2538. Difference Between Maximum and Minimum Price Sum Tree Depth-First Search Array Dynamic Programming Leetcode Link Hard

Problem Description

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 ith node. The "price sum" of a path in the tree is the sum of prices of all nodes along that path.

undirected, connections between nodes are bidirectional. The nodes are labeled from 0 to n - 1. We are also given a 2D array edges

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

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.

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.

At each step, we compare the current node's price with the discovered price sums from its children. As DFS unwinds, we update the

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

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 1, 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

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

6. The global variable ans is updated during each call to dfs using the recursion stack to explore the subtree. It takes the maximum

represents the maximum price sum from the current node's subtree when it's not considered.

the maximum price sum from the subtree under j excluding itself.

bottom-up update representing dynamic programming.

programming states (a, b), avoiding any redundant calculations.

simplicity), will traverse the tree in a depth-first manner.

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.

Example Walkthrough

Following the solution approach:

the required answer.

Solution Approach

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

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 place 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.

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.

Throughout this implementation, the code maintains a single traversal of the tree using DFS while cleverly updating the dynamic

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:

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]}.

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

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,

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

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

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

any price to add), we also consider 4 + 7 = 11. We take the max of these to update ans.

def maxOutput(self, n: int, edges: List[List[int]], price: List[int]) -> int:

current_output, alternative_output = price[node], 0

return current_output, alternative_output

Build a graph as an adjacency list from edges

Initialize the answer (max output) to zero

Start DFS traversal from node 0 with no parent (-1)

// Calculates the maximum output by traversing the graph

// Initialize the adjacencies list of the graph

// Assign the price array to the global variable

// If the neighbor is not the parent node

// Perform DFS on the neighbor

return pll{chooseNode, notChooseNode};

// Return the maximum price as the answer

6 // Adjacency list global variable to represent the graph

// Populate the graph with edges

edges.forEach((edge) => {

// Initiate depth-first search from node 0, with no parent

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

// Return the maximum output computed

private long[] dfs(int node, int parent) {

// Traverse all the connected nodes

for (int neighbor : graph[node]) {

if (neighbor != parent) {

long includeCurrent = nodePrices[node];

// Start the DFS from node 0 with no parent (-1 indicates no parent)

// Performs a DFS on the graph and returns the max production values

// 'a' captures the max production when node 'i' is included

// 'b' captures the max production when node 'i' is excluded

Arrays.setAll(graph, k -> new ArrayList<>());

public long maxOutput(int n, int[][] edges, int[] prices) {

Return the answer after all possible max outputs are considered

Depth First Search function to traverse graph and calculate maxOutput

Initialize current output and alternative output to the node's price

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.

3, which is 3.

so we move to node 1.

to cover all possible rootings.

from typing import List

class Solution:

19

20

21

22

23

24

25

26

27

28

29

33

34

35

36

5

8

9

10

11

12

13

14

20

21

22

23

24

25

26

27

28

30

31

32

33

34

35

36

37

38

from collections import defaultdict

def dfs(node, parent):

graph = defaultdict(list)

for start, end in edges:

self.ans = 0

return self.ans

dfs(0, -1)

graph[start].append(end)

graph[end].append(start)

// Graph represented as adjacent lists

// Max possible output stored globally

private List<Integer>[] graph;

private long maxPossibleOutput;

private int[] nodePrices;

graph = new List[n];

this.nodePrices = prices;

return maxPossibleOutput;

long excludeCurrent = 0;

dfs(0, -1);

// Prices associated with each node

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

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

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

- By using DFS and dynamic programming, we can efficiently calculate the maximum possible cost for any rooting of the tree. Python Solution
- 10 11 # Explore all the connected nodes. 12 for connected_node in graph[node]: if connected_node != parent: # Ensuring we don't backtrack 13 max_with_current, max_without_current = dfs(connected_node, node) 14 # Update the maximum answer found so far by considering new paths 16 self.ans = max(self.ans, current_output + max_without_current, alternative_output + max_with_current) 17 # Update current max outputs current_output = max(current_output, price[node] + max_with_current) 18

alternative_output = max(alternative_output, price[node] + max_without_current)

Java Solution

```
// Build graph with given edges
15
            for (int[] edge : edges) {
16
                int from = edge[0], to = edge[1];
                graph[from].add(to);
17
18
                graph[to].add(from);
19
```

class Solution {

```
39
                     long[] neighborValues = dfs(neighbor, node);
 40
                     long includeNeighbor = neighborValues[0];
 41
                     long excludeNeighbor = neighborValues[1];
 42
 43
                     // Max output may include this node and exclude neighbor
 44
                     // or exclude this node but include neighbor
                     maxPossibleOutput = Math.max(maxPossibleOutput,
 45
 46
                         Math.max(includeCurrent + excludeNeighbor, excludeCurrent + includeNeighbor));
 47
 48
                     // Update local production values for the inclusion or exclusion of this node
                     includeCurrent = Math.max(includeCurrent, nodePrices[node] + includeNeighbor);
 49
                     excludeCurrent = Math.max(excludeCurrent, nodePrices[node] + excludeNeighbor);
 50
 51
 52
 53
             // Return the max production values when this node is included and excluded
             return new long[]{includeCurrent, excludeCurrent};
 54
 55
 56
 57
C++ Solution
    #include <vector>
  2 #include <functional>
     #include <algorithm>
    class Solution {
         // 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) {
  8
             // Adjacency list to represent the graph
  9
 10
             std::vector<std::vector<int>> graph(n);
             for (auto& edge : edges) {
 11
                 int from = edge[0], to = edge[1];
 12
 13
                 graph[from].push_back(to);
                 graph[to].push_back(from);
 14
 15
 16
 17
             using ll = long long;
 18
             using pll = std::pair<ll, ll>;
 19
             ll answer = 0;
 20
 21
             // Depth-first search to explore the graph
 22
             // It returns the maximum price choosing and not choosing the current node
 23
             std::function<pll(int, int)> dfs = [&](int node, int parent) {
 24
                 ll chooseNode = prices[node]; // Max price when choosing the current node
 25
                 ll notChooseNode = 0;
                                                // Max price when not choosing the current node
 26
 27
                 for (int neighbor : graph[node]) {
 28
                     if (neighbor != parent) {
 29
                         // Explore child nodes
 30
                         auto [chooseChild, notChooseChild] = dfs(neighbor, node);
 31
 32
                         // Update the answer to the maximum output so far
 33
                         answer = std::max({answer, chooseNode + notChooseChild, notChooseNode + chooseChild});
 34
 35
                         // Recurrence relations that update the max prices
 36
                         chooseNode = std::max(chooseNode, prices[node] + chooseChild);
 37
                         notChooseNode = std::max(notChooseNode, prices[node] + notChooseChild);
```

// 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 11 12 graph = new Array(n).fill(0).map(() => []); 13

14

15

38

39

40

41

42

43

44

45

46

47

48

50

49 };

};

Typescript Solution

2 type Prices = number[];

3 type Graph = number[][];

let graph: Graph = [];

1 type Edge = [number, number];

dfs(0, -1);

return answer;

```
16
             const [from, to] = edge;
            graph[from].push(to);
 17
            graph[to].push(from);
 18
 19
         });
 20
 21
        // Variable to store the final result
 22
         let answer: bigint = BigInt(0);
 23
 24
        // Depth-first search function that explores the graph
 25
        // It returns a tuple with the maximum price when choosing or not choosing the current node
 26
         function dfs(node: number, parent: number): Pair {
 27
             let chooseNode: bigint = BigInt(prices[node]); // Max price when choosing the current node
             let notChooseNode: bigint = BigInt(0);
 28
                                                    // Max price when not choosing the current node
 29
             graph[node].forEach((neighbor) => {
 31
                if (neighbor !== parent) {
 32
                    // Perform DFS on child nodes
                    const [chooseChild, notChooseChild] = dfs(neighbor, node);
 33
 34
 35
                    // Update the answer with the maximum output so far
                    answer = BigInt(Math.max(Number(answer), Number(chooseNode + notChooseChild), Number(notChooseNode + chooseChild)))
 36
 37
 38
                    // Update the max prices based on the recursive calls
                    chooseNode = BigInt(Math.max(Number(chooseNode), Number(prices[node] + chooseChild)));
 39
                    notChooseNode = BigInt(Math.max(Number(notChooseNode), Number(prices[node] + notChooseChild)));
 40
 41
 42
            });
 43
             return [chooseNode, notChooseNode];
 44
 45
 46
        // Start the DFS from node 0, assuming there's no parent for the root
 47
         dfs(0, -1);
 48
 49
         // Return the maximum price computed
 50
         return answer;
 51 }
 52
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
```

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

The space complexity consists of:

Space Complexity:

direction.

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).

connects), running the DFS starting from the initial node (node 0) will result in traversing each edge two times — once for each

However, inside the DFS, for each node, we loop through all its connected neighbors. The sum total of all the iterations through

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.

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

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).