Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 321 (2004) 5–12
Theoretical Computer Science
www.elsevier.com/locate/tcs

The Level Ancestor Problem simplified

Michael A. Bender[a,*,1], Martín Farach-Colton[b,2]

[a] *Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA*

[b] *Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA*

Received 15 August 2002; received in revised form 2 May 2003; accepted 22 May 2003

Abstract

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

Keywords: Data structures; Rooted trees; Level Ancestor Problem

1. Introduction

A fundamental algorithmic problem on trees is how to find *Level Ancestors* of nodes. A *Level Ancestor Query* LA(u,d) requests the depth d ancestor of node u. The *Level Ancestor Problem* is thus to preprocess a given n-node rooted tree T to support level ancestor queries. Both the preprocessing time and the query time must be optimized.

The natural solution of climbing up the tree from u has $O(n)$ query time, and the alternative solution of precomputing the results of all possible queries has $O(n^2)$ preprocessing. Solutions with $O(n)$ preprocessing and $O(1)$ query time were given by Dietz [4] and by Berkman and Vishkin [3], though this latter algorithm has an unwieldy constant factor,[3] and the former algorithm requires fancy word tricks. A substantially

* Corresponding author.
E-mail addresses: bender@cs.sunysb.edu (M.A. Bender), martin@farach-colton.com (M. Farach-Colton).

[1] Supported in part by Sandia National Laboratories and NSF Grants EIA-0112849, CCR-0208670, and ACI-0324974.

[2] Partially supported by NSF CCR 9820879. This work was performed while the author was working at Google, Inc.

[3] Alstrup and Holm [1] calculate this constant to be $2^{2^{28}}$.

0304-3975/$ - see front matter
doi:10.1016/j.tcs.2003.05.002

simplified algorithm was given by Alstrup and Holm [1], though their main focus was on dynamic trees, rather than on simplifying LA computations.

In this paper we present an algorithm that requires no "heavy" machinery. Our algorithm is appropriate for (advanced) undergraduates, especially because it is composed of simple components that are combined in the end.

The remainder of the paper is organized as follows. In Section 2, we provide definitions and initial lemmas. In Section 3, we present an algorithm 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 $\text{depth}(u)$, is the number of *edges* on the shortest path from u to the root. Thus, the root has depth 0. The *height* of a node u in tree T, denoted $\text{height}(u)$, is the number of *vertices* on the path from u to its deepest descendant. Thus, the leaves have height 1.

Let $\text{LA}_T(u,d) = v$ 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:
Preprocessing: A rooted tree T having n vertices.
Querying: For node u in rooted tree T, query $\text{LevelAncestor}_T(u,d)$ returns $\text{LA}_T(u,d)$, if it exists and `false`, otherwise. (When the context is clear, we drop the subscript T.)

Thus, $\text{LevelAncestor}_T(u,0)$ returns the root, and $\text{LevelAncestor}_T(u,\text{depth}(u))$ returns u.

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 [2] may be of independent interest.[4] 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 2 no greater than x. Thus, $x/2 < \lfloor\lfloor x \rfloor\rfloor \leqslant x$. 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

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

a solution with complexity $\langle O(n \log n), O(1)\rangle$. The Table Algorithm will be used in the faster algorithms in Section 4.

3.1. The Table Algorithm: an $\langle O(n^2), O(1)\rangle$ solution

As noted above, 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 one table lookup.

Lemma 2. *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 Section 4. In the lookup table as described, we store the label 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 does not increase the asymptotic bounds, but will allow 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 up to $\log n$ pointers, which we call *jump pointers*, with each vertex. Jump pointers "jump" up the tree by powers of 2; there is a pointer from u to u's ℓth ancestor, for $\ell = 1, 2, 4, 8, \ldots, \lfloor\lfloor \text{depth}(u) \rfloor\rfloor$. Thus, the ith jump pointer, denoted $\text{JUMP}_u[i]$, points to the 2^ith ancestor of u, that is, $\text{JUMP}_u[i] = \text{LA}(u, \text{depth}(u) - 2^i)$.

We emphasize the following point:

Observation 3. *In a single pointer dereference we can travel at least halfway from u to* $\text{LA}(u, d)$, *for any* d. *Finding the appropriate pointer takes* $O(1)$ *time.*

Proof. We let $\delta = \text{depth}(u) - d$. We can travel up by $\lfloor\lfloor \delta \rfloor\rfloor$, which is at least $\delta/2$. The pointer to follow is $\text{JUMP}_u\left[\lfloor \log \delta \rfloor\right]$. □

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

As a consequence of Observation 3, we obtain the following lemma:

Lemma 4. *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 $\text{LEVELANCESTOR}_T(u,d)$ in $O(\log n)$ time, we repeatedly follow the pointers that travels at least halfway to $\text{LA}_T(u,d)$. Therefore after at most $\log n$ jumps, we locate $\text{LA}_T(u,d)$. □

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*.

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 in (optimal) complexity $\langle O(n), O(1)\rangle$ is trivial. We maintain an array $\text{LADDER}[0 \ldots n-1]$, where the ith array position corresponds to the depth-i node on the path. To answer $\text{LEVELANCESTOR}_T(u,d)$, we return $\text{LADDER}[d]$ (which takes $O(1)$ time).

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

3.3.1. Stage 1: long-path decomposition

Greedily break T into long disjoint paths as follows. 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 $T_1, T_2, \ldots$. Recursively split these subtrees by removing their longest root-leaf paths. The base case is when the tree is a single path, because the removal yields the empty forest. Note that if a node has height h, it is on a long path containing at least h nodes.

If we now put each long path into a LADDER array, 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 p, and jump up p's long path, and so forth. The time taken to reach $\text{LA}(u,d)$ 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, which yields an $\langle O(n), O(\sqrt{n})\rangle$ algorithm. [5]

3.3.2. Stage 2: extending the long paths into ladders

We have already allocated an array of length h' to a path of length h'. Now, we allocate $2h'$ by adding the h' immediate ancestors at the top of the path to the array. We call these doubled long-paths *ladders*; 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.

Doubling the ladder yields the following key properties, which we will use to speed up queries.

[5] A heavy path decomposition can reduce this number to $O(\log n)$, which yields an $\langle O(n), O(\log n)\rangle$ algorithm. However, we do not know how to speed up the heavy-path-based algorithm without substantial complications, while the long path decomposition admits an elegant improvement.

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

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

Corollary 6. *If a node v has height h, then v's ladder includes a node of height $2h$ or it includes the root.*

The properties from Lemma 5 and Corollary 6 are the basis for the *Ladder Algorithm*, in which we repeatedly climbing ladders until we reach the queried vertex.

Lemma 7. *The Ladder Algorithm solves the Level Ancestor Problem in time* $\langle O(n), O(\log n)\rangle$.

Proof. Find the long-path decomposition of tree T in $O(n)$ time as follows. In linear time, 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: the jump-pointer transports us halfway there, the ladder climb carries us the rest of the way. Thus, we obtain the following theorem.

Theorem 8. *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 $O(n \log n)$.

Queries can be answered by following a single jump pointer and climbing a single ladder. Consider query $\text{LevelAncestor}_T(u,d)$. Let $\tau = \lfloor\lfloor \text{depth}(u) - d \rfloor\rfloor$. The jump pointer leads to vertex v that has depth $\text{depth}(u) - \tau$ and height at least τ. The distance from v to $\text{LA}_T(u,d)$ is at most τ, 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 the $O(n \log n)$ jump pointers. Our first step in improving the $\langle O(n \log n), O(1)\rangle$ is to exploit the following observation.

Observation 9. *We need not assign jump pointers to a vertex v if a descendant w of v has jump pointers because $LA_T(v,d) = LA_T(w,d)$, for all $d \leqslant \text{depth}(v)$.*

Since we do not need jump pointers on all vertices, we call vertices having jump pointers assigned to them *jump nodes*. A suggestion based on Observation 9 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 8 as noted in Observation 9. This extension requires one depth first search to find a jump-node descendant for each macronode. We will use a different technique for microtrees.

4.1. Dealing with jump nodes

We pick as jump nodes the maximally deep vertices having at least $\log n/4$ descendants. By maximally deep, we mean that each child of these vertices has *fewer* than $\log n/4$ descendants. The $\frac{1}{4}$ will come into play when we take care of microtrees.

Lemma 10. *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 4, we used a simple dynamic program to compute jump pointers at every node. Here, we only compute jump pointers at a few nodes so do not have all the intermediate values needed for the dynamic program. However, we can compute the parent of every jump node in constant time. The parent has height at least 2, so its ladder will carry us another 2 nodes. We keep jumping up ladders, and so compute the jump pointers for any node in $O(\log n)$ time. □

4.2. Dealing with macro nodes

Lemma 11. *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 JumpDesc(u) for each macro node u. Finally, by Observation 9, compute LevelAncestor(u, d) by computing LevelAncestor(JumpDesc(u), d) using one jump pointer and one ladder, as in Theorem 8. □

4.3. Dealing with microtrees

In short, we use the standard data structural technique [5] of enumerating the solutions for all *small* instances of the problem. In particular, we note that microtrees do not come in too many shapes, $O(\sqrt{n})$ in fact. Therefore, we make an exhaustive list of all microtree shapes and preprocess them via the Table algorithm. We use the preprocessing on these canonical trees to compute level ancestors on micro nodes in T. All that remains are a few details.

Lemma 12. *Microtrees come in at most $\sqrt{n}$ shapes.*

Proof. There is a direct existential proof for this bound using Catalan numbers. However, we prefer the following constructive proof, since it yields a particular $\log n/2$-bit encoding to be used in our algorithm.

First, recall that each microtree has fewer than $\log n/4$ vertices. For a DFS, call a *down edge* an edge traversed from parent to child, and an *up edge* one traversed from child to parent. The shape of a tree is completely determined by the pattern of up and down edges. A microtree has fewer than $\log n/4$ edges, each of which is traversed twice.

Consider any length $2 \log n/4$ bit pattern c. Bit pattern c determines a microtree as follows: Let each 0 represent a down edge and each 1 represent an up edge on the DFS. Consider the longest prefix pre(c) of c that encodes a valid DFS of a tree. This (possibly empty) prefix pre(c) must satisfy two properties:

(1) Every prefix of prefix pre(c) must have at least as many 0's as 1's.
(2) Prefix pre(c) must have an equal number of 0's and 1's.

Thus, bit pattern c encodes the tree whose DFS is encoded by pre(c). (If pre(c) is the empty string, then c encodes a one-node tree.)

Note that every micro tree can be encoded by (at least one) bit pattern c. Thus, there are at most $2^{\log n/2} = \sqrt{n}$ possible trees. □

We conclude with the following.

Theorem 13. *The Level Ancestor problem can be solved in $\langle O(n), O(1)\rangle$ time.*

Proof. First, we enumerate all microtree shapes and apply the Table algorithm to each of these. This takes $O(\sqrt{n} \log^2 n)$ time. Thus, we do all precomputation for all microtree shapes in T in $O(n)$ time.

Recall that in the Table Algorithm, we added one level of indirection based on DFS numbers. For each microtree, we compile a mapping from DFS number to nodes. Then, when we look up a level ancestor in the tables, we can use the local DFS mapping to determine the desired ancestor.

This finishes the problem of finding a level ancestor within a microtree. The micro node may want an ancestor outside of its microtree. In this case, we 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 it is for macro nodes, and in either case the query time is $O(1)$. □

References

[1] S. Alstrup, J. Holm, Improved algorithms for finding level-ancestors in dynamic trees, in: 27th Internat. Colloquium on Automata, Languages and Programming (ICALP), Geneva, Switzerland, Lecture Notes in Comput. Sci. Vol. 1853, Springer, Berlin, 2000, pp. 73–84.

[2] M.A. Bender, E. Demaine, M. Farach-Colton, Cache-oblivious B-trees, in: 41st Annual Symp. on Foundations of Computer Science (FOCS), 2000, pp. 399–409.

[3] O. Berkman, U. Vishkin, Finding level-ancestors in trees, J. Comput. System Sci. 48 (2) (1994) 214–230.

[4] P.F. Dietz, Finding level-ancestors in dynamic trees, in: Workshop on Algorithms and Data Structures (WADS), Ottawa, Canada, 1991, pp. 32–40.

[5] H.N. Gabow, R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. System Sci. 30 (2) (1985) 209–221.