2477. Minimum Fuel Cost to Report to the Capital

Medium Tree Depth-First Search Breadth-First Search Graph Leetcode Link

Problem Description

In this problem, we are presented with a country that has `n` cities arranged in a tree structure, meaning all cities are connected by roads in such a way that there is exactly one unique path between any two cities, and there are no cycles. The cities are numbered from `0` to `n - 1`, where city `0` is the capital. The connections between cities are described by the `roads` array, where each element represents a bidirectional road between two cities.

A meeting is scheduled to take place in the capital city, and every city needs to send a representative to the meeting. Each city has one car with a certain number of `seats` which can be used to travel to the meeting. When traveling from one city to another, it costs one liter of fuel. If necessary, representatives can switch cars by riding with another representative from a different city.

The goal is to calculate the minimum amount of fuel needed for all representatives to reach the capital city for the meeting.

Intuition

To approach this problem, we have to recognize that because the cities are connected in a tree structure, there is only one path for each representative to take to the capital city, making the task of finding the path straightforward. The challenge is finding out the optimal arrangement of representatives and cars to minimize fuel consumption.

Since this is a tree structure, we can use the Depth First Search (DFS) algorithm. We start from the capital city and explore each branch, that is, each connected city recursively. At each city, we calculate the number of representatives that need to be transferred from that city to the capital city, including the representatives from its connected cities.

The key insight to minimize fuel is to fill each car to its maximum capacity before sending it to the capital. This ensures the least number of trips is made, and subsequently, the minimum amount of fuel is used.

The intuition behind the solution code is that we perform a DFS starting from the capital. During the DFS, we keep track of the total number of representatives that need to be moved from each branch (i.e., from each city and its descendants). As representatives are transported to the capital city, we calculate the number of cars needed to move them based on the capacity of each car (`seats`). Each car that needs to move from a city to the capital city uses one liter of fuel, hence we increase the fuel count by the number of cars needed for that branch.

The `dfs` function updates the total fuel cost (`ans`) as it calculates the number of representatives that need to be moved from each city to its parent. The calculation for the number of cars needed is done by `(t + seats - 1) // seats`, where `t` is the number of representatives, accounting for the last car that may not be full.

Solution Approach

The implementation of the solution primarily relies on a recursive Depth-First Search (DFS) algorithm. The algorithm traverses the input tree, starting from the capital city (node `0`), going through each city and figuring out the number of seats required to transport the representatives from the subtree rooted at that city to the capital.

The code uses the following elements in its approach:

- A `defaultdict` from the `collections` module to represent the graph which describes the tree structure of the country. Keys are cities, and values are lists of neighboring cities effectively forming the adjacency list for each node.
- A recursive `dfs` function, which serves as the DFS traversal method, taking two parameters: the current city `a`, and its parent city `fa`. The parent city helps in avoiding traversing back on the path we came from.
- A nonlocal variable `ans`, which keeps track of the total fuel cost calculated during the DFS traversal. The variable is defined as `nonlocal` because it needs to be accessed and modified within the nested `dfs` function.

The `dfs` function works as follows:

1. For each city `a`, it initializes a local variable `size` to `1`, which implicitly includes the city's own representative.
2. It iterates over all connected cities `b` in `g[a]`. If `b` is not the city we just visited (`fa`), it recursively calls `dfs(b, a)`.
3. Within each recursive call, it computes the total number of representatives `t` (including those from the subtrees) that need to travel from city `b` through city `a` toward the capital.
4. The fuel cost for the subtree rooted at `b` is then updated by adding `(t + seats - 1) // seats` to `ans`. The expression `(t + seats - 1) // seats` calculates the smallest number of cars needed to transport `t` representatives given each car has `seats` number of seats. It uses integer division rounding up, which is achieved by adding `seats - 1` before dividing.
5. The function then returns the size of the subtree rooted at the current city `a` to its parent call which aggregates sizes from all child calls.

After defining the `dfs` and `g`, the function calls `dfs(0, -1)` to start the traversal from the capital city (which has no parent, hence `-1` is used), and finally returns `ans`.

This algorithm ensures that each representative is counted exactly once and the minimum number of cars needed to transport representatives from each subtree to the root (capital city) is calculated, thus leading to the calculation of the minimum fuel cost.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have a country with 5 cities (n = 5) arranged in a tree structure with the following connections and seats in cars:

- Roads: [(0, 1), (0, 2), (1, 3), (1, 4)]
- Seats: 4 for all cars (for simplicity)

The tree structure based on the roads array is as follows:

```
    0 (capital)
   / \
  1   2
 / \
3   4
```

Each city except for the capital has one representative who needs to get to the capital. Now, let's walk through the DFS approach:

1. We start the DFS from the capital city (0). City 0 has 2 neighbors: city 1 and city 2.
2. We explore city 1 (which has not been visited). City 1 has 2 neighbors: city 3 and city 4.
3. We go to city 3 (not visited), and it has no neighbors. We determine one representative needs to travel from city 3 to city 1. Since each car has 4 seats, only one car (and thus one liter of fuel) is used. The DFS function returns a subtree size for city 3 as 1 (just the representative from city 3).
4. Back to city 1, we next explore city 4 with a similar process as city 3. One liter of fuel is used, and the subtree size is 1.
5. Now back at city 1, we have collected the subtree sizes from city 3 and city 4. The total number of representatives at city 1 is 1 (its own) + 1 (from city 3) + 1 (from city 4) = 3. Only one car with 4 seats is needed, so one more liter of fuel is used to move these representatives to the capital.
6. We go back to the capital and move to city 2. It has no neighbors to explore, so we use one car and one liter of fuel to transport the representative to the capital. The total size of city 2's subtree is 1.
7. Finally, we collect the total fuel cost as the sum of the fuel needed for the subtrees: 1 (from city 3) + 1 (from city 4) + 1 (from city 1 to capital) + 1 (from city 2 to capital) = 4 liters.

The answer for our example is 4 liters as the minimum amount of fuel needed for all representatives to reach the capital.

Python Solution

```python
from collections import defaultdict

class Solution:
    def minimumFuelCost(self, roads, seats):
        # Recursive function to perform depth-first search (DFS) on the graph.
        def dfs(current, parent):
            # Initialize the size of the current subtree, starting with 1 for the current node.
            subtree_size = 1

            # Iterate through adjacent nodes of the current node.
            for next_node in graph[current]:
                # Avoid traversing the edge leading back to the parent node.
                if next_node != parent:
                    # Recursively call dfs for the adjacent node.
                    child_size = dfs(next_node, current)

                    # Update the total number of trips required to move people from the current subtree,
                    # ensuring each trip has no more people than the number of available seats.
                    self.total_trips += (child_size + seats - 1) // seats

                    # Update the size of the current subtree by adding the size of the child subtree.
                    subtree_size += child_size

            # Return the size of the current subtree.
            return subtree_size

        # Initialize graph as an adjacency list representation.
        graph = defaultdict(list)
        # Build the graph using roads information.
        for start, end in roads:
            graph[start].append(end)
            graph[end].append(start)

        # The variable to store the total number of trips required.
        self.total_trips = 0

        # Start DFS from node 0 assuming 0 as the root with no parent (-1).
        dfs(0, -1)

        # Return the total number of trips calculated.
        return self.total_trips

# Example usage:
# sol = Solution()
# print(sol.minimumFuelCost([[0, 1], [1, 2]], 2))  # Example call to the method.
```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {
    private List<Integer>[] graph; // "g" is renamed to "graph" for clarity
    private long totalFuelCost; // "ans" is renamed to "totalFuelCost" for clarity
    private int maxSeats; // "seats" is renamed to "maxSeats" for clarity

    // Function to calculate the minimum fuel cost
    public long minimumFuelCost(int[][] roads, int seats) {
        int nodeCount = roads.length + 1; // Calculate the number of nodes
        graph = new List[nodeCount]; // Initialize the graph
        Arrays.setAll(graph, k -> new ArrayList<>()); // Create a list for each node
        maxSeats = seats; // Set the maximum number of seats
        for (int[] edge : roads) { // Iterate over all roads
            int from = edge[0], to = edge[1]; // Get the from and to nodes for each road
            graph[from].add(to); // Add the to node into the from node's list
            graph[to].add(from); // Add the from node into the to node's list
        }
        dfs(0, -1); // Perform a DFS traversal starting from node 0 with no parent (-1)
        return totalFuelCost; // Return the total fuel cost
    }

    // Depth-First Search function to traverse the graph and calculate fuel cost
    private int dfs(int node, int parent) {
        int subtreeSize = 1; // Initialize the subtree size (1 for the current node)
        for (int adjacent : graph[node]) { // Iterate over the adjacent nodes
            if (adjacent != parent) { // Check if it's not coming back to the parent
                int childTreeSize = dfs(adjacent, node); // Recursively call dfs for the child node
                // Calculate and add fuel cost for reaching the child node's subtree
                totalFuelCost += (long) Math.ceil((double) childTreeSize / maxSeats);
                subtreeSize += childTreeSize; // Update the size of the current subtree
            }
        }
        return subtreeSize; // Return the size of the subtree rooted at the current node
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to calculate the minimum fuel cost for a road network
    // 'roads' represent connections where roads[i] = [a, b] is a road between a and b
    // 'seats' represents the number of seats in the vehicle
    long long minimumFuelCost(vector<vector<int>>& roads, int seats) {
        // Number of stops in the network is one more than number of roads
        int numStops = roads.size() + 1;

        // Graph 'g' to represent the network connections
        vector<vector<int>> graph(numStops);

        // Building the adjacency list for the graph
        for (auto& edge : roads) {
            int from = edge[0], to = edge[1];
            graph[from].emplace_back(to);
            graph[to].emplace_back(from);
        }

        // Variable to store the total fuel cost
        long long totalFuelCost = 0;

        // Depth-first search function to calculate the size of each subtree and add fuel costs
        function<int(int, int)> dfs = [&](int currentNode, int parentNode) -> int {
            int subtreeSize = 1; // Each stop contributes size 1 to the subtree

            // Traverse all the connected stops
            for (int nextNode : graph[currentNode]) {
                if (nextNode != parentNode) {
                    // Calculate the size of child subtree
                    int childSize = dfs(nextNode, currentNode);
                    // Add fuel cost for trips from this child
                    totalFuelCost += (childSize + seats - 1) / seats;
                    // Increment subtree size with the size of the child
                    subtreeSize += childSize;
                }
            }
            return subtreeSize;
        };

        // Start DFS from stop 0, considering it as the root and no parent (-1)
        dfs(0, -1);

        // Return the total fuel cost after performing DFS
        return totalFuelCost;
    }
};
```

Typescript Solution

```typescript
// Importing necessary functionalities from libraries
import { Vector } from "prelude-ts";

// Function to calculate the size of each subtree and add fuel costs using DFS
function depthFirstSearch(currentNode: number, parentNode: number, graph: Vector<Vector<number>>, seats: number): number {
    let subtreeSize: number = 1; // Each stop contributes size 1 to the subtree

    // Get all the connected stops for the current node
    const connectedStops: Vector<number> = graph.get(currentNode).getOrElse(Vector.empty());

    // Traverse all the connected stops and avoid visiting the parent node again
    connectedStops.forEach((nextNode: number) => {
        if (nextNode !== parentNode) {
            // Calculate the size of child subtree
            const childSize: number = depthFirstSearch(nextNode, currentNode, graph, seats);
            // Add fuel cost for trips from this child, round up trips given the number of seats
            totalFuelCost += Math.ceil(childSize / seats);
            // Increment subtree size with the size of the child
            subtreeSize += childSize;
        }
    });

    return subtreeSize;
}

// Function to calculate the minimum fuel cost for a road network
function minimumFuelCost(roads: Vector<Vector<number>>, seats: number): number {
    // Number of stops in the network is one more than the number of roads
    const numStops: number = roads.length() + 1;

    // Initialise a graph to represent the network connections as an adjacency list
    let graph: Vector<Vector<number>> = Vector.fill(numStops, () => Vector.empty<number>());

    // Building the adjacency list for the graph from the road connections provided
    roads.forEach((edge: Vector<number>) => {
        const from: number = edge.get(0).getOrElse(0);
        const to: number = edge.get(1).getOrElse(0);
        graph = graph.updateAt(from, (edges: Vector<number>) => edges.append(to));
        graph = graph.updateAt(to, (edges: Vector<number>) => edges.append(from));
    });

    // Variable to store the total fuel cost, initialised to 0
    totalFuelCost = 0;

    // Start DFS from stop 0, considering it as the root with no parent (-1)
    depthFirstSearch(0, -1, graph, seats);

    // Return the total fuel cost after performing DFS
    return totalFuelCost;
}

// Global variable to store the total fuel cost
let totalFuelCost: number = 0;
```

Time and Space Complexity

The time complexity of the given Python code is `O(N)`, where `N` is the number of nodes (intersections) in the road network represented as a tree. This is because each node is visited exactly once thanks to the Depth-First Search (DFS) approach used in the function `dfs`. In each call to `dfs`, it goes through all the edges connected to a node exactly once to calculate the number of fuel stops needed for each subtree.

The space complexity of the code is `O(N)` as well. The additional space used is for the recursive call stack of the `dfs` function and the adjacency list `g` that stores the tree. Since the tree is not heavily skewed, the height of the tree (which impacts the call stack size) will not be more than `N` in the worst case (a straight line tree). Thus, the recursive call stack size is at most `N`. The adjacency list also stores up to `2(N - 1)` edges (since every edge is stored twice, once for each node it connects), which is within a constant factor of the number of nodes.

So, the overall space complexity remains linear with the number of nodes `O(N)`.