2782. Number of Unique Categories

Medium Union Find Counting

Leetcode Link

Problem Description

In this problem, we are given n different elements, each with an associated category. Our task is to determine the total number of unique categories among all elements. We are given a class `CategoryHandler`, which has a function `haveSameCategory(a, b)` that tells us if two elements a and b belong to the same category or not. The input n indicates the total number of elements, indexed from 0 to $n - 1$. The `haveSameCategory` function returns `true` if elements a and b are in the same category, `false` if they are in different categories, or `false` if either a or b isn't a valid index within the range. The goal is to find and return the count of unique categories.

Intuition

The approach to solve this problem is akin to a classic *Disjoint Set Union (DSU) or Union-Find* problem. The idea here is to treat categories as disjoint sets, where each element initially is in its own category set. As we go through the elements, we use the `haveSameCategory` method from `CategoryHandler` to determine if two elements are in the same category. If they are, we "union" their sets by assigning them the same representative (root). The function `find` is used to get the representative of an element's set, and if two elements are in the same category, we make one's representative the other's parent, effectively merging their sets.

This operation is applied for each pair that share a category until all elements that are in the same category are connected and have the same root. After processing all elements, the number of unique categories is equal to the number of sets that have themselves as their representative, which we count at the end and return as the final answer.

The solution uses path compression in the `find` function which ensures that each element directly points to the set representative after the first time it's looked up, which optimizes the process and keeps the time complexity under control. By iterating over each element and connecting elements in the same category, we're essentially building a forest of trees, where each tree represents a unique category. The end result is that we obtain the number of unique categories by counting the number of roots in this forest.

Solution Approach

The provided solution uses the Disjoint Set Union (DSU) data structure, also known as Union-Find, to efficiently merge elements that share the same category and determine the number of unique categories.

Here's a step-by-step breakdown of the implementation:

- A list `p` is initialized with n elements, where each element is initially set to its own index, implying that each element is its own representative.
- A nested loop is used to iterate through all pairs of elements. For each pair `(a, b)`, we check whether they belong to the same category by calling `categoryHandler.haveSameCategory(a, b)`.
- If the returned value is `true`, we find the representatives (parents) of both elements using the `find` function and merge them by setting the representative of `a` to be the representative of `b`. This effectively merges the sets that `a` and `b` belong to.
- The `find` function is a crucial part of the DSU pattern. It follows the path of parent pointers from the element `x` up to the root (representative) of the set containing `x`. It also implements path compression, a significant optimization where each element on the path is set to point directly to the root after the first lookup.
- After processing all pairs, the representatives are merged into disjoint sets with a unique representative for each set. To count the number of unique categories, we iterate over the list `p` and increment the count for each element that is its own representative (indicating it's the root of a set).
- The final count, which represents the number of unique categories, is then returned.

The algorithms and patterns used in this solution include:

- **Disjoint Set Union (DSU)** pattern for maintaining and merging disjoint sets.
- **Path Compression** optimization to flatten the structure of the tree for any affiliated elements, reducing the time complexity of subsequent `find` operations.

An example of a mathematical formula used in the implementation is the equivalence relation that's being checked in the `find` function: if $p[x] != x$, it indicates that x is not a root, and we recursively call $find(p[x])$ to retrieve the representative of x.

The primary data structure used is a one-dimensional array (`p`) which keeps track of representatives (roots) for each element, allowing us to efficiently perform unions and finds.

The solution has a time complexity that's nearly linear (amortized $O(n \cdot \alpha(n))$ where α is the Inverse Ackermann function) due to the use of path compression, making the solution highly efficient even for large values of n.

Example Walkthrough

Let's say we are given $n = 5$ elements, indexed from 0 to 4. Assume the following `haveSameCategory` results when checking pairs:

- `haveSameCategory(0, 1)` returns `false`
- `haveSameCategory(1, 2)` returns `true`
- `haveSameCategory(2, 3)` returns `true`
- `haveSameCategory(3, 4)` returns `false`
- `haveSameCategory(0, 4)` returns `true`

Based on the description above, we start by initializing a list `p` with n elements where `p = [0, 1, 2, 3, 4]`, indicating that each element is initially its own representative.

Next, we iterate through all pairs and use the `haveSameCategory` function from our `CategoryHandler`:

1. Since `haveSameCategory(0, 1)` returns `false`, we do nothing as they are not in the same category.
2. For the pair `(1, 2)`, `haveSameCategory` returns `true`. We find the representative of `1`, which is `1` itself, and the representative of `2`, which is `2` itself. We then union these two into the same set, resulting in `p = [0, 1, 1, 3, 4]`, indicating that `2` is now the representative of `2` as well.
3. With the pair `(2, 3)`, `haveSameCategory` returns `true`. Since they need to be in the same category, we find the representative of `2`, which we updated to `1` in the previous step, and the representative of `3`, which is `3`. After union, `p = [0, 1, 1, 1, 4]` because `1` is now the representative for `3`.
4. Next, `haveSameCategory(3, 4)` returns `false`, so no action is taken.
5. Lastly, `haveSameCategory(0, 4)` returns `true`. We find the representatives of `0` and `4`, which are `0` and `4`, respectively. After union, `p = [0, 1, 1, 1, 0]`, which means `0` is now the representative for `4`.

We have completed the iterations. Now, to count the number of unique categories, we count the number of set representatives:

- Index `0` is its own representative, indicating it is a unique category.
- Index `1` is the representative for indices `2` and `3`, indicating another unique category.
- Indices `2`, `3`, and `4` are not their own representatives.

So, we have two unique categories represented by `p = [0, 1, 1, 1, 0]`. The final answer is `2`.

Python Solution

```python
# A class definition for handling categories.
# Note: The actual implementation of CategoryHandler.haveSameCategory should be provided.
class CategoryHandler:
    def haveSameCategory(self, a: int, b: int) -> bool:
        # This method should determine if elements `a` and `b` have the same category.
        pass

class Solution:
    def numberOfCategories(
        self, n: int, categoryHandler: Optional['CategoryHandler']
    ) -> int:
        # Function to find the root of an element `x` using path compression.
        def find_root(x: int) -> int:
            if parent[x] == x:
                return x  # Base case: x is the root (representative).
            parent[x] = find_root(parent[x])  # Path compression for efficiency.
            return parent[x]

        # Initialization of each element to be its own parent.
        parent = list(range(n))

        # Iterate through all possible pairs of elements.
        for a in range(n):
            for b in range(a + 1, n):  # To avoid unnecessary checks, start from a + 1.
                # If they are commonly belong to the same category, unite their sets.
                if category_handler.haveSameCategory(a, b):
                    parent[find_root(a)] = find_root(b)

        # Count the number of unique parents (root elements) to determine
        # the number of different categories.
        # This is done by checking if the parent of an element equals its index.
        return sum(i == x for i, x in enumerate(parent))  # i == x
```

```python
# Example usage:
# Assuming that `CustomCategoryHandler` is a class derived from `CategoryHandler`
# with a specific implementation for the method `haveSameCategory`.
# solution = Solution()
# number_of_categories = solution.numberOfCategories(n, CustomCategoryHandler())
```

Java Solution

```java
/**
 * The Solution class implements the numberOfCategories method which uses the union-find algorithm to determine the number of unique
 * a.
 */
class Solution {
    // The array 'parents' is used to track the parent of each element, initializing each element as its own parent.
    private int[] parents;

    /**
     * This method calculates the number of unique categories given the 'CategoryHandler' object.
     *
     * @param n The total number of elements to be considered.
     * @param categoryHandler An object that provides a method to check if two elements belong to the same category.
     * @return The number of unique categories among the elements.
     */
    public int numberOfCategories(int n, CategoryHandler categoryHandler) {
        // Initialize the parents array, setting each item's parent to itself.
        parents = new int[n];
        for (int i = 0; i < n; ++i) {
            parents[i] = i;
        }

        // Iterate over all pairs of elements and union them if they are in the same category.
        for (int a = 0; a < n; ++a) {
            for (int b = a + 1; b < n; ++b) {
                if (categoryHandler.haveSameCategory(a, b)) {
                    union(a, b);
                }
            }
        }

        // Count the number of unique categories by counting the number of elements that are their own parent.
        int countCategories = 0;
        for (int i = 0; i < n; ++i) {
            if (i == parents[i]) {
                ++countCategories;
            }
        }
        return countCategories;
    }

    /**
     * The find method is used for finding the root parent of 'x'. Also, path compression is applied for optimization.
     *
     * @param x The element to find the root parent of.
     * @return The root parent of 'x'.
     */
    private int find(int x) {
        // If 'x' is not its own parent, recursively find its parent and apply path compression.
        if (parents[x] != x) {
            parents[x] = find(parents[x]);
        }
        return parents[x];
    }

    /**
     * The union method merges two containing 'a' and 'b' by attaching one's root parent to the other's parent.
     *
     * @param a The first element to union.
     * @param b The second element to union.
     */
    private void union(int a, int b) {
        // Find the root parents of both 'a' and 'b' and make one the parent of the other.
        int rootA = find(a);
        int rootB = find(b);
        parents[rootA] = rootB; // Union by attaching rootA's parent to rootB.
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric>
#include <functional>
using namespace std;

/**
 * A definition for a category handler.
 */
class CategoryHandler {
public:
    CategoryHandler(vector<vector<int>> categories);
    bool haveSameCategory(int a, int b);
};

class Solution {
public:
    /**
     * Counts the number of unique categories among 'n' items.
     * @param n Number of items.
     * @param categoryHandler Pointer to a CategoryHandler instance.
     * @return The count of unique categories.
     */
    int numberOfCategories(int n, CategoryHandler& categoryHandler) {
        vector<int> parents(n);
        // Initialize the parent vector with elements pointing to themselves.
        iota(parents.begin(), parent.end(), 0);

        // Function to find the root of the set that 'x' belongs to.
        function<int(int)> find = [&](int x) {
            if (parents[x] != x) { // Path compression.
                parents[x] = find(parents[x]);
            }
            return parents[x];
        };

        // Traverse all pairs and union the sets if they are of the same category.
        for (int a = 0; a < n; ++a) {
            for (int b = a + 1; b < n; ++b) {
                if (categoryHandler->haveSameCategory(a, b)) {
                    parents[find(a)] = find(b);
                }
            }
        }

        // Count the number of items that are their own parent (root of the set).
        int count = 0;
        for (int i = 0; i < n; ++i) {
            count += i == parents[i]; // Increment count for each root.
        }
        return count;
    }
};
```

Typescript Solution

```typescript
// Utility function to find the root of the category set that a given element belongs to
const findRoot = (element: number, parents: number[]): number => {
    if (parents[element] !== element) {
        parents[element] = findRoot(parents[element], parents); // Path compression for optimization
    }
    return parents[element];
};

// Main function to calculate the number of distinct categories
const calculateNumberOfCategories = (n: number, categoryHandler: CategoryHandler): number => {
    // Initialize an array to represent the parent of each element, with each element initially being its own parent
    const parents = new Array(n).fill(0).map((_, index) => index);

    // Iterate through all possible pairs checking if they have the same category
    for (let a = 0; a < n; ++a) {
        for (let b = a + 1; b < n; ++b) {
            // If two elements are found to be in the same category, unite their sets by assigning the same root
            if (categoryHandler.haveSameCategory(a, b)) {
                parents[findRoot(a, parents)] = findRoot(b, parents);
            }
        }
    }

    // Iterate through all the elements to count the number of distinct roots/categories
    let distinctCategories = 0;
    for (let i = 0; i < n; ++i) {
        // If the element is the root (parent of itself), then it's a distinct category
        if (i === parents[i]) {
            ++distinctCategories;
        }
    }

    return distinctCategories;
};
```

Time and Space Complexity

Time Complexity

The time complexity of this code is primarily influenced by the double loop where we check if two elements have the same category using `categoryHandler.haveSameCategory(a, b)`. This loop runs n from 0 to $n-1$ and nested loop i runs from $a+1$ to n.

- The outer loop runs n times (from 0 to $n-1$).
- For each iteration of the outer loop, the inner loop runs $n - a - 1$ times.

The number of times the inner loop runs can be described by the sum: $Sigma(n - a - 1)$ from $a=0$ to $n-1$, which equals $(n-1) + (n-2) + ... + 1$, which is a known arithmetic series sum that equals to $(n*(n-1))/2$. Each call to the inner loop incurs a call to `haveSameCategory`, which could have its own time complexity depending on its implementation. If we assume it's $O(1)$, the total time complexity becomes $O(n^2)$.

Additionally, there's a union-find algorithm applied through the `find` function, which has a complexity of $O(alpha(n))$ for each `find` operation, where $alpha$ is the Inverse Ackermann function which is very slowly growing and for all practical purposes is considered almost constant.

So we need to also consider this complexity for each call to `haveSameCategory`. If there are n calls to `haveSameCategory` that return true (and thus perform the union operation), the total time complexity for union-find would be $O(n*alpha(n))$.

However, since every element can be part of at most one union operation (due to the properties of the disjoint set union structure), n will at most be n, and thus, we end up with $O(n*alpha(n))$.

The final line of the code, which computes the number of categories present, is simply an iteration over the array p of length n.

Assuming the complexity of `haveSameCategory` is $O(1)$, the total time complexity of the code is dominated by the double loop and it is $O(n^2 + n*alpha(n))$. But $n*alpha(n)$ is negligible compared to n^2, so we can simply state the total time complexity as $O(n^2)$.

Space Complexity

The space complexity of the code is $O(n)$. This comes from the space required to store the parent array p, which has one element for each of the n nodes. Space complexity is not greatly affected by the recursion in the `find` function because the disjoint set union-find structure ensures that the trees are very shallow (due to path compression), and thus the call stack depth is effectively bounded by $alpha(n)$, which is almost constant.

Therefore, the dominant space requirement is the parent array, resulting in $O(n)$ space complexity.