Paint the edges of the tree

Table of Contents

This is a fairly common task. Given a tree G with N vertices. There are two types of queries: the first one is to paint an edge, the second one is to query the number of colored edges between two vertices.

Here we will describe a fairly simple solution (using a segment tree) that will answer each query in $O(\log N)$ time. The preprocessing step will take $O(N)$ time.

Algorithm

First, we need to find the LCA to reduce each query of the second kind (i, j) into two queries (l, i) and (l, j), where l is the LCA of i and j. The answer of the query (i, j) will be the sum of both subqueries. Both these queries have a special structure, the first vertex is an ancestor of the second one. For the rest of the article we will only talk about these special kind of queries.

We will start by describing the **preprocessing** step. Run a depth-first search from the root of the tree and record

the Euler tour of this depth-first search (each vertex is added to the list when the search visits it first and every time we return from one of its children). The same technique can be used in the LCA preprocessing.

This list will contain each edge (in the sense that if i and j are the ends of the edge, then there will be a place in the list where i and j are neighbors in the list), and it appear exactly two times: in the forward direction (from i to j, where vertex i is closer to the root than vertex j) and in the opposite direction (from j to i).

We will build two lists for these edges. The first one will store the color of all edges in the forward direction, and the second one the color of all edges in the opposite direction. We will use 1 if the edge is colored, and 0 otherwise. Over these two lists we will build each a segment tree (for sum with a single modification), let's call them $T1$ and $T2$.

Let us answer a query of the form (i, j), where i is the ancestor of j. We need to determine how many edges are painted on the path between i and j. Let's find i and j in the Euler tour for the first time, let it be the positions p and q (this can be done in $O(1)$ if we calculate these positions in advance during preprocessing). Then the **answer** to the query is the sum $T1[p..q-1]$ minus the sum $T2[p..q-1]$.

Why? Consider the segment $[p; q]$ in the Euler tour. It contains all edges of the path we need from i to j but also contains a set of edges that lie on other paths from

i. However there is one big difference between the edges we need and the rest of the edges: the edges we need will be listed only once in the forward direction, and all the other edges appear twice: once in the forward and once in the opposite direction. Hence, the difference $T1[p..q-1] - T2[p..q-1]$ will give us the correct answer (minus one is necessary because otherwise, we will capture an extra edge going out from vertex j). The sum query in the segment tree is executed in $O(\log N)$.

Answering the **first type of query** (painting an edge) is even easier - we just need to update $T1$ and $T2$, namely to perform a single update of the element that corresponds to our edge (finding the edge in the list, again, is possible in $O(1)$, if you perform this search during preprocessing). A single modification in the segment tree is performed in $O(logN)$.

Implementation

Here is the full implementation of the solution, including LCA computation:

```
const int INF = 1000 * 1000 * 1000;

typedef vector<vector<int>> graph;

vector<int> dfs_list;
vector<int> edges_list;
vector<int> h;
```

```
void dfs(int v, const graph& g, const graph& e
    h[v] = cur_h;
    dfs_list.push_back(v);
    for (size_t i = 0; i < g[v].size(); ++i) {
        if (h[g[v][i]] == -1) {
            edges_list.push_back(edge_ids[v][i
            dfs(g[v][i], g, edge_ids, cur_h +
            edges_list.push_back(edge_ids[v][i
            dfs_list.push_back(v);
        }
    }
}

vector<int> lca_tree;
vector<int> first;

void lca_tree_build(int i, int l, int r) {
    if (l == r) {
        lca_tree[i] = dfs_list[l];
    } else {
        int m = (l + r) >> 1;
        lca_tree_build(i + i, l, m);
        lca_tree_build(i + i + 1, m + 1, r);
        int lt = lca_tree[i + i], rt = lca_tre
        lca_tree[i] = h[lt] < h[rt] ? lt : rt;
    }
}

void lca_prepare(int n) {
    lca_tree.assign(dfs_list.size() * 8, -1);
    lca_tree_build(1, 0, (int)dfs_list.size()

    first.assign(n, -1);
```

```
    for (int i = 0; i < (int)dfs_list.size();
        int v = dfs_list[i];
        if (first[v] == -1)
            first[v] = i;
    }
}

int lca_tree_query(int i, int tl, int tr, int
    if (tl == l && tr == r)
        return lca_tree[i];
    int m = (tl + tr) >> 1;
    if (r <= m)
        return lca_tree_query(i + i, tl, m, l,
    if (l > m)
        return lca_tree_query(i + i + 1, m + 1
    int lt = lca_tree_query(i + i, tl, m, l, m
    int rt = lca_tree_query(i + i + 1, m + 1,
    return h[lt] < h[rt] ? lt : rt;
}

int lca(int a, int b) {
    if (first[a] > first[b])
        swap(a, b);
    return lca_tree_query(1, 0, (int)dfs_list.
}

vector<int> first1, first2;
vector<char> edge_used;
vector<int> tree1, tree2;

void query_prepare(int n) {
    first1.resize(n - 1, -1);
    first2.resize(n - 1, -1);
```

```
    for (int i = 0; i < (int)edges_list.size()
        int j = edges_list[i];
        if (first1[j] == -1)
            first1[j] = i;
        else
            first2[j] = i;
    }

    edge_used.resize(n - 1);
    tree1.resize(edges_list.size() * 8);
    tree2.resize(edges_list.size() * 8);
}

void sum_tree_update(vector<int>& tree, int i,
    tree[i] += delta;
    if (l < r) {
        int m = (l + r) >> 1;
        if (j <= m)
            sum_tree_update(tree, i + i, l, m,
        else
            sum_tree_update(tree, i + i + 1, m
    }
}

int sum_tree_query(const vector<int>& tree, in
    if (l > r || tl > tr)
        return 0;
    if (tl == l && tr == r)
        return tree[i];
    int m = (tl + tr) >> 1;
    if (r <= m)
        return sum_tree_query(tree, i + i, tl,
    if (l > m)
```

```
        return sum_tree_query(tree, i + i + 1,
    return sum_tree_query(tree, i + i, tl, m,
            sum_tree_query(tree, i + i + 1, m +
}

int query(int v1, int v2) {
    return sum_tree_query(tree1, 1, 0, (int)ed
            sum_tree_query(tree2, 1, 0, (int)ed
}

int main() {
    // reading the graph
    int n;
    scanf("%d", &n);
    graph g(n), edge_ids(n);
    for (int i = 0; i < n - 1; ++i) {
        int v1, v2;
        scanf("%d%d", &v1, &v2);
        --v1, --v2;
        g[v1].push_back(v2);
        g[v2].push_back(v1);
        edge_ids[v1].push_back(i);
        edge_ids[v2].push_back(i);
    }

    h.assign(n, -1);
    dfs(0, g, edge_ids);
    lca_prepare(n);
    query_prepare(n);

    for (;;) {
        if () {
            // request for painting edge x;
```

```
                // if start = true, then the edge
                // is removed
                edge_used[x] = start;
                sum_tree_update(tree1, 1, 0, (int)
                                start ? 1 : -1);
                sum_tree_update(tree2, 1, 0, (int)
                                start ? 1 : -1);
            } else {
                // query the number of colored edg
                int l = lca(v1, v2);
                int result = query(l, v1) + query(
                // result - the answer to the requ
            }
        }
    }
```