<https://massivealgorithms.blogspot.com/2017/05/lintcode-432-find-weak-connected.html>

## **Question**

Find the number Weak Connected Component in the directed graph. Each node in the graph contains a label and a list of its neighbors. (a connected set of a directed graph is a subgraph in which any two vertices are connected by direct edge path.)

**Notice**

Sort the element in the set in increasing order

**Example**

Given graph:

A----------> B C

\ | |

\ | |

\ | |

\ v v

--> D E <- F

Return {A,B,D}, {C,E,F}. Since there are two connected component which are {A,B,D} and {C,E,F}

***Weakly Connected:*** *A directed graph is weakly connected if there is a path between every two vertices in the underlying undirected graph (i.e., the graph formed when the direction of the edges are removed).*

**Attempt 1: 2022-12-17**

**Solution 1:  Union Find (10 min)**

/\*\*

\* Definition for Directed graph.

\* class DirectedGraphNode {

\* int label;

\* ArrayList<DirectedGraphNode> neighbors;

\* DirectedGraphNode(int x) { label = x; neighbors = new ArrayList<DirectedGraphNode>(); }

\* };

\*/

class Solution {

public List<List<Integer>> connectedSet2(ArrayList<DirectedGraphNode> nodes) {

// Initialize parent in Hashmap style

Set<Integer> set = new HashSet<Integer>();

for(DirectedGraphNode node : nodes) {

set.add(node.label);

for(DirectedGraphNode neighbour : node.neighbors) {

set.add(neighbor.label);

}

}

Map<Integer, Integer> parent = new HashMap<Integer, Integer>();

for(int num : set) {

parent.put(num, num);

}

// Find and Union

for(DirectedGraphNode node : nodes) {

for(DirectedGraphNode neighbour : node.neighbors) {

int rootA = find(node.label, parent);

int rootB = find(neighbour.label, parent);

if(rootA != rootB) {

parent.put(rootA, rootB);

}

}

}

// Print result

List<List<Integer>> result = new ArrayList<List<Integer>>();

Map<Integer, List<Integer>> paths = new HashMap<Intger, List<Integer>>();

for(int i : set) {

int root = find(i, parent);

if(!paths.containsKey(root)) {

paths.put(root, new ArrayList<Integer>());

}

paths.get(root).add(i);

}

for(List<Integer> list : paths.values()) {

Collections.sort(list);

result.add(list);

}

return result;

}

private int find(int x, Map<Integer, Integer> parent) {

int y = parent.get(x);

if(x == y) {

return x;

}

return y = find(y, parent);

}

private int find2(int x, Map<Integer, Integer> parent) {

while(x != parent.get(x)) {

parent.get(x, parent.get(x));

x = parent.get(x);

}

return x;

}

}

**Refer to**

<https://massivealgorithms.blogspot.com/2017/05/lintcode-432-find-weak-connected.html>

这题和[Connected Component in Undirected Graph](http://xuezhashuati.blogspot.com/2017/03/lintcode-432-find-weak-connected.html)其实是一模一样的。

唯一的区别是这题是有向图。所以不能用BFS做。用union-find来做就和上面这题一样了。

**这题的UF由于上来不知道点的值的大小，所以不能开一个数组，只能用HashMap来实现。**

把所有的点和对应的root存进HashMap。

最后需要输出结果的时候，把相同root的点放进同一个list。最后把每个list排个序。

<https://aaronice.gitbook.io/lintcode/union_find/find_the_weak_connected_component_in_the_directed_graph>

/\*\*

\* Definition for Directed graph.

\* class DirectedGraphNode {

\* int label;

\* ArrayList<DirectedGraphNode> neighbors;

\* DirectedGraphNode(int x) { label = x; neighbors = new ArrayList<DirectedGraphNode>(); }

\* };

\*/

public class Solution {

class UnionFind{

HashMap<Integer, Integer> father = new HashMap<Integer, Integer>();

UnionFind(HashSet<Integer> hashSet){

for(Integer now : hashSet) {

father.put(now, now);

}

}

int find(int x){

int parent = father.get(x);

while(parent!=father.get(parent)) {

parent = father.get(parent);

}

return parent;

}

int compressed\_find(int x){

int parent = father.get(x);

while(parent!=father.get(parent)) {

parent = father.get(parent);

}

int temp = -1;

int fa = father.get(x);

while(fa!=father.get(fa)) {

temp = father.get(fa);

father.put(fa, parent) ;

fa = temp;

}

return parent;

}

void union(int x, int y){

int fa\_x = find(x);

int fa\_y = find(y);

if(fa\_x != fa\_y)

father.put(fa\_x, fa\_y);

}

}

List<List<Integer> > print(HashSet<Integer> hashSet, UnionFind uf, int n) {

List<List <Integer> > ans = new ArrayList<List<Integer>>();

HashMap<Integer, List <Integer>> hashMap = new HashMap<Integer, List <Integer>>();

for(int i : hashSet){

int fa = uf.find(i);

if(!hashMap.containsKey(fa)) {

hashMap.put(fa, new ArrayList<Integer>() );

}

List <Integer> now = hashMap.get(fa);

now.add(i);

hashMap.put(fa, now);

}

for( List <Integer> now: hashMap.values()) {

Collections.sort(now);

ans.add(now);

}

return ans;

}

public List<List<Integer>> connectedSet2(ArrayList<DirectedGraphNode> nodes){

// Write your code here

HashSet<Integer> hashSet = new HashSet<Integer>();

for(DirectedGraphNode now : nodes){

hashSet.add(now.label);

for(DirectedGraphNode neighbour : now.neighbors) {

hashSet.add(neighbour.label);

}

}

UnionFind uf = new UnionFind(hashSet);

for(DirectedGraphNode now : nodes){

for(DirectedGraphNode neighbour : now.neighbors) {

int fnow = uf.find(now.label);

int fneighbour = uf.find(neighbour.label);

if(fnow!=fneighbour) {

uf.union(now.label, neighbour.label);

}

}

}

return print(hashSet , uf, nodes.size());

}

}