January 3, 2023 ▪ Data Structure / Graph

Accounts Merge – DSU: G-50

Problem Statement: Given a list of accounts where each element account [i] is a list of strings, where the first element account [i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the

rest of the elements are emails in sorted order.

Note: Accounts themselves can be returned in any order.

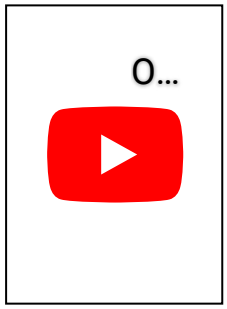
Pre-requisite: Disjoint Set data structure

Examples:

```
Example 1:
Input: N = 4
accounts [ ] =
[["John","johnsmith@mail.com","john_
["John","johnsmith@mail.com","john0(
["Mary","mary@mail.com"],
["John","johnnybravo@mail.com"]]

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

Explanation: The first and the
second John are the same person
as they have a common email. But
the third Mary and fourth John
are not the same as they do not
have any common email.  The
result can be in any order but
the emails must be in sorted
order. The following is also a
valid result:
[['Mary', 'mary@mail.com'],
['John',
'johnnybravo@mail.com'],
['John', 'john00@mail.com' ,
'john_newyork@mail.com',
'johnsmith@mail.com' ]]
```



Recent Posts

```
Example 2:
Input: N = 6
accounts [ ] =
[["John","j1@com","j2@com","j3@com"]
["John","j4@com"],
["Raj",”r1@com”, “r2@com”],
["John","j1@com","j5@com"],
["Raj",”r2@com”, “r3@com”],
["Mary","m1@com"]]

Output:
[["John","j1@com","j2@com","j3@com",
["John","j4@com"],
["Raj",”r1@com”, “r2@com”,
“r3@com”],
["Mary","m1@com"]]

Explanation: The first and the
fourth John are the same person
here as they have a common
email. And the third and the
fifth Raj are also the same
person. So, the same accounts
are merged.
```

Solution

Disclaimer: *Don’t jump directly to the solution, try it out yourself first.*

Problem link.

Solution:

Let’s quickly understand the question before moving on to the solution part. According to the question, we can only merge two accounts with the same name only if the

accounts contain at least one common email. After merging the accounts accordingly, we should return the answer where for each account the emails must be in the sorted order. But the order of the accounts does not matter. In order to solve this problem we are going to use the Disjoint Set data structure. Now, let's discuss the approach using the following example:

```
Given: N = 6
accounts [ ] =
[["John","j1@com","j2@com","j3@com"]
["John","j4@com"],
["Raj",”r1@com”, “r2@com”],
["John","j1@com","j5@com"],
["Raj",”r2@com”, “r3@com”],
["Mary","m1@com"]]
```

First, we will try to iterate over every single email and add them with their respective indices(i.e. Index of the accounts the email belongs to) in a map data structure. While doing this, when we will reach out to "j1@com" in the fourth account, we will find that it is already mapped with index 0. This incident means that we are currently in an account that can be merged. So, we will perform the union operation between the current index i.e. 3, and index 0(As in this case, we are following 0-based indexing). It will mean that the ultimate parent of index 3 is index 0. Similarly, this incident will repeat

in the case of the third and fifth Raj. So we will perform the union of index 2 and 4.

After completing the above process, the situation will be like the following:

Now, it's time to merge the emails. So, we will iterate over each email and will add them to the ultimate parent of the current account's index. Like, while adding the emails of account 4, we will add them to index 2 as the ultimate parent of 4 is index 2.

Finally, we will sort the emails for each account individually to get our answers in the format specified in the question.

Approach:

Note:

- Here we will perform the disjoint set operations on the indices of the accounts considering them as the nodes.

- As in each account, the first element is the name, we will start iterating from the second element in each account to visit only the emails sequentially.

The algorithm steps are the following:

1. First, we will ***create a map data structure.*** Then we will store each email with the respective index of the account(the email belongs to) in that map data structure.
2. While doing so, if we encounter an email again(i.e. If any index is previously assigned for the email), we will perform union(***either unionBySize() or unionByRank()***) of the current index and the previously assigned index.
3. After completing step 2, now it's time to ***merge the accounts***. For merging, we will iterate over all the emails individually and find the ultimate parent(***using the findUPar() method***) of the assigned index of every email. Then we will add the email of the current account to the index(account index) that is the ultimate parent. Thus the accounts will be merged.
4. Finally, we will ***sort the emails for every account separately*** and store the final results in the answer array accordingly.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

C++ Code

```
#include <bits/stdc++.h>
using namespace std;
//User function Template for C++
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++)
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUP
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
```

```
            if (size[ulp_u] < size[ulp_v
                parent[ulp_u] = ulp_v;
                size[ulp_v] += size[ulp_
            }
            else {
                parent[ulp_v] = ulp_u;
                size[ulp_u] += size[ulp_
            }
        }
    };

    class Solution {
    public:
        vector<vector<string>> accountsM
            int n = details.size();
            DisjointSet ds(n);
            sort(details.begin(), detail
            unordered_map<string, int> m
            for (int i = 0; i < n; i++)
                for (int j = 1; j < deta
                    string mail = detail
                    if (mapMailNode.find
                        mapMailNode[mail
                    }
                    else {
                        ds.unionBySize(i
                    }
                }
            }

            vector<string> mergedMail[n]
            for (auto it : mapMailNode)
                string mail = it.first;
                int node = ds.findUPar(i
                mergedMail[node].push_ba
            }

            vector<vector<string>> ans;

            for (int i = 0; i < n; i++)
                if (mergedMail[i].size()
                sort(mergedMail[i].begin
                vector<string> temp;
                temp.push_back(details[i
                for (auto it : mergedMai
                    temp.push_back(it);
```

```
                }
                ans.push_back(temp);
            }
            sort(ans.begin(), ans.end())
            return ans;
        }
    };


    int main() {

        vector<vector<string>> accounts
            {"John", "j4@com"},
            {"Raj", "r1@com", "r2@com"},
            {"John", "j1@com", "j5@com"}
            {"Raj", "r2@com", "r3@com"},
            {"Mary", "m1@com"}
        };


        Solution obj;
        vector<vector<string>> ans = obj
        for (auto acc : ans) {
            cout << acc[0] << ":";
            int size = acc.size();
            for (int i = 1; i < size; i+
                cout << acc[i] << " ";
            }
            cout << endl;
        }
        return 0;
    }
```

Output:

John:j1@com j2@com j3@com j5@com

John:j4@com

Mary:m1@com

Raj:r1@com r2@com r3@com

Time Complexity: O(N+E) + O(E*4α) + O(N*

(ElogE + E)) where N = no. of indices or nodes

and E = no. of emails. The first term is for visiting all the emails. The second term is for merging the accounts. And the third term is for sorting the emails and storing them in the answer array.

Space Complexity: O(N)+ O(N) +O(2N) ~ O(N) where N = no. of nodes/indices. The first and second space is for the 'mergedMail' and the 'ans' array. The last term is for the parent and size array used inside the Disjoint set data structure.

Java Code

```
import java.io.*;
import java.util.*;



//User function Template for Java
class DisjointSet {
    List<Integer> rank = new ArrayLi
    List<Integer> parent = new Array
    List<Integer> size = new ArrayLi
    public DisjointSet(int n) {
        for (int i = 0; i <= n; i++)
            rank.add(0);
            parent.add(i);
            size.add(1);
        }
    }

    public int findUPar(int node) {
        if (node == parent.get(node)
            return node;
        }
        int ulp = findUPar(parent.ge
        parent.set(node, ulp);
        return parent.get(node);
    }
```

```
        public void unionByRank(int u, i
            int ulp_u = findUPar(u);
            int ulp_v = findUPar(v);
            if (ulp_u == ulp_v) return;
            if (rank.get(ulp_u) < rank.g
                parent.set(ulp_u, ulp_v)
            } else if (rank.get(ulp_v) <
                parent.set(ulp_v, ulp_u)
            } else {
                parent.set(ulp_v, ulp_u)
                int rankU = rank.get(ulp
                rank.set(ulp_u, rankU +
            }
        }

        public void unionBySize(int u, i
            int ulp_u = findUPar(u);
            int ulp_v = findUPar(v);
            if (ulp_u == ulp_v) return;
            if (size.get(ulp_u) < size.g
                parent.set(ulp_u, ulp_v)
                size.set(ulp_v, size.get
            } else {
                parent.set(ulp_v, ulp_u)
                size.set(ulp_u, size.get
            }
        }
    }
    class Solution {
        static List<List<String>> accoun
            int n = details.size();
            DisjointSet ds = new Disjoin
            HashMap<String, Integer> map

            for (int i = 0; i < n; i++)
                for (int j = 1; j < deta
                    String mail = detail
                    if (mapMailNode.cont
                        mapMailNode.put(
                    } else {
                        ds.unionBySize(i
                    }
                }
            }

            ArrayList<String>[] mergedMa
```

```
            for (int i = 0; i < n; i++)
            for (Map.Entry<String, Integ
                String mail = it.getKey(
                int node = ds.findUPar(i
                mergedMail[node].add(mai
            }

            List<List<String>> ans = new

            for (int i = 0; i < n; i++)
                if (mergedMail[i].size()
                Collections.sort(mergedM
                List<String> temp = new
                temp.add(details.get(i).
                for (String it : mergedM
                    temp.add(it);
                }
                ans.add(temp);
            }
            return ans;

        }
    }

    class Main {
        public static void main (String[
            List<List<String>> accounts
                {
                    add(new ArrayList<St
                    add(new ArrayList<St
                    add(new ArrayList<St
                    add(new ArrayList<St
                    add(new ArrayList<St
                    add(new ArrayList<St

                }
            };

            Solution obj = new Solution(
            List<List<String>> ans = obj

            int n = ans.size();
            for (int i = 0; i < n; i++)
                System.out.print(ans.get
                int size = ans.get(i).si
                for (int j = 1; j < size
```

```
                    System.out.print(ans
                }

                System.out.println("");
            }

        }
    }
```

Output:

John:j1@com j2@com j3@com j5@com
John:j4@com
Mary:m1@com
Raj:r1@com r2@com r3@com

Time Complexity: O(N+E) + O(E*4α) + O(N*(ElogE + E)) where N = no. of indices or nodes and E = no. of emails. The first term is for visiting all the emails. The second term is for merging the accounts. And the third term is for sorting the emails and storing them in the answer array.

Space Complexity: O(N)+ O(N) +O(2N) ~ O(N) where N = no. of nodes/indices. The first and second space is for the 'mergedMail' and the 'ans' array. The last term is for the parent and size array used inside the Disjoint set data structure.

> Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you

> want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

« Previous Post
Number of Operations to Make Network Connected – DSU: G-49.

Next Post »
Number of Provinces – Disjoint Set: G-48

Load Comments