305. Number of Islands II (/problems/number-of-islands-ii/)

Dec. 27, 2017 | 2.8K views

A 2d grid map of `m` rows and `n` columns is initially filled with water. We may perform an *addLand* operation which turns the water at position (row, col) into a land. Given a list of positions to operate, **count the number of islands after each *addLand* operation**. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example:

Given `m = 3, n = 3`, `positions = [[0,0], [0,1], [1,2], [2,1]]`.
Initially, the 2d grid `grid` is filled with water. (Assume 0 represents water and 1 represents land).

```
0 0 0
0 0 0
0 0 0
```

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.

```
1 0 0
0 0 0   Number of islands = 1
0 0 0
```

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.

```
1 1 0
0 0 0   Number of islands = 1
0 0 0
```

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.

```
1 1 0
0 0 1   Number of islands = 2
0 0 0
```

Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.

```
1 1 0
0 0 1   Number of islands = 3
0 1 0
```

We return the result as an array: `[1, 1, 2, 3]`

Challenge:

Can you do it in time complexity O(k log mn), where k is the length of the `positions` ?

Approach #1 (Brute force) [Time Limit Exceeded]

Algorithm

Reuse the code for Problem 200: Number of Islands (https://leetcode.com/problems/number-of-islands/description/), for each *addLand* operation, just call the `numIslands` function of Problem 200 to get the number of islands after performing that operation.

C++ Java Copy

```cpp
class Solution {
private:
  void dfs(vector<vector<char>>& grid, int r, int c, vector<vector<bool>>& visited) {
    int nr = grid.size();
    int nc = grid[0].size();

    if (r < 0 || c < 0 || r >= nr || c >= nc || grid[r][c] == '0' || visited[r][c]) return;

    visited[r][c] = true;
    dfs(grid, r - 1, c, visited);
    dfs(grid, r + 1, c, visited);
    dfs(grid, r, c - 1, visited);
    dfs(grid, r, c + 1, visited);
  }

  int numIslands(vector<vector<char>>& grid) {
    int nr = grid.size();
    int nc = grid[0].size();

    vector<vector<bool>> visited (nr, vector<bool>(nc, false));
    int num_islands = 0;
    for (int r = 0; r < nr; ++r) {
      for (int c = 0; c < nc; ++c) {
        if (grid[r][c] == '1' && !visited[r][c]) {
          ++num_islands;
          dfs(grid, r, c, visited);
        }
```

Complexity Analysis

- Time complexity : $O(L \times m \times n)$ where L is the number of operations, m is the number of rows and n is the number of columns.
- Space complexity : $O(m \times n)$ for the `grid` and `visited` 2D arrays.

Approach #2: (Ad hoc) [Accepted]

Algorithm

Use a `HashMap` to map index of a land to its island_ID (starting from 0). For each *addLand* operation at position (row, col), check if its adjacent neighbors are in the `HashMap` or not and put the `island_ID` of identified neighbors into a `set` (where each element is unique):

- if the `set` is empty, then the new land at position (row, col) forms a new island (monotonically increasing island_ID by 1);
- if the `set` contains only one island_ID, then the new land belongs to an existing island and island_ID remains unchanged;
- if the `set` contains more than one island_ID, then the new land bridges these separate islands into one island, we need to iterate through the `HashMap` to update this information (time consuming!) and decrease the number of island appropriately.

C++ Java Copy

```
class Solution {
public:
  vector<int> numIslands2(int m, int n, vector<pair<int, int>>& positions) {
    vector<int> ans;
    unordered_map<int, int> land2id; // land index : island ID
    int num_islands = 0;
    int island_id = 0;
    for (auto pos : positions) {
      int r = pos.first;
      int c = pos.second;
      // check pos's neighbors to see if they are in the existing islands or not
      unordered_set<int> overlap; // how many existing islands overlap with 'pos'
      if (r - 1 >= 0 && land2id.count((r-1) * n + c)) overlap.insert(land2id[(r-1) * n + c]);
      if (r + 1 < m && land2id.count((r+1) * n + c)) overlap.insert(land2id[(r+1) * n + c]);
      if (c - 1 >= 0 && land2id.count(r * n + c - 1)) overlap.insert(land2id[r * n + c - 1]);
      if (c + 1 < n && land2id.count(r * n + c + 1)) overlap.insert(land2id[r * n + c + 1]);

      if (overlap.empty()) { // no overlap
        ++num_islands;
        land2id[r * n + c] = island_id++; // new island
      } else if (overlap.size() == 1) { // one overlap, just append
        auto it = overlap.begin();
        land2id[r * n + c] = *it;
      } else { // more than 1 overlaps, merge
        auto it = overlap.begin();
        int root_id = *it;
        for (auto& kv : land2id) { // update island id
```

Complexity Analysis

- Time complexity : $O(L^2)$, for each operation, we have to traverse the entire HashMap to update island id and the number of operations is L.
- Space complexity : $O(L)$ for the `HashMap` .

P.S. C++ solution was accepted with 1409 ms runtime, but Java solution got an TLE (Time Limit Exceeded).

Approach #3: Union Find (aka Disjoint Set) [Accepted]

Intuition

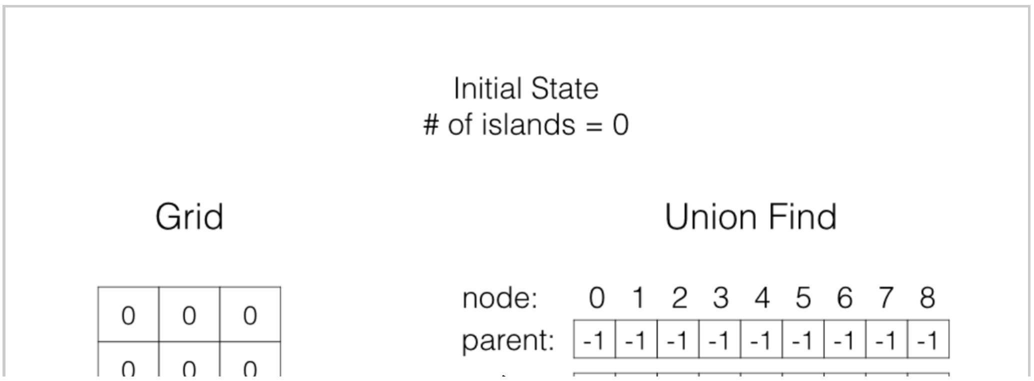
Treat the 2d grid map as an undirected graph (formatted as adjacency matrix) and there is an edge between two horizontally or vertically adjacent nodes of value `1` , then the problem reduces to finding the number of connected components in the graph after each *addLand* operation.

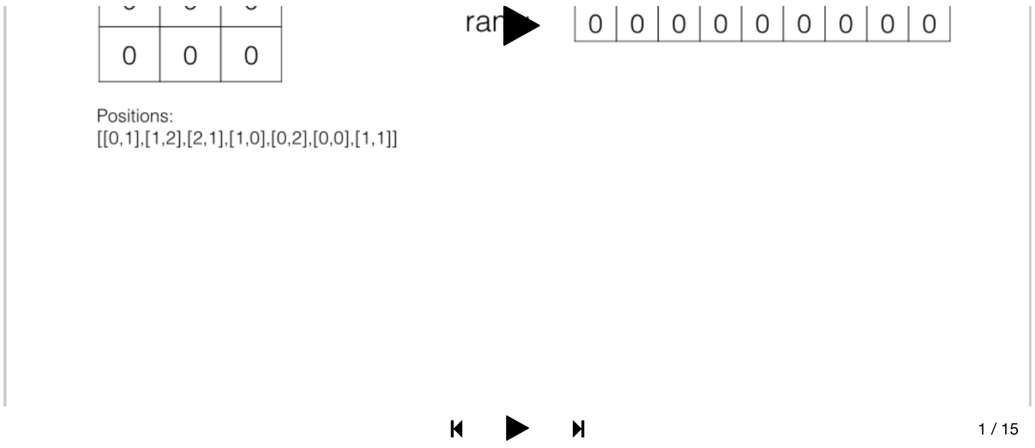
Algorithm

Make use of a `Union Find` data structure of size `m*n` to store all the nodes in the graph and initially each node's parent value is set to `-1` to represent an empty graph. Our goal is to update `Union Find` with lands added by *addLand* operation and union lands belong to the same island.

For each *addLand* operation at position (row, col), union it with its adjacent neighbors if they belongs to some islands, if none of its neighbors belong to any islands, then initialize the new position as a new island (set parent value to itself) within `Union Find` .

For detailed description of `Union Find` (implemented with path compression and union by rank), you can refer to this article (https://leetcode.com/articles/redundant-connection/).





C++ Java Copy

```
class UnionFind {
public:
  UnionFind(int N) {
    count = 0;
    for (int i = 0; i < N; ++i) {
      parent.push_back(-1);
      rank.push_back(0);
    }
  }

  bool isValid(int i) const {
    return parent[i] >= 0;
  }

  void setParent(int i) {
    parent[i] = i;
    ++count;
  }

  int find(int i) { // path compression
    if (parent[i] != i) parent[i] = find(parent[i]);
    return parent[i];
  }

  void Union(int x, int y) { // union with rank
    int rootx = find(x);
    int rooty = find(y);
```

Complexity Analysis

- Time complexity : $O(m \times n + L)$ where L is the number of operations, m is the number of rows and n is the number of columns. it takes $O(m \times n)$ to initialize UnionFind, and $O(L)$ to process positions. Note that Union operation takes essentially constant time[1] when UnionFind is implemented with both path compression and union by rank.
- Space complexity : $O(m \times n)$ as required by UnionFind data structure.

Analysis written by: @imsure (https://leetcode.com/imsure).

Footnotes

1. https://en.wikipedia.org/wiki/Disjoint-set_data_structure (https://en.wikipedia.org/wiki/Disjoint-set_data_structure) ↩

Rate this article:
(/ratings/107/359/?return=/articles/number-of-islands-ii/) (/ratings/107/359/?return=/articles/num

Previous (/articles/cracking-the-safe/) Next (/articles/powx-n/)

Join the conversation

Login to Reply

K

kimjianzsu commented last month

(https://discuss.leetcode.com/user/kimjianzsu)

Create parent[mn+1] and reserve parent[0] to indicate whether a cell is water.
Since the init value of new int[m * n + 1] is 0, you can avoid the array initialization of complexity O(mn).
you can achieve O(klog(mn)).

View original thread (https://discuss.leetcode.com/topic/115192)