Minimum Diameter after merging two trees

Wang Xiyu

24 Dec 2024

Problem Statement

Given 2 undirected unweighted trees, find the minimal possible diameter of a new tree formed by connecting the two trees with exactly one new edge.

Problem Constraints

Both graphs are trees, meaning both are connected and acyclic.

Concept Proof

Theorem 0.1. *The diameter of the new merged tree is at least the diameters of both original trees.*

Proof. Suppose not, the diameter of one tree G is k, a path of length k between v_1 to v_2. By merging, there exists a new path $v_1 \vdash^* v_2$ that is of length l, that $l < k$. Now there exist a cycle in G, $v_1 \vdash^* v_2 \vdash^* v_1$ that is of length $l + k$. The resulting graph is not a tree, a contradiction. □

Theorem 0.2. *Given the diameter of tree 1 k_1 and diameter of tree k_2, where $max(k_1, k_2) \leq 2 \times min(k_1, k_2)$ the minimal diameter of the merged tree is $max(k_1, k_2, \lceil \frac{k_1}{2} \rceil + \lceil \frac{k_2}{2} \rceil + 1)$*

Proof. Consider one path between v and v', $v, v' \in V_1$ in tree 1 with diameter k_1 in tree 1, find the center of the path v'', $v \vdash^{\lceil \frac{k_1}{2} \rceil} v'' \wedge v'' \vdash^{\lfloor \frac{k_1}{2} \rfloor} v'$. Do the same on the other tree with diameter k_2 A new edge will be added between v'' and a node in the other tree, the diameter of the merged tree will become the larger between $\lceil \frac{k_1}{2} \rceil + max(\lfloor \frac{k_1}{2} \rfloor, \lceil \frac{k_2}{2} \rceil + 1)$ and $\lceil \frac{k_2}{2} \rceil + max(\lfloor \frac{k_2}{2} \rfloor, \lceil \frac{k_1}{2} \rceil + 1)$, corresponding to the longer by 1 halves of diameter path of both trees, plus the new edge joining them together. □

Theorem 0.3. *The shortest longest path in a tree both starts and ends in a leaf node.*

Proof. Suppose not, a shortest longest path (v, v') ends in a non-leaf node v', $\exists v'' \in V, v''$ is a leaf and $v' \vdash^* v''$. By definition of trees, a leaf has at most one parent, $!\exists e \in E, e = (v, v''), len(e) = len((v, v')) + len((v', v''))$. With no negatively weighted edges, $len((v, v'')) > len((v, v')) + len((v', v''))$, (v, v') is not a shortest longest path. Contradiction. □

Lemma 0.4. *The diameter of a tree can be found by performing 2 DFS.*

Proof. Both ends of a shortest longest path are a leaf, and in a tree every other node is reachable from any node via a unique path (trivial). Therefore a longest possible shortest longest path in a tree, is the longest shortest path walkable from one leaf to another leaf. By performing one DFS, we can find a leaf in $O(V)$ time, and another DFS starting from the leaf found in the first DFS can find the length of the shortest path from the leaf to every other node; among which the longest is the diameter of the tree. □

Solution

```
class Solution {
    public:
        int minimumDiameterAfterMerge(vector<vector<int>>& edges1,
                                      vector<vector<int>>& edges2) {
            int dia1 = findDiameter(edges1);
            int dia2 = findDiameter(edges2);
            int a = ceil(dia1/2.0);
            int b = ceil(dia2/2.0);
            return max({dia1, dia2, (1 + a + b)});
        }
        int findDiameter(vector<vector<int>>& edges) {
            if (edges.empty()) {
                return 0;
            }
            int len = edges.size() + 1;
            vector<vector<int>> adj = processAdj(edges, len); // process input to form adjacency
                 list
            if (adj.empty() || (adj.size() == 1 && adj[0].empty())) {
                return 0; // Single node or empty adjacency list
            }
            vector<bool> visited(len, false); // initialize visited array
            stack<pair<int, int>> s;
            s.push({0, 0});
            int depth = 0;
            int furthest = 0;
            while (!s.empty()) {
                pair<int, int> n = s.top();
                s.pop();
                int currNode = n.first;
                int currLevel = n.second;
                if (!visited[currNode]) {
                    visited[currNode] = true;
                    if (currLevel > depth) {
                        depth = currLevel;
                        furthest = currNode;
                    }
                    for (int i = 0; i < adj[currNode].size(); i++) {
                        s.push(make_pair(adj[currNode][i], currLevel + 1));
                    }
                }
            }
            vector<bool> visited2(len, false);
            s.push({furthest, 0});
            int diameter = 0;
            while(!s.empty()) {
                pair<int, int> n = s.top();
                s.pop();
                int currNode = n.first;
                int currLevel = n.second;
                if (!visited2[currNode]) {
                    visited2[currNode] = true;
                    diameter = max(diameter, currLevel);
                    if (adj[currNode].size() > 0) {
                        for (int i = 0; i < adj[currNode].size(); i++) {
                            s.push(make_pair(adj[currNode][i], currLevel + 1));
                        }
```

```
                }
            }
        }
        return diameter;
    }
    vector<vector<int>> processAdj(vector<vector<int>>& edges, int len) {
        vector<vector<int>> adj(len);
        if (len == 1) {
            return adj;
        }
        for (int i = 0; i < edges.size(); i++) {
            adj[edges[i][0]].push_back(edges[i][1]);
            adj[edges[i][1]].push_back(edges[i][0]);
        }
        return adj;
    }
};
```