2246. Longest Path With Different Adjacent Characters

Hard Tree Depth-First Search Graph Topological Sort Array String Leetcode Link

Problem Description

In this LeetCode problem, we are given a special type of graph called a tree. This tree has `n` nodes, each numbered from `0` to `n - 1`, and node `0` serves as the root node. The relationships between nodes and their parents are represented by an array `parent` where `parent[i]` indicates the parent node of node `i`. By definition, since node `0` is the root, it has no parent, which is denoted by `parent[0] == -1`.

Along with the tree structure, each node `i` is assigned a character given by the string `s[i]`. The goal is to identify the longest path in the tree where adjacent nodes on the path have different characters. The length of the path is defined as the number of nodes in that path.

The problem is ultimately asking to find the maximum length path in the tree that meets the criteria of having no two consecutive nodes with the same character assigned to them.

Intuition

To solve the problem, one can take a recursive approach, which is commonly used to traverse trees. The intuition here is to use Depth-First Search (DFS) to explore the tree from the root node, tracking the longest path that meets the condition along the way.

The recursive DFS function explores each child of the current node and determines the maximum path length within the subtrees of those children. While traversing, it keeps track of the length of the longest path ending at the current node (`mx`) and updates the global answer (`ans`) if a longer path is found.

The critical insight is that, if the current node and its child have different characters, the path can be extended by one. We compare the length of the longest path through each child node and pick the two longest paths to possibly update our answer. The trick is to add the path lengths of two longest non-similar child paths to the global answer. After completing the DFS traversal, the final answer is adjusted by adding one to account for the length of a single node path.

The algorithm essentially constructs a graph from the `parent` array to track the children of each node and performs DFS starting from the root. During DFS, it checks the characters assigned to the nodes and determines the longest path where adjacent nodes have different characters.

Solution Approach

The solution approach involves depth-first search (DFS), a common technique for exploring all the nodes in a tree from a given starting point. The implementation uses a helper function `dfs(i)` that is designed to recursively travel through the tree starting from node `i`.

Here's a step-by-step explanation of how the code achieves this:

1. A dictionary type `defaultdict(list)` called `g` is created to store the graph representation of the tree, with each key corresponding to a parent node and its value being the list of child nodes.

2. The graph `g` is populated by iterating over the indices of the `parent` list, starting from index 1 (since index 0 is the root and has no parent), and appending each index `i` to the list `g[parent[i]]`. This effectively builds the adjacency list for each node.

3. A `dfs(i)` function is defined to use recursive DFS traversal through the tree starting from node `i`. The function returns the maximum length of the path ending at node `i` that meets the non-adjacent-character condition.

4. Within `dfs(i)`, we loop through each child `j` of the current node `i` by accessing `g[i]`, and recursively call `dfs(j)` to continue the exploration further down the tree. The returned value from the recursive call represents the length of the maximum path from child `j` to a leaf that satisfies the condition, plus one (accounting for the edge to node `i`).

5. The function checks if the characters at the current node and its child node are different using `s[i] != s[j]`. If so, the `ans` variable (which is tracking the global maximum length) is updated with the sum of `mx` (the current maximum path length ending at node `i`) and `x` (the path length from the child node `j`), since this forms a valid path with distinct adjacent characters.

6. The path length `x` is compared with `mx` and updates `mx` if it's longer, ensuring `mx` always contains the length of the longest path that can be extended from the current node `i`.

After defining the `dfs(i)` function and initializing the adjacency list, the DFS traversal is kicked off at the root, `dfs(0)`. Since `dfs` only counts the length of the path without including the starting node, `ans + 1` is returned to account for the root node itself, giving the final answer.

Key Points:

- **Recursion**: The DFS algorithm is implemented using recursion, a natural fit for exploring trees.
- **Graph Representation**: Even though the tree is initially represented as a `parent` array, it's converted into a graph using adjacency lists for easier traversal.
- **Non-local Variable**: The `nonlocal` keyword is used for variable `ans` to allow its modification within the nested `dfs` function.
- **Max Tracking**: Two local maximum path lengths (`mx` and `x`) are used to keep track of the paths and to update the global maximum length `ans`.

Using DFS and careful updates to the maximum path lengths allow for an efficient search through the tree, yielding the longest path with the required property.

Example Walkthrough

Let's take a small example to illustrate the solution approach:

Suppose we have a tree with `n = 5` nodes and the following parent relationship array: `parent = [-1, 0, 0, 1, 1]`. This means that node `0` is the root, node `1` and node `2` are children of node `0`, and nodes `3` and `4` are children of node `1`. The characters assigned to each node are represented by the string `s = "ababa"`.

Now we will walk through the steps of the DFS solution to find the longest path with different adjacent characters:

1. First, we'll create the graph `g` from the `parent` array, which will look like this:

```
g = {
  0: [1, 2],
  1: [3, 4]
}
```

2. We define the `dfs(i)` function to start the DFS traversal. We will also initialize `ans = 0` to store the length of the longest path.

3. We start the DFS from the root node `dfs(0)`:

   - Since node `0` has two children (`1` and `2`), we call `dfs(1)` and `dfs(2)`.

4. In the `dfs(1)` call:

   - Node `1` has children `3` and `4`. We call `dfs(3)` and `dfs(4)` respectively.
   - The character at node `1` is `b`, and at node `3` it's `a`. Since they are different, `ans` can be updated to `2` if `dfs(3)` returns `1`.
   - Similarly, `dfs(4)` returns `1` because node `4`'s character is similar to `1`, making the path length `1`. Now `ans` would be updated to `3` because `mx + x = 2 + 1` (path through node `1` to node `3` and then from node `1` to node `4`).

5. For `dfs(2)`, since node `2` has no child and its character is `a` which is different from the root's character `a`, `dfs(2)` will simply return `1`.

6. As we return back to `dfs(0)`, we check the character at node `0`, which is `a`, and compare it with its children's characters. Node `2` has the same character, so we cannot form a longer path through node `2`. The longest path at this point is from node `0` to node `1` to node `3`, and node `1` to node `4`.

7. After traversing all nodes, `ans + 1` will give us the final answer. We add one because `ans` is tracking the number of edges in the longest path, and we want to count the number of nodes, which is one more than the number of edges.

In this particular example, the longest path with different adjacent characters has a length of `4`: through the nodes `0 -> 1 -> 3` and `1 -> 4`. Our `ans` was updated to `3` at most during the DFS traversal, and thus the final answer will be `ans + 1 = 4`.

Key Points of Clarification:

- While calculating the path lengths, we consider the length as the number of edges between nodes on the path.
- We need to return `ans + 1` at the end of the traversal since the counting starts at `0` and the problem asks for the number of nodes in the path, not the number of edges.
- This example demonstrates how `dfs` helps in efficiently finding and updating the longest path in the tree with the desired property.

Python Solution

```python
from collections import defaultdict
from typing import List

class Solution:
    def longestPath(self, parents: List[int], s: str) -> int:
        # Depth-First Search function to explore the graph
        def dfs(node_index: int) -> int:
            max_depth = 0  # Stores the maximum depth of child nodes
            nonlocal longest_path_len  # Refers to the non-local variable 'longest_path_len'
            for child_index in graph[node_index]:  # Iterate through child nodes
                child_depth = dfs(child_index) + 1  # Depth of child is parent depth + 1
                # If the characters are different, we can extend the path
                if s[node_index] != s[child_index]:
                    longest_path_len = max(longest_path_len, max_depth + child_depth)
                    max_depth = max(max_depth, child_depth)
            return max_depth  # Return the maximum depth encountered

        # Build the graph from the parent list
        graph = defaultdict(list)
        # Create adjacency list for each node except the root
        for index in range(1, len(parents)):
            graph[parents[index]].append(index)

        longest_path_len = 0  # Initialize the answer to track the maximum length of the path
        dfs(0)  # Start DFS from the root node

        # longest_path_len is the length of the path without root.
        # We add 1 to include the root in the final path length.
        return longest_path_len + 1
```

Java Solution

```java
class Solution {
    private List<Integer>[] graph; // Graph represented as an adjacency list
    private String labels; // String storing the labels of each node
    private int maxPathLength; // The maximum length of the path found that conforms to the question's rules

    // Method that returns the longest path where the consecutive nodes have different labels
    public int longestPath(int[] parents, String labels) {
        int n = parents.length;
        graph = new List[n]; // Initialize the graph to the size of the parent array
        this.labels = labels; // Assign the global variable to the input labels
        Arrays.setAll(graph, k -> new ArrayList<>()); // Initialize each list in the graph

        // Construct the graph from the parent array
        for (int i = 1; i < n; i++) {
            graph[parents[i]].add(i);
        }

        dfs(0); // Start the depth-first search from the root node (0)
        return maxPathLength + 1; // Add 1 because the path length is edges count, so nodes count is edges count + 1
    }

    // Helper method for depth-first search that computes the longest path
    private int dfs(int node) {
        int maxDepth = 0; // The max depth of the subtree rooted at the current node
        // Iterate through the children of the current node
        for (int child : graph[node]) {
            int depth = dfs(child) + 1; // Get the depth for each child and increment it as we move down
            // Only consider paths whose consecutive nodes have different labels
            if (labels.charAt(child) != labels.charAt(node)) {
                maxPathLength = Math.max(maxPathLength, maxDepth + depth); // Update maxPathLength if needed
                maxDepth = Math.max(maxDepth, depth); // Update the maxDepth if the current depth is greater
            }
        }
        return maxDepth; // Return the max depth found for this node's subtree
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <functional> // Include for std::function

class Solution {
public:
    // Function to find the longest path where each character is different
    // from its parent in a tree defined by parent-child relationships and node values given by string s.
    int longestPath(vector<int>& parent, string& s) {
        int numNodes = parent.size(); // Total number of nodes in the tree.
        vector<vector<int>> graph(numNodes);

        // Build the adjacency list representation of the tree from the parent array.
        for (int i = 1; i < numNodes; ++i) {
            graph[parent[i]].push_back(i);
        }

        // Variable to store the length of the longest path found.
        int longestPathLength = 0;

        // Define the depth-first search (DFS) function using lambda notation.
        std::function<int(int)> dfs = [&](int currentNode) -> int {
            // The maximum path length through this node.
            int maxLengthThroughCurrent = 0;

            // Explore all child nodes of the current node.
            for (int child : graph[currentNode]) {
                // Recursively perform DFS from the child node.
                int pathLengthFromChild = dfs(child) + 1; // +1 for the edge from the current node to the child node.

                // If the current node and the child have different characters,
                // try to extend the path.
                if (s[currentNode] != s[child]) {
                    // Update the longest path if combining two paths through this node
                    // results in a longer path.
                    longestPathLength = max(longestPathLength, maxLengthThroughCurrent + pathLengthFromChild);

                    // Update the maximum length of the path that goes through the current node.
                    maxLengthThroughCurrent = max(maxLengthThroughCurrent, pathLengthFromChild);
                }
            }

            // Return the max length of the path through the current node to its parent.
            return maxLengthThroughCurrent;
        };

        // Start DFS from the root node (0).
        dfs(0);

        // Return the length of the longest path. We add 1 because the path length
        // is the number of nodes on the path, but longestPathLength stores the number of edges.
        return longestPathLength + 1;
    }
};
```

Typescript Solution

```typescript
function longestPath(parents: number[], s: string): number {
    // The number of nodes in the tree
    const nodeCount = parents.length;

    // Adjacency list representing the tree graph
    // Each index corresponds to a node, which contains an array of its children
    const graph: number[][] = Array.from({ length: nodeCount }, () => []);

    // Building the graph from the parent array
    for (let i = 1; i < nodeCount; ++i) {
        graph[parents[i]].push(i);
    }

    // The variable to store the length of the longest path
    let longestPathLength = 0;

    // Depth-First Search function to explore nodes
    const dfs = (node: number): number => {
        // To hold the max path through the current node
        let maxPathThroughNode = 0;

        // Iterating through each child of the current node
        for (const child of graph[node]) {
            // Determine the path length including this child if unique character
            const childPathLength = dfs(child) + 1;

            // We only consider this path if it contains unique characters
            if (s[node] !== s[child]) {
                // Update the longest path combining paths from two children
                longestPathLength = Math.max(longestPathLength, maxPathThroughNode + childPathLength);
                // Update the max path length through this node with the length including the current child
                maxPathThroughNode = Math.max(maxPathThroughNode, childPathLength);
            }
        }

        // Return the max path length from current node's children
        return maxPathThroughNode;
    };

    // Start Depth-First Search from the root node (0)
    dfs(0);

    // The longest path will be longestPathLength + 1, as the count starts from 0
    return longestPathLength + 1;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(N)`, where `N` is the number of nodes in the input list `parent`. This complexity arises because the code visits every node exactly once through depth-first search (DFS). Each node processing (not counting the DFS recursion) takes constant time, leading to a linear time complexity relative to the number of nodes.

Space Complexity

The space complexity of the code is also `O(N)`. This space is required for the adjacency list `g` and the call stack during the recursive DFS calls. Each node can contribute at most one frame to the call stack (in the case of a linear tree), and the adjacency list can store up to `2(N - 1)` edges (considering an undirected representation of the tree for understanding, although the actual directed edges are less and do not contribute to space more than `O(N)`). As a result, the overall space complexity remains linear with respect to `N`.