Added: 2 Mar 2009 17:45
Edited: 2 Mar 2009 17:45

Lowest common ancestor. Finding for $O(1)$offline (Tarjan algorithm)

Contents [hide]

Given a tree $G$with $n$vertices and given m query form (a_i, b_i). For each query (a_i, b_i)you want to find the lowest common ancestor of vertices $a_i$and b_i, ie a vertex c_i, which is farthest from the root of the tree, and at the same time is the ancestor of both vertices $a_i$and b_i.

We consider the problem in the offline mode, ie Considering that all requests are known in advance. Algorithm described below allows you to reply to all $m$queries for the total time $O(n + m)$, ie sufficiently large $m$for $O(1)$the request.

Tarjan's algorithm

The basis for the algorithm is a data structure "system of disjoint sets" , which was invented Tarjanne (Tarjan).

Algorithm is actually a detour into the depths of the root of the tree, in the which are gradually responding to requests. Namely, the response to the request $(v, u)$is when a detour to the depth is at the top u, and the top $v$has already been visited, or vice versa.

So let detour into the depths is at the top v(and have already been performed transitions in her sons), and found that for some query (v, u)vertex $u$already has had bypass in depth. Then learn how to find LCAthese two vertices.

Note that $\mathrm{LCA}(v, u)$is either the very top v, or one of its ancestors. So, we need to find the lowest vertex of the ancestors v(including itself it), for which the vertex $u$is a descendant. Note that for fixed $v$on such grounds (ie, what the smallest ancestor $v$is also an ancestor of some vertex) tree top tree split into a set of disjoint classes. For each ancestor $p \neq v$tops $v$its class contains the very top of this, as well as all the subtrees with roots in those of her sons, who are "left" of the way up v(ie, that were processed earlier than has been achieved v).

We need to learn how to effectively support all of these classes, for which we apply the data structure "system of disjoint sets." Each class will meet in the structure of the set, with the representative for this set, we define the quantity $\mathrm{ANCESTOR}$- that top p, which forms the class.

Let us consider in detail the implementation of bypass in depth. Suppose we are at a vertex v. Put it in a separate class in the structure of disjoint sets $\mathrm{ANCESTOR}[v] = v$. As usual crawled deep, iterate all outgoing edges (v, to). For each of these to, we first need to call the tour into the depths of this node, and then add the vertex with all its subtree in the top of the class v. This is realized by the operation $\mathrm{Union}$of the data structure "system of disjoint sets", with the subsequent installation $\mathrm{ANCESTOR} = v$for a representative set (as representative of the class after the merger could change). Finally, after processing all the edges, we iterate through all kinds of questions (v, u), and if $u$has been marked as visited detour into the depths, the response to this query will be a vertex $\mathrm{LCA}(v, u) = \mathrm{ANCESTOR}[\mathrm{FindSet}(u)]$. It is easy to see that for each

request, this condition (that one vertex is the current request, and the other has had before) is executed exactly once.

We estimate **the asymptotic behavior** . It consists of several parts. Firstly, it is the asymptotic behavior of traversal in depth, which in this case is $O(n)$. Secondly, this business combination sets that total for all reasonable $n$expend $O(n)$operations. Third, it is for each request verification of (twice on request) and the definition of the result (once at your request), each, again, for all reasonable $n$performed for $O(1)$. The final asymptotic behavior is obtained $O(n+m)$, which means that for sufficiently large m($n = O(m)$) response for $O(1)$one request.

Implementation

We give a full implementation of this algorithm, including a slightly modified (supporting $\mathrm{ANCESTOR}$) the implementation of a system of intersecting sets (randomized version).

```
const int MAXN = максимальное число вершин в графе;
vector<int> g[MAXN], q[MAXN]; // граф и все запросы
int dsu[MAXN], ancestor[MAXN];
bool u[MAXN];

int dsu_get (int v) {
        return v == dsu[v] ? v : dsu[v] = dsu_get (dsu[v]);
}

void dsu_unite (int a, int b, int new_ancestor) {
        a = dsu_get (a),  b = dsu_get (b);
        if (rand() & 1)  swap (a, b);
        dsu[a] = b,  ancestor[b] = new_ancestor;
}

void dfs (int v) {
        dsu[v] = v,  ancestor[v] = v;
        u[v] = true;
        for (size_t i=0; i<g[v].size(); ++i)
                if (!u[g[v][i]]) {
                        dfs (g[v][i]);
                        dsu_unite (v, g[v][i], v);
                }
        for (size_t i=0; i<q[v].size(); ++i)
                if (u[q[v][i]]) {
                        printf ("%d %d -> %d\n", v+1, q[v][i]+1,
                                ancestor[ dsu_get(q[v][i]) ]+1);
}

int main() {
        ... чтение графа ...

        // чтение запросов
        for (;;) {
                int a, b = ...; // очередной запрос
                --a, --b;
                q[a].push_back (b);
                q[b].push_back (a);
        }
```

```
        // обход в глубину и ответ на запросы
        dfs (0);
}
```