2378. Choose Edges to Maximize Score in a Tree

Medium Tree Depth-First Search Dynamic Programming Leetcode Link

Problem Description

In this LeetCode problem, we're tasked with finding the maximum sum of edge weights in a weighted, rooted tree such that no two selected edges are adjacent. The tree consists of n nodes labeled from 0 to $n - 1$, with node 0 being the root. A 2D array `edges` represents the tree's structure and the weights of its edges where each entry `edges[i]` is a pair `[ai, wi]` signifying that node `ai` is the parent of node i and the weight of the edge connecting them is `weight_i`. Note that for the root node, `edges[0]` will be `[-1, -1]` as it does not have a parent.

The goal is to select a subset of these edges, ensuring that none of them share a common node (i.e., they are not adjacent), such that the total weight of the selected edges is as large as possible. The result should be the maximum achievable sum of weights.

Intuition

The solution to the problem can be found using a technique known as "tree dynamic programming". The main intuition behind this approach is to consider each node in the tree and make an optimal decision for that node's sub-tree, considering whether it's more advantageous to include the edge leading to this node in our sum or not.

Here's the thinking process that leads to the solution:

1. Process the tree in a bottom-up manner using a depth-first search (DFS).
2. At each node, we need to decide whether to include the edge from its parent or not. This decision affects the sub-tree rooted at this node.
3. Make two calculations for each node: the maximum sum of weights if we exclude the edge to the parent node (we'll call this a), and the maximum sum of weights if we include the edge to the parent node (we'll call this b).
4. For every child of the current node, calculate the sum of a and b and add the child's b to the current node's a and the child's b to the current node's b. This aggregates the maximum weights from the subtrees where the parent edges are not included.
5. To compute the current node's b, find the child with the maximum difference between including its parent edge and excluding it (which is $b - a$ for each child) where a is the sum including the edge and b is the sum excluding the edge and b is the weight of the edge to the parent). Add this difference to current node's a to get the maximum sum where the edge to the current node's parent is included.
6. The final step is to start this process at the root and get the answer b for the root, which will be the overall maximum sum.

In the provided Python code, a recursive `dfs()` function handles the computations for each node, and the tree is represented as a dictionary of lists `g`, making it easy to traverse the children of any node. The final answer is obtained by the `dfs(0)[1]`, which invokes the DFS starting from the root node and extracting the b value as the result.

Solution Approach

The implementation of the solution to our tree dynamic programming problem makes use of several key programming concepts, which include recursion, depth-first search (DFS), and memoization. Here's the walk-through of how these concepts are intertwined in the solution:

1. **Recursion**: A recursive function, `dfs(i)`, is utilized to perform a post-order traversal of the tree. This means it first visits all of a node's descendants before processing the node itself. This is vital for dynamic programming on trees, as it ensures that we have the solutions for all child sub-problems ready when we are computing the solution for the parent.
2. **Depth-First Search (DFS)**: To explore the whole tree starting from the root node 0, we carry out a DFS. This helps us visit every node in the right order to implement our dynamic programming solution.
3. **Memoization**: Although not explicitly in the form of a table, our recursive function employs memoization by calculating and returning two values for each node – the maximum sum including the node's parent edge and the maximum sum excluding it. The pair of values (`a`, `b`) represents this duo of sums. Any repeat computation is avoided by building upon the results of the subtrees.
4. **Data Structures Used**:
   - **Dictionary of Lists (`g`)**: Represents our tree, mapping each parent node to a list of tuples, with each tuple containing a child node and the corresponding edge weight.
   - **Tuples (`a`, `b`, `t`)**: Used to store intermediate values during the traversal and comparisons.
5. **Algorithm**: Initially, a graph, `g`, is built from our `edges`, which represents each non-root node's parent and the weight of the connecting edge.

   We then run a `dfs(0)` call on the root node, which triggers a series of recursive calls through the entire tree structure, processing child nodes first.

   In the `dfs` function:
   - Variables `a` and `b` are initialized to i, and t as well, which will track the maximum transition from one state to another.
   - Loop over each child j with weight w of the current node i. Recursively call `dfs(j)` to get x, the maximum sum including j's parent edge, and y, the maximum sum excluding it.
   - Update a to include all children's maximum sums excluding their parent edge because including a child edge means we can't include the parent edge (they are adjacent).
   - Calculate the transition t which is the maximum difference in the sum we can get by potentially including the current node's parent edge. This involves picking the child that gives the maximum extra benefit $= -y + w$.
   - Finally, b is updated as $a + t$, which incorporates the best case where including the child's edge maximizes the sum. After the recursive calls have been made, `dfs(0)[1]` gives us b for the root node which, as per our memoization strategy, represents the maximum weighted sum without any adjacent edges.

This algorithm works efficiently due to the inherent properties of trees (acyclic, one path between any two nodes) and allows us to solve the problem with a time complexity that is linear in the number of nodes.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Assume we have $n = 5$ nodes in our tree, and the `edges` array is as follows:

```
edges = [
  [-1, -1], // Node 0 is the root
  [0, 2],   // Node 1 is a child of Root 0 with edge weight 2
  [0, 3],   // Node 2 is a child of Root 0 with edge weight 3
  [1, 5],   // Node 3 is a child of Node 1 with edge weight 5
  [1, 4]    // Node 4 is a child of Node 1 with edge weight 4
]
```

Using the above edges array, we can construct the following tree structure:

```
    0
   / \
  2   3
 / \
5   4
```

Let's walk through the algorithm:

1. We start the DFS on the root node, `0`.
2. The `dfs(1)` function explores the children of the root node: nodes `1` and `2`.
3. At node `2`, which is a leaf node, there are no children to explore. Thus for node `2`:
   - We calculate `a = 0` (max sum if we exclude the edge from its parent).
   - We calculate `b = 3` (max sum if we include the edge from its parent, which is 0 since it's a leaf).
4. At node `1`, it has two children, `3` and `4`. We run `dfs()` on both children.
5. For node `3` (a leaf node):
   - `a = 0` (excluding the parent edge).
   - `b = 5` (including the parent edge, because it's a leaf).
6. For node `4` (also a leaf):
   - `a = 0`.
   - `b = 4`.
7. Back at node `1`, we process the info from its children:
   - For node `3`, `x - y + w` translates to `0 - 0 + 5 = 5`.
   - For node `4`, `x - y + w` translates to `0 - 0 + 4 = 4`.

   Since node `1` has the largest transition, we choose it, resulting in:
   - `a = 0 + 0 + 0` (the sum of maximum weights from child nodes excluding their parent edge).
   - `b = 0 + 4` (adding the maximum extra benefit of including node `1`'s edge).
8. At the root node `0`:
   - From node `1`, we get `a = 0, b = 4`.
   - From node `2`, we get `a = 0, b = 0`.

   We can only include one of the children's edges, and since `b` from node `1` is larger, we include node `1`:
   - `a = 0` (as it's the root and has no parent edge).
   - The transition for node `1` is `b + weight` which is `4 + 3 = 7`.
   - The transition for node `2` is `0 + 2 = 2`.

   We pick the transition from node `1` because it's larger:
   - `b = a + transition` giving us `b = 0 + 7 = 7`.

So, for this tree, the maximum sum with no adjacent edges would be `7`, obtained by selecting the edge between nodes `0` and `1` and the edge between nodes `1` and `3`.

By executing `dfs(0)[1]`, we retrieve `b` for the root node, which is `7`, and that's our solution.

Python Solution

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

class Solution:
    def max_score(self, edges: List[List[int]]) -> int:
        # Depth-first search (DFS) function for traverse graph and calculate score
        def dfs(node, index):
            base_score = best_score = total_gain = 0
            # Explore each child node and update the respective weights
            for child_node, child_index in graph[node]:
                child_base, child_best = dfs(child_node, child_index)
                base_score += child_best
                child_gain = child_base - child_best
                total_gain = max(total_gain, child_base - child_best + weight)
            # Update the best score to account for the total gain from the most profitable child
            best_score += total_gain
            return base_score, best_score

        # Convert edge list to graph representation for easier traversal
        graph = defaultdict(list)
        for node, (parent, weight) in enumerate(edges[1:], 1):
            graph[parent].append((node, weight))

        # Initiate DFS from the root node (index 0) and return the best score
        return dfs(0)[1]

# Example usage:
# sol = Solution()
# print(sol.max_score([[],[2, 2],[1,4]], 2, 0))
```

Java Solution

```java
class Solution {
    private List<int[]>[] adjacencyList; // Using an array of lists to represent the graph

    public long maxScore(int[][] edges) {
        int numNodes = edges.length; // The number of edges given as the count of nodes
        adjacencyList = new List[numNodes]; // Initialize the adjacency list
        // Fill the adjacency list with array lists for each node
        Arrays.setAll(adjacencyList, index -> new ArrayList<>());
        // Construct the graph with the given edge weights
        for (int i = 1; i < numNodes; ++i) {
            int parent = edges[i][0], weight = edges[i][1];
            // Add a directed edge from 'parent' to 'i' with 'weight'
            adjacencyList[parent].add(new int[] {i, weight});
        }
        // Call the dfs method on node 0 and return the maximum score from the second value of the array
        return dfs(0)[1];
    }

    private long[] dfs(int node) {
        long sumOfSubtreeScores = 0; // Stores the sum of scores within the subtree
        long maxDifference = Long.MIN_VALUE; // Tracks the maximum difference for the current node
        long maxGift = 0; // Stores the maximum gift received between child score with and without the current node

        for (int[] next : adjacencyList[node]) {
            int childNode = next[0], edgeWeight = next[1];
            // Perform DFS on the child node
            long[] childScores = dfs(childNode);
            // Add the child's score without the current edge to the total sum of scores
            sumOfSubtreeScores += childScores[0];
            maxDifference = Math.max(maxDifference, childScores[1] + edgeWeight);
            // Find the child that maximizes the difference
            maxGift = Math.max(maxGift, childScores[0] - childScores[1] + edgeWeight);
        }

        // Incorporate the maximum difference into the max score including the current node
        sumOfSubtreeScores += maxGift;
        // Return both the sum of subtree scores and the max score including the current node
        return new long[] {sumOfSubtreeScores, maxDifference + sumOfSubtreeScores};
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional>

class Solution {
public:
    // Function to calculate the maximum score achievable from the given tree.
    long long maxScore(std::vector<std::vector<int>>& edges) {
        // Number of nodes in the tree.
        int numNodes = edges.size();

        // Graph representation: each node has a list of (child, weight) pairs.
        std::vector<std::vector<std::pair<int, int>>> graph(numNodes);

        // Constructing the graph from the given edges, starting at node 1, as node 0 is the root.
        for (int i = 1; i < numNodes; ++i) {
            int parent = edges[i][0], weight = edges[i][1];
            graph[parent].push_back({i, weight}); // Add the child and associated weight to the parent's list.
        }

        // Lambda function for depth-first search (DFS) to calculate scores.
        std::function<std::pair<long long, long long>(int)> dfs = [&](int node) -> std::pair<long long, long long> {
            long long withoutCurrent = 0; // Score without taking current node's edge.
            long long withCurrent = 0;    // Score with taking current node's edge.
            long long maxGain = 0;        // The maximum difference between taking and not taking an edge.

            // Iterate over all children of the current node.
            for (auto& [child, weight] : graph[node]) {
                // Recursively call DFS on the child node.
                auto [childWithout, childWith] = dfs(child);
                // Update the score without taking the current node's edge.
                withoutCurrent += childWith;
                // Update the score with taking the current node's edge.
                withCurrent += childWith;
                // Find the maximum difference when switching from childWithout to childWith along one edge.
                maxGap = std::max(maxGap, childWithout - childWith + weight);
            }

            // Include the maxGap to 'withCurrent' to reflect the maximum score.
            withCurrent += maxGap;

            // Return {withoutCurrent, withCurrent} as a pair of scores.
            return {withoutCurrent, withCurrent};
        };

        // Start DFS traversal from the root node (0) and return max score with root.
        return dfs(0).second;
    }
};
```

Typescript Solution

```typescript
type Edge = [number, number]; // Represents an edge with a parent and weight.
type ScorePair = [number, number]; // Represents a pair of scores.

// Represents each node having a list of child nodes along with the weight of the edge connecting to the child.
let graph: Edge[][] = [[]];

// Function to calculate the maximum score achievable from the given tree.
function maxScore(edges: Edge[]): number {
    // Number of nodes in the tree.
    let numNodes = edges.length;

    // Initialize the graph based on the number of nodes.
    graph = Array(numNodes).fill([]);

    for (let i = 0; i < numNodes; ++i) {
        graph[i] = [];
    }

    // Construct the graph from the edge list, starting at node 0 (root node).
    for (let i = 1; i < numNodes; ++i) {
        // Destructuring to get parent and weight.
        let [parent, weight] = edges[i];
        // Add the child node and associated weight to the parent's list of children.
        graph[parent].push([i, weight]);
    }

    // Recursive function to perform Depth-First Search (DFS) and calculate scores.
    const dfs = (node: number): ScorePair => {
        // Score without taking the current node's edge.
        let withoutCurrent = 0;
        // Score including the current node's edge.
        let withCurrent = 0;
        // The maximum score difference by choosing one child's edge to include.
        let maxGap = 0;

        // Iterate over each child and update the scores.
        for (const [child, weight] of graph[node]) {
            // Recursively call DFS for each child, getting their scores.
            const [childWithout, childWith] = dfs(child);
            // Add the maximum child's score to the score without the current edge.
            withoutCurrent += childWith;
            // Find the maximum gap when considering the scores with and without the child's edge.
            maxGap = Math.max(maxGap, childWithout - childWith + weight);
        }

        // Calculate the score including the current node's edge by adding the max gap.
        withCurrent = withoutCurrent + maxGap;

        // Return the pair of scores without and with the current node's edge.
        return [withoutCurrent, withCurrent];
    };

    // Start DFS traversal from the root node (0) and return the maximum score with the root's edge.
    return dfs(0)[1];
}

// Example usage:
// let edges: Edge[] = [
//     [-1, 0], // Root node does not have a parent, so it's often represented by -1.
//     [0, 1], // Node 1 has a parent node 0 and an edge weight of 1.
//     [0, 2], // Node 2 has a parent node 0 and an edge weight of 2, and so on.
//     // Add more nodes accordingly
// ];
// let result = maxScore(edges);
```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by the DFS (Depth-First Search) that is performed over the tree structure. The DFS function `dfs` is called recursively for each node in the tree. In the worst case, each node is visited exactly once during the DFS traversal. Thus, the time complexity is $O(N)$, where N is the number of nodes in the input tree represented by the `edges` list.

Space Complexity

The space complexity is defined by the space required for the DFS recursion stack as well as the space needed for the adjacency list `g`. For the recursion stack, in the worst case, the stack size will be equal to the height of the input tree, which can be $O(N)$ in the case of a skewed tree. For the adjacency list `g`, it holds all the edges, and the space required is proportional to the number of nodes, which is also $O(N)$. Therefore, the space complexity of the algorithm is $O(N)$, where N is the number of nodes in the input tree.