2872. Maximum Number of K-Divisible Components

Hard Tree Depth-First Search Dynamic Programming Leetcode Link

Problem Description

The given problem defines an undirected tree consisting of `n` nodes with each node labeled from `0` to `n - 1`. The structure of the tree is described using a list of edges, where each edge is represented by a pair `[a_i, b_i]` connecting the nodes `a_i` and `b_i`. Along with the structure, each node has an associated value provided in the array `values`.

The task is to find a way to split this tree into multiple components by possibly removing some edges. The condition for a valid split is that the total value of each resulting component (sum of all node values in the component) must be divisible by a given integer `k`. The ultimate goal is to maximize the number of such valid components after the split.

To state it simply, you're asked to figure out the maximum number of groups you can create from the tree's nodes where the sum of values within each group is a multiple of 'k'. We must decide which edges to remove, if any, to achieve this while keeping the most groups.

Intuition

The intuitive solution approach stems from two insights:

1. **Divisibility Propagation:** Since the entire tree's value is divisible by `k`, breaking off any subtree whose value is also divisible by `k` will leave the remaining tree with a sum still divisible by `k`. This holds true recursively for any subtrees split off from the main tree. Thus, we can look for divisible subtrees to count as separate components.

2. **DFS Utilization:** To identify subtrees conforming to the divisibility rule, we need to examine the sums of each possible subtree in the tree. This is a classic use-case for Depth-First Search (DFS). Starting from the root node (or any node, as it's a connected tree), DFS can help us explore each branch to its leaves, summing the values of nodes as we backtrack up the tree. When we find a subtree sum divisible by `k`, we mark it as a potential split point and continue searching for more.

By employing DFS, we iterate over the tree nodes while keeping a running sum of the subtree rooted at each node. Whenever the sum of a subtree equals a multiple of `k`, we have found a valid component. By effectively adding these components to our count and continuing the DFS, we can determine the maximum number of valid components we can split the tree into, hence arriving at the solution to our problem.

Solution Approach

To implement the solution, we utilize a depth-first search (DFS) algorithm, which is well-suited for exploring tree structures and their subtrees. Here's a step-by-step breakdown of how the DFS-based algorithm is applied to achieve our goal:

1. **Build the Graph Representation:** Before we begin the DFS, we need to convert the edge list into a graph representation that makes it easier to traverse. In Python, this often takes the form of an adjacency list, which is a list of lists where each sublist corresponds to a node and contains the nodes that it's directly connected to. This is achieved by iterating through the `edges` array and populating the adjacency list (`g`) accordingly.

2. **Depth-First Search (DFS) Implementation:**

   - A recursive `dfs` function is created, which takes a node index (`i`) and its parent index (`fa`) as arguments. The purpose of this function is to compute the sum of values for the subtree rooted at node `i`.
   - Inside the function, we begin by assigning the node's value to a variable `s`. Then, we iterate over all the direct children or neighbors of node `i` (given by `g[i]`).
   - For each neighbor `j`, we check if it's not the parent (`fa`) to avoid unnecessary backtracking, and then we recursively call `dfs(j, i)` to add the sum of that subtree to `s`.
   - After processing all children, we check if the sum `s` is divisible by `k`. If it is, our global `ans` counter is incremented to reflect that we've found a valid subtree (or component).

3. **Initiate DFS and Handle Return Values:**

   - The DFS is initiated by calling the `dfs` function on the root node (typically node `0` in a 0-indexed tree) with a parent index of `-1` (since the root does not have a parent).
   - The `dfs` initialization ensures that we explore the entire tree, summing subtree values and finding divisible components. The DFS function naturally operates by post-order traversal, examining children before processing the current node. This is crucial since subtree sums need to be calculated from the leaves upward.

4. **Count the Result and Minus One:**

   - The final count must be decremented by one because the original tree's sum is divisible by `k` and thus counts as one entire component. When the DFS includes the root's sum in the final answer, we effectively double-count this component.
   - Therefore, the last line of the provided solution function `maxKDivisibleComponents` returns `ans - 1` to adjust for this overcounting and to provide the actual maximum number of components that can result from any valid split.

In terms of data structures, this solution uses an adjacency list for representing the graph and relies on recursion to maintain state during the DFS traversal. As for patterns, using global variables such as `ans` in conjunction with DFS is a common technique to accumulate results throughout the recursion.

The elegance of this solution lies in its efficient use of the DFS traversal to examine the entire tree and identify points where the tree can be split while simultaneously ensuring that the resultant component values meet the divisibility requirement.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we have a tree consisting of 5 nodes with the following edges and values:

- `edges = [[0, 1], [0, 2], [1, 3], [1, 4]]`
- `values = [4, 2, 6, 3, 3]`
- `k = 5`

The adjacency list representation would look like this based on the given edges:

- `g = [[1, 2], [0, 3, 4], [0], [1], [1]]`

Here are the steps of applying the approach to this example:

1. First, we perform DFS starting from the root node `0`. The node values are `[4, 2, 6, 3, 3]`. The total sum of all values in the tree is `18`, which is not divisible by `k = 5`.

2. From node `0`, we DFS into its children `1` and `2`. Node `2` is a leaf node, so it contributes its value of `6` and returns it to the parent node `0`.

3. After exploring node `2`, we move to node `1`. Node `1` has children `3` and `4`. They are both leaf nodes and return their values `3` and `3`.

4. Back at node `1`, we add its value `2` to the sum of its subtree, which is `2 + 3 + 3 = 8`. The sum `8` is divisible by `k = 5`, meaning we've found one valid component. We'll increment our global `ans` counter to `1`.

5. Return to root node `0`, the subtree sums of its children have been calculated, we sum them up with node `0`'s own value: `4 + 6 + 8 = 18`. This is again not divisible by `k = 5`, so no increment to `ans`.

6. The DFS is completed. Our global `ans` counter is `1`, but we have to decrement it by one, because we're not considering the entire tree as one component. Hence, the final answer is `1 - 1 = 0`.

7. However, if we sum the entire tree value, it's `18`, which is still not divisible by `5`. Therefore, our example doesn't allow dividing the tree into components with each sum divisible by `k = 5`.

Although we couldn't split the tree into valid components in this example under the divisibility rule given by `k`, the procedure demonstrates how a depth-first search helps to explore all subtrees to identify the possible splits, increment a counter when a divisible subtree is found, and understanding why we subtract one in the end.

Python Solution

```python
class Solution:
    def maxKDivisibleComponents(self, n: int, edges: List[List[int]], values: List[int], k: int) -> int:
        # Helper method for Depth-First Search (DFS)
        def dfs(node: int, parent: int) -> int:
            # Start with the value of the current node.
            total_sum = values[node]

            # Traverse through the graph.
            for neighbor in graph[node]:
                # If the neighbor is not the parent node, perform DFS recursively.
                if neighbor != parent:
                    total_sum += dfs(neighbor, node)

            # Increment the counter if the sum is divisible by k.
            nonlocal num_divisible_subtrees
            num_divisible_subtrees += total_sum % k == 0

            # Return the sum for the current subtree.
            return total_sum

        # Initialize graph as adjacency list representation.
        graph = [[] for _ in range(n)]
        for edge in edges:
            a, b = edge
            graph[a].append(b)
            graph[b].append(a)

        # Counter for the number of subtrees with sums divisible by k.
        num_divisible_subtrees = 0
        # Perform DFS starting from node 0 assuming 0 as root node.
        dfs(0, -1)

        # Decrement to exclude the sum of the whole tree,
        # as by the problem definition it should not be counted as a subtree.
        return num_divisible_subtrees - (values[0] % k == 0)

# Note:
# The List type import from typing module is not included in the snippet above,
# but it should be imported at the top of the script as follows:
# from typing import List

```

Java Solution

```java
class Solution {
    private int answer;  // Renamed 'ans' to 'answer' for better readability
    private List<Integer>[] graph; // Renamed 'g' to 'graph' to clarify this represents a graph
    private int[] nodeValues; // Renamed 'values' to 'nodeValues' to avoid confusion with the method parameter 'values'
    private int divisor; // Renamed 'k' to 'divisor' for clarity

    // Method to find the maximum number of components divisible by 'k'
    public int maxKDivisibleComponents(int n, int[][] edges, int[] values, int k) {
        this.graph = new List[n]; // Initialize the graph as a list of integer lists
        Arrays.setAll(this.graph, i -> new ArrayList<>()); // Create an adjacency list for each node i

        // Add edges to the graph to represent the bidirectional relationships between nodes
        for (int[] edge : edges) {
            int nodeFrom = edge[0], nodeTo = edge[1];
            graph[nodeFrom].add(nodeTo);
            graph[nodeTo].add(nodeFrom);
        }

        this.nodeValues = values; // Assign the provided node values to the instance variable
        this.divisor = k; // Store the divisor as an instance variable

        dfs(0, -1); // Invoke the depth-first search from the root node (assuming node 0 as root)
        return answer; // Return the total number of components with sum divisible by 'k'
    }

    // Depth-first search to find the components with sum of values divisible by 'k'
    private long dfs(int nodeIndex, int parentIndex) {
        long sum = nodeValues[nodeIndex]; // Start with the node's own value
        // Recursively visit all the connected nodes (children) that are not the parent
        for (int adjacentNode : graph[nodeIndex]) {
            if (adjacentNode != parentIndex) {
                sum += dfs(adjacentNode, nodeIndex); // Add the sum of the subtree to the current node's sum
            }
        }
        // If the sum is divisible by 'k', increment the answer
        if (sum % divisor == 0) {
            answer++;
        }
        return sum; // Return the sum of the node values in the subtree rooted at this node
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional>
using namespace std;

class Solution {
public:
    int maxKDivisibleComponents(int nodeCount, vector<vector<int>>& edges, vector<int>& values, int divisor) {
        int componentCount = 0; // Used to count components with value sum divisible by k
        vector<int> adjacencyList[nodeCount]; // Adjacency list to represent the graph

        // Building the graph from edges
        for (const auto& edge : edges) {
            int fromNode = edge[0];
            int toNode = edge[1];
            adjacencyList[fromNode].push_back(toNode);
            adjacencyList[toNode].push_back(fromNode);
        }

        // Depth First Search (DFS) lambda function to traverse the graph and calculate sums
        function<long long(int, int)> depthFirstSearch = [&](int currentNode, int parent) {
            long long sum = values[currentNode]; // Start with the value of the current node

            // Visit all adjacent nodes except the parent
            for (int adjacentNode : adjacencyList[currentNode]) {
                if (adjacentNode != parent) {
                    sum += depthFirstSearch(adjacentNode, currentNode);
                }
            }

            // Increase the count if sum of values in this component is divisible by k
            componentCount += (sum % divisor == 0);
            return sum; // Return the sum of values from this component
        };

        // Starting DFS from node 0 assuming 0 is always part of the graph
        depthFirstSearch(0, -1);

        // Return the count of components with value sum divisible by k
        return componentCount;
    }
};

```

Typescript Solution

```typescript
function maxKDivisibleComponents(
    nodeCount: number,
    edges: number[][],
    values: number[],
    k: number,
): number {
    // Create an adjacency list to represent the graph
    const graph: number[][] = Array.from({ length: nodeCount }, () => []);

    // Populate the graph with edges
    for (const [node1, node2] of edges) {
        graph[node1].push(node2);
        graph[node2].push(node1);
    }

    // Initialize a variable to keep track of the number of components
    let componentCount = 0;

    // Depth-First Search function to explore nodes
    const dfs = (currentNode: number, parent: number): number => {
        // Start with the current node's value
        let sum = values[currentNode];

        // Explore all connected nodes that are not the parent of the current node
        for (const adjacentNode of graph[currentNode]) {
            if (adjacentNode !== parent) {
                // Aggregate the values from child nodes
                sum += dfs(adjacentNode, currentNode);
            }
        }

        // If the aggregate sum is divisible by k, increment the component count
        if (sum % k === 0) {
            componentCount++;
        }

        // Return the sum of values for this subtree
        return sum;
    };

    // Begin the DFS traversal from the first node (assuming 0-indexed)
    dfs(0, -1);

    // Return the total number of components where the sum is divisible by k
    // Note: We decrement the componentCount by 1 to exclude the sum of the entire tree
    return componentCount - 1;
}

```

Time and Space Complexity

The given code defines a function that calculates the maximum number of components in a graph where each component's sum of values is divisible by `k`. The graph is represented as a tree with `n` nodes (vertices) and its edges.

Time Complexity:

The time complexity of the function is `O(n)`. This is because the function performs a Depth-First Search (DFS) on the tree, starting from node `0`. Each node in the graph is visited only once during the DFS, and at each node, the sum of values is updated and checked for divisibility by `k`. Since there are `n` nodes and the function visits each node exactly once, the overall time complexity is `O(n)`.

Space Complexity:

The space complexity is also `O(n)`. This includes:

- The space needed to store the graph `g`, which is an adjacency list representation of the tree. There are `n` lists, one for each node, and each list contains the adjacent nodes. Since the tree has `n-1` edges, the total number of elements across all the adjacency lists will be `2 * (n - 1)` (each edge is stored twice, once for each node it connects). However, the big-O notation only considers the highest order term, so this is simplified to `O(n)`.
- The space needed for the DFS recursion call stack. In the worst case, if the tree is implemented as a linked list (i.e., each node has only one child), the maximum depth of the stack would be `n`. Thus, the space complexity due to the call stack is `O(n)` as well.

Given that both of these space usages are linear with respect to the number of nodes, the combined space complexity of the function remains `O(n)`.