559. Maximum Depth of N-ary Tree

Easy Tree Depth-First Search Breadth-First Search Leetcode Link

Problem Description

The given problem focuses on determining the maximum depth of a n-ary tree. In a tree data structure, the depth represents the length of the path from the root node to the farthest leaf node. For a n-ary tree, which means that each node can have an arbitrary number of children, we define the tree's maximum depth as the largest number of steps it takes to travel from the root node to a leaf node. It should be noted that the root node itself is counted as a step. In this context, a leaf node is a node with no children. The tree is provided in a serialized form that reflects its level-order traversal where groups of children nodes are separated by a `null` value.

Intuition

The solution relies on a depth-first search (DFS) approach, where we recursively explore each subtree rooted at each child of the current node. The base case for the recursive function is when it encounters a `None` value, which indicates an empty node or the end of a branch, and therefore it should return 0 as there are no further nodes down that path.

When the recursive function is called on a non-empty node, it computes the maximum depth of all subtrees rooted at each of its children. This is done by calling the recursive function for each child node and collecting these depths into a list. We then obtain the maximum depth from these subtrees and add 1 to it to account for the current node. The `+1` represents that the path from the current node to the deepest leaf in each subtree is one step longer because of the current node.

If a node does not have any children, the max function would return a default value of 0. When we have the maximum depths of all children (if any), we select the maximum value among them, add 1 to represent the current node, and return this number. This recursive process continues until the root of the tree is reached, at which point we obtain the maximum depth of the entire n-ary tree. The recursive nature of this approach effectively navigates through all possible paths and finds the length of the longest one, thus solving the problem.

Solution Approach

The implementation begins with checking if the `root` node is `None`. This check ensures that if the tree is empty (hence, there is no tree whatsoever), the function returns 0. After bypassing the base case, the solution uses a depth-first search (DFS) strategy to find the maximum depth.

In the context of a n-ary tree where nodes can have multiple children, each child node can potentially branch out to a tree that is a subtree. For each child node, we are interested in finding the maximum depth from that child down to its farthest leaf node.

We use a list comprehension to apply the `maxDepth` function recursively on each of the child nodes of the root:

```
[self.maxDepth(child) for child in root.children]
```

This recursively calculates the depth for each subtree rooted at `child`. As we are interested in the maximum depth, we then use the `max` function to find the maximum value in this list. Since it's possible for a node to have no children, leading to an empty list, we specify a default value (which is 0) to the `max` function. This default value is returned when the `max` function is called on an empty list:

```
max([self.maxDepth(child) for child in root.children], default=0)
```

It should be noted that if a node does have children, each recursive `maxDepth` call will return some integer greater than or equal to 1, since each node itself is counted as one step.

After finding the maximum depth among all children, we need to include the current node in the calculation, so we add 1 to this maximum value:

```
return 1 + max([self.maxDepth(child) for child in root.children], default=0)
```

This addition accounts for the current node as one more step on the path from the node to the deepest leaf. The recursive nature of this function ensures that we explore each node and its subsequent children to exhaustively determine the maximum depth of the n-ary tree.

In summary, the algorithm employs a recursive depth-first search strategy, efficiently calculates the depths of all subtrees of the n-ary tree, and elegantly combines those values to determine the overall maximum depth. This approach neatly uses Python's list comprehension, recursive function calls, and the `max` function with the `default` parameter to handle the specifics of a n-ary tree structure.

Example Walkthrough

Let's visualize the solution approach using a simple n-ary tree. Consider the following n-ary tree:

```
     1
    / \
   2   3
  /|   |\
 4 5   6 7
```

Here's how the solution approach works on this tree:

- Start with the root node (`1`). Since it is not `None`, proceed to its children.
- Apply the `maxDepth` function to the children of `1`, which are nodes `2` and `3`.
  - For node `2`, it has children `4` and `5`. The `maxDepth` will be called on each of them.
    - When `maxDepth` is called on `4`, which is a leaf node, it will return `1`.
    - Similarly, calling `maxDepth` on `5` will also return `1`.
    - The maximum depth of the children of node `2` will be `max([1, 1], default=0)`, which is `1`.
    - Since node `2` is one step itself, add `1` to the maximum depth of its children, giving us a total of `1 + 1 = 2`.
  - For node `3`, apply the same logic. It has children `6` and `7`, which are leaf nodes.
    - The `maxDepth` function will return `1` for both `6` and `7`.
    - The maximum depth of the children of node `3` will be `max([1, 1], default=0)`, which is `1`.
    - Including node `3` itself, the total depth is `1 + 1 = 2`.
- Now, consider the depths calculated for nodes `2` and `3`. Both are `2`.
- For the `root` node, the maximum depth will be `max([2, 2], default=0)`, which is `2`.
- The final step is to include the root node itself in the depth count, resulting in `1 + 2 = 3`.

Therefore, the maximum depth of the tree is `3`. This walk-through exemplifies how the recursive depth-first search systematically computes the depth of each subtree before combining the results to find the maximum depth of the entire tree.

Python Solution

```python
class Node:
    """
    A class representing a node in an n-ary tree.
    """

    def __init__(self, value=None, children=None):
        """
        Initialize a Node with a value and children.

        Params:
        value - The value of the node (an integer or None by default)
        children - A list of child nodes (an empty list by default if None)
        """
        self.value = value
        self.children = children if children is not None else []


class Solution:
    def maxDepth(self, root: 'Node') -> int:
        """
        Calculate the maximum depth of an n-ary tree.

        Params:
        root - The root node of the n-ary tree.

        Returns:
        An integer representing the maximum depth of the tree.
        """
        # If the root node is None, the depth is 0
        if root is None:
            return 0

        # If the root node has no children, the maximum depth is 1
        if not root.children:
            return 1

        # Calculate the depth of each subtree and find the maximum.
        # For each child node, calculate the max depth recursively
        # Add 1 to account for the current node's level
        max_depth = 0
        for child in root.children:
            child_depth = self.maxDepth(child)
            max_depth = max(max_depth, child_depth)

        # Return the maximum depth found among all children, plus 1 for the root
        return 1 + max_depth

```

Java Solution

```java
import java.util.List;

// Class definition for a Node in an N-ary tree
class Node {
    public int val;                         // Node's value
    public List<Node> children;          // List of Node's children

    public Node() {}

    // Constructor with node's value
    public Node(int val) {
        this.val = val;
    }

    // Constructs node with value and list of children
    public Node(int val, List<Node> children) {
        this.val = val;
        this.children = children;
    }
}

class Solution {
    // Method to calculate the maximum depth of an N-ary tree
    public int maxDepth(Node root) {
        // An empty tree has a depth of 0.
        if (root == null) {
            return 0;
        }

        int maxDepth = 0; // Initialize max depth as 0

        // Loop through each child of root node
        for (Node child : root.children) {
            // Calculate the depth for each child and compare it with current max depth
            // 1 is added to include the current node's depth
            maxDepth = Math.max(maxDepth, 1 + maxDepth(child));
        }

        // Since we're already at root, we add 1 to account for the root's depth
        return maxDepth + 1;
    }
}

```

C++ Solution

```cpp
#include <algorithm> // Include algorithm library for max function
#include <vector>    // Include vector library for the vector type

// Definition for a Node.
class Node {
public:
    int val;                       // Value of the node
    std::vector<Node*> children; // Vector of pointers to child nodes

    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, std::vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};

class Solution {
public:
    // Function to calculate the maximum depth of an n-ary tree
    int maxDepth(Node* root) {
        // If the root is null, the depth is 0
        if (!root) return 0;

        int max_depth = 1; // Initialize max_depth to 1 to account for the root level

        // Iterate through each child of the root node
        for (auto& child : root->children) {
            // Recursive call to maxDepth for each child, adding 1 to the result,
            // then update max_depth with the maximum of it and the current max_depth
            max_depth = std::max(max_depth, 1 + maxDepth(child));
        }

        // Return the maximum depth of the tree
        return max_depth;
    }
};

```

Typescript Solution

```typescript
// Definition for a Node.
class Node {
    val: number;                    // Value of the node
    children: Node[];               // Array of child nodes

    constructor(val: number, children: Node[] = []) {
        this.val = val;
        this.children = children;
    }
}

// Function to calculate the maximum depth of an n-ary tree
function maxDepth(root: Node | null): number {
    // If the root is null, the depth is 0
    if (!root) return 0;

    let maxDepthValue = 1; // Initialize maxDepthValue to 1 to account for the root level

    // Iterate through each child of the root node
    root.children.forEach(child => {
        // Recursive call to maxDepth for each child, adding 1 to the result,
        // then update maxDepthValue with the maximum of it and the current maxDepthValue
        maxDepthValue = Math.max(maxDepthValue, 1 + maxDepth(child));
    });

    // Return the maximum depth of the tree
    return maxDepthValue;
}

```

Time and Space Complexity

The provided code defines a function to determine the maximum depth (height) of an N-ary tree using recursion. Here's the analysis:

Time Complexity

The time complexity of the function is `O(N)`, where `N` is the total number of nodes in the N-ary tree. This is because the function must visit every node exactly once to determine the depth. The recursive calls are made for each child of every node. Since each node is processed once and only once, the time complexity is linear with respect to the number of nodes.

Space Complexity

The space complexity of the function is also `O(N)` in the worst-case scenario. This happens when the tree is highly unbalanced, for example, when the tree degenerates into a linked list (each node has only one child). In such a case, there will be `N` recursive calls on the stack at the same time, where `N` is the depth of the tree, which is also the number of nodes in this case.

In a balanced tree, the space complexity would be `O(log N)` due to the height of the tree dictating the number of stack frames. However, since it is not mentioned that the tree is balanced, the worst-case space complexity is considered which is `O(N)`.