<https://www.lintcode.com/problem/178/description>

**Description**

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.

**Example**

**Example 1:**

Input: n = 5 edges = [[0, 1], [0, 2], [0, 3], [1, 4]]

Output: true.

**Example 2:**

Input: n = 5 edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]]

Output: false.

**Attempt 1: 2022-11-14**

**Solution 1:  Recursive traversal as DFS (10min)**

public class Solution {

/\*\*

\* @param n: An integer

\* @param edges: a list of undirected edges

\* @return: true if it's a valid tree, or false

\*/

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

// 1.No cycle in undirected graph

// 2.All nodes connected in undirected graph

// Build graph

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

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

graph.put(i, new ArrayList<Integer>());

}

for(int[] edge : edges) {

graph.get(edge[0]).add(edge[1]);

graph.get(edge[1]).add(edge[0]);

}

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

// Since the graph is undirected, when we visit a child we would always add

// parent to the next visit list(when create graph as adjacencies list we

// mutually add node as parent -> child and child -> parent since its

// undirected graph, so child node's parent will always in child's

// adjacencies list). This creates a trivial cycle and not the real cycle

// we want. We can avoid detecting trivial cycle but adding an additional

// parent state in the DFS call, when find child node's parent in its

// adjacencies list, just continue to check next adjacency node

int parent = -1;

// Traverse from random node, let's start from 0

int cur = 0;

visited.add(cur);

boolean noCycle = helper(graph, visited, parent, cur);

return noCycle && visited.size() == n;

}

private boolean helper(Map<Integer, List<Integer>> graph, Set<Integer> visited, int parent, int cur) {

List<Integer> adjacencies = graph.get(cur);

for(int adjacency : adjacencies) {

// If find parent in child's adjacencies list, its not real cycle,

// just continue to check next adjacency node

if(adjacency == parent) {

continue;

}

// If not parent but already visited before, we find a cycle

if(visited.contains(adjacency)) {

return false;

}

visited.add(adjacency);

boolean noCycle = helper(graph, visited, cur, adjacency);

if(!noCycle) {

return false;

}

}

return true;

}

}

Time Complexity: O(N + E) Creating the adjacency list of N nodes and E edges will take O(E) + O(N) = O(N+E) time. As each node is added to the data structure only once, there will be N iterations and for each node, its adjacent edges will be traversed only once. We are ensuring this by keeping a visited array. Therefore, total E edges are iterated over once by the loop, and hence, the time complexity is O(N + E).

Space Complexity: O(N + E) The adjacency list consists of N nodes with E edges, hence O(N + E) space. In the worst case, the stack/queue will have all N nodes on it at the same time, resulting in a total of O(N) space. Hence the space complexity is O(E + N).

**Refer to**

<https://algomonster.medium.com/leetcode-261-graph-valid-tree-f27c212c1db1>

Before solving the problem, we have to know the definitions.

**Tree vs. Graph**

A tree is a special undirected graph. It satisfies two properties

1. It is connected
2. It has no cycle.

Being connected means you can start from any node and reach any other node. To prove it, we can do a DFS and add each node we visit to a set. After we visited all the nodes, we compare the number of nodes in the set with the total number of nodes. If they are the same then every node is accessible from any other node and the graph is connected.

To prove an undirected graph having no cycle, we can also do a DFS. If a graph contains a cycle, then we would visit a certain node more than once. There is a minor caveat, since the graph is undirected, when we visit a child we would always add parent to the next visit list. This creates a trivial cycle and not the real cycle we want. We can avoid detecting trivial cycle but adding an additional parent state in the DFS call.

We can check both properties in one DFS call since cycle detection always keeps track of a visited set.

class Solution:

def validTree(self, n: int, edges: List[List[int]]) -> bool:

from collections import defaultdict

graph = defaultdict(list)

# build the graph

for src, dest in edges:

graph[src].append(dest)

graph[dest].append(src)

visited = set()

def dfs(root, parent): # returns true if graph has no cycle

visited.add(root)

for node in graph[root]:

if node == parent: # trivial cycle, skip

continue

if node in visited:

return False

if not dfs(node, root):

return False

return True

return dfs(0, -1) and len(visited) == n

**Alternative Solution**

There’s actually an even simpler version. Condition 2 no cycle can also be expressed as # of nodes == # of edges + 1.

class Solution:

def validTree(self, n: int, edges: List[List[int]]) -> bool:

from collections import defaultdict

graph = defaultdict(list)

# build the graph

for src, dest in edges:

graph[src].append(dest)

graph[dest].append(src)

visited = set()f

def dfs(root):

visited.add(root)

for node in graph[root]:

if node in visited:

continue

dfs(node)

dfs(0)

return len(visited) == n and len(edges) == n - 1

**Note: For Alternative Solution**

<https://www.lintcode.com/problem/178/solution/18270>

判断一个图是否是树有三个条件：

1. 联通性
2. **边数 = 节点数 - 1**
3. 是否有环

**另外根据图论：以上三条满足任意两条即可**

**Solution 2:  Recursive traversal as BFS (10min)**

public class Solution {

/\*\*

\* @param n: An integer

\* @param edges: a list of undirected edges

\* @return: true if it's a valid tree, or false

\*/

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

// 1.No cycle in undirected graph

// 2.All nodes connected in undirected graph

// Build graph

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

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

graph.put(i, new ArrayList<Integer>());

}

for(int[] edge : edges) {

graph.get(edge[0]).add(edge[1]);

graph.get(edge[1]).add(edge[0]);

}

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

// Similar as DFS to find cycle in undirected graph requires 'parent' check

// for trivial cycle which is not real cycle

int[] parent = new int[n];

Arrays.fill(parent, -1);

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

// Traverse from random node, let's start from 0

int cur = 0;

q.offer(cur);

visited.add(cur);

while(!q.isEmpty()) {

int node = q.poll();

List<Integer> adjacencies = graph.get(node);

for(int adjacency : adjacencies) {

if(!visited.contains(adjacency)) {

visited.add(adjacency);

q.offer(adjacency);

parent[adjacency] = node;

// If visited before and its parent has been altered (when not visited

// condition enter by other node and re-assign new value), means real cycle find

} else if(parent[node] != adjacency) {

return false;

}

}

}

return visited.size() == n;

}

}

Time Complexity: O(N + E) Creating the adjacency list of N nodes and E edges will take O(E) + O(N) = O(N+E) time. As each node is added to the data structure only once, there will be N iterations and for each node, its adjacent edges will be traversed only once. We are ensuring this by keeping a visited array. Therefore, total E edges are iterated over once by the loop, and hence, the time complexity is O(N + E).

Space Complexity: O(N + E) The adjacency list consists of N nodes with E edges, hence O(N + E) space. In the worst case, the stack/queue will have all N nodes on it at the same time, resulting in a total of O(N) space. Hence the space complexity is O(E + N).

**Step by Step explain how BFS detect cycle**

Input: int[][] edges = new int[][]{{0,1},{1,2},{2,3},{1,3},{1,4}};

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

// 1.No cycle in undirected graph

// 2.All nodes connected in undirected graph

// Build graph

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

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

graph.put(i, new ArrayList<Integer>());

}

for(int[] edge : edges) {

graph.get(edge[0]).add(edge[1]);

graph.get(edge[1]).add(edge[0]);

}

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

// Similar as DFS to find cycle in undirected graph requires 'parent' check

// for trivial cycle which is not real cycle

int[] parent = new int[n];

Arrays.fill(parent, -1);

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

// Traverse from random node, let's start from 0

int cur = 0;

q.offer(cur);

visited.add(cur);

while(!q.isEmpty()) {

int node = q.poll();

System.out.println("Poll node: " + node + " from q");

System.out.println("q is: " + q.toString());

List<Integer> adjacencies = graph.get(node);

System.out.println("node: " + node + " -> adjacencies: " + adjacencies.toString());

for(int adjacency : adjacencies) {

if(!visited.contains(adjacency)) {

System.out.println("visited not contains: " + adjacency);

visited.add(adjacency);

System.out.println("visited add: " + adjacency);

q.offer(adjacency);

System.out.println("q add: " + adjacency + ", q is:" + q.toString());

parent[adjacency] = node;

System.out.println("parent set for: " + adjacency + " -> " + node);

System.out.println("---------------------------------------------");

// If visited before and its parent has been altered (when not visited

// condition enter by other node and re-assign new value), means real cycle find

} else if(parent[node] != adjacency) {

System.out.println(adjacency + " has visited before and recorded in visited set");

System.out.println("But now parent of : " + node + " is " + parent[node] + " not equal to its adjacency: " + adjacency + ", find a real cycle");

return false;

}

}

}

return visited.size() == n;

}

e.g

\_\_\_\_\_

| |

0-1-2-3

|

4

0={1}

1={0,2,3,4}

2={1,3}

3={1,2}

4={1}

Poll node: 0 from q

q is: []

node: 0 -> adjacencies: [1]

visited not contains: 1

visited add: 1

q add: 1, q is:[1]

parent set for: 1 -> 0

---------------------------------------------

Poll node: 1 from q

q is: []

node: 1 -> adjacencies: [0, 2, 3, 4]

visited not contains: 2

visited add: 2

q add: 2, q is:[2]

parent set for: 2 -> 1

---------------------------------------------

visited not contains: 3

visited add: 3

q add: 3, q is:[2, 3]

parent set for: 3 -> 1

---------------------------------------------

visited not contains: 4

visited add: 4

q add: 4, q is:[2, 3, 4]

parent set for: 4 -> 1

---------------------------------------------

Poll node: 2 from q

q is: [3, 4]

node: 2 -> adjacencies: [1, 3]

3 has visited before and recorded in visited set

But now parent of : 2 is 1 not equal to its adjacency: 3, find a real cycle

false

=============================================

We have a conflict for 2's parent and cause a real cycle is based on re-assign

for parent of node 2 happen when build the graph

Initially we have 2={1,3} which set 1 as 2's parent, but then we have 3={1,2}

which set 3 as 2's parent, the re-assign parent from 1 to 3

**Another BFS solution refer to**

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

在于每次遍历邻接表时，对于node的neighbor，放入BFS的queue中之后，就把node本身从neighbor的邻接表中移除。

for(int neighbor : graph.get(node))

{

queue.offer(neighbor);

graph.get(neighbor).remove((Integer)node);

}

或者，只在neighbor没有被遍历过时才放入queue中：

for(int i: map.get(top)){

// only putting new node into the queue

if(!visited[i])

queue.offer(i);

}

这样，在遍历出队列时，如果遇到元素被二次访问就说明有cycle。

// check cycle

int top = queue.poll();

if(visited[top]) return false;

最后遍历visited[]，确认每一个元素都被遍历到，才是valid tree（没有落单的节点）

// fully connected

for(boolean b: visited){

if(!b) return false;

}

实现

class Solution {

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

List<Set<Integer>> adjList = new ArrayList<>();

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

adjList.add(new HashSet<>());

}

for (int[] edge : edges) {

adjList.get(edge[0]).add(edge[1]);

adjList.get(edge[1]).add(edge[0]);

}

boolean[] visited = new boolean[n];

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

queue.add(0);

while (!queue.isEmpty()) {

int curNode = queue.remove();

// found loop

if (visited[curNode]) {

return false;

}

visited[curNode] = true;

for (int nextNode : adjList.get(curNode)) {

queue.add(nextNode);

adjList.get(nextNode).remove(curNode);

}

}

for (boolean e : visited) {

if (!e) {

return false;

}

}

return true;

}

}

**Solution 3: Union Find (10min)**

**Style 1:  Path compression**

public class Solution {

/\*\*

\* @param n: An integer

\* @param edges: a list of undirected edges

\* @return: true if it's a valid tree, or false

\*/

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

if(n <= 0 || edges.length != n - 1) {

return false;

}

UnionFind uf = new UnionFind(n);

for(int[] edge : edges) {

uf.union(edge[0], edge[1]);

}

return uf.getCount() == 1;

}

}

class UnionFind {

private int[] parents;

private int count;

public UnionFind(int n) {

count = n;

parents = new int[n];

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

parents[i] = i;

}

}

public void union(int m, int n) {

int src = find(m);

int dst = find(n);

if(src != dst) {

parents[src] = dst;

count--;

}

}

//public int no\_path\_compression\_find(int x) {

// if(parents[x] == x) {

// return x;

// }

// return find(parents[x]);

//}

public int find(int x) {

if(parents[x] == x) {

return x;

}

return parents[x] = find(parents[x]);

}

public int getCount() {

return count;

}

}

Refer to

Union Find (Disjoint Set Union) Path Compression and Union by size or rank optimization

https://cp-algorithms.com/data\_structures/disjoint\_set\_union.html#path-compression-optimization

Time Complexity: O(logN) -> after path compression reduce from O(N) to O(logN)

Space Complexity: O(N)

**Style 2: Path Compression and Rank Weighted**

public class Solution {

/\*\*

\* @param n: An integer

\* @param edges: a list of undirected edges

\* @return: true if it's a valid tree, or false

\*/

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

if(n <= 0 || edges.length != n - 1) {

return false;

}

UnionFind uf = new UnionFind(n);

for(int[] edge : edges) {

uf.weightedUnion(edge[0], edge[1]);

}

return uf.getCount() == 1;

}

}

class UnionFind {

private int[] parents;

private int[] ranks;

private int count;

public UnionFind(int n) {

count = n;

parents = new int[n];

ranks = new int[n];

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

parents[i] = i;

ranks[i] = 1;

}

}

public void weightedUnion(int m, int n) {

int src = find(m);

int dst = find(n);

if(src != dst) {

if(ranks[src] > ranks[dst]) {

parents[dst] = src;

ranks[src] += ranks[dst];

} else {

parents[src] = dst;

ranks[dst] += ranks[src];

}

count--;

}

}

public int find(int x) {

if(parents[x] == x) {

return x;

}

return parents[x] = find(parents[x]);

}

public int getCount() {

return count;

}

}

Refer to

Union Find (Disjoint Set Union) Path Compression and Union by size or rank optimization

https://cp-algorithms.com/data\_structures/disjoint\_set\_union.html#path-compression-optimization

Time Complexity: O(alpha(n)) -> kind of constants time complexity

Space Complexity: O(N)

**Refer to**

<https://segmentfault.com/a/1190000003791051>

**并查集**

**复杂度**

时间 O(N^M) 空间 O(1)

**思路**

判断输入的边是否能构成一个树，我们需要确定两件事：

1. 这些边是否构成环路，如果有环则不能构成树
2. 这些边是否能将所有节点连通，如果有不能连通的节点则不能构成树

因为不需要知道具体的树长什么样子，只要知道连通的关系，所以并查集相比深度优先搜索是更好的方法。我们定义一个并查集的数据结构，并提供标准的四个接口：

* union 将两个节点放入一个集合中
* find 找到该节点所属的集合编号
* areConnected 判断两个节点是否是一个集合
* count 返回该并查集中有多少个独立的集合

具体并查集的原理，参见[这篇文章](https://link.segmentfault.com/?enc=iTqwCgOCGoDmVoKyPjmlJw%3D%3D.5jxOjLaxRxTJ9p%2BTRBGykpK0sc9ynLZJSkP7xeCSvOL%2BiFJ642KiBgBLgohyC3CbX7wP5C2EpUNfLmrsfMd1ew%3D%3D)。简单来讲，就是先构建一个数组，节点0到节点n-1，刚开始都各自独立的属于自己的集合。这时集合的编号是节点号。然后，每次union操作时，我们把整个并查集中，所有和第一个节点所属集合号相同的节点的集合号，都改成第二个节点的集合号。这样就将一个集合的节点归属到同一个集合号下了。我们遍历一遍输入，把所有边加入我们的并查集中，加的同时判断是否有环路。最后如果并查集中只有一个集合，则说明可以构建树。

**注意**

因为要判断是否会产生环路，union方法要返回一个boolean，如果两个节点本来就在一个集合中，就返回假，说明有环路

public class Solution {

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

UnionFind uf = new UnionFind(n);

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

// 如果两个节点已经在同一集合中，说明新的边将产生环路

if(!uf.union(edges[i][0], edges[i][1])){

return false;

}

}

return uf.count() == 1;

}

public class UnionFind {

int[] ids;

int cnt;

public UnionFind(int size){

this.ids = new int[size];

//初始化并查集，每个节点对应自己的集合号

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

this.ids[i] = i;

}

this.cnt = size;

}

public boolean union(int m, int n){

int src = find(m);

int dst = find(n);

//如果两个节点不在同一集合中，将两个集合合并为一个

if(src != dst){

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

if(ids[i] == src){

ids[i] = dst;

}

}

// 合并完集合后，集合数减一

cnt--;

return true;

} else {

return false;

}

}

public int find(int m){

return ids[m];

}

public boolean areConnected(int m, int n){

return find(m) == find(n);

}

public int count(){

return cnt;

}

}

}

**Two more tips**

**1.Additional edges = n - 1 check is more robust**

**Refer to**

<https://www.lintcode.com/problem/178/solution/16793?fromId=222&_from=collection>

直接用的Union Find最基本的模版做的。这道题就是两步走：

**1.判断edges数目必须等于n-1**

2.连通图总集合数为1。这样就保证了图中所有点连通且无环。

class UnionFind{

private int[] father;

private int count;

public UnionFind(int n) {

father = new int[n];

count = n;

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

father[i] = i;

}

}

public int getCount() {

return count;

}

public int find(int x) {

if(father[x]==x) {

return x;

}

return father[x] = find(father[x]);

}

public void connect(int a, int b) {

int rootA = find(a);

int rootB = find(b);

if(rootA!=rootB) {

father[rootA] = rootB;

count--;

}

}

}

public class Solution {

/\*

\* @param n: An integer

\* @param edges: a list of undirected edges

\* @return: true if it's a valid tree, or false

\*/

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

// write your code here

if(n<=0 || edges==null) {

return false;

}

if(edges.length!=n-1) {

return false;

}

UnionFind uf = new UnionFind(n);

for(int[] edge : edges) {

int num1 = edge[0];

int num2 = edge[1];

uf.connect(num1, num2);

}

return uf.getCount()==1;

}

}

**2.Union Find different styles: simple, path compression or path compression with weighted**

**Refer to**

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

**Union Find 思路**

初始化Union Find的father map，让每一个节点的初始parent指向自己（自己跟自己是一个Group）；在循环读取edge list时，查找两个节点的parent，如果相同，说明形成了环（Cycle），那么这便不符合树（Tree）的定义，反之，如果不相同，则将其中一个节点设为另一个的parent，继续循环。

此外还有需要注意的是对于vertex和edge的validation，|E| = |V| - 1，也就是要验证 edges.length == n - 1，如果该条件不满足，则Graph一定不是valid tree。

**Union Find Solution**

public class Solution {

class UnionFind {

// Implemented with array instead of HashMap for simplicity

int[] parent;

// Constructor

UnionFind (int n) {

parent = new int[n];

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

parent[i] = i;

}

}

public int find(int x) {

return compressed\_find(x);

}

public int compressed\_find(int x) {

int x\_parent = parent[x];

while (x\_parent != parent[x\_parent]) {

x\_parent = parent[x\_parent];

}

int temp = -1;

int t\_parent = parent[x];

while (t\_parent != parent[t\_parent]) {

temp = parent[t\_parent];

parent[t\_parent] = x\_parent;

t\_parent = temp;

}

return x\_parent;

}

public void union(int x, int y) {

int x\_parent = find(x);

int y\_parent = find(y);

if (x\_parent != y\_parent) {

parent[x\_parent] = y\_parent;

}

}

}

/\*\*

\* @param n an integer

\* @param edges a list of undirected edges

\* @return true if it's a valid tree, or false

\*/

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

// Validation of |V| - 1 = |E|

if (n - 1 != edges.length || n == 0) {

return false;

}

UnionFind uf = new UnionFind(n);

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

if (uf.find(edges[i][0]) == uf.find(edges[i][1])) {

return false;

}

uf.union(edges[i][0], edges[i][1]);

}

return true;

}

}

**Simplified Union Find (0ms 100%)**

class Solution {

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

if (edges.length != n - 1) return false;

int[] parent = new int[n];

Arrays.fill(parent, -1);

for (int[] edge : edges) {

int x = find(edge[0], parent);

int y = find(edge[1], parent);

if (x == y) return false;

parent[x] = y;

}

return true;

}

int find(int node, int[] parent) {

if (parent[node] == -1) return node;

return parent[node] = find(parent[node], parent);

}

}

**Union Find - with path compression and weighted (1ms, 74.31%)**

class Solution {

public class UnionFind {

int[] parent;

int[] rank;

int count;

public UnionFind(int n) {

parent = new int[n];

rank = new int[n];

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

parent[i] = i;

rank[i] = 1;

}

count = n;

}

public int find (int p) {

int root = p;

while (parent[p] != p) {

parent[p] = parent[parent[p]];

p = parent[p];

}

return p;

}

public void union(int a, int b) {

int rootA = find(a);

int rootB = find(b);

if (rootA == rootB) {

return;

}

if (rank[rootA] > rank[rootB]) {

parent[rootB] = rootA;

rank[rootA] += rank[rootB];

} else {

parent[rootA] = rootB;

rank[rootB] += rank[rootA];

}

count--;

}

public int getCount() {

return count;

}

}

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

UnionFind uf = new UnionFind(n);

for (int[] edge: edges) {

if (uf.find(edge[0]) == uf.find(edge[1])) {

return false;

}

uf.union(edge[0], edge[1]);

}

return uf.getCount() == 1;

}

}