1615. Maximal Network Rank

Medium Graph

Leetcode Link

Problem Description

The problem defines an infrastructure consisting of `n` cities and some roads connecting these cities in a bidirectional manner. Given this setup, the concept of "network rank" for a pair of different cities is introduced. The network rank for any two different cities is the total number of roads that are connected to either of the two cities. However, if a road connects both cities directly, it is only counted once for their joint network rank.

Your task is to calculate the "maximal network rank" for the entire infrastructure. This is defined as the maximum network rank that can be achieved between any pair of different cities within the network. To find this, you are provided with two inputs: an integer `n`, representing the number of cities, and an array `roads`, where each element is a pair of integers representing a road between two cities.

The goal is to return the maximal network rank, which represents the highest connectivity between any two cities in the infrastructure, disregarding if any road is counted more than once across different pairs.

Intuition

To figure out the solution to this problem, we start by considering the straightforward approach for calculating the network rank of all possible pairs of different cities. We can do that by creating a graph representation where each node corresponds to a city, and the edges represent the roads between them.

The intuitive way to represent the graph is using a data structure where for each city, we maintain a set of the cities that are directly connected to it by a road. This can be done using a dictionary of sets in Python, represented as `g` in the provided solution.

Once we have the graph, we can iterate through all pairs of cities. For each pair, we calculate the sum of the roads connected to city A and the roads connected to city B. If there is a direct road between city A and B, we should decrease this sum by one because that road is counted twice - once for each city.

By iterating over each possible pair of different cities, we can keep track of the maximal network rank found. The intuition behind this approach is that we want to consider all the unique contributions of roads for every possible pair and find the pair with the highest connectivity.

The value is then stored in the variable `ans`, which gets updated every time we find a pair with a higher network rank. After considering all pairs, `ans` will hold the maximal network rank value, which is what we will return.

Solution Approach

The solution approach is based on a graph representation and iteration over all possible city pairs to find the maximal network rank. Here are the steps and key components of the algorithm:

1. **Graph Representation**: The graph is stored in a dictionary called `g`. Each key in this dictionary represents a city, and the corresponding value is a set of all cities that have direct roads connected to this city. The use of a set here is to ensure unique entries and to easily check for the presence of a direct road between any two cities.

```
g = defaultdict(set)
for a, b in roads:
    g[a].add(b)
    g[b].add(a)
```

2. **Initialize Maximal Network Rank**: A variable `ans` is initialized to 0 to keep track of the maximal network rank encountered during iteration.

```
ans = 0
```

3. **Iterating Over City Pairs**: Two nested loops are used to iterate over all possible pairs of different cities. This is done by iterating through all cities as `a` in the outer loop and all cities after `a` as `b` in the inner loop to ensure that each pair is only considered once.

```
for a in range(n):
    for b in range(a + 1, n):
```

4. **Calculating the Network Rank**: For each pair of cities `(a, b)`, the network rank is computed as the sum of the number of cities connected to `a` and the number of cities connected to `b`. If city `b` is directly connected to city `a`, one is subtracted from the sum since that road is counted in both sets of connections.

```
if (t := len(g[a]) + len(g[b]) - (a in g[b])) > ans:
    ans = t
```

Using Python's walrus operator `:=`, we can calculate and compare the network rank in one line. The network rank for the pair is stored temporarily in the variable `t`, and if this is greater than the current `ans`, it means a new maximal network rank has been found, so `ans` is updated accordingly.

5. **Return the Result**: After all pairs have been considered, the highest value of `ans` reflects the maximal network rank for the infrastructure.

```
return ans
```

This solution ensures that all city pairs are considered and the maximal network rank is found in an efficient manner. The time complexity of this solution is O(n^2) due to the nested loops over the city pairs, which is reasonable considering the pairwise nature of the network rank calculation.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach described previously. Consider an infrastructure with `n = 4` cities and the following array of `roads` connecting them:

```
roads = [[0, 1], [0, 3], [1, 2], [1, 3]]
```

In this case, we have 4 cities and 4 roads. City 0 is connected to cities 1 and 3, city 1 is connected to cities 0, 2, and 3, city 2 is connected to city 1, and city 3 is connected to cities 0 and 1.

Step 1: Graph Representation

We represent our graph as a dictionary where each key is a city, and the corresponding value is a set of cities that it is directly connected to.

```
g = defaultdict(set)
g[0] = {1, 3}
g[1] = {0, 2, 3}
g[2] = {1}
g[3] = {0, 1}
```

Step 2: Initialize Maximal Network Rank

We initialize our `ans` variable to zero.

```
ans = 0
```

Step 3: Iterating Over City Pairs

We then loop over all pairs of different cities `(a, b)`. Taking `n = 4`, we would loop over pairs `(0, 1)`, `(0, 2)`, `(0, 3)`, `(1, 2)`, `(1, 3)`, and `(2, 3)`.

Step 4: Calculating the Network Rank

As we iterate over the pairs, we calculate the network rank for each pair:

- For `(0, 1)`: `len(g[0]) + len(g[1]) - (0 in g[1])` => `2 + 3 - 1 = 4`.
- For `(0, 2)`: `len(g[0]) + len(g[2]) - (0 in g[2])` => `2 + 1 - 0 = 3`.
- For `(0, 3)`: `len(g[0]) + len(g[3]) - (0 in g[3])` => `2 + 2 - 1 = 3`.
- For `(1, 2)`: `len(g[1]) + len(g[2]) - (1 in g[2])` => `3 + 1 - 1 = 3`.
- For `(1, 3)`: `len(g[1]) + len(g[3]) - (1 in g[3])` => `3 + 2 - 1 = 4`.
- For `(2, 3)`: `len(g[2]) + len(g[3]) - (2 in g[3])` => `1 + 2 - 0 = 3`.

We find that the maximal network rank for the pairs `(0, 1)` and `(1, 3)` is 4.

Step 5: Return the Result

After considering all pairs, we conclude that the maximal network rank is 4, as this is the highest network rank found from the pairwise considerations.

The returned result from our approach would be:

```
return ans # which is 4 in this case
```

Python Solution

```
from collections import defaultdict
from typing import List

class Solution:
    def maximalNetworkRank(self, n: int, roads: List[List[int]]) -> int:
        # Create a graph using a dictionary, where each node points to a set of connected nodes
        graph = defaultdict(set)

        # Build the graph with the given list of roads
        for a, b in roads:
            graph[a].add(b)
            graph[b].add(a)

        # Initialize the maximum network rank to zero
        max_network_rank = 0

        # Check every pair of cities to find the maximal network rank
        for city_a in range(n):
            for city_b in range(city_a + 1, n):
                # Calculate the network rank for the pair (a, b), which is the sum of
                # their connected cities minus one if they are directly connected
                connected_cities = len(graph[city_a]) + len(graph[city_b])
                if city_a in graph[city_b]:
                    connected_cities -= 1

                # Update the maximum network rank if this pair has a higher rank
                if connected_cities > max_network_rank:
                    max_network_rank = connected_cities

        # Return the maximum network rank
        return max_network_rank
```

Java Solution

```
class Solution {
    public int maximalNetworkRank(int n, int[][] roads) {
        int[][] graph = new int[n][n];
        int[] count = new int[n];
        // Fill the graph matrix and count array
        for (int[] road : roads) {
            int city_a = road[0];
            int city_b = road[1];
            graph[city_a][city_b] = 1; // Mark the road as existing between city_a and city_b
            graph[city_b][city_a] = 1; // Mark the road as existing between city_b and city_a
            // Increment the count of roads connected to city_a and city_b
            ++count[city_a];
            ++count[city_b];
        }

        int maximalNetworkRank = 0;
        // Iterate through all pairs of cities to find
        // the maximal network rank of the transportation network
        for (int city_a = 0; city_a < n; ++city_a) {
            for (int city_b = city_a + 1; city_b < n; ++city_b) {
                // Calculate the network rank for the pair (city_a, city_b)
                // The road between them if it exists
                int networkRank = count[city_a] + count[city_b] - graph[city_a][city_b];
                // Update the maximum network rank if the current rank is higher
                maximalNetworkRank = Math.max(maximalNetworkRank, networkRank);
            }
        }
        // Return the maximum network rank found
        return maximalNetworkRank;
    }
}
```

C++ Solution

```
#include <vector> // Include vector header for using std::vector
#include <string> // Include string header for using memset
#include <algorithm> // Include algorithm header for using std::max

class Solution {
public:
    int maximalNetworkRank(int n, vector<vector<int>>& roads) {
        // Initialize count and graph matrices using vectors for dynamic size allocation
        vector<int> count(n, 0);
        vector<vector<int>> graph(n, vector<int>(n, 0));

        // Iterate over the roads to populate the graph and counts of connected roads for each city
        for (auto& road : roads) {
            int city_a = road[0], city_b = road[1];
            graph[city_a][city_b] = graph[city_b][city_a] = 1; // Mark the cities as connected
            ++count[city_a]; // Increment road count for city_a
            ++count[city_b]; // Increment road count for city_b
        }

        int maxRank = 0; // Initialize the max network rank

        // Iterate over pairs of cities to calculate the network rank
        for (int city_a = 0; city_a < n; ++city_a) {
            for (int city_b = city_a + 1; city_b < n; ++city_b) {
                // Calculate rank for the pair (city_a, city_b) and update the maxRank if it's higher
                maxRank = std::max(maxRank, count[city_a] + count[city_b] - graph[city_a][city_b]);
            }
        }

        return maxRank; // Return the maximum network rank found
    }
};
```

Typescript Solution

```
function maximalNetworkRank(n: number, roads: number[][]): number {
    // Create an adjacency matrix for the graph
    const graph: number[][] = Array.from(new Array(n), () => new Array(n).fill(0));
    // Initialize a counter array to keep track of the number of roads each city has
    const roadCount: number[] = new Array(n).fill(0);

    // Build the graph by incrementing the road count for each city and
    // marking the cities as connected in the adjacency matrix
    for (const [city_a, city_b] of roads) {
        graph[city_a][city_b] = 1;
        graph[city_b][city_a] = 1;
        ++roadCount[city_a];
        ++roadCount[city_b];
    }

    // Initialize maximum network rank to 0
    let maxRank = 0;

    // Iterate over each pair of cities and calculate the network rank
    for (let city_a = 0; city_a < n; ++city_a) {
        for (let city_b = city_a + 1; city_b < n; ++city_b) {
            // Network rank is the sum of roads for both cities, subtracting 1 if they have a direct road
            maxRank = Math.max(maxRank, roadCount[city_a] + roadCount[city_b] - graph[city_a][city_b]);
        }
    }

    // Return the maximum network rank found
    return maxRank;
}
```

Time and Space Complexity

The given code snippet calculates the maximal network rank, which is defined as the maximum number of directly connected roads to any two different cities.

Time Complexity

The time complexity of the code can be broken down into two parts:

1. **Building the graph**: The first for loop goes through all the roads, which is given by the size of the input `roads`. If the number of roads is `R`, then this part of the code has a complexity of `O(R)` where `R` is the number of edges (roads) in the graph.
2. **Calculating the maximal network rank**: The second for loop considers each pair of distinct cities exactly once. Since there are `n` cities, and the loop for the second city starts at one more than the current first city, this results in a total of `n * (n - 1) / 2` iterations. Within each iteration, the code performs a constant amount of work; hence, the time complexity of this part is `O(n^2)`.

Therefore, the total time complexity is the sum of both parts, which can be expressed as `O(R + n^2)`.

Space Complexity

The space complexity of the code can be analyzed as follows:

1. **Graph storage**: A `defaultdict(set)` data structure is used to store the graph, which will contain at most `R` edges. Since each edge is stored twice (once for each node it connects), the space required for the graph storage is `O(2R)`, which simplifies to `O(R)`.
2. **Additional variables**: The code uses a constant amount of additional space for variables such as `ans`, `a`, `b`, and `t`.

Therefore, the space complexity of the entire code is `O(R)` considering the graph's representation is the dominant factor.