1722. Minimize Hamming Distance After Swap Operations

Medium | Depth-First Search | Union Find | Array | Leetcode Link

Problem Description

In this problem, we have two arrays named `source` and `target`, both of the same length `n`. Additionally, we are given an array `allowedSwaps` containing pairs of indices `[ai, bi]`. These pairs indicate that we can swap the elements at these particular indices in the `source` array as many times as we want.

The task is to find the minimum Hamming distance between the `source` and `target` arrays after performing any number of allowed swaps. The Hamming distance is defined as the count of positions where the elements of the two arrays differ.

Thus, the main objective is to minimize the differences between the two arrays using the swaps permitted by the `allowedSwaps` array.

Intuition

To minimize the Hamming distance, we want to arrange the elements in `source` such that as many of them as possible match with the corresponding elements in `target`.

The intuition behind the solution is to consider that each swap allows us to group certain elements together. If we think of these swaps as connections in a graph, we can use the Union-Find (Disjoint Set Union) algorithm to find the groups of indices that can be considered one interchangeable set. Within these sets, we can rearrange the elements freely.

Once we form the groups, we check for each element in the `target` array: if the corresponding group in `source` contains that element, we reduce the count for that element (as we can match it with the target). If it's not present, we increment the result, signifying a Hamming distance increase.

This approach efficiently calculates the minimum Hamming distance by determining and utilizing the flexibility provided by `allowedSwaps`.

Solution Approach

The solution utilizes the Union-Find data structure, also known as the Disjoint Set Union (DSU) algorithm. Union-Find helps in managing the grouping and merging of index sets that can be interchanged freely due to the allowed swaps.

Here are the key steps and elements of the solution:

1. **Initialization**: We start by initializing a parent array `p`, where each index is initially the parent of itself. This represents that initially, each index can only be swapped within its own singleton set.

2. **Union Operation**: We iterate through each swap pair in `allowedSwaps` and perform a union operation. The union is done by finding the parent of both elements in the swap pair and making one the parent of the other. This effectively merges the sets, allowing all indices in the merged set to be interchangeable.

   The `find` function is used to recursively find the root parent of an index, implementing path compression by assigning the root parent directly to each index along the path. This process reduces the complexity of subsequent `find` operations.

3. **Grouping Elements of `source`**: We use a dictionary `mp` that maps each set (identified by its parent index) to a `Counter` that tracks the frequency of each element in that set within the `source` array.

4. **Calculating the Hamming Distance**: We iterate over each element in the `target` array and look for the set in `source` that corresponds to its index. If the element in `target` exists in this set (the set's `Counter` has a count greater than zero), we reduce the count for that element and do not increment the Hamming distance. If it does not exist, we increment the result as it represents a difference between `source` and `target`.

5. **Returning the Result**: After checking all elements, the `res` variable holds the minimum Hamming distance, which we return as the final output.

The Union-Find algorithm is crucial here, as it allows us to efficiently determine which elements in `source` can be swapped to match elements in `target`. The use of the `Counter` within the groups formed by Union-Find enables us to track and match elements between `source` and `target`, minimizing the Hamming distance. The end result is the minimum number of positions where `source` and `target` arrays differ after performing the swaps.

Example Walkthrough

Let's illustrate the solution approach with a given example:

- `source = [1, 2, 3, 4]`
- `target = [2, 1, 4, 5]`
- `allowedSwaps = [[0, 1], [2, 3]]`

We want to find the minimum Hamming distance between `source` and `target` after performing any number of allowed swaps.

1. **Initialization**: Create a parent array `p` of the same length as `source`. Each index starts with itself as its parent: `p = [0, 1, 2, 3]` - This implies each index is initially in its own set.

2. **Union Operation**:
   - For the first pair in `allowedSwaps` (`[0, 1]`), we perform a union operation. We find the parents for `0` and `1`, which are themselves. Then, we make one of them the parent of the other. Assume we make `1` the parent of `0`: `p = [1, 1, 2, 3]`.
   - For the second pair (`[2, 3]`), we similarly merge these two indices' sets. `2` becomes the parent of `3`: `p = [1, 1, 2, 2]`.
3. **Grouping Elements of `source`**: We create a dictionary `mp` which will hold `Counter` objects representing the frequency of each element in `source`, grouped by their parent index after all unions: `mp = {1: Counter({1: 1, 2: 1}), 2: Counter({3: 1, 4: 1})}`

4. **Calculating the Hamming Distance**:

   - We start with `res = 0`. For every element in `target`, we find its corresponding set in `source` using `p`. For example:
     - `target[0] = 2` should look in the set at `p[0]` which is `1`. Since `2` is present there, we match `source[0] = 1` with `target[0] = 2` without incrementing `res`.
     - `target[1] = 1` looks in set `p[1] = 1`. The element `1` is present there, so no increment to `res`.
     - `target[2] = 4` refers to the set at `p[2] = 2`. The element `4` is present, so no change to `res`.
     - `target[3] = 5` goes to the set at `p[3] = 2`, but `5` is not in `Counter({3: 1, 4: 1})`. So, `res` increments by 1.
5. **Returning the Result**: We've determined the Hamming distance by iterating through all the elements in `target`. As only one element, `5`, couldn't be matched after performing allowed swaps, our final Hamming distance is `res = 1`.

After applying the Union-Find algorithm and efficiently tracking elements in each group, we conclude that the minimum number of differing positions between `source` and `target` is 1, which we return as the final result.

Python Solution

```python
from collections import defaultdict, Counter
from typing import List

class Solution:
    def minimumHammingDistance(self, source: List[int], target: List[int], allowedSwaps: List[List[int]]) -> int:
        # Get the length of source which is the same as the length of the target
        n = len(source)

        # Initialize parent array for disjoint set (union-find)
        parent = list(range(n))

        # Function to find the representative (root) of a set for element x
        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x])  # Path compression
            return parent[x]

        # Perform union operations for each allowed swap
        for i, j in allowedSwaps:
            # Union by setting the parent of the representative of i to the representative of j
            parent[find(i)] = find(j)

        # Map to store counts of elements grouped by their disjoint set representative
        disjoint_set_counter = defaultdict(Counter)
        # Populate the counts of elements for each representative
        for i in range(n):
            disjoint_set_counter[find(i)][source[i]] += 1

        # Variable to store the result of minimum Hamming distance
        result = 0
        # Calculate the Hamming distance
        for i in range(n):
            # If the element in the target is in the disjoint set and count is greater than 0
            if disjoint_set_counter[find(i)][target[i]] > 0:
                # Decrement the count of the target element in the disjoint set
                disjoint_set_counter[find(i)][target[i]] -= 1
            else:
                # If the element is not found in the set, increment result
                result += 1

        # Return the final result of minimum Hamming distance
        return result

```

Java Solution

```java
class Solution {
    // Parent array representing the root of each element's set.
    private int[] parent;

    // This function calculates the minimum Hamming distance between source and target arrays.
    public int minimumHammingDistance(int[] source, int[] target, int[][] allowedSwaps) {
        int n = source.length;
        parent = new int[n];

        // Initialize parent array so that each element is its own root.
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }

        // Apply union operation for allowed swaps.
        for (int[] swap : allowedSwaps) {
            int rootA = find(swap[0]);
            int rootB = find(swap[1]);
            parent[rootA] = rootB;
        }

        // Create a mapping from root to a frequency map of the elements associated with that root.
        Map<Integer, Map<Integer, Integer>> frequencyMap = new HashMap<>();
        for (int i = 0; i < n; i++) {
            int root = find(i);
            Map<Integer, Integer> rootFrequencyMap = frequencyMap.computeIfAbsent(root, k -> new HashMap<>());
            int elementCount = rootFrequencyMap.getOrDefault(source[i], 0);
            rootFrequencyMap.put(source[i], elementCount + 1);
        }

        // Calculate Hamming distance after applying allowed swaps.
        int minDistance = 0;
        for (int i = 0; i < n; i++) {
            int root = find(i);
            Map<Integer, Integer> rootFrequencyMap = frequencyMap.get(root);
            if (rootFrequencyMap.getOrDefault(target[i], 0) > 0) {
                // If the element in target exists in the frequency map, decrement its count.
                rootFrequencyMap.put(target[i], rootFrequencyMap.get(target[i]) - 1);
            } else {
                // Otherwise, increment the Hamming distance.
                minDistance++;
            }
        }

        return minDistance; // Return the minimum Hamming distance.
    }

    // The 'find' function implements union-find algorithm to find the root of an element.
    private int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Path compression.
        }
        return parent[x];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>

class Solution {
public:
    std::vector<int> parent; // The 'p' vector is renamed to 'parent' for better readability.

    // Helper function to find the root of an element using path compression.
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    // Main function to calculate the minimum Hamming distance after allowed swaps.
    int minimumHammingDistance(std::vector<int>& source, std::vector<int>& target, std::vector<std::vector<int>>& allowedSwaps) {
        int n = source.size();
        parent.resize(n);
        // Initialize each element's root to be the element itself.
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }
        // Apply union operations for each allowed swap.
        for (auto& e : allowedSwaps) {
            parent[find(e[0])] = find(e[1]);
        }

        // Use a map to count occurrences of elements within each connected component.
        std::unordered_map<int, std::unordered_map<int, int>> countMap;
        for (int i = 0; i < n; ++i) {
            ++countMap[find(i)][source[i]];
        }

        int result = 0; // This will hold the final result, the minimum Hamming distance.
        // Compare the target array against the count map to calculate the Hamming distance.
        for (int i = 0; i < n; ++i) {
            if (countMap[find(i)][target[i]] > 0) {
                --countMap[find(i)][target[i]];
            } else {
                ++result;
            }
        }
        return result; // Return the minimum Hamming distance calculated.
    }
};

```

Typescript Solution

```typescript
// Import statements are not required in TypeScript for vector-like structures;
// instead, we use Array and TypeScript's included Map object.

// Global variable 'parent' to track the root of each element.
let parent: number[] = [];

// Helper function to find the root of an element using path compression.
const find = (x: number): number => {
    if (parent[x] !== x) {
        parent[x] = find(parent[x]);
    }
    return parent[x];
};

// Main function to calculate the minimum Hamming distance after allowed swaps.
const minimumHammingDistance = (source: number[], target: number[], allowedSwaps: number[][]): number => {
    const n = source.length;
    parent = Array.from({ length: n }, (_, index) => index); // Initialize each element's root to be the element itself.

    // Apply union operations for each allowed swap.
    allowedSwaps.forEach(([a, b]) => {
        parent[find(a)] = find(b);
    });

    // Use a map to count occurrences of elements within each connected component.
    const countMap: Map<number, Map<number, number>> = new Map();
    source.forEach((value, index) => {
        const root = find(index);
        if (!countMap.has(root)) {
            countMap.set(root, new Map());
        }
        const componentCountMap = countMap.get(root);
        if (componentCountMap) {
            componentCountMap.set(value, (componentCountMap.get(value) || 0) + 1);
        }
    });

    let result = 0; // This will hold the final result, the minimum Hamming distance.
    // Compare the target array against the count map to calculate the Hamming distance.
    target.forEach((value, index) => {
        const root = find(index);
        const componentCountMap = countMap.get(root);
        if (componentCountMap) {
            if (componentCountMap.get(value)) {
                componentCountMap.set(value, componentCountMap.get(value)! - 1);
                // If an item is present in both target and source within the same component, decrement the count.
                if (componentCountMap.get(value) === 0) {
                    componentCountMap.delete(value); // Remove the entry if count becomes zero to save space.
                }
            } else {
                result++; // Increment the result if the item was not found.
            }
        }
    });

    return result; // Return the minimum Hamming distance calculated.
};

```

Time and Space Complexity

The code snippet provided is a solution for computing the minimum Hamming Distance by allowing swaps between indices as dictated by the `allowedSwaps` list. Let's analyze the time and space complexity of the given code.

Time Complexity

The time complexity of the code can be broken down as follows:

1. The `find` function essentially implements the path compression technique in union-find and it runs in amortized `O(1)` time.

2. The loop over `allowedSwaps` to unite indices using the union-find data structure performs `O(m)` operations, where `m` is the length of `allowedSwaps`. Since the `find` function has an amortized complexity of `O(1)` for each operation, this step will have a time complexity of `O(m)`.

3. Building the `mp` dictionary, which maps the root parent from the union-find structure to a counter of elements, involves iterating over each element in `source`. For each element, we call `find`, which is `O(1)` amortized, so the loop results in a time complexity of `O(n)`.

4. The final for-loop iterates over `target` and checks if the corresponding element exists in the counter associated with its group's root parent. This is once again `O(n)` time because it involves `n` lookups and updates to the dictionary.

Combining these steps, the overall time complexity is `O(m + 2n)` which simplifies to `O(m + n)`.

Space Complexity

The space complexity of the code considers the following factors:

1. The disjoint set data structure (union-find) represented by the list `p` which uses `O(n)` space.

2. The `mp` dictionary containing at most `n` keys (in the worst case, no elements are swapped) and their associated counters, which in total will not exceed `n` elements. Therefore, `mp` utilizes `O(n)` space.

3. There is a negligible space used for variables like `i`, `j`, and `res`, which do not scale with the input.

Thus, the space complexity of the algorithm is also `O(n)`.

In summary:

- Time Complexity: `O(m + n)`
- Space Complexity: `O(n)`