990. Satisfiability of Equality Equations

Medium Union Find Graph Array String

Problem Description

In the LeetCode problem presented, we are given a set of equations that express relationships between variables. Each variable is represented by a single lowercase letter, and each equation is a string of 4 characters. The equations come in two forms:

1. `"x!=y"` which indicates that the two variables represented by `x` and `y` must be equal.
2. `"x==y"` which specifies that the two variables must not be equal, where `x` and `y` are any lowercase letters.

We need to determine if there is a way to assign integers to each variable such that all the equality and inequality equations are satisfied simultaneously. If there is a way to do so, we should return `true`, else return `false`.

Intuition

To solve this problem, we can use the **union-find** algorithm, which is a data structure that is very efficient at grouping elements into disjoint sets and determining whether two elements are in the same set.

Here is the intuition broken into steps:

1. **Initialization**: We first initialize 26 elements, one for each lowercase letter, to represent each variable. Each element is its own parent in the union-find data structure in the beginning.
2. **Processing Equalities**: We iterate through all the equalities - equations that say `x==y`. For each equality, we find the parent representations of the variables `x` and `y`. If they are different, then we merge the sets by assigning one parent to both, effectively stating that they are equal.
3. **Processing Inequalities**: After dealing with all the equalities, we iterate through the inequalities - equations that say `x!=y`. For each inequality, again, we find the parent representations of `x` and `y`. If they have the same parent, this means we have previously determined they are equal, which contradicts the current inequality equation. Therefore, we can return `false`.
4. **Conclusion**: After checking all inequalities, if none have caused a contradiction, it means that all equations can be satisfied, so we return `true`.

Using union-find allows us to efficiently merge groups and keep track of the connected components or disjoint sets. The two-pass approach first ensures all equalities are accounted for which forms the base state for dealing with inequalities.

Solution Approach

The solution leverages the **union-find** algorithm, also known as the **disjoint set union** (DSU) algorithm. This is well suited for problems that deal with connectivity and component relationships, like the one we have at hand.

Here's a step-by-step breakdown of how the union-find algorithm is applied in this solution:

1. **Find Function**: A `find` function is defined to determine the parent or the representative of a set to which a particular element belongs. The purpose is to find the topmost parent (the representative) of a variable. If a variable's parent is not itself, the function recursively updates and assigns the variable's parent to be the result of the `find` function. This also implements path compression, where during the find operation, we make each looked-up node point to the root (representative) directly to speed up future lookups.

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

2. **Initializing Parent Array**: A parent array `p` is initialized to keep track of the representative of each element (in this case, variables represented by lowercase letters). It starts with each element being its own parent, which means they are each in their own separate set.

```
p = list(range(26))  # 26 for the number of lowercase English letters
```

3. **Union Operation**: For each equality equation (denoted by `==`), we union the sets containing the variables of the equation. This involves changing the parent of one element to be the parent of the other element, therefore establishing that they are in the same set (they are connected or equal).

```
for e in equations:
    if e[1] == '=':
        a, b = ord(e[0]) - ord('a'), ord(e[3]) - ord('a')
        p[find(a)] = find(b)
```

   Here, `ord` function converts a character into its corresponding ASCII value, and the subtraction of `ord('a')` is done to get an index from 0 to 25 for each letter.

4. **Checking Inequalities**: After equalities are processed, we iterate over the inequality equations (denoted by `!=`). For each inequality, we check if the variables are in the same set by comparing their parents. If they have the same parent, it implies they are equal (by the union operations done previously), which contradicts the inequality condition.

```
for e in equations:
    if e[1] == '!' and find(a) == find(b):
        return False
```

5. **Returning the Result**: If no contradictions are found in the inequality equations, we return `true` since all equations can be satisfied with the unions performed.

By applying the union-find data structure for the equality relationships first and then checking for any violations in the inequality relationships later, we can efficiently resolve if all equations can hold true concurrently or not.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have the following equations:

```
["a==b", "b!=c", "c==a"]
```

Our goal is to determine if we can assign integers to each variable (`a`, `b`, and `c`) such that these equations are simultaneously satisfied.

1. **Initial Parent Array**: We initialize an array `p` representing the parents of each variable.

```
p = [0, 1, 2]  // since we have 26 letters, for simplicity let's consider only indices for a, b, and c.
```

2. **Process Equalities**: Following the equalities, `"a==b"` and `"c==a"`:

   - For `"a==b"`:
     - We find the parents of `a` (which is 0) and `b` (which is 1).
     - Since `a` and `b` are not in the same set, we perform a union by setting the parent of `a` to be the parent of `b`. Now, `p[0]` is 1.

```
p = [1, 1, 2]  // a and b are now in the same set.
```

   - For `"c==a"`:
     - We find the parents of `c` (which is 2) and `a` (now 1 after compression).
     - We perform a union by setting the parent of `c` to be the parent of `a`. Now, `p[2]` is 1.

```
p = [1, 1, 1] // a, b, and c are now all connected.
```

3. **Checking Inequalities**: We go over the inequality `"b!=c"`:
   - We find the parents for `b` and `c`, which is index 1 for both after compression.
   - Since `b` and `c` have the same parent, they are considered to be in the same set, implying they are equal according to our previous unions, which contradicts the inequality `"b!=c"`.
   - Thus, we return `false` at this step because there is no way we can satisfy this inequality given the equality connections that we have already made.

With this example, it is clear that the union-find algorithm helps efficiently determine the connectivity between variables, and due to the contradicting inequality, the entire set of equations cannot be satisfied simultaneously.

Python Solution

```
from typing import List

class Solution:
    def equationsPossible(self, equations: List[str]) -> bool:
        # Define a function to find the root of an element x in the parent array
        def find(x):
            # Recursively find the root parent of x. Path compression is used for optimization.
            if parent[x] != x:
                parent[x] = find(parent[x])
            return parent[x]

        # Initialize a parent array for union-find, where each element is its own root initially
        parent = list(range(26))

        # Union phase: process all equations that are equalities to unify groups
        for eq in equations:
            if eq[1] == '=':
                # Extract the variables from the equation and convert to numeric indices
                index1 = ord(eq[0]) - ord('a')
                index2 = ord(eq[3]) - ord('a')
                # Unify the groups by assigning the same root parent
                parent[find(index1)] = find(index2)

        # Check phase: process all equations that are inequalities to detect conflicts
        for eq in equations:
            if eq[1] == '!':
                # Extract the variables from the equation and convert to numeric indices
                index1 = ord(eq[0]) - ord('a')
                index2 = ord(eq[3]) - ord('a')
                # If the two variables belong to the same group, the equations are not possible
                if find(index1) == find(index2):
                    return False

        # If no conflicts are found, all equations are possible
        return True
```

Java Solution

```
public class Solution {
    // Parent array to represent the disjoint set (union-find) data structure.
    private int[] parent;

    // Function to determine if a given set of equations is possible.
    public boolean equationsPossible(String[] equations) {
        // Initialize parent array, where each element is its own parent.
        parent = new int[26];
        for (int i = 0; i < 26; ++i) {
            parent[i] = i;
        }

        // Process all equations of the type "a==b"
        for (String equation : equations) {
            if (equation.charAt(1) == '=') {
                int var1 = equation.charAt(0) - 'a';
                int var2 = equation.charAt(3) - 'a';
                // Union the sets in which the variables belong
                parent[find(var1)] = find(var2);
            }
        }

        // Process all equations of the type "a!=b"
        for (String equation : equations) {
            if (equation.charAt(1) == '!') {
                int var1 = equation.charAt(0) - 'a';
                int var2 = equation.charAt(3) - 'a';
                // If the variables belong to the same set, the equation is not possible
                if (find(var1) == find(var2)) {
                    return false;
                }
            }
        }

        // If we did not find a contradiction, the equations are possible
        return true;
    }

    // Helper function to find the representative of the set to which element x belongs.
    private int find(int x) {
        // Path compression: make every visited node point directly to the set representative
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
}
```

C++ Solution

```
#include <vector>
#include <string>
using namespace std;

class Solution {
public:
    vector<int> parent; // Vector to store the parent of each character

    // Finds the parent of a given element x
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Path compression
        }
        return parent[x];
    }

    // Function to check if a list of equations is possible
    bool equationsPossible(vector<string>& equations) {
        // Initialize parent vector with element itself as the parent
        parent.resize(26);
        for (int i = 0; i < 26; ++i) {
            parent[i] = i;
        }

        // First pass to process all "equal" equations
        for (auto& equation : equations) {
            int char1 = equation[0] - 'a'; // Convert char to index
            int char2 = equation[3] - 'a'; // Convert char to index
            if (equation[1] == '=') {
                // Union the sets of the two characters
                parent[find(char1)] = find(char2);
            }
        }

        // Second pass to process all "not equal" equations
        for (auto& equation : equations) {
            int char1 = equation[0] - 'a'; // Convert char to index
            int char2 = equation[3] - 'a'; // Convert char to index
            if (equation[1] == '!') {
                // If the two characters are in the same set, the equations are not possible
                if (find(char1) == find(char2)) {
                    return false;
                }
            }
        }

        return true; // All equations are possible
    }
};
```

Typescript Solution

```
// Initial parent array, where each element's index represents a unique character (a-z),
// and the value at that index represents the parent in the union-find structure.
const parents: number[] = Array.from({ length: 26 }, (_, i) => i);

// The find function locates the root of the set that the character at the given index belongs to.
// It implements path compression to flatten the structure for faster future lookups.
function find(index: number): number {
    if (parents[index] !== index) {
        parents[index] = find(parents[index]);
    }
    return parents[index];
}

// The union function joins two sets together by linking the root of one set to the root of the other.
function union(index1: number, index2: number): void {
    parents[find(index1)] = find(index2);
}

// The equationsPossible function checks if a series of equations is satisfiable.
// It uses the union-find structure to represent relationships and separations between characters.
function equationsPossible(equations: string[]): boolean {
    for (const equation of equations) {
        const index1 = equation.charCodeAt(0) - 'a'.charCodeAt(0);
        const index2 = equation.charCodeAt(3) - 'a'.charCodeAt(0);

        // Handle equivalence relationships by uniting the sets of the two characters.
        if (equation.charAt(1) === '=') {
            union(index1, index2);
        }
    }

    for (const equation of equations) {
        const index1 = equation.charCodeAt(0) - 'a'.charCodeAt(0);
        const index2 = equation.charCodeAt(3) - 'a'.charCodeAt(0);

        // Check for conflicts: if characters that should not be equal are found in the same set,
        // the equations are not possible.
        if (equation.charAt(1) === '!') {
            if (find(index1) === find(index2)) {
                return false;
            }
        }
    }

    // If there are no conflicts, the equations are possible.
    return true;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code consists of two main parts: the union operations that occur when equations with `"=="` are processed, and the find operations to check for contradictions in equations with `"!="`.

- The `find` function performs a path compression, which is an optimization of the Disjoint Set Union (DSU) data structure. With path compression, the complexity of each `find` operation is amortized to $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows very slowly and is considered almost constant for all practical purposes.
- During the first loop, there are potentially n union operations (one for each `"=="` equation).
- During the second loop, there may be up to n find operations (one for each `"!="` equation).

Since n is the number of equations given, the time complexity approximates to $O(n * \alpha(n))$, but it's commonly denoted as $O(n)$ due to the negligible growth of $\alpha(n)$.

Space Complexity

The space complexity is determined by the space required for the parent array `p`.

- There is a fixed size parent array `p` of size `26` to account for each lowercase letter from `'a'` to `'z'`.

Therefore, the space complexity of the code is $O(1)$, as space does not scale with the input size and remains constant because we are only dealing with lowercase English letters, which are just 26.