added: 24 Aug 2008
Edited: 30 Mar 2012

Lowest common ancestor. Finding the O (1) preprocessing with O (N) (algorithm Farah-Colton and Bender)

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 V 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 t 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.

Farah algorithm described here-Colton and Bender (Farach-Colton, Bender) is asymptotically optimal, and thus a relatively simple (compar to other algorithms, e.g., gate-Vishkina).

Algorithm

We use the classical reducing the problem of LCA **to RMQ problem** (at least on the interval) (see more details. lowest common ancestor. Looking for O (sqrt (N)) and O (log N) preprocessing with O (N)). Now learn how to solve the problem RMQ in this particular case with preprocessing O (N) and O (1) on the request.

Note that the problem RMQ, to which we have reduced the problem LCA, is very specific: any two adjacent elements in the array **differ by exactly one** (since the elements of the array - this is nothing more than the height of the vertices visited in the traversal, and we either go t child, then the next item will be 1 more or go to the ancestor, then the next item will be 1 less). Actually algorithm Farah-Colton and Bender is a solution of this problem RMQ.

Let A be the array over which queries are running RMQ, and N - the size of the array.

We first construct an algorithm that solves this problem **with preprocessing O (N log N) and O (1) on the request** . This is easy to do: cre a so-called Sparse Table T [l, i], where each element of T [l, i] is equal to the minimum A in the interval [l; l + 2^{i}). Obviously, 0 <= i <= ⌈log and therefore the size of Sparse Table is O (N log N). Build it is also easy for the O (N log N), if we note that T [l, i] = min (T [l, i-1], T [l + 2^{i-} 1]). As it is now to respond to every request RMQ in O (1)? Let received a request (l, r), then the answer would be min (T [l, sz], T [r-2^{sz} + sz]), where sz - the largest power of two not exceeding r-l + 1. Indeed, we seem to take the interval (l, r) and cover it with two runs of length ^{sz} - one starting at l, and the other ending in r (and these segments overlap, which in this case does not bother us). To really achieve the asymptotics O (1) on the request, we must predposchitat sz values for all possible lengths from 1 to N.

We will now describe **how to improve** this algorithm to the asymptotic behavior of O (N).

We divide the array A into blocks of size K = 0.5 $\log_2$ N. For each block, calculate the minimum element in him and his position (as for the solution of LCA are important to us not the lows, and their positions). Let B - is an array of size N / K, composed of these minima in each bl We construct the array B Sparse Table, as described above, the size Sparse Table and time of its construction will be:

```
N / K log N / K = (2N / log N) log (2N / log N) =
= (2N / log N) (1 + log (N / log N)) <= 2N / log N + 2N = O (N)
```

Now we just need to learn how to quickly respond to requests RMQ **within each block** . In fact, if the request comes RMQ (l, r), then if r an are in different blocks, the answer will be a minimum of the following values: a minimum block l, starting from l to the end of the block, then minimum units after l and up to r (not inclusive), and finally the minimum block r, from the beginning of the block to r. At the prompt "at least the" we can be responsible for the O (1) using the Sparse Table, there were only questions RMQ in blocks.

Here we use "+ -1 property." Note that, if within each block of each of its elements take the first element, then all blocks will be uniquely determined by the sequence of length K-1, consisting of the numbers ± 1. Consequently, the amount of the various blocks will be equal to:

$$2^{K-1} = 2^{0.5 \log N - 1} = 0.5 \text{ sqrt } (N)$$

Thus, the number of different blocks will be O (sqrt (N)), and therefore we can predposchitat RMQ results in all of the different units of the C (sqrt (N) K^2) = O (sqrt (N) $\log^2$ N) = O (N). In terms of implementation, we can characterize each block bit mask of length K-1 (which obviously fits into standard type int), and stored in a predposchitannye RMQ array R [mask, l, r] size O (sqrt (N) $\log^2$ N).

So, we have learned predposchityvat RMQ results within each block, as well as by the RMQ over blocks, all for a total of O (N), and respon every request RMQ in O (1) - using only the precomputed values in the worst case four: in Block l, at block r, and blocks between l and r ar inclusive.

Implementation

At the beginning of the program are given constants MAXN, LOG_MAXLIST and SQRT_MAXLIST, determine the maximum number of vert in the graph, which, if necessary, should be increased.

```
const int MAXN = 100 * 1000;
const int MAXLIST = MAXN * 2;
const int LOG_MAXLIST = 18;
const int SQRT_MAXLIST = 447;
const int MAXBLOCKS = MAXLIST / ((LOG_MAXLIST + 1) / 2) + 1;

int n, root;
vector <int> g [MAXN];
int h [MAXN]; // Vertex height
```

```
vector <int> a; // Dfs list
int a_pos [MAXN]; // Positions in dfs list
int block; // Block size = 0.5 log A.size ()
int bt [MAXBLOCKS] [LOG_MAXLIST + 1]; // Sparse table on blocks (relative minimum positions in blocks)
int bhash [MAXBLOCKS]; // Block hashes
int brmq [SQRT_MAXLIST] [LOG_MAXLIST / 2] [LOG_MAXLIST / 2]; // Rmq inside each block, indexed by block hash
int log2 [2 * MAXN]; // Precalced logarithms (floored values)

// Walk graph
void dfs (int v, int curh) {
        h [v] = curh;
        a_pos [v] = (int) a.size ();
        a.push_back (v);
        for (size_t i = 0; i <g [v] .size (); ++ i)
                if (h [g [v] [i]] == -1) {
                        dfs (g [v] [i], curh + 1);
                        a.push_back (v);
                }
}

int log (int n) {
        int res = 1;
        while (1 << res <n) ++ res;
        return res;
}

// Compares two indices in a
inline int min_h (int i, int j) {
        return h [a [i]] <h [a [j]]? i: j;
}

// O (N) preprocessing
void build_lca () {
        int sz = (int) a.size ();
        block = (log (sz) + 1) / 2;
        int blocks = sz / block + (sz% block? 1: 0);

        // Precalc in each block and build sparse table
        memset (bt, 255, sizeof bt);
        for (int i = 0, bl = 0, j = 0; i <sz; ++ i, ++ j) {
                if (j == block)
                        j = 0, ++ bl;
                if (bt [bl] [0] == -1 || min_h (i, bt [bl] [0]) == i)
                        bt [bl] [0] = i;
        }
        for (int j = 1; j <= log (sz); ++ j)
                for (int i = 0; i <blocks; ++ i) {
                        int ni = i + (1 << (j-1));
                        if (ni> = blocks)
                                bt [i] [j] = bt [i] [j-1];
                        else
                                bt [i] [j] = min_h (bt [i] [j-1], bt [ni] [j-1]);
                }

        // Calc hashes of blocks
        memset (bhash, 0, sizeof bhash);
        for (int i = 0, bl = 0, j = 0; i <sz || j <block; ++ i, ++ j) {
                if (j == block)
                        j = 0, ++ bl;
                if (j> 0 && (i> = sz || min_h (i-1, i) == i-1))
                        bhash [bl] + = 1 << (j-1);
        }

        // Precalc RMQ inside each unique block
        memset (brmq, 255, sizeof brmq);
        for (int i = 0; i <blocks; ++ i) {
                int id = bhash [i];
                if (brmq [id] [0] [0]! = -1) continue;
                for (int l = 0; l <block; ++ l) {
                        brmq [id] [l] [l] = l;
                        for (int r = l + 1; r <block; ++ r) {
                                brmq [id] [l] [r] = brmq [id] [l] [r-1];
                                if (i * block + r <sz)
                                        brmq [id] [l] [r] =
                                                min_h (i * block + brmq [id] [l] [r], i * block + r) - i * bloc
                        }
                }
        }

        // Precalc logarithms
        for (int i = 0, j = 0; i <sz; ++ i) {
                if (1 << (j + 1) <= i) ++ j;
```

```
                log2 [i] = j;
        }
}

// Answers RMQ in block #bl [l; r] in O (1)
inline int lca_in_block (int bl, int l, int r) {
        return brmq [bhash [bl]] [l] [r] + bl * block;
}

// Answers LCA in O (1)
int lca (int v1, int v2) {
        int l = a_pos [v1], r = a_pos [v2];
        if (l> r) swap (l, r);
        int bl = l / block, br = r / block;
        if (bl == br)
                return a [lca_in_block (bl, l% block, r% block)];
        int ans1 = lca_in_block (bl, l% block, block-1);
        int ans2 = lca_in_block (br, 0, r% block);
        int ans = min_h (ans1, ans2);
        if (bl <br - 1) {
                int pw2 = log2 [br-bl-1];
                int ans3 = bt [bl + 1] [pw2];
                int ans4 = bt [br- (1 << pw2)] [pw2];
                ans = min_h (ans, min_h (ans3, ans4));
        }
        return a [ans];
}
```


arch • 10 месяцев назад

r-2sz+1 - наверное, имелось в виду просто r-sz+1?