684. Redundant Connection

Problem Description

In this problem, we're given a graph that was a tree initially, but an additional edge was added to it. An edge in this context is represented by a pair of nodes that it connects, given as `[a, b]`. Since a proper tree has no cycles and is connected, the additional edge creates a single cycle in the graph. Our task is to identify and remove this redundant edge, restoring the graph to a tree. The graph is represented as an array `edges` of `n` edges, where each edge connects two different nodes from `1` to `n`, and `n` is the number of nodes in the tree. The goal is to find the redundant edge that can be removed to make the graph a tree again. If there are multiple redundant edges, we need to return the one that appears last in the `edges` array. This problem checks the understanding of graph properties, specifically tree structures, and graph algorithms.

Intuition

The intuition behind this solution is based on the Union-Find algorithm, which is a classic data structure for keeping track of disjoint sets. The idea is to iterate through the list of edges and connect the nodes using the Union operation. We start by initializing a parent array `p`, representing each node's parent in the disjoint set. Initially, each node is its own parent, indicating that they are each in separate sets.

As we process the edges, we use a Find operation, implemented as the `find` function, to determine the set representative (parent) for each node. If the nodes connected by the current edge already have the same parent, it means that adding this edge would form a cycle, which is contrary to the tree definition. When we identify such an edge, we know it's redundant and must be removed.

The clever part of the algorithm is the use of path compression in the Find operation. This optimization steps speeds up subsequent queries for the set representative by making each node along the path point to the ultimate parent directly.

Therefore, when we come across an edge connecting two nodes that are already in the same set (have the same parent), this edge is the redundant one, and we should return it as the answer. In the case that we process all edges without returning any, though highly unlikely for this problem's constraints, we return an empty list. However, since the problem states there's always one additional edge, we're guaranteed to find a redundant connection during our iteration through the edge list.

Solution Approach

The solution approach involves the following steps, which are implemented with the Union-Find algorithm:

1. **Initialization**: We initialize an array `p` with `1010` elements, which is assumed to be more than enough to cover the node labels up to `n`. Here, each element is initialized to its index value, i.e., `p[i] = i`. This means each node starts as the parent of itself, representing a unique set for each node.

2. **Union-Find Algorithm**:
   - The key operations in the Union-Find algorithm are `find` and `union`.
   - **Find Operation**: This operation is implemented as a recursive function:

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

     The `find` function aims to locate the representative or the "parent" of the set that a node belongs to. If the node's parent is not itself, it proceeds to find the ultimate parent recursively and applies path compression by setting `p[x]` to the ultimate parent. This optimizes the data structure so that subsequent `find` operations are faster.
   - **Union Operation**: This operation is implicitly handled within the for loop:

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

     For each edge `(a, b)` in `edges`, we first find the parent of both `a` and `b`. If they have the same parent, we found our redundant edge which is returned immediately because adding it to the tree would create a cycle. Otherwise, we perform the union by setting the parent of node `a`'s set to be the parent of node `b`'s set, effectively merging the two sets.

3. **Returning the Result**:
   - The loop iterates over each edge only once and the redundant edge will be found during this iteration because we're guaranteed there is exactly one extra edge that creates a cycle.
   - The function `return [a, b]` statement is triggered when a redundant connection is found, which is the edge that causes the cycle.
   - If the loop finishes without finding a redundant edge (which should not happen given the problem constraints), the function defaults to returning an empty list by `return []`.

In summary, the solution leverages the efficient Union-Find data structure with path compression to identify a cycle in the graph, which is the signature of the redundant connection in an otherwise acyclic tree structure.

Example Walkthrough

Let's use a small example graph to illustrate the solution approach as described. Suppose our graph, which was originally a tree, has 4 nodes, and the `edges` array is as follows: `[[1, 2], [2, 3], [3, 1], [4, 3]]`. We can already see that there is a cycle formed by the edges `[1, 2]`, `[2, 3]`, and `[3, 1]`. According to the problem statement, we need to find the redundant edge that appears last in the `edges` list, which upon a preliminary look seems to be `[3, 1]`.

1. **Initialization**:

   We initialize our parent array `p` with enough space for our 4 nodes (in reality, our solution sets this up for more nodes, but for simplicity, here we'll limit it to the nodes we have). Thus, `p = [0, 1, 2, 3, 4]` (we include `0` just to match the node indices with their parent indices for convenience).

2. **Union-Find Algorithm**:

   We iterate through the `edges` array using the `find` and `union` operations as described:

   - For the first edge `[1, 2]`, we find the parents of `1` and `2`, which are themselves, so we unite them by setting `p[find(1)] = find(2)`. Now `p` becomes `[0, 2, 2, 3, 4]`.
   - For the second edge `[2, 3]`, we find the parents of `2` and `3`, which are `2` and `3` respectively. Since they are different, we unite them by setting `p[find(2)] = find(3)`. Now `p` becomes `[0, 3, 2, 3, 4]` and after path compression `[0, 3, 3, 3, 4]`.
   - For the third edge `[3, 1]`, we find the parents of `3` and `1`. Both are `1` due to the path compression that has already occurred, indicating we've encountered a cycle. As this is the redundant edge we're looking for, we immediately return `[3, 1]`.
   - The fourth edge `[4, 3]` does not get processed as we've already found and returned the redundant edge in the step above.

3. **Returning the Result**:
   - We successfully find that `[3, 1]` is the redundant edge that creates a cycle in the graph.
   - The function returns `[3, 1]` as the answer, which is the edge that must be removed to eliminate the cycle and restore the graph to a proper tree.

In this example, we can see that the Union-Find algorithm effectively identifies the edge causing a cycle due to the immediate parent check in the `find` operation, assisted by the path compression that optimizes this check as we progress through the array of edges.

Python Solution

```python
from typing import List

class Solution:
    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        # Function to find the set leader using path compression.
        def find_set_leader(vertex):
            # If the parent of the current vertex is not itself,
            # recursively find the set leader and apply path compression.
            if parent[vertex] != vertex:
                parent[vertex] = find_set_leader(parent[vertex])
            return parent[vertex]

        # Initialize the parent array, with each vertex as its own set leader at the start.
        parent = list(range(len(edges) + 1))

        # Iterate through all the edges and apply union-find.
        for a, b in edges:
            # Find the set leaders for both vertices of the edge.
            leader_a = find_set_leader(a)
            leader_b = find_set_leader(b)

            # If both vertices have the same set leader, a cycle is found, return the edge.
            if leader_a == leader_b:
                return [a, b]

            # Union step: assign vertex b's leader to be the leader of vertex a's set.
            parent[leader_a] = leader_b

        # If no redundant connection found (no cycle), return an empty list.
        return []

# Example usage:
# sol = Solution()
# print(sol.findRedundantConnection([[1,2], [1,3], [2,3]]))  # Output: [2, 3]
```

Java Solution

```java
class Solution {

    // Array representing the parents of each node in the disjoint-set (union-find).
    private int[] parent;

    // Function to find the redundant connection, taking edges of a graph.
    public int[] findRedundantConnection(int[][] edges) {

        // Initialize the parent array for a maximum of 1010 nodes.
        parent = new int[1010];
        for (int i = 0; i < parent.length; i++) {
            // Initially, each node is its own parent (self loop).
            parent[i] = i;
        }

        // Iterate through all edges.
        for (int[] edge : edges) {
            int node1 = edge[0];
            int node2 = edge[1];

            // If both nodes have the same parent, the edge is redundant.
            if (find(node1) == find(node2)) {
                // Return edge.
                return edge;
            }

            // Union the sets of two nodes by setting the parent of one as the other.
            parent[find(node1)] = find(node2);
        }

        // If no redundant connection is found, return null (should not happen according to the problem statement).
        return null;
    }

    // Recursive function to find the parent of a given node.
    private int find(int node) {

        // Path compression: if the node is not its own parent, recursively find its parent and update the reference.
        if (parent[node] != node) {
            parent[node] = find(parent[node]);
        }
        // Return the parent of the node.
        return parent[node];
    }
}
```

C++ Solution

```cpp
#include <vector>
using std::vector;

class Solution {
public:
    // Parent array for Union-Find structure.
    vector<int> parent;

    // Function to find redundant connection in a graph represented as an edge list.
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {
        // Initialize the parent array for Union-Find, with self-pointing indices.
        parent.resize(1010);
        for (int i = 0; i < parent.size(); ++i) parent[i] = i;

        // Iterate over the edges.
        for (auto& edge : edges) {
            int node1 = edge[0], node2 = edge[1];
            // If both nodes already have the same parent, they have the same parent),
            // they create a cycle (i.e., they have the same parent),
            if (find(node1) == find(node2)) return edge; // If yes, then current edge is redundant.
            // Union the two sets.
            parent[find(node1)] = find(node2);
        }
        // If no redundant edge is found (which shouldn't happen), return an empty array.
        return {};
    }

    // Helper function to find the root parent of a node, with path compression.
    int find(int node) {
        if (parent[node] != node) { // Check if node is not its own parent (not root node).
            parent[node] = find(parent[node]); // Recursively find and set the root node.
        }
        return parent[node]; // Return the root node.
    }
};
```

Typescript Solution

```typescript
/**
 * Function to find the redundant connection in a graph.
 * Returns the edge that, when removed, would convert a graph into a tree (a graph with no cycles).
 * The graph is represented as a list of edges.
 */
function findRedundantConnection(edges: number[][]): number[] {
    // parent array to keep track of the root parent of each node in the graph.
    const parents: number[] = Array.from({ length: 1010 }, (_, index) => index);

    /**
     * Helper function to find the root parent of a node.
     * Implements path compression to optimize the union-find structure.
     *
     * @param {number} node - The node to find the root parent for.
     * @return {number} - The root parent of the node.
     */
    function find(node: number): number {
        if (parents[node] !== node) {
            parents[node] = find(parents[node]);
        }
        return parents[node];
    }

    // Iterate through each edge to find the redundant connection.
    for (let [node0, node1] of edges) {
        // If the two nodes have the same root parent, a cycle is detected.
        if (find(node0) === find(node1)) {
            // Return the redundant edge.
            return [node0, node1];
        }
        // Union the sets.
        parents[find(node0)] = find(node1);
    }

    // If no redundant connection is found, return an empty array.
    return [];
};

export { findRedundantConnection }; // Optional: Use this if you want to import the function elsewhere.
```

Time and Space Complexity

The given code implements a Union-Find algorithm to detect a redundant connection in a graph. Let's analyze the time and space complexity:

Time Complexity

The time complexity of the Union-Find algorithm primarily depends on the two operations: `find` and `union`. Each edge triggers both operations once.

- The `find` operation uses path compression, which gives us an amortized time complexity of almost constant time, "$O(\alpha(n))$", where α is the Inverse Ackermann function and is very slowly growing, therefore it's nearly constant for all practical purposes.
- The `union` takes "$O(1)$" time since it's just assigning the root of one element to another.

Since the operations are performed for each edge and there are "n" edges, the amortized time complexity of the algorithm is "$O(n\alpha(n))$", which simplifies to "$O(E)$" considering the very slow growth of $\alpha(n)$.

Space Complexity

For space complexity, we have:

- An array `p`, which stores the parents of each node in the graph. The size of the array is fixed at 1010, therefore, it requires "$O(1)$" space since it's a constant space and doesn't depend on the size of the input.

Thus, the overall space complexity of the given code is "$O(1)$".