2538. Difference Between Maximum and Minimum Price Sum

Hard | Tree | Depth-First Search | Array | Dynamic Programming | Leetcode Link

Problem Description

In this problem, we're given an undirected tree that consists of `n` nodes. A tree is a connected graph with no cycles, and since it's undirected, connections between nodes are bidirectional. The nodes are labeled from `0` to `n - 1`. We are also given a 2D array `edges` which contains `n - 1` pairs of integers, where each pair `[a_i, b_i]` represents an edge between node `a_i` and node `b_i`.

Additionally, every node has an associated price, as defined in the integer array `price`, where `price[i]` is the price of the `i`th node. The "price sum" of a path in the tree is the sum of prices of all nodes along that path.

The task is to determine the maximum possible "cost" of any rooting of the tree. Here, the "cost" after choosing a node `root` to be the root of the tree is defined as the difference between the maximum and minimum price sum of all paths that start from the chosen `root`.

To solve this problem, you must consider all possible ways to root the tree and calculate the cost for each. The answer is the maximum of these costs.

Intuition

The intuition behind the solution involves dynamic programming and depth-first search (DFS). Since the input is a tree, we can start from any node and perform a DFS to search through all nodes and find the price sum of all paths starting from the current node.

Now, when we pick a node to root the tree, each connected node can either contribute to the maximum price sum or the minimum price sum based on the path originating from the root. So for each node, we need to track two values during DFS:

- The maximum price sum `a` that can be obtained by including the current node in the path.
- The second best (or the maximum price sum of the subtree excluding the current node) `b`.

At each step, we compare the current node's price with the discovered price sums from its children. As DFS unwinds, we update the global maximum `ans` by considering the possible paths that can be formed by including the current node and the best price sums from its children.

The recursive `dfs` function achieves this by returning the best (maximum) and second best (maximum excluding the current) price sums for each node. The variables `a` and `b` are continually updated as the DFS explores the tree, and every time we call `dfs` recursively, we consider the node's own price plus the best and second-best prices from the child.

The solution concludes after the DFS traversal, where `ans` holds the maximum possible cost among all possible root choices, which is the required answer.

Solution Approach

To solve the problem, the reference solution uses depth-first search (DFS) to traverse the tree and dynamic programming (DP) to keep track of the best and second-best price sums for paths originating from each node. Here's a walkthrough of how the code accomplishes this:

1. A defaultdict `g` is used to represent the graph with an adjacency list. Each element in `g` is a list that contains all the nodes connected to a particular node.
2. The `dfs` function is defined which will perform a depth-first search starting from a given node `i`, where `fa` is the node's parent (to avoid revisiting it).
3. In the `dfs` function, two variables, `a` and `b`, are initialized. `a` represents the maximum price sum including the current node, and `b` represents the maximum price sum from the current node's subtree when it's not considered.
4. The function iterates over all the nodes `j` connected to the current node `i`. If `j` is not the parent (i.e., it's not the node we came from), it recursively calls the `dfs` function to explore the subtree rooted at `node j`.
5. The `dfs` function for child node `j` returns a pair of values `c` and `d`, where `c` is the maximum price sum including node `j`, and `d` is the maximum price sum from the subtree under `j` excluding itself.
6. The global variable `ans` is updated during each call to `dfs` using the recursion stack to explore the subtree. It takes the maximum of the current `ans`, and two new potential maximums: `a + d` and `b + c`. The first value is the situation where we include the current node in the path, while the second value is the situation where we exclude the current node.
7. The function then updates `a` and `b` for the current node `i` based on the values returned by its children, which is effectively a bottom-up update representing dynamic programming.
8. After traversing and updating all connected nodes to `i`, the function returns the pair `(a, b)`.
9. The DFS starts with the first node (node `0` here as an arbitrary choice, since it's a tree and the result is not dependent on the choice of the root for DFS) and its parent set to `-1` (as it has no parent).
10. After the complete traversal, `ans` will contain the maximum possible cost after exploring all possible roots and paths in the tree.

Throughout this implementation, the code maintains a single traversal of the tree using DFS while cleverly updating the dynamic programming states `(a, b)`, avoiding any redundant calculations.

In summary, the problem is solved by a single pass over the tree with DFS, which is both efficient and effective for trees, along with the usage of dynamic programming techniques to track and update the price sums.

Example Walkthrough

Let's consider a small example where we have a tree with 4 nodes, and the `edges` array is `[[0, 1], [1, 2], [1, 3]]`. The `price` array is `[4, 2, 1, 3]`, which means the prices for nodes 0, 1, 2, and 3 are 4, 2, 1, and 3, respectively.

The tree would look like this:

```
    0
    |
    1
   / \
  2   3
```

Following the solution approach:

1. We first convert the `edges` array into a graph representation using an adjacency list. For the example, `g` will be `{0: [1], 1: [0, 2, 3], 2: [1], 3: [1]}`.

2. Initialize the global variable `ans` to 0. This will keep track of the maximum possible cost.

3. We define a recursive `dfs` function that, starting with the root (we can start with any node since it's a tree; let's pick node 0 for simplicity), will traverse the tree in a depth-first manner.

4. When we start the DFS from node 0, we find that it is connected to node 1. Since 0 is the root in this traversal, it has no parent, so we move to node 1.

5. At node 1, we explore its children, nodes 2 and 3. In the recursion for node 2, we find that the price sum `a` is 2 (price of node 1) + 1 (price of node 2) = 3, and `b` is just the price of node 2 itself, which is 1.

6. Similarly, DFS on node 3 gives a price sum `a` of 2 (price of node 1) + 3 (price of node 3) = 5, and `b` is again just the price of node 3, which is 3.

7. For node 1, which has now gathered information from its children, we calculate `a` as the max of its own price plus each child's `a`, giving us max(2 + 3, 2 + 5) = 7. The `b` for node 1 will be the max of `b` of its children since `b` represents the value excluding the node itself, so max(1, 3) = 3.

8. Every time the `dfs` function returns to node 1 from its children nodes 2 and 3, we check to update our global `ans`. Since node 1 is connected to node 0 with price 4, the potential maximums could be `a + b` from children, yielding potential values 4 + 3 (node 1's price plus the second-best price sum from its subtree) = 7. As `b` from node 0 is 0 (since it's the root in our DFS and doesn't have any price to add), we also consider 4 + 7 = 11. We take the max of these to update `ans`.

9. After exploring both branches of the tree, we find that the maximum possible price difference for node 1 as the root is 11.

10. In a complete solution, we would run such a DFS starting from each node, but since we know the tree structure doesn't actually change with different roots (just the way we calculate price sums), the `ans` we obtained is, in fact, the maximum cost after considering all possible root selections in our small example. For a larger tree, we would have to repeat the DFS from each node to cover all possible rootings.

By using DFS and dynamic programming, we can efficiently calculate the maximum possible cost for any rooting of the tree.

Python Solution

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

class Solution:
    def maxOutput(self, n: int, edges: List[List[int]], price: List[int]) -> int:
        # Depth First Search function to traverse graph and calculate maxOutput
        def dfs(node, parent):
            # Initialize current output and alternative output to the node's price
            current_output, alternative_output = price[node], 0

            # Explore all the connected nodes.
            for connected_node in graph[node]:
                if connected_node != parent:  # Ensuring we don't backtrack
                    max_with_current, max_without_current = dfs(connected_node, node)
                    # Update the maximum answer found so far by considering new paths
                    self.ans = max(self.ans, current_output + max_without_current, alternative_output + max_with_current)
                    # Update current max outputs
                    current_output = max(current_output, price[node] + max_with_current)
                    alternative_output = max(alternative_output, price[node] + max_without_current)

            return current_output, alternative_output

        # Build a graph as an adjacency list from edges
        graph = defaultdict(list)
        for start, end in edges:
            graph[start].append(end)
            graph[end].append(start)

        # Initialize the answer (max output) to zero
        self.ans = 0
        # Start DFS traversal from node 0 with no parent (-1)
        dfs(0, -1)

        # Return the answer after all possible max outputs are considered
        return self.ans
```

Java Solution

```java
class Solution {
    // Graph represented as adjacent lists
    private List<Integer>[] graph;
    // Max possible output stored globally
    private long maxPossibleOutput;
    // Prices associated with each node
    private int[] nodePrices;

    // Calculates the maximum output by traversing the graph
    public long maxOutput(int n, int[][] edges, int[] prices) {
        // Initialize the adjacencies list of the graph
        graph = new List[n];
        Arrays.setAll(graph, k -> new ArrayList<>());
        // Build graph with given edges
        for (int[] edge : edges) {
            int from = edge[0], to = edge[1];
            graph[from].add(to);
            graph[to].add(from);
        }
        // Assign the price array to the global variable
        this.nodePrices = prices;
        // Start the DFS from node 0 with no parent (-1 indicates no parent)
        dfs(0, -1);
        // Return the maximum output computed
        return maxPossibleOutput;
    }

    // Performs a DFS on the graph and returns the max production values
    private long[] dfs(int node, int parent) {
        // 'a' captures the max production when node 'i' is included
        long includeCurrent = nodePrices[node];
        // 'b' captures the max production when node 'i' is excluded
        long excludeCurrent = 0;
        // Traverse all the connected nodes
        for (int neighbor : graph[node]) {
            // If the neighbor is not the parent node
            if (neighbor != parent) {
                // Perform DFS on the neighbor
                long[] neighborValues = dfs(neighbor, node);
                long includeNeighbor = neighborValues[0];
                long excludeNeighbor = neighborValues[1];

                // Max output may include this node and exclude neighbor
                // or exclude this node but include neighbor
                maxPossibleOutput = Math.max(maxPossibleOutput,
                    Math.max(includeCurrent + excludeNeighbor, excludeCurrent + includeNeighbor));

                // Update local production values for the inclusion or exclusion of this node
                includeCurrent = Math.max(includeCurrent, nodePrices[node] + includeNeighbor);
                excludeCurrent = Math.max(excludeCurrent, nodePrices[node] + excludeNeighbor);
            }
        }
        // Return the max production values when this node is included and excluded
        return new long[]{includeCurrent, excludeCurrent};
    }
}
```

C++ Solution

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

class Solution {
public:
    // Calculate the maximum output for any node based on the provided graph and prices.
    long long maxOutput(int n, std::vector<std::vector<int>>& edges, std::vector<int>& prices) {
        // Adjacency list to represent the graph
        std::vector<std::vector<int>> graph(n);
        for (auto& edge : edges) {
            int from = edge[0], to = edge[1];
            graph[from].push_back(to);
            graph[to].push_back(from);
        }

        using ll = long long;
        using pll = std::pair<ll, ll>;
        ll answer = 0;

        // Depth-first search to explore the graph
        // It returns the maximum price choosing and not choosing the current node
        std::function<pll(int, int)> dfs = [&](int node, int parent) {
            ll chooseNode = prices[node];  // Max price when choosing the current node
            ll notChooseNode = 0;          // Max price when not choosing the current node

            for (int neighbor : graph[node]) {
                if (neighbor != parent) {
                    // Explore child nodes
                    auto [chooseChild, notChooseChild] = dfs(neighbor, node);

                    // Update the answer to the maximum output so far
                    answer = std::max({answer, chooseNode + notChooseChild, notChooseNode + chooseChild});

                    // Recurrence relations that update the max prices
                    chooseNode = std::max(chooseNode, prices[node] + chooseChild);
                    notChooseNode = std::max(notChooseNode, prices[node] + notChooseChild);
                }
            }
            return pll{chooseNode, notChooseNode};
        };

        // Initiate depth-first search from node 0, with no parent
        dfs(0, -1);

        // Return the maximum price as the answer
        return answer;
    }
};
```

Typescript Solution

```typescript
type Edge = [number, number];
type Prices = number[];
type Graph = number[][];
type Pair = [number, number]; // Represents a pair with two long numbers, used to hold maximum prices

// Adjacency list global variable to represent the graph
let graph: Graph = [];

// Function to calculate the maximum output for any node, using the provided graph and prices
function maxOutput(n: number, edges: Edge[], prices: Prices) : bigint {
    // Initialize the graph as an array of arrays with size n
    graph = new Array(n).fill(0).map(() => []);

    // Populate the graph with edges
    edges.forEach((edge) => {
        const [from, to] = edge;
        graph[from].push(to);
        graph[to].push(from);
    });

    // Variable to store the final result
    let answer: bigint = BigInt(0);

    // Depth-first search function that explores the graph
    // It returns a tuple with the maximum price when choosing or not choosing the current node
    function dfs(node: number, parent: number): Pair {
        let chooseNode: bigint = BigInt(prices[node]); // Max price when choosing the current node
        let notChooseNode: bigint = BigInt(0);        // Max price when not choosing the current node

        graph[node].forEach((neighbor) => {
            if (neighbor !== parent) {
                // Perform DFS on child nodes
                const [chooseChild, notChooseChild] = dfs(neighbor, node);

                // Update the answer with the maximum output so far
                answer = BigInt(Math.max(Number(answer), Number(chooseNode + notChooseChild), Number(notChooseNode + chooseChild)))

                // Update the max prices based on the recursive calls
                chooseNode = BigInt(Math.max(Number(chooseNode), Number(prices[node] + chooseChild)));
                notChooseNode = BigInt(Math.max(Number(notChooseNode), Number(prices[node] + notChooseChild)));
            }
        });
        return [chooseNode, notChooseNode];
    }

    // Start the DFS from node 0, assuming there's no parent for the root
    dfs(0, -1);

    // Return the maximum price computed
    return answer;
}
```

Time and Space Complexity

The given Python code defines a method `maxOutput` that calculates the maximum output based on certain conditions using a Depth First Search (DFS) algorithm.

Time Complexity:

The time complexity of the DFS is `O(N)`, where `N` is the number of nodes in the graph. This is because the DFS algorithm visits each node exactly once. Since the graph is represented using an adjacency list, and each edge is considered twice (once for each node it connects), running the DFS starting from the initial node (node 0) will result in traversing each edge two times — once for each direction.

However, inside the DFS, for each node, we loop through all its connected neighbors. The sum total of all the iterations through neighbors over the course of the entire DFS will equal the number of edges, which for an undirected graph is `2*(N-1)` (since it's a connected tree and there are `N-1` edges for `N` nodes).

Therefore, the overall time complexity of the code is `O(N)` because there's a constant amount of work done per edge (the number of edges is proportional to the number of nodes in a tree).

Space Complexity:

The space complexity consists of:

1. The space taken by the recursive call stack during DFS: In the worst case, this can be `O(N)` if the graph is structured as a linked list (degenerates to a linked list).
2. The space taken by the adjacency list `g`: This will also be `O(N)`, since it stores all `N-1` edges in both directions.
3. The space needed for any additional variables is constant and does not scale with `N`, so we can ignore this in our analysis.

Therefore, the space complexity of the method is `O(N)` for storing the graph and `O(N)` for the recursion stack, giving a combined space complexity of `O(N)`.