Least Common Ancestor

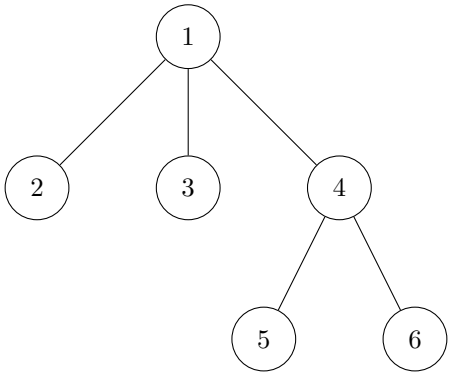
Kevin Shan

December 2022

1 Introduction

The **Least Common Ancestor** of two nodes a and b in a tree is defined as the lowest node that contains both node a and node b in its subtree.

This operation, when done quickly, is very useful for problems that involve querying on trees, or dynamic programming on trees.



For example, in the given tree, $LCA(5, 6) = 4$.

A naive solution to the LCA problem would be to perform a DFS from node a to node b, and return the node with the highest depth on the path from a to b. While this solution is not ideal ($O(M + N)$), it does provide a useful observation: that the LCA between two nodes on a tree always lies on the path between a and b.

We aim to find a solution that falls under $O(NlogN)$ time to set up, and $O(logN)$ to compute one query.

2 Tree Flattening (Euler Tour)

We will perform a **Euler Tour** on our tree in order to "flatten" the tree to make it easier to process.

A Euler tour involves traversing through a tree via DFS, and keeping track of the instance you first enter and leave a node. Learn more here: https://en.wikipedia.org/wiki/Euler_tour_technique

A Euler Tour gives us some nice properties that we can work with:

1. Given the start and end time $(c_s[i], c_t[i])$ of a node i, all nodes x with $c_s[x] > c_s[i]$ and $c_t[x] < c_t[i]$ are in the subtree of i.
2. The start and end time of two nodes will either be contained within each other, or not overlap. This aligns with the structure of a tree.

We will see some additional modifications to the Euler Tour later on.

```
vector<int> adj[N];
int be[N], end[N], up[N][logN];
int cnt = 1;
void dfs(int x, int p){
    be[x] = cnt++;
    up[x][0] = p;
    for(int i=1; i<=l; i++) up[x][i] =
                    up[up[x][i-1]][i-1];
    for(int i:adj[x]){
        if(i!=p)
            dfs(i, x);
    }
    en[x] = cnt;
}
```

3 LCA with Range Minimum Queries

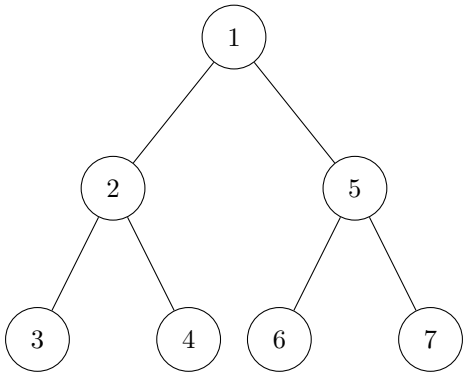
3.1 Range Minimum Queries

RMQs can be performed with some data structure such as a Segment Tree, but there is also a simpler implementation of RMQs that has a query complexity time of $O(1)$. Learn more here: https://en.wikipedia.org/wiki/Range_minimum_query#Solution_using_constant_time_after_linearithmic_space_pre-computation

This data structure can be summarized as follows: Precompute all intervals of size $2^0, 2^1, 2^2 ... 2^l$ such that $l = log_2(N)$. We represent $dp[i][j]$ as the minimum on the range $[i, i + 2^j]$. Precompuation takes $O(NlogN)$. We can actually query in linear time: note that, due to the nature of minimizing functions, the minimum on the range $[l, r]$ is the same as the mininimum of the minimums of two overlapping ranges, $[l, r_1]$ and $[l_1, r]$ such that $r_1 \leq r$ and $l_1 \geq l$. Consequently, when answering a query for the minimum of $[l, r]$, when $s = r - l + 1$ and $g = \lfloor \log_2 s \rfloor$, we simply take $\min(dp[l][g], dp[r - 2^g][g])$ as the answer. Note that these two ranges overlap, and consequently, yield us the correct value.

3.2 Euler Tour Modification

We can modify our Euler Tour to fit the demands for the LCA problem. Instead of keeping track of just the start and end times of each node, we also include each time our DFS "returns" to the node.



When a DFS is performed on this graph, the nodes will be traversed in the same order as their numbering. When appending each visitation of a node to an array as demonstrated by the code below, we end up with the following:

```
vector<int> v;
```

```
vector<int> adj[N];
void dfs(int x, int p){
   v.pb(x);
   for(int a:adj[x]){
      if(a==p) continue
dfs(a, x);
v.pb(x);
}
}
// v = {1, 2, 3, 2, 4, 2, 1, 5, 6, 5, 7, 5, 1}
```

Doing this yields some characteristics not seen before: When querying the path between two nodes, all nodes that occur between the index of the first time node a occurs, and the index of the first time node b occurs, are located on the subtree of the $LCA(a, b)$.

Consequently, the algorithm for finding $LCA(a, b)$ is as follows: Given the modified Euler Tour of a tree, when querying the LCA two nodes, we just need to find the node with the lowest depth on the subarray from the $c_s[a]$ to $c_s[b]$, inclusive. When implemented with an RMQ, the query time complexity for $LCA(a, b)$ is constant.

4 LCA with Binary Lifting

There is another, more intuitive solution to LCA that employs Binary Lifting. Binary Lifting the the process of traversing vertically on a tree in incremental distances of powers of two. The critical observation is that any distance can be reached in under $log(N)$ steps.

5 Euler Tour

For this version of LCA, the standard Euler Tour mentioned previously will be used, where we find $c_s[i]$ and $c_t[i]$ for each node i. We will also be using one particular characteristic of Euler Tours, where we found that given the start and end time $(c_s[i], c_t[i])$ of a node i, all nodes x with $c_s[x] > c_s[i]$ and $c_t[x] < c_t[i]$ are in the subtree of i.

5.1 Checking if Nodes are Ancestors

Before we review the LCA algorithm, first we need to find how to check if a node is an ancestor of another. With our flattened tree, we can create the following condition: Given node a and b, if $c_s[a] < c_s[b]$ and $c_t[a] > c_t[b]$, then node a is an ancestor of node b.

5.2 Binary Lifting

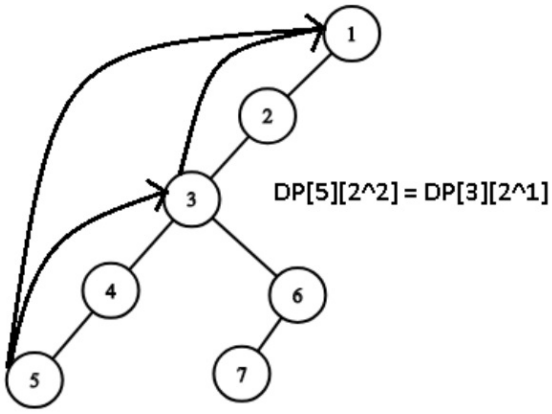
We need to find the lowest node that is an ancestor of both nodes a and b. To do this, we locate the highest node that is **not** an ancestor of both nodes, and find the parent of such a node. We can do this through Binary Lifting, where we move up in increments of powers of two.

However, to calculate these instances, we will need to precompute the jumps. We can accomplish this through dynamic programming (dp). For each node i, we will precompute all of $dp[i][b]$, where $dp[i][b]$ is the ancestor of node i that is 2^b layers of depth above it. Because of the structure of trees, only one of these exists.

Assume we have computed $dp[i][b]$ for all of $dp[j][b]$ such that j is an ancestor of i. Starting from $b = 0$, we can first set $dp[i][0]$ equal to the parent of i, which we call p. From there, we can assign $dp[i][1] = dp[p][0]$. Then, we can assign $dp[i][2] = dp[dp[i][1]][1]$. We continue this for all values of b, with the following recurrence relation:

$$dp[i][b] = dp[dp[i][b-1]][b-1].$$

Below is a diagram illustrating visually how this transition functions:



Why is this useful? Assume we know that $LCA(a, b) = x$, and the difference in depth between x and node b is some value. If we were to represent this difference in depth in binary, each bit in the bitmask representation corresponds to a jump that we have already precalculated.

With this established, we can easily find the LCA of two nodes: Set node x to be the node are searching on, and initiate it arbitrarily to a. We will iterate b from $\lfloor log_2(N) \rfloor$ to 0, where at each step, we check if $dp[x][b]$ is an ancestor of b. If it is not, then we set $x = dp[x][b]$. Otherwise, continue to the next iteration. Doing so allows us to bring x closer to $LCA(a, b)$, in a maximum of $\lfloor log_2(N) \rfloor$ steps.

The code for an LCA implementation can be found below:

```
int n;
vector<int> adj[MAXN];
int be[MAXN];
int en[MAXN];
int cnt = 1;
int up[MAXN][100];
int l; //max power of 2 (log2 of n)
void dfs(int x, int p){
    be[x] = cnt++;
    up[x][0] = p;
    for(int i=1; i<=l; i++) up[x][i] = up[up[x][i-1]][i-1];
    for(int i:adj[x]){
        if(i!=p) dfs(i, x);
    }
    en[x] = cnt;
}
bool anc(int u, int v){
    return be[u]<=be[v] && en[u]>=en[v];
}
int lca(int u, int v){
    if (anc(u, v)) return u;
    if (anc(v, u)) return v;
    for(int i = l; i>=0; i--){
        if(!anc(up[u][i], v))
            u = up[u][i];
    }
    return up[u][0];
}
```