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.

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`.

1. Given `n = 5` and `edges = [[0, 1], [1, 2], [3, 4]]`, what should your return? Is this case a valid tree?
2. According to the definition of tree on Wikipedia: “a tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a 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`.

Solution 1

```java
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        // initialize n isolated islands
        int[] nums = new int[n];
        Arrays.fill(nums, -1);

        // perform union find
        for (int i = 0; i < edges.length; i++) {
            int x = find(nums, edges[i][0]);
            int y = find(nums, edges[i][1]);

            // if two vertices happen to be in the same set
            // then there's a cycle
            if (x == y) return false;

            // union
            nums[x] = y;
        }

        return edges.length == n - 1;
    }

    int find(int nums[], int i) {
        if (nums[i] == -1) return i;
        return find(nums, nums[i]);
    }
}
```

written by jeantimex original link here

Solution 2

There are so many different approaches and so many different ways to implement each. I find it hard to decide, so here are several :-)

In all of them, I check one of these tree characterizations:

- Has n-1 edges and is acyclic.
- Has n-1 edges and is connected.

Solution 1 ... Union-Find

The test cases are small and harmless, simple union-find suffices (runs in about 50~60 ms).

```
def validTree(self, n, edges):
    parent = range(n)
    def find(x):
        return x if parent[x] == x else find(parent[x])
    def union(xy):
        x, y = map(find, xy)
        parent[x] = y
        return x != y
    return len(edges) == n-1 and all(map(union, edges))
```

A version without using `all(...)`, to be closer to other programming languages:

```
def validTree(self, n, edges):
    parent = range(n)
    def find(x):
        return x if parent[x] == x else find(parent[x])
    for e in edges:
        x, y = map(find, e)
        if x == y:
            return False
        parent[x] = y
    return len(edges) == n - 1
```

A version checking `len(edges) != n - 1` first, as `parent = range(n)` could fail for huge `n`:

```
def validTree(self, n, edges):
    if len(edges) != n - 1:
        return False
    parent = range(n)
    def find(x):
        return x if parent[x] == x else find(parent[x])
    def union(xy):
        x, y = map(find, xy)
        parent[x] = y
        return x != y
    return all(map(union, edges))
```

Solution 2 ... DFS

```
def validTree(self, n, edges):
    neighbors = {i: [] for i in range(n)}
    for v, w in edges:
        neighbors[v] += w,
        neighbors[w] += v,
    def visit(v):
        map(visit, neighbors.pop(v, []))
    visit(0)
    return len(edges) == n-1 and not neighbors
```

Or check the number of edges first, to be faster and to survive unreasonably huge `n`:

```
def validTree(self, n, edges):
    if len(edges) != n - 1:
        return False
    neighbors = {i: [] for i in range(n)}
    for v, w in edges:
        neighbors[v] += w,
        neighbors[w] += v,
    def visit(v):
        map(visit, neighbors.pop(v, []))
    visit(0)
    return not neighbors
```

For an iterative version, just replace the three "visit" lines with

```
    stack = [0]
    while stack:
        stack += neighbors.pop(stack.pop(), [])
```

Solution 3 ... BFS

Just like DFS above, but replace the three "visit" lines with

```
    queue = [0]
    for v in queue:
        queue += neighbors.pop(v, [])
```

or, since that is not guaranteed to work, the safer

```
    queue = collections.deque([0])
    while queue:
        queue.extend(neighbors.pop(queue.popleft(), []))
```

written by StefanPochmann original link here

Solution 3

```
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        // initialize adjacency list
        List<List<Integer>> adjList = new ArrayList<List<Integer>>(n);

        // initialize vertices
        for (int i = 0; i < n; i++)
            adjList.add(i, new ArrayList<Integer>());

        // add edges
        for (int i = 0; i < edges.length; i++) {
            int u = edges[i][0], v = edges[i][1];
            adjList.get(u).add(v);
            adjList.get(v).add(u);
        }

        boolean[] visited = new boolean[n];

        // make sure there's no cycle
        if (hasCycle(adjList, 0, visited, -1))
            return false;

        // make sure all vertices are connected
        for (int i = 0; i < n; i++) {
            if (!visited[i])
                return false;
        }

        return true;
    }

    // check if an undirected graph has cycle started from vertex u
    boolean hasCycle(List<List<Integer>> adjList, int u, boolean[] visited, int pa
rent) {
        visited[u] = true;

        for (int i = 0; i < adjList.get(u).size(); i++) {
            int v = adjList.get(u).get(i);

            if ((visited[v] && parent != v) || (!visited[v] && hasCycle(adjList,
v, visited, u)))
                return true;
        }

        return false;
    }
}
```

written by jeantimex original link here