入选理由

- 并查集的一大重要的应用就是 **** （此处略去七个字），而这道题就是****（此处略去七个字）。

标签

- 并查集

难度

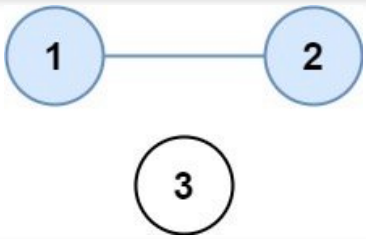
- 中等

题目地址(547. 省份数量)

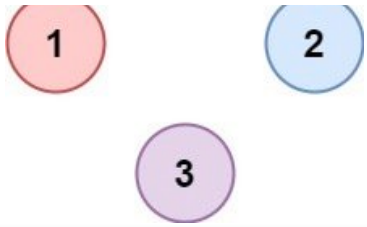
https://leetcode-cn.com/problems/number-of-provinces/

题目描述

```
有 n 个城市，其中一些彼此相连，另一些没有相连。如果城市 a 与城市 b 直接相连，且城市 b 与城市 c 直接相连，那么城市 a 与城市 c 间接相连。

省份 是一组直接或间接相连的城市，组内不含其他没有相连的城市。

给你一个 n x n 的矩阵 isConnected ，其中 isConnected[i][j] = 1 表示第 i 个城市和第 j 个城市直接相连，而 isConnected[i][j] = 0 表示

返回矩阵中 省份 的数量。

示例 1:
```




```
输入: isConnected = [[1,1,0],[1,1,0],[0,0,1]]
输出: 2
示例 2:
```




```
输入：isConnected = [[1,0,0],[0,1,0],[0,0,1]]
输出：3


提示：

1 <= n <= 200
n == isConnected.length
n == isConnected[i].length
isConnected[i][j] 为 1 或 0
isConnected[i][i] == 1
isConnected[i][j] == isConnected[j][i]
```

DFS

思路

每个学生看作图中的一个节点，那么问题就转化为求图的强连通分量。这里提供三种解法，这三种方法在求解联通分量个数的时候都可以使用。

首先要讲的是 DFS，这是一种相对直接容易想到且代码较为通俗易懂的解法。

算法流程：

- 选定一个节点，开始深度优先搜索，将遍历到的节点标记为 visited，直到无法继续遍历，连通图数目加一
- 选取下一个**未遍历**的节点，重复上述过程，直到所有节点都被遍历

代码

代码支持：JS,Java，Python

```
/*
 * @lc app=leetcode.cn id=547 lang=javascript
 *
 * [547] 朋友圈
 */

// @lc code=start
/**
```

```
 * DFS
 * @param {number[][]} M
 * @return {number}
 */
var findCircleNum = function (M) {
  const visited = Array.from({ length: M.length }).fill(0);
  let res = 0;
  for (let i = 0; i < visited.length; i++) {
    if (!visited[i]) {
      visited[i] = 1;
      dfs(i);
      res++;
    }
  }
  return res;

  function dfs(i) {
    for (let j = 0; j < M.length; j++) {
      if (i !== j && !visited[j] && M[i][j]) {
        visited[j] = 1;
        dfs(j);
      }
    }
  }
};
```

Java Code

```
class Solution {
    public int findCircleNum(int[][] isConnected) {
        //城市数量
        int n = isConnected.length;
        //表示哪些城市被访问过
        boolean[] visited = new boolean[n];
        int count = 0;
        //遍历所有的城市
        for(int i = 0; i < n; i++){
        //如果当前城市没有被访问过，说明是一个新的省份，
        //count要加1，并且和这个城市相连的都标记为已访问过，
        //也就是同一省份的
            if(!visited[i]){
                dfs(isConnected, visited, i);
                count++;
            }
        }
        return count;
    }

    public void dfs(int[][] isConnected, boolean[] visited, int i){
```

```
        for(int j = 0; j < isConnected.length; j++){
            if(isConnected[i][j] == 1 && !visited[j]){
                //如果第i和第j个城市相连，说明他们是同一个省份的，把它标记为已访问过
                visited[j] = true;
                //然后继续查找和第j个城市相连的城市
                dfs(isConnected, visited, j);
            }
        }
    }
}
```

Python Code：

```
class Solution:
    def findCircleNum(self, isConnected: List[List[int]]) -> int:

        def dfs(i):
            visited.add(i)
            for j in range(len(isConnected[i])):
                if j not in visited and isConnected[i][j]==1:
                    dfs(j)

        visited = set()
        provinces = 0
        for i in range(len(isConnected)):
            if i not in visited:
                dfs(i)
                provinces +=1

        return provinces
```

复杂度分析 令 n 为矩阵 M 的大小。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

BFS

思路

和上面思路类似。不过搜索的方向不再是深度优先，而是广度优先。由于我们并不会提前终止，因此使用 bfs 算法效率并没有提升。

代码

代码支持：JS

```
var findCircleNum = function (M) {
  const visited = Array(M.length).fill(0);
  let res = 0;
  const queue = [];
  for (let i = 0; i < M.length; i++) {
    if (!visited[i]) {
      visited[i] = 1;
      res++;
      queue.push(i);
    }

    while (queue.length) {
      const cur = queue.shift();
      for (let j = 0; j < M.length; j++) {
        if (cur !== j && M[cur][j] && !visited[j]) {
          queue.push(j);
          visited[j] = 1;
        }
      }
    }
  }
  return res;
};
```

令 n 为矩阵 M 的大小。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

并查集

思路

使用并查集求联通分量个数再合适不过了。我们只需要在合并的过程中记录一下联通分量的个数即可。

具体来说，我们初始化 n 个联通分量，如果两个人是朋友，那么将其合并。如果合并之前他们已经在一个朋友圈了，那么联通分量个数不会变化，否则联通分量个数会减 1。

算法：

- 初始时，强连通分量为 count = M.length
- MAKE-SET, 将每个节点的 parent 指向其本身
- FIND, 并查集常规搜索，添加路径压缩

- UNION(x, y)
  - 如果(x, y)属于同一个子集，返回
  - 如果(x, y)属于不同子集，将两个子集合并，count--
- 最终返回 count

代码

代码支持： JS，Python,Java

JS Code：

```
var findCircleNum = function (M) {
  let count = M.length;
  let parents = Array.from(M).map((item, index) => index);
  function find(x) {
    if (parents[x] === x) {
      return x;
    }
    return (parents[x] = find(parents[x]));
  }

  function union(x, y) {
    if (find(x) === find(y)) {
      return;
    }
    parents[parents[x]] = parents[y];
    // 两个集合合并，集合数 -1
    count--;
  }

  for (let i = 0; i < M.length; i++) {
    for (let j = i + 1; j < M[i].length; j++) {
      if (M[i][j]) {
        // 如果两个人有边，尝试合并
        union(i, j);
      }
    }
  }

  return count;
};
```

Python Code：

```python
class UF:
    def __init__(self, n) -> None:
        self.parent = {i: i for i in range(n)}
        self.size = n

    def find(self, i):
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def connect(self, i, j):
        root_i, root_j = self.find(i), self.find(j)
        if root_i != root_j:
            self.size -= 1
            self.parent[root_i] = root_j


class Solution:
    def findCircleNum(self, isConnected: List[List[int]]) -> int:
        n = len(isConnected)
        uf = UF(n)
        for i in range(n):
            for j in range(n):

                if isConnected[i][j]:
                    uf.connect(i, j)
        return uf.size
```

Java Code:

```java
class Solution {
    public int findCircleNum(int[][] isConnected) {
        int n = isConnected.length;
        UF uf = new UF(n);
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (isConnected[i][j] == 1) {
                    uf.union(i, j);
                }
            }
        }
        return uf.size;
    }
    private static class UF {
        int[] parent;
        int size;
        public UF(int n) {
            parent = new int[n];
            size = n;
```

```
            for (int i = 0; i < n; i++) {
                parent[i] = i;
            }
        }
        public int find(int x) {
            while (parent[x] != x) {
                x = parent[x];
            }
            return x;
        }
        public void union(int x, int y) {
            if (find(x) != find(y)) {
                parent[find(x)] = find(y);
                size--;
            }
        }
    }
}
```

复杂度分析

令 n 为矩阵 M 的大小。

- 时间复杂度：由于仅使用了一种优化（路径压缩），因此时间复杂度为 $O(nlogn)$。
- 空间复杂度：$O(n)$。

上一页 下一页