**【第六周】684. Redundant Connection**

2017年10月16日 00:33:34

阅读数：226

**原题：**

In this problem, a tree is an undirected graph that is connected and has no cycles.

The given input is a graph that started as a tree with N nodes (with distinct values 1, 2, ..., N), with one additional edge added. The added edge has two different vertices chosen from 1 to N, and was not an edge that already existed.

The resulting graph is given as a 2D-array of edges. Each element of edges is a pair [u, v] with u < v, that represents an undirected edge connecting nodes u and v.

Return an edge that can be removed so that the resulting graph is a tree of N nodes. If there are multiple answers, return the answer that occurs last in the given 2D-array. The answer edge [u, v] should be in the same format, with u < v.

Example 1:

Input: [[1,2], [1,3], [2,3]]

Output: [2,3]

Explanation: The given undirected graph will be like this:

1

/ \

2 - 3

Example 2:

Input: [[1,2], [2,3], [3,4], [1,4], [1,5]]

Output: [1,4]

Explanation: The given undirected graph will be like this:

5 - 1 - 2

| |

4 - 3

Note:

The size of the input 2D-array will be between 3 and 1000.

Every integer represented in the 2D-array will be between 1 and N, where N is the size of the input array.

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28

**leetcode地址：[https://leetcode.com/problems/redundant-connection/description/](https://leetcode.com/problems/redundant-connection/description/" \t "_blank)**

**解题思路**

**题目大意是：给出一个无向无环图的所有顶点和边，以及一条多余的边，这条边会使图成环；找出这条边。**

 求解的思路大概是这样：   
 一开始，所有的点是各不连通的；然后每添加一条边，便连通两个顶点；当要添加某条边时，如果发现此边的两个顶点已经是连通状态，则此边就是多余的边。

 那么，如何判断两个顶点是否连通呢？同时，图又要用什么数据结构来保存信息呢？

 起初我采用的是map<int, set<int> > 这样的数据结构来保存信息：顶点编号当做key，将同在一个连通子图中的其它顶点放在一个set中，将此set当做value；检查连通性时，只要比较两个set是否相同；如果不同，则将两个set合并为一个，并更新成新的value，用以连通这两个图；如果set相同，则表示已连通，此边多余。

 但是以上的想法实现时遇到了问题：添加一条边时，只会将这两个顶点对应的set更新；而同在一个连通子图中的其他顶点的set没有跟随改变。也就是说，这两个顶点更新set时，没有”通知“到其他连通的顶点。 尝试了把map<int, set<int> >变成map<int, set<int>\* >，即所有连通的顶点映射到同一个set，但运行时发生问题了，最后一直也没解决。

 然后网上寻求思路，才发现这类型的问题属于**动态连通性问题**，一般使用的是一种叫做**并查集**的数据结构，称为Union-Find。其实上面的思路已经类似并查集了，只是在数据结构的设计上出现了问题。然后了解了并查集算法，重新完成代码，最终AC。

 简单的并查集算法思路如下：   
 首先每个节点都是各自独立的，故每个节点的根节点都是其本身。连通两个节点A、B时，检查A与B的根节点；如果根节点相同，则表示A、B已经是连通的；如果根节点不同，则将A的根节点修改为B的根节点。对于并查集算法而言，关键要掌握的点在于findRoot()方法，即寻找一个节点的根节点的方法。这个方法要同时具备”查找”与”修改”的功能。具体代码见下。

**代码**

class Solution {

public:

vector<int> findRedundantConnection(vector<vector<int>>& edges) {

int N = edges.size();

int parent[1001] = {};

for (int i = 0; i < 1001; i++) { //初始化所有节点的根为自身；

parent[i] = i;

}

for (int i = 0; i < N; i++) {

vector<int> cur = edges[i];

int a = cur[0], b = cur[1];

if (findRoot(parent, a) == findRoot(parent, b)) return cur; //根节点相同则返回此边

parent[findRoot(parent, b)] = findRoot(parent, a); //根节点不同，则修改根节点，连通两个子图

}

}

int findRoot(int\* parent, int n) { //寻根方法

if (parent[n] == n) return n; //如果某节点的根就是它本身，它则是根节点

parent[n] = findRoot(parent, parent[n]); //如果不是，则向上追溯到根节点并修正成新的根节点

return parent[n]; // 返回修正后的根节点

}

};

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

**总结**

1、并查集的概念   
2、动态连通性问题