2285. Maximum Total Importance of Roads

Medium Greedy Graph Sorting Heap (Priority Queue) Leetcode Link

Problem Description

In this LeetCode problem, we are tasked with assigning values to cities such that the total importance of all roads is maximized. Each city is represented by an integer from `0` to `n - 1`, where `n` is the total number of cities. There is also a list of roads represented as pairs `[a_i, b_i]`, which corresponds to a bidirectional road between cities `a_i` and `b_i`. The importance of a single road is the sum of the values assigned to the two cities it connects.

Constraints of the Problem:

- Each city must be assigned a unique value from `1` to `n`, and no value may be used more than once.
- The goal is to find an assignment of values that results in the highest possible total road importance, which is the sum of importances of all individual roads.

We need to find the optimal way to assign these values in order to achieve the maximum total importance of all roads.

Intuition

A key observation for this problem is that roads connected to cities with high degrees of connectivity (i.e., cities with many incoming and outgoing roads) should be assigned higher values in order to maximize the total importance. The reasoning is that a high-value city will contribute its value to the total importance of the network multiple times – once for each road it is connected to.

To derive an optimal solution, we can follow these steps:

1. Calculate the degree of each city, which is the number of roads connected to it.
2. Sort the cities based on their degree in non-decreasing order. This ensures that cities with the most roads are considered last.
3. Assign values starting from `1` to `n` based on the sorted degrees; cities with lower degrees get lower values and those with higher degrees get higher values.
4. Calculate the sum of the importance for each road using the assigned values.

This approach works because it maximizes the contribution of the most connected cities to the total road importance by ensuring that they are assigned the highest possible values.

The provided solution code implements this strategy effectively by first initializing an array `deg` to keep track of the degree for each city. It then iterates over the list of roads, incrementing the degree of each connected city. After sorting the `deg` array, it uses a simple loop to sum up the product of each value with its corresponding degree. By starting the enumeration from `1`, we ensure that the lowest value assigned to cities is `1` and it goes up to `n`, as per the constraints.

Solution Approach

The solution approach can be broken down into the following steps:

1. **Initialization**: A list `deg` of length `n` is initialized to store the degree of each city. The degree of a city is defined as the number of roads that connect it to other cities.

   ```
   deg = [0] * n
   ```

2. **Calculating Degrees**: The given list of roads is iterated over, and for each road `[a, b]`, the degree of city `a` and city `b` is incremented by 1 since the road is bidirectional and connects both cities.

   ```
   for a, b in roads:
       deg[a] += 1
       deg[b] += 1
   ```

3. **Sorting**: The `deg` list is sorted in non-decreasing order. This ensures that cities with higher degrees, which should receive higher values, are at the end of the list.

   ```
   deg.sort()
   ```

4. **Assigning Values**: An `enumerate` loop is used, starting from `1`, to calculate the sum of the products of the indices and the corresponding degrees. The index `i` effectively represents the value to be assigned to the city (as per sorted degrees), and `v` represents the degree of that city.

   ```
   return sum(i * v for i, v in enumerate(deg, 1))
   ```

   Since the degree array is sorted, the city with the lowest degree is given the value `1`, the next one is given `2`, and so on, until the city with the highest degree is given the value `n`.

This approach guarantees that the sum of the road importances is maximized because it assigns the highest values to the cities with the highest degrees. The sorting step is essential to accomplish this allocation efficiently; otherwise, we would have to manually choose the highest values for the cities with the most roads.

Computational Complexity:

- The computational complexity of counting the degrees is `O(E)`, where `E` is the number of roads since we loop through each road only once.
- Sorting the `deg` list takes `O(N log N)` time, where `N` is the number of cities.
- The final enumeration and sum take `O(N)` time.

Hence, the overall time complexity of the solution is dominated by the sorting step, making it `O(N log N)`. The space complexity is `O(N)` due to the extra `deg` list used to store the degrees.

Example Walkthrough

Assume we have `n = 4` cities and the following list of bidirectional roads: `roads = [[0, 1], [0, 2], [0, 3], [1, 2]]`. Our task is to assign integer values to these cities to maximize the total importance of the roads where the importance of each road is the sum of the values of the cities it connects.

1. **Initialization**: We start by initializing a list `deg` to store the degree (number of roads connected) for each of the `4` cities.

   ```
   deg = [0] * 4  # [0, 0, 0, 0]
   ```

2. **Calculating Degrees**: For each road, increment the degree of both cities involved in the road by `1`.

   ```
   # roads = [[0, 1], [0, 2], [0, 3], [1, 2]]
   deg[0] += 1  # For road [0, 1]
   deg[1] += 1
   deg[0] += 1  # For road [0, 2]
   deg[2] += 1
   deg[0] += 1  # For road [0, 3]
   deg[3] += 1
   deg[1] += 1  # For road [1, 2]
   deg[2] += 1
   # Now deg = [3, 2, 2, 1]
   ```

3. **Sorting**: Sort the `deg` list in non-decreasing order.

   ```
   deg.sort()  # [1, 2, 2, 3]
   ```

4. **Assigning Values**: Assign values from `1` to `n` following the sorted order of degrees and calculate the total importance.

   ```
   # Using enumerate starting from 1, assign values and calculate total importance
   total_importance = sum(i * v for i, v in enumerate(deg, 1))
   # total_importance = 1*1 + 2*2 + 3*2 + 4*3 = 1 + 4 + 6 + 12 = 23
   ```

The total importance of all roads with this assignment is `23`, which is the maximum we can achieve. To see if it's the best assignment:

- City `0` has the highest degree, `3`, so it gets the highest value, `4`.
- Cities `1` and `2` each have a degree of `2`, so they get the next highest values, `3` and `2`.
- City `3` has the lowest degree, `1`, thus it gets the lowest value, `1`.

The sum of the importances for the roads is:

- For road `[0, 1]`: `4 (city 0) + 3 (city 1) = 7`
- For road `[0, 2]`: `4 (city 0) + 2 (city 2) = 6`
- For road `[0, 3]`: `4 (city 0) + 1 (city 3) = 5`
- For road `[1, 2]`: `3 (city 1) + 2 (city 2) = 5`

Add these up for a total importance of `7 + 6 + 5 + 5 = 23`.

By following the solution approach, we've maximized the total importance by giving the cities with more connections higher values, and this example demonstrates that the total sum of the importance is indeed optimized.

Python Solution

```python
class Solution:
    def maximumImportance(self, n: int, roads: List[List[int]]) -> int:
        # Initialize a list to store the degree of each city.
        # Degree means the number of direct connections to other cities
        city_degrees = [0] * n

        # Calculate the degree for each city by iterating over each road
        for road in roads:
            # Increment the degree for both cities in the road connection
            city_degrees[road[0]] += 1
            city_degrees[road[1]] += 1

        # Sort the degrees in ascending order so that the city with the highest
        # degree gets the highest importance value
        city_degrees.sort()

        # Compute the total importance score
        # Importance is calculated by multiplying each city's degree by its
        # importance rank which is determined by its position in the sorted list
        total_importance = sum(importance * degree for importance, degree in enumerate(city_degrees, 1))

        return total_importance

```

Java Solution

```java
class Solution {

    // Function to calculate the maximum importance of the city network
    public long maximumImportance(int n, int[][] roads) {
        // Array to store the degree (number of roads) of each city
        int[] degree = new int[n];

        // Increment the degree for both cities involved in each road
        for (int[] road : roads) {
            ++degree[road[0]];
            ++degree[road[1]];
        }

        // Sort the array to process cities with smaller degrees first
        Arrays.sort(degree);

        // Initialize answer to calculate the sum of importances
        long answer = 0;

        // Assign importance values to cities. The importance starts from 1 and goes up to n.
        // The cities with the smallest degrees get the smallest importance values.
        for (int i = 0; i < n; ++i) {
            // Calculate importance for current city and add it to the answer.
            // Importance is calculated by multiplying the city's degree with its importance value.
            answer += (long) (i + 1) * degree[i];
        }

        // Return the total importance for all cities
        return answer;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the vector header for using vectors
#include <algorithm> // Include the algorithm header for sorting

class Solution {
public:
    // Calculates the maximum importance of the cities based on the roads
    long long maximumImportance(int n, vector<vector<int>>& roads) {
        // Create a vector to store the degree (number of roads) for each city
        vector<int> degreeCounts(n, 0);

        // Iterate through each road to increment the degree of the two cities connected by the road
        for (const auto& road : roads) {
            ++degreeCounts[road[0]];
            ++degreeCounts[road[1]];
        }

        // Sort the degree counts in non-decreasing order
        sort(degreeCounts.begin(), degreeCounts.end());

        // Initialize a variable to store the maximum importance sum
        long long maxImportance = 0;

        // Assign the importance to each city based on its degree count
        // The city with the smallest degree gets an importance of 1, the next one gets 2, and so on
        for (int i = 0; i < n; ++i) {
            maxImportance += static_cast<long long>(i + 1) * degreeCounts[i];
        }

        // Return the calculated maximum importance sum
        return maxImportance;
    }
};

```

Typescript Solution

```typescript
// Importing the `Sort` from array helper functions to use sorting
import { Sort } from "./arrayHelpers";

// Function to calculate the maximum importance of the cities based on the roads
function maximumImportance(n: number, roads: number[][]): number {
    // Create an array to store the number of roads connecting to each city
    let degreeCounts: number[] = new Array(n).fill(0);

    // Iterate through each road to increment the road count for the connected cities
    roads.forEach(road => {
        degreeCounts[road[0]]++;
        degreeCounts[road[1]]++;
    });

    // Sort the degree counts in non-decreasing order
    degreeCounts.sort((a, b) => a - b);

    // Initialize a variable to store the maximum importance sum
    let maxImportance: number = 0;

    // Assign importance to each city based on its road count
    // The city with the smallest number of connecting roads gets an importance of 1, the next one gets 2, and so on
    for (let i = 0; i < n; i++) {
        maxImportance += (i + 1) * degreeCounts[i];
    }

    // Return the calculated maximum importance sum
    return maxImportance;
}

// Note: The above code assumes the existence of a 'Sort' method for sorting arrays, which is not a built-in function in TypeScript.
// A sorting function would need to be implemented or imported from a utility library for the sorting to work correctly.
```

In TypeScript, there's no native `Sort` method as it's written in the comment, because TypeScript leverages the JavaScript array `sort` method directly. The import at the top is demonstrating the structure but isn't needed for native JavaScript/TypeScript `sort` method. Here's the correct implementation without an import:

```typescript
// Function to calculate the maximum importance of the cities based on the roads
function maximumImportance(n: number, roads: number[][]): number {
    let degreeCounts: number[] = new Array(n).fill(0);

    roads.forEach(road => {
        degreeCounts[road[0]]++;
        degreeCounts[road[1]]++;
    });

    degreeCounts.sort((a, b) => a - b);

    let maxImportance: number = 0;

    for (let i = 0; i < n; i++) {
        maxImportance += (i + 1) * degreeCounts[i];
    }

    return maxImportance;
}

```

Time and Space Complexity

Time Complexity:

The time complexity of the code consists of the following parts:

- Initializing the degree list `deg` with `n` zeros has a time complexity of `O(n)`.
- Iterating over the list of roads to calculate the degree for each node: As there is one iteration over the list of roads, where the list's length is the number of roads `r`, this part has a time complexity of `O(r)`.
- Sorting the `deg` list: The sorting operation has a time complexity of `O(n log n)`, as Python uses TimSort (a hybrid sorting algorithm derived from merge sort and insertion sort) for sorting lists.
- The list comprehension with `enumerate` to compute the sum of importance: This iteration is done over the `n` elements of the sorted `deg` list, so its time complexity is `O(n)`.

The overall time complexity is determined by the largest of these, which is `O(n log n)` due to the sorting operation. Thus, the total time complexity is `O(n log n)`.

Space Complexity:

The space complexity of the code includes:

- The `deg` list, which consumes `O(n)` space.
- The internal space required for sorting, which in Python's sort implementation can be `O(n)`.
- The space required for the list in list comprehension when `enumerate` is called. However, since there's no separate list being created and it is an iterator, it does not add to the space complexity.

Considering all the above, the overall space complexity of the function is `O(n)`. This is because, while there is temporary space used during sorting, the main permanent space used is the `deg` array which is linear in size to the number of nodes `n`.