Subway Solution

In this problem, our aim is to transform one tree into another, using as few edge moves as possible, such that the tree is still connected (i.e., still a tree) at each step. We start by noting that a lower bound on the number of edge moves is given by the difference between the trees: if an edge is in the first tree but not the second, we certainly need to move it at some point. Similarly, if an edge is in the second tree but not the first, we need to move some edge there eventually.

Let us denote the first set of edges by S, the second by T, and the common edges between the trees by C. Then $|S| = |T| = n - 1 - |C|$, and we have a lower bound on the number of edge moves of $|S|$. We will show that this bound is attainable.

Consider any edge e in S. Removing it would disconnect the tree: it would get two sides L and R. Since the second tree spans the whole graph, it must have some edge e_2 passing between L and R. Replacing e by e_2 would then preserve connectivity of the graph, while reducing the size of S by one. We can repeat this until S is empty to get an optimal tree transformation.

The problem then becomes: how can we efficiently find such an e_2?

A cubic solution

We can pick $e \in S$ and $e_2 \in T$ at random, and check if the tree is still connected when replacing e by e_2. This happens with probability approximately $1/n$, and connectivity check takes time $O(n)$[1] Since we replace $n - 1$ in the worst case, the complexity becomes $O(n^3)$ on average.

A quadratic solution

After removing an arbitrary edge $e \in S$, we can perform a depth-first traversal starting from an arbitrary node to find the two sides of the graph. Then for each $e_2 \in T$ we can check in constant time if its two vertices lie on different sides. This results in complexity $O(n^2)$.

An $O(n \log n)$ solution

To find pairs of edges e, e_2 quicker, we will restrict our choice of e a bit. If we could pick e adjacent to a leaf in the graph, we could pick as e_2 an edge adjacent to the same leaf, and we would automatically get connectivity. This we could implement using just vectors of edges adjacent to all nodes, and a queue containing nodes that have become leaf nodes.

Unfortunately, there are not always such edges, so we will have to be slightly smarter. If two nodes are connected via edges in C (shared between the two trees), they will continue to be connected even after edge movements. Hence, we can look at the induced tree where such edges are collapsed. Any edge in

[1] A link-cut tree or similar fancy data structure could reduce this to $O(\log n)$. This is also what was used in the output validator for this problem.

this tree corresponds to an edge in S. As long as the tree is non-trivial, it has at least one leaf, and we can use the method from above. Implementation-wise, we can keep a union-find structure with all components that we have collapsed together using edges from C. When we merge two components, we merge their corresponding lists of outgoing edges. See the reference solution for more detail.