Number of Distinct Islands II

Given a non-empty 2D array `grid` of 0's and 1's, an **island** is a group of `1`'s (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Count the number of **distinct** islands. An island is considered to be the same as another if they have the same shape, or have the same shape after **rotation** (90, 180, or 270 degrees only) or **reflection** (left/right direction or up/down direction).

Example 1:

```
11000
10000
00001
00011
```

Given the above grid map, return `1`.

Notice that:

```
11
1
```

and

```
 1
11
```

are considered **same** island shapes. Because if we make a 180 degrees clockwise rotation on the first island, then two islands will have the same shapes.

Example 2:

```
11100
10001
01001
01110
```

Given the above grid map, return `2`.

Here are the two distinct islands:

```
111
1
```

and

```
1
1
```

Notice that:

```
111
1
```

and

```
1
111
```

are considered **same** island shapes. Because if we flip the first array in the up/down direction, then they have the same shapes.

Note: The length of each dimension in the given `grid` does not exceed 50.

Solution 1

After we get coordinates for an island, compute all possible rotations/reflections (https://en.wikipedia.org/wiki/Dihedral_group) of it and then sort them using the default comparison. The first representation in this order is then the canonical one.

```
class Solution {
public:
    map<int, vector<pair<int,int>>> mp;

    void dfs(int r, int c, vector<vector<int>> &g, int cnt) {
        if ( r < 0 || c < 0 || r >= g.size() || c >= g[0].size()) return;
        if (g[r][c] != 1) return;
        g[r][c] = cnt;
        mp[cnt].push_back({r,c});
        dfs(r+1,c,g,cnt);
        dfs(r-1,c,g,cnt);
        dfs(r,c+1,g,cnt);
        dfs(r,c-1,g,cnt);
    }

    vector<pair<int,int>> norm(vector<pair<int,int>> v) {
        vector<vector<pair<int,int>>> s(8);
        // compute rotations/reflections.
        for (auto p:v) {
            int x = p.first, y = p.second;
            s[0].push_back({x,y});
            s[1].push_back({x,-y});
            s[2].push_back({-x,y});
            s[3].push_back({-x,-y});
            s[4].push_back({y,-x});
            s[5].push_back({-y,x});
            s[6].push_back({-y,-x});
            s[7].push_back({y,x});
        }
        for (auto &l:s) sort(l.begin(),l.end());
        for (auto &l:s) {
            for (int i = 1; i < v.size(); ++i)
                l[i] = {l[i].first-l[0].first, l[i].second - l[0].second};
            l[0] = {0,0};
        }
        sort(s.begin(),s.end());
        return s[0];
    }

    int numDistinctIslands2(vector<vector<int>>& g) {
        int cnt = 1;
        set<vector<pair<int,int>>> s;
        for (int i = 0; i < g.size(); ++i) for (int j = 0; j < g[i].size(); ++j) if
(g[i][j] == 1) {
            dfs(i,j,g, ++cnt);
            s.insert(norm(mp[cnt]));
        }

        return s.size();
    }
};
```

written by elastico original link here

Solution 2

By using the same DFS approach as in "Number of Distinct Island", we can calculate the number of "distinct" islands by hashing. The DFS part is trivial, the real challenge is how to calculate (hash) the "Shape" of an island.

Here is my AC C++ implementation using primes as parameters when calculating the hash.

The parameters were carefully chosen as some islands of distinct shapes will collapse into the same hash value If we change some parameter ... :-(

Point it out if you find it fails on certain test cases.

Updated
A better compute_hash function was given below in the same thread.

```
class Solution {
public:
    int numDistinctIslands2(vector<vector<int>>& grid) {
        int m = grid.size();
        if (m == 0) return 0;
        int n = grid[0].size();
        if (n == 0) return 0;

        std::unordered_set<int> signatures;
        for (int r = 0; r < m; ++r) {
            for (int c = 0; c < n; ++c) {
                if (grid[r][c]) {
                    vector<pair<int, int>> pts;
                    visit(grid, m, n, r, c, pts);
                    int hash = compute_hash(pts);
                    signatures.insert(hash);
                }
            }
        }

        return signatures.size();
    }

private:
    int compute_hash(vector<pair<int, int>>& pts) {
        int n = pts.size();
        int hash = 0;
        int xmin = INT_MAX, ymin = INT_MAX, xmax = INT_MIN, ymax = INT_MIN;
        for (int i = 0; i < n; ++i) {
            const auto& pt1 = pts[i];
            xmin = std::min(xmin, pt1.first);
            ymin = std::min(ymin, pt1.second);
            xmax = std::max(xmax, pt1.first);
            ymax = std::max(ymax, pt1.second);
            for (int j = i + 1; j < n; ++j) {
                const auto& pt2 = pts[j];
                int delta_x = pt1.first - pt2.first;
                int delta_y = pt1.second - pt2.second;
                if (delta_x == 0 || delta_y == 0) {
                    hash += 19 * (delta_x * delta_x + delta_y * delta_y);
                    hash += 19 * (delta_x * delta_x + delta_y * delta_y);
                } else {
                    hash += 31 * (delta_x * delta_x + delta_y * delta_y);
                }

            }
        }


        int delta_x = xmax - xmin, delta_y = ymax - ymin;
        hash += 193 * (delta_x * delta_x + delta_y * delta_y) + 97 * delta_x * delt
a_y;
        return hash;
    }

    int compute_hash_improved(vector<pair<int, int>>& pts) {
        int n = pts.size();
        int hash = 0;
        std::unordered_map<int, int> stats1, stats2;

        for (int i = 0; i < n; ++i) {
            const auto& pt1 = pts[i];
            stats1[pt1.first]++;
            stats2[pt1.second]++;
            for (int j = i + 1; j < n; ++j) {
                const auto& pt2 = pts[j];
                int delta_x = pt1.first - pt2.first;
                int delta_y = pt1.second - pt2.second;
                if (delta_x == 0 || delta_y == 0) {
                    hash += 19 * (delta_x * delta_x + delta_y * delta_y);
                } else {
                    hash += 31 * (delta_x * delta_x + delta_y * delta_y);
                }
            }
        }

        for (auto& kv : stats1) {
            hash += 73 * kv.second * kv.second;
        }
        for (auto& kv : stats2) {
            hash += 73 * kv.second * kv.second;
        }

        int delta_x = stats1.size(), delta_y = stats2.size();
        hash += 193 * (delta_x * delta_x + delta_y * delta_y) + 97 * delta_x * delt
a_y;
        return hash;
    }

    void visit(vector<vector<int>>& grid, int m, int n, int r, int c,
               vector<pair<int, int>>& pts) {
        bool outOfBound = (r < 0 || r >= m || c < 0 || c >= n);
        if (outOfBound || grid[r][c] == 0) return;

        grid[r][c] = 0;
        pts.emplace_back(r, c);
        visit(grid, m, n, r - 1, c, pts);
        visit(grid, m, n, r + 1, c, pts);
        visit(grid, m, n, r + 1, c, pts);
        visit(grid, m, n, r, c - 1, pts);
        visit(grid, m, n, r, c + 1, pts);
    }
};
```

written by bigoffer4all original link here

Solution 3

Find every island by running `dfs()` to every cell and assign each island a unique `ID`.
This also takes care of the "mark visited" operation for us (and that's why the `ID` should begin with 2 instead of 1).
We also want to know the minimum rectangle that wraps the island; that's what the `boarder` variable does.
With `boarder` and `ID`, we can extract the island (marking the island body as 1). It's possible other island might be partially or totally included within current boarder so make sure we ignore them when the `ID` mismatched.
By using the hashable tuple in Python, there's no need to manually do hash code calculation. All we need is a `set()` to allow constant time existence checking.
For each extracted island, try every possible rotation and reflection. Early return in case of existence, otherwise it's an unseen island shape and we want to add it to our `set`.
The answer is the size of this `set`.

```python
class Solution(object):
    def numDistinctIslands2(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        m, n = len(grid), len(grid[0])
        def dfs(i, j, ID, boarder):
            grid[i][j] = ID
            boarder[0] = min(boarder[0], i)
            boarder[1] = max(boarder[1], i)
            boarder[2] = min(boarder[2], j)
            boarder[3] = max(boarder[3], j)
            for k, d in enumerate([(-1, 0), (1, 0), (0, -1), (0, 1)]):
                if 0 <= i+d[0] < m and 0 <= j+d[1] < n and grid[i+d[0]][j+d[1]] ==
1:
                    dfs(i+d[0], j+d[1], ID, boarder)

        def extract(ID, bi, ei, bj, ej):
            return tuple(tuple(map(lambda x: '1' if x == ID else ' ', grid[k][bj:ej
+1])) for k in xrange(bi, ei+1))

        def rotate(t):
            t = t[::-1]
            return tuple(zip(*t))

        def addToSet(t):
            for i in xrange(4):
                t = rotate(t)
                # Left/Right mirror
                lrm = tuple(tuple(r[::-1]) for r in t)
                # Up/Down mirror
                udm = tuple(tuple(t[i]) for i in xrange(len(t)-1, -1, -1))
                if t in self.ans or lrm in self.ans or udm in self.ans:
                    return
            self.ans.add(t)

        ID = 2
        self.ans = set()
        for i in xrange(m):
            for j in xrange(n):
                if grid[i][j] == 1:
                    boarder = [i, i, j, j]
                    dfs(i, j, ID, boarder)
                    t = extract(ID, *boarder)
                    addToSet(t)
                    ID += 1

        return len(self.ans)
```

written by franco3 original link here