Redundant Connection II

In this problem, a rooted tree is a **directed** graph such that, there is exactly one node (the root) for which all other nodes are descendants of this node, plus every node has exactly one parent, except for the root node which has no parents.

The given input is a directed graph that started as a rooted tree with N nodes (with distinct values 1, 2, ..., N), with one additional directed 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]` that represents a **directed** edge connecting nodes `u` and `v`, where `u` is a parent of child `v`.

Return an edge that can be removed so that the resulting graph is a rooted tree of N nodes. If there are multiple answers, return the answer that occurs last in the given 2D-array.

Example 1:

```
Input: [[1,2], [1,3], [2,3]]
Output: [2,3]
Explanation: The given directed graph will be like this:
  1
 / \
v   v
2-->3
```

Example 2:

```
Input: [[1,2], [2,3], [3,4], [4,1], [1,5]]
Output: [4,1]
Explanation: The given directed graph will be like this:
5  2
     ^    |
     |    v
     4
```

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.

Solution 1

We can not simply apply the code from REDUNDANT CONNECTION I, and add codes checking duplicated-parent only.
Here are 2 sample failed test cases:
`[[4,2],[1,5],[5,2],[5,3],[2,4]]`
got `[5,2]` but `[4,2]` expected
and
`[[2,1],[3,1],[4,2],[1,4]]`
got `[3,1]` but `[2,1]` expected
(Thanks @niwota and @wzypangpang)
The problem is we can not consider the two conditions separately, I mean the duplicated-parents and cycle.
This problem should be discussed and solved by checking 3 different situations:

1. No-Cycle, but 2 parents pointed to one same child
2. No dup parents but with Cycle
3. Possessing Cycle and dup-parents

Those 2 failed test cases are all in situation 3), where we can not return immediately current edge when we found something against the tree's requirements.
The correct solution is detecting and recording the whole cycle, then check edges in that cycle by reverse order to find the one with the same child as the duplicated one we found.
A more clear explanation and code have been posted by @niwota
https://discuss.leetcode.com/topic/105087/share-my-solution-c

written by JadenPan original link here

Solution 2

This problem is limited to a graph with N nodes and N edges. No node is singled out if a edge is removed. For example, [[1,2],[2,4],[3,4]], 4 nodes 3 edges, is not applicable to this problem. You cannot remove [3,4] to single out node 3.

There are 3 cases:

1. No loop, but there is one node who has 2 parents.
2. A loop, and there is one node who has 2 parents, that node must be inside the loop.
3. A loop, and every node has only 1 parent.

Case 1: e.g. `[[1,2],[1,3],[2,3]]` ,node 3 has 2 parents ([1,3] and [2,3]). Return the edge that occurs last that is, return [2,3].
Case 2: e.g. `[[1,2],[2,3],[3,1],[4,1]]` , {1->2->3->1} is a loop, node 1 has 2 parents ([4,1] and [3,1]). Return the edge that is inside the loop, that is, return [3,1].
Case 3: e.g. `[[1,2],[2,3],[3,1],[1,4]]` , {1->2->3->1} is a loop, you can remove any edge in a loop, the graph is still valid. Thus, return the one that occurs last, that is, return [3,1].

```
class Solution {
private:
    struct Node{
        vector<int> from; //parent(s), at most one node has 2 parents in a graph
        vector<int> to; //children, can have many children
    };
    unordered_map<int,Node> getNode;
    unordered_map<int,unordered_map<int,int>> edgeOrder;

public:
    vector<int> findRedundantDirectedConnection(vector<vector<int>>& edges) {
        //construct the graph, and record the node which has 2 parents if possible
        int N = edges.size();
        for(int n=1; n<=N; ++n)
            getNode[n] = Node();
        int node2parents = -1;
        for(int i=0; i<N; ++i){
            int p = edges[i][0];
            int c = edges[i][1];
            edgeOrder[p][c] = i;
            getNode[p].to.push_back(c);
            getNode[c].from.push_back(p);
            if(getNode[c].from.size()==2) //we find a node with 2 parents
                node2parents = c;
        }

        //doing DFS to find the loop if loop exists
        vector<int> status(N+1,0); // status 0,1,2 ==> 0:unvisited, 1:visiting, 2:v
isited
        stack<int> loop;
        bool loopfound = false;
        for(int i=1; i<=N; ++i){
            if(loopfound)   break;
```

```
            if(status[i] == 0){ //DFS started with node i
                status[i] = 1;
                stack<int> stk({i});
                DFS(stk,status,loopfound,loop);
                status[i] = 2;
            }
        }

        if(!loopfound){ // Case 1
            int parent1 = getNode[node2parents].from[0];
            int parent2 = getNode[node2parents].from[1];
            return (edgeOrder[parent1][node2parents] > edgeOrder[parent2][node2paren
ts]) ?
                    vector<int>({parent1,node2parents}) : vector<int>({parent2,node
2parents});
        }

        int last_occur_order = 0;
        vector<int> last_occur_edge;
        int begin = loop.top();
        while(!loop.empty()){
            int child = loop.top();
            loop.pop();
            int parent = loop.top();
            if(node2parents != -1 && child == node2parents) // Case 2
                return vector<int>({parent,child});
            int order = edgeOrder[parent][child];
            if(order > last_occur_order){
                last_occur_order = order;
                last_occur_edge = vector<int>({parent,child});
            }
            if(parent == begin)
                break; //loop ends
        }

        return last_occur_edge; // Case 3
    }

    void DFS(stack<int>& stk, vector<int>& status, bool& flag, stack<int>& loop){
        for(int c : getNode[stk.top()].to){
            if(flag)   return;
            if(status[c] == 1){
                stk.push(c);
                loop = stk;
                flag = true;
                return;
            }
            else if(status[c] == 0){
                stk.push(c);
                status[c] = 1;
                DFS(stk,status,flag,loop);
                status[c] = 2;
                stk.pop();
            }
        }
    }
};
```

written by niwota original link here

Solution 3

This problem is very similar to "Redundant Connection". But the description on the parent/child relationships is much better clarified.

```
There are two cases for the tree structure to be invalid.
1) A node having two parents;
   including corner case: e.g. [[4,2],[1,5],[5,2],[5,3],[2,4]]
2) A circle exists
```

If we can remove exactly 1 edge to achieve the tree structure, a single node can have at most two parents. So my solution works in two steps.

```
1) Check whether there is a node having two parents.
    If so, store them as candidates A and B, and set the second edge invalid.
2) Perform normal union find.
    If the tree is now valid
           simply return candidate B
    else if candidates not existing
           we find a circle, return current edge;
    else
           remove candidate A instead of B.
```

If you like this solution, please help upvote so more people can see.

```cpp
class Solution {
public:
    vector<int> findRedundantDirectedConnection(vector<vector<int>>& edges) {
        int n = edges.size();
        vector<int> parent(n+1, 0), candA, candB;
        // step 1, check whether there is a node with two parents
        for (auto &edge:edges) {
            if (parent[edge[1]] == 0)
                parent[edge[1]] = edge[0];
            else {
                candA = {parent[edge[1]], edge[1]};
                candB = edge;
                edge[1] = 0;
            }
        }
        // step 2, union find
        for (int i = 1; i <= n; i++) parent[i] = i;
        for (auto &edge:edges) {
            if (edge[1] == 0) continue;
            int u = edge[0], v = edge[1], pu = root(parent, u);
            // Now every node only has 1 parent, so root of v is implicitly v
            if (pu == v) {
                if (candA.empty()) return edge;
                return candA;
            }
            parent[v] = pu;
        }
        return candB;
    }
private:
    int root(vector<int>& parent, int k) {
        if (parent[k] != k)
            parent[k] = root(parent, parent[k]);
        return parent[k];
    }
};
```

Java version by MichaelLeo

```java
class Solution {
    public int[] findRedundantDirectedConnection(int[][] edges) {
        int[] can1 = {-1, -1};
        int[] can2 = {-1, -1};
        int[] parent = new int[edges.length + 1];
        for (int i = 0; i < edges.length; i++) {
            if (parent[edges[i][1]] == 0) {
                parent[edges[i][1]] = edges[i][0];
            } else {
                can2 = new int[] {edges[i][0], edges[i][1]};
                can1 = new int[] {parent[edges[i][1]], edges[i][1]};
                edges[i][1] = 0;
            }
        }
        for (int i = 0; i < edges.length; i++) {
            parent[i] = i;
        }
        for (int i = 0; i < edges.length; i++) {
            if (edges[i][1] == 0) {
                continue;
            }
            int child = edges[i][1], father = edges[i][0];
            if (root(parent, father) == child) {
                if (can1[0] == -1) {
                    return edges[i];
                }
                return can1;
            }
            parent[child] = father;
        }
        return can2;
    }

    int root(int[] parent, int i) {
        while (i != parent[i]) {
            parent[i] = parent[parent[i]];
            i = parent[i];
        }
        return i;
    }
}
```

written by zestypanda original link here