The Level Ancestor Problem Simplified

Michael A. Bender[1*] and Martín Farach-Colton[2]

[1] Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA. bender@cs.sunysb.edu.
[2] Google Inc, 2400 Bayshore Parkway, Mountain View, California 94043, USA, & Department of Computer Science, Rutgers University. Partially supported by NSF CCR 9820879. martin@farach-colton.com.

Abstract. We present a very simple algorithm for the *Level Ancestor Problem*. A *Level Ancestor Query* $\mathrm{LA}(v, d)$ requests the depth d ancestor of node v. The Level Ancestor Problem is thus: preprocess a given rooted tree T to answer level ancestor queries. While optimal solutions to this problem already exist, our new optimal solution is simple enough to be taught and implemented.

1 Introduction

A fundamental algorithmic problem on trees is how to find *Level Ancestors* of nodes. A *Level Ancestor Query* $\mathrm{LA}(u, d)$ requests the depth d ancestor of node u. The *Level Ancestor Problem* is thus: preprocess a given n-node rooted tree T to answer level ancestor queries. Thus, one must optimize both the preprocessing time and the query time.

The natural solution of simply climbing up the tree from u is $O(n)$ at query time, and the other solution of precomputing all possible queries has $O(n^2)$ preprocessing.

Solutions with $O(n)$ preprocessing and $O(1)$ query time were given by Dietz [8] and by Berkman and Vishkin [6], though this latter algorithm has a unwieldy constant factor[1], and the former algorithm requires fancy word tricks. A substantially simplified algorithm was given by Alstrup and Holm [1], though their main focus was on dynamic trees, rather than on simplifying LA computations.

We present an algorithm that requires no "heavy" machinery. This algorithm is suitable for teaching data structures to (advanced) undergraduates, unlike previous algorithms. It is perhaps surprising that a problem that heretofore required heavy lifting can be simplified to such an extent, and the algorithm we present is made up of such simple pieces. This last point makes this algorithm particularly suitable for teaching.

The remainder of the paper is organized as follows. In Section 2, we provide some definitions and initial lemmas. In Section 3, we present an algorithm

* Supported in part by HRL Laboratories, Sandia National Laboratories, and NSF ITR grant EIA–0112849.

[1] In fact, $2^{2^{28}}$.

for Level Ancestors that takes $O(n \log n)$ for preprocessing, and $O(1)$ time for queries. In Section 4, we show how to speed up the preprocessing to an optimal $O(n)$.

2 Definitions

We begin with some basic definitions. The *depth* of a node u in tree T, denoted $\mathrm{depth}(u)$, is the shortest distance from u to the root. Thus, the root has depth 0. The *height* of a node u in tree T, denoted $\mathrm{height}(u)$, is the number of nodes on the path from u to its deepest descendant. Thus, the leaves have height 1.

Let $\mathrm{LA}_T(u, d) = v$ where v is an ancestor of u and $depth(v) = d$, if such a node exists, and `undefined` otherwise. Now we define the *Level Ancestor Problem* formally.

Problem 1. The *Level Ancestor Problem*:

Structure to Preprocess: A rooted tree T having n nodes.
Query: For node u in rooted tree T, query $\textsc{LevelAncestor}_T(u, d)$ returns $\mathrm{LA}_T(u, d)$, if it exists and `false` otherwise. Thus, $\textsc{LevelAncestor}_T(u, 0)$ returns the root, and $\textsc{LevelAncestor}_T(u, \mathrm{depth}(u))$ returns u. (When the context is clear, we drop the subscript T.)

In order to simplify the description of algorithms that have both preprocessing and query complexity, we introduce the following notation. If an algorithm has preprocessing time $f(n)$ and query time $g(n)$, we will say that the algorithm has complexity $\langle f(n), g(n)\rangle$.

One of our notational conventions, which we introduced in [2], is of independent interest.[2] We define the *hyperfloor* of x, denoted $\lfloor\lfloor x \rfloor\rfloor$, to be $2^{\lfloor \log x \rfloor}$, i.e., the largest power of two no greater than x. Thus, $x/2 < \lfloor\lfloor x \rfloor\rfloor \leq x$. Similarly, the *hyperceiling* $\lceil\lceil x \rceil\rceil$ is defined to be $2^{\lceil \log x \rceil}$.

3 An $\langle O(n \log n), O(1)\rangle$ Solution to the Level Ancestor Problem

We now present three simple algorithms for solving the Level Ancestor Problem, which we call the Table Algorithm, the Jump-Pointers Algorithm, and the Ladder Algorithm. At the end of this section we combine the two latter algorithms to obtain a solution with complexity $\langle O(n \log n), O(1)\rangle$. The Table Algorithm will be used in the faster algorithms in the next section.

3.1 The Table Algorithm: An $\langle O(n^2), O(1)\rangle$ Solution

We first observe that the Level Ancestor Problem has a solution with complexity $\langle O(n^2), O(1)\rangle$: build a table storing answers to all of the at most n^2 possible queries. Answering a Level-Ancestor query requires just one table lookup.

[2] All logarithms are base 2 if not otherwise specified.

Lemma 1. *The Table Algorithm solves the Level Ancestor Problem in time* $\langle O(n^2),\, O(1)\rangle$.

Proof. The lookup table can be filled in $O(n^2)$ by a simple dynamic program.

We make one more note here, which we use in the next section. In the table as described, we store the id of a node as the answer to a query. Instead, we introduce one level of indirection. We assign a depth first search (DFS) number to each node, and store these in the table. Then when we retrieve the DFS number of the answer, we look up the corresponding node in the tree. This extra level of indirection clearly does not increase the asymptotic bounds, but allows us to share preprocessing amongst different subtrees.

3.2 The Jump-Pointers Algorithm: An $\langle O(n \log n),\, O(\log n)\rangle$ Solution

In the Jump-Pointers Algorithm, we associate $\log n$ pointers with each vertex, which we call *jump pointers*. Jump pointers "jump" up the tree by powers of 2. Thus, there is a pointer from u to u's ℓ-th ancestor, for $\ell = 1, 2, 4, 8, \ldots, \lfloor\!\lfloor \mathrm{depth}(u) \rfloor\!\rfloor$. We refer to these pointers as $\textsc{Jump}_u[i]$, where $\textsc{Jump}_u[i] = \mathrm{LA}(u, \mathrm{depth}(u) - 2^i)$.

We emphasize the following point:

Observation 2 *In a single pointer dereference we can travel at least halfway from u to* $\mathrm{LA}(u, d)$*, for any d. Finding the appropriate pointer takes $O(1)$ time.*

Proof. We let $\delta = \mathrm{depth}(u) - d$. We can travel up by $\lfloor\!\lfloor \delta \rfloor\!\rfloor$, which is at least $\delta/2$. The pointer to follow is simply $\textsc{Jump}_u[\lfloor \log \delta \rfloor]$.

(Note that since the floor and log operations are word computations, the algorithm is a RAM algorithm.)

As a consequence of Observation 2, we obtain the following lemma:

Lemma 3. *The Jump-Pointers Algorithm solves the Level Ancestor Problem in time* $\langle O(n \log n),\, O(\log n)\rangle$.

Proof. To achieve $O(n \log n)$ preprocessing, we apply a trivial dynamic program. To answer query $\textsc{LevelAncestor}_T(u, d)$ in $O(\log n)$ time, we repeatedly follow the pointers that will get us halfway to $\mathrm{LA}_T(u, d)$. Therefore after at most $\log n$ jumps, we locate $\mathrm{LA}_T(u, d)$.

3.3 The Ladder Algorithm: An $\langle O(n),\, O(\log n)\rangle$ Solution

In the Ladder Algorithm, we decompose the tree T into (nondisjoint) paths, which we call *ladders* because they help us climb up the tree. Our choice of how we do the decomposition may seem peculiar at first, but it is an integral part of the fast algorithms.

To understand why it is advantageous to break the tree into paths, observe that solving the level ancestor problem on a single path of length n is trivial.

Observation 4 *On a path of length n, the Level-Ancestor Problem can trivially be solved with (optimal) complexity $\langle O(n), O(1)\rangle$.*

Proof. We maintain a Ladder array $\textsc{Ladder}[0 \ldots n-1]$, where the i-th array position corresponds to the depth-i node on the path. To answer $\textsc{LevelAncestor}_T(u, d)$, we return $\textsc{Ladder}[d]$, which takes $O(1)$ time.

We now describe the *ladder decomposition* of the tree T, which proceeds in two stages: The first stage requires us to find a *long-path decomposition* of the tree T, which greedily decomposes the tree into disjoint paths.

Stage 1: Long-Path Decomposition. We greedily break T into long disjoint paths as follows. We find a longest root-leaf path in T, breaking ties arbitrarily, and remove it from the tree. This removal breaks the remaining tree into subtrees $T_1, T_2, \ldots$. We recursively split these subtrees by removing their longest paths. The base case is when the tree is a single path, because the removal yields the empty tree. Note that if a node has height h, it is on a long-path with at least h nodes.

If we now apply the above ladder algorithm to each long-path, we may still have a slow algorithm. In particular, we can only jump up to the top of our long-path. Then we must step to its parent, and jump up its long path, and so forth. The time taken to reach $\mathrm{LA}(u, d)$ is the number of long-paths we must traverse. There can be as many as $\Theta(\sqrt{n})$ paths on one leaf-to-root walk.[3]

Stage 2: Extending the Long Paths into Ladders. The problem with the long-path climbing algorithm sketched above is that jumping to the ancestor at the top of a long-path may not help much. Since we have already allocated an array of length h' to a path of length h', we might as well allocate $2h'$. We do this by adding the h' immediate ancestors of the top of the path to the array.

We call these doubled long-paths *ladders*, and note that while ladders overlap, they still have total size at most $2n$. We say that *vertex v's ladder* is the ladder derived from the long path containing v, and note that since long-paths partition the tree, each node v has a unique ladder, but may be listed in many ladders.

Now, if a node has height h, we know that its ladder includes a node of height at least $2h$, or the root, which we can reach in constant time.

This observation will collapse the running time of queries, as the following lemma and corollaries shows:

Lemma 5. *Consider any vertex v of height h. The top of v's ladder is at least distance h above, that is, vertex v has at least h ancestors in its ladder.*

Proof. The top of v's long-path has height $h' \geq h$. Thus, it has h' ancestors in its ladder. Node v has $2h' - h \geq h$ ancestors in its ladder.

[3] A heavy path decomposition can reduce this number to $O(\log n)$, but will not ultimately help us for future optimizations.

Lemma 6. *The Ladder Algorithm solves the Level Ancestor Problem in time* $\langle O(n), O(\log n)\rangle$.

Proof. We find the long-path decomposition of tree T in $O(n)$ time as follows. In linear time, we preprocess the tree to compute the height of every node. Each node picks one of its maximal-height children to be its child on the long-path decomposition. Extending the paths into ladders requires another $O(n)$ time.

We now show how to answer queries. Consider any vertex u of height h. If we travel to the top of u's ladder, we reach a vertex v of height at least $2h$. Since all nodes have height at least 1, after i ladders we reach a node of height at least 2^i, and therefore we find our level ancestor after at most $\log n$ ladders and time.

3.4 Putting it Together: An $\langle O(n \log n), O(1)\rangle$ Solution

The Jump-Pointer Algorithm and the Ladder Algorithm complement each other, since the Jump-Pointer Algorithm makes exponentially decreasing hops up the tree, whereas the Ladder Algorithm makes exponentially increasing hops up the tree.

We combine these approaches into an algorithm that follows a single jump-pointer and climbs only one ladder. Since the jump-pointer transports us halfway there, the ladder climb carries us the rest of the way. Thus, we obtain the following theorem.

Theorem 1. *The Level Ancestor Problem can be solved with complexity* $\langle O(n \log n), O(1)\rangle$.

Proof. We perform the preprocessing of both the Jump-Pointer Algorithm and the Ladder Algorithm in time $O(n \log n)$.

We show that queries can be answered by following a single jump pointer and climbing a single ladder. Consider query $\text{LevelAncestor}_T(u, d)$. Let $\delta = \llbracket \text{depth}(u) - d \rrbracket$. The jump pointer leads to vertex v that has depth $\text{depth}(u) - \delta$ and height at least δ. The distance from v to $\text{LA}_T(u, d)$ is at most δ, so by Lemma 5, v's ladder includes $\text{LA}_T(u, d)$.

4 The Macro-Micro-Tree Algorithm: An $\langle O(n), O(1)\rangle$ Solution

Since ladders only take linear time to precompute, we can afford to use them in the fast solution. The bottleneck is computing jump pointers. Our first step in improving the $\langle O(n \log n), O(1)\rangle$ is to exploit the following observation.

Observation 7 *We need not assign jump pointers to a vertex if a descendant of the vertex has jump pointers. That is, if vertex w is a descendant of vertex v, then* $\text{LA}_T(v, d) = \text{LA}_T(w, d)$, *for all* $d \leq \text{depth}(v)$, *so w's jump pointers are good enough.*

Since we do not need jump pointers on all vertices, we call vertices having jump pointers assigned to them *jump nodes*. An immediate suggestion based on Observation 7 is to designate only the leaves as jump nodes. Unfortunately, this approach only speeds things up enough in the special case when the tree contains $O(n/\log n)$ leaves.

Our immediate goal is to designate $O(n/\log n)$ jump nodes that "cover" as much of the tree as possible. We define any ancestor of a jump node to be a *macro node* and all others to be *micro nodes.* The macro nodes form a connected subtree of T, which we refer to as the *macrotree*, and we define *microtrees* to be the connected components obtained by deleting all macro nodes.

We can deal with all macro nodes by slightly extending the algorithm from Theorem 1 as noted in Observation 7. We will use a different technique for microtrees.

We pick as jump nodes the maximally deep vertices having at least $\log n/4$ descendants. By maximally deep, we mean that the children of these vertices have *fewer* than $\log n/4$ descendants. The $1/4$ is carefully chosen and will come into play when we take care of microtrees.

Lemma 8. *There are at most $O(n/\log n)$ jump nodes. We can compute all jump node pointers in linear time.*

Proof. In the proof of Lemma 3, we used a simple dynamic program to compute jump pointers at every node. Here, we are only computing jump pointers at a few nodes so do not have all the intermediate values needed for the dynamic program. However, notice that for every jump node we can compute its parent in constant time. The parent has height at least 2, so its ladder will carry us another 2 nodes. We can keep jumping up ladders, and so we can compute the jump pointers for any node in $O(\log n)$ time.

4.1 Dealing with Macro Nodes

Lemma 9. *We can solve the level ancestor problem for all macro nodes in $\langle O(n), O(1)\rangle$.*

Proof. We perform a ladder decomposition and compute the jump pointers of all jump nodes in $O(n)$ time. Then, with one depth first search, we find a jump node descendant $\textsc{JumpDesc}(u)$ for each macro node u. Finally, as noted above, compute $\textsc{LevelAncestor}(u, d)$ by computing $\textsc{LevelAncestor}(\textsc{JumpDesc}(u), d)$ using Theorem 1.

4.2 Dealing with Microtrees

In short, we deal with microtrees by noting that they do not come in too many shapes, $O(\sqrt{n})$ in fact. Therefore we can make an exhaustive list of all microtree shapes and preprocess them via the Table algorithm. We show how to use the preprocessing on these canonical trees to compute level ancestors on micro nodes in T. All that remains are a few details.

Lemma 10. *Microtrees come in at most $\sqrt{n}$ shapes.*

Proof. First, recall that each microtree has fewer than $\log n/4$ vertices. For a DFS, call a *down edge* an edge being traversed from parent to child, and an *up edge* those from child to parent. The shape of a tree is characterized by the pattern of up and down edges. A microtree has fewer than $\log n/4$ edges, each of which is traversed twice. While not every pattern of up and down edges is a valid tree, every valid tree forms some such pattern, and so we have at most $2^{\log n/2} = \sqrt{n}$ possible trees. This bound is not tight, but a tighter bound is not necessary. Also, we now see why we selected $\log n/4$ as the jump node threshold, rather than, e.g., $\log n$.

We conclude with the following.

Theorem 2. *The Level Ancestor problem can be solved in $\langle O(n), O(1)\rangle$ time.*

Proof. The only thing that remains is a few details about how to handle microtrees. First, we enumerate all microtree shapes and apply the Table algorithm to these. This takes $O(\sqrt{n}\log^2 n)$ time. Furthermore, we address the tables so produced by the bit pattern of ups and downs of the DFS of the trees. Thus, we do all precomputation for all microtrees in T in $O(n)$ time.

Finally, note that in the Table Algorithm, we added one level of indirection based on DFS numbers. This means that we need only assign DFS numbers to the nodes in each microtree in T. Then, when we lookup a level ancestor in the tables, we can use the local DFS numbering to decode which actual node is the desired ancestor.

This finishes the problem of finding a level ancestor within a microtree. The other case is when a micro node wants an ancestor outside of its microtree. In this case, we can jump to the root of the microtree in constant time, and then to its parent. This will be a macro node, and so we revert to the macro node algorithm.

Summing up, the preprocessing time for micro nodes is $O(n)$, as for macro nodes, and in either case the query time is $O(1)$.

References

1. S. Alstrup and J. Holm. Improved algorithms for finding level-ancestors in dynamic trees. In *27th International Colloquium on Automata, Languages and Programming (ICALP '00), LNCS. 1853*, pages 73–84, 2000.
2. M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.
3. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN*, pages 88–94, 2000.
4. O. Berkman and U. Vishkin. Recursive *-tree parallel data-structure. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 196–202, 1989.

5. O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, Apr. 1993.
6. O. Berkman and U. Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, Apr. 1994.
7. R. Cole and R. Hariharan. Dynamic LCA queries on trees. In *Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 235–244, 1999.
8. P. F. Dietz. Finding level-ancestors in dynamic trees. In *Workshop on Algorithms and Data Structures*, pages 32–40, 1991.
9. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 135–143, 1984.
10. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
11. B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17:1253–1262, 1988.
12. R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, Oct. 1979.
13. B. Wang, J. Tsai, and Y. Chuang. The lowest common ancestor problem on a tree with unfixed root. *Information Sciences*, 119:125–130, 1999.
14. Z. Wen. New algorithms for the LCA problem and the binary tree reconstruction problem. *Inf. Process. Lett.*, 51(1):11–16, 1994.