added: 10 Jun 2008 22:30
Edited: 31 Dec 2011 1:58

Lowest common ancestor. Finding for the O (sqrt (N)) and O (log N) with preprocessing O (N)

Contents [hide]

Suppose we are given a tree G. The input receives requests form (V1, V2), for each request is required to find their least common ancestor, ie a vertex V, which lies in the path from the root to V1, the path from the root to V2, and all of these peaks should be chosen lowermost. In other words, the required vertex V - ancestor and V1, and V2, and among all such common ancestors selected lower. It is obvious that the lowest common ancestor of vertices V1 and V2 - it is their common ancestor, which lies on the shortest path from V1 to V2. In particular, for example, if V1 is the ancestor of V2, then V1 is their least common ancestor.

In English, this problem is called the problem LCA - Least Common Ancestor.

The idea of the algorithm

Before responding, perform a so-called **preprocessing** . Start the tour in the depth of the root, which will build a list of visits vertices Order (current node is added to the list at the entrance to this mountain, and after each return of her son), it is easy to see that the final size of the list is O (N). And build an array of First [1..N], in which each vertex is specified position in the array of Order, which is this vertex, ie Order [First [I]] = I for all I. Also, using depth-first search will find the height of each node (the distance from the root to it) - H [1..N].

As it is now responding? Suppose we have a current request - a pair of vertices V1 and V2. Consider the list of Order between indices First [V1] and First [V2]. It is easy to notice that in this range will be required and LCA (V1, V2), as well as many other peaks. However, LCA (V1, V2) is different from the rest of the vertices so that it will be the vertex with the lowest height.

So, to answer your query, we just need to **find the vertex with the lowest altitude** in the array Order in the range between First [V1] and First [V2]. Thus, the **problem reduces to the LCA problem RMQ** ("minimum interval"). And the last problem is solved by data structures (see. task RMQ).

If you use the **sqrt-decomposition** , it is possible to obtain a solution that responds to the request for **O (sqrt (N))** and performs preprocessing for the O (N).

If you use the **segment tree** , you can get a solution to meet the request for the **O (log (N))** and performs preprocessing for the O (N).

Implementation

There will be given a ready implementation of LCA with wood segments:

```
typedef vector < vector<int> > graph;
typedef vector<int>::const_iterator const_graph_iter;


vector<int> lca_h, lca_dfs_list, lca_first, lca_tree;
vector<char> lca_dfs_used;

void lca_dfs (const graph & g, int v, int h = 1)
{
        lca_dfs_used[v] = true;
        lca_h[v] = h;
        lca_dfs_list.push_back (v);
        for (const_graph_iter i = g[v].begin(); i != g[v].end(); ++i)
```

```
                if (!lca_dfs_used[*i])
                {
                        lca_dfs (g, *i, h+1);
                        lca_dfs_list.push_back (v);
                }
}

void lca_build_tree (int i, int l, int r)
{
        if (l == r)
                lca_tree[i] = lca_dfs_list[l];
        else
        {
                int m = (l + r) >> 1;
                lca_build_tree (i+i, l, m);
                lca_build_tree (i+i+1, m+1, r);
                if (lca_h[lca_tree[i+i]] < lca_h[lca_tree[i+i+1]])
                        lca_tree[i] = lca_tree[i+i];
                else
                        lca_tree[i] = lca_tree[i+i+1];
        }
}

void lca_prepare (const graph & g, int root)
{
        int n = (int) g.size();
        lca_h.resize (n);
        lca_dfs_list.reserve (n*2);
        lca_dfs_used.assign (n, 0);

        lca_dfs (g, root);

        int m = (int) lca_dfs_list.size();
        lca_tree.assign (lca_dfs_list.size() * 4 + 1, -1);
        lca_build_tree (1, 0, m-1);

        lca_first.assign (n, -1);
        for (int i = 0; i < m; ++i)
        {
                int v = lca_dfs_list[i];
                if (lca_first[v] == -1)
                        lca_first[v] = i;
        }
}

you lca_tree_min (you s, you sl, sr're, you're, you're r)
{
        if (sl == l && sr == r)
                return lca_tree[i];
        int sm = (sl + sr) >> 1;
        if (r <= sm)
                return lca_tree_min (i+i, sl, sm, l, r);
        if (l > sm)
                return lca_tree_min (i+i+1, sm+1, sr, l, r);
        int ans1 = lca_tree_min (i+i, sl, sm, l, sm);
        int ans2 = lca_tree_min (i+i+1, sm+1, sr, sm+1, r);
        return lca_h[ans1] < lca_h[ans2] ? ans1 : ans2;
}

int lca (int a, int b)
{
        int left = lca_first[a],
                right = lca_first[b];
```

```
        if (left > right)  swap (left, right);
        return lca_tree_min (1, 0, (int)lca_dfs_list.size()-1, left, right);
}

int main()
{
        graph g;
        int root;
        Graph reading ... ...

        lca_prepare (g, root);

        for (;;)
        {
                int v1, v2; // Postupil request
                int v = lca (v1, v2); // Answer the request
        }
}
```