Lowest Common Ancestor - Tarjan's off-line algorithm

Table of Contents

We have a tree G with n nodes and we have m queries of the form (u, v). For each query (u, v) we want to find the lowest common ancestor of the vertices u and v, i.e. the node that is an ancestor of both u and v and has the greatest depth in the tree. The node v is also an ancestor of v, so the LCA can also be one of the two nodes.

In this article we will solve the problem off-line, i.e. we assume that all queries are known in advance, and we therefore answer the queries in any order we like. The following algorithm allows to answer all m queries in $O(n + m)$ total time, i.e. for sufficiently large m in $O(1)$ for each query.

Algorithm

The algorithm is named after Robert Tarjan, who discovered it in 1979 and also made many other contributions to the Disjoint Set Union data structure, which will be heavily used in this algorithm.

The algorithm answers all queries with one DFS traversal of the tree. Namely a query (u, v) is answered at node u, if node v has already been visited previously, or vice versa.

So let's assume we are currently at node v, we have already made recursive DFS calls, and also already visited the second node u from the query (u, v). Let's learn how to find the LCA of these two nodes.

Note that $\mathrm{LCA}(u, v)$ is either the node v or one of its ancestors. So we need to find the lowest node among the ancestors of v (including v), for which the node u is a descendant. Also note that for a fixed v the visited nodes of the tree split into a set of disjoint sets. Each ancestor p of node v has his own set containing this node and all subtrees with roots in those of its children who are not part of the path from v to the root of the tree. The set which contains the node u determines the $\mathrm{LCA}(u, v)$: the LCA is the representative of the set, namely the node on lies on the path between v and the root of the tree.

We only need to learn to efficiently maintain all these sets. For this purpose we apply the data structure DSU. To be able to apply Union by rank, we store the real

representative (the value on the path between v and the root of the tree) of each set in the array `ancestor`.

Let's discuss the implementation of the DFS. Let's assume we are currently visiting the node v. We place the node in a new set in the DSU, `ancestor[v] = v`. As usual we process all children of v. For this we must first recursively call DFS from that node, and then add this node with all its subtree to the set of v. This can be done with the function `union_sets` and the following assignment `ancestor[find_set(v)] = v` (this is necessary, because `union_sets` might change the representative of the set).

Finally after processing all children we can answer all queries of the form (u, v) for which u has been already visited. The answer to the query, i.e. the LCA of u and v, will be the node `ancestor[find_set(u)]`. It is easy to see that a query will only be answered once.

Let's us determine the time complexity of this algorithm. Firstly we have $O(n)$ because of the DFS. Secondly we have the function calls of `union_sets` which happen n times, resulting also in $O(n)$. And thirdly we have the calls of `find_set` for every query, which gives $O(m)$. So in total the time complexity is $O(n + m)$, which means that for sufficiently large m this corresponds to $O(1)$ for answering one query.

Implementation

Here is an implementation of this algorithm. The implementation of DSU has been not included, as it can be used without any modifications.

```
vector<vector<int>> adj;
vector<vector<int>> queries;
vector<int> ancestor;
vector<bool> visited;

void dfs(int v)
{
    visited[v] = true;
    ancestor[v] = v;
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs(u);
            union_sets(v, u);
            ancestor[find_set(v)] = v;
        }
    }
    for (int other_node : queries[v]) {
        if (visited[other_node])
            cout << "LCA of " << v << " and "
                 << " is " << ancestor[find_se
    }
}

void compute_LCAs() {
    // initialize n, adj and DSU
    // for (each query (u, v)) {
    //    queries[u].push_back(v);
    //    queries[v].push_back(u);
```

```
    // }

    ancestor.resize(n);
    visited.assign(n, false);
    dfs(0);
}
```