Optimal algorithms for finding nearest common ancestors in dynamic trees

Stephen Alstrup[*]

Abstract

We consider the problem of finding the nearest common ancestor of two given nodes x and y (denoted by $nca(x, y)$) in a collection of dynamic rooted trees. Interspersed with nca-queries are on-line commands $link(x, y)$ where x but not necessarily y is a tree root. The effect of a command $link(x, y)$ is to combine the trees containing x and y by making y the parent of x. This problem was originally proposed by Aho, Hopcroft and Ullman (SIAM J. Comput. 5(1), 115-132, 1976). We present a pointer machine algorithm which performs n link and m nca in time $O(n + m \log\log n)$, matching a lower-bound by Harel and Tarjan (SIAM J. Comput. 13(2), 338-355, 1984). Improving the best known result $O((n + m)\log n)$, of Sleator and Tarjan (J. Comput. System Sci. 26(3), 362-391, 1983).

1 Introduction

Aho, Hopcroft and Ullman [2] consider the following problem: Given a collection of rooted trees, answer queries of the form, "What is the nearest common ancestor (nca) of vertices x and y". They consider three different versions of the problem (1, 2 and 5 listed below), which are more or less dynamic depending upon whether the queries are all specified in advance and how much the trees change during the course of the queries. The following five problems are considered:

- **Problem 1 (off-line).** The collection of trees is static and the entire sequence of queries is specified in advance.
- **Problem 2 (static).** The collection of trees is static but the queries are given on-line, so each query must be answered before the next one is known.
- **Problem 3 (link root).** The queries are given on-line. Interspersed with the queries are on-line commands of the form $link\_root(x, y)$ where x and y are tree roots. The effect of a command $link\_root(x, y)$ is to combine the trees containing x and y by making y the parent of x.
- **Problem 4 (add leaf).** The queries are given on-line on a tree T. Interspersed with the queries are on-line commands of the form $add\_leaf(x, y)$ where y is any node in T and x is a new leaf. The effect of a command $add\_leaf(x, y)$ is to insert x as a new leaf in T by making y the parent of x.
- **Problem 5 (link).** The queries are given on-line. Interspersed with the queries are on-line commands of the form $link(x, y)$ where x but not necessarily y is a tree root. The effect of a command $link(x, y)$ is to combine the trees containing x and y by making y the parent of x.

[*]E-mail:stephen@diku.dk. W^3-homepage : http://www.diku.dk/users/stephen/. Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark.

Through this paper n is the numbers of nodes in the collection of trees and m the numbers of *nca*-queries. Notice that the number of nodes limits the number of tree operations *add_leaf*, *link* and *link_root*. The problems 1-5 have been investigated both for RAM [1] and pointer machine [10]. In a pointer machine, memory consists of a collection of records and each record consists of a fixed number of cells. The fields have associated types, such as *pointer*, *integer* and *real*, and access to memory is only possible through *pointers* and not by address arithmetic. In a RAM the memory is an array of words, where each word holds an integer expressed in binary. A pointer machine differs from a RAM because address arithmetic is not possible on a pointer machine which implies that hash table e.g. is not possible. Harel and Tarjan [7] show that a pointer machine requires $\Omega(\log\log n)$ time in worst-case per query for the problems 2-5. On the other hand they give a RAM-algorithm for problem 2, which requires only $O(1)$ time to solve each query. For measuring time we use the *uniform cost measure*, in which all operations require $O(1)$ time. Several papers improve and extend the results from [2]. The best known results for problems 1-5 are summarizes in table 1.

	Pointer machine		RAM machine	
Problem	Algorithm	Time	Algorithm	Time
1. off-line	Tarjan [9]	$O((n+m)\alpha(n+m,n))$	Harel and Tarjan [7]	$O(n+m)$
2. static	Leeuwen and Tsakalidis [11]	$O(n+m\log\log n)$	Harel and Tarjan [7]	$O(n+m)$
3. link root	Leeuwen [13]	$O((n+m)\log\log n)$	Harel and Tarjan [7]	$O((n+m)\alpha(n+m,n))$
4. add leaf	Tsakalidis [12]	$O(n+m\log n)$	Gabow [5]	$O(n+m)$
5. link	Sleator and Tarjan [8]	$O((n+m)\log n)$	Gabow [5]	$O((n+m)\alpha(n+m,n))$
Optimal $O(n+m\log\log n)$ time algorithms is given to problems 3-5 for pointer machine				

Table 1. Fastest algorithm 1996. All the algorithms use $O(n)$ space

In this paper we give an optimal $O(n+m\log\log n)$ algorithm for problem 5 for a pointer machine and thereby improving the best known results for problems 3-5 for pointer machine. The lower bound $\Omega(\log\log n)$ for each query from [7] show that the algorithm is optimal. The given algorithm complexity is fully amortized but we show that we can achieve $O(\log\log n)$ for each query for problem 4[1]. In section 2 we describe the overall model that we use to construct algorithms. In section 3 an optimal algorithm for problem 4 is given. In section 4 we extend this solution to an optimal algorithm for problem 5. In the conclusion open problems for RAM and pointer machines is given.

2 A model for the algorithms

The algorithm in this paper follow a variant of the scheme of a MicroMacroUnivers, as proposed for different RAM algorithms [6]. This universe is build by partitioned the set of nodes from a original tree T into disjoint subsets, where each subset induces a subtree of T, called a microtree. The partitioned can be thought of as removing edges from T, leaving a forest of microtrees. The root from each microtree is inserted in a macrotree. The macrotree contains an edge between two nodes *iff* T has a path between the two nodes which do not contain any other microtree roots. To maintain this structure we use three algorithms.

[1] In [3] is given an $O(n)$ algorithm for n *add_leaf* and *link_root* where each *nca*-query is bounded by $O(\log\log n)$.

- A MacroAlgorithm to the macrotree which is optimal to *nca*-queries.
- A MicroAlgorithm to the microtrees which is optimal to tree operations.
- An optimal algorithm which combine the Micro- and MacroAlgorithm.

Notation : We denote the original trees T, the microtrees T' and the macrotrees T''. If a node is inserted in a secondary structure (e.g. microtree) a new node in the secondary structure is created and pointers are set between the two nodes in the primary (e.g. original tree) and secondary structure. If it not is obvious from context in which tree an operation is used then subscript is used: $p_{T'}(x)$ is the parent to x in the microtree T'. A node x in a microtree have a pointer, $root_{T'}(x)$, to the root in the microtree where to it belongs. To decide nca_T we use an extended *nca*-operation, $nca*$, in the macrotree. Let $a = nca(x,y)$. For $v = x, y$ let a_v be the ancestor of v immediately preceding a; if $a = v$ then $a_v = a$. Now we define $nca^*(x,y)$ as the triple (a, a_x, a_y). With these operations $nca_T(x,y)$ can be found as followed.

```
1. if $root_{T'}(x) = root_{T'}(y)$ then return $nca_{T'}(x,y)$;
2. $(a, a_1, a_2) = nca^*_{T''}(root_{T'}(x), root_{T'}(y))$;
3. if $a = a_1$ then $x' := x$ else $x' := p_T(a_1)$;
4. if $a = a_2$ then $y' := y$ else $y' := p_T(a_2)$;
5. return $nca_{T'}(x', y')$;
```

Algorithm 1

3 An optimal algorithm for the operation add leaf

In this section we present an optimal $O(n + m \log\log n)$ algorithm for *add_leaf*, where n is the number of nodes in the tree. The previous best algorithm of Tsakalidis has complexity $O(n + m \log h)$, where h is the height of the tree. We could use Tsakalidis algorithm as a subroutine (MicroAlgorithm) for our optimal algorithm, but because we will need a more general MicroAlgorithm, we present our own (and more simple[2]) in subsection 3.2. Furthermore we use an $O(n \log n + m \log\log n)$ MacroAlgorithm for n *add_leaf* and m nca^*-queries, which is presented in subsection 3.1.
The Micro- and MacroAlgorithm are giving an optimal algorithm together as follows. The nodes in T have a field $size_{T'}$, where $size_{T'}(x)$ is the number of the nodes in the microtree with root x; if x is not a microtree root then $size_{T'}(x)$ is undefined. If a microtree contains $l = 1 + \lceil \log n \rceil$ nodes it is denoted as full and otherwise as not full[3]. When a new node x is inserted as a leaf with parent y ($add\_leaf(x,y)$) it is inserted in the same microtree as the parent, if the microtree which y belongs to is not full ($size_{T'}(root_{T'}(y)) < l$). Otherwise a new microtree is created with the root x. In both cases the fields $size_{T'}$ and $root_{T'}$ is easily updated. Only root nodes from full microtrees are inserted in the macrotree. The microtree root z is inserted when its microtree become full ($size_{T'}(z) = l$). Let r be the root in the original tree T, then the macrotree T'' at any time will be determined by the edges $T'' = \{(v, root_{T'}(p_T(v))) | v \in T \backslash \{r\} \wedge root_{T'}(v) = v \wedge size_{T'}(v) = l\}$. With this structure $nca_T(x,y)$ can be found using algorithm 1 except in cases where nodes are in not full microtrees; but if x is in a not full microtree different from y microtree ($root_{T'}(x) \neq root_{T'}(y)$) then $nca(x,y)_T = nca_T(p_T(root_{T'}(x)), y)$.

[2] e.g. Tsakalidis use 18 pages to describe the algorithm.
[3] If the total number of nodes not is known in advance then n is guessed to be a constant and each times the number of nodes extend the guess we double the guess and reconstruct the structure.

Theorem 1 *Given a $O(n + m \log h)$ MicroAlgorithm for n add_leaf and m nca where h is the height of the tree AND a $O(n \log n + m \log logn)$ MacroAlgorithm for n add_leaf and m nca* we can construct an $O(n + m \log\log n)$ algorithm for n add_leaf and m nca.*

Proof (sketch). An *add_leaf* command takes (amortized) constant time except for insertion in the macrotree; but only root nodes from full microtrees are inserted in a macrotree. Because full microtrees have size $l = 1 + \lceil \log n \rceil$ at most$\lfloor n/l \rfloor$ nodes is in a macrotree which give the complexity $O((n/l)\log(n/l)) = O(n)$. A *nca*-query can result in at most 6 constant time calls to $root_{T'}$, an $O(\log\log n)$ query in a macrotree and a query in a microtree. The query in a microtree takes a time logarithmic to the height of the microtree which is $l = O(\log n)$. □

3.1 MacroAlgorithm for the operation add leaf

In this section we present an $O(n \log n + m \log\log n)$ algorithm for n *add_leaf* and m *nca** in a tree T and show how to extend it to a MacroAlgorithm for *link*. In order to compute *nca* in a static tree (problem 2) Harel and Tarjan [7] use a secondary compressed C-tree. We use a similar tree modified for the dynamic algorithm. In the C-tree are the same nodes as in T, but C is constructed such that its height is logarithmic to the number of nodes. The C-tree is established from T by classifying the edges in T as either heavy or light. If an edge $(v, p_T(v))$ is heavy then v is denoted as the heavy child of $p_T(v)$ and otherwise as a light child of $p_T(v)$. The heavy and light edges in T are chosen to meet the following conditions :

Condition 1 Each node in T which have a child have exactly one heavy child.

Condition 2 A heavy child v to a node $p_T(v)$ is chosen such that for any other child z of $p_T(v)$ it is true that $(5/4) * size_T(z) < size_T(p_T(v))$, where *size* is the number of decedents to a node.

The heavy edges in T partitioned the nodes in T on heavy paths such that each node belongs to precisely one heavy path (a leaf which is not a heavy child is a heavy path with one node). The node on a heavy path which is nearest the root of the tree is the *apex* node for the heavy path and $apex(v)$ is the *apex* node on the heavy path containing the node v. The tree C for the tree T with root r can now be described as the edges $C = \{(v, apex(p_T(v)) | v \in T \backslash \{r\}\}$.

Lemma 1 *Let h be the height of C then* $h \in O(\log n)$.

Proof. A path in C from a leaf v to the root contains exactly all the light nodes from v to the root plus node v if it is light, hence $depth(v) \le \lceil \log_{(5/4)} n \rceil$ by condition 2.□

The main idea to compute $nca*_T(x, y)$ is to use the C-tree to find the two nodes x' and y' which are respectively the first ancestor of x and y on the heavy path containing $apex(nca(x, y))$. To determine *nca* for two nodes on the same heavy path, we associate a d-label to each node, $d(v) \in \mathcal{Z}$, such that for any two nodes on the same heavy path, we have $d(w) < d(v)$ if depth of w is less than depth of v. Notice that the *apex* node for a node x is x if x is root or a light child; otherwise it is $p_C(x)$. To maintain the C-tree we use the MicroAlgorithm described in subsection 3.2, with complexity $O(n + m \log h)$ for n *add_leaf* and *delete_leaf* AND m *nca**. An algorithm for computing for $(b, b_1, b_2) = nca*_T(x, y)$

```
1. $(a, a_1, a_2) := nca*_C(x, y)$. {if $x \neq y$ then $a = apex(nca_T(x, y))$ };
2. if $(a = x)$ or $(a = y)$ then begin $b := a$; goto 6; end;
```

```
3. if $apex(a_1) \neq a$ then $a'_1 := p_T(a_1)$ else $a'_1 := a_1; \{= x\}$
4. if $apex(a_2) \neq a$ then $a'_2 := p_T(a_2)$ else $a'_2 := a_2; \{= y\}$
5. if $d(a'_1) < d(a'_2)$ then $b := a'_1$ else $b := a'_2$;
6. if $x = b$ then $b_1 := b$ else if $p_T(a_1) = b$ then $b_1 := a_1$ else $b_1 := heavychild(b)$;
7. if $y = b$ then $b_2 := b$ else if $p_T(a_2) = b$ then $b_2 := a_2$ else $b_2 := heavychild(b)$;
```

Algorithm 2

Lemma 2 *Given a $O(n + m \log h)$ MicroAlgorithm we can construct a MacroAlgorithm where each nca* is bound by $O(\log \log n)$.*

Proof. Except from constant time operations a $nca*_T$-query is reduced to a $nca*_C$ query which have the complexity $O(\log h)$ and from lemma 1 $h \in O(\log n)$ so $nca*_T$ is bound by $O(\log \log n)$. The proof for the correctness of algorithm 2 is omitted but the crucial point is, that unless (b_1, b) (or (b_2, b)) is a light edge then $b_1(b_2)$ must be the heavy child of b.□

We now show how to maintain the structure when adding a leaf. The hard part is condition 2 – to make sure that the C-tree height is logarithmic to the number of nodes. To fulfill this it is necessary to control if a light edge have to change to heavy. From the definition of the C-tree we have that for a light child v that $size_T(v) = size_C(v)$. The field $size_C$ can easily be updated : Adding a leaf x to C we update $size_C$ with 1 for each of x ancestors in C. Because C hight is limited it takes $O(\log n)$ time. We can not afford to control a light child each time it is updated but by condition 2 we only have to control a node when its $size_C$ has doubled. The nodes in C get an extra field named $nextsize_C$ and each time $nextsize_C = size_C$ we control if the node should be heavy; if not we double $nextsize_C$. When controlling if a node v should be heavy we can not relay on $size_C$ to determine $size_T$ for the heavychild of $p_T(v)$. Therefore the control is done by counting the decedents to the heavychild in T. For this purpose we need *left-* and *rightsibling* pointers to insert siblings in a double circular list. We now give the full algorithm for inserting a new leaf x.

1. Insert x in C. {and ensuring condition 1 }
2. Update $size_C$ for all ancestors to $p_T(x)$ in C with $size_C(x)$. {=1, if x is a new leaf}.
3. Control if $p_T(x)$ have an ancestor in C except from the root where $size_C \geq nextsize_C$. If such a node exist the node w with least depth is chosen and the light edge $(w, p_T(w))$ is updated in the next steps.
4. Let $v = heavychild(p_T(w))$. {Because w is an *apex* its parent must have another heavy child}.
5. Control if $size_T(w) \geq 2 * size_T(v)$ by counting at most $\lceil size_C(w)/2 \rceil$ decedents of v.
6. If $size_T(w) \geq 2 * size_T(v)$ then remove the decedent of v and w in T from C with $delete\_leaf_C$ and make the edge $(w, p_T(w))$ heavy. The remove nodes d, $size_C$ etc. fields are updated and the nodes are inserted in C again with add_leaf_C. { In the next subsection we show that the MicroAlgorithm also support *delete_leaf* in $O(1)$ }
7. Let z be that of the two nodes v and w which is light after step 6. Set $nextsize(z) = 2 * max\{nextsize_C(z), size_C(heavychild(p_T(z)))\}$.

Algorithm 3

Lemma 3 *The MacroAlgorithm have the complexity $O(n \log n + m \log \log n)$ for n add_leaf and m nca*.*

Proof. From lemma 2 we achieve the $O(\log\log n)$ bound for nca*. Next we consider the complexity of counting decedents in T and adding/removing of leaves in C. Recall that the C-tree is maintained by the $O(n + m\log h)$ MicroAlgorithm. Each time an edge $(w, p_T(w))$ is controlled we have to count up to $\lceil size_C(w)/2\rceil$ nodes and update at most $2 * size_T(w)$ nodes in T and C which cost O($size_T(w)$). Let m_0 be the root of T and let it have children $m_1, m_2 \cdots m_L$ where $size_T(m_1) \geq size_T(m_2) \geq \cdots \geq size_T(m_L)$. Let $size_T(m_0) = n$, $size_T(m_1) = \alpha$ and $\sum_{i=2}^{L} size_T(m_i) = \beta$, hence $n = \alpha + \beta + 1$. Between two times a node is controlled its size have at least been doubled and the last time a node m_i, $i > 1$, is controlled its size is at most $size_T(m_i)$ and the last time node m_1 is controlled its size is at most $4 * \beta$. This implies that the total amount of work for updating the children to m_0 is $O(\beta + \sum_{j=2}^{L}\sum_{i=1}^{\lfloor\log_2 size_T(m_j)\rfloor} 2^i) = O(\beta)$. For each of the trees $T^{m_2}, T^{m_3} \cdots T^{m_L}$ we have that they are at most half so big as the tree T^{m_0} and these trees contain β nodes which establish the complexity $O(n\log n)$ for all updates in T.□

Given a detailed description of the MacroAlgorithm for *add_leaf* we now shortly show how to construct a $O(n\log n + m\log\log n)$ MacroAlgorithm for n link and m *nca**. We will use the same structure so we only have to show how to perform $link(x, y)$. Recall that x but not necessarily y is a tree root and notice that the root node yr in the tree to which y belongs can be determined in $O(\log n)$ by following the path $y \cdots yr$ in C. If $size_C(x) < size_C(yr)$ we simply process algorithm 3 (step 2-6) with the exception that if the tree is not updated then the nodes from C^x is inserted in the compressed tree C^{yr}. Next we show how to process when $size_C(x) \geq size_C(yr)$. Recall that when a node is inserted in a secondary structure we actually create a new node and set up two pointers between the node in the primary and secondary structure, hence two nodes can in $O(1)$ time exchange nodes in the secondary structure by moving only four pointers; let us call this operation an exchange operation. We are now able to use a variant of a technique from [5] to link two trees. The main idea is to let the path $x \cdots yr$ be one heavy path. This can be done in three steps. 1) Exchange pointers for node x and yr, 2) Add node x to the compressed tree as a leaf to node yr in the compressed tree and 3) Update and insert in C the former proper decedents to yr. Step 3 can be done by d-labelling the nodes on the heavy path $P = y..yr$ with labels $d(x) - 1 \cdots d(x) - |P|$ inserting the nodes on the path in C by adding them as leaves to yr and then updating (including d-labelling and insertion in C) these nodes proper decedents in a breadth-first manner.

Lemma 4 *The MacroAlgorithm have complexity $O(n\log n + m\log\log n)$ for n link and m nca*.*

Proof. Each times two trees are linked the smallest tree is updated and in the first case nodes from the greatest tree can be updated if an edge is controlled. To update the smallest trees takes $O(n\log n)$ for n *link* and to update light edges does not exceed this complexity which can be seen from the proof of the former lemma. The easy proof of correctness is omitted. □

3.2 MicroAlgorithm for the operation add leaf

In this subsection we describe a $O(n + m\log h)$ MicroAlgorithm for n *add_leaf* and *delete_leaf* AND m *nca** where h is the hight of the tree, which is a variant of an algorithm from [2]. First we give an $O(n\log n + m\log h)$ algorithm for n *add_leaf* and *delete_leaf* AND m *nca**. To compute *nca** in a tree T we use a kind of binary search in the tree. For this purpose each node has a double linked list L connected to it with length $\lceil\log_2 n\rceil$ containing information about ancestors. Let L[0] denote the first post in the list and L[i] the i'th post in the list. Beside sibling pointers (*left and right*) do each post contain a Q-pointer. The Q-pointer in post L[i] for a node x points to the post L[i] for the ancestor to x in depth $depth(x) - 2^i$; if the node does not exist then the Q-pointer is NULL. For a node x do $right(x)$ point to its first post in the list L and $left(\mathrm{L}[0])$ point to the node in the tree. With this structure can we fast compute $ancestor(x, d)$ which is the ancestor to x with depth $depth(x) - d$; if $depth(x) - d \leq 0$ then $ancestor(x, d) = x$.

Lemma 5 *We can construct an $O(\log d)$ algorithm for ancestor(x,d).*

Proof. Let $d = (b_i 2^i b_{i-1} 2^{i-1} \dots b_0 2^0)_2$. To find $ancestor(x, d)$ we first travel through the $(i+1)$ *right*-pointers from x to the post L[i] for x. Next we for $k = i$ to 0 follow the $left$ siblings if $b_k = 0$ or otherwise $(b_k = 1)$ we follow the Q-pointer and then a left pointer. □

Next we show how to compute $nca*$. Assume $depth(x) \geq depth(y)$. In order to compute $nca(x, y)$ we use parallel binary search from the nodes y and $x' = ancestor(x, depth(x) - depth(y))$. Let $d = depth(x') = depth(y) = (b_i 2^i, b_{i-1} 2^{i-1} \cdots b_0 2^0)_2$ and let L_a[j] be the j'th post for node a. Set $\mathrm{L}_{x'}$=L_x[i] and L_y=L_y[i] and for $k = i$ to 0 do process as follow. If $\mathrm{L}_{x'} \neq \mathrm{L}_y$ then set $\mathrm{L}_a = left(\mathrm{Q}(\mathrm{L}_a))$ and otherwise set $\mathrm{L}_a = left(\mathrm{L}_a)$ for x' and y. After deciding $nca(x, y)$ it is trivial to extend the solution to $nca*$ using the ancestor procedure (for implementation details see [3]).

Lemma 6 *We can construct a $(\log h)$ algorithm for nca*.*

Proof. Omitted. □

When inserting a new leaf x we have to set its depth and construct a new list L beside the parent pointer. The first post in the list L is easy set and Q(L[i]) is set to right(Q(Q(L[i-1]))).

Lemma 7 *We can construct an $O(n \log n + m \log h)$ algorithm for m nca* AND n add_leaf and delete_leaf.*

Proof. Omitted. □

Next we show how to convert this algorithm to an optimal MicroAlgorithm. The main idea is that if the height h of the final tree is known in advance then the length of the list can be limited to $l = \lceil \log_2(h+1) \rceil$ and we can limit the number of nodes with a list to $O(n/l)$[4]. The limitation of nodes with a list is done as follows. Only nodes where $(depth \bmod l) = 0$ have a list and a node is not connected to a list until it gets a decedent in depth $2 * l - 1$ and the list is not removed until all its decedents have depth less than $l - 1$. Let nodes with a list be called Lnodes. Beside the tree T we now maintain a tree T^α with edges

$$\mathrm{T}^\alpha = \{(v, w) | v, w \in T; v, w \text{ are Lnodes and } w \text{ is } v \text{ first Lnode ancestor in T}\}$$

Let the tree T^α be maintained by the algorithm defined above in this subsection. We now give the algorithm to find $nca*_T(x, y)$ using the tree T^α. Let each node in T have a pointer FL to its first ancestor in depth i where $(i \bmod l) = 0$. First we find the first Lnodes ancestors x' and y' respectively to x and y in constant time. Either $x' = y'$ and the search-space is reduced to $O(l) = O(\log h)$ or we find $nca*_{T^\alpha}(x', y')$ and the search-space is again reduced to $O(l)$ (implementations details is given in [3]). When inserting a node in depth $d \geq 2 * l$ where $(d \bmod l) = 0$ we connect a list to its ancestor in depth $d - 2 * l$ if it does not already have one and we increment $\mathrm{R}(\mathrm{FL}(x))$ with one, so $\mathrm{R}(y)$ is the number of decedents to y in depth $depth(y) + l$. When deleting a node x in depth $d \geq l$ we decrement $\mathrm{R}(\mathrm{FL}(x))$ with one and if $\mathrm{R}(\mathrm{FL}(x))$=0 the list is removed from the node $\mathrm{FL}(x)$.

[4]Both the micro- and macrotrees height is known in advance to be logarithmic to the number of nodes.

Lemma 8 *The MicroAlgorithm have the complexity $O(n + m \log h)$ for m nca* AND n add_leaf and delete_leaf. If k is the actually number of nodes in the tree the MicroAlgorithm is using O(k) space.*

Proof. Except from constant time operations a $nca^*{}_T$ query is reduced to a $nca^*{}_{T^\alpha}$ query with complexity $O(\log h)$ and a travelling of at most $O(\log h)$ nodes. To any time in a tree with k nodes at most $\lfloor k/l \rfloor - 1$ nodes have a list of length l which give the space complexity $O(k)$. A node is only (un)connected to a list if l of its decedents (which have no influence on other nodes) have been removed or added since the last time the node was (un)connected to a list.□

4 A optimal algorithm for the operation link

In this section we describe an $O(n + m \log\log n)$ algorithm for n *link* and m *nca*. The algorithm is using the model of a MicroMacroUnivers. As MacroAlgorithm we will use the $O(n \log n + m \log\log n)$ for n *link* and m *nca** as described in the end of subsection 3.1. As MicroAlgorithm we will use the optimal algorithm from the previous section. This implies that the MicroAlgorithm used in this section performs n *add_leaf* and m *nca* in $O(n + m \log\log n)$-time. Using the optimal algorithm for *add_leaf* as a MicroAlgorithm for *link* it is not necessary to limit the size of the microtrees, so now the only problem is to make sure that they are big enough so the macrotrees is limited - and this is the new problem with the operation link. We solve the problem by partitioning the nodes in a tree in two levels such that a path from a node in level 1 to the root node contains only level 1 nodes. Removing the nodes in level 1 from a tree leave a forest of trees, each of these trees will be denoted as subtrees of the original tree. Each of the subtrees will be partitioned in microtrees and connected with a macrotree such that each of the subtrees in the forest is a single MicroMacroUnivers. For two nodes in the same subtree *nca* can be found by algorithm 1 in $O(\log\log n)$ so the only remaining problem for *nca* is a fast *nca*-algorithm for nodes in level 1. For this purpose we construct a LevelAlgorithm. The final optimal algorithm is then constructed by using a slightly modified LevelAlgorithm on it self.

4.1 LevelAlgorithm

In this subsection an $O(n + m \log n)$ algorithm for n *link* and m *nca* is described. The nodes in a tree T are partitioned in two levels:

Level 1 Level 1 contains up to n nodes. The nodes in level 1 are not inserted in any secondary structure but are marked as belonging to level 1.

Level 2 Removing the nodes in level 1 from a tree T leave the nodes in level 2 in a forest of trees. Each of these trees will be denoted as subtrees of T and for a node x in level 2 is $sub_T(x)$ the root of the subtree. Each of the subtrees is treated in the following way : The nodes are partitioned in microtrees and the microtrees are maintained by the optimal algorithm for *add_leaf*. Each node in a microtree have a pointer $root_{T'}$ which point to the root of the microtree to which the node belongs. Each of the microtree can contain up to n nodes and contain at least $l = 1 + \lceil \log n \rceil$. The microtree root nodes are inserted as usually in a macrotree which is maintained by the algorithm last in subsection 3.1.

First we describe how to decide *nca* in this universe and next how to *link* two trees. If two nodes x and y are in the same subtree ($sub_T(x) = sub_T(y)$) then

$nca(x, y)$ is found using algorithm 1[5]. If they are not in the same subtree the problem is reduced to find nca for two nodes in level 1 (if one of the nodes, say x, are in level 2 then substitute it with $p_T(sub_T(x))$). Determination of $nca(x, y)$ for two nodes in level 1 is done by travelling from each of the nodes to the root node. If the travelling contains at least $l = 1 + \lceil \log n \rceil$ nodes the nodes from level 1 are inserted in a new microtree and the root is inserted in a new macrotree. When collecting the nodes in level 1 we also collect all roots from the subtrees in level 2 in a list L at the same time (because we have to collect nodes we need three more pointers : *leftsibling* and *rightsibling* pointers so siblings are in a double circular list and a *child* pointer so we from a node can reach one of its child). Now it only remainins to link all the macrotrees to one macrotree, this is done as follows : let r be the root in T then $link_{T''}(v, r)$ for each node in L.

Lemma 9 *Except from the time charged to link micro- and macrotrees a nca query takes $O(\log n)$.*

Proof. Determination of nca for two nodes in the same subtree have the complexity $O(\log\log n)$ using algorithm 1. Determination of nca for two nodes in level 1 takes $O(\log n)$ if the nodes are not collected in a microtree but if so this time is charged to link micro- and macrotrees. □

Now we describe how to $link$ two trees. Recall that $link(x, y)$ combines two trees by making x a child of y in the tree T (by setting the pointers p, *child*, $leftsibling$ and *rightsibling*). To maintain the other trees, we split up in cases depending on which of the two levels the two nodes can be in. If both nodes are in level 1 or x are in level 2 and y is in level 1 we are done. If both nodes are in level 2 we combine the macrotrees to which they belong by the command $link_{T''}(x, root_{T'}(y))$ (x is root in a macrotree and y must belong to a microtree which root is a node in another macrotree). The last case is when x is in level 1 and y in level 2. All the nodes from level 1 in T^x are inserted in the microtree which y belongs. This can been done in a breadth-first manner search in T^x adding the nodes with the command $add\_leaf_{T'}$. While inserting the nodes from level 1 in the microtree we also collect the roots from the subtrees in T^x in a list L at the same time, these nodes are also the root nodes from the macrotrees belonging to T^x which have to be linked below the node $root_{T'}(y)$. This is done by the operation $link_{T''}(v, root_{T'}(y))$ for each node in L.

Lemma 10 *The LevelAlgorithm have complexity $O(n + m \log n)$ for n link and m nca.*

Proof. We already have argument by lemma 9 that nca takes $O(\log n)$ unless the nodes are inserted in a microtree. The $link$ commands takes constant time except for updating the micro- and macrotrees. So we only have to proof that maintaining micro- and macrotrees takes linear time. The nodes are only inserted once in a microtree so this part takes $O(n)$ using the optimal $O(n + m \log\log n)$ algorithm from last section to the microtrees (see theorem 1). The MacroAlgorithm have the complexity $O(n \log n + m \log\log n)$ for n *link* and m *nca*∗ (see lemma 4). Only root nodes from microtrees are inserted in a macrotree and a microtree contains at least $l = 1 + \lceil \log n \rceil$ nodes so the number of nodes in macrotrees are at most $O(n/\log n)$ giving the complexity $O(n)$ to maintain the macrotrees.□

4.2 Combining

Now we are able to describe the optimal algorithm for $link$. The construction of the algorithm is done by using the LevelAlgorithm as follows. We maintain the same

[5]The root node of a subtree can be deduced from a node in $O(\log\log n)$ using binary search "ancestor function" from the node $root_{T'}(x)$ in the macrotree, see lemma 5.

structure as in the last subsection but now the nodes in level 1 are inserted in a secondary structure which is maintained by the LevelAlgorithm. The LevelAlgorithm determines *nca* in $O(\log n)$ time so the remaining problem is to limit the number of nodes in level 1 to $O(\log n)$ and this is the new problem. When linking two nodes x and y both in level 1 there is no direct way from y to the root so it is not possible to update the numbers of nodes in level 1 in constant time. To make a direct way to the root we could let the nodes in level 1 be in a set (UNION-FIND) but this would give the complexity $O(n\alpha(n,n) + m \log\log n)$. Our method to remove the inverse Ackerman function build on the following technique. The LevelAlgorithm used on nodes in level 1 is modified in such a way that a command to it will either respond as normally or it will report that there is to many nodes. This modification can be done on the LevelAlgorithm as follows. When linking two macrotrees we now control that the size of the two macrotrees together not extend $\log n / \log\log n$. Because each node in a macrotree represent a microtree with a least $\log\log n$ nodes a failure report from the LevelAlgorithm implies that level 1 contains at least $\log n$ nodes which again implies that there are enough nodes in level 1 to collect them in to a new microtree.

Theorem 2 *We can construct an $O(n + m \log\log n)$ algorithm for n link and m nca.*

Proof. The only non-linear algorithm used by the LevelAlgorithm is the MacroAlgorithm with the the complexity $O(n \log n)$ for n link, but with the control instants there are at most $\log n / \log\log n$ nodes in one macrotree which achieve the bound $O(n)$.$\Box$

The two algorithms given above offer different worst-case for on *nca*-query. Using UNION-FIND we get $O(\log n)$ and without $O(n)$. A third possibility is to use a technique from Gabow [5]. Given an algorithm with the complexity $O(f(n) + g(m))$ for m *nca** and n *add_leaf* and *add_root*[6] it is showed how to construct and $O(f(n)\alpha(L,n) + Lg(m))$ algorithm where the worst-case for one *nca* is $O(L\beta)$ where $O(\beta)$ is the worst-case for *nca* for the previous algorithm. In [3] it is showed how to construct a $O(n + m \log\log n)$ algorithm for m *nca** and n *link_root* and *add_leaf* where worst-case for one *nca* is $O(\log\log n)$. This gives three alternatives which are collected in table 2.

Amortize complexity for n link and m nca	Worst-case for one nca
$O(n + m \log\log n)$	$O(n)$
$O(n\alpha(n,n) + m \log\log n)$	$O(\log n)$
$O(n\alpha(L,n) + Lm \log\log n)$	$O(L \log\log n)$

Table 2. Three alternatives to link.

5 Conclusion

In this paper we have given a $O(n+m \log\log n)$ algorithm for n *link* and m *nca* and thereby improving to optimal the algorithms to problem 3-5 for pointer machine in table 2. Mike Fredman has pointed out that the result from [4] can be extended to show that Gabow algorithm for *link* is optimal for the cell probe model of computation. Tarjan's off-line algorithm for pointer machine is a direct application of the UNION-FIND algorithm and therefore in some sense also optimal. This leaves only the RAM algorithm for problem 3 to be either optimize or proofed optimal. In [7] do Harel and Tarjan also consider the *cut*-operation where *cut*(x) split a tree in two

[6] The effect of the command *add_root*(x, y) where x is a single node and y is the root of a tree is the tree constructed by setting x as a parent of y .

trees by removing the edge $(x, p_T(x))$. The fastest algorithm [8] for both a RAM and pointer machine have the complexity $O(m \log n)$ for m query, *cut* and *link* operation in a forest with n nodes. Harel and Tarjan conjecture (but do not proof) that this is optimal for a pointer machine, but is it also optimal for a RAM machine?

Acknowledgement. I thanks Martin Appel, Peter W. Lauridsen and Mikkel Thorup for reading preprint of this paper and giving many good suggestions.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. On finding lowest common ancestor in trees. *SIAM Journal on computing*, 5(1):115–132, 1976. See also STOC 1973.

[3] S. Alstrup. Optimal algorithms for finding nearest common ancestors in dynamic trees. Technical Report 95-30, Department of Computer Science, University of Copenhagen, 1995.

[4] M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Annual ACM symp. on theory of comp. (STOC)*, pages 345–354, 1989.

[5] H.N. Gabow. Data structure for weighted matching and nearest common ancestors with linking. In *Annual ACM-SIAM Symposium on discrete algorithms (SODA)*, volume 1, pages 434–443, 1990.

[6] H.N. Gabow and R.E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, 1985.

[7] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *Siam J. Comput*, 13(2):338–355, 1984.

[8] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983. See also STOC 1981.

[9] R.E. Tarjan. Applications of path compression on balanced trees. *Journal of the association for computing machinery (J.ACM)*, 26(4):690–715, 1979.

[10] R.E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of computer and system sciences*, 18(2):110–127, 1979.

[11] A.K. Tsakalides and J. van Leeuwen. An optimal pointer machine algorithm for finding nearest common ansectors. Technical Report RUU-CS-88-17, Department of Computer Science, University of Utrecht, 1988.

[12] A.K. Tsakalidis. The nearest common ancestor in a dynamic tree. *Acta informatica*, (25):37–54, 1988.

[13] J. van Leeuwen. Finding lowest common ancestors in less than logarithmic time. Unpublish technical report, 1976.