Nearest Common Ancestors : A survey and a new distributed algorithm

Stephen Alstrup* Cyril Gavoille† Haim Kaplan‡ Theis Rauhe§

Abstract

Several papers describe linear time algorithms to preprocess a tree, such that one can answer subsequent nearest common ancestor queries in constant time. Here, we survey these algorithms and related results. A common idea used by all the algorithms for the problem is that a solution for complete balanced binary trees is straightforward. Furthermore, for complete balanced binary trees we can easily solve the problem in a distributed way by labeling the nodes of the tree such that from the labels of two nodes alone one can compute the label of their nearest common ancestor. Whether it is possible to distribute the data structure into short labels associated with the nodes is important for several applications such as routing. Therefore, related labeling problems have recieved a lot of attention recently.

Previous optimal algorithms for nearest common ancestor queries work using some mapping from a general tree to a completely balanced binary tree. However, it is not clear how to distribute the data structures obtained via these mappings. We conclude our survey with a new simple algorithm that labels the nodes of a rooted tree such that from the labels of two nodes alone one can compute in constant time the label of their nearest common ancestor. The labels assigned by our algorithm are of size $O(\log n)$ bits where n is the number of nodes in the tree. The algorithm runs in $O(n)$ time.

1 Introduction

Let T be a rooted tree. A node $x \in T$ is an *ancestor* of a node $y \in T$ if the path from the root of T to y goes through x. A node $v \in T$ is a *common ancestor* of x and y if it is an ancestor of both x and y. The *nearest common ancestor*, NCA, of two nodes x, y is the common ancestor of x and y whose distance to x (and to y) is smaller than the distance to x of any other common ancestor of x and y. We denote the NCA of x and y by $\mathrm{nca}(x, y)$. The problem of efficiently computing nearest common ancestors has been studied extensively over the last three decades in an online and offline settings, and in various models of computation (See e.g. [45, 2, 3, 35, 58, 16, 7]).

The problem of finding NCAs has numerous applications. A procedure solving it is used by algorithms for finding a maximum weighted matching in a graph [31], a minimum spanning tree in a graph [38, 19], and a dominator tree in a directed flowgraph [3]. It is also proved useful in several string algorithms [33, 23], for dynamic planarity testing [67], in network routing [62], for solving various geometric problems [12] including range searching [30], for finding evolutionary trees [24], and in bounded treewidth algorithms [13].

*The IT University of Copenhagen, Denmark, E-Mail: `stephen@it-c.dk`
†LaBRI - Université de Bordeaux, France, E-Mail: `gavoille@labri.fr`
‡School of computer science, Tel Aviv University, Tel Aviv, Israel, E-Mail: `haimk@math.tau.ac.il`
§The IT University of Copenhagen, Denmark, E-Mail: `theis@it-c.dk`

One of the most fundamental results on computing NCAs is that of Harel and Tarjan [35, 34]. Harel and Tarjan describe a linear time algorithm to preprocess a tree and build a data structure that allows subsequent NCA queries to be answered in constant time. Following this result of Harel and Tarjan's several simpler algorithms with essentially the same properties but better constant factors have been proposed [51, 44, 30, 58, 9] (See also Section 2.2). These algorithms, including the one of Harel and Tarjan, use the observation that it is rather easy to solve the problem when the input tree is a completely balanced binary tree.

To solve the problem when the input is a completely balanced binary tree one has to label the nodes by their index in an inorder traversal of the tree. If the tree has n nodes each such number occupies $\ell = \lceil \log n \rceil$ bits[1]. Let inorder(x) and inorder(y) be the inorder indexes of x and y, respectively. Let i be the maximum among 1) the index of the leftmost bit in which inorder(x) and inorder(y) differ, 2) the index of the rightmost 1 in inorder(x), 3) the index of the rightmost 1 in inorder(y). It is easy to prove by induction that the inorder number of $\text{nca}(x, y)$ consists of leftmost $\ell - i$ bits of inorder(x) (or inorder(y) if the maximum above was the index of the rightmost 1 in inorder(y)) followed by a 1 and $i - 1$ zeros.

Note that this algorithm for completely balanced binary trees is distributed in the sense that it constructs the inorder index of $\text{nca}(x, y)$ from the inorder indices of x and y alone without accessing the original tree or any other global data structure. In case the application does not identify nodes by their inorder indices then tables converting a node identifiers to inorder indices and vice versa have to be constructed. In the common case where a node is identified by a pointer to a structure representing it then we can simply store the inorder index as an extra field in that structure. This will allow to get the inorder index of a node giving a pointer to its corresponding structure in constant time. For mapping the inorder index of the NCA back to a pointer to the corresponding node a table of $O(2^\ell) = O(n)$ entries can be easily constructed.

The algorithms in [35, 51, 44, 30, 58, 9] for general trees all work using some mapping of the tree to a completely balanced binary tree. Thereby exploiting the fact that for completely balanced binary trees the problem is easier. Different algorithms differ by the way they do the mapping. Unfortunately, all algorithm, as a result of doing the mapping, have to use some precomputed auxiliary data structures in addition to the labels of the nodes, in order to compute NCAs. In contrast with the algorithm for completely binary trees these algorithms for general trees do not allow to compute an unique identifier of $\text{nca}(x, y)$ from short labels associated with x and y alone.

In this paper we show how to label the nodes of an arbitrary tree so that from the labels of x and y alone one can determine the label of $\text{nca}(x, y)$. In particular we constructively prove the following theorem.

Theorem 1 *There is a linear time algorithm that labels the n nodes of a rooted tree T with labels of length $O(\log n)$ bits such that from the labels of $x, y \in T$, one can compute in constant time the label of* $\text{nca}(x, y)$.

Our algorithm is as simple as the simpler among the non distributed algorithms mentioned above. In a scenario when nodes have to be identified by some predetermined identifiers one can use our algorithm together with a table converting labels to those predetermined identifiers and vice versa. Thus our algorithm provides an alternative to any of the algorithms mentioned above also in a non distributed settings. Theorem 1 should put in contrast with a recent lower bound of Peleg [50]. Peleg shows that labels which allow to compute directly a $\Theta(\log n)$ bit predetermined identifier of $\text{nca}(x, y)$ from the labels of x and y alone must be of length $\Omega(\log^2 n)$ bits. So to obtain a theorem like Theorem 1 one has to exploit the freedom to choose the identifiers of the nodes. A

[1]All the logarithms are in base two.

distinguished feature in our algorithm is its use of alphabetic codes [32] to generalize the inorder approach for completely balanced binary trees.

A labeling scheme to identify NCAs may be useful in routing messages on tree networks. When a message has to be sent from node a x to node y in a tree it has to go through $\text{nca}(x, y)$. Therefore the ability to compute the identifier of $\text{nca}(x, y)$ from the identifiers of x and y may prove useful. Our particular NCA labeling scheme also allows to identify the first edge on the shortest path from x to y from the labels of x and y alone. In that sense it generalizes recent labeling schemes for routing on trees [25, 62] (see also Section 3). Another possible application of our algorithm arises in XML search engines. Such search engines typically maintain a reverse index. This index is a hash table mapping each word or a name of a tag to all XML-documents containing it. The engine can exploit the fact that an XML-document is essentially a tree and label each such tree using our algorithm. Then it can attach to each occurrence of a word in a document the label of the corresponding node. By doing that the engine can process sophisticated queries by accessing only the hash table rather than the documents themselves.

Outline of the paper: In Section 2 we survey previous algorithm for computings NCA's, including related results and applications. One of the most important applications of NCA's is range searching. Up to now the connection between range searching and NCA have been given through several reductions. In the survey we make observations which give a more direct relations between these two problems. In Section 4 we prove Theorem 1. Since our new algorithm makes it possible to identify the NCA from the two input nodes alone, its potential applications may avoid several lookups to perhaps expensive external memory. Further motivation for our new approach is given in Section 3.

2 NCA and related results

First we describe the relationship between NCA's and range searching, since the most simple NCA algorithms are constructed using this relationship. Next we survey previous NCA algorithms for the RAM model of computation where the tree is static. Finally, we list results for finding NCA's in other models of computation and in cases where the topology of the tree can change. We also discuss few closely related problems.

2.1 NCA and discrete range searching

Gabow, Bentley, and Tarjan [30] observed that the one-dimensional Discrete Range Searching (DRS) problem, is equivalent to the NCA problem. The DRS problem is defined as follows. Given a sequence of real numbers $x_1, \ldots, x_n$, preprocess the sequence so that one can answer efficiently subsequent queries of the form "given a pair of indices (i, j), what is the index of a maximum element among $x_i, \ldots, x_j$?". We denote such query by $\max(i, j)$. The DRS problem is a fundamental geometric searching problem and orthogonal range searching problems in two and higher dimensions are often reduced to DRS [30].

Gabow et al. (see also [40]) reduce the DRS problem to the NCA problem by constructing a Cartesian tree for the sequence $x_1, \ldots, x_n$, as defined by Vuillemin [65]. The *Cartesian tree* of the sequence $x_1, \ldots, x_n$ is a binary tree with n nodes each containing a number x_i. Let $x_j = \max\{x_1, \ldots, x_n\}$. The root of the Cartesian tree for $x_1, \ldots, x_n$ contains x_j (and possibly also the index j). The left subtree of the root is a Cartesian tree for $x_1, \ldots, x_{j-1}$, and the right subtree of the root is a Cartesian tree for $x_{j+1}, \ldots, x_n$. Vuillemin shows how to construct the Cartesian tree of $x_1, \ldots, x_n$ in $O(n)$ time. It is easy to see that the maximum among $x_i, \ldots, x_j$ corresponds to the NCA of the node containing x_i and the node containing x_j.

Gabow et al. also show how to reduce the NCA problem to the DRS problem. Given a tree they first construct a sequence of its nodes by doing a depth first traversal [60]. Each time we visit a node we add it to the end of the sequence so each node appears in the sequence as many times as its degree. Note that this sequence is an Euler tour of the tree. From the Euler tour they obtain a sequence of integers by associating the depth of node with each of its occurrences. To compute $\text{nca}(x, y)$ we pick arbitrary two elements x_i, x_j in the sequence representing x and y, respectively, and compute the index of the maximum among $x_i, \ldots, x_j$. It is easy to see that the node corresponding to this maximum element is $\text{nca}(x, y)$.

Combining the equivalence between the NCA problem and the DRS problem with Theorem 1 we obtain the following corollary.

Corollary 2 *Let $x_1, \ldots, x_n$ be a sequence of n real numbers. We can assign in linear time a label of length $O(\log n)$ bits to each element, such that, given the labels of x_i, x_j, the label of a maximum among $x_i, \ldots, x_j$ can be computed in constant time from the labels of x_i and x_j alone.*

If the value x_i is required rather than its label we need one lookup in a global table to map a label to a real number. We can implement this mapping either using sufficient space, using linear space and expected linear preprocessing time [28], or using linear space with $O(n \log n)$ deterministic preprocessing time [48].

2.2 Previous static algorithms for computing NCA

As mentioned in Section 1, Harel and Tarjan [35] were the first who described how to preprocess a tree in linear time such that one can answer NCA queries in constant time per query. However they already point out in [35] that one could simplify some parts of their algorithm. Subsequently, Scheiber and Vishkin [58] contracted some of the preprocessing steps of Harel and Tarjan into a single step and obtained a simpler algorithm which they could also parallelize easily. Powell [51] describes a simplification of the algorithm of Scheiber and Vishkin.

Berkman and Vishkin [9] describe a simpler algorithm which is based on the equivalence between the NCA problem and the DRS problem. Let $x_1, \ldots, x_n$ be the input to the DRS problem. We build a complete binary tree with leaves $x_1, \ldots, x_n$ from left to right. With each internal node we associate a *left value* which is the maximum among the values in its left subtree and a *right value* which is the maximum among the values in its right subtree. We also associate with each leaf l two tables each with $\log n$ entries, denoted by right_l and left_l. For a leaf l the value $\text{right}_l(i)$ is the maximum among the right values of the i closest ancestors of l. The value of $\text{left}_l(i)$ is defined similarly. To answer a query of the form $\max(i, j)$ we find the depth d of $\text{nca}(x_i, x_j)$ using a simple NCA algorithm for complete binary trees. Then we return the maximum among $\text{right}_{x_i}(d-1)$, and $\text{left}_{x_j}(d-1)$. Obviously, the drawback of this simple algorithm is that it requires $O(n \log n)$ preprocessing time and space.

To overcome this difficulty Berkman and Vishkin observed that the sequence of depths of nodes on an Euler tour of the tree (obtained by reducing the NCA problem to the DRS problem) consists of integers such that the absolute difference between consecutive integers is at most 1. Based on this observation they suggested a three level algorithm for the DRS (and the NCA) problem which runs in linear time. In their scheme the sequence is partitioned into *blocks* each of size $O(\log n)$, and each block is further partitioned into *microblocks* each of size $O(\log \log n)$. They use the observation that $|x_i - x_{i+1}| = 1$, for $1 \leq i \leq n-1$, to precompute the answers to all possible queries inside all possible microblocks in one large table. Then they apply the nonlinear algorithm described above to each sequence of microblocks in a block, and to the sequence of blocks, representing each block or microblock by the maximum element in it. Since the number of microblocks in a block is $\log n / \log\log n$ and the number of blocks is $n/\log n$ the resulting structures are of linear size. We answer a maximum query in constant time using the algorithm above on the relevant blocks and two lookups in the table of precomputed answers to queries in microblocks. Farach and Bender [44] show that in fact one can in linear time precompute all answers to queries inside blocks if the size of the block is $\frac{1}{2}\log n$. Therefore the microblock level of [9] is not needed and a simpler two level algorithm based on the same principals exists.

The algorithm of Berkman and Vishkin implicitly reduce the NCA problem for general trees to the NCA problem for completely balanced binary trees. By solving the NCA problem for completely balanced binary trees they obtain an algorithm for DRS on inputs where $|x_i - x_{i+1}| = 1$, for $1 \leq i \leq n-1$. Then they reduce the general NCA problem to this special case of DRS. Using the reduction from DRS to NCA described in Section 1 this also implies a solution to DRS for general sequences.

Last, we observe that in fact one can solve the DRS problem for general sequences directly, without going through the long reduction sequence to the special case of DRS just described. The solution to DRS described above exploited the restriction that $|x_i - x_{i+1}| = 1$ only to solve small subproblems of size $\frac{1}{2}\log n$. Alternatively, we can solve a DRS problem $x_1, \ldots, x_m$ of size $m = \lfloor \log n \rfloor$ as follows. For each x_i we compute $g(x_i) = \max\{k \mid k < i \text{ and } x_k > x_i\} \cup \{-1\}$. It is not hard to see that one can compute $g(x_i)$ for every $1 \leq i \leq m$, in $O(m)$ time by maintaining the largest elements in all suffixes of the sequence on a stack. Based on the values of $g(x_i)$ we associate a label, $l(x_i)$, of size m bits with x_i, for every $1 \leq i \leq m$. The jth bit of $l(x_i)$ corresponds to element x_j, $1 \leq j \leq m$, and it is set if and only if $j < i$, $x_j > x_i$, and for every x_k, $j < k < i$, $x_k < x_j$. We compute these labels recursively as follows. The label $l(x_1)$ is 0. For $i > 1$, $l(x_i)$ is the same as $l(x_{g(x_i)})$ but with bit $g(x_i)$ also set. Now for $i < j$, we find $\max(i,j)$ from $l(x_j)$ as follows. We clear all bits with index smaller than i in $l(x_j)$, getting a word w. Then we return x_j if $w = 0$, and otherwise we return $x_{\mathrm{lsb}(w)}$, where $\mathrm{lsb}(w)$ is the index of the least significant bit in w[2]. Using this technique for the blocks we obtain a simpler DRS and NCA algorithms. None of these algorithms however implies Theorem 1 or Corollary 2. If we try to distribute the data structures of any of the algorithms described so far so that it is possible to answer queries from the labels of the corresponding nodes alone, then the shortest labels we can get consists of $\Omega(\log^2 n)$ bits.

A somewhat different and interesting approach for computing NCAs on trees of depth $O(\log n)$ has been suggested by King [39]. King's algorithm labels each edge of the tree by either 0 or 1 randomly by flipping a fair coin. Then we label each node v with the concatenation of the labels of the edges on the path from the root to v. To find $\mathrm{nca}(x,y)$ we first find the length, d, of the longest common prefix of the label of x and the label of y. Let $\mathrm{ancestor}(x,d)$ be the ancestor of depth d of x. If $\mathrm{ancestor}(x,d)$ is also an ancestor of y then it is the NCA of x and y. Otherwise for $i = 1, 2, \ldots$ we check whether $\mathrm{ancestor}(x, d-i)$ is an ancestor of y until we find such a node which must also be the NCA of x and y. It is easy to see that we would have to perform k ancestor queries with probability at most $1/2^k$. Therefore, on average, we need to perform a constant number of ancestor queries that involve ancestors of x at a particular level. The drawback of this scheme is that it requires external data structures to compute the ancestor of a node with a specified depth and to answer ancestor queries [53].

[2] As previous NCA algorithms (e.g. [35, 9]) we assume that bit operations, on words of size $\log n$, can be precomputed in $O(n)$ time it they not are supported to begin with.

2.3 Dynamic NCA problems and different models for computation

All algorithms mentioned above for the online NCA problem use random access to achieve constant query time. In contrast, on a pointer machine, Harel and Tarjan [35] proved a lower bound of $\Omega(\log \log n)$ on the query time. This matches a corresponding upper bound of Van Leeuwen and Tsakalidis [63]. This lower bound stems from the need to answer a query by giving a pointer to the corresponding node. If some other identifier of the NCA is as good then the $\Omega(\log \log n)$ lower bound can be beaten. Indeed our new algorithm can run on a pointer machine and return the *label* of the NCA in constant time.

The NCA problem is of interest also in an offline settings where the list of, say m, queries is given in advance as input. Recently Buchsbaum et al. [11] gave an $O(m+n)$ time algorithm for the offline NCA problem on a pointer machine, improving by a factor of inverse ackermann's function the previous best result [3, 61].

Back with respect to the online NCA problem, extensive research has also been done on how to allow update operations on the tree that are interspersed between NCA queries. Gabow [31] and Alstrup and Thorup [7] show how to add new leafs to the tree in amortized constant time while keeping the constant worst-case query time on a RAM, or the $O(\log \log n)$ worst-case query time on pointer machine. Cole and Hariharan [16] gave an algorithm with *worst-case* constant update time for a RAM. The problem of computing NCAs in a forest of n nodes subject to dynamic updates has also been studied. Alstrup and Thorup [7] considered updates in which a root of one tree is made a child of a root of another tree. They gave a pointer machine algorithm with amortized constant update time and $O(\log \log n)$ worst-case query time. For a RAM, Harel and Tarjan [35] gave an algorithm which for m operations in a forest of n nodes use time $O(n + m\alpha(m + n, n))$ time. If we allow roots to be linked the problem is at least as hard as the Union-Find problem [31]. Thus the amortized inverse ackermann lower bound in [29] is applicable.

A more general update operation on a forest would allow to make any node in one tree a parent of a root of another tree. Alstrup and Thorup in [7] give an algorithm for a pointer machine using $O(n + m \log \log n)$ time. This algorithm uses in the worst case $O(n)$ time both for queries and updates. For the RAM, an optimal algorithm which uses $O(i)$ worst case time for queries and $O(a(i, n))$ amortized time for updates, where i is arbitrary and a the row inverse Ackerman function, is given in [31]. This tradeoff matches the lower bound given in [4]. If we allow also to delete edges, pointer machine algorithm is given in [59] using $O(\log n)$ time per operation, where n is the size of the tree(s) involved in the operation. Other updates have also recieved some attention in connection with NCA, such as splitting an edge by inserting a new node on the edge [67, 16].

Let $level(x, i)$ return the ancestor to x which depth is i less than the depth of x. If we substitute NCA with *level* for the different dynamic settings the same bounds mentioned above hold for the RAM [18, 5, 10, 59]. For a pointer machine, however, we have a trivial lower bound on $\Omega(\log n)$ even for the static case, hence for the pointer machine Sleator and Tarjan's dynamic trees [59] give optimal results.

People have also studied the parallel and distributed complexities [8, 52, 64, 42, 43, 57, 54, 55, 41, 15, 56, 66, 58], and the I/O-complexity [14] of several of the above problems. Some natural extensions of the problems to rooted directed trees [47] and graphs [21, 22] have also been given.

3 Labeling schemes for distributed environment

Motivated by applications in the construction of XML search engines and network routing [1, 37, 49] labeling schemes that allow ancestor queries have been recently developed. Santoro and Khatib [53] suggested to label the leaves of the tree from left to right by consecutive integers, and then to label each internal node by the pair of the labels of its leftmost and rightmost leaf descendants. One can then answer ancestor queries by checking if the corresponding intervals are nested. Clearly the maximum length of a label according to this scheme is $2 \log n$ bits. Recently, Alstrup and Rauhe [6] building upon the work of Abiteboul, Kaplan, and Milo [1] gave a more complicated recursive labeling scheme for ancestor queries that generates labels of length at most $\log n + O(\sqrt{\log n})$ bits. Labeling schemes for parent queries, and other functions have also been studied [36, 37, 50]. Unfortunately these labeling schemes for ancestor queries do not allow to identify the nearest common ancestor of x and y when they are unrelated. The labeling scheme that we describe in this paper do allow this extra functionality.

In [62] and [25] it is demonstrated how compact labelings of a tree, can improve routing in trees and graphs. In the routing problem one assigns two labels to every node of the graph. The first label is is the *address* of the node whereas the second label is a data structure called *local routing table*. The labels are assigned such that at every source node x, and for every destination node y one can find the first edge (or an identifier of that edge) on a shortest path from x to y from the local routing table of x and the address of y. So the path from x to y is built in a distributed way by all intermediate nodes encountered along the way. The goal is to obtain such labeling with labels as small as possible.

Cowen in [17] shows how to construct a labeling for trees that uses $3 \log n$ bits for the addresses and $O(\min\{d \log n, \sqrt{n} \log n\})$ bits for the local routing table, where d is the degree of the node. In [20], Eilam et al. show that any labeling in which the length of the addresses is at most $\log n$ bits will require $\Omega(\sqrt{n})$ bits for the routing table in some trees. However, Gavoille and Fraigniaud [25] showed that $c \log n$ bits, for a small constant c, are enough to encode both the address and the local routing table of each node of the tree. Thorup and Zwick [62] even show that both the address and the local routing table of each node can be encoded together using $\log n + O(\log n / \log \log n)$ bits (so $c = 1 + o(1)$ suffices). Thorup and Zwick also show how to use their tree labeling to construct labeling for general graphs based on tree covering.

The labeling schemes for routing mentioned above do not allow to determine NCAs without some extensions. The labeling scheme that we suggest in this paper, however, allows not only to identify $\mathrm{nca}(x, y)$ from the labels of x and y but also to identify the first edge on the shortest path from x to y. Therefore it is more general than the routing labeling schemes.

A particular routing context where a labeling schemes for NCA may prove useful is the design of routing labeling schemes for graphs that are close to a tree, say c-decomposable graphs [26, 27] for $c = O(1)$. These graphs admit a tree-decomposition T where each node of T represents a separator of the graph of size at most c. So, one can use a compact labeling of T in order to determine the nearest separator S between the source x and the destination y. If x is contained in the component $X \in T$, and y is contained in the component $Y \in T$, then S is precisely the nearest common ancestor of X and Y. Since, any path from x to y have to cross some node of S, computing $\mathrm{nca}(X, Y)$ from the labels of x and y may simplify the local routing tables.

4 A NCA algorithm for distributed environment

4.1 Preliminaries

We denote by $\langle y \rangle_k$ a sequence of objects $y_1, y_2, \ldots, y_k$ (such as integers or binary strings). For binary strings $a, b \in \{0,1\}^*$, $a <_{\mathrm{lex}} b$ if and only if a precedes b in the lexicographic order on binary strings. I.e., a is prefix of b or the first bit in which a and b differ is 0 in a and 1 in b. An *alphabetic* sequence $\langle y \rangle_k$ is a sequence of binary strings $\langle b \rangle_k$, $b_i \in \{0,1\}^*$, where $b_i <_{\text{lex}} b_j$, for all $1 \le i < j \le k$. Let $|s|$ denote the length of a binary string $s \in \{0,1\}^*$. Observe that given machine words that contain a, b, $|a|$, and $|b|$, respectively, in their least significant bits, it is possible to determine whether $a <_{\text{lex}} b$ in a constant number of operations. (For instance, first align a and b by shifting to the left the smallest string, and then use standard integer comparison operators on the resulting words. If the two words are then equal – e.g., a was a prefix of b – break the tie according to the length of a and b). When the strings $\langle b \rangle_k$ are also prefix-free (no string is a prefix of another) we call $\langle b \rangle_k$ an *alphabetic code*. The following lemma due to Gilbert and Moore [32] states the result which we need for the alphabetic codes.

Lemma 3 (Gilbert and Moore [32]) *A sequence $\langle y \rangle_k$ of positive numbers with $n = \sum_{i=1}^{k} y_i$ has an alphabetic code $\langle b \rangle_k$ where $|b_i| \le \log n - \log y_i + O(1)$ for all i.*

In fact for our purposes it would suffice that $\langle b \rangle_k$ is alphabetic (strings can be prefixes of one another). We can construct an alphabetic sequence satisfying the length bounds in Lemma 3 for an integer sequence $\langle y \rangle_k$ is in $O(k)$ time as follows. Let $s_i = \sum_{j=1}^{i} y_j$, $i = 1, \ldots, k-1$, and let $s_0 = 0$. Also let $I_i = [s_{i-1}+1, s_{i-1}+y_i]$, $i = 1, \ldots, k$.

and $f_i = \lfloor \log y_i \rfloor$. In the interval I_i there must be a number z_i such that $z_i \bmod 2^{f_i} = 0$. If $s_{i-1}+1 \bmod 2^{f_i} = 0$, then $z_i = s_{i-1}+1$. Otherwise $z_i = s_{i-1}+1-((s_{i-1}+1) \bmod 2^{f_i}) + 2^{f_i}$. Hence, z_i can be represented in a word with $w = \lceil \log n \rceil$ bits, having the f_i less significant bits set to 0. Then we can let b_i be the bit string consisting of the $w - f_i$ most significant bit from z_i. Thus we get alphabetic codes where $1 \le |b_i| \le \log n - \log y_i + 2$. The algorithm runs in $O(k)$ time if machine operations to compute shifting, modulus, and discrete logarithm on $O(\log n)$ bit words are supported. In a machine that does not support such operations we can using $O(n)$ prepossessing time and space construct a table representing these functions. This will only increase the preprocessing time of our labeling algorithm which is described in Section 4.2 by a constant factor. Note that Mehlhorn [46] gives a somewhat more complicated algorithm that produce an *alphabetic code*, $\langle b \rangle_k$, for an arbitrary sequence $\langle y \rangle_k$ of positive real numbers with the same bound on the lengths, i.e., $|b_i| \le \log n - \log y_i + 2$. Mehlhorn's algorithm can be implemented to run in $O(k)$ time.

4.2 Labeling for NCA queries

For a tree T, let $|T|$ denote its number of nodes. Let T_v be the subtree rooted by v, and let $\text{size}(v) = |T_v|$. As in [35] we divide the tree into disjoint paths. Let $\text{parent}(v)$ be the parent of v, and $\text{children}(v)$ be the set of children of v. We classify each node of T as either *heavy* or *light* as follows. The root is light. For each internal node v, we pick a child w of v, where $\text{size}(w) = \max\{\text{size}(z) \mid z \in \text{children}(v)\}$ and classify w as *heavy*. We classify each of the remaining children of v as *light*. We call an edge to a light child a *light edge*, and an edge to a heavy child a *heavy edge*. For a node v with a heavy child w, let $\text{lsize}(v) = \text{size}(v) - \text{size}(w)$. The nearest ancestor of v which is light (possibly v itself if v is light) is denoted by $\text{apex}(v)$. By removing the light edges T is partition into paths, which we call *heavy paths*. A node w belongs to the same heavy path as the nodes of the set $\text{HP}(w) = \{v \mid v \in T, \text{apex}(v) = \text{apex}(w)\}$. See Figure 1.

First we traverse the tree once to compute in linear time $\text{size}(v)$, $\text{lsize}(v)$, $\text{apex}(v)$ and the partition of T into heavy paths. Next we assign a label, $\text{hlabel}(v)$, to each node $v \in T$, and a label $\text{llabel}(v)$ to each light node v. These labels are defined as follows.

For the root r, $\text{llabel}(r)$ is the empty string. Then, for each light node $w \neq r$, $\text{llabel}(w)$ is a binary string such that

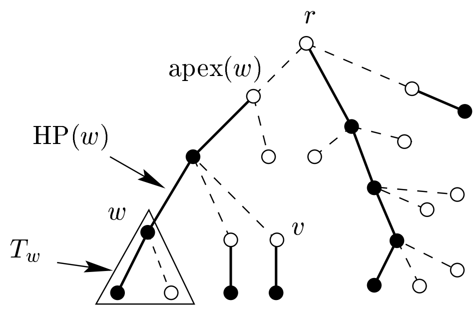

Figure 1: Heavy and light nodes are black and white points respectively. Heavy paths are subtrees composed of heavy edges only (solid edge). In this example, $\text{apex}(v) = v$.

- $\text{llabel}(w) \notin \{\text{llabel}(z) \mid z \neq w,\, z \in \text{children}(\text{parent}(w))\}$.

Let w be any node. Then $\text{hlabel}(w)$ is a binary string such that

- $\text{hlabel}(w) <_{\text{lex}} \min_{\text{lex}} \{\text{hlabel}(z) \mid z \neq w,\, z \in T_w \cap \text{HP}(w)\}$, where lex is the lexicographic order of two strings.

Heavy and light labels can be constructed in linear time as described in Section 4.1. We remark that lexicographic codes are not required for light labels, since we just impose that labels of different light children of a node are different.

Next we assign a label $l(v)$ to each node $v \in T$ topdown as follows. We define $l(\text{parent}(r))$, and $\text{llabel}(r)$, to be the empty string. Then for every node v, $l(v) = l(\text{parent}(\text{apex}(v))) \cdot \text{llabel}(\text{apex}(v)) \cdot \text{hlabel}(v)$.

A label $l(v)$ consists of the concatenation of alternating heavy and light labels, thus $l(v) = h_1 \cdot l_1 \cdot h_2 \cdot l_2 \cdots$. For a bit string s, we let $s[i]$ be the bit i in s. Beside $l(v)$ we need a label $k(v)$ of the same length, where $k(v)[i] = 1$ if and only if $l(v)[i]$ is the beginning of either a light or heavy label. This is doable because $|\text{llabel}(v)|, |\text{hlabel}(v)| \geq 1$. The label, $\text{label}(v)$, assigned to a node consists of the concatenation of $l(v)$ and $k(v)$. Once heavy and light labels of all the nodes have been computed, one can perform a depth first traversal [60] of the tree to compute in linear time $l(v)$ and $k(v)$.

Before showing how to compute the NCA given two labels we bound the length of these labels. Using the alphabetic code from the preliminary section, we have

- $|\text{llabel}(w)| \leq \log \text{lsize}(\text{parent}(w)) - \log \text{size}(w) + O(1)$, and
- $|\text{hlabel}(w)| \leq \log \sum_{v \in \text{HP}(w)} \text{lsize}(v) - \log \text{lsize}(w) + O(1)$

Lemma 4 *For any node $v \in T$, $|\text{label}(v)| = O(\log n)$.*

Proof: First, note that $|\text{label}(w)| = |l(w)| + |k(w)| = 2|l(w)|$. The number of heavy labels in $l(w)$ is at most $\log n$. Hence the additive $O(1)$ terms in the light and heavy labels of $l(w)$ sum to $O(\log n)$. So, in the remaining analysis we skip the additive constant term. We prove by induction on the depth of the node that $|l(w)| \leq \log n - \log \text{lsize}(w)$. For a node w, where $\text{apex}(w) = r$, in a tree with root r, $|l(w)| = |\text{hlabel}(w)| \leq \log n - \log \text{lsize}(w)$ so the statement holds. For another node w, where $\text{apex}(w) \neq r$, $l(w) = l(\text{parent}(\text{apex}(w))) \cdot \text{llabel}(\text{apex}(w)) \cdot \text{hlabel}(w)$. Now, by induction, $|l(\text{parent}(\text{apex}(w)))| \leq \log n - \log \text{lsize}(\text{parent}(\text{apex}(w)))$, and also by the definitions of the heavy labels and the light labels we have $|\text{llabel}(\text{apex}(w))| \leq \log \text{lsize}(\text{parent}(\text{apex}(w))) - \log \text{size}(\text{apex}(w))$, and $|\text{hlabel}(w)| \leq \log \sum_{v \in \text{HP}(w)} \text{lsize}(v) - \log \text{lsize}(w)$. By summing up the last three bounds and using the fact that $\sum_{v \in \text{HP}(w)} \text{lsize}(v) \leq \text{size}(\text{apex}(w))$ we obtain the lemma. □

Lemma 5 *Let x and y be two vertices of T,*

1 *If $l(x) = h_1 \cdot l_1 \cdots h_i \cdot l_i \cdot t$ and $l(y) = h_1 \cdot l_1 \cdots h_i \cdot l'_i \cdots$, where $l_i \neq l'_i$ or $l_i \cdot t$ is empty, then $l(\text{nca}(x,y)) = h_1 \cdot l_1 \cdots h_{i-1} \cdot l_{i-1} \cdot h_i$.*

2 *If $l(x) = h_1 \cdot l_1 \cdots h_i \cdots$ and $l(y) = h_1 \cdot l_1 \cdots h'_i \cdots$, where $h'_i \neq h_i$, then $l(\text{nca}(x,y)) = h_1 \cdot l_1 \cdots h_{i-1} \cdot l_{i-1} \cdot \min_{\text{lex}} \{h_i, h'_i\}$.*

Proof: Let $z = \text{nca}(x,y)$. By definition $l(\text{parent}(\text{apex}(z)))$ is a prefix of both $l(x)$ and $l(y)$. Let w be the heavy child of z. If $x \in T_a$, $y \in T_b$, $a, b \in \text{children}(z) \setminus \{w\}$, we have case 1; otherwise case 2. □

To implement the calculation of $\text{nca}(x,y)$ described in Lemma 5 we need to be able to identify the maximum i such that both $l(x)$ and $l(y)$ have $h_1 \cdot l_1 \cdots h_i \cdot l_i$ as a prefix. To that end we calculate the length j_1 of the maximum common prefix of $l(x)$ and $l(y)$, and the length j_2 of the maximum common prefix of $k(x)$ and $k(y)$. Then we take $j = min\{j_1, j_2\}$ and calculate whether this bit (jth from the left) is part of a light label or a heavy label. To easily determine whether bit j occurs within a light label or a heavy label we can for example add to the label of v a mask containing a 1 in every bit that belong to a light label or alternatively use a table that counts the parity of the number of bits set to one in a word. Once we identify whether j occurs within a light or heavy label we can extract the label of $\text{nca}(x,y)$ using straightforward bit manipulations.

Remarks: 1) In case we want to solve the unrestricted DRS problem using a cartesian tree, we can let all the light labels be an empty string, since a cartesian tree is binary.
2) If we use an alphabetic code for the heavy labels (no heavy label on a path is a prefix of another) and any prefix code for the light labels then $l(v)$ uniquely identify v, and there is no need to look at $k(v)$ in order to figure the maximum i such that $h_1 \cdot l_1 \cdots h_i \cdot l_i$ is a prefix of both $l(x)$ and $l(y)$.

References

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In 12^{th} *Annual ACM Symposium on Discrete Algorithms (SODA)*, pages 547–556, 2001.

[2] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal on Computing*, 10(3):405–421, 1981.

[3] 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.

[4] S. Alstrup, A. Ben-Amram, and T. Rauhe. Worst-case and amortised optimality in union-find. In 31^{th} *Annual ACM Symposium on Theory Computing (STOC)*, pages 499–506, 1999.

[5] S. Alstrup and J. Holm. Improved algorithms for finding level ancestors in dynamic trees. In 27^{th} *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of LNCS, pages 73–84. Springer-Verlag, 2000.

[6] S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In 13^{th} *Annual ACM Symposium on Discrete Algorithms (SODA)*, 2002.

[7] S. Alstrup and M. Thorup. Optimal pointer algorithms for finding nearest common ancestors in dynamic trees. *Journal of Algorithms*, 35(2):169–188, 2000.

[8] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In 21^{st} *Annual ACM Symposium on Theory of Computing (STOC)*, pages 309–319, 1989.

[9] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[10] O. Berkman and U. Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, April 1994.

[11] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for LCA's, MST verification, and dominators. In 30^{th} *Annual ACM Symposium on Theory of Computing (STOC)*, 1998.

[12] S. Carlsson and B. J. Nilsson. Computing vision points in polygons. *Algorithmica*, 24(1):50–75, 1999.

[13] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewdith. Part II: Optimal parallel algorithms. *Theoretical Computer Science*, 203(2):205–223, August 1998.

[14] Y-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms (extended abstract). In 6^{th} *Annual ACM Symposium on Discrete Algorithms (SODA)*, pages 139–149, January 1995.

[15] C. Chien and I. D. Scherson. Self-routing least common ancestor networks. In *Proc. Frontiers '92: 4^{th} Symposium on Massively Parallel Computation*, pages 513–514. IEEE, 1992.

[16] R. Cole and R. Hariharan. Dynamic lca queries on trees. In 10^{th} *Annual ACM Symposium on Discrete Algorithms (SODA)*, 1999.

[17] L. J. Cowen. Compact routing with minimum stretch. *Journal of Algorithms*, 38:170–183, 2001.

[18] P. F. Dietz. Finding level-ancestors in dynamic trees. In *2nd Work. on Algo. and Data Struc.*, volume 1097 of *LNCS*, pages 32–40, 1991.

[19] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[20] T. Eilam, C. Gavoille, and D. Peleg. Compact routing schemes with low stretch factor. In 17^{th} *Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 11–20, August 1998.

[21] D. Eppstein. Finding common ancestors and disjoint paths in DAGs. Technical Report ICS-TR-95-52, University of California, Irvine, Department of Information and Computer Science, 1995.

[22] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998. Including the paper [21].

[23] M. Farach-Colton. Optimal suffix tree construction with large alphabets. In 38^{th} *Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.

[24] M. Farach-Colton, S. Kannan, and T. Warnow. A robust model for finding optimal evolutionary trees. *Algorithmica*, 13(1/2):155–179, 1995.

[25] P. Fraigniaud and C. Gavoille. Routing in trees. In 28^{th} *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of LNCS, pages 757–772, 2001.

[26] G. N. Frederickson and R. Janardan. Separator-based strategies for efficient message routing. In 27^{th} *Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 428–437, October 1986.

[27] G. N. Frederickson and R. Janardan. Space-efficient message routing in c-decomposable networks. *SIAM Journal on Computing*, 19(1):164–181, February 1990.

[28] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[29] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In 21^{st} *Annual ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.

[30] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In 16^{th} *Annual ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

[31] H.N. Gabow. Data structure for weighted matching and nearest common ancestors with linking. In 1^{st} *Annual ACM Symposium on Discrete Algorithms (SODA)*, pages 434–443, 1990.

[32] E. N. Gilbert and E. F. Moore. Variable–length binary encodings. Technical report, Bell System Technical Journal, 1959.

[33] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[34] D. Harel. A linear time algorithm for the lowest common ancestors problem (extended abstract). In 21^{st} *Annual IEEE Symposium on Foundations of Computer Science(FOCS)*, pages 308–319, 1980.

[35] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal Computing*, 13(2):338–355, 1984.

[36] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematic*, 5(4):596–603, 1992. Preliminary version appeared in STOC 1988.

[37] H. Kaplan and T. Milo. Short and simple labels for small distances and other functions. In 7^{th} *International Workshop on Algorithms and Data Structures (WADS)*, volume 2125 of LNCS, pages 32–40. Springer, August 2001.

[38] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[39] V. King. private communication, July 1999.

[40] Janos Komlos. Linear verification for spanning trees. *Combinatorica*, pages 57–65, 1985.

[41] R. Lin. Fast algorithms for lowest common ancestors on a processor array with reconfigurable buses. *Information Processing Letters*, 40(4):223–230, 1991.

[42] R. Lin and S. Olariu. A simple optimal parallel algorithm to solve the lowest common ancestor problem. In *ICCI: Advances in Computing and Information–ICCI: International Conference on Computing and Information*. LNCS, Springer-Verlag, 1991.

[43] P. Looges. Optimal solution to the least common ancestor problem on the linear array with reconfigurable global buses. Technical Report TR-92-32, Old Dominion University, 1992.

[44] M. Farach-Colton M. A. Bender. The LCA problem revisted. In 4^{th} *LATIN*, pages 88–94, 2000.

[45] D. Maier. A space efficient method for the lowest common ancestor problem and an application to finding negative cycles. In 18^{th} *Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 132–141, 1977.

[46] K. Mehlhorn. A best possible bound for the weighted path lenght of binary search trees. *SIAM Journal on Computing*, 6(2):235–239, June 1977.

[47] M. Nykänen and E. Ukkonen. Finding lowest common ancestors in arbitrarily directed trees. *Information Processing Letters*, 50(6):307–310, 1994.

[48] R. Pagh. Faster deterministic dictionaries. In 11^{th} *Annual ACM Symposium on Discrete Algorithms (SODA)*, pages 487–49, 2000.

[49] D. Peleg. Proximity-preserving labeling schemes and their applications. In 25^{th} *International Worshop, Graph-Theoretic Concepts in Computer Science (WG)*, volume 1665 of LNCS, pages 30–41, 1999.

[50] D. Peleg. Informative labeling schemes for graphs. In 25^{th} *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 1893 of LNCS, pages 579–588. Springer, August 2000.

[51] P. Powell. A further improved LCA algorithm. Technical Report TR90-01, University of Minneapolis, 1990.

[52] P. Radge. The parallel simplicity of compaction and chaining. In 17^{th} *International Colloquium on Automata, Languages and Programming (ICALP)*, 1990.

[53] N. Santoro and R. Khatib. Labeling and implicit routing in networds. *The Computer Journal*, 28:5–8, 1985.

[54] E. Schenk. Parallel dynamic lowest common ancestors. *Nordic Journal of Computing*, 1(4):402–432, Winter 1994. Selected papers of the 4^{th} Scandinavian Workshop on Algorithm Theory (SWAT '94) (Århus, 1994).

[55] I. D. Scherson and C.-K. Chien. Least common ancestor networks. In V. K. Prasanna, editor, 7^{th} *International Parallel Processing Symposium (IPPS)*, pages 507–513, Newport Beach, CA, April 1993. IEEE Computer Society Press.

[56] I. D. Scherson and P. Y. Wang. On the parallel processing capabilities of LCA networks. In *Proc. Frontiers '92: 4^{th} Symposium on Massively Parallel Computation*, pages 571–572, 1992.

[57] B. Schieber. Parallel lowest common ancestor computation. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 259–273. Morgan Kaufmann Publishers, 1993. Chap. 6.

[58] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal of Computing*, 17:1253–1262, 1988.

[59] 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.

[60] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[61] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[62] M. Thorup and U. Zwick. Compact routing schemes. In 13^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001.

[63] 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.

[64] Tsin. Finding lowest common ancestors in parallel. *IEEE Transactions on Computers*, 35, 1986.

[65] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[66] Z. Wen. New algorithms for the LCA problem and the binary tree reconstruction problem. *Information Processing Letters*, 51(1):11–16, July 1994.

[67] J. Westbrook. Fast incremental planarity testing. In Werner Kuich, editor, 19^{th} *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 623 of LNCS, pages 342–353. Springer-Verlag, 1992.