323. Number of Connected Components in an Undirected Graph

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

Problem Description

In this problem, we have a graph that consists of n nodes. We are given an integer n and an array edges where each element edges[i] = [a_i, b_i] represents an undirected edge between nodes a_i and b_i in the graph. The goal is to determine the number of connected components in the graph. A connected component is a set of nodes in a graph that are connected to each other by paths, and those nodes are not connected to any other nodes outside of the component. The task is to return the total count of such connected components.

Intuition

The objective is to count the number of separate groups of interconnected nodes where there are no connections between the groups. We can think of each node as starting in its own group, and each edge as a connection that, potentially, merges two groups into one larger group. Therefore, we are looking to find the total number of distinct sets that cannot be merged any further.

To address this problem, we use the Union-Find algorithm (also known as Disjoint Set Union or DSU). The Union-Find algorithm is efficient in keeping track of elements which are split into one or more disjoint sets. It has two primary operations: find, which determines the identity of the set containing a particular element, and union, which merges two sets together.

1. **Initialization**: We start by initializing each node to be its own parent, representing that each node is the representative of its own set (group).

2. **Union Operation**: We loop through the list of edges. For each edge, we apply the union operation. This means we find the parents (representatives) of the two nodes connected by the edge. If they have different parents, we set the parent of one to be the parent of the other, effectively merging the two sets.

   The function find(x) is a recursive function that finds the root (or parent) of the set to which x belongs. It includes path compression, which means that during the find operation, we make each looked-up node point directly to the root. Path compression improves efficiency, making subsequent find operations faster.

3. **Counting Components**: After processing all the edges, we iterate through the nodes and count how many nodes are their own parent. This count is equal to the number of connected components since nodes that are their own parent represent the root of a connected component.

This solution's beauty is that it is relatively simple but powerful, allowing us to solve the connectivity problem in nearly linear time complexity, which is very efficient.

Solution Approach

The implementation of the solution follows the Union-Find algorithm, using two primary functions: find and union. The key to understanding this approach is recognizing how these functions work together to determine the number of connected components.

Data Structure Used:

- **Parent Array**: An array p is used to keep track of the representative (or parent) for each node in the graph. Initially, each node's representative is itself, meaning p[i] = i.

Algorithms and Patterns Used:

The solution has three main parts:

1. **The Find Function**: The find function is used to find the root (or parent) of the node in the graph. When calling find(x), the function checks if p[x] != x. If this condition is true, it means that x is not its own parent, and there is a path to follow to find the root. This is done recursively until the root is found. The line p[x] = find(p[x]) applies path compression by directly connecting x to the root of its set, which speeds up future find operations.

```
def find(x):
    if p[x] != x:
        p[x] = find(p[x])
    return p[x]
```

2. **Union Operation**: The union operation is not explicitly defined but is performed within the loop that iterates over the edges array. For each edge, the roots of the nodes a and b are found using the find operation. If they have different roots, it means they belong to different sets and should be connected. The line p[find(a)] = find(b) effectively merges the two sets by setting the parent of a's root to be b's root.

```
for a, b in edges:
    p[find(a)] = find(b)
```

3. **Counting Components**: Finally, the number of connected components is determined by counting the number of nodes that are their own parents (i.e., the roots). This is done by iterating through each node i in the range 0 to n-1 and checking if i == find(i). If this condition is true, it means that i is the representative of its component, contributing to the total count.

```
return sum(i == find(i) for i in range(n))
```

The solution approach effectively groups nodes into sets based on the connections (edges) between them, and then it identifies the unique sets to arrive at the total number of connected components. Each connected component is represented by a unique root node, which is why counting these root nodes gives the correct count for the connected components.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Imagine we have n = 5 nodes and the following edges: edges = [[0,1], [1,2], [3,4]].

1. **Initialization**: We start with each node being its own parent, hence p = [0, 1, 2, 3, 4].
2. **Union Operation**: We process the edges one by one:
   - For the edge [0,1], we find the parents of 0 and 1, which are 0 and 1, respectively. Since they are different, we connect them by setting p[find(0)] = find(1) resulting in p = [1, 1, 2, 3, 4].
   - Next, for the edge [1,2], we find the parents of 1 and 2, which are 1 and 2. They are different, so we connect them by setting p[find(1)] = find(2), leading to p = [1, 1, 2, 3, 4].
   - Lastly, for the edge [3,4], we find the parents of 3 and 4, which are 3 and 4. Again, they are different, so we unite them, resulting in p = [1, 1, 1, 4, 4].
3. **Counting Components**: Now we count the number of nodes that are their own parents, which corresponds to the root nodes:
   - 0 is not its own parent (its parent is 1).
   - 1 is the parent of itself, so it counts as a component.
   - 2 is not its own parent (its parent is 1).
   - 3 is not its own parent (its parent is 4).
   - 4 is the parent of itself, so it counts as a component.

In this example, we have two nodes (1 and 4) that are their own parents, so the total number of connected components is 2.

Python Solution

```
class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        # Function to find the root of a node using path compression
        def find(node):
            if parent[node] != node:
                parent[node] = find(parent[node])
            return parent[node]

        # Initialization of parent list where each node is its own parent initially
        parent = list(range(n))

        # Union operation: join two components by pointing the parent of one's root to the other's root
        for a, b in edges:
            parent[find(a)] = find(b)

        # Count the number of components by checking how many nodes are their own parents
        return sum(i == find(i) for i in range(n))
```

Java Solution

```
class Solution {
    private int[] parent; // This array will hold the parent for each node representing the components

    public int countComponents(int n, int[][] edges) {
        // Initialize parent array, where initially each node is its own parent
        parent = new int[n];
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }

        // For each edge, perform a union of the two vertices
        for (int[] edge : edges) {
            int vertex1 = edge[0], vertex2 = edge[1];
            unionElements(vertex1, vertex2);
        }

        // Count the number of components by counting the nodes that are their own parents
        int count = 0;
        for (int i = 0; i < n; ++i) {
            if (i == find(i)) { // If the node's parent is itself, it's the root of a component
                ++count;
            }
        }
        return count; // Return the total count of connected components
    }

    // Find function with path compression
    private int find(int node) {
        if (parent[node] != node) {
            parent[node] = find(parent[node]); // Path compression for efficiency
        }
        return parent[node]; // Return the root parent of the node
    }

    // Union function to join two subsets into a single subset
    private void unionElements(int node1, int node2) {
        int root1 = find(node1); // Find the root parent of the first node
        int root2 = find(node2); // Find the root parent of the second node
        parent[root1] = root2; /* Make one root parent the parent of the other */
    }
}
```

C++ Solution

```
#include <vector>
#include <numeric>
#include <functional>

class Solution {
public:
    // Function to count the number of connected components in an undirected graph
    int countComponents(int n, vector<vector<int>>& edges) {
        // Parent array to represent disjoint sets
        vector<int> parent(n);

        // Initialize each node to be its own parent, forming n separate sets
        iota(parent.begin(), parent.end(), 0);

        // Lambda function to find the representative (leader) of a set
        function<int(int)> find = &{int x} -> int {
            if (parent[x] != x) {
                // Path compression: update the parent of x to be the parent of its current parent
                parent[x] = find(parent[x]);
            }
            return parent[x];
        };

        // Iterate through all edges to merge sets
        for (auto& edge : edges) {
            int node0 = edge[0], node0 = edge[1];
            // Union by rank not used, simply attach the tree of `node0` to `node0`
            parent[find(node0)] = find(node0);
        }

        // Count the number of sets by counting the number of nodes that are self-parented
        int componentCount = 0;
        for (int i = 0; i < n; ++i) {
            if (i == find(i)) { // If the node is the leader of a set
                componentCount++;
            }
        }
        return componentCount;
    }
};
```

Typescript Solution

```
/**
 * Counts the number of connected components in an undirected graph
 * @param {number} n - The number of nodes in the graph
 * @param {number[][]} edges - The edges of the graph
 * @return {number} - The number of connected components
 */
function countComponents(n: number, edges: number[][]): number {
    // Parent array to represent the disjoint set forest
    let parent: number[] = new Array(n);
    // Initialize each node to be its own parent
    for (let i = 0; i < n; ++i) {
        parent[i] = i;
    }

    // Function to find the root of the set that 'x' belongs to
    function find(x: number): number {
        if (parent[x] !== x) {
            // Path compression for efficiency
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    // Union the sets that the edges connect
    for (const [a, b] of edges) {
        // Perform union by setting the parent of the representative of 'a'
        // to the representative of 'b'
        parent[find(a)] = find(b);
    }

    // Count the number of connected components
    // by counting the number of nodes that are their own parent
    let count = 0;
    for (let i = 0; i < n; ++i) {
        if (i === find(i)) {
            count++;
        }
    }
    return count;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily derived from two parts: the union-find operation that involves both path compression in the find function and the union operation in the loop iterating through the edges.

1. **Path Compression (`find` function)** - The recursive find function includes path compression which optimizes the time complexity of finding the root representative of a node to $O(\log n)$ on average and $O(\alpha(n))$ in the amortized case, where α is the inverse Ackermann function, which grows very slowly. For most practical purposes, it can be considered almost constant.

2. **Union Operation (The for loop)** - The time to traverse all edges and perform the union operation for each edge. There are 'e' edges, with e being the length of the edges list. As the path compression makes the union operation effectively near-constant time, this also takes $O(e)$ operations.

3. **Summation Operation** - The final sum iterates over all nodes to count the number of components, taking $O(n)$ time.

Considering all parts, the overall time complexity is $O(e + n)$.

Space Complexity

The space complexity consists of:

1. **Parent Array (p)** - It's a list of length n, so it uses $O(n)$ space.

2. **Recursive Stack Space** - Because the find function is recursive (though with path compression), in the worst case, it could go as deep as $O(n)$ before path compression is applied. However, with path compression, this is drastically reduced.

Given these considerations, the overall space complexity is $O(n)$.