Lowest Common Ancestor - Binary Lifting

Table of Contents

Let G be a tree. For every query of the form `(u, v)` we want to find the lowest common ancestor of the nodes `u` and `v`, i.e. we want to find a node `w` that lies on the path from `u` to the root node, that lies on the path from `v` to the root node, and if there are multiple nodes we pick the one that is farthest away from the root node. In other words the desired node `w` is the lowest ancestor of `u` and `v`. In particular if `u` is an ancestor of `v`, then `u` is their lowest common ancestor.

The algorithm described in this article will need $O(N \log N)$ for preprocessing the tree, and then $O(\log N)$ for each LCA query.

Algorithm

For each node we will precompute its ancestor above him, its ancestor two nodes above, its ancestor four above, etc. Let's store them in the array `up`, i.e. `up[i]` `[j]` is the `2^j`-th ancestor above the node `i` with `i=1...N`, `j=0...ceil(log(N))`. These information allow us to jump from any node to any ancestor above it in $O(\log N)$ time. We can compute this array using a DFS traversal of the tree.

For each node we will also remember the time of the first visit of this node (i.e. the time when the DFS discovers the node), and the time when we left it (i.e. after we visited all children and exit the DFS function). We can use this information to determine in constant time if a node is an ancestor of another node.

Suppose now we received a query `(u, v)`. We can immediately check whether one node is the ancestor of the other. In this case this node is already the LCA. If `u` is not the ancestor of `v`, and `v` not the ancestor of `u`, we climb the ancestors of `u` until we find the highest (i.e. closest to the root) node, which is not an ancestor of `v` (i.e. a node `x`, such that `x` is not an ancestor of `v`, but `up[x][0]` is). We can find this node `x` in $O(\log N)$ time using the array `up`.

We will describe this process in more detail. Let `L = ceil(log(N))`. Suppose first that `i = L`. If `up[u][i]` is not an ancestor of `v`, then we can assign `u = up[u][i]` and decrement `i`. If `up[u][i]` is not an ancestor, then we just decrement `i`. Clearly after doing this for all non-negative `i` the node `u` will be the desired node - i.e. `u` is still not an ancestor of `v`, but `up[u][0]` is.

Now, obviously, the answer to LCA will be `up[u][0]` - i.e., the smallest node among the ancestors of the node `u`, which is also an ancestor of `v`.

So answering a LCA query will iterate `i` from `ceil(log(N))` to `0` and checks in each iteration if one node is the ancestor of the other. Consequently each query can be answered in $O(\log N)$.

Implementation

```
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];

    for (int u : adj[v]) {
        if (u != p)
            dfs(u, v);
    }

    tout[v] = ++timer;
}
```

```
bool is_ancestor(int u, int v)
{
    return tin[u] <= tin[v] && tout[u] >= tout
}

int lca(int u, int v)
{
    if (is_ancestor(u, v))
        return u;
    if (is_ancestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
    }
    return up[u][0];
}

void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root);
}
```