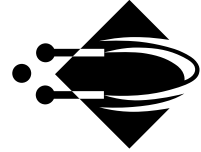

The IT University of Copenhagen

Improved labeling scheme for ancestor queries

Stephen Alstrup
Theis Rauhe

IT-C Technical Report Series **2001-5**

ISSN *1600–6100* **August 2001**

Copyright © 2000, Stephen Alstrup
Theis Rauhe

The IT University of Copenhagen
All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

ISSN *1600–6100*

ISBN *87–7949–009–3*

Copies may be obtained by contacting:

The IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 38 16 88 88
Telefax: +45 38 16 88 99
Web www.it-c.dk

Improved labeling scheme for ancestor queries

Stephen Alstrup* Theis Rauhe†

Abstract

We present a labeling scheme for rooted trees that supports ancestor queries. Given a tree, the scheme assigns to each node a label which is a binary string. Given the labels of any two nodes u and v, it can in constant time be determined whether u is ancestor to v alone from these labels. For trees of size n our scheme assigns labels of size bounded by $\log n + O(\sqrt{\log n})$ bits to each node. This improves a recent result of Abiteboul, Kaplan and Milo at SODA'01, where a labeling scheme with labels of size $3/2 \log n + O(\log \log n)$ was presented. The problem is among other things motivated in connection with efficient representation of information for XML-based search engines for the internet.

*IT University in Copenhagen, Glentevej 67, DK-2400 Copenhagen NV, Denmark. Email: {stephen, theis}@it-c.dk

1 Introduction

A labeling scheme for ancestor queries for a rooted tree assigns labels to each node v of a given tree T, where each label $l(v) \in \{0,1\}^*$ is a binary string. For any two nodes u, v, the labels $l(u), l(v)$ suffices to determine whether u is ancestor to v. We present a labeling scheme for a tree of size n whose labels have size bounded by $\log n + O(\sqrt{\log n})$[1], where n is the size of T. The labels for T can be computed in linear time, and a query can be computed in constant time on a standard random access machine with word size $\Omega(\log n)$. We use a standard instruction set such as bitwise boolean operations, shifts, addition etc.

Motivation: There are several nice properties of a labeling scheme for ancestor queries. Since query computation is based solely on local label information such schemes can avoid costly access to large tables that stores global information for the tree. For large global tables this may avoid expensive memory lookups. Furthermore, the local distributed information is also required in some routing schemes. In [11] Zwick and Thorup show how to assign short labels to a packet which should be routed, minimizing the overhead information associated to the packet such that the ancestor relation can be determined when the packet arrives to a node in the network. Abiteboul, Kaplan and Milo [1] also study the problem and presents as the main application its use for efficient representation of information for XML search engines. The XML Web-standard [13] allows for the development of advanced search engines that support sophisticated queries. Web documents obeying the XML standard can be viewed as trees, and some of the advanced queries correspond to testing the ancestor relationship in these trees. For such queries it is crucial to make efficient computation by minimizing access to files/global memory. In [7] Kannan, Naor, and Rudich examine labeling schemes for various families of graphs including trees. The main motivation for their study is the construction of small vertex-induced universal graphs. For trees they consider the ancestor and parent/adjacency relations. They show how to obtain labels of size $2 \log n$ for the ancestor relation. We refer the reader to [11, 1, 7] for further details of appilcations of labeling schemes supporting ancestor queries.

Related results: Tsakalidis [12] shows that if we assign the preorder and postorder number to each node in a tree, a node v is an ancestor to a node w iff $\mathit{preorder}(v) \leq \mathit{preorder}(w) \leq \mathit{postorder}(v)$. This leads to a simple labeling scheme with labels of size $2 \log n$ for the ancestor relation. Similarly, simple $2 \log n$ labeling schemes are presented in [7] and [9]. In [1] a labeling scheme with labels of size $3/2 \log n + O(\log \log n)$ is presented using more involved recursive tree decomposition techniques. This work left as an open problem to improve this bound. The best lower bound is the trivial $\lceil \log n \rceil$, *i.e.,* needed in order to distinguish all nodes. Our result, $\log n + O(\sqrt{\log n})$ bits per label have been cited in [11] and [8] as personal communication [4]. In [11] efficiently labeling schemes is used to improve routing schemes, and it is shown how to construct small labels to test for ancestor and parent relationship. Specifically, [11] obtain labeling scheme for ancestor queries with labels of size $\log n + O(\log n / \log \log n)$, improving [1], using techniques different from ours. In [8] the authors examine besides other things, how to support queries for parent and sibling relations between two nodes u, v, and they also show how to test if u is ancestor to v with depth d less than u, for constant d, using labels of size $\log n + O(\sqrt{\log n})$. But it is not shown how to support ancestor queries as studied in this paper. In [8] they write

> However, we can combine the technique presented in this paper with a recent labeling scheme for ancestor queries suggested by Abiteboul et. al. [1] and subsequently improved by Alstrup and Rauhe [4]. The resulting labeling scheme will allow to identify the d

[1] log is the logarithm with base 2 throughout the paper

closest ancestors of any node, and, additionally, to answer ancestors queries between *every* pair of nodes. Although the combined scheme is more complicated that the one we present here, the maximum label length is still $\log n + \mathrm{O}(\sqrt{\log n})$.

The reference to [4] is the work presented in this paper. As stated in the above quotation, our labeling scheme can be combined with other techniques to test not only for the ancestor relation, but also the parent relation. However, in order to present the labeling scheme as simple as possible, we only present the labeling scheme for the ancestor relation.

The techniques used in our labeling schme is very similar to the ones used by Abiteboul, Milo and Kaplan [1]. The main new ingredient is the use of optimal alphabetic codes [6] along certain paths in the recursive tree composition instead of a counter.

2 Preliminaries

Alphabetic codes: Let $\langle y\rangle_k$ denote a sequence of elements from an ordered universe.

For binary strings $a, b \in \{0,1\}^*$, $a <_{\text{lex}} b$ iff a is less than b in the lexicographic order. An *alphabetic* sequence of length 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 $1 \le i < j \le k$. If the sequence is prefix-free (no string is a prefix of another), we call it an *alphabetic code.*

The following lemma states a well-known result for alphabetic codes.

Lemma 1 (Gilbert and Moore) *[6] An integer sequence $\langle y\rangle_k$ with $n = \sum_{i\le k} y(i)$ has an alphabetic code $\langle b\rangle_k$ where $|b(i)| \le \log n - \log y(k) + \mathrm{O}(1)$ for all i.*

For our purposes it suffices if $\langle b\rangle_k$ is alphabetic. In $\mathrm{O}(k)$ time can an alphabetic sequence of length k be constructed as follows. Let $k(i) = \sum_{j<i} y(j)$ and $Iv(i) = k(i) + 1 \cdots k(i) + y(i)$, and $f(i) = \lfloor \log y(i) \rfloor$. In the interval $Iv(i)$ there must be a number $z(i)$, where $z(i) \mod 2^{f(i)} = 0$. If $k(i)+1 \mod 2^{f(i)} = 0$, then $z(i) = k(i)+1$; otherwise $z(i) = k(i)+1-(k(i)+1 \mod 2^{f(i)})+2^{f(i)}$. 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 bitstring consisting of the $w - f(i)$ most significant bit from $z(i)$. Hence, alphabetic sequences of length k can be constructed in $\mathrm{O}(k)$ time if machine operations to compute $\lfloor \log \cdot \rfloor$ and $\mod$ are supported. If not, we can using $\mathrm{O}(n)$ prepossessing time and space construct a table representing these functions. This will only contribute with additional linear time for the preprocessing step in our algorithm.

Tree Terminology: We denote the set of nodes and edges in T as $V(T)$ and $E(T)$ respectively. We let $T(u)$ denote the subtree of T rooted at node $u \in V(T)$. If $w \in V(T(u))$ then u is an ancestor to w, and we write $u \prec w$. If $w \in V(T(u)) \setminus \{u\}$ then u is a proper ancestor to w. If u is a (proper) ancestor to w, then w is a (proper) descendant to u. For nodes u and v in T we denote the path between u and v including u and v by $u \rightsquigarrow v$.

We say that a rooted tree is a *binary tree* if all nodes in the tree have at most two children. If a tree T is not binary it is straightforward to construct a tree T', where $2|V(T)| \le |V(T')|$ and a mapping $f : V(T) \to V(T')$, such that for $v, w \in V(T)$, we have that $v \prec_T w$ iff $f(v) \prec_{T'} f(w)$. Hence, in the remaining of this paper we assume without loss of generality that T is a binary tree.

Depth first labeling: We need the following simple *DFS* labeling scheme. Let T be a tree of size n. To each internal node we assign a $first$ and $last$ value, and to a leaf we only assign a $first$ value as follows. Perform a depth first traversal [10] of the tree from the root r. Let $first(r) = counter = 0$. The first time a node v is visited we set $first(v) = counter$, and

increment the counter with 1. The last time an internal node v is visited we set $last(v) = counter$. Now an internal node v is a proper ancestor to a node w iff $first(v) < first(w) < last(v)$. The standard binary representation of the numbers $first$ and $last$ uses at most $\log n + 2$ bits. Hence, this scheme uses at most $\log n + 2$ bits for leafs and $2 \log n + 4$ bits for internal nodes. We will use one more property from this labeling which holds for binary trees: If v is a leaf with parent p then either $first(v) = first(p) + 1$ or $first(v) + 1 = last(p)$.

Clustering: Let T be a tree of size $n = |V(T)| > 1$. For a connected subgraph C of T, we call a node in $V(C)$ incident with a node in $V(T) \setminus V(C)$ a *boundary node*. The boundary nodes of C are denoted as ∂C. A *cluster* is a connected subgraph of T where $|\partial C| \leq 2$. The definitions of clusters comes from [3, 2, 5].

A set of clusters CS is a *cluster partition* of a tree T with root r iff $V(T) = \cup_{C \in CS} V(C)$, $E(T) = \cup_{C \in CS} E(C)$, and for any $C_1, C_2 \in CS$, $E(C_1) \cap E(C_2) = \emptyset$, $|E(C_1)| \geq 1$, $r \in \partial C$ if $r \in V(C)$, and if $v \in \partial C$ has two children a and b, then no clusters include both a and b.

From [3, 2, 5] it follows that:

Lemma 2 *Given a tree T, $n > 1$, and a parameter x, where $\lceil n/x \rceil \geq 2$, it is possible to construct a cluster partition CS in linear time, where $|CS| \leq k \cdot x$, for a constant k, and $|V(C)| \leq \lceil n/x \rceil$ for $C \in CS$.*

In [1] a similarly partitioning is defined.

For the sake of completeness we show how to make a cluster partition for a tree T with root r.

Choose r to be a boundary node and its two children a and b to be boundary nodes, constructing two clusters including the node $\{r, a\}$ and $\{r, b\}$. Next we proceed to pick boundary nodes in the two trees $T(a)$ and $T(b)$. Let S be the set of boundary nodes picked in $T(a)$, initially $S = \{a\}$. Now recursively, choose a node $v \in S$ and examine its two children z and y. If $|V(T(z))| + 1 \leq \lceil n/x \rceil$, let the nodes $V(T(z))$ and v be a cluster, with the boundary node v. Proceed similarly with the node b. Now, if c is a child to v where $|V(T(c))| > n/x$ then proceed as follows. Let $w \in V(T(c))$ be the node with minimum depth, where $|V(T(c))| + 2 - |V(T(w))| \leq \lceil n/x \rceil$. Now pick w and v as the boundary nodes in the cluster consisting of the nodes $V(T(c)) \setminus V(T(w)) \cup \{v, w\}$. Implementing this algorithm topdown spend linear time for a partitioning.

3 A $\log n + O(\sqrt{\log n})$ labeling scheme

3.1 Tree decomposition

Let T be a binary tree with root r. As explained above we can assume T is a binary tree. Let CS be a cluster partition of T as described in Lemma 2. We will fix the parameter x later.

Next we define a *macro* tree T^m for the clusters CS. For each cluster $C \in CS$, where $|\partial C| = 2$ and $v, w \in \partial C$, we have the edges $(v, s(v, w)), (s(v, w), w), (s(v, w), l(v, w))$ in T^m, introducing two new nodes $s(v, w)$ and $l(v, w)$. For each cluster $C \in CS$, where $|\partial C| = 1$ and $v \in \partial C$, we have an edge $(v, l(v, C))$ introducing a new node $l(v, C)$. Since r is a boundary node, $r \in V(T^m)$, and we can root T^m in r.

Proposition 1 *For the macro tree we have the following properties which follows directly from the definitions:*

- *T^m is a binary tree.*
- *$|V(T^m)| \leq x \cdot k \cdot 4$, where k is the constant given in Lemma 2.*

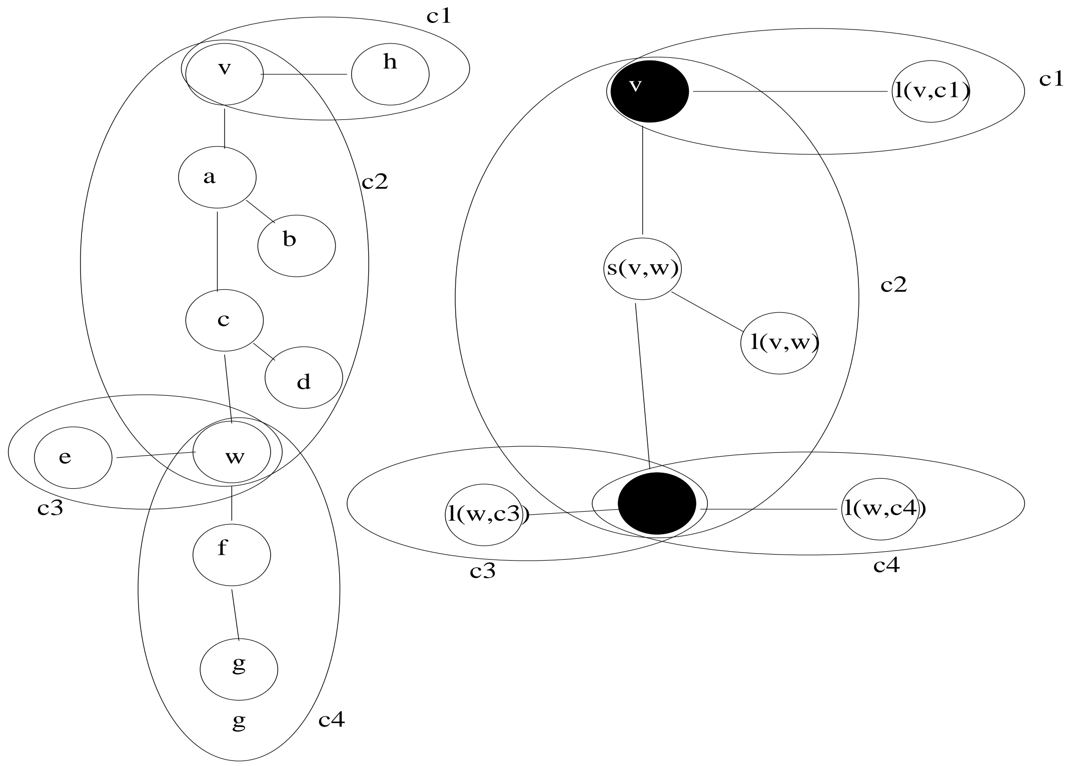

Figure 1: To the left a tree T decomposed to four clusters, where $V(c1) = \{v, h\}$, $\partial c1 = \{v\}$, $V(c2) = \{v, a, b, c, d, w\}$, $\partial c2 = \{v, w\}$, $V(c3) = \{w, e\}$, $\partial c3 = \{w\}$, $V(c4) = \{w, f, g\}$, and $\partial c4 = \{w\}$. To the right the tree T^m, illustrating from which clusters the nodes from T^m originates.

- *if z is a boundary node in a cluster, it is a boundary node in every cluster that contains it.*
- *if z not is a boundary node, it belongs to a unique cluster $C(z)$.*
- *The leafs in T^m are the nodes $l(\cdot, \cdot)$.*
- *For two boundary nodes $v, w \in T$, $v \prec_T w$ iff $v \prec_{T^m} w$.*
- *For $v, w \in \partial C$, either v or w is an ancestor in T to all nodes in $V(C)$.*

3.2 Node types

Given a cluster partition CS for a tree T, we associated to each node $v \in T$ a node $a(v) \in T^m$. The node $a(v)$ is only used to explain our approach and will not be a part of the label for v. Let $s(v, w) \in T^m$, then $\mathcal{P}(s(v, w)) = v \rightsquigarrow_T w \setminus \{v, w\}$ is denoted as the *spine path* associated to $s(v, w)$. In Figure 1 $\mathcal{P}(s(v, w)) = \{a, c\}$.

Definition 1 *In the following let $z \in V(C)$ and $C \in CS$. We have four disjoint cases:*

- *If z is a boundary node then $a(z) = z$.*
- *If $v, w \in \partial C$, $v \neq w$, and $z \in \mathcal{P}(s(v, w))$, then $a(z) = s(v, w)$, and we denote the node as a* spine node.

- *If $\partial C = \{v\}$ and $z \neq v$, then $a(z) = l(v, C)$, and we denote the node as a* leaf clustered node.
- *If $v, w \in \partial C$, $v \neq w$, and $z \notin v \rightsquigarrow_T w$, then $a(z) = l(v, w)$, and we denote the node as a* internal clustered node.

In Figure 1 v and w are boundary nodes, a and c are spine nodes, b and d are internal clustered nodes, and e, f, g, and h are leaf clustered nodes.

In the following we need some properties and definitions for spine nodes. Let z be a spine node:

Proposition 2

z has a child which either is a boundary node or a spine node.

If z has two children then one of its children is an internal clustered node v.

If z has a child v which is an internal clustered node and $w \in T(v)$, then w is an internal node and $w \in V(C(z))$.

Let z be a spine node with a child w, where w is a boundary node or a spine node. For a spine node z we define $Isubtree(z)$ to be $T(z)$ from which we have removed the subtree $T(w)$ and the edge (w, z). Hence $V(Isubtree(z))$ consists of z and the internal clustered nodes which are descendants to z. For $w \in V(Isubtree(z))$, we let $sub(v) = z$. The association of $sub(v)$ to a node v is, as for $a(v)$, only used to explain our approach. For the nodes SP in T on the same spine path, we let α be an ordering of the nodes, such that for $v, w \in SP$, $v <_\alpha w$, iff $depth_T(v) < depth_T(w)$.

3.3 Query computation depending on node type

To test the ancestor relation between nodes $u, v \in V(T)$ we distinguish between the cases below depending on the value of $a(u)$ and $a(v)$. First note that a necessary (but not sufficient) condition for $u \prec_T v$ is that $a(u) \prec_{T^m} a(v)$.

Proposition 3

(i) Case Spine nodes. $a(u) = a(v) = s(u', v')$. Here $v, u \in \mathcal{P}(s(u', v'))$, and $v \prec_T u$ iff $v \leq_\alpha u$.

(ii) Case internal clustered nodes. $a(u) = a(v) = l(u', v')$. Then $v \prec_T u$ iff $sub(v) = sub(u)$ and $v \prec_{T(sub(u))} u$.

(iii) Case leaf clustered nodes. $a(u) = a(v) = l(u', C)$. Here $u, v \in V(C)$, and $u \prec_T v$ if $u \prec_C v$.

(iv) Case spine and internal clustered nodes, where $a(u) = s(u', v')$ and $a(v) = l(u', v')$. Hence, $u, v \in C(V)$. Here $u, sub(v) \in \mathcal{P}(s(u', v'))$, and $u \prec_T v$ iff $u \leq_\alpha sub(v)$.

(v) In all other cases where $a(u) \neq a(v)$, $u \prec_T v$ iff $a(u) \prec_{T^m} a(v)$.

Hence, we have

Proposition 4 *If $v \prec_T w$ then*

- *v is either a spine node or a boundary node, otherwise*
- *v and w are both internal clustered nodes, and $sub(v) = sub(w)$, otherwise*

- *v and w are both leaf clustered nodes, and $v, u \in C(v)$.*

To each node in $v \in V(T)$, we use the first two bits of its labeling $L(v)$ to specify the type of node according to Definition 1, *i.e.,* specifying whether it is is a spine node, a boundary node, a leaf clustered node, or an internal clustered node. This will make us able to distinguish between the above cases.

3.4 Labeling of a node

Let $L^m : V(T^m) \to \{0,1\}^*$ be the *DFS* labeling of T^m. Hence each internal node v in T^m is associated a label consisting of at most $|L^m(v)| \leq 2\log x + O(1)$ bits. For a leaf l in T^m, $|L^m(l)| \leq \log x + O(1)$ bits. Let $\cdot$ denote the binary operator for concatenation of two strings.

After the prefix in a label that determinies the node type, the following bits depend on the type of the node, and the labeling are as follows.

v is a boundary node. In addition to the type bits, $L(v)$ is simply the label $L^m(a(v))$.

$$L(v) = \texttt{boundary} \cdot L^m(a(v)).$$

Thus $|L(v)| \leq 2\log x + O(1)$.

v is a spine node. Let $a(v) = s(v', w') \in T^m$ and $SP = \mathcal{P}(s(v', w'))$ be the nodes on the spine path that v belongs to. Let $k = |SP|$ be the number of nodes on this spine path. Let the nodes from SP be ordered $v_1, v_2 \cdots v_k$, such that for $i < j$, $depth_T(v_i) < depth_T(v_j)$. Let $T_i = Isubtree(v_i)$. To the spine path we let $\langle c \rangle_k$ denote the alphabetic sequence relative to sequence $\langle |V(T_1)|, \ldots, |V(T_k)| \rangle$ from Lemma 1. Hence, $c(j)$ is the alphabetic code for $V(T_j)$, and $|c(j)| \leq \log \sum_i |V(T_i)| - \log |V(T_j)| + O(1) \leq \log(n/x) - \log|V(T_j)|$, since we at most n/x nodes in a cluster. For the nodes v_i, v_j, $i < j$, we have that $v_i <_\alpha v_j$ iff $c(i) <_{\text{lex}} c(j)$. The labeling of v is now defined to be

$$L(v) = \texttt{spine} \cdot L^m(s(v', w')) \cdot c(j).$$

Thus $|L(v)| \leq 2\log x + \log(n/x) - \log|V(T_j)| + O(1) \leq \log x + \log n + O(1)$.

v is an internal clustered node. Let $a(v) = l(v', w')$ and $v_j = sub(v)$ be the spine node associated to v. As above, let $\langle c \rangle_k$ denote the alphabetic sequence for the spine path $\mathcal{P}(v', w')$. Hence, $c(j)$ is the alphabetic sequence for T_j. The root of T_j has one child c, let $T'_j = T_j(c)$. Let $L^b(v)$ denote the label of node v in T'_j with the *recursive* ancestor labeling again relative to x as decomposition threshold. The labeling of node v for tree T is

$$L(v) = \texttt{internal_clustered} \cdot L^m(l(v', w')) \cdot c(j) \cdot L^b(v).$$

Thus $|L(v)| \leq \log x + \log(n/x) - \log|T_j| + |L^b(v)| + O(1) = \log n - \log|T_j| + |L^b(v)| + O(1)$, where $|L^b(v)|$ is the length of the recursive ancestor labeling of v in T'_j.

v is an leaf clustered node. Let $a(v) = l(v', C)$.

$$L(v) = \texttt{leaf_clustered} \cdot L^m(l(v', C)) \cdot L^b(v).$$

Thus $|L(v)| \leq \log x + |L^b(v)| + O(1)$, where $|L^b(v)|$ is the length of the recursive ancestor labeling of v in T'_j, where $T'_j = T(c)$, c is the only child to a boundary node in $C(v)$.

We say that the prefix of $L(v)$ before the recursive labeling $L^b(v)$ is a *block* of the (recursive) labeling $L(v)$. Similarly $L^b(v)$ consists of such recursive blocks, *i.e.*, $L(v) = b_1 b_2 \dots b_k$, where $b_i \in \{0,1\}^*$ is a block of bits starting with the type bit information and ending just before the type bit information of block b_{i+1}. The block b_k is the last block. This block is either for a spine node or a cluster node, since there is non subsequent recursive label.

Till now we have not discussed the special case where a subtree have constant size. Let $L^b(v)$ be the recursive code for v in a tree of constant size. In this case we could let $L^b(v)$ consist of a single block, starting with a special stop bit in the beginning of the block, followed by any trivial constant size labeling that enables us to test the ancestor relationship among the constant number of nodes in the subtree.

Consider a node $v \in V(T)$ and its label $L(v)$ with blocks $b_1 \dots b_k$. We define subtrees of T relative to prefixes of this label. For the empty bitstring ϵ we let $T(\epsilon) = T$. The tree $T(b_1)$ is the subtree of T for which the recursive labeling $L^b(v) = b_2 \dots b_k$ is defined for, and generally we let $T(b_1 b_2 \dots b_i)$ denote the subtree for which the label $(b_{i+1} \dots b_k)$ is defined for.

In addition to the labeling defined above we also keep the index of the first bit position of the last block b_k, *i.e.*, $|b_1 \dots b_{k-1}|$, using $\log\log n$ bits. This index is stored at a fixed position at the end of the word that stores the label.

3.5 Length of labels

We analyse the worst-case length of a label by the above scheme. Let L define the labeling mapping for T, and let v denote a node in T. Besides the $\log\log n$ bits for the index of the last index, we use:

- v is a boundary node: $|L(v)| = 2\log(x) + \mathrm{O}(1)$.
- v is a spine node: $\log n + \log x + \mathrm{O}(1)$.
- v is an internal clustered node: $\log n - \log |V(T_j)| + |L^b(v)| + \mathrm{O}(1)$, where $L^b(v)$ is the label for v in a tree of size $|V(T_j)| - 1$.
- v is a leaf clustered node: $\log x + |L^b(v)| + \mathrm{O}(1)$, where $L^b(v)$ is the label for v in a tree of size $|V(T_j)| - 1$.

Define $s(z)$ to be the worst-case label length of a node in any binary tree of size $z \leq n$, for our labeling where using fixed decomposition threshold x, depending only on n.

Then for an internal clustered node v in T, we can bound its label length to $\log n - \log m + s(m) + \mathrm{O}(1)$, where $m = |V(T_j)| \leq n/x$.

For a leaf clustered node v in T, we can bound its label length to $\log x + s(m) + \mathrm{O}(1)$, where $m = |V(T_j)| \leq n/x$.

Hence in general, for decomposition threshold $x \leq \sqrt{n}$ we can bound the longest label length $s(n)$ by

$$\begin{aligned} s(n) &\leq \max\big(\max_{m \leq n/x} (\log n - \log m + \mathrm{O}(1)) + s(m), \\ &\qquad \log x + s(m), 2\log x + \mathrm{O}(1), \log n + \log x + \mathrm{O}(1)\big) \\ &\leq \max(\log n + \mathrm{O}(\log x), \log n + \mathrm{O}(\log n / \log x)). \end{aligned}$$

Hence choosing $x = 2^{\sqrt{\log n}}$ we obtain the claimed bound of $\log n + \mathrm{O}(\sqrt{\log n})$ for the worst-case length of a label in T.

3.6 Computing a query

Let v, w be two nodes in the tree T with labels $L(v) = a_1, a_2, \cdots$ and $L(w) = b_1, b_2, \cdots$, where a_i, b_i are blocks as defined above. Let j be index of the last block of $L(v)$ (which is stored separately for the label of v as described above). Assume $v \prec w$. From Property 4 we have that v is either a cluster node or a spine node in $T(a_1, a_2, \cdots, a_{j-1})$. Furthermore, v and w must both be either leaf clustered nodes or internal clustered nodes in $T_i = T(a_1, a_2, \cdots a_i)$ for $i < j-1$, and included in the same subtree in T_i. Hence, in order to test if $v \prec w$, we first test if $a_1, a_2, \cdots a_{j-1} = b_1, b_2, \cdots b_{j-1}$. If this is not the case we can return false. Otherwise assume $a_l = b_l$ for $l < j$ and let $T' = T(a_1, a_2, \cdots a_{j-1}) = T(b_1, b_2, \cdots, b_{j-1})$.

Since j is the first index of the last block of labeling $L(v)$, the node v in T' is either a spine node or a boundary node, while w may be all four type of nodes. Let $dfs(v)$ and $dfs(w)$ be the DFS labeling in block a_j and b_j respectively. If $a(v) = a(w)$ in T' then $dfs(v) = dfs(w)$. Now, we can use the case analyse described in Properties 3.3 to see if $v \prec w$ or not.

Case (i), both nodes are spine nodes on the same spine path. Let $c(v)$ and $c(w)$ be the code from alphabetic sequence for block a_j and b_j respectively. Then $v \prec w$ if $v \leq_\alpha w$, which is true if $c(v) <_{\text{lex}} c(w)$.

Case (iv), v is an spin node and w is an internal clustered node. As described for the DFS labeling we can test using the DFS label whether $a(v)$ is a child to $a(w)$. If this is the case we check the codes for the alphabetic sequence as above.

In the remaining cases $v \prec w$ iff $a(v) \prec a(w)$ which we can check from the DFS labeling.

To find out where the code for the alphabetic sequence start in a labeling, we for boundary nodes and spine nodes encode in front of the DFS labeling the length l of the DFS labeling. This can easily be encoded using additional $\mathrm{O}(\log l)$ bits for spin and boundary nodes. Since, a node only is a spine or a boundary in a single block it will not increase the labeling size. By letting the DFS-labeling have a fixed size for a macro tree, we also know the length of the DFS-labeling of the node to which we compare with spine or boundary nodes.

Note that the above tests for the ancestor relation using the DFS labeling and codes for alphabetic sequences only uses a constant number of basic and fast RAM operations, *i.e.,* bit-wise AND/OR, left/right shifts and less-than comparisons. Our labeling scheme avoids the sometime more costly operations such as multiplication, retrieval of most significant bit or non-standard operations pre-computed and stored in a pre-computed table.

Hence, we have:

Theorem 1 *There exists a labeling scheme for ancestor queries of a tree of size n with maximal fixed length labels of size* $\log n + \mathrm{O}(\sqrt{\log n})$.

Finally, the prepossessing time is linear using the linear time algorithms for alphabetic sequences, clustering etc.

References

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proceedings of the twelfth annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 547–556, 2001.

[2] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 270–280. Springer-Verlag, 7–11 July 1997.

[3] S. Alstrup, J. Holm, and M. Thorup. Maintaining median and center in dynamic trees. In *7th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 1851 of *Lecture Notes in Computer Science*, pages 46–56. Springer-Verlag, 2000.

[4] S. Alstrup and T. Rauhe, January 2001. Personal communication at SODA'01.

[5] G. Frederickson. Ambivalent data structures for dynamic 2–edge–connectivity and k smallest spanning tree. *SIAM J. Computing*, 26(2):484–538, 1997. See also FOCS'91.

[6] E. N. Gilbert and E. F. Moore. Variable–length binary encodings. Technical report, Bell System Technical Journal, 1959. Vol. 38, pp. 933–967.

[7] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. *SIAM J. DISC. MATH.*, 1992. Preliminary version appeared in STOC'88.

[8] H. Kaplan and T. Milo. Short and simple labels for small distances and other functions. In *7nd Work. on Algo. and Data Struc.*, LNCS, 2001.

[9] N. Santoro and R. Khatib. Labeling and implicit routing in networks. *The computer J.*, 28:5–8, 1985.

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

[11] M. Thorup and U. Zwick. Compact routing schemes. In *ACM Symposium on Parallel Algorithms and Architectures*, volume 13, 2001.

[12] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, 1984.

[13] Extensible markup languange (xml) 1.0. http://www.w3.org/TR/REC-xml.