Accounts Merge

Given a list `accounts`, each element `accounts[i]` is a list of strings, where the first element `accounts[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 email that is common 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**. The accounts themselves can be returned in any order.

Example 1:

```
Input:
accounts = [["John", "johnsmith@mail.com", "john00@mail.com"], ["John", "johnnybravo@mail.com"], ["John", "johnsmith@mail.com", "john_newyork@mail.com"], ["Mary", "mary@mail.com"]]
Output: [["John", 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com'], ["John", "johnnybravo@mail.com"], ["Mary", "mary@mail.com"]]
Explanation:
The first and third John's are the same person as they have the common email "johnsmith@mail.com".
The second John and Mary are different people as none of their email addresses are used by other accounts.
We could return these lists in any order, for example the answer [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],
['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']] would still be accepted.
```

Note:

- The length of `accounts` will be in the range `[1, 1000]`.
- The length of `accounts[i]` will be in the range `[1, 10]`.
- The length of `accounts[i][j]` will be in the range `[1, 30]`.

Solution 1

We give each account an ID, based on the index of it within the list of accounts.

```
[
["John", "johnsmith@mail.com", "john00@mail.com"], # Account 0
["John", "johnnybravo@mail.com"], # Account 1
["John", "johnsmith@mail.com", "john_newyork@mail.com"],  # Account 2
["Mary", "mary@mail.com"] # Account 3
]
```

Next, build an `emails_accounts_map` that maps an email to a list of accounts, which can be used to track which email is linked to which account. This is essentially our graph.

```
# emails_accounts_map of email to account ID
{
  "johnsmith@mail.com": [0, 2],
  "john00@mail.com": [0],
  "johnnybravo@mail.com": [1],
  "john_newyork@mail.com": [2],
  "mary@mail.com": [3]
}
```

Next we do a DFS on each account in accounts list and look up `emails_accounts_map` to tell us which accounts are linked to that particular account via common emails. This will make sure we visit each account only once. This is a recursive process and we should collect all the emails that we encounter along the way.

Lastly, sort the collected emails and add it to final results, `res` along with the name.

- *Yangshun*

```python
class Solution(object):
    def accountsMerge(self, accounts):
        from collections import defaultdict
        visited_accounts = [False] * len(accounts)
        emails_accounts_map = defaultdict(list)
        res = []
        # Build up the graph.
        for i, account in enumerate(accounts):
            for j in range(1, len(account)):
                email = account[j]
                emails_accounts_map[email].append(i)
        # DFS code for traversing accounts.
        def dfs(i, emails):
            if visited_accounts[i]:
                return
            visited_accounts[i] = True
            for j in range(1, len(accounts[i])):
                email = accounts[i][j]
                emails.add(email)
                for neighbor in emails_accounts_map[email]:
                    dfs(neighbor, emails)
        # Perform DFS for accounts and add to results.
        for i, account in enumerate(accounts):
            if visited_accounts[i]:
                continue
            name, emails = account[0], set()
            dfs(i, emails)
            res.append([name] + sorted(emails))
        return res
```

written by yangshun original link here

Solution 2

I have tried my best to make my code clean. Hope the basic idea below may help you. Happy coding!

Basicly, this is a graph problem. Notice that each account[i] tells us some edges. What we have to do is as follows:

1. Use these edges to build some components. Common email addresses are like the intersections that connect each single component for each account.
2. Because each component represents a merged account, do DFS search for each components and add into a list. Before add the name into this list, sort the emails. Then add name string into it.

Please check the clean version posted by @zhangchunli below. Thanks for updating.

```
class Solution {
    public List<List<String>> accountsMerge(List<List<String>> accounts) {
        List<List<String>> res = new ArrayList<>();
        Map<String, Node> map = new HashMap<>();    // <Email, email node>

        // Build the graph;
        for (List<String> list : accounts) {
            for (int j = 1; j < list.size(); j++) {
                String email = list.get(j);

                if (!map.containsKey(email)) {
                    Node node = new Node(email, list.get(0));
                    map.put(email, node);
                }

                if (j == 1) continue;
                //Connect the current email node to the previous email node;
                map.get(list.get(j - 1)).neighbors.add(map.get(email));
                map.get(email).neighbors.add(map.get(list.get(j - 1)));
            }
        }

        // DFS search for each components(each account);
        Set<String> visited = new HashSet<>();
        for (String s : map.keySet()) {
            if (visited.add(s)) {
                List<String> list = new LinkedList<>();
                list.add(s);
                dfs(map.get(s), visited, list);
                Collections.sort(list);
                list.add(0, map.get(s).username);
                res.add(list);
            }
        }
        return res;
    }

    public void dfs(Node root, Set<String> visited, List<String> list) {
        for (Node node : root.neighbors) {
            if (visited.add(node.val)) {
```

```
            if (visited.add(node.val)) {
                list.add(node.val);
                dfs(node, visited, list);
            }
        }
    }

    class Node {
        String val;         // Email address;
        String username;    // Username;
        List<Node> neighbors;
        Node(String val, String username) {
            this.val = val;
            this.username = username;
            neighbors = new ArrayList<>();
        }
    }
}
```

written by FLAGbigoffer original link here

Solution 3

1. The key task here is to `connect` those `emails`, and this is a perfect use case for union find.
2. to group these emails, each group need to have a `representative`, or `parent`.
3. At the beginning, set each email as its own representative.
4. Emails in each account naturally belong to a same group, and should be joined by assigning to the same parent (let's use the parent of first email in that list);

Simple Example:

```
a b c // now b, c have parent a
d e f // now d, e have parent d
g a d // now abc, def all merged to group g

parents populated after parsing 1st account: a b c
a->a
b->a
c->a

parents populated after parsing 2nd account: d e f
d->d
e->d
f->d

parents populated after parsing 3rd account: g a d
g->g
a->g
d->g
```

Java

```java
class Solution {
    public List<List<String>> accountsMerge(List<List<String>> acts) {
        Map<String, String> owner = new HashMap<>();
        Map<String, String> parents = new HashMap<>();
        Map<String, TreeSet<String>> unions = new HashMap<>();
        for (List<String> a : acts) {
            for (int i = 1; i < a.size(); i++) {
                parents.put(a.get(i), a.get(i));
                owner.put(a.get(i), a.get(0));
            }
        }
        for (List<String> a : acts) {
            String p = find(a.get(1), parents);
            for (int i = 2; i < a.size(); i++)
                parents.put(find(a.get(i), parents), p);
        }
        for (List<String> a : acts) {
            for (int i = 1; i < a.size(); i++) {
                String p = find(a.get(i), parents);
                if (!unions.containsKey(p)) unions.put(p, new TreeSet<>());
                unions.get(p).add(a.get(i));
            }
        }
        List<List<String>> res = new ArrayList<>();
        for (String p : unions.keySet()) {
            List<String> emails = new ArrayList(unions.get(p));
            emails.add(0, owner.get(p));
            res.add(emails);
        }
        return res;
    }
    private String find(String s, Map<String, String> p) {
        return p.get(s) == s ? s : find(p.get(s), p);
    }
}
```

C++

```cpp
class Solution {
public:
    vector<vector<string>> accountsMerge(vector<vector<string>>& acts) {
        map<string, string> owner;
        map<string, string> parents;
        map<string, set<string>> unions;
        for (int i = 0; i < acts.size(); i++) {
            for (int j = 1; j < acts[i].size(); j++) {
                parents[acts[i][j]] = acts[i][j];
                owner[acts[i][j]] = acts[i][0];
            }
        }
        for (int i = 0; i < acts.size(); i++) {
            string p = find(acts[i][1], parents);
            for (int j = 2; j < acts[i].size(); j++)
                parents[find(acts[i][j], parents)] = p;
        }
        for (int i = 0; i < acts.size(); i++)
            for (int j = 1; j < acts[i].size(); j++)
                unions[find(acts[i][j], parents)].insert(acts[i][j]);

        vector<vector<string>> res;
        for (pair<string, set<string>> p : unions) {
            vector<string> emails(p.second.begin(), p.second.end());
            emails.insert(emails.begin(), owner[p.first]);
            res.push_back(emails);
        }
        return res;
    }
private:
    string find(string s, map<string, string>& p) {
        return p[s] == s ? s : find(p[s], p);
    }
};
```

C++ Lambda

```
class Solution {
public:
    vector<vector<string>> accountsMerge(vector<vector<string>>& acts) {
        map<string, string> owner;
        map<string, string> parents;
        function<string(string)> find = [&](string s) {return parents[s] == s ? s :
find(parents[s]); };
        for (int i = 0; i < acts.size(); i++) {
            for (int j = 1; j < acts[i].size(); j++) {
                parents[acts[i][j]] = acts[i][j];
                owner[acts[i][j]] = acts[i][0];
            }
        }
        for (int i = 0; i < acts.size(); i++) {
            string p = find(acts[i][1]);
            for (int j = 2; j < acts[i].size(); j++) {
                parents[find(acts[i][j])] = p;
            }
        }
        map<string, set<string>> unions;
        for (int i = 0; i < acts.size(); i++) {
            for (int j = 1; j < acts[i].size(); j++) {
                unions[find(acts[i][j])].insert(acts[i][j]);
            }
        }
        vector<vector<string>> merged;
        for (pair<string, set<string>> p : unions) {
            vector<string> emails(p.second.begin(), p.second.end());
            emails.insert(emails.begin(), owner[p.first]);
            merged.push_back(emails);
        }
        return merged;
    }
};
```

written by alexander original link here