue used to store nodes at any level of the tree. In the worst case, when the tree is a complete binary tree, the maximum width is reached at the last level, which would contain approximately `N/2` nodes (half of the total number of nodes). Therefore, the space complexity is `O(N)` in the worst case.