## Finding the lowest common ancestor in rooted trees

I wrote about [depth-first search](http://csengerg.github.io/2015/12/23/depth-first-search.html) in my last post. At the end of the post, I included a simple problem, to show, how to use depth-first search for solving it. In this post, we will discuss the problem of finding the lowest common ancestor of two nodes in a rooted tree, and see, how can we use depth-first search for preprocessing our tree.

In computer science [trees](https://en.wikipedia.org/wiki/Tree_(graph_theory)) are playing a big role. On competitions, you can be asked, to find the lowest common ancestor of any two nodes in a rooted tree many times. I will describe a method with which you will be able to find the lowest common ancestor in [**O(log N)**](https://en.wikipedia.org/wiki/Big_O_notation) time(if you have a tree consisting of N nodes).

**Preprocessing**

In order to provide a fast query time (we can be asked repeatedly about lowest common ancestors), we preprocess our tree first. And here comes the above mentioned depth-first search. For each node, we will need to know two things:

1. it’s **depth** (it’s distance from the root node)
2. it’s **2^k.** parent for every k

I think the first part is clear enough, we just need to start a depth-first search from the root node and store the depth of all the other nodes:

dfs(node, distance):

explored[node] = true

depth[node] = distance

for all the undiscovered neighbours:

//we increase distance by one

dfs(neighbor,distance+1)

The variable distance stores the actual distance from the root node, so executing dfs(root, 0) will preprocess the information we need in the first part.

**A couple words about parents - understanding the method**

The second part is a bit harder, but understanding this will lead to a deep understanding of the full algorithm. Under the *2^k.* parent we mean the **1st, 2nd, 4th, 8th …** parent of a given node. This means, that for every node we need to store at most log2(N) parents. This can be done with a N x log2(N) big two-dimensional array. Let us call this array pa: pa[v][i] will store the *2^i*. parent of node *v*, if it exists, else it’s value will be -1.

You might ask now:

*OK, but why is it good, and how can we fill this entire array?*

Firstly, it’s good, because it doesn’t requires big memory. The factor log2(N) increases very slowly in respect of *N* (imagine a tree with **ten million** vertices - log2(10 000 000) is still **less than 24!!!**).

Filling the array is actually very easy. For explaining the idea, let’s consider a family tree from the real life:

* my dad is my “first parent”
* my dad’s dad is my dad’s “first parent” and therefore my “second parent”(grandfather, as we call it)
* my grandfather’s grandfather is my “second parent’s” “second parent”(my great-great-grandfather) and my “fourth parent”

Yes, I just described a **recursive formula** for calculating somebody’s *2^k.* parent. We can go further, and see, that this idea holds for trees as well:

We can sum up this with a single line of code: pa[node][i] = pa[2^(i-1). parent of node][i-1], where the *2^(i-1).* parent of node is pa[node][i-1], so: pa[node][i] = pa[pa[node][i-1]][i-1]Let’s update our depth-first search, to make sure, we know pa[v][0](i. e. the first parent) for every node:

dfs(node, parent, distance):

explored[node] = true

depth[node] = distance

pa[node][0] = parent //this node's parent node

for all the undiscovered neighbours:

// increase distance by one,

// pass the current node as new parent

dfs(neighbor, node, distance+1)

Note, that pa[root][0]=-1, because it has no parent node, so in our algorithm we will execute this code segment as dfs(root, -1, 0).

From now, we can use the recursive formula from above, and fill the pa array:

for i = 1 to log2(N):

for j = 0 to N:

if pa[j][i-1] != -1:

pa[j][i] = pa[pa[j][i-1]][i-1]

**The exact algorithm**

Before discussing the algorithm in general, we will see first a special case: when the two asked nodes are on the same level:

You can easily see here, that for any pair of nodes x (from the left side) and y (from the right side) the following statement is true: lca(u,v)=lca(x,y). That’s why we will start to jump upwards.

*How big steps can we make upwards?*

We will jump according to our pa array, so that we jump always to the **highest** node **below** the LCA node on both sides. These nodes(*u’,v’,u’‘,v’‘*) will have such properties, that they are the first parents in the pa array for *u* and *v*, which are different. Therefore we can write the following code for this case:

lca\_special(u, v):

for i = log2(N) to 0:

if pa[u][i] != pa[v][i] and pa[u][i] != -1:

u = pa[u][i]

v = pa[v][i]

return pa[u][0]

After this we will end up with two child nodes of the LCA node, so we just read out the parent of *u* or *v* from the pa array and we are done.

When *u* and *v* are not on the same level, we will lead the problem back to the previous case: we can manage *u* and *v* so, that they will end up on the same level with the help of the previous idea.

Firstly, make sure, that *u* is deeper than *v*. Now jump upwards again until *u’* is not on the same level as *v*:

lca\_general(u, v):

if depth[u] < depth[v]:

swap(u, v)

for i = log2(N) to 0:

//if we have the opportunity to jump 2^i upwards, we jump

if depth[u] - 2^i >= depth[v]:

u = pa[u][i]

//they can equal after this... in this case we found the LCA

if u == v: return u

return lca\_special(u, v)

**Summary**

Let’s see the time complexity of our algorithm. I will use the big O notation:

|  |  |  |
| --- | --- | --- |
| **Part of the algorithm** | **Time** | **Explanation** |
| Depth-first search | *O(N)* | We visit each node exactly once, so linear in respect of the number of nodes |
| Filling the paarray | *O(N\*log2(N))* | Linear in respect of the size of the array, which is N \* log2(N)big |
| One query | *O(log2(N))* | We make two times at most *log2(N)* jumps, as you can see in the pseudocode too |

Our preprocessing requires **O(N log N)** time, and a query can be answered in **O(log N)** time. To be absolutely complete, I share my C++ code too (a test file can be found in the same directory in [my Github repository](https://github.com/CsengerG/competitive-programming/tree/master/LCA)):

|  |  |
| --- | --- |
|  | // Algorithm for finding the lowest common ancestor of two nodes in a tree |
|  | // with O(N log N) preprocessing and O(log N) query time |
|  |  |
|  | #include <bits/stdc++.h> |
|  |  |
|  | #define ll long long |
|  | #define FOR(i,n) for(int i=0;i<n;++i) |
|  | #define pb push\_back |
|  | #define sz size |
|  | #define MAXN 100000 |
|  | #define LN 17 |
|  | #define MAXLN 20 |
|  |  |
|  | using namespace std; |
|  |  |
|  | vector<ll> g[MAXN]; |
|  |  |
|  | //We use the array pa[i][j] for determining |
|  | //the 2^i. parent of the vertex j |
|  | ll pa[MAXLN][MAXN]; |
|  |  |
|  | //depth[i] stores the distance of the i. vertex from the root |
|  | ll depth[MAXN]; |
|  |  |
|  | //We initialize the pa and the depth array with a depth-first-search |
|  | void dfs(ll v, ll parent, ll d){ |
|  | depth[v]=d; |
|  | pa[0][v]=parent; |
|  |  |
|  | FOR(i,g[v].sz()){ |
|  | if(depth[g[v][i]]==-1) dfs(g[v][i],v,d+1); |
|  | } |
|  | } |
|  |  |
|  | ll LCA(ll u, ll v){ |
|  | //lets make sure u is deeper than v, and if they are not on the same level in the tree, |
|  | //find an another vertex instead of u which is on the same level with v |
|  | if(depth[u] < depth[v]) swap(u,v); |
|  |  |
|  | //search for a new v vertex |
|  | for(int i=LN;i>=0;--i){ |
|  | if( depth[u] - (1<<i) >= depth[v] ){ |
|  | u = pa[i][u]; |
|  | } |
|  | } |
|  |  |
|  | if( u == v ) return u; //if u and v are the same, we are done |
|  |  |
|  | //now find the lowest common ancestor |
|  | for(int i=LN;i>=0;--i){ |
|  | if( pa[i][u] != -1 and pa[i][u] != pa[i][v] ){ |
|  | u = pa[i][u]; |
|  | v = pa[i][v]; |
|  | } |
|  | } |
|  |  |
|  | return pa[0][u]; |
|  | } |
|  |  |
|  | int main(void){ |
|  | //Initialize the depth and pa array |
|  | FOR(i,MAXN) depth[i]=-1; |
|  | FOR(i,LN) FOR(j,MAXN) pa[i][j]=-1; |
|  |  |
|  | //Read the input |
|  | ll n,q; cin>>n>>q; |
|  |  |
|  | FOR(i,n-1){ |
|  | ll a,b; |
|  | cin>>a>>b; |
|  | a--; b--; |
|  | g[a].pb(b); |
|  | g[b].pb(a); |
|  | } |
|  |  |
|  | //Start the dept-first-search |
|  | dfs(0,-1,0); |
|  |  |
|  | //Fill the pa array |
|  | for(int i=1;i<LN;++i){ |
|  | for(int j=0;j<n;++j){ |
|  | if( pa[i-1][j] != -1 ){ |
|  | pa[i][j] = pa[i-1][pa[i-1][j]]; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | //Handle the queries |
|  | FOR(i,q){ |
|  | ll u,v; |
|  | cin>>u>>v; |
|  | u--; v--; |
|  | cout<<u+1<<" "<<v+1<<" "<<LCA(u,v)+1<<endl; |
|  | } |
|  |  |
|  | return 0; |
|  | } |

[**view raw**](https://gist.github.com/CsengerG/3c93b069523672b8771d/raw/54031ca078615bb895571a945f24fdc521ba354e/lowest-common-ancestor.cpp)[**lowest-common-ancestor.cpp**](https://gist.github.com/CsengerG/3c93b069523672b8771d#file-lowest-common-ancestor-cpp) hosted with ❤ by **[GitHub](https://github.com/)**