1522. Diameter of N-Ary Tree

Medium Tree Depth-First Search Leetcode Link

Problem Description

The problem involves finding the diameter of an N-ary tree. The diameter is defined as the length (number of edges) of the longest path between any two nodes within the tree. A key point to note is that this path is not required to go through the tree's root. Understanding how to handle N-ary trees, which are trees where a node can have any number of children, is crucial in solving this problem. Level order traversal serialization (where each group of children is separated by the null value) is used for input representation.

Intuition

When finding the diameter of an N-ary tree, a depth-first search (DFS) approach can be applied effectively. The diameter of the tree could potentially be the distance between two nodes that are below the root of the tree, which means it might not include the root itself. Therefore, for each node, we need to consider the two longest paths that extend from this node down its children, because these paths could potentially contribute to the maximum diameter if they are connected through the current node.

To implement this, we use a recursive DFS function that computes and returns the height of the current subtree while simultaneously calculating the potential diameter that passes through the current node (the sum of the two longest child paths connected at the node). We track the overall longest path seen so far with a nonlocal variable 'ans', which gets updated when a larger diameter is found.

The key steps in our approach are:

1. Traverse each node with DFS, starting from the root.
2. For each node visited, calculate the lengths of the longest (max) and second-longest (smax) paths among its child nodes.
3. Update the overall potential diameter 'ans' if the sum of max and smax is greater than the current 'ans'.
4. At each node, return the height of this subtree to its parent, which is 1 plus the longest path length among the child nodes.

By doing this for each node, once the traversal is complete, 'ans' will hold the length of the longest path, which is the diameter of the N-ary tree.

Solution Approach

The solution approach for calculating the diameter of an N-ary tree involves a depth-first search (DFS) algorithm. The DFS is customized to perform two operations at once – calculate the height of the current subtree and use this information to evaluate the longest path that passes through each node (potential diameters).

Here's a step-by-step breakdown of the implementation using the provided solution code:

1. We define a DFS helper function `dfs` that takes a node of the tree as an argument. This function returns the height of the tree rooted at the given node.

2. Inside the `dfs` function:

   - We start by checking if the current `root` node is `None` (base case). If it is, we return 0 since a non-existent node does not contribute to height or diameter.

   - We then introduce two variables `m1` and `m2` initialized to 0, which will store the lengths of the longest (`m1`) and second-longest (`m2`) paths found within the children of the current node.

   - We loop through each child of the current node and recursively call `dfs(child)`. The returned value `t` is the height of the subtree rooted at `child`. We use this value to potentially update `m1` and `m2`.

     - If `t` is greater than `m1`, we set `m2` to `m1` and then `m1` to `t`, thus keeping `m1` as the max height and `m2` as the second max height among the children.

     - If `t` is not greater than `m1` but is greater than `m2`, we update `m2` with `t`.

   - After considering all children, we update the `ans` variable (which is kept in the outer scope of `dfs` and declared as `nonlocal`) with the maximum of its current value and the sum of `m1 + m2`, representing the potential diameter at this node (the longest path through the node).

3. The `dfs` function concludes by returning `1 + m1`, which represents the height of the subtree rooted at the current node (1 for the edge connecting to its parent and `m1` as the height of its longest subtree).

4. In the `diameter` method of the `Solution` class, we initialize the `ans` variable to 0 (to globally keep track of the longest path seen) and call `dfs(root)` to kick off the recursive DFS from the root of the tree.

5. Finally, the `diameter` method returns the `ans` variable, which now contains the diameter of the tree.

The use of recursion to traverse the tree, combined with updating the two longest paths at each node, harnesses the depth-first search pattern efficiently to solve the problem of finding the tree's diameter.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have an N-ary tree represented using level order serialization as follows:

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

We want to find the diameter of this tree, which is the longest path between any two nodes. We will walk through the DFS approach using the provided solution template:

1. We start the DFS with the `root`, which is node 1. We initialize `ans` to 0. The `dfs` function will traverse the tree and track the longest path at each node.

2. When we process the `root` (node 1), we initialize `m1` and `m2` both to 0. These variables will track the longest and second-longest heights among the children of each node.

3. We recursively call `dfs` for each child of the `root`.

   - For child node 2, the recursive call to `dfs` returns 1 (since node 5 is its only child and adds one edge to the height).
   - For child node 3, the recursive call to `dfs` returns 0 (as it has no child).
   - For child node 4, the DFS must consider both children. Nodes 6 and 7 contribute a height of 1 each, and since these are the longest and second-longest heights for this subtree, after considering both, node 4 returns 2 as the height (1 for the longest child plus 1 for the edge to node 4).

4. Now, with the heights from the children of the `root`, we update `m1` and `m2` for the `root`. `m1` becomes 2 (from node 4), and `m2` becomes 1 (from node 2). Therefore, the potential diameter passing through `root` at this stage is `m1 + m2 = 2 + 1 = 3`. We update `ans` to 3.

5. The DFS continues recursively for all nodes but finds no longer path than the current `ans`.

6. After the DFS is complete and all nodes have been visited, `ans` holds the diameter of the tree. In this case, it remains 3, representing the length of the longest path, which happens to be from node 5 to node 7 via the `root` (5 → 2 → 1 → 4 → 6 or 7).

7. The `diameter` method then returns `ans`, which is 3, as the diameter of the tree.

This walkthrough illustrates the algorithm's efficiency in computing the diameter by determining the longest path through each node without necessarily passing through the root.

Python Solution

```python
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []

class Solution:
    def diameter(self, root: 'Node') -> int:
        # Helper function to perform depth-first search
        def dfs(node: 'Node') -> int:
            # base case: if the current node is None, return 0
            if node is None:
                return 0
            # accessing the non-local variable 'max_diameter' to update its value within this helper function
            nonlocal max_diameter
            # initialize the two longest paths from the current node to 0
            longest_path = second_longest_path = 0
            # iterate over all the children of the current node
            for child in node.children:
                # recursively find the longest path for each child
                path_length = dfs(child)
                # check if the current path is longer than the longest recorded path
                if path_length > longest_path:
                    # update the second longest and longest paths accordingly
                    second_longest_path, longest_path = longest_path, path_length
                # else if it's only longer than the second longest, update the second longest
                elif path_length > second_longest_path:
                    second_longest_path = path_length
            # update the maximum diameter encountered so far based on the two longest paths from this node
            max_diameter = max(max_diameter, longest_path + second_longest_path)
            # return the longer path increased by 1 for the edge between this node and its parent
            return 1 + longest_path

        # Initialize max_diameter to 0 before starting DFS
        max_diameter = 0
        # Call the dfs function starting from the root node
        dfs(root)
        # Once DFS is complete, return the max_diameter found
        return max_diameter

```

Java Solution

```java
/**
 * Definition for a N-ary tree node.
 */
class Node {
    public int val;
    public List<Node> children;

    // Constructor initializes value and empty children list
    public Node() {
        children = new ArrayList<Node>();
    }

    // Constructor initializes node with a value and empty children list
    public Node(int val) {
        this.val = val;
        children = new ArrayList<Node>();
    }

    // Constructor initializes node with a value and given children list
    public Node(int val, ArrayList<Node> children) {
        this.val = val;
        this.children = children;
    }
}

public class Solution {
    // This class variable keeps track of the diameter of the tree
    private int diameter;

    /**
     * Computes the diameter of an N-ary tree.
     * The diameter of an N-ary tree is the length of the longest path between any two nodes in a tree.
     *
     * @param root the root node of the tree
     * @return the diameter of the tree
     */
    public int diameter(Node root) {
        diameter = 0;
        depthFirstSearch(root);
        return diameter;
    }

    /**
     * Uses Depth First Search (DFS) to find the length of the longest path through the N-ary tree from the current node.
     *
     * @param root the current node being traversed
     * @return the maximum depth emanating from the current node
     */
    private int depthFirstSearch(Node root) {
        // Leaf nodes have a depth of 0
        if (root == null) {
            return 0;
        }

        int maxDepth1 = 0; // Tracks the longest path
        int maxDepth2 = 0; // Tracks the second longest path

        // Recursively obtain the depth for children nodes to find the longest paths
        for (Node child : root.children) {
            int depth = depthFirstSearch(child);

            // Check and set the longest and second-longest distances found
            if (depth > maxDepth1) {
                maxDepth2 = maxDepth1;
                maxDepth1 = depth;
            } else if (depth > maxDepth2) {
                maxDepth2 = depth;
            }
        }

        // Update the diameter if the sum of two longest paths through the current node is greater than the current diameter
        diameter = Math.max(diameter, maxDepth1 + maxDepth2);

        // Return the maximum depth plus one for the current path
        return 1 + maxDepth1;
    }
}

```

C++ Solution

```cpp
// Definition for a Node is provided as per the question context
class Node {
public:
    int val;                    // The value contained within the node.
    vector<Node*> children;     // A vector containing pointers to its children.

    Node() {}

    Node(int _val) {
        val = _val;
    }

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

class Solution {
public:
    int maxDiameter;  // Class variable to store the maximum diameter found.

    // Public method which is the starting point for finding the diameter of the tree.
    int diameter(Node* root) {
        maxDiameter = 0; // Initialize the max diameter to 0.
        dfs(root);       // Call the depth-first search helper method.
        return maxDiameter; // Return the maximum diameter calculated.
    }

    // Private helper method for DFS traversal which calculates the depths and updates the diameter.
    // It returns the maximum depth from the current node to its furthest leaf node.
    int dfs(Node* root) {
        if (!root) return 0; // Base case: If the node is null, return 0 (no depth).

        int maxDepth1 = 0; // To store the maximum length of the paths in the children.
        int maxDepth2 = 0; // To store the second maximum length of the paths in the children.

        // Iterate through each child node of the current root.
        for (Node* child : root->children) {
            int depth = dfs(child); // Recursive call to get the depth for each child.
            // Update the two maximum depths found among children nodes
            if (depth > maxDepth1) {
                maxDepth2 = maxDepth1; // Update the second max if the new max is found
                maxDepth1 = depth;    // Update the new max depth
            } else if (depth > maxDepth2) {
                maxDepth2 = depth; // Update the second max if greater than it but less than the max depth
            }
        }

        // Update the maximum diameter if the sum of the two largest depths is greater than the current diameter.
        maxDiameter = max(maxDiameter, maxDepth1 + maxDepth2);

        // Return the maximum depth of this subtree to its parent caller, which is 1 plus the max depth of its children.
        return 1 + maxDepth1;
    }
};

```

Typescript Solution

```typescript
// Define the type for a Node that includes a value and an array of Node references as children
type Node = {
  val: number;
  children: Node[];
};

let maxDiameter: number; // Global variable to store the maximum diameter found.

// This function is the entry point for finding the diameter of the n-ary tree.
function diameter(root: Node | null): number {
  maxDiameter = 0; // Initialize the max diameter to 0.
  dfs(root); // Call the depth-first search helper function.
  return maxDiameter; // Return the maximum diameter calculated.
}

// Helper function for DFS traversal that calculates depths and updates the diameter.
// It returns the maximum distance from the current node to its furthest leaf node.
function dfs(root: Node | null): number {
  if (root === null) return 0; // Base case: If the node is null, return 0 (no depth).

  let maxDepthOne = 0; // To store the maximum depth among the paths in the children.
  let maxDepthTwo = 0; // To store the second maximum depth among the paths in the children.

  // Iterate through each child node of the current root.
  for (let child of root.children) {
    let depth = dfs(child); // Recursive call to get the depth for each child.
    // Update the two maximum depths found among children nodes
    if (depth > maxDepthOne) {
      maxDepthTwo = maxDepthOne; // Second max updated if a new max is found
      maxDepthOne = depth; // Set as the new max depth
    } else if (depth > maxDepthTwo) {
      maxDepthTwo = depth; // Update the second max if it's greater than the current second max
    }
  }

  // Update the maximum diameter if the sum of the two largest depths is greater than the current maximum diameter.
  maxDiameter = Math.max(maxDiameter, maxDepthOne + maxDepthTwo);

  // Return the maximum depth of this subtree to its parent, which is 1 plus the maximum depth of its children.
  return 1 + maxDepthOne;
}

```

Time and Space Complexity

The code provided defines a function `dfs(root)` that is used to compute the diameter of an N-ary tree. The diameter of an N-ary tree is defined as the length of the longest path between any two nodes in the tree.

Time Complexity:

The time complexity of the provided code is $O(N)$, where N is the total number of nodes in the tree. This is because the `dfs()` function is called recursively for each node exactly once. Within each call to `dfs()`, it performs a constant amount of work for each child. Since the sum of the sizes of all children's lists across all nodes is $N - 1$ (there are $N - 1$ edges in a tree with N nodes), the overall time complexity is linear relative to the number of nodes in the tree.

Space Complexity:

The space complexity of the code can be analyzed in two parts: the space required for the recursion call stack and the space required for the `dfs()` function execution.

- The worst-case space complexity for the recursion call stack is $O(H)$, where H is the height of the tree. In the worst case, the tree can be skewed, resembling a linked list, thus the height can become N, leading to a worst-case space complexity of $O(N)$.

- The additional space used by the `dfs()` function is constant, as it only uses a few integer variables (`m1`, `m2`, and `t`).

Considering both parts, the total space complexity is $O(H)$, which is $O(N)$ in the worst case (when the tree is a linear chain), but can be better (such as $O(\log N)$) if the tree is balanced.