547. Number of Provinces

Medium Depth-First Search Breadth-First Search Union Find Graph

Problem Description

In this problem, we are given a total of `n` cities and a matrix called `isConnected` which is an `n x n` matrix. The element `isConnected[i][j]` will be `1` if there is a direct connection between city `i` and city `j`, and `0` if there is no direct connection. A set of cities is considered a **province** if all cities within the set are directly or indirectly connected to each other, and no city outside of the set is connected to any city within the set.

Our task is to determine how many provinces there are given the `isConnected` matrix.

Intuition

To find the solution, we conceptualize the cities and the connections between them as a graph, where each city is a node and each direct connection is an edge. Now, the problem translates to finding the number of connected components in the graph. Each connected component will represent one province.

To do this, we use Depth-First Search (DFS). Here's the intuition behind using DFS:

1. We start with the first city and perform a DFS to mark all cities that are connected directly or indirectly to it. These cities form one province.
2. Once the DFS is completed, we look for the next city that hasn't been visited yet and perform a DFS from that city to find another province.
3. We repeat this process until all cities have been visited.

Each time we initiate a DFS from a new unvisited city, we know that we've found a new province, so we increment our province count. The DFS ensures that we navigate through all the cities within a province before moving on to the next one.

By doing the above steps using a `vis` (visited) list to keep track of which cities have been visited, we can effectively determine and count all the provinces.

Solution Approach

The solution uses a Depth-First Search (DFS) algorithm to explore the graph formed by the cities and connections. It utilizes an array `vis` to keep track of visited nodes (cities) to ensure we don't count the same province multiple times. Below is a step-by-step walk-through of the implementation:

1. Define a recursive function `dfs(i: int)` that will perform a depth-first search starting from city `i`.
2. Inside the `dfs` function, mark the current city `i` as visited by setting `vis[i]` to `True`.
3. Iterate over all cities using `j` (which correspond to the columns of `isConnected[i]`).
4. For each city `j`, check if `j` has not been visited (`not vis[j]`) and is directly connected to `i` (`isConnected[i][j] == 1`).
5. If that's the case, call `dfs(j)` to visit all cities connected to `j`, marking the entire connected component as visited.

The solution then follows these steps using the `vis` list:

6. Initialize the `vis` list to be of the same length as the number of cities (`n`), with all elements set to `False`, indicating that no cities have been visited yet.
7. Initialize a counter `ans` to `0`, which will hold the number of provinces found.
8. Iterate through all cities `i` from `0` to `n - 1`.
9. For each city `i`, check if it has not been visited yet (`not vis[i]`).
10. If it hasn't, it means we've encountered a new province. Call `dfs(i)` to mark all cities within this new province as visited.
11. Increment the `ans` counter by `1` as we have found a new province.
12. Continue the loop until all cities have been visited and all provinces have been counted.

At the end of the loop, `ans` will contain the total number of provinces, which the function returns. This completes the solution implementation.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider there are `4` cities, and the `isConnected` matrix is as follows:

```
isConnected = [
    [1, 1, 0, 0],
    [1, 1, 0, 0],
    [0, 0, 1, 1],
    [0, 0, 1, 1]
]
```

Here, cities `0` and `1` are connected, as well as cities `2` and `3`, forming two distinct provinces.

We initialize our `vis` list as `[False, False, False, False]` and set our province counter `ans` to `0`.

Now let's perform the steps of the algorithm:

1. We start with city `0` and run `dfs(0)`.
   - In `dfs(0)`, city `0` is marked visited: `vis = [True, False, False, False]`.
   - We find that city `0` is connected to city `1`, `dfs(1)` is called.
     - In `dfs(1)`, city `1` is marked visited: `vis = [True, True, False, False]`.
     - There are no unvisited cities connected to city `1`, so `dfs(1)` ends.
2. Since all cities connected to city `0` are now visited, `dfs(0)` ends. We've found our first province, so we increment `ans` to `1`.
3. Next, we move to city `1`, but since it's already visited, we proceed to city `2` and run `dfs(2)`.
   - In `dfs(2)`, city `2` is marked visited: `vis = [True, True, True, False]`.
   - We find that city `2` is connected to city `3`, `dfs(3)` is called.
     - In `dfs(3)`, city `3` is marked visited: `vis = [True, True, True, True]`.
     - There are no unvisited cities connected to city `3`, so `dfs(3)` ends.
4. At this point, all cities connected to city `2` are visited, ending `dfs(2)`. We've found another province, incrementing `ans` to `2`.
5. Finally, we move to city `3` and see it's already visited.

Now, we've visited all cities, and there are no unvisited cities to start a new `dfs` from. Thus, we conclude there are `2` provinces in total, which is the value of `ans`.

The full algorithm will perform similarly on a larger scale, incrementing the province count each time it initiates a DFS on an unvisited city, and continuing until all cities are visited. The final result is the total number of provinces.

Python Solution

```python
from typing import List

class Solution:
    def findCircleNum(self, isConnected: List[List[int]]) -> int:
        # Depth-First Search function which marks the nodes as visited
        def dfs(current_city: int):
            visited[current_city] = True  # Mark the current city as visited
            for adjacent_city, connected in enumerate(isConnected[current_city]):
                # If the adjacent city is not visited and there is a connection,
                # then continue the search from that city
                if not visited[adjacent_city] and connected:
                    dfs(adjacent_city)

        # Number of cities in the given matrix
        num_cities = len(isConnected)
        # Initialize a visited list to keep track of cities that have been visited
        visited = [False] * num_cities
        # Counter for the number of provinces (disconnected components)
        province_count = 0
        # Loop over each city and perform DFS if it hasn't been visited
        for city in range(num_cities):
            if not visited[city]:  # If the city hasn't been visited yet
                dfs(city)  # Start DFS from this city
                # After finishing DFS, we have found a new province
                province_count += 1
        # Return the total number of disconnected components (provinces) in the graph
        return province_count

```

Java Solution

```java
class Solution {
    // This variable stores the connection graph.
    private int[][] connectionGraph;
    // This array keeps track of visited cities to avoid repetitive checking.
    private boolean[] visited;

    // The method finds the number of connected components (provinces or circles) in the graph.
    public int findCircleNum(int[][] isConnected) {
        // Initialize the connection graph with the input isConnected matrix.
        connectionGraph = isConnected;
        // The number of cities is determined by the length of the graph.
        int numCities = connectionGraph.length;
        // Initialize the visited array for all cities, defaulted to false.
        visited = new boolean[numCities];
        // Initialize the count of provinces to zero.
        int numProvinces = 0;
        // Iterate over each city.
        for (int i = 0; i < numCities; ++i) {
            // If the city is not yet visited, it's a new province.
            if (!visited[i]) {
                // Perform a depth-first search starting from this city.
                dfs(i);
                // Increment the number of provinces upon returning from DFS.
                ++numProvinces;
            }
        }
        // Return the total number of provinces found.
        return numProvinces;
    }

    // Depth-first search recursive method that checks connectivity.
    private void dfs(int cityIndex) {
        // Mark the current city as visited.
        visited[cityIndex] = true;
        // Iterate over all possible destinations from the current city.
        for (int destination = 0; destination < connectionGraph.length; ++destination) {
            // If the destination city is not yet visited and is connected to the current city,
            // perform a DFS on it.
            if (!visited[destination] && connectionGraph[cityIndex][destination] == 1) {
                dfs(destination);
            }
        }
    }
}

```

C++ Solution

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

class Solution {
public:
    int findCircleNum(std::vector<std::vector<int>>& isConnected) {
        // Get the number of cities (nodes).
        int cities = isConnected.size();

        // Initialize the count of provinces (initially no connection is found).
        int provinceCount = 0;

        // Visited array to keep track of the visited cities.
        bool visited[cities];

        // Initialize all cities as unvisited.
        std::memset(visited, false, sizeof(visited));

        // Define depth-first search (DFS) as a lambda function.
        std::function<void(int)> dfs = [&](int cityIndex) {
            // Mark the current city as visited.
            visited[cityIndex] = true;

            // Visit all the cities connected to the current city.
            for (int j = 0; j < cities; ++j) {
                // If the city is not visited and is connected, perform DFS on it.
                if (!visited[j] && isConnected[cityIndex][j]) {
                    dfs(j);
                }
            }
        };

        // Iterate over each city to count the number of provinces.
        for (int i = 0; i < cities; ++i) {
            // If the city is not yet visited, it is part of a new province.
            if (!visited[i]) {
                dfs(i); // Perform DFS to visit all cities in the current province.
                ++provinceCount; // Increment the count of provinces.
            }
        }

        // Return the total number of provinces found.
        return provinceCount;
    }
};

```

Typescript Solution

```typescript
// Function to find the number of connected components (circles of friends) in the graph
function findCircleNum(isConnected: number[][]): number {
    // Total number of nodes in the graph
    const nodeCount = isConnected.length;
    // Array to track visited nodes during DFS traversal
    const visited: boolean[] = new Array(nodeCount).fill(false);

    // Depth-First Search (DFS) function to traverse the graph
    const depthFirstSearch = (node: number) => {
        // Mark current node as visited
        visited[node] = true;
        for (let adjacentNode = 0; adjacentNode < nodeCount; ++adjacentNode) {
            // For each unvisited adjacent node, perform DFS traversal
            if (!visited[adjacentNode] && isConnected[node][adjacentNode]) {
                depthFirstSearch(adjacentNode);
            }
        }
    };

    // Counter to keep track of the number of connected components (circles)
    let circleCount = 0;

    // Loop through all nodes
    for (let node = 0; node < nodeCount; ++node) {
        // If the node hasn't been visited, it's the start of a new circle
        if (!visited[node]) {
            depthFirstSearch(node); // Perform DFS from this node
            circleCount++; // Increment the number of circles
        }
    }

    // Return the total number of circles found
    return circleCount;
}

```

Time and Space Complexity

The given code snippet represents the Depth-First Search (DFS) approach for finding the number of connected components (which can be referred to as 'circles') in an undirected graph represented by an adjacency matrix `isConnected`.

Time Complexity

The time complexity of the algorithm is `O(N^2)`, where `N` is the number of vertices (or cities) in the graph. This complexity arises because the algorithm involves visiting every vertex once and, for each vertex, iterating through all possible adjacent vertices to explore the edges. In the worst-case scenario, this results in checking every entry in the `isConnected` matrix once, which has `N^2` entries.

The `dfs` function explores all connected vertices through recursive calls. Since each edge and vertex will only be visited once in the DFS traversal, the total number of operations performed will be related to the total number of vertices and edges. However, because the graph is represented by an `N x N` adjacency matrix and it must be fully inspected to discover all connections, the time is bounded by the size of the matrix (`N^2`).

Space Complexity

The space complexity of the algorithm is `O(N)`, which comes from the following:

1. The recursion stack for DFS, which in the worst case, could store up to `N` frames if the graph is implemented as a linked structure such as a list of nodes (a path with all nodes connected end-to-end).
2. The `vis` visited array, which is a boolean array of length `N` used to track whether each vertex has been visited or not to prevent cycles during the DFS.

Given that `N` recursion calls could happen in a singly connected component spanning all the vertices, the space taken by the call stack should be considered in the final space complexity, merging it with the space taken by the array to `O(N)`.