990. Satisfiability of Equality Equations

Medium Union Find Graph String Array

Problem Description

represented by a single lowercase letter, and each equation is a string of 4 characters. The equations come in two forms: 1. "xi==yi" which indicates that the two variables represented by xi and yi must be equal.

In the LeetCode problem presented, we are given a set of equations that express relationships between variables. Each variable is

Leetcode Link

- 2. "xi!=yi" which specifies that the two variables must not be equal, where xi and yi 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.

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 xi=yi. For each equality, we find the parent

representations of the variables xi and yi. If they are different, then we merge the sets by assigning one parent to both,

Here is the intuition broken into steps:

effectively stating that they are equal. 3. Processing Inequalities: After dealing with all the equalities, we iterate through the inequalities - equations that say xi!=yi. For

each inequality, again, we find the parent representations of xi and yi. 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.
- approach first ensures all equalities are accounted for which forms the base state for dealing with inequalities. Solution Approach

Using union-find allows us to efficiently merge groups and keep track of the connected components or disjoint sets. The two-pass

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.

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

2. Initializing Parent Array: A parent array p is initialized to keep track of the representative of each element (in this case, variables

compression, where during the find operation, we make each looked-up node point to the root (representative) directly to speed up future lookups.

return p[x]

same set (they are connected or equal).

satisfied with the unions performed.

Suppose we have the following equations:

Example Walkthrough

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

satisfied.

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

represented by lowercase letters). It starts with each element being its own parent, which means they are each in their own separate set.

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

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

1 for e in equations: if e[1] == '=': a, b = ord(e[0]) - ord('a'), ord(e[-1]) - 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.

inequality, we check if the variables are in the same set by comparing their parents. If they have the same parent, it implies they

4. Checking Inequalities: After equalities are processed, we iterate over the inequality equations (denoted by !=). For each

5. Returning the Result: If no contradictions are found in the inequality equations, we return true since all equations can be

By applying the union-find data structure for the equality relationships first and then checking for any violations in the inequality

are equal (by the union operations done previously), which contradicts the inequality condition.

relationships later, we can efficiently resolve if all equations can hold true concurrently or not.

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

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

We find the parents of a (which is 0) and b (which is 1).

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

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

previous unions, which contradicts the inequality "b!=c".

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

- 1 for e in equations: if e[1] == '!' and find(a) == find(b): return False
- Let's walk through a small example to illustrate the solution approach.

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

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

• 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.

For "c==a":

we have already made.

def find(x):

if parent[x] != x:

for eq in equations:

if eq[1] == '=':

if eq[1] == '!':

return True

public class Solution {

private int[] parent;

Python Solution

8

11

14

15

16

17

18

19

26

27

28

29

30

31

32

33

34

35

36

3

5

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

42

43

44

45

46

47

48

11

12

13

14

15

For "a==b":

We find the parents of c (which is 2) and a (now 1 after compression).

• We find the parents for b and c, which is index 1 for both after compression.

the contradicting inequality, the entire set of equations cannot be satisfied simultaneously.

■ We perform a union by setting the parent of c to be the parent of a. Now, p[2] is 1.

3. Checking Inequalities: We go over the inequality "b!=c":

Since b and c have the same parent, they are considered to be in the same set, implying they are equal according to our

With this example, it is clear that the union-find algorithm helps efficiently determine the connectivity between variables, and due to

Thus, we return False at this step because there is no way we can satisfy this inequality given the equality connections that

from typing import List class Solution: def equationsPossible(self, equations: List[str]) -> bool:

index1 = ord(eq[0]) - ord('a')

index1 = ord(eq[0]) - ord('a')

index2 = ord(eq[3]) - ord('a')

return False

if find(index1) == find(index2):

If no conflicts are found, all equations are possible

parent[x] = find(parent[x]) 9 return parent[x] 10 # Initialize a parent array for union-find, where each element is its own root initially 12 parent = list(range(26)) 13

Define a function to find the root of an element x in the parent array

Union phase: process all equations that are equalities to unify groups

Recursively find the root parent of x. Path compression is used for optimization.

Extract the variables from the equation and convert to numeric indices

Extract the variables from the equation and convert to numeric indices

If the two variables belong to the same group, the equations are not possible

- 20 index2 = ord(eq[3]) - ord('a')21 # Unify the groups by assigning the same root parent 22 parent[find(index1)] = find(index2) 23 24 # Check phase: process all equations that are inequalities to detect conflicts for eq in equations:
- **Java Solution**

// If the variables belong to the same set, the equation is not possible

// Initialize parent array, where each element is its own parent. parent = new int[26]; 8 for (int i = 0; i < 26; ++i) { 9 10 parent[i] = i; 11 12

// Union the sets to which the variables belong

// If we did not find a contradiction, the equations are possible

vector<int> parent; // Vector to store the parent of each character

parent[x] = find(parent[x]); // Path compression

// Parent array to represent the disjoint set (union-find) data structure.

// Function to determine if a given set of equations is possible.

public boolean equationsPossible(String[] equations) {

// Process all equations of the type "a==b"

// Process all equations of the type "a!=b"

if (equation.charAt(1) == '!') {

int var1 = equation.charAt(0) - 'a';

int var2 = equation.charAt(3) - 'a';

int var1 = equation.charAt(0) - 'a';

int var2 = equation.charAt(3) - 'a';

if (find(var1) == find(var2)) {

parent[find(var1)] = find(var2);

if (equation.charAt(1) == '=') {

for (String equation : equations) {

for (String equation : equations) {

return false;

parent[x] = find(parent[x]);

// Finds the parent of a given element x

if (parent[x] != x) {

function find(index: number): number {

if (parent[index] !== index) {

return parent[index];

parent[index] = find(parent[index]);

15 function union(index1: number, index2: number): void {

function equationsPossible(equations: string[]): boolean {

parent[find(index1)] = find(index2);

for (const equation of equations) {

if (equation.charAt(1) === '=') {

// the equations are not possible.

if (find(index1) === find(index2)) {

// If there are no conflicts, the equations are possible.

if (equation.charAt(1) === '!') {

return false;

union(index1, index2);

return parent[x];

if (parent[x] != x) {

return parent[x];

```
return true;
37
38
39
       // Helper function to find the representative of the set to which element x belongs.
       private int find(int x) {
40
           // Path compression: make every visited node point directly to the set representative
41
```

C++ Solution

#include <vector>

class Solution {

public:

#include <string>

using namespace std;

int find(int x) {

16 17 // Function to check if a list of equations is possible 18 bool equationsPossible(vector<string>& equations) { 19 // Initialize parent vector with element itself as the parent 20 parent.resize(26); 21 for (int i = 0; i < 26; ++i) { 22 parent[i] = i; 23 24 25 // First pass to process all "equal" equations 26 for (auto& equation : equations) { 27 int char1 = equation[0] - 'a'; // Convert char to index int char2 = equation[3] - 'a'; // Convert char to index 28 29 if (equation[1] == '=') { 30 // Unite the sets of the two characters 31 parent[find(char1)] = find(char2); 32 33 34 35 // Second pass to process all "not equal" equations 36 for (auto& equation : equations) { 37 int char1 = equation[0] - 'a'; // Convert char to index 38 int char2 = equation[3] - 'a'; // Convert char to index 39 if (equation[1] == '!' && find(char1) == find(char2)) { 40 // If the two characters are in the same set, the equations are not possible return false; 41 42 43 44 45 return true; // All equations are possible 46 47 }; 48 Typescript Solution 1 // Initial parent array, where each element's index represents a unique character (a-z), 2 // and the value at that index represents the parent in the union-find structure. const parent: 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.

14 // The union function joins two sets together by linking the root of one set to the root of the other.

20 // It uses the union-find structure to represent equivalences and separations between characters.

// Handle equivalence relationships by uniting the sets of the two characters.

// Check for conflicts: if characters that should not be equal are found in the same set,

6 // It employs path compression to flatten the structure for faster future lookups.

19 // The equationsPossible function checks if a series of equations is satisfiable.

const index1 = equation.charCodeAt(0) - 'a'.charCodeAt(0);

const index2 = equation.charCodeAt(3) - 'a'.charCodeAt(0);

29 30 31 32 for (const equation of equations) { 33 const index1 = equation.charCodeAt(0) - 'a'.charCodeAt(0); 34 const index2 = equation.charCodeAt(3) - 'a'.charCodeAt(0);

return true;

Space Complexity

Time and Space Complexity

10

11

13

16

18

22

23

24

25

26

27

28

35

37

38

39

40

41

42

43

44

45

46

48

47 }

17 }

12 }

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 "!=".

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).

The find function performs a path compression, which is an optimization of the Disjoint Set Union (DSU) data structure. With

Since n is the number of equations given, the time complexity approximates to 0(n * alpha(n)), but it's commonly denoted as 0(n)due to the negligible growth of alpha(n).

During the second loop, there may be up to n find operations (one for each "!=" equation).

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

are only dealing with lowercase English letters, which are just 26.

- 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 0(1), as space does not scale with the input size and remains constant because we