947. Most Stones Removed with Same Row or Column

Medium Depth-First Search Union Find Graph Hash Table Leetcode Link

Problem Description

In this problem, we're given `n` stones positioned at integer coordinate points in a 2-dimensional plane. The key rule is that no two stones can occupy the same coordinate point. A stone can be removed if there is at least one other stone sharing the same row or column on the grid, and that other stone is still on the plane (i.e., it hasn't been removed).

Our task is to find out the maximum number of stones that can be removed while following the stated rule. The stones are represented in an array named `stones`, with each element `stones[i]` being a list of 2 integers: `[xi, yi]`, which corresponds to the location of the `i-th` stone on the 2D plane.

Intuition

The intuition behind the solution begins by understanding that if two stones share the same row or the same column, they are part of the same connected component. In the context of the problem, connected components are groups of stones that are all connected by rows or columns in such a way that if you pick any stone in a connected component, you'd be able to trace to any other stone in the same connected component through shared rows or columns. The last stone in such a connected component can never be removed as there would be no other stone in the same row or column.

Now, the key observation is that the maximum number of stones that can be removed is equal to the total number of stones minus the number of these connected components. To illustrate, imagine you have three stones forming a straight line either horizontally or vertically. You can remove the two endpoint stones but have to leave the middle one, making the connected components count as one.

The provided Python solution employs the union-find (or disjoint-set) data structure to efficiently find connected components. This structure helps to keep track of elements that are connected in some way, and in this case, is used to identify stones that are in the same row or column.

Here's the breakdown of how the union-find algorithm is applied:

1. Each stone is represented by a node, whose initial "parent" is itself, indicating that it is initially in its own connected component.
2. Iterate over all stones. For each stone `(x, y)`, unify the component containing `x` with the component containing `y + n` (we offset `y` by `n` to avoid collisions between row and column indices as they could be the same).
3. After all stones have been iterated over, we count unique representatives of the connected components which are the parent nodes for the rows and columns.
4. The answer is the total number of stones minus the number of unique representatives (connected components).

By applying union-find, the solution achieves a merging process of the stones that lie in the same connected component quickly, leading to an efficient way to calculate the maximum number of stones that can be removed.

Solution Approach

The solution implements a Union-Find data structure to keep track of connected components. This data structure is particularly useful in dealing with problems that involve grouping elements into disjoint sets where the connectivity between the elements is an essential attribute.

Here's a step-by-step explanation of the solution:

Initialization

- An array `p` is created with size `2*n`. This is because we have `n` possible x-coordinates (0 to n-1) and `n` possible y-coordinates (0 to n-1), but we need to separate the representation to avoid collision, hence `n` is doubled. The array `p` represents the parent of each node, where a node is either a row or a column index.
- Initially, every element is set to be its own parent.

The `find` function

- The `find` function is a standard function in Union-Find, which returns the representative (root parent) of the disjoint set that `x` belongs to.
- If `x` does not point to itself, we recursively find the parent of `x` until we reach the root while applying path compression. Path compression means we set `p[x]` to its root directly to flatten the structure, which speeds up future `find` operations on elements in the same set.

Unification Process

- The main loop iterates over every stone.
- For a given stone at coordinates `(x, y)`, we unify the set containing `x` with the set containing `y + n` to ensure we don't mix up rows and columns.
- The unification is done by setting the parent of the set containing `x` to be the parent of the set containing `y + n` using the `find` function.

Counting Connected Components

- A set `s` is used to store unique parents of the stones, which comes from applying the `find` function on every stone's x-coordinate.
- The number of unique parents indicates the number of connected components.

Calculating the Answer

- Finally, the solution is the total number of stones minus the number of connected components. To see why this is the case, for each connected component, one stone will inevitably remain, making all others removable.
- `len(stones) - len(s)` gives us the count of removable stones, satisfying the problem's requirement.

By implementing Union-Find, the solution approach efficiently identifies and unifies stones into connected components and calculates the maximum number of stones that can be removed, leading to an effective strategy for this problem.

Example Walkthrough

Let's consider a small example with 5 stones to illustrate the solution approach.

Assume we have `n = 5` stones at the following coordinates on a 2D plane: [[0,0], [0,2], [1,1], [2,0], [2,2]]. So our `stones` array looks like this: `stones = [[0,0], [0,2], [1,1], [2,0], [2,2]]`.

Steps According to the Solution Approach:

1. **Initialization**: We create an array `p` of size `2*n = 10`. This array will help us keep track of the parent of each coordinate considering both `x` and `y` coordinates.

2. **The `find` function**: Let's define our `find` function which takes an index `x` and recursively finds the representative of `x`'s set, while applying path compression.

3. **Unification Process**:

   - Looking at `stones[0] = [0,0]`, we unify the x-coordinate (0) with the y-coordinate (0 + 5) to avoid collision with the x-coordinates.
   - Then, `stones[1] = [0,2]`, we unify 0 with 2 + 5.
   - For `stones[2] = [1,1]`, we unify 1 with 1 + 5.
   - Next, `stones[3] = [2,0]`, we unify 2 with 0 + 5.
   - Finally, `stones[4] = [2,2]`, we unify 2 with 2 + 5.
4. **Counting Connected Components**:

   - We keep a set `s` and insert the root parent for each x-coordinate after unification.
   - After the unification, we have root parents for the x-coordinates: [0, 1, 2]. Let's assume union-find identifies 0, 1, and 2 as the roots for our example, meaning we have 3 connected components.
5. **Calculating the Answer**:

   - We have a total of 5 stones, and we've identified 3 connected components. The maximum number of stones that we can remove is `len(stones) - len(s)` which is `5 - 3 = 2`. Thus, we can remove a maximum of 2 stones while ensuring that no stone is isolated.

This walkthrough demonstrates the solution approach using the union-find algorithm to efficiently compute the maximum number of stones that can be removed according to the given rules. By identifying the connected components where stones share the same row or column, union-find allows us to easily count these connected components and therefore calculate the maximum number of removable stones.

Python Solution

```python
class Solution:
    def removeStones(self, stones: List[List[int]]) -> int:
        def find_root(root_id):
            """
            Recursive function that finds the root of a disjoint set.
            Path compression is applied to flatten the structure for efficiency.
            """
            if parent[root_id] != root_id:
                parent[root_id] = find_root(parent[root_id])
            return parent[root_id]

        # Given the constraints of the problem, there can be at most 20000 nodes
        # because stones could be placed in rows and columns 0 through 9999.
        num_nodes = 10000
        # Create an array representing the disjoint set forest with an initial parent of itself.
        parent = list(range(num_nodes * 2))

        # Iterate through each stone and unify their row and column into the same set.
        for x, y in stones:
            parent[find_root(x)] = find_root(y + num_nodes)

        # Use a set comprehension to store the unique roots of all stones' rows and columns.
        unique_roots = {find_root(x) for x, _ in stones}

        # Calculate how many stones can be removed by subtracting the number of unique sets
        # (which have to remain) from the total number of stones.
        return len(stones) - len(unique_roots)

# Note: This code assumes that the List class has been imported from the typing module:
# from typing import List

```

Java Solution

```java
class Solution {
    private int[] parent; // Array to represent the parent of each element in the Disjoint Set Union (DSU)

    public int removeStones(int[][] stones) {
        int n = 10010; // Representative value for scaling row and column indices
        parent = new int[n << 1]; // Initialize the parent array for the DSU
        for (int i = 0; i < parent.length; ++i) {
            parent[i] = i; // Initially, each element is its own parent
        }
        for (int[] stone : stones) {
            // Perform union of stone's row and column by setting the parent of the stone's row to the representative
            // of the stone's adjusted column index
            parent[find(stone[0])] = find(stone[1] + n);
        }
        Set<Integer> uniqueRoots = new HashSet<>(); // Set to store unique roots after unions have been performed
        for (int[] stone : stones) {
            // Add the representative of each stone's row to the set of unique roots
            uniqueRoots.add(find(stone[0]));
        }
        // The number of stones that can be removed is the total number of stones minus the number of unique roots,
        // since each root represents a connected component
        return stones.length - uniqueRoots.size();
    }

    private int find(int x) {
        // Recursively find the root representative of the element `x`. Path compression is applied here to flatten
        // the structure of the tree, effectively speeding up future `find` operations.
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Class member to hold the parent pointers for the Union-Find data structure
    vector<int> parents;

    // Function to remove as many stones as possible while ensuring at least one
    // stone is left in the same row or column
    int removeStones(vector<vector<int>>& stones) {
        // Define a large enough value for the maximum coordinate
        int maxCoord = 10010;
        // Resize the parents vector to double the value since we are mapping
        // 2-D coordinates to a 1-D array
        parents.resize(maxCoord << 1);
        // Initialize the parent of each element to be itself
        for (int i = 0; i < parents.size(); ++i) {
            parents[i] = i;
        }

        // Perform union operations for the stones
        for (auto& stone : stones) {
            parents[find(stone[0])] = find(stone[1] + maxCoord);
        }

        // Using a set to store unique roots after path compression
        unordered_set<int> uniqueRoots;
        for (auto& stone : stones) {
            uniqueRoots.insert(find(stone[0]));
        }

        // The number of stones that can be removed is the total number of stones
        // minus the number of unique roots in the disjoint-set forest
        return stones.size() - uniqueRoots.size();
    }

    // Function to find the root of the element with path compression
    int find(int x) {
        if (parents[x] != x) {
            parents[x] = find(parents[x]); // Path compression step
        }
        return parents[x];
    }
};

```

Typescript Solution

```typescript
// Type definition for a 2D point.
type Point = [number, number];

// Global variable to hold parent pointers for the disjoint-set (Union-Find) data structure.
const parents: number[] = [];

// Function to initialize 'parents' whereby each element is its own parent.
function initParents(size: number): void {
    for (let i = 0; i < size; i++) {
        parents[i] = i;
    }
}

// Function to find the root of element 'x' with path compression.
function find(x: number): number {
    if (parents[x] !== x) {
        parents[x] = find(parents[x]); // Path compression step
    }
    return parents[x];
}

// Function to remove as many stones as possible while ensuring
// at least one stone is left in the same row or column.
function removeStones(stones: Point[]): number {
    // Define a large enough value for the maximum coordinate.
    const maxCoord = 10010;
    // Initialize the 'parents' collection to twice the value of 'maxCoord'.
    initParents(maxCoord << 1);

    // Perform union operations for the stones.
    stones.forEach(([row, col]) => {
        const rowParent = find(row);
        const colParent = find(col + maxCoord);
        parents[rowParent] = colParent;
    });

    // Using a set to store unique roots after path compression,
    // which indicates the connected components.
    const uniqueRoots = new Set<number>();
    stones.forEach(([row, _]) => {
        uniqueRoots.add(find(row));
    });

    // The number of stones that can be removed is the total number of stones
    // minus the number of unique roots in the disjoint-set forest.
    return stones.length - uniqueRoots.size;
}

```

Time and Space Complexity

The given code snippet uses the Union-Find algorithm to remove as many stones as possible on a 2D grid while ensuring at least one stone is left in a connected group (a group of stones connected by either the same row or the same column). The time and space complexity analysis of this code is as follows:

Time Complexity:

1. The `find` function is at most traversing the depth of the union-find tree, which, with path compression, results in nearly constant time per operation. However, worst-case without any optimizations can be $O(n)$, but realistically, with path compression, it is closer to $O(\alpha(n))$, where $\alpha(n)$ is the Inverse Ackermann function, which grows very slowly and is practically considered constant time.
2. There is a loop that runs once for each stone, and inside this loop, two `find` operations are performed, one for the x-coordinate and one for the y-coordinate shifted by `n` to keep the x and y coordinates distinct in the union-find structure.

Considering there are `m` stones, each union-find operation is $O(\alpha(n))$, the overall time complexity across all stones will be $O(m * \alpha(n))$.

Since the Inverse Ackermann function $\alpha(n)$ is practically constant for all reasonable values of n, the time complexity simplifies to $O(m)$.

Space Complexity:

1. An array `p` of size `n << 1` (or 2 * n) is created to represent the union-find structure, where `n` is a constant representing the maximum number of different row/column values to expect, set to 10010 in the code. Hence, the space used by `p` is $O(n)$.
2. A set `s` is created to store unique roots after path compression. In the worst case, this set will contain all the stones if none share the same row or column, hence an $O(m)$ space complexity at worst.

Combining these, the total space complexity is $O(n + m)$ which, given the constant size of `n`, simplifies to $O(m)$, where `m` is the number of stones.

In summary:

- Time Complexity: $O(m)$, where m is the number of stones.
- Space Complexity: $O(m)$, where m is the number of stones.