LeetCode 323

https://leetcode.com/problems/number-of-connected-components-in-an-undirected-graph/description/

Yifeng Zeng

Description

323. Number of Connected Components in an Undirected Graph

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 1:

```
     0          3
     |          |
     1 --- 2    4
```

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], return 2.

Example 2:

```
     0           4
     |           |
     1 --- 2 --- 3
```

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3, 4]], return 1.

Note:
You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

Idea Report

We already know there are n nodes, so for each unvisited node, we can do a search to find all the connected nodes and mark as visited, then this is one connected component. And return the total number of different components. Because the input is int[][] edges, so for each node 0 to n - 1, we can create a map of neighbours of the nodes. The map key is the node number itself, the map value is the neighbours that connects to this key node. This is basically a adjacency list which is just for faster access of the node neighbours. And we loop all the unvisited nodes from 0 to n - 1 and do a search to find all their connected neighbours as one connected component and also increase out counter. In the end, after all n nodes are traversed, we can just simply return our counter. We can do both BFS and DFS for the search.

Code:

```
class Solution {
    // BFS AC
    public int countComponents(int n, int[][] edges) {
        Map<Integer, Set<Integer>> map = initMap(edges, n);
        Set<Integer> visited = new HashSet<>();

        int count = 0;
        for (int i = 0; i < n; i++) {
            if (!visited.contains(i)) {
                Deque<Integer> q = new LinkedList<>();
                q.offer(i);
                visited.add(i);
                while (!q.isEmpty()) {
                    int cur = q.poll();
                    for (int nei : map.get(cur)) {
                        if (!visited.contains(nei)) {
                            q.offer(nei);
                            visited.add(nei);
                        }
                    }
                }
                count++;
            }
        }

        return count;
    }

    private Map<Integer, Set<Integer>> initMap(int[][] edges, int n) {
        Map<Integer, Set<Integer>> map = new HashMap<>();
        for (int i = 0; i < n; i++) {
```

```
            map.put(i, new HashSet<>());
        }
        for (int[] edge : edges) {
            map.get(edge[0]).add(edge[1]);
            map.get(edge[1]).add(edge[0]);
        }
        return map;
    }
}
```

```
class Solution {
    // DFS AC
    public int countComponents(int n, int[][] edges) {
        Map<Integer, Set<Integer>> map = initMap(edges, n);
        Set<Integer> visited = new HashSet<>();

        int count = 0;
        for (int i = 0; i < n; i++) {
            if (!visited.contains(i)) {
                dfsHelper(map, visited, i);
                count++;
            }
        }

        return count;
    }

    private void dfsHelper(Map<Integer, Set<Integer>> map,
                           Set<Integer> visited, int cur) {
        for (int nei : map.get(cur)) {
            if (!visited.contains(nei)) {
                visited.add(nei);
                dfsHelper(map, visited, nei);
            }
        }
    }

    private Map<Integer, Set<Integer>> initMap(int[][] edges, int n) {
        Map<Integer, Set<Integer>> map = new HashMap<>();
        for (int i = 0; i < n; i++) {
            map.put(i, new HashSet<>());
        }
        for (int[] edge : edges) {
            map.get(edge[0]).add(edge[1]);
            map.get(edge[1]).add(edge[0]);
        }
        return map;
    }
}
```

We can also use another advanced data structure which is called union find data structure. Basically we have an array of n (int[] parents), represeting the parent node of each node 0 to n - 1. And we update this parents array while based on nodes connection. Initially each node is itself's parent. And we loop over the edges. For each edge, there are two nodes, a and b. We find their parents pa and pb. If pa == pb, it means they are already have same parent, meaning they belongs to the same connected component. If pa != pb, it means they are not connected yet, since the current edge connects node a and b, we just need to point one parent to another to perform the connection. After looping all the edges, we count how many parents are there, that means how many connected components are there.

```
class Solution {
    // union find AC
    public int countComponents(int n, int[][] edges) {
        int[] parents = new int[n];
        for (int i = 0; i < n; i++) {
            parents[i] = i;
        }

        for (int[] edge : edges) {
            int pa = find(parents, edge[0]);
            int pb = find(parents, edge[1]);
            if (pa != pb) {
                parents[pa] = pb;
            }
        }

        int count = 0;
        for (int i = 0; i < n; i++) {
            if (parents[i] == i) {
                count++;
            }
        }
        return count;
    }

    private int find(int[] parents, int i) {
        while (parents[i] != i) {
            parents[i] = parents[parents[i]];
            i = parents[i];
        }
        return i;
    }
}
```

Summary

- Create adjacency list of the graph for fatser neighbour access.
- Union find.
- Similar questions like 305. Number of Islands II, 261. Graph Valid Tree, 547. Friend Circles. may also use union find.