Minimum Height Trees

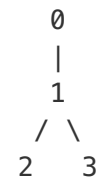
For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format
The graph contains `n` nodes which are labeled from `0` to `n - 1`. You will be given the number `n` and a list of undirected `edges` (each edge is a pair of labels).

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`.

Example 1:

Given `n = 4`, `edges = [[1, 0], [1, 2], [1, 3]]`



return `[1]`

Example 2:

Given `n = 6`, `edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]`

return `[3, 4]`

1. How many MHTs can a graph have at most?

Note:

(1) 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.”

(2) The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

See here for a better view

First let's review some statement for tree in graph theory:

> (1) A tree is an undirected graph in which any two vertices are connected by exactly one path.
>
> (2) Any connected graph who has `n` nodes with `n-1` edges is a tree.
>
> (3) The degree of a vertex of a graph is the number of edges incident to the vertex.
>
> (4) A leaf is a vertex of degree 1. An internal vertex is a vertex of degree at least 2.
>
> (5) A path graph is a tree with two or more vertices that is not branched at all.
>
> (6) A tree is called a rooted tree if one vertex has been designated the root.
>
> (7) The height of a rooted tree is the number of edges on the longest downward path between root and a leaf.

OK. Let's stop here and look at our problem.

Our problem want us to find the minimum height trees and return their root labels. First we can think about a simple case -- a path graph.

For a path graph of `n` nodes, find the minimum height trees is trivial. Just designate the middle point(s) as roots.

Despite its triviality, let design a algorithm to find them.

Suppose we don't know `n`, nor do we have random access of the nodes. We have to traversal. It is very easy to get the idea of two pointers. One from each end and move at the same speed. When they meet or they are one step away, (depends on the parity of `n`), we have the roots we want.

This gives us a lot of useful ideas to crack our real problem.

For a tree we can do some thing similar. We start from every end, by end we mean vertex of degree 1 (aka leaves). We let the pointers move the same speed. When two pointers meet, we keep only one of them, until the last two pointers meet or one step away we then find the roots.

It is easy to see that the last two pointers are from the two ends of the longest path in the graph.

The actual implementation is similar to the BFS topological sort. Remove the leaves, update the degrees of inner vertexes. Then remove the new leaves. Doing so level by level until there are 2 or 1 nodes left. What's left is our answer!

The time complexity and space complexity are both O(n).

Note that for a tree we always have `V = n`, `E = n-1`.

Java

```java
public List<Integer> findMinHeightTrees(int n, int[][] edges) {
    if (n == 1) return Collections.singletonList(0);

    List<Set<Integer>> adj = new ArrayList<>(n);
    for (int i = 0; i < n; ++i) adj.add(new HashSet<>());
    for (int[] edge : edges) {
        adj.get(edge[0]).add(edge[1]);
        adj.get(edge[1]).add(edge[0]);
    }

    List<Integer> leaves = new ArrayList<>();
    for (int i = 0; i < n; ++i)
        if (adj.get(i).size() == 1) leaves.add(i);

    while (n > 2) {
        n -= leaves.size();
        List<Integer> newLeaves = new ArrayList<>();
        for (int i : leaves) {
            int j = adj.get(i).iterator().next();
            adj.get(j).remove(i);
            if (adj.get(j).size() == 1) newLeaves.add(j);
        }
        leaves = newLeaves;
    }
    return leaves;
}

// Runtime: 53 ms
```

Python

```python
def findMinHeightTrees(self, n, edges):
    if n == 1: return [0]
    adj = [set() for _ in xrange(n)]
    for i, j in edges:
        adj[i].add(j)
        adj[j].add(i)

    leaves = [i for i in xrange(n) if len(adj[i]) == 1]

    while n > 2:
        n -= len(leaves)
        newLeaves = []
        for i in leaves:
            j = adj[i].pop()
            adj[j].remove(i)
            if len(adj[j]) == 1: newLeaves.append(j)
        leaves = newLeaves
    return leaves

# Runtime : 104ms
```

written by dietpepsi original link here

Solution 2

The basic idea is **"keep deleting leaves layer-by-layer, until reach the root."**

Specifically, first find all the leaves, then remove them. After removing, some nodes will become new leaves. So we can continue remove them. Eventually, there is only 1 or 2 nodes left. If there is only one node left, it is the root. If there are 2 nodes, either of them could be a possible root.

Time Complexity: Since each node will be removed at most once, the complexity is **O(n)**.

Thanks for pointing out any mistakes.

Updates: More precisely, if the number of nodes is V, and the number of edges is E. The space complexity is O(V+2E), for storing the whole tree. The time complexity is O(E), because we gradually remove all the neighboring information. As some friends pointing out, for a tree, if V=n, then E=n-1. Thus both time complexity and space complexity become O(n).

```
class Solution {
public:

    struct Node
    {
        unordered_set<int> neighbor;
        bool isLeaf()const{return neighbor.size()==1;}
    };

    vector<int> findMinHeightTrees(int n, vector<pair<int, int>>& edges) {

        vector<int> buffer1;
        vector<int> buffer2;
        vector<int>* pB1 = &buffer1;
        vector<int>* pB2 = &buffer2;
        if(n==1)
        {
            buffer1.push_back(0);
            return buffer1;
        }
        if(n==2)
        {
            buffer1.push_back(0);
            buffer1.push_back(1);
            return buffer1;
        }

        // build the graph
        vector<Node> nodes(n);
        for(auto p:edges)
        {
            nodes[p.first].neighbor.insert(p.second);
            nodes[p.second].neighbor.insert(p.first);
        }
```

```
            // find all leaves
            for(int i=0; i<n; ++i)
            {
                if(nodes[i].isLeaf()) pB1->push_back(i);
            }

            // remove leaves layer-by-layer
            while(1)
            {
                for(int i : *pB1)
                {
                    for(auto n: nodes[i].neighbor)
                    {
                        nodes[n].neighbor.erase(i);
                        if(nodes[n].isLeaf()) pB2->push_back(n);
                    }
                }
                if(pB2->empty())
                {
                    return *pB1;
                }
                pB1->clear();
                swap(pB1, pB2);
            }

        }
    };
```

written by TTester original link here

Solution 3

I am sharing two of my solutions, one is based on the longest path, and the other is related to Tree DP.

Longest Path

It is easy to see that the root of an MHT has to be the middle point (or two middle points) of the longest path of the tree. Though multiple longest paths can appear in an unrooted tree, they must share the same middle point(s).

Computing the longest path of a unrooted tree can be done, in O(n) time, by tree dp, or simply 2 tree traversals (dfs or bfs). The following is some thought of the latter.

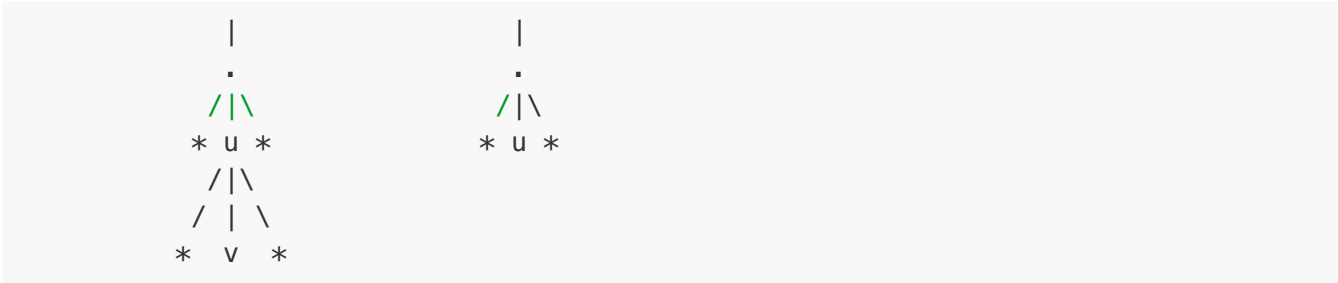
Randomly select a node x as the root, do a dfs/bfs to find the node y that has the longest distance from x. Then y must be one of the endpoints on some longest path. Let y the new root, and do another dfs/bfs. Find the node z that has the longest distance from y.

Now, the path from y to z is the longest one, and thus its middle point(s) is the answer. Java Solution

Tree DP

Alternatively, one can solve this problem directly by tree dp. Let dp[i] be the height of the tree when the tree root is i. We compute dp[0] ... dp[n - 1] by tree dp in a dfs manner.

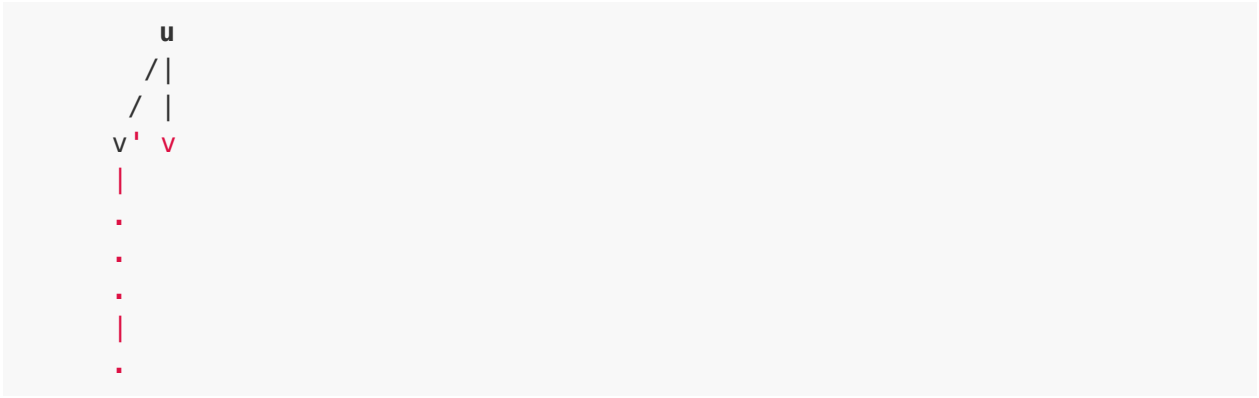
Arbitrarily pick a node, say node 0, as the root, and do a dfs. When we reach a node u, and let T be the subtree by removing all u's descendant (see the right figure below). We maintain a variable acc that keeps track of the length of the longest path in T with one endpoint being u. Then dp[u] = max(height[u], acc) Note, acc is 0 for the root of the tree.



. denotes a single node, and * denotes a subtree (possibly empty).

Now it remains to calculate the new acc for any of u's child, v. It is easy to see that the new acc is the max of the following

1. acc + 1 --- extend the previous path by edge uv;
2. max(height[v'] + 2), where v != v' --- see below for an example.



In fact, the second case can be computed in O(1) time instead of spending a time proportional to the degree of u. Otherwise, the runtime can be quadratic when the degree of some node is Omega(n). The trick here is to maintain two heights of each node, the largest height (the conventional height), and the second largest height (the height of the node after removing the branch w.r.t. the largest height).

Therefore, after the dfs, all dp[i]'s are computed, and the problem can be answered trivially. The total runtime is still O(n). Java Solution

written by lixx2100 original link here