Compact Labeling Scheme for Ancestor Queries*

Serge Abiteboul † Stephen Alstrup ‡ Haim Kaplan §¶ Tova Milo §
Theis Rauhe ‡

September 4, 2005

Abstract

We consider the following problem. Given a rooted tree T, label the nodes of T in the most compact way such that given the labels of two nodes u and v one can determine in constant time, by looking only at the labels, if u is ancestor of v. The best known labeling scheme is rather straightforward and uses labels of length at most $2\log_2 n$ bits each, where n is the number of nodes in the tree. Our main result in this paper is a labeling scheme with maximum label length $\log_2 n + \mathrm{O}(\sqrt{\log n})$. Our motivation for studying this problem is enhancing the performance of web search engines. In the context of this application each indexed document is a tree and the labels of all trees are maintained in main memory. Therefore even small improvements in the maximum label length are important.

1 Introduction

The problem we study in this paper is the following. Given a rooted tree T, what are the shortest labels that one can attach to the nodes of the tree such that, given the labels of two nodes v and w, it is possible to determine in constant time – by looking only at the labels – if v is an ancestor of w. When measuring the quality of a labeling scheme, we will look at the length of the maximal label, as a function of the size of the tree. We are interested in the **exact** function rather than its order of growth - constant factors do matter! (e.g. labels of length bounded by, say $1.5\log n$ bits, will be considered significantly better than labels

*This paper includes the results from two conference papers [2, 7] presented at ACM-SIAM Symposium on Discrete Algorithms 2001 and 2002

†INRIA-Rocquencourt, 78153 Le Chesnay CEDEX, France. Email: `Serge.Abiteboul@inria.fr`

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

§School of computer Science, Tel Aviv University, Tel Aviv 69978. Email: `{haimk,milo}@cs.tau.ac.il`.

¶Work supported in part by Israel Science Foundation (ISF) grant no. 548/00.

of length $2\log n$ bits[1]). Note that clearly one cannot go below $\log n$ bits - if the maximal label has less than $\log n$ bits, there are not enough labels in the domain even to just identify all nodes. But how much more is actually needed for answering ancestor queries? This somewhat abstract problem arises in optimizing web search engines.

Before presenting our results, let us consider a simple labeling scheme which we call the *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 [28] 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 by 1. The last time an internal node v is visited we set $last(v) = counter$, and increment the counter by 1. Now an internal node v is a proper ancestor of a node w if and only if $first(v) < first(w) < last(v)$. The standard binary representation of the numbers $first$ and $last$ consist of $\log n + 2$ bits. Hence, this scheme uses at most $\log n + 2$ bits for a label of a leaf and at most $2\log n + 4$ bits for a label of an internal node. Interestingly, although rather naive, the bound obtained by this simple scheme is in fact the tightest we could find in the existing literature. We present a labeling scheme that improves this bound to $\log n + \mathrm{O}(\sqrt{\log n})$ bits.

We use the RAM model of computation, and assume that the length of a computer word is $\Omega(\log n)$ bits (hence the basic operations on the labels can be performed in constant time). The algorithm only use the basic and fast RAM operations as assignment, less-than comparison, bitwise AND, OR, and XOR. Our labeling scheme avoids the sometime more costly operations such as multiplication.

Motivation: Most of the data on the web today is written in HTML format [33]. An HTML document consists of text interspersed with tag fields such as `<I> ... </I>` to describe the presentation of the page, the inclusion of pictures, hyperlinks, forms etc. These tags do not provide however any information about the semantic nature of the document components, which makes the analysis of the content of documents difficult. To remedy this, a new web standard, the XML data exchange format [31], has been proposed. Just like HTML, XML documents feature tags. But rather than providing instructions on how the document is to be displayed[2] they provide information on the logical structure of the document, with each semantic element starting with an <itemname> tag and ending with an </itemname> tag. To see an example, consider Figure 1 which shows a fragment of an XML document describing books catalog. The `<Category>` and `</Category>` tags are used to delimit the information corresponding to one catalog category. Each category item consists of a sequence of tagged sub-items such as `Category_Name`, `Book`, etc.

XML is being widely adopted as web standard and it is believed that although a large portion of the web will remain unstructured or hidden behind interactive graphics and forms, a large and content-wise essential portion of the web will soon be available in XML [25, 31].

[1]All logarithms in this paper are base 2.
[2]This may be specified in a *style-sheet*.

```
<Catalog>
  <Category>
    <Category_Name> Literature & Fiction </Category_Name>
    <Book>
      <ISBN>12445</ISBN> <Title>Bridget Jones's Diary</Title>
      <Authors><Author> Helen Fielding </Author></Authors>
      <Price> $10.39</Price> ...
    </Book>
    <Book> ... </Book>
    ...
  </Category>
  <Category> ... </Category>
  ...
</Catalog>
```

Figure 1: The Books Catalog

To retrieve data from the web, people use search engines like Alta vista [8] or Google [11]. Today these search engines support full-text queries, namely the user gives a few words and the engine returns documents containing these particular words. When querying XML data it is desired to support also queries that utilize the document structure to ask for more specific data. For example it is desired to support structural queries like "find all book items containing "Fielding" as an author and a price less than 12$" [1, 13, 32].

Going back to the problem stated at the introduction, observe that an XML document can be viewed as a tree (essentially the parse tree of the document) whose nodes are the document items and whose edges correspond to the component-of relationship among data items. With this view, a query of the above form amounts to finding nodes with particular tags (book, author, price) having certain ancestor relationship between them. In our example we ask for documents containing book nodes that are ancestors of particular qualifying author and price nodes.

To understand how to implement such structural queries consider the following rather typical implementation of a search engine. The engine's heart is a big hash table containing all words occurring in the database, where for each word we keep the identifiers of the documents in which it appears. For XML documents we refine this structure as follows. First in addition to the words that appear in text items, we also keep the tags of the nodes (like "book", "author" etc.) in the hash table. Second, to allow structural queries we give each node in the XML document (tree) a unique label, and associate with each tag in the hash table the labels of all nodes with this tag in each particular document which contains it.

Now if we give the nodes meaningful labels that reflect the hierarchical structure of the documents (namely, given labels of two items we can determine if one is a descendant, i.e.

component, of the other), queries of the above form can be answered by using the index only, without access to the actual document. We first use the hash table to retrieve the labels of the relevant nodes, and then iterate over the retrieved label sets and select those having the appropriate ancestor relationships.

To allow for good performance it is essential that the index structure (or at least a large part of it) reside in main memory. Observe that we are talking here about an extremely large structure. Experiments to evaluate the effect of adding the structural information to the index show that adding a label of just one integer (4 bytes) to each word occurrence almost doubles the index size [34]. Since the length of the node labels is a main factor of the index size, reducing this, even by a constant factor, is a critical issue, contributing directly to hardware cost reduction and performance improvement.

Consider now the actual physical representation of the labels in the index. In a fixed-length representation each label is allocated a fixed space, determined by the maximal potential label length. In a variable-length representation labels are allocated a variable size space (consisting of a fixed prefix stating the actual size of the label, followed by the label itself) and the size of the index is determined by the average label length plus the log of the maximum potential length of a label (the space allocated for the fixed prefix). We study here the case of fixed-length representation, aiming to improve the bound on the maximal label length. The case of variable-length representation is left for future research.

Related Work. Variants of the *DFS* labeling scheme have been described in several papers including [30, 12, 18, 27, 23]. Implementations of such approaches are integrated in the Xyleme XML warehouse system [34]. As explained by Kannan *et al.* [18], the names of the nodes in a graph[3] typically convey no information about the graph itself, and so memory is wasted. Therefore [18] focus on labeling the nodes in such a way that various information can be, as in this paper, detected solely by looking at names of the nodes. Kannan *et al.* [18] considered determining ancestor relationship and to that end they suggested a variant of the *DFS* labeling scheme. They also considered the parent relation and other information.

Extracting local information from the node is helpful for some routing applications. In [29] Zwick and Thorup show how to assign short labels to a packet which should be routed, that allows to determine the ancestor relationship between the packet's destination and the currently visited node. Thus avoiding costly access to external memory. Specifically, after the work presented in [7], Zwick and Thorup [29], independently, obtained a labeling scheme for ancestor queries with labels of length $\log n + \mathrm{O}(\log n/\log\log n)$. Recently, Alstrup, Bille, and Rauhe established that labels which carry ancestor information for every tree must have length $\log n + \Omega(\log\log n)$ [3] for some trees. An experimental comparison of different labeling schemes for testing ancestor relationship on real XML data can be found in [20].

Labeling schemes for other types of queries have recently received significant interest.

[3]Typically the integers 1 to n.

Labeling schemes for various distance queries have been studies in [16, 22, 26, 19]. Alstrup *et al.* [4] study a labeling scheme for computing nearest common ancestors in trees. Labels for Vertex-connectivity and the adjacency relation are described in [19, 9, 10, 18, 21, 27]. In [15] one finds an extensive survey of labeling schemes and their applications.

The rest of this paper is organized as follows. Section 2 gives some definitions and background on labellings of trees, alphabetic codes, and partitions of trees. We present our labeling scheme in Section 3. In section 4 we describe the algorithm that answers a query based on two labels. Section 5 analyzes the lengths of the labels. Finally, in Section 6 we show how to implement our algorithm so it runs in linear time.

2 Preliminaries

In this section we give a more formal definition of labeling scheme. Furthermore, we define an alphabetic code, subtree partition of a tree, and state the known results we use to construct our labeling.

Labeling schemes: An ancestor labeling scheme for a family $\mathcal{F}$ of trees with n nodes, consist of two mappings EC and DC. For a given tree $T \in \mathcal{F}$, the encoder EC label the nodes of the tree, i.e., $EC_T = EC(T)$ maps the nodes from T into labels. The decoder DC maps a pair of labels into $\{0, 1\}$ such that for all T, $DC(EC_T(v), EC_T(w)) = 1$ if and only if v is an ancestor to w in T. Notice that DC is independent of the tree from which the pair of labels is taken, it depends only on the two labels. In different papers [21, 18] there are additional requirements, such as different time constraints on the function DC. The function DC presented in this paper can be computed in constant time on a RAM assuming a standard set of operations as explained in the introduction.

Alphabetic codes: 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 $a, b \in \{0, 1\}^*$, $a <_{\text{lex}} b$ 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. A sequence $\langle b \rangle_k$ of binary strings is *lexicographically ordered* if $b_i <_{\text{lex}} b_j$, for all $1 \leq i < j \leq k$. 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 as follows. Assume $|a| \leq |b|$. We first align a and b by shifting a to the left by $|b| - |a|$ bits. Then we use standard integer comparison on the resulting words and return the result of this integer comparison if these words differ. If the two words are equal – this will happen if b equal to a followed by a sequence of zeros – we break the tie according to the lengths of a and b.

For a sequence of binary strings $\langle b \rangle_k$, we say it is *prefix-free* if no string is a prefix of another in the sequence. The following lemma due to Gilbert and Moore [17] states the existence of lexicographically ordered, prefix-free sequence of binary strings called an alphabetic code.

Lemma 2.1 (Gilbert and Moore [17]) *For any sequence $\langle y \rangle_k$ of positive integers with $n = \sum_{i=1}^{k} y_i$ there exists a prefix-free, lexicographically ordered sequence $\langle b \rangle_k$ where $|b_i| \leq \log n - \log y_i + O(1)$ for all i.*

For the sake of completeness we show how to construct an alphabetic code for an integer sequence $\langle y \rangle_k$ in $O(k)$ time. Let $s_0 = 0$, $s_i = \sum_{j=1}^{i} y_j$ for $1 \leq i \leq k$, $I_i = [s_{i-1} + 1, s_{i-1} + y_i]$, and $f_i = \max\{0, \lfloor \log y_i \rfloor - 1\}$. To each y_i we will find a number z_i in the interval I_i, such that 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 bits of z_i. Thus we get alphabetic codes where $1 \leq |b_i| \leq \log n - \log y_i + O(1)$. We choose z_i such that $z_i + 2^{f_i} - 1$ belongs to I_i to make the bit strings prefix-free. The computation can be done as follows. 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 $z_i = s_{i-1} + 1 - (s_{i-1} + 1 \bmod 2^{f_i}) + 2^{f_i}$.

The algorithm runs in $O(k)$ time if machine operations for shift, remainder in a division by a power of 2, and discrete logarithm on words of $O(\log n)$ bits are available. In a machine that does not support such operations we can construct a table representing these functions, using $O(n)$ time and space. This will only increase the preprocessing time of our labeling algorithm by a constant factor. Note that Mehlhorn [24] 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. Mehlhorn's algorithm can be implemented to run in $O(k)$ time.

Tree Terminology: We denote the sets of nodes and edges in T by $V(T)$ 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* of w, and we write $u \prec w$. If $w \in V(T(u)) \setminus \{u\}$ then u is a *proper ancestor* of w. If u is a (proper) ancestor of w, then w is a *(proper) descendant* of u. For nodes u and v in T we denote the path between u and v including u and v by $u \leadsto v$.

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 binary tree T', such that $|V(T')| \leq 2|V(T)|$ and a mapping $f : V(T) \to V(T')$, such that for $v, w \in V(T)$, we have that $v \prec_T w$ if and only if $f(v) \prec_{T'} f(w)$. Hence, in the remaining of this paper we assume without loss of generality that T is a binary tree.

Clustering: Let T be a rooted tree of size $n = |V(T)| > 1$. Let C be a connected subgraph of T. A node x in $V(C)$ is a *boundary* node if either $x = r$, where r is the root of T, or x is adjacent to a node in $V(T) \setminus V(C)$. The boundary nodes of C are denoted by ∂C. A *cluster* is a connected subgraph of T where $|\partial C| \leq 2$. Clusters are also used in [5, 6, 14].

Next we define a *cluster partition* of a tree T. If T is a single vertex then its *cluster partition* consists of one cluster containing this vertex. Otherwise a set of clusters CS is a *cluster partition* of a tree T if the following conditions hold

1. For every $C \in CS$, $|E(C)| \geq 1$.

2. $V(T) = \cup_{C \in CS} V(C)$, and $E(T) = \cup_{C \in CS} E(C)$.

3. For any $C_1, C_2 \in CS$, $E(C_1) \cap E(C_2) = \emptyset$.

4. If $v \in \partial C$ has two children a and b, then no cluster includes both a and b.

Note that the sets of edges of the different clusters form a partition of the set of edges of T. A vertex, on the other hand, may belong to more than one cluster. Figure 2 gives an example of a cluster partition. A variant of the following lemma was also used in [6, 5, 14].

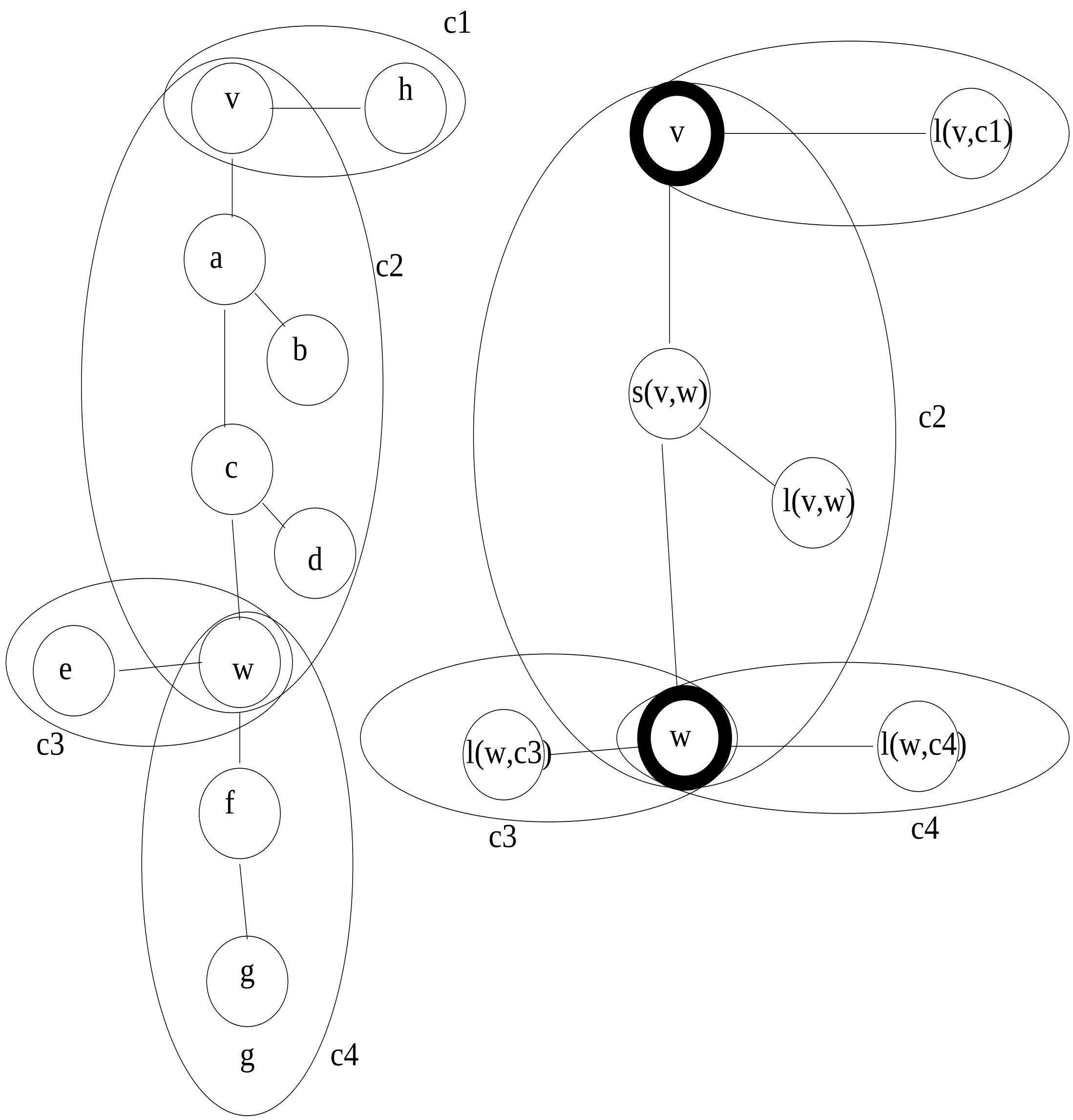

Figure 2: To the left a tree T decomposed to four clusters, where $V(c1) = \{v, h\}$, $\partial c1 = \{v\}$, $V(c2) = \{v, a, b, c, d, w\}$, $\partial c2 = \{v, w\}$, $V(c3) = \{w, e\}$, $\partial c3 = \{w\}$, $V(c4) = \{w, f, g\}$, and $\partial c4 = \{w\}$. To the right the tree T^m, illustrating from which clusters the nodes of T^m originate.

Lemma 2.2 *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, such that $|CS| \leq k \cdot x$, for a constant k, and $|V(C)| \leq \lceil n/x \rceil$ for every $C \in CS$.*

Proof. We describe an algorithm that constructs such a cluster partition. Let S be the set of boundary nodes picked until now; initially $S = \{r\}$. Now recursively, choose a node $v \in S$ not yet examined. Let the children of v be a and b. We treat the two children similarly, hence in the following we can assume $c = a$ (or $c = b$). If $|V(T(c))| + 1 \leq \lceil n/x \rceil$, we let the nodes $V(T(c))$ and v be a cluster, with the boundary node v. Otherwise, let $w \in V(T(c))$ be a maximal node such that $|V(T(c))| + 2 - |V(T(w))| \leq \lceil n/x \rceil$. (By *maximal* we mean that this inequality holds for w but does not hold for any of the children of w.) We let $V(T(c)) \setminus V(T(w)) \cup \{v, w\}$ be a cluster, with boundary nodes w and v, and add w to S. Finally, we remove v from S. It is easy to see that one can implement this algorithm to run in linear time.

Clearly we have $|V(C)| \leq \lceil n/x \rceil$ for every $C \in CS$. Notice that each node is a boundary node in at most three different clusters and each cluster has at least one boundary node. Therefore the number of boundary nodes is within a constant factor of the number clusters. So to bound the number of clusters we will prove an upper bound on the number of boundary nodes.

To bound the number of boundary nodes we consider separately those boundary nodes that have two children that are roots of subtrees of T of size $\geq \lceil n/x \rceil$, and those boundary nodes which have at least one child which roots a subtree of size $\leq \lceil n/x \rceil - 1$.

Consider first the set S of boundary nodes which have two children that are roots of subtrees of size $\geq \lceil n/x \rceil$. It is easy to see that by the definition of our algorithm if y and z are nodes in S then their lowest common ancestor must be a boundary node and therefore belongs to S. So if we connect each node in S to its immediate descendants that are in S we obtain a tree with all internal nodes having two children. Since the number of leaves in this tree is $O(x)$ (each roots a tree with at least $2\lceil n/x \rceil + 1$ nodes) we obtain that there are $O(x)$ boundary nodes in S.

Consider now a boundary node z that has one child y which roots a subtree of size $\leq \lceil n/x \rceil - 1$. By the definition of our algorithm z and the subtree rooted by y form a cluster C. Assume that z is not the root. Then by the definition of our algorithm the cluster C, and the other one or two clusters for which z is a boundary node, contain together at least $\lceil n/x \rceil$ nodes. (Otherwise z should not have been selected as a boundary node.) Therefore, if we charge each such boundary node evenly among the nodes of the clusters it is a boundary of, each node in these clusters is charged by at most x/n. Since each node is charged by a constant number of boundary nodes the total charge of a node is $O(x/n)$, and the total charge summed over all nodes is $O(x)$. So there are $O(x)$ boundary nodes with at most one child which roots a subtree of size $\leq \lceil n/x \rceil - 1$. ∎

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

Our labeling scheme is based on a particular tree decomposition technique. We start out in Section 3.1 defining this tree decomposition, and characterizing the ancestor relationship in terms of it. The labeling itself is defined in Section 3.2.

3.1 Tree decomposition

Let T be a binary tree with root r. (As explained in Section 2 when we defined our tree terminology, we assume T is a binary tree.) Let CS be a cluster partition of T as described in Lemma 2.2. We will fix the parameter x later. By the definition of a cluster partition we have that, 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, which we denote by $C(z)$. Furthermore, for $v, w \in \partial C$, either v or w is an ancestor in T of all nodes in $V(C)$. Let C be a cluster such that $\partial C = \{v, w\}$ we define the *spine path* of C to be the set of nodes on the path from v to w in T excluding v and w. We denote this path by $\mathcal{P}(v, w)$. In the example of Figure 2, $\mathcal{P}(v, w) = \{a, c\}$.

Given a cluster partition CS of T we define a *macro* tree, and denote it by T^m as follows. The vertex set of T^m consist of the boundary nodes of all clusters in CS, together with one or two new nodes for each cluster. For a cluster C with a single boundary node, say $\partial C = \{v\}$, we introduce one new node, denoted by $\ell(v, C)$. For a cluster C with two boundary nodes, say $\partial C = \{v, w\}$, where v is an ancestor of w we introduce two new nodes denoted by $\ell(v, w)$ and $s(v, w)$. The edges of T^m consist of an edge $(v, \ell(v, C))$ for every cluster C with a single boundary node, and the three edges $(v, s(v, w))$, $(s(v, w), w)$, and $(s(v, w), \ell(v, w))$, for every cluster C with two boundary nodes. See Figure 2.

Since r is a boundary node, then $r \in V(T^m)$. Furthermore, since there is a path from r to every boundary node $v \in T$, there is a corresponding path from r to v in T^m, and therefore T^m is connected. We can obtain T^m by deleting the edges that do not belong to spine paths, and then replacing each spine path $\mathcal{P}(v, w)$ by a path of two edges connecting v and w through $s(v, w)$. Finally for every cluster with a single boundary node we add a leaf $\ell(v, C)$ and connect it to v, and we also connect a leaf $\ell(v, w)$ to every node $s(v, w)$. From this description it is clear that T^m is indeed a tree. We define r to be the root of T^m. The properties of T^m that are summarized in the following lemma easily follow from the definition of T^m and Lemma 2.2.

Lemma 3.1 *Let T be a binary tree, and let T^m be the macro tree corresponding to a cluster partition of T. Then*

- *T^m is a binary tree.*
- *$|V(T^m)| \leq 4kx$, where k is the constant given in Lemma 2.2.*

- *The leaves of T^m are the nodes $\ell(\cdot,\cdot)$.*
- *For two boundary nodes $v, w \in T$, $v \prec_T w$ if and only if $v \prec_{T^m} w$.*

We classify each node of $v \in T$ into one of four different types as follows.

1. **Boundary node.** If v is a boundary node of a cluster in the cluster partition.
2. **Spine node.** If $C(v)$ has two boundary nodes and v is on the spine path of $C(v)$.
3. **Leaf clustered node.** If $C(v)$ has a single boundary node and v is in $C(v) \setminus \partial C(v)$.
4. **Internal clustered node.** If $C(v)$ has two boundary nodes, say u and w, and v is in $C(v) \setminus \{u, w\} \setminus \mathcal{P}(u, w)$.

For every node $v \in T$ we associate a *representative node* in T^m, which we denote by $a(v)$, as follows. If v is a boundary node then $a(v) = v$. If v is a spine node where $\partial C(v) = \{u, w\}$ then $a(v) = s(u, w)$. If v is a leaf clustered node then $a(v) = \ell(v, C(v))$, and if v is an internal clustered node then $a(v) = \ell(u, w)$ where u and w are the boundary nodes of $C(v)$. In the example of Figure 2, 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.

Let $z \in T$ be a spine node. By the definition of a spine node, node z has a child w which is either a boundary node or a spine node. We define $Isubtree(z)$ to be $T(z)$ from which we have removed the subtree $T(w)$ and the edge (w, z). It follows that $V(Isubtree(z))$ consists of z and all the internal clustered nodes in $C(z)$ whose nearest node on the spine path is z. For $w \in V(Isubtree(z))$, we define $sub(w) = z$. Thus, the $sub()$ mapping maps each internal clustered node to its closest ancestor on the spine path of its cluster. For nodes v and w on the same spine path, we define an ordering $<_\alpha$ such that $v <_\alpha w$, if and only if $depth_T(v) < depth_T(w)$.

It would be instructive at this point to understand how the ancestor relationship between nodes in T relates to the classification of the nodes, and their representatives in T^m. We summarize this relation in the following lemma.

Lemma 3.2 *Let $v \in T$, the followings cases give necessary and sufficient conditions for $v \prec_T w$ to hold, according to the type of v.*

1. *If v is a leaf clustered node then $v \prec w$ if and only if w is also a leaf clustered node, $a(v) = a(w)$, and v is an ancestor of w in $C(v)$.*
2. *If v is an internal clustered node then $v \prec w$ if and only if w is also an internal clustered node, $a(v) = a(w)$, $sub(v) = sub(w)$, and $v \prec_{Isubtree(sub(v))} w$.*
3. *If v is a spine node then $v \prec w$ if and only if one of the following conditions holds.*
   (a) $C(v) = C(w)$, w is also a spine node and $v <_\alpha w$.

(b) $C(v) = C(w)$*, w is an internal clustered node and* $v = sub(w)$ *or* $v <_\alpha sub(w)$*.*

(c) $C(v) \neq C(w)$ *and* $a(v)$ *is an ancestor of* $a(w)$ *in* T^m*.*

4. If v is a boundary node then $v \prec w$ *if and only if* $a(v)$ *is an ancestor of* $a(w)$ *in* T^m*.*

Lemma 3.2 indicates that we can compute ancestor relationship in T by computing one of the followings.

1. Ancestor relationship in T^m.

2. Ancestor relationship in $Isubtree(v)$ for a spine node v.

3. Ancestor relationship in a cluster C with a single boundary node.

4. Whether $v <_\alpha w$ for two spine nodes v and w.

We achieve 1 by using a DFS labeling of T^m. We achieve 2 and 3 by recursively decomposing clusters with a single boundary node and subtrees hanging off spine nodes. We achieve 4 by using an alphabetic code for the spine nodes on each spine path. The details of how exactly we construct the labels are in the next section.

3.2 The labels

Let $L^m : V(T^m) \to \{0,1\}^*$ be the *DFS* labeling of T^m. Recall that this *DFS* labeling associates a label consisting of $|L^m(v)| \leq 2\log x + \mathrm{O}(1)$ bits with each internal node v in T^m, and a label of $|L^m(\ell)| \leq \log x + \mathrm{O}(1)$ bits with each leaf ℓ in T^m. For binary strings b_1 and b_2 we denote the concatenation of b_1 and b_2 by $b_1 \cdot b_2$.

The label of each node starts with a prefix of constant length stating the type of the node. We denote the prefix which corresponds to a boundary node, a spine node, an internal clustered node, and a leaf clustered node, by `boundary`, `spine`, `int_clustered`, and `leaf_clustered`, respectively. The bits following this constant length prefix depend on the type of the node v as follows.

Boundary node. In addition to the type bits, $L(v)$ contains the label of v in T^m. That is

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

It follows that in this case $|L(v)| \leq 2\log x + \mathrm{O}(1)$.

Spine node. Let $a(v) = s(v', w') \in T^m$ and let $SP = \mathcal{P}(v', w')$ be the spine path of $C(v)$. Let $k = |SP|$ be the number of nodes on this spine path, and let $v_1, v_2 \cdots v_k$ be the nodes on SP ordered such that for $i < i'$, $depth_T(v_i) < depth_T(v_{i'})$. Clearly v is a node on this spine path, say $v = v_j$, for some $1 \leq j \leq k$. Let $T_i = Isubtree(v_i)$ and let $\langle c \rangle_k$ be an alphabetic code of the sequence $\langle |V(T_1)|, \ldots, |V(T_k)| \rangle$ constructed according to Lemma 2.1. Hence, c_j is the string corresponding to $V(T_j)$ in $\langle c\rangle_k$. Note that $|c_j| \leq \log \sum_i |V(T_i)| - \log |V(T_j)| + \mathrm{O}(1) \leq \log(n/x) - \log |V(T_j)| + O(1)$, since all nodes in T_i, $1 \leq i \leq k$, belong to the same cluster which contains no more than n/x nodes. The label of v is defined to be

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

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

Internal clustered node. Let $a(v) = \ell(v', w')$ and let $v_j = sub(v)$ be the spine node associated with v. As in the previous case, let $v_1, \ldots, v_k$ be the nodes on the spine path $\mathcal{P}(v', w')$, and let $\langle c\rangle_k$ be an alphabetic code for the sequence $\langle |V(T_1)|, \ldots, |V(T_k)|\rangle$, where $T_i = Isubtree(v_i)$. So c_j is the string corresponding to T_j in the code $\langle c\rangle_k$. By its definition the root of T_j has one child c in T_j. We define $T'_j = T_j(c)$ to be subtree of T_j rooted by this child. We recursively apply the labeling algorithm to T'_j. (with the same decomposition threshold x, which is not scaled relative to the size of T'_j.) Let $L^b(v)$ denote the label of node v in the recursive labeling of T'_j. The label of v in the tree T is defined to be

$$L(v) = \texttt{int\_clustered} \cdot L^m(\ell(v', w')) \cdot c_j \cdot L^b(v).$$

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

Leaf clustered node. Let c be the only child of the boundary node of $C(v)$ in $C(v)$. We recursively label the tree $T(c)$ (with the same decomposition threshold x, which is not scaled relative to the size of $T(c)$.). Let $L^b(v)$ be the label of v in this labeling of $T(c)$. Also let $a(v) = \ell(v', C)$. The label of v is defined to be

$$L(v) = \texttt{leaf\_clustered} \cdot L^m(\ell(v', C)) \cdot L^b(v).$$

We obtain in this case that $|L(v)| \leq \log x + |L^b(v)| + \mathrm{O}(1)$, where $|L^b(v)|$ is the length of $L^b(v)$.

We say that the prefix of $L(v)$ before the recursive labeling $L^b(v)$ is a *block* of the labeling $L(v)$. Similarly $L^b(v)$ consists of such recursive blocks, *i.e.,* $L(v) = b_1 b_2 \cdots b_k$, 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}. By the definition of the labels, the last block b_k is either a label of a spine node or a label of a boundary node.

The recursion ends when we get to label a tree of size no larger than x, in which case $\lceil |T|/x \rceil \leq 1$ and Lemma 2.2 does not apply.

In this case we consider all nodes as boundary nodes and take the macro tree T^m to be identical to T.

Consider a node $v \in V(T)$ and its label $L(v)$ with blocks $b_1 \cdots b_k$. 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 containing v, whose recursive labeling defined the suffix of the label of v to be $b_2 \cdots b_k$. In general for every $i < k$ we let $T(b_1 b_2 \cdots b_i)$ denote the subtree of T containing v whose recursive labeling defined the suffix of the label of v to be $b_{i+1} \cdots b_k$.

In addition to the labeling defined above we also keep the index of the first bit of the last block b_k, *i.e.,* $|b_1 \cdots b_{k-1}|$, using $\log \log n$ bits. This index which is of fixed length is stored at a fixed position of the word storing the label. Note that each block b_i stores a DFS label of a node in the macro tree corresponding to $T(b_1 b_2 \cdots b_{i-1})$. This DFS label occupies a fix number of bits at the beginning of the block. This number of bits is either $2 \log x + O(1)$ in case of a boundary node or a spine node and $\log x + O(1)$ in case of an internal clustered node or a leaf clustered node.

To parse the labels while answering a query the decoder needs to know the length and the position of the field storing $|b_1 \cdots b_{k-1}|$. Furthermore, it also needs to know our decomposition threshold x. The decoder have this information if it knows n, the size of the tree, since both these lengths are functions of n: the first is $\log \log n$ and the second will be fixed to $2^{\sqrt{\log n}}$. Since both lengths change quite slowly with n, the decoder in fact does not have to know n exactly, but up to a precision that allows it to determine these values uniquely. For example knowing $\log n$ certainly suffices. In case the value of $\log n$ is not available to the decoder one can encode it within each label using $\log \log n$ bits.

4 Answering 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, and b_i are the blocks of the labels of v and w, respectively, as defined in Section 3.2. We now show how to decide based on $L(v)$ and $L(w)$ whether $v \prec_T w$. The test consists of two phases defined as follows.

Let j be the index of the last block of $L(v)$, and assume first that $j > 1$. (If $j = 1$ we do nothing in the first phase and jump directly to the second phase.) It follows from Lemma 3.2 that if $v \prec w$ then v and w must both be either leaf clustered nodes or internal clustered nodes in the same cluster of the cluster partition of $T_i = T(a_1, a_2, \cdots a_i)$ for every $i < j$. Furthermore, in each of these cluster partitions where they are internal clustered nodes then it must be the case that $sub(w) = sub(v)$. Therefore, if $v \prec w$ and $j > 1$ we must have that for every $i < j$, $a_i = b_i$. So in the first phase of the query algorithm we verify that for every $i < j$, $a_i = b_i$. If this test fails we report that $v \not\prec_T w$ and otherwise we continue to the second phase.

To efficiently test that indeed $a_i = b_i$ for every $i < j$ without decomposing $L(v)$ and $L(w)$ into their components $a_1, a_2, \cdots, a_{j-1}$ and $b_1, b_2, \cdots, b_{j-1}$, respectively, we use the fact that

a_i cannot be a prefix of b_i and b_i cannot be a prefix of a_i. This follows from the fact that the part of a_i and b_i containing a DFS label of a node in $T_{i-1} = T(a_1, a_2, \cdots a_{i-1})$ is of fixed length, and in case v and w are internal clustered nodes of the same cluster in T_{i-1} also from the fact that the strings in an alphabetic code are prefix free.

Since a_i is not a prefix of b_i and b_i is not a prefix of a_i one can easily prove by induction that $a_1 a_2 \cdots a_{j-1}$ is a prefix of $b_1 b_2 \cdots$ if and only if $a_i = b_i$ for every $1 \leq i \leq j-1$. So we start the processing of the query by testing if $a_1 a_2 \cdots a_{j-1}$ is a prefix of $L(w)$. Note that we can extract $a_1 a_2 \cdots a_{j-1}$ from $L(v)$ since we store with $L(v)$ the index of the first bit of b_j. If $a_1 a_2 \cdots a_{j-1}$ is not a prefix of $L(w)$ then v is not an ancestor of w. Otherwise we proceed to the second phase of the algorithm which is defined as follows.

Let $\overline{T} = T(a_1, a_2, \cdots a_{j-1})$. Next we check whether $v \prec w$ in $\overline{T}$ since at this point we know that $v \prec w$ in T if and only if $v \prec w$ in $\overline{T}$. Since j is the index of the last block of $L(v)$, node v is either a spine node or a boundary node in the cluster partition of $\overline{T}$. Node w, on the other hand, may be of any type. We check whether v is an ancestor of w in $\overline{T}$ according to Lemma 3.2. Let $dfs(v)$, and $dfs(w)$ be the *DFS* labels of $a(v)$ and $a(w)$, respectively, in the *DFS* labeling of $\overline{T}^m$. We extract $dfs(v)$, and $dfs(w)$ from the blocks a_j and b_j of $L(v)$ and $L(w)$, respectively. Then we can determine whether the representative $a(v)$ of v in $\overline{T}^m$ and the representative $a(w)$ of w in $\overline{T}^m$ are equal, and whether $a(v)$ is an ancestor of $a(w)$, by comparing $dfs(v)$ and $dfs(w)$.

If $dfs(v)$ is equal to $dfs(w)$ then $a(v)$ is the same node as $a(w)$. We know that $a(v)$ is either a spine node or a boundary node. Since a boundary node v is not $a(w)$ for any $w \neq v$, and $a(v) = a(w)$, it follows that v and w are spine nodes. So by Lemma 3.2, $v \prec_T w$ if and only if w is a descendant of v on their spine path, i.e. if and only if $v <_\alpha w$. Let $c(v)$ and $c(w)$ be the alphabetic codes in blocks a_j and b_j of v and w, respectively. By the definition of an alphabetic code and the definition of the labels we have that $v <_\alpha w$ if and only if $c(v) <_{\text{lex}} c(w)$. So in this case we answer that $v \prec_T w$ if and only if $c(v) <_{\text{lex}} c(w)$.

If $dfs(v)$ is not equal to $dfs(w)$ then $a(v)$ is not equal to $a(w)$. By Lemma 3.2, a necessary condition for v to be an ancestor of w is that $a(v)$ would be an ancestor of $a(w)$ in $\overline{T}^m$, so we check that using $dfs(v)$ and $dfs(w)$. Furthermore, by Lemma 3.2, if $a(v)$ is indeed an ancestor of $a(w)$ in $\overline{T}^m$ then we can conclude that $v \prec_T w$ except when v and w are in the same cluster, v is a spine node, and w is an internal clustered node. In this case $a(v) = s(u_1, u_2)$ is the parent of $a(w) = \ell(u_1, u_2)$, where u_1 and u_2 are the boundary nodes of $C(v) = C(w)$. Fortunately, the DFS labeling scheme allows us to determine whether $a(w)$ is indeed a child of $a(v)$ and thereby identify this case where v is a spine node and w is an internal clustered node of the same cluster. First we check whether $a(w)$ is indeed a leaf, by verifying that $dfs(w)$ consists only of a single number. Assume this is indeed the case, let $k = dfs(w)$, and let $(first, last) = dfs(v)$. Then $a(v)$ is a leaf child of $a(w)$ if and only if $k = first + 1$ or $last = k + 1$.

If indeed v is a spine node and w is an internal clustered node of some cluster then by Lemma 3.2, $v \prec_T w$ if and only if $sub(w)$ is a descendant of v on v's spine path. Let $c(v)$

and $c(w)$ be the alphabetic codes in blocks a_j and b_j of $L(v)$ and $L(w)$, respectively. By the definition of the labels and the definition of an alphabetic code $sub(w)$ is a descendant of v on v's spine path if and only if $c(v) = c(w)$ or $c(v) <_{\text{lex}} c(w)$.

Since the decoder does not know the length of $c(w)$ it cannot easily extract it. However since $c(w)$ is not a proper prefix of $c(v)$ (as $c(w)$ and $c(v)$ are strings in an alphabetic code) we have that $c(v) \leq_{\text{lex}} c(w)$ if and only if $c(v)$ is lexicographically smaller than the suffix of the label of w starting from and including $c(w)$. So in case v is a spine node and w is an internal clustered node of some cluster of $\overline{T}$, the second phase concludes that $v \prec_T w$ if and only if $c(v)$ is lexicographically smaller than the suffix of the label of w starting from and including $c(w)$.

Note that the above test for the ancestor relation using the DFS labeling and alphabetic codes 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 sometimes more costly operations such as multiplication, retrieval of the most significant bit or non-standard operations pre-computed and stored in a pre-computed table.

5 The length of the labels

In this section we give an upper bound on the length of $L(v)$ for any $v \in T$. We first bound the length of any block which is not the last block.

Lemma 5.1 *Let $L(v) = b_1 b_2 \dots b_k$ for some $k \geq 2$. Then,*

$$b_1 \leq \log\left(\frac{n}{|T(b_1)|}\right) + O(1), \text{ and for } 1 < i < k, \;\; |b_i| \leq \log\left(\frac{|T(b_1 \dots b_{i-1})|}{|T(b_1 \dots b_{i-1} b_i)|}\right) + O(1)$$

Proof. Since b_i is not the last block in $L(v)$ then v is either an internal clustered node or a leaf clustered node in $T(b_1 \dots b_{i-1})$ or in T in case $i = 1$. In case v is an internal clustered node the claim follows from the definition of the label of an internal clustered node and the definition of $|T(b_1 \dots b_{i-1} b_i)|$ (see Section 3.2). In case v is a leaf clustered node then by the definition of the label we have that $|b_i| \leq \log x + O(1)$. But in this case we also have that $|T(b_1 \dots b_{i-1} b_i)| \leq \lceil |T(b_1 \dots b_{i-1})|/x \rceil$ by the definition of a cluster partition. ∎

The following lemma gives a bound on the length of the last block.

Lemma 5.2 *Let $L(v) = b_1 b_2 \dots b_k$. Then*

$$|b_k| \leq \log(|T(b_1 \dots b_{k-1})|) + O(\log x) + O(1)$$

Proof. Since b_k is the last block v is either a boundary node or a spine node in $T(b_1 \dots b_{k-1})$. If v is a boundary node then $|b_k| \leq 2 \log x + O(1)$. If v is a spine node then $|b_k| \leq \log(|T(b_1 \dots b_{k-1})|) + \log x + O(1)$. ∎

Last we bound the number of blocks.

Lemma 5.3 *Let $L(v) = b_1 b_2 \dots b_k$. Then $k \leq \frac{\log n}{\log x}$.*

Proof. By the definition of a cluster partition we have for $i < k$ that $|T(b_1 \dots b_{i-1})|/|T(b_1 \dots b_i)| \geq x$. Since $|T(\epsilon)| = |T| = n$ the lemma follows. ∎

Now from these three lemma we immediately obtain that

Lemma 5.4 *For every $v \in T$, $|L(v)| \leq \log n + O(\log x) + O(\frac{\log n}{\log x})$.*

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.

6 Preprocessing time

It is rather straightforward to compute the labels in time $\mathrm{O}(n\sqrt{\log n})$, spending linear time calculating the part of the labels in each of the $\sqrt{\log n}$ recursive levels. In each recursive level we calculate the cluster partition in each of the trees that get labeled in that level, and we also calculate alphabetic code for the nodes on every spine path.

We can improve this preprocessing time to linear by slightly changing the labeling scheme, while keeping the labels of length $\log n + \mathrm{O}(\sqrt{\log n})$. We do that as follow. We obtain a cluster partition of T with $x = n/\sqrt{\log n}$ as described in Section 2. Hence the clusters are of size at most $\sqrt{\log n}$. For this cluster partition we construct a tree T' that is similar to T^m defined in Section 3.1. We define T' as follows. For each cluster with two boundary nodes v and w, we have in T' the nodes v, w, $s(v,w)$, and two edges $(v, s(v,w))$ and $(w, s(v,w))$. For each cluster C with one boundary node v, we have in T' the nodes v and $\ell(v,C)$, and the edge $(v, \ell(v,C))$. (In contrast with T^m, in T' we do not have a leaf $\ell(v,w)$ for a cluster with boundary nodes v and w.) By Lemma 3.1, the tree T' consists of $\mathrm{O}(n/\sqrt{\log n})$ nodes. We label T' by the labeling scheme of Section 3 using the naive labeling algorithm mentioned above. Since the size of T' is $\mathrm{O}(n/\sqrt{\log n})$ computing this labeling of T' takes linear time. We denote the label of a node v in T' by $L'(v)$.

We also label the nodes of each cluster separately for ancestor queries using the *DFS* labeling scheme. This takes linear time for all clusters. We denote the label of v in its cluster by $CL(v)$.

For each node v in T we associate a node in T', denoted by $b(v)$. (This mapping is similar to the mapping $a()$ from Section 3.1.) If v is a boundary node then $b(v) = v$. If v is a spine node or an internal clustered node in a cluster with two boundary nodes u, w then $b(v) = s(u,w)$. If v is a leaf clustered node in a cluster C with boundary node w then $b(v) = \ell(w,C)$.

We define the label of a node $v \in T$ to be $L(v) = Q(v) \cdot CL(v) \cdot L'(b(v))$ where $Q(v)$ are two bits that determine the type of the node with respect to the cluster partition (see Section 3.2.) Since $|L'(b(v))| \leq \log n + O(\sqrt{\log n})$ and $|CL(v)| \leq O(\log\log n)$ we obtain that $|L(v)| \leq \log n + O(\sqrt{\log n})$.

To test if v is an ancestor of w in T we proceed as follow. If v is a boundary node, we check if $b(v)$ is an ancestor of $b(w)$. If v is a spine node and $b(v) \neq b(w)$, we test if $b(v)$ is ancestor of $b(w)$. If v is a spine node, and $b(v) = b(w)$, we use $CL(v)$ and $CL(w)$, to test if v is ancestor of w. In the remaining cases, for v to be an ancestor of w, $b(v)$ must be equal to $b(w)$, and if that is indeed the case then we check that v is an ancestor of w in their cluster using $CL(v)$ and $CL(w)$.

We conclude this section with the following theorem which is the main result of this paper.

Theorem 6.1 *We can label the nodes of a tree of size n in $O(n)$ time by labels of length at most $\log n + \mathrm{O}(\sqrt{\log n})$ such that given the labels of two nodes v and w we can determine whether v is an ancestor of w in T in $O(1)$ time.*

7 Conclusions

We described a labeling scheme for ancestor queries where the length of each label is bounded by $\log n + O(\sqrt{\log n})$ bits. The labeling is computed in linear time and an ancestor query takes O(1) time. As we mentioned, Alstrup, Bille, and Rauhe [3] proved that for every labeling scheme there is a tree that requires a label of length $\log n + \Omega(\log \log n)$ bits. Closing this gap between the upper and the lower bounds is an intriguing open question.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan-Kaufmann, 340 Pine Street, Sixth Floor, San Francisco, CA 94104, 1999.

[2] 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.

[3] S. Alstrup, P. Bille, and T. Rauhe. Labeling schemes for small distances in tree. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 689–698, 2003.

[4] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37:441–456, 2004.

[5] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming, 24th International Collo-*

quium (ICALP), volume 1256 of *Lecture Notes in Computer Science*, pages 270–280. Springer-Verlag, 7–11 July 1997.

[6] 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.

[7] S. Alstrup and T. Rauhe. Improved labeling schemes for ancestor queries. In *Proceedings of the thirteen annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[8] Altavista. http://www.altavista.com.

[9] M. A. Breuer. Coding vertexes of a graph. *IEEE Trans. on Information Theory*, IT–12:148–153, 1966.

[10] M. A. Breuer and J. Folkman. An unexpected result on coding vertices of a graph. *J. of Mathemathical analysis and applications*, 20:583–600, 1967.

[11] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *7th WWW*, 1998. http://www.google.com/.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1991.

[13] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for xml. In *International World Wide Web Conference*, 1999.

[14] 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.

[15] C. Gavoille and D. Peleg. Compact and localized distributed data structures. *Journal of Distributed Computing*, 16:111–120, May 2003.

[16] C. Gavoille, D. Peleg, S. Pérennès, and R. Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85–112, 2004.

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

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

[19] 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.

[20] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of the thirteen annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 954–963, 2002.

[21] M. Katz, N. A. Katz, A. Korman, and D. Peleg. Labeling schemes for flow and connectivity. *SIAM Journal on Computing*, 34(1):23–40, 2004.

[22] M. Katz, N. A. Katz, and D. Peleg. Distance labeling schemes for well-separated graph classes. *Discrete Applied Mathematics*, 145:384–402, 2005.

[23] Q. Li and B. Moon. Indexing and querying xml data for regular path expression. In *Prooceding of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 361–370, 2001.

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

[25] Oasis. Organization for the Advancement of Structures Information Standards. http://www.oasis-open.org/.

[26] D. Peleg. Proximity-preserving labeling schemes. *Journal of Graph Theory*, 33:167–176, 2000.

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

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

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

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

[31] W3. Extensible Markup Languange (XML) 1.0. http://www.w3.org/XML.

[32] W3C. Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL/.

[33] HyperText Markup Language (HTML). http://www.w3.org/MarkUp.

[34] Xyleme. A dynamic data warehouse for the XML data of the web. http://www.xyleme.com.