Simple $O(1)$ Query Algorithm for Level Ancestors

Sanjeev Saxena*
Dept. of Computer Science and Engineering,
Indian Institute of Technology,
Kanpur, INDIA-208 016

July 25, 2022

Abstract

This note describes a very simple $O(1)$ query time algorithm for finding level ancestors. This is basically a serial (re)-implementation of the parallel algorithm.

Earlier, Menghani and Matani described another simple algorithm; however, their algorithm takes $O(\log n)$ time to answer queries.

Although the basic algorithm has preprocessing time of $O(n \log n)$, by having additional levels, the preprocessing time can be reduced to almost linear or linear.

Keywords: Level Ancestors; Rooted Trees; Algorithms; Graphs; Euler Traversal

1. Introduction

In the level ancestor problem, we are given a rooted tree, which is to be preprocessed to answer queries of the type:

find the i^{th}ancestor of a node v.

Several sequential and parallel algorithms are known for this problem [6, 2, 4, 8, 1, 9].

This note describes a very simple $O(1)$ query time algorithm for finding level ancestors. This is basically a serial (re)-implementation of the parallel algorithm of Berkman and Vishkin [6]. Ben-Amram [2] also gave a serial version of the parallel algorithm [6]; however, the proposed description of the "basic" constant-time algorithm is still simpler and more complete.

Earlier, Menghani and Matani [9] described another simple algorithm. However, their algorithm takes $O(\log n)$ time to answer queries.

Although the basic algorithm has preprocessing time of $O(n \log n)$, the preprocessing time can be reduced

*E-mail: ssax@iitk.ac.in

to almost linear by having additional levels. The bound can be made linear by constructing a lookup table of all possible instances of small size [5].

Many steps of the parallel algorithm [6] have alternate, simple serial implementations. In serial case, some parameters can also be changed. We modify the size of the Near array (from $\sqrt{k}$ to k). And we do not require two other arrays used in the parallel algorithm [6].

Some preliminary techniques are discussed in Section 2. The basic $O(n \log n)$ preprocessing algorithm is discussed in Section 3. This is used to get $O(1)$ query algorithm in Section 4. The use of two and multi-level structure is discussed in Section 5. A practical algorithm for "reasonable" values of n ($n < 10^{75}$) is described in Section 6.

2. Preliminaries

Given a rooted tree, we first find each node's level (distance from root). The level of the root is zero, and if w is the parent of v, then level$[v]$ = 1+level$[w]$. Levels can be computed in linear time by traversing the tree.

Next, for each edge between v and its parent w, we create two directed edges (v, w) and (w, v). As the in-degree of each node is the same as the out-degree, the graph is Eulerian, and an Euler-tour can be found [10, 3]. Let us put the vertices of the tour in an array. If u and v are two successive vertices, then level$[u] = 1 \pm$level$[v]$; thus, levels of two successive vertices differ by exactly one (in absolute terms).

The ancestor of v at level d is the first vertex at level d in the Euler Tour after (say, after the last) occurrence of v [6, 2]. Thus, it is sufficient to solve the following Find Smaller (FS) problem [2, 6]:

> Process an array $A[1 : n]$ such that, given query FS(i, x), find the smallest $j \geq i$ such that $a_j \leq x$.

Berkman, Schieber and Vishkin [7] introduced the Nearest Smallers (NS) problem:

> Given an array $A[1 : n]$, for each i, find the smallest $j > i$ such that $a_j < a_i$.

The NS problem can be solved in linear time using a stack [7]. Basically, if the current item is larger than the stack top, then we push it into the stack. Else, keep popping from the stack and make the current item the nearest smaller of all items popped (as long as the current item is smaller). Finally, we push the current item. In more detail:

Push n in stack
for $i = n - 1$ downto 1 do
    let t be the stack top
    if $a_i > a_t$ then NS$[i] = t$ else push i into the stack.

Items which are in the stack do not have a nearest (right) smaller.

3: Preprocessing:Basic Algorithm

Solution for a "representative" set of "FS-queries" is precomputed and stored in tables. In the basic algorithm, depending on the values of i, a_i and x, we compute two integers i_1 and d, and return $(i_1, d)^{\text{th}}$entry, which will contain the index of the desired item. The table is called [6], FAR-array; actually, it is a two-dimensional array.

For each i, a different array FAR_i (of size depending on i) is constructed. The j^{th}entry of the array:

> $\text{FAR}_i[j]$ will contain the index of the first location right of a_i having a value less than or equal to $a_i - j$.

Thus, if $\text{FAR}_i[j] = h$ then h is the smallest index (with $h > i$) such that, $a_h \leq a_i - j$.

Given query $\text{FS}[i, x]$ we compute $d = a_i - x$. Then $\text{FAR}_i[d]$ is the first location right of a_i having value less than or equal to $a_i - d = a_i - (a_i - x) = x$.

If the depth of the tree is d, computing all legal FAR_i values make take $O(nd)$ time (and space) hence only some of the FAR_i values are computed. If $i - 1 = s2^r$, i.e., 2^r is the largest power of 2 dividing $i - 1$, then we will have $3 * 2^r$ entries in FAR_i. For each of following numbers:

$$a_i - 1, a_i - 2, \ldots, a_i - 3 * 2^r,$$

we have to find the left most index k, $k > i$ such that $a_k \leq a_i - j$, for $j = 1, 2, \ldots, 3 * 2^r$.

All FAR-arrays can be easily computed using Nearest Smallers: Assume $\text{NS}[i] = q$. Then, all items $a_{i+1}, \ldots, a_{q-1}$ are larger than a_i. We can make $\text{FAR}_i[1] = q$. If $d = a_i - a_q$, then we can also make $\text{FAR}_i[2] = q$,$\text{FAR}_i[3] = q$, ... , $\text{FAR}_i[d] = q$. In case, we need to find $\text{FAR}_i[d + 1]$, we again find $\text{NS}[q]$ and proceed.

Thus, if Nearest Smallers are known, then each FAR entry can be filled in $O(1)$ time.

Lemma 1 *All "FAR" arrays can be computed in $O(n \log n)$ time and space.*

Proof: If $i - 1 = s2^r$, i.e., 2^r is the largest power of 2 dividing $i - 1$, then we have $3 * 2^r$ entries in FAR_i.

As $n/2^i$ integers are multiple of 2^i and $n/2^{i+1}$ integers are multiple of 2^{i+1}, it follows that $n/2^i - n/2^{i+1} = n/2^{i+1}$ integers are multiple of 2^{i+1} but not of 2^i. Or for $n/2^{i+1}$ integers, the largest power of 2 which can divide $i - 1$ is 2^i, hence for these i the size of FAR_i array will be $3 * 2^i(n/2^i + 1)$, or size of all FAR-arrays together will be:

$$\sum 3 * 2^i(n/2^i + 1) = \sum(3n/2) = (3n/2) \log n$$

Or computing all FAR-arrays will take $O(n \log n)$ time and space. ∎

4. Query Answering: Basic Algorithm

Let us assume that the query is $\text{FS}[i, x]$. Let $d = a_i - x$. And let 2^p be the largest power of 2 not larger than d, i.e., $2^p \leq d < 2^{p+1}$. For query $\text{FS}(i, x)$, we have to find the first item after location i smaller than x. We proceed as follows:

1. Let $d = a_i - x$.
2. Let p be s.t., $2^p \leq d < 2^{p+1}$, i.e., 2^p is the largest power of 2 not larger than d; or p is the number of zeroes in d.
3. Let i_1 be the largest index less than (or equal to) i s.t., 2^p divides $i_1 - 1$, i.e.,

$$i_1 = \left\lfloor \frac{i-1}{2^p} \right\rfloor \times 2^p + 1$$

4. Return $\text{FAR}_{i_1}[a_{i_1} - x]$.

Correctness of the above method follows from:

Lemma 2 ([6, 2]) *The first element to the right of a_i with a value less than or equal to x is also the first element to the right of a_{i_1} with a value less than or equal to x.*

Proof: As $i - i_1 < 2^p$, and as a_t and a_{t+1} can differ by at most one, it follows, that for any $i_1 \leq j \leq i$, $a_j > a_i - 2^p > a_i - d = x$. ∎

Thus, we can answer the query by also reporting $\text{FS}[i_1, x]$. And $\text{FAR}_{i_1}[a_{i_1} - x]$ is the first location right of a_{i_1} with value less than or equal to $a_{i_1} - (a_{i_1} - x) = x$. As $a_{i_1} - x < a_i + 2^p - x = d + 2^p < 2^{p+1} < 3 * 2^p$, value $\text{FAR}_{i_1}[a_{i_1} - x]$ has been computed [6, 2].

5. Two Level Structure: Reducing Pre-processing Time and Space

We divide the array (containing levels) into parts of size k. For each part, we find and put the minimum value of that part into another "global" array. For these n/k minimum values, we construct a new instance of the FS-problem. However, the two minimum values may now differ by up to k. Thus, we have to modify the algorithm of the previous sections [6, 2].

Modified FAR Array

We use the original definition of $\text{FAR}_i[j]$ [6]; we will call the FAR array of this section as "modified" FAR array to differentiate it from that of Section 3. Let $e = \left\lfloor \frac{a_i}{k} \right\rfloor$ (thus, $(e-1)k < a_i \leq ek$). Again, we let 2^r be the largest power of 2 which divides $i - 1$. $\text{FAR}_i[j]$ will give the first location right of a_i with value less than or equal to $(e-j)k$, again for $1 \leq j \leq 3 * 2^r$.

Let i_1 be as before. And let $e_1 = \left\lfloor \frac{a_{i_1} - x}{k} \right\rfloor$. Then $\text{FAR}_{i_1}[e_1]$ will give the first location right of a_{i_1} with value less than or equal to $(e - e_1)k = k\left(\left\lfloor \frac{a_{i_1}}{k} \right\rfloor - \left\lfloor \frac{a_{i_1} - x}{k} \right\rfloor\right) \leq k\left(1 + \left\lfloor \frac{x}{k} \right\rfloor\right) \leq k + x$.

For computing the modified FAR-values, we use another array $B[1:n]$, with $b_i = \left\lfloor \frac{a_i}{k} \right\rfloor$. And find the nearest smallers in the B-array (assuming that in case of duplicates, the first entry is smaller). Now, $e = \left\lfloor \frac{a_i}{k} \right\rfloor = b_i$. If $\text{FAR}_i[j] = t$, then a_t is the first number right of a_i in A with value less than or equal to $(e-j)k = (b_i - j)k$; or equivalently, a_t/k is is the first number right of b_i in B with value less than or equal to $(b_i - j)$. Thus, the modified FAR-values can be computing as in Section 3, using array B instead (of A).

REMARK To make sure that all divisions are by a power of 2, we can choose k to be a number between $\frac{1}{8}\log n$ and $\frac{1}{4}\log n$ which is a power of 2.

From previous analysis, we know that number of entries of FAR will be $O(n' \log n) = O(\frac{n}{k} \log n) = O(n)$, if $k = \theta(\log n)$.

Hence, we can preprocess the global array in $O(\frac{n}{k} \log n + \frac{n}{k} k) = O(n)$ time, if $k = \theta(\log n)$.

We can preprocess each local part in $O(k \log k) = O(k \log \log n)$ time using the algorithm of Section 3.

Near Array

To get the "exact" location, we use another array Near [6]. $\text{Near}_i[j]$, for $1 \leq j \leq k$ will give the first location right of a_i with value less than $a_i - j$ (just like the FAR array of Section 3). As we are storing k entries for each a_i, total number of entries will be $O\left(\frac{n}{k} k\right) = O(n)$. If $d = a_i - x$. Then $\text{Near}_i[d]$ is the first location right of a_i having value less than or equal to $a_i - d = a_i - (a_i - x) = x$ (provided, $d \leq k$).

Again, the Near-table can be filled using Nearest smallers value, with $O(1)$ time per entry.

Query

Again we have:

Lemma 3 ([6, 2]) *First element to the right of a_i with value less than or equal to x is also the first element to the right of a_{i_1} with value less than or equal to x.*

Proof: As $i - i_1 < 2^p$, and as a_t and a_{t+1} can differ by at most k, it follows, that for any $i_1 \leq j \leq i$, $a_j > a_i - k2^p > a_i - d = x$. ∎

Thus, we can answer the query by also reporting $\text{FS}[i_1, x]$.As $0 < a_{i_1} - x < 3k * 2^p$, we have $0 \leq \frac{a_{i_1}}{k} - \frac{x}{k} < 3 * 2^p$, or $0 \leq b_i - \frac{x}{k} < 3 * 2^p$.

For queries, we first determine the "local" group (using the global algorithm), and then the index in the corresponding local group (of k-items) using the basic algorithm. This will give an $O(\frac{n}{k}(k \log k) = O(n \log \log n)$ preprocessing time and $O(1)$ query time algorithm.

Lemma 4 *There is an algorithm to solve FS-problem, in which two items differ by at most one with $O(n \log \log n)$ preprocessing time and $O(1)$ query time.*

Alternatively, we make k a power of 2 between $\frac{1}{8} \log n$ and $\frac{1}{4} \log n$, and build a table for all possible instances of size k (see eg, [5]), this will give an algorithm with $O(n)$ preprocessing cost. FS-problem, in local group is now solved using a table lookup. As the details are very similar to that used for the lowest common ancestor problem [5], they are omitted.

Lemma 5 *There is an algorithm to solve FS-problem, in which two items differ by at most one with $O(n)$ preprocessing time and $O(1)$ query time.*

6. Towards more Practical Algorithms

If the value of n is not too large, say $n < 10^{75}$, then we can get an almost linear time algorithm, which will be linear for all practical purposes (the value of function for $n < 10^{75}$ is less than 3). Instead of using the basic algorithm for local groups, we can use the algorithm of Lemma 4 instead.

We, as before, divide the array (containing levels) into parts of size $k = \theta(\log n)$. For each part, we find and put the minimum value of that part into another "global" array. For these n/k minimum values, we construct a new instance of the FS-problem. The global instance is preprocessed for FS-queries in $O(n)$ time, as in Section 5.

However, after determining the correct part, the corresponding local problem for the part is solved using the algorithm of Lemma 4. Each local group is individually preprocessed using method of Lemma 4 in $O(k \log\log k) = O(k \log^{(3)} n)$ time. Or total time for preprocessing all groups is $O\left(\frac{n}{k} k \log^{(3)} n\right) = O(n \log^{(3)} n)$.

REMARK 1: As $\log^{(3)} n \leq 3$ for $n < 10^{75}$, the above algorithm may be faster in practice for all reasonable values of n.

REMARK 2: In fact, by using a constant number of levels, the preprocessing time can be made $O(n \log^{(r)} n)$, for any $r > 1$, and query time becomes $O(r)$.

References

[1] Stephen Alstrup and Jacob Holm, Improved Algorithms for Finding Level Ancestors in Dynamic Trees. ICALP 2000: 73-84 (2000)

[2] Amir M. Ben-Amram, The Euler Path to Static Level-Ancestors. CoRR abs/0909.1030 (2009)

[3] B.G.Baumgart, A polyhedron representation for computer vision, Proc. 1975 National computer conf., AFIPS conference proceedings vol 44, 589-596 (1975).

[4] Michael A. Bender and Martin Farach-Colton, The Level Ancestor Problem Simplified. Theor. Comput. Sci. 321: 5-12(2004).

[5] M.A.Bender and M.Farach-Colton, The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds) Theoretical Informatics. LATIN 2000. LNCS 1776 (2000) Springer, Berlin, Heidelberg.

[6] O. Berkman and U. Vishkin. Finding level-ancestors in trees. J. of Comp. and Sys. Scie., 48(2):214-230 (1994).

[7] Omer Berkman, Baruch Schieber and Uzi Vishkin: Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. J. Algorithms 14(3): 344-370 (1993)

[8] P. Dietz. Finding level-ancestors in dynamic trees. In 2nd Work. on Algo. and Data Struc., LNCS 1097, 32-40 (1991).

[9] Gaurav Menghani, Dhruv Matani, A Simple Solution to the Level-Ancestor Problem, arXiv:1903.01387v2 (2021).

[10] Robert Endre Tarjan and Uzi Vishkin, An Efficient Parallel Biconnectivity Algorithm. SIAM J. Comput. 14(4): 862-874 (1985)