检测graph是否有环：BFS, DFS

DFS – 207. Course Schedule

* Directed 需要从ArrayList[i] 的0~n-1的每个i，检查是否hasCycle
* Undirected 从任意一点开始，检查一次即可，如果没有环，会经过所有的点，之后所有states[i]均为2。所以在检查之后，需要遍历一次states, 如果有的为0，代表对应的点没有连接。

BFS – Graph Valid Tree

* Directed 从indegree为0的点开始，依次一层层加入queue，并把相连的node的degree--
* Undirected 从任意一点开始，依次加入queue，其余相同

**Union Find相关知识：**

**一个很好的博客：**<http://blog.csdn.net/dm_vincent/article/details/7655764>

解决动态连通性这一类问题，对于联通的节点，可以认为它们属于一个组，不连通的属于不同的组。

输入为pair, eg: [0, 1]代表这两个数字联通，属于一个组。

可能的操作：

* 初始化 UF(int N), 初始化为不同的组，0~n-1
* 查询节点所在的组 int find(int p)
* 判断两个节点是否属于同一个组 boolean connected(int p, int q)
* 连接两个节点，使之属于同一个组 void union(int p, int q) 建立p,q间连接
* 获取组的数目 int count()

**Quick-Find算法：**

尽量加快find的速度，但是添加新路径时会涉及到组号的修改，需要对整个数组遍历，需要O(n)

public class UF {

private int[] id; *// id[i]: group id of i.*

private int count; *// number of components*

public UF(int N) { *// Initialize component id array.*

count = N;

id = new int[N];

for (int i = 0; i < N; i++) {

id[i] = i;

}

}

public int count() { return count; }

public boolean connected(int p, int q) { return find(p) == find(q); }

public int find(int p) { return id[p]; }

public void union(int p, int q) {

int pID = find(p); // GET groupid of p and q

int qID = find(q);

if (pID == qID) return;

for (int i = 0; i < id.length; i++) { // change groupid, make them belong to same group

if (id[i] == pID) id[i] = qID;

}

count--;

}

}

**Quick-Union算法：**

同一组的数据保存在一棵树里，union时将一棵树变为另一棵树的子树。

如何在数组中保存parent – child关系？利用id[], id[kid] = parent, 并且id[root] = root，判断是否为root。

需要使union后的树尽量平衡，所以根据树的size，把小的树作为大树的子树。

int[] id = new int[n]; *// id[i]: i所在的tree的root的id*

int[] size = new int[n]; // init: 1， i所在的tree的size, 只有root所对应的size确定正确

public UF() {

Arrays.fill(size, 1);

for (int i = 0;i < n;i++) id[i] = i; // 初始化为“每个i都是树，每个树只有一个节点”

}

private int find(int p) {

while (p != id[p]) p = id[p]; *// id[root] == root*

return p;

}

public void union(int p, int q) {

*// Set p and q to the same root.*

int pRoot = find(p);

int qRoot = find(q);

if (pRoot == qRoot) return;

if (size[p] < size[q]) { *// p is smaller, make it the sub-tree of q*

id[pRoot] = qRoot; *// make p become subtree of q.*

size[qRoot] += size[pRoot]; // update size of q.

} else {

id[qRoot] = pRoot;

size[pRoot] += size[qRoot];

}

count--;

}

**进一步优化：**

在以上的代码中，find的time： O(logN), 主要时间都花在找root上面，可以进行压缩：

private int find(int p) {

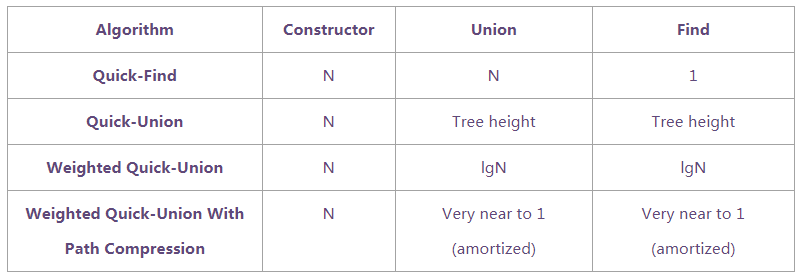
while (p != id[p]) {

id[p] = id[id[p]]; *// go faster*

p = id[p];

}

return p;

}

207. Course Schedule

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

For example: 2, [[1,0],[0,1]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

Note: The input prerequisites is a graph represented by a list of edges, not adjacency matrices.

**S1. DFS visited[] 存储访问状态**

首先建立adjacency array, 利用ArrayList[], 保存和每个node连接的其他node，

然后以每个点为起点，检查是否有cicle，方法是：

DFS, 一条路走到黑，所以保存boolean[] visited, 初始值是0，每次走到一个点时，把visited置为1，表示正走在这条路上，正在访问，（随后再进行访问neighbours的操作），等结束后再置为2表示访问结束。

所以如果路上遇到visited 为1的点，表示是第二次遇到，说明有环。

PS. 利用0,1,2三种状态，可以在hasCycle中剪枝，visited为2的不需要再检查。

public boolean canFinish(int numCourses, int[][] prerequisites) {

*// graph: graph.get(0) store all the nodes 0 pointed to.*

*// eg: (0, 1), (0, 2), then graph[0] store 1 and 2.*

ArrayList[] graph = new ArrayList[numCourses];

int[] visited = new int[numCourses];

for (int i = 0;i < numCourses;i++) {

graph[i] = new ArrayList<Integer>();

}

for (int[] num:prerequisites) {

int pre = num[1]; *// to take cur, you need to take pre first.*

int cur = num[0];

graph[pre].add(cur);

}

*// start from every node, check if there is a cycle*

for (int i = 0;i < numCourses;i++) {

if (hasCycle(i, visited, graph)) return false;

}

return true;

}

private boolean hasCycle(int cur, int[] visited, ArrayList[] graph) {

visited[cur] = 1;

boolean hasCycle = false;

for (int i = 0;i < graph[cur].size();i++) {

int next = (int)graph[cur].get(i);

if (visited[next] == 1) {

return true;

} else if (visited[next] == 0) {

*// check all the nodes connected with cur*

hasCycle = hasCycle || hasCycle(next, visited, graph);

}

}

visited[cur] = 2;

return hasCycle;

}

**S2. BFS**

保留所有节点的indegree, 同时存储每个节点的children，

* 首先在queue中加入root, indegree = 0的点，
* 每次加入一个node，它的所有的children的indegree -1,
* indegree = 0的节点变为下一轮的起始节点，加入queue

计数访问的点的数目，判断与总数是否一致即可。如果有环，环的部分永远无法加入。

public boolean canFinish2(int numCourses, int[][] prerequisites) {

ArrayList[] graph = new ArrayList[numCourses];

int[] degree = new int[numCourses];

for (int i = 0;i < numCourses;i++) {

graph[i] = new ArrayList<Integer>();

}

*// init indegree and graph structure: parent-children*

for (int[] num:prerequisites) {

int pre = num[1]; *// to take cur, you need to take pre first.*

int cur = num[0];

graph[pre].add(cur);

degree[cur]++;

}

Queue<Integer> q = new LinkedList<>();

for (int i = 0;i < numCourses;i++) { *// add root(indegree = 0)*

if (degree[i] == 0) q.add(i);

}

int visitedCount = 0;

while (!q.isEmpty()) {

int cur = q.poll();

visitedCount++;

for (int i = 0;i < graph[cur].size();i++) {

int next = (int)graph[cur].get(i);

degree[next]--;

if (degree[next] == 0) {

q.add(next);

}

}

}

return visitedCount == numCourses;

}

261. Graph Valid Tree

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function

to check whether these edges make up a valid tree.

Note: you can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the

same as [1, 0] and thus will not appear together in edges. For example:

Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false.

图是一棵树的条件，满足任意一个均可：

* 没有环 – DFS, BFS
* 顶点数是边数+1 && 图只有一个联通分量 – Union Find，并记录访问过的顶点总数

**S1.DFS**

检查是否有环，首先建立无向图，然后从任意一点开始，check与之相连的所有点，

public boolean validTree(int n, int[][] edges) {

ArrayList[] graph = new ArrayList[n];

for (int i = 0;i < n;i++) {

graph[i] = new ArrayList<>();

}

int[] states = new int[n]; *// 0: unvisited, 1:ing, 2:visited*

for (int[] edge:edges) {

int pre = edge[0];

int cur = edge[1];

graph[pre].add(cur);

graph[cur].add(pre);

}

*// start from any node, check all the nodes connected.*

if (hasCycle(-1, graph, 0, states)) return false; // if valid tree: loop through all nodes

*// check if there is any node not connected with any others*

for (int state:states) {

if (state == 0) return false;

}

return true;

}

boolean hasCycle(int prev, ArrayList[] graph, int cur, int[] states) {

states[cur] = 1;

boolean hascycle = false;

for (int i = 0;i < graph[cur].size();i++) {

int next = (int) graph[cur].get(i); // 必须强制转换，不然会报错

if (next != prev) { // make sure wont’ go back.

if (states[next] == 1) return true;

else if (states[next] == 0 && hasCycle(cur, graph, next, states)) {

return true;

}

}

}

states[cur] = 2;

return hascycle;

}

**S2. BFS**

public boolean validTree(int n, int[][] edges) {

ArrayList[] graph = new ArrayList[n];

for (int i = 0;i < n;i++) {

graph[i] = new ArrayList<>();

}

int[] states = new int[n]; *// 0: unvisited, 1:ing, 2:visited*

for (int[] edge:edges) {

int pre = edge[0], cur = edge[1];

graph[pre].add(cur);

graph[cur].add(pre);

}

Queue<Integer> q = new LinkedList<>();

q.offer(0);

int count = 0;

states[0] = 1;

while (!q.isEmpty()) {

int cur = q.poll();

count++;

for (int i = 0;i < graph[cur].size();i++) {

int kid = (int)graph[cur].get(i);

if (states[kid] == 1) return false;

else if (states[kid] == 0) {

states[kid] = 1;

q.offer(kid);

}

}

states[cur] = 2;

}

return count == n;

}

**S3.Union Find**

public boolean validTree(int n, int[][] edges) {

if (n == 0) return true;

return (edges.length == n-1 && connected(n, edges));

}

boolean connected(int n, int[][] edges) {

int[] size = new int[n];

int[] prev = new int[n];

Arrays.fill(size, 1);

for (int i = 0;i < n;i++) prev[i] = i;

int maxSize = 1;

for (int[] edge:edges) {

int r1 = edge[0], r2 = edge[1];

*// union:*

while (prev[r1] != r1) r1 = prev[r1];

while (prev[r2] != r2) r2 = prev[r2];

if (size[r1] > size[r2]) {

prev[r2] = r1;

size[r1] += size[r2];

} else {

prev[r1] = r2;

size[r2] += size[r1];

}

maxSize = size[r2];

}

return maxSize == n;

}

Topological Sort

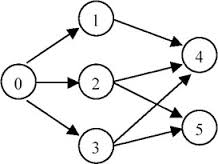
Lintcode - Topological Sorting

Given an directed graph, a topological order of the graph nodes is defined as follow:

- For each directed edge A -> B in graph, A must before B in the order list.

- The first node in the order can be any node in the graph with no nodes direct to it.

Find any topological order for the given graph. For graph as follow:



The topological order can be: [0, 1, 2, 3, 4, 5] [0, 2, 3, 1, 5, 4]...

其实就是在图里找一个path，先放indegree = 0的点，然后是第二层的点...每一层内的点顺序无关紧要。

**S1.DFS**

class DirectedGraphNode {

int label;

ArrayList<DirectedGraphNode> neighbors;

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

};

public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {

ArrayList<DirectedGraphNode> res = new ArrayList<>();

if (graph == null) return res;

Set<DirectedGraphNode> visited = new HashSet<>();

for (DirectedGraphNode n:graph) {

dfs(res, n, visited);

}

return res;

}

private void dfs(ArrayList<DirectedGraphNode> res, DirectedGraphNode root,

Set<DirectedGraphNode> visited) {

if (visited.contains(root)) return;

for (DirectedGraphNode n:root.neighbors) {

dfs(res, n, visited);

}

res.add(0, root); *// 在children都已经add之后，再add root, keep order*

visited.add(root);

}

**S2.BFS**

public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {

ArrayList<DirectedGraphNode> res = new ArrayList<>();

if (graph == null) return res;

*// 1. Count in-degree of each node*

Map<DirectedGraphNode, Integer> map = new HashMap<>();

for (DirectedGraphNode node:graph) {

for (DirectedGraphNode neighbor:node.neighbors) {

if (!map.containsKey(neighbor)) {

map.put(neighbor, 1);

} else {

map.put(neighbor, map.get(neighbor) + 1);

}

}

}

Queue<DirectedGraphNode> q = new LinkedList<>();

for (DirectedGraphNode i:graph) {

if (!map.containsKey(i)) { *// first round*

res.add(i);

q.add(i);

}

}

while (!q.isEmpty()) {

DirectedGraphNode cur = q.poll();

for (DirectedGraphNode next:cur.neighbors) {

map.put(next, map.get(next) - 1);

if (map.get(next) == 0) {

res.add(next);

q.add(next);

}

}

}

return res;

}

210. Course Schedule II

Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

需要返回的是上课的顺序

**S1. BFS**

本质是directed graph，已经知道没有环，需要**topological sort**

与上一道题的区别就是需要记录顺序, 也就是拓扑排序。也可以用DFS做。

public int[] findOrder(int numCourses, int[][] prerequisites) {

*// graph[0]: courses that have graph[0] as prerequisite*

ArrayList[] graph = new ArrayList[numCourses];

int[] degree = new int[numCourses];

int[] order = new int[numCourses];

int idx = 0;

for (int i = 0;i < numCourses;i++) { *// create ArrayList*

graph[i] = new ArrayList<>();

}

for (int[] n:prerequisites) { *// depth. the first course to take, degree = 0*

int pre = n[1];

int cur = n[0];

graph[pre].add(cur);

degree[cur]++;

}

Queue<Integer> q = new LinkedList<>();

int visitedCount = 0;

for (int i = 0;i < numCourses;i++) {

if (degree[i] == 0) q.add(i);

}

while (!q.isEmpty()) {

int cur = (int)q.poll(); *// Integer -> int need casting*

visitedCount++; *// check if all courses can be taken or not*

order[idx++] = cur;

for (int i = 0;i < graph[cur].size();i++) {

int next = (int)graph[cur].get(i);

degree[next]--;

if (degree[next] == 0) {

q.add(next);

}

}

}

return visitedCount == numCourses ? order : new int[0];

}

630. Course Schedule III

There are n different online courses numbered from 1 to n. Each course has some duration(course length) t and closed on dth day. A course should be taken continuously for t days and must be finished before or on the dth day. You will start at the 1st day.

Given n online courses represented by pairs (t,d), your task is to find the maximal number of courses that can be taken.

**S. PriorityQueue**

用sum保存截止到目前的time used, 用PQ保存已经加入sum的所有课程的时间，

每次如果超过可用的时间，则减去一个时间最长的。

time O(nlogn)

public int scheduleCourse(int[][] courses) {

if (courses == null || courses.length == 0) return 0;

Arrays.sort(courses, (a, b)->(a[1] - b[1]));

*// reverse order, decrease*

PriorityQueue<Integer> pq = new PriorityQueue<>((a, b)->(b-a));

int sum = 0; *// keep track of sum of time used.*

for (int[] course:courses) {

sum += course[0];

pq.add(course[0]);

if (sum > course[1]) {

sum -= pq.poll(); *// drop the course that takes most time.*

}

}

return pq.size();

}

444. Sequence Reconstruction

Check whether the original sequence org can be uniquely reconstructed from the sequences in seqs. The org sequence is a permutation of the integers from 1 to n, with 1 ≤ n ≤ 104. Reconstruction means building a shortest common supersequence of the sequences in seqs (i.e., a shortest sequence so that all sequences in seqs are subsequences of it). Determine whether there is only one sequence that can be reconstructed from seqs and it is the org sequence.

Input: org: [1,2,3], seqs: [[1,2],[1,3]] Output: false

Explanation:

[1,2,3] is not the only one sequence that can be reconstructed, because [1,3,2] is also a valid sequence that can be reconstructed.

Input: org: [4,1,5,2,6,3], seqs: [[5,2,6,3],[4,1,5,2]] Output: true

**S Graph – 分析性质**

如果要保证可行&唯一，需要满足两个条件：

1. seqs中的每个seq，都是org的subsequence, 不一定连续
2. org中每相连的两个数字，都在seqs中一定相连，不然无法保证顺序唯一。

org里的数字范围在1~n, 所以可以建立idx数组，保存每个数字的index， 检查性质1是否满足；

再建立一个boolean[], 在遍历seqs中，每次都检查相连的两个数字index是否相连，如果相连则设为true，

如果唯一可行，那么到最后boolean[]的每一位都会为true。

public boolean sequenceReconstruction(int[] org, List<List<Integer>> seqs) {

if (seqs == null || seqs.size() == 0) return false;

int n = org.length;

*// the index of each number in org*

int[] idx = new int[n + 1];

boolean[] pairs = new boolean[n - 1];

for (int i = 0;i < n;i++) {

idx[org[i]] = i;

}

int count = 0;

for (List<Integer> seq:seqs) {

int size = seq.size();

count += size;

for (int i = 0;i < size;i++) {

if (seq.get(i) < 0 || seq.get(i) > n) return false;

*// index of seq[i-1] should always < index of seq[i]*

if (i > 0 && idx[seq.get(i-1)] >= idx[seq.get(i)]) return false;

if (i > 0 && idx[seq.get(i-1)] + 1 == idx[seq.get(i)]) {

pairs[idx[seq.get(i-1)]] = true;

}

}

}

if (count < n) return false;

for (boolean pair:pairs) {

if (pair == false) return false;

}

return true;

}

Just BFS / DFS

399. Evaluate Division

Equations are given in the format A / B = k, where A and B are variables represented as strings, and k is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return -1.0.

Example:

Given a / b = 2.0, b / c = 3.0.

queries are: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .

equations = [ ["a", "b"], ["b", "c"] ],

values = [2.0, 3.0],

queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].

return [6.0, 0.5, -1.0, 1.0, -1.0 ].

**S1. DFS**

使用两个map，分别存储与每个String有关联的其他String，以及对应的values, 通过index进行匹配，

通过Set<String> 记录来时走过的路/String，避免回头check重复的，进行剪枝pruning

public double[] calcEquation(String[][] equations, double[] values, String[][] queries) {

Map<String, List<String>> nodes = new HashMap<>();

Map<String, List<Double>> weights = new HashMap<>();

for (int i = 0;i < equations.length;i++) {

String[] equation = equations[i];

if (!nodes.containsKey(equation[0])) {

nodes.put(equation[0], new ArrayList<String>());

weights.put(equation[0], new ArrayList<Double>());

}

if (!nodes.containsKey(equation[1])) {

nodes.put(equation[1], new ArrayList<String>());

weights.put(equation[1], new ArrayList<Double>());

}

nodes.get(equation[0]).add(equation[1]);

nodes.get(equation[1]).add(equation[0]);

weights.get(equation[0]).add(values[i]);

weights.get(equation[1]).add(1/values[i]);

}

double[] res = new double[queries.length];

for (int i = 0;i < queries.length;i++) {

String[] query = queries[i];

res[i] = dfs(query[0], query[1], nodes, weights, new HashSet<>(), 1.0);

if (res[i] == 0.0) res[i] = -1.0;

}

return res;

}

*// set: used for pruning, won't check back, a-b-a...*

double dfs(String start, String end, Map<String, List<String>> nodes, Map<String, List<Double>> weights, Set<String> visited, double cur) {

if (visited.contains(start)) return 0.0;

if (!nodes.containsKey(start)) return 0.0;

if (start.equals(end)) return cur;

visited.add(start);

List<String> kids = nodes.get(start);

double res = 0.0;

for (int i = 0;i < kids.size();i++) {

res = dfs(kids.get(i), end, nodes, weights, visited, cur\*weights.get(start).get(i));

if (res != 0.0) break;

}

visited.remove(start);

return res;

}

Union-Find 典型题目

**305. Number of Islands II**

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).

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

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

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

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

0 0 0 1 0 0 1 1 0 1 1 0 1 1 0

0 0 0 0 0 0 0 0 0 0 0 1 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

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

**S1.Union-Find**

每次加一个点，并查看一次有几个land，非常典型的Union-Find问题，返回的就是count。

变化是初始值不同，增加了一个add函数，一次添加一个点的初始值，不能在一开始设置好。

其他不变，建立一个class UF来统一处理， time O(k\*(logm + logn)).

每次放进一个新的点后，查看上下左右有没有island，这一点通过查看size实现，是否> 0.

public List<Integer> numIslands2(int m, int n, int[][] positions) {

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

int[][] biases = new int[][]{{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

UF uf = new UF(m, n);

for (int[] position:positions) {

int x = position[0], y = position[1];

int p = uf.add(x, y);

for (int[] bias:biases) {

int q = uf.getIndex(x + bias[0], y + bias[1]);

*// make sure there's 1 at place q.*

if (uf.size[q] > 0 && !uf.connected(p, q)) uf.union(p, q);

}

res.add(uf.count);

}

return res;

}

class UF {

int[] id; *// id[0], size[0] reserved for invalid conditions*

int[] size;

int m, n, count; *// count: # of groups*

public UF(int m, int n) {

this.m = m;

this.n = n;

id = new int[m\*n + 1];

size = new int[m\*n + 1];

}

int getIndex(int x, int y) {

if (x < 0 || x >= m || y < 0 || y >= n) return 0;

return x\*n + y + 1;

}

*// add a new node and return index*

int add(int x, int y) {

int idx = getIndex(x, y);

id[idx] = idx;

size[idx] = 1;

count++;

return idx;

}

*// find the root of p*

int find(int p) {

while (p != id[p]) p = id[p];

return p;

}

boolean connected(int p, int q) {

return find(p) == find(q);

}

public void union(int p, int q) {

int pRoot = find(p);

int qRoot = find(q);

if (pRoot == qRoot) return;

if (size[pRoot] > size[qRoot]) {

id[qRoot] = pRoot;

size[pRoot] += size[qRoot];

} else {

id[pRoot] = qRoot;

size[qRoot] += size[pRoot];

}

count--;

}

}

323. Number of Connected Components in an Undirected Graph

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function

to find the number of connected components in an undirected graph.

Example 1:

0 3

| |

1 --- 2 4

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], return 2.

Example 2:

0 4

| |

1 --- 2 --- 3

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3, 4]], return 1.

Note:You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as

[1, 0] and thus will not appear together in edges.

**S1. Union-Find**

非常经典的应用，求count

Time: O(n\*height), n is # of edges, height is the height of tree, ~O(n logn)

public int countComponents(int n, int[][] edges) {

int count = n;

int[] id = new int[n];

int[] size = new int[n];

Arrays.fill(size, 1);

for (int i = 0;i < n;i++) id[i] = i;

for (int[] edge:edges) {

int node1 = edge[0], node2 = edge[1];

while (node1 != id[node1]) node1 = id[node1]; // find the root.

while (node2 != id[node2]) node2 = id[node2];

if (node1 == node2) continue;

if (size[node1] > size[node2]) { // 优化，尽量使树平衡

id[node2] = node1;

size[node1] += size[node2];

} else {

id[node1] = node2;

size[node2] += size[node1];

}

count--;

}

return count;

}

**Union-Find 优化策略：**

主要的优化策略一共有两个，

一个是weighted union, id[p] = q, 本身的意义是p会作为q的子树，那么为了树尽量平衡，最好把小的树作为大的树的子树，所以通过比较size，来决定加到哪里；

另一个是quick-union, 最理想的情况是树非常扁平，所有的child都直接连到root，保证find的最高效率。所以可以对路径进行压缩：

public int countComponents(int n, int[][] edges) {

int count = n;

int[] id = new int[n];

int[] size = new int[n];

Arrays.fill(size, 1);

for (int i = 0;i < n;i++) id[i] = i;

for (int[] edge:edges) {

int node1 = edge[0], node2 = edge[1];

node1 = findRoot(id, node1);

node2 = findRoot(id, node2);

if (node1 == node2) continue;

if (size[node1] > size[node2]) {

id[node2] = node1;

size[node1] += size[node2];

} else {

id[node1] = node2;

size[node2] += size[node1];

}

count--;

}

return count;

}

// 主要是这个函数做的改变，加了一行

int findRoot(int[] id, int cur) {

while (cur != id[cur]) {

id[cur] = id[id[cur]]; // 直接指到爷爷节点，compression 压缩路径

cur = id[cur];

}

return cur;

}

547. Friend Circles

There are N students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a direct friend of B, and B is a direct friend of C, then A is an indirect friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

Given a N\*N matrix M representing the friend relationship between students in the class. If M[i][j] = 1, then the ith and jth students are direct friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

Example 1: Input:

[ [1,1,0],

[1,1,0],

[0,0,1]]

Output: 2

Explanation:The 0th and 1st students are direct friends, so they are in a friend circle. The 2nd student himself is in a friend circle. So return 2.

Example 2: Input:

[ [1,1,0],

[1,1,1],

[0,1,1]]

Output: 1

Explanation:The 0th and 1st students are direct friends, the 1st and 2nd students are direct friends,

so the 0th and 2nd students are indirect friends. All of them are in the same friend circle, so return 1

N is in range [1,200], M[i][i] = 1 for all students., If M[i][j] = 1, then M[j][i] = 1.

**S1.Union-Find**

其实也是Union-Find找count，初始值其实是对角线全部为1，其他全部为0，自己是自己的direct friends，并且count的初始值为n：

1 0 0 1 1 0

0 1 0 0 1 0

0 0 1 0 0 1

事实上只有n个人，# of circles的最大值为n，因此int[] size, int[] id, 长度只需要为n就可以。

同时考虑到矩阵对称symmetric，可以只遍历一半，比如上图可以只遍历上半个三角形，如果其中有一点值为1，比如图2，就执行union操作，也就相当于把两个对角线上的点union到一起，归属于一个group。

public int findCircleNum(int[][] M) {

if (M == null || M.length == 0) return 0;

int n = M.length;

UnionFind uf = new UnionFind(n);

for (int i = 1;i < n;i++) {

for (int j = 0;j < i;j++) {

if (M[i][j] == 1) uf.union(i, j);

}

}

return uf.count();

}

class UnionFind {

private int[] id;

private int[] size;

private int count;

public UnionFind(int n) {

id = new int[n];

size = new int[n];

Arrays.fill(size, 1);

for (int i = 0;i < n;i++) {

id[i] = i;

}

count = n;

}

private int find(int p) { *// find root of p*

while (p != id[p]) {

id[p] = id[id[p]]; *// compression*

p = id[p];

}

return p;

}

public boolean connected(int p, int q) {

return find(p) == find(q);

}

public void union(int p, int q) {

int rp = find(p), rq = find(q);

if (rp == rq) return;

if (size[rp] > size[rq]) {

id[rq] = rp;

size[rp] += size[rq];

} else {

id[rp] = rq;

size[rq] += size[rp];

}

count--;

}

public int count() {return count;}

}

**遍历所有点**

133. Clone Graph

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

**S1 DFS**

保存一个<old node, new node>之间的map，采用直接的DFS方法，如果发现走过就return，没有走过就全走一遍neighbors

*/\*\**

*\* Using DFS, add all the new nodes of neighbors*

*\* new nodes which has same label as neighbors of node*

*\*/*

*// map: <old, new>*

Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null) return null;

if (map.containsKey(node)) return map.get(node);

UndirectedGraphNode root = new UndirectedGraphNode(node.label);

map.put(node, root);

for (UndirectedGraphNode neighbor:node.neighbors) {

root.neighbors.add(cloneGraph(neighbor));

}

return root;

}

**S2. BFS**

BFS也可以做，需要先取到所有的nodes，再建立neighbor的联系：

Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null) return node;

*// 1. loop through all the nodes, create new nodes and put into hashmap*

List<UndirectedGraphNode> nodeList = getNodes(node);

for (UndirectedGraphNode n: nodeList) {

map.put(n, new UndirectedGraphNode(n.label));

}

*// 2. loop through all the nodes, and update neighbors*

for (UndirectedGraphNode n: nodeList) {

UndirectedGraphNode newNode = map.get(n);

for (UndirectedGraphNode neighbor:n.neighbors) {

newNode.neighbors.add(map.get(neighbor));

}

}

return map.get(node);

}

private List<UndirectedGraphNode> getNodes(UndirectedGraphNode node) {

Set<UndirectedGraphNode> visited = new HashSet<>();

Queue<UndirectedGraphNode> queue = new LinkedList<>();

queue.add(node);

visited.add(node);

while (!queue.isEmpty()) {

UndirectedGraphNode cur = queue.poll();

for (UndirectedGraphNode neighbor:cur.neighbors) {

if (!visited.contains(neighbor)) {

visited.add(neighbor);

queue.add(neighbor);

}

}

}

return new ArrayList<>(visited);

}

138. Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null. Return a deep copy of the list.

**S1 graph, HashMap**

同样也是，先create node, 建立<old, new>的对应关系，再设置node的next和random指针。

public RandomListNode copyRandomList(RandomListNode head) {

if (head == null) return null;

Map<RandomListNode, RandomListNode> map = new HashMap<RandomListNode, RandomListNode>();

*// 1. create new node and put into map*

RandomListNode cur = head;

while (cur != null) {

map.put(cur, new RandomListNode(cur.label));

cur = cur.next;

}

*// 2. link new nodes together*

cur = head;

while (cur != null) {

map.get(cur).next = map.get(cur.next);

map.get(cur).random = map.get(cur.random);

cur = cur.next;

}

return map.get(head);

}

Graph 性质

Houzz - check whether a Graph is bipartite or not

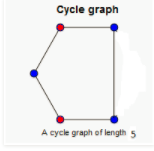
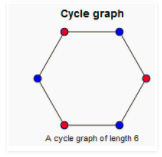
关于这个题的具体描述和思路在这里：

<http://www.geeksforgeeks.org/bipartite-graph/>

a graph is bipartite == graph由nodes组成，能不能把nodes涂成两种颜色，并且相同的颜色不挨着。

如果有self-circle, 那么一定不是bipartite.

Bipartite: non-bipartite:



**S1. BFS**

从随意一个点开始，用map <node, color> 标上红色，然后把相邻的都标上蓝色，

继续往后走，如果遇到一个点，想给它标红的时候发现已经标蓝，或者相反，说明non-bipartite.

176. Route Between Two Nodes in Graph

Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

思路：从s出发，用一个set装已经visited的点，避免走圈

while (s not empty) {

for all neighbors:

if is t, return true;

else if not in set, add to stack, add to set.

}

return false.