Union Find/Disjoint Set

Yifan, Mar 2021

Definition

A *disjoint-set data structure* is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

A *union-find algorithm* is an algorithm that performs two useful operations on such a data structure.

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

Time Complexity/Space Complexity

1. **Path compression**
2. **Union by size/rank**

If you link the sets together arbitrarily instead of using union-by-rank or union-by-size, then path compression alone will achieve *O(m log* n)* time for any sequence of *n* unions and *m* finds (with m > n). That makes the ***amortized* cost of a find operation *O(log* n)* -- *iterated logarithm function***

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf

Template

```python
class DSU1(object):
    # just path compression
    def __init__(self, n):
        self.parents = list(range(n))

    def find(self, x):
        if self.parents[x] != x: # if x is not root
            self.parents[x] = self.find(self.parents[x]) #
        return self.parents[x]

    def union(self, x, y):
        self.parents[self.find(x)] = self.find(y)

    def isConnected(self, x, y):
        return self.find(x) == self.find(y)
```

```python
class DSU2(object):
    # path compression + union by size
    def __init__(self, n):
        self.parents = list(range(n))
        self.size = [1] * n

    def find(self, x):
        if self.parents[x] != x: # if x is not root
            self.parents[x] = self.find(self.parents[x]) # r
        return self.parents[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px != py:
            if self.size[px] < self.size[py]:
                px, py = py, px # px's size is always bigger
            self.parent[py] = px
            self.size[px] += self.size[py]
            self.size[py] = self.size[px]
```

```python
class DSU3(object):
    # path compression + union by rank
    def __init__(self, n):
        self.parents = list(range(n))
        self.rank = [1] * n

    def find(self, x):
        if self.parents[x] != x: # if x is not root
            self.parents[x] = self.find(self.parents[x])
        return self.parents[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px != py:
            if self.rank[px] < self.rank[py]:
                self.parents[px] = py
            elif self.rank[py] < self.rank[px]:
                self.parents[py] = px
            else:
                self.parents[px] = py
                self.rank[py] += 1
```

1722. Minimize Hamming Distance After Swap Operations

Input: source = [1,2,3,4], target = [2,1,4,5], allowedSwaps = [[0,1],[2,3]]

Output: 1

Explanation: source can be transformed the following way:

- Swap indices 0 and 1: source = [2,1,3,4]

- Swap indices 2 and 3: source = [2,1,4,3]

The Hamming distance of source and target is 1 as they differ in 1 position: index 3.

```python
class Solution:
    def minimumHammingDistance(self, source: List[int], target: List[int], allowedSwaps: List[List[int]]) -> int:
        # step 1: union all swaps
        n = len(target)
        uf = DSU(n)
        for u, v in allowedSwaps:
            uf.union(u, v)

        # Step 2: create group of lists based on union results
        d1, d2 = defaultdict(list), defaultdict(list)
        for i in range(n):
            d1[uf.find(i)].append(source[i])
            d2[uf.find(i)].append(target[i])

        # Step 3: define a count function to count overlapped items for single group
        def common_count(l1, l2):
            cnt = 0
            c1, c2 = Counter(l1), Counter(l2)
            for k1 in c1:
                if k1 in c2:
                    cnt += min(c1[k1], c2[k1])
            return cnt

        # Step 4: loop dict and accumulate overlapped items for all groups
        res = 0
        for k in d1.keys():
            l1, l2 = d1[k], d2[k]
            res += common_count(l1, l2)

        # Step 5: return unmatched items
        return n - res
```

684. Redundant Connection

undirected edge connecting nodes

union find cannot easily find cycle in directed graph, for example:

Redundant Connection II

```python
class Solution:
    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        node_set = set()
        for u, v in edges:
            node_set.add(u)
            node_set.add(v)
        n = len(node_set)
        dsu = DSU(n + 1)
        for u, v in edges:
            if not dsu.isConnect(u, v):
                dsu.union(u, v)
            else:
                return [u, v]
```

1135. Connecting Cities With Minimum Cost

Return the minimum cost so that for every pair of cities, there exists a path of connections (possibly of length 1) that connects those two cities together. The cost is the sum of the connection costs used. If the task is impossible, return -1.

```python
class Solution:
    def minimumCost(self, N: int, connections: List[List[int]]) -> int:
        # MST
        dsu = DSU(N + 1)
        # union by weight of edge
        cost, cnt = 0, N
        for u, v, w in sorted(connections, key=lambda x:x[2]):
            if not dsu.isConnect(u, v):
                dsu.union(u, v)
                cost += w
                cnt -= 1
            if cnt == 1: # MST is bulit! we have N - 1 edges/unions
                return cost
        return -1
```

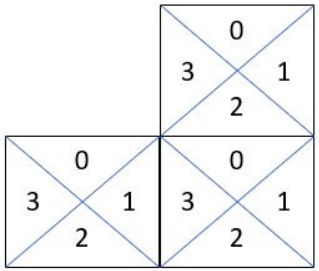
1489. Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree

Find *all the critical and pseudo-critical edges in the given graph's minimum spanning tree (MST)*. An MST edge whose deletion from the graph would cause the MST weight to increase is called a *critical edge*. On the other hand, a pseudo-critical edge is that which can appear in some MSTs but not all.

```python
class Solution:
    def findCriticalAndPseudoCriticalEdges(self, n: int, edges: List[List[int]]) -> List[List[int]]:
        edges = [(u, v, w, i) for i, (u, v, w) in enumerate(edges)]
        edges.sort(key=lambda x: x[2]) # O(ElogE)

        critial, pseudo = [], []
        for iu, iv, iw, i in edges:
            dsu1, dsu2 = DSU(n), DSU(n)
            dsu1.union(iu, iv) # use this edge in dsu1 but not dsu2
            s1, s2 = iw, 0
            for u, v, w, j in edges:
                if i == j:
                    continue
                if not dsu1.isConnect(u, v):
                    dsu1.union(u, v)
                    s1 += w
                if not dsu2.isConnect(u, v):
                    dsu2.union(u, v)
                    s2 += w
            if s1 == s2:
                pseudo.append(i)
            elif s1 < s2 or not dsu2.isConnect(iu, iv): # corner case: dsu2 still needs (iu, iv)
                critial.append(i)
            # actually there is a third case, edge is not in MST
        return [critial, pseudo]
```

959. Regions Cut By Slashes



In a N x N `grid` composed of 1 x 1 squares, each 1 x 1 square consists of a `/`, `\`, or blank space. These characters divide the square into contiguous regions.

(Note that backslash characters are escaped, so a `\` is represented as `"\\"`.)

Return the number of regions.

Input:

```
[
  " /",
  "  "
]
```

Output: 1

```python
class Solution_uf:
    def regionsBySlashes(self, grid: List[str]) -> int:
        n = len(grid)
        dsu = DSU(4 * n * n)
        for i in range(n):
            for j in range(n):
                root = 4 * (n * i + j)
                if grid[i][j] in "/ ":
                    dsu.union(root, root + 1)
                    dsu.union(root + 2, root + 3)
                if grid[i][j] in "\\ ":
                    dsu.union(root, root + 2)
                    dsu.union(root + 1, root + 3)
                if i > 0:
                    dsu.union(root, root + 3 - 4 * n)
                if i < n - 1:
                    dsu.union(root + 3, root + 4 * n)
                if j > 0:
                    dsu.union(root + 1, root - 2)
                if j < n - 1:
                    dsu.union(root + 2, root + 5)
        return sum(dsu.find(x) == x for x in range(n * n * 4))
```

721. Accounts Merge

Input: accounts = [["John","johnsmith@mail.com","john_newyork@mail.com"],["John","johnsmith@mail.com","john00@mail.com"],["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]]

Output: [["John","john00@mail.com","john_newyork@mail.com","johnsmith@mail.com"],["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]]

Two accounts definitely belong to the same person if there is some common email to both accounts.

```python
class Solution:
    def accountsMerge(self, accounts: List[List[str]]) -> List[List[str]]:
        # Union find
        # email -> name
        # email -> index
        # index union index
        # combine all the email with the same index
        # add name for every group
        ans = []
        email2name = {}
        email2index = {}
        i = 0
        uf = DSU(10001)
        for account in accounts:
            name = account[0]
            for email in account[1:]:
                if email not in email2index:
                    email2name[email] = name
                    email2index[email] = i
                    i += 1
                uf.union(email2index[account[1]], email2index[email])
        # print(uf.parents[:len(email2index)])
        # print(email2index)
        emailGroup = collections.defaultdict(list)
        for email in email2index:
            emailGroup[uf.find(email2index[email])].append(email)
        # print(emailGroup)

        return [[email2name[emails[0]]] + sorted(emails) for emails in emailGroup.values()]
```

200. Number of Islands

Given an `m x n` 2D binary grid `grid` which represents a map of `'1'`s (land) and `'0'`s (water), return *the number of islands*.

```python
class Solution(object):
    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        if not grid:
            return 0

        m, n = len(grid), len(grid[0])
        dirs = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        cnt = sum(grid[i][j]=='1' for i in range(m) for j in range(n))
        dsu = DSU(m * n, cnt)

        for i in range(m):
            for j in range(n):
                if grid[i][j] == '1':
                    for dx, dy in dirs:
                        x, y = i + dx, j + dy
                        if 0 <= x < m and 0 <= y < n and grid[x][y] == '1':
                            dsu.union(x * n + y, i * n + j)
        return dsu.count
```

305. Number of Islands II

We may perform an add land operation which turns the water at position into a land. You are given an array positions where positions[i] = [ri, ci] is the position (ri, ci) at which we should operate the ith operation.

Return *an array of integers* answer *where* answer[i] *is the number of islands after turning the cell* (ri, ci) *into a land.*

Input: m = 3, n = 3, positions = [[0,0],[0,1],[1,2],[2,1]]

Output: [1,1,2,3]

```python
class Solution:
    def numIslands2(self, m: int, n: int, positions: List[List[int]]) -> List[int]:
        dirs = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        cnt = len(positions)
        dsu = DSU(m * n, cnt)

        res = []
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        for x, y in positions:
            index = x * n + y # flatten a 2d coordinate into a 1d value
            dsu.setParent(index)
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < m and 0 <= ny < n and nx * n + ny in dsu.parents:
                    dsu.Union(index, nx * n + ny)
            res.append(dsu.count)
        return res
```

765. Couples Holding Hands

N couples sit in 2N seats arranged in a row and want to hold hands. We want to know the minimum number of swaps so that every couple is sitting side by side. A *swap* consists of choosing any two people, then they stand up and switch seats.

The people and seats are represented by an integer from `0` to `2N-1`, the couples are numbered in order, the first couple being `(0, 1)`, the second couple being `(2, 3)`, and so on with the last couple being `(2N-2, 2N-1)`.

```
Input: row = [0, 2, 1, 3]

Output: 1

  Explanation: We only need to swap
  the second (row[1]) and third
  (row[2]) person.
```

```python
class Solution:
    def minSwapsCouples(self, row: List[int]) -> int:
        # min number of swaps = N - number of connected components
        n = len(row) // 2
        dsu = DSU(n)
        for i in range(n):
            # pick 2 neighbor people
            person1, person2 = row[2 * i], row[2 * i + 1]
            # union if each person is from different couples
            # if u, v are couples already, u//2 = v//2 (couple id)
            couple1, couple2 = person1 // 2, person2 // 2
            dsu.union(couple1, couple2)
        return n - dsu.cnt
```

1202. Smallest String With Swaps

You are given a string `s`, and an array of pairs of indices in the string `pairs` where `pairs[i] = [a, b]` indicates 2 indices(0-indexed) of the string.

You can swap the characters at any pair of indices in the given `pairs` any number of times.

Return the lexicographically smallest string that `s` can be changed to after using the swaps.

```python
class Solution:
    def smallestStringWithSwaps(self, s: str, pairs: List[List[int]]) -> str:
        n = len(s)
        dsu = DSU(n)
        res = []
        d = defaultdict(list)
        for u, v in pairs:
            dsu.union(u, v)
        for i in range(n):
            d[dsu.find(i)].append(s[i])
        for comp_id in d.keys():
            d[comp_id].sort(reverse=True)

        # find the lowest possible character that can be exchanged
        for i in range(n):
            res.append(d[dsu.find(i)].pop()) # each char use once
        return ''.join(res)
```