A comparison of labeling schemes for ancestor queries

Haim Kaplan * *Tova Milo* * *Ronen Shabo* *

July 29, 2001

Abstract

XML documents are often viewed as trees (basically the parse tree of the document), and queries over such documents typically test for ancestor relationships among tree nodes. Search engines process such queries using an index structure summarizing the ancestor relations. In the index, each document item (tree node), is identified using some logical id (node label), such that, given two labels, the engine can determine the ancestor relationship between the corresponding nodes, without accessing the actual document. The length of the labels is a main factor of the index size. Therefore, reducing this length, even by a constant factor, is a critical issue.

Labelings currently being used by actual systems are all variants of the following interval scheme: number the leaves from left to right and label each node with a pair consisting of the numbers of its smallest and largest leaf descendants. An ancestor test then amounts to an interval containment test on the labels. The maximum label length with this scheme is $2\log n$, where n is the number of nodes in the tree. A considerable amount of theoretical research has been devoted recently to reduce the worst case bounds on the labels length, with the current best scheme producing labels with maximum length of about $\log n + O(\sqrt{\log n})$. In contrast, we focus here on finding a scheme that works best *in practice* on real XML data. For that we propose several new prefix-based labeling schemes, where an ancestor query roughly amounts to testing whether one label is a prefix of the other. We analyze our new schemes both theoretically and empirically, comparing their performance to that of previously suggested schemes. Our experimental study shows that prefix-based schemes considerably improve the space consumption on real XML data.

The problem with the current theoretically best labeling schemes is that the constant multiplying the additive $\sqrt{\log n}$ is large and makes this term be the dominant when n is the current average size of an XML file. Nevertheless, to evaluate the practicality of the approach we have also implemented and tested two simplified versions with smaller additive factors (at the price of inferior asymptotic bounds). Our study shows that even this smaller factor is still dominating and the schemes did not perform on XML data as well as our new prefix based schemes. To obtain a better grasp of the additive constants we also tested the performance on some families of random trees (rather than just XML trees). Our experiments show that these schemes become competitive on trees with at least 20K nodes.

*School of Computer Science, Faculty of exact sciences, Tel-Aviv University, Tel Aviv 69978, Israel. E-mail: {haimk,milo,ronens}@post.tau.ac.il.

1 Introduction

The Web is constantly growing and contains a huge amount of useful information. To retrieve such data, people typically use search engines like Altavista [6] or Google [9] which provide full-text indexing services (the user gives a few words and the engine returns documents containing those words). The new emerging XML Web-standard [23] allows more sophisticated queries of documents. XML allows to describe the semantic nature of the document components, enabling users not only to ask full-text queries (e.g. find documents containing the word "Fielding") but also utilize the document structure to ask for more specific data (e.g. find all book items containing "Fielding" as an author and a price less than 12$) [12, 24, 1, 10, 3]. The key observation guiding the design of a search engine that supports structural queries, is that an XML document can be viewed as a tree whose nodes are the document items and whose edges correspond to the component-of relationship among data items. With this view, a structural query amounts to finding nodes with particular tags (book, price, author, etc.) having certain ancestor relationship between them (e.g. book nodes that are ancestors of qualifying author and price nodes).

The heart of a typical search engine [26, 25] is a big hash table whose entries are the item tag names and the words of the indexed documents. For each such tag or word the table contains the identifiers of all the documents containing it. To allow structural queries one adds to each such document-identifier an additional label associated with the particular node which contains the item within the document (tree). These labels are given such that one can decide whether one node is an ancestor of the other based on the labels of the two nodes alone. Thus structural queries can be answered by using the index only, without access to the actual document.

To allow for good performance it is essential that the index structure (or at least a large part of it) resides in main memory. Observe that we are talking here about an extremely large memory size. Just to give a rough measure, it is estimated that the current number of Web pages is around one billion and will grow to 100 billion by 2002 [11]. Search engines typically index 15% of these pages, consuming around 0.3 of the original document size, (even when only the text with the document ids is indexed, not including separation to items or ancestor information). With 10KB text per page on the average, this already leads to about 500 giga for now and 50,000 for 2002. 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. Therefore our goal in this research is to design a compact labeling scheme for the nodes of a tree such that given the labels of two nodes one can determine whether one is the ancestor of the other.

Note however that the exact minimization objectives depend on the specific physical representation of the labels in the index:

- Using a *fixed-length representation* every label is allocated the same amount of space. Therefore, the longest possible label determines the length of every label, and the size of the index. Using a fixed-length representation, the lower the worst-case guarantee on the length of a label, the smaller the space we need to allocate for the index.

- Using a *variable-length representation* each label may have a different length, consisting of a fixed-length prefix stating the actual length of the label, followed by the label itself. Here, the size of the index is essentially determined by the average length of a label (rather than the maximum length of a label). However, the maximum length of a label also has some importance (even if smaller). This is because the length of the fixed prefix of every label is the logarithm of the maximum length of a label.

Labeling schemes which are currently being used by actual systems [26, 25] are variants of the following *interval* scheme first suggested by Santoro and Khatib [21]. We number the leaves from left to right and label each node with a pair consisting of its smallest and largest leaf descendants. An ancestor query then amounts to an interval containment test on the labels. It is easy to see that the size of the labels generated by this scheme is bounded by $2\log n$ where n is the number of nodes in the tree. A variant of this scheme is integrated in the Xyleme XML warehouse system [26]. A considerable amount of work has been devoted recently to develop labeling schemes with worst case bounds smaller than $2\log n$. The best such labeling schemes use a recursive construction, and generates labels of length $\log n + O(\sqrt{\log n})$ [5] ([5] build upon the work of [2]. A related scheme is also suggested in [22].) Although theoretically best, these recursive schemes are not expected to improve on the simple interval scheme in practice. This is because the constant multiplying the additive $\sqrt{\log n}$ is large and makes this term be the dominant when n is the current average size of an XML file. In this paper we suggest labeling schemes which are likely to be winners in practice. We compare our schemes experimentally, using real XML data, to the interval scheme as well as to the recursive schemes of [2, 5, 22]. We also analyze our new schemes theoretically and prove worst case guarantees on the maximum label length.

Our Results. In contrast to the interval scheme we suggest to use a *prefix-based* approach, where the labeling is such that an ancestor query roughly amounts to testing whether one label is a prefix of the other. We start with a simple scheme that assigns a prefix-free collection of binary strings to the edges outgoing of each node. The label of a node v is the concatenation of the labels of the edges on the path from the root to v. It is easy to see that u is an ancestor of v iff the label of u is a prefix of the label of v. The way that the binary strings are assigned to the edges depends on the desired physical representation. For variable length representation we assign the binary strings so as to minimize the average length of a label. For fixed length representation we assign the strings so as to minimize the maximum length of a label.

This *simple prefix scheme* already exploits the structure of real XML files. It reduces the size of the index by 10% compared to the interval scheme using variable length representation, and by more than 20% using fixed length representation. Nevertheless this scheme suffers from bad worse case guarantees. For example for a tree which is a long path the labels are of length $\Theta(n)$. Therefore, a single skewed XML file, may increase the size of the index considerably, in particular for fixed length representation.

To remedy this we introduce the *compressed prefix scheme* that first transform the tree into a balanced one. We perform this balancing by partitioning the tree into paths and contracting each path into a single virtual node. Then we assign a collection of prefix free binary strings to the edges of the compressed tree outgoing of each virtual node. We label the nodes of the original tree using the assignment in the compressed tree. The labels and the ancestor test are only slightly more evolved than without compression. As for the simple prefix scheme, the way we assign binary strings to the edges of the compressed tree depends upon whether we use variable length representation or fixed length representation. Using compression we obtain an $O(\log n)$ worst case upper bound on the length of a label for both variable length representation and fixed length representation. As a result we can handle the indexing of isolated skewed trees without a large penalty in the size of the index. Furthermore, for fixed length representation, where the goal is to minimize the maximum length of a label, we give an algorithm which uses compression and produce labels of length at most $2\log n$. Thus matching the upper bound of the simple interval scheme.

The compressed prefix scheme not only has good worst case guarantees but it also performs well in practice on our data. For variable length representation we obtain using the compressed prefix scheme an additional 10% reduction in the size of the index compared to the simple prefix scheme. Summing up, this is a total of 20% reduction with respect to the interval scheme. Recall that in terms of the actual numbers we previously mentioned this means reducing the current index size, by over 100 giga bytes. For fixed length representation our experiments show that the compressed prefix scheme never produced labels of length larger than $1.5\log n$. The total reduction of the index size compared to the interval scheme was about 25%. We point out that our collection of XML files did not contain any particularly skewed tree. Therefore the advantage of the compressed prefix scheme over the simple prefix scheme has not been as large as it would have been in case even a single such tree appears.

The implementations of the simple prefix scheme and the compressed prefix scheme are straightforward. Although the asymptotic worst case time complexity of our labeling algorithms is slightly larger than that of the interval scheme[1], our experiments show that the actual running time of all labeling algorithms is negligible relative to the time required to load and parse the document. Therefore, the use of prefix labeling contributes no significant overhead to the time needed to prepare the index.

We also compared our prefix labeling schemes to a simplified version of the recursive labeling scheme with the currently best asymptotic bound of $\log n + O(\sqrt{\log n})$ bits on the length of the labels [5]. Rather than using a recurrence of depth $O(\sqrt{\log n})$ as in [5] (see also [22]) our implementation limits the recurrence to a two level partition of the tree. This, in order to reduce the constant factors while compromising for an asymptotic guarantee of $\frac{3}{2}\log n + O(1)$ bits. We also present a new 2-level scheme which combines our compressed prefix labeling scheme with ideas from [2, 5]. We prove that the new hybrid 2-level scheme has an upper bound of $\frac{3}{2}\log n + O(\log\log n)$ on the length of the labels. We also implemented and evaluated this new scheme in practice.

Despite careful tuning of the implementations of the 2-level schemes to avoid large constant factors, our experiments show that these schemes perform worse than the compressed prefix scheme and even the simple prefix scheme on real XML data. To get a better idea of the additive constant factors of the 2-level schemes, and to compare the performance of all algorithm on different distributions of trees besides XML, we used all algorithms to label random trees (from two different distributions) of varying sizes. This experiment shows that 2-level schemes become competitive on tree with at least 200K nodes.

One degenerate variant of the 2-level schemes that prunes every leaf which is a single child of its parent[2], performed particularly good on the XML data using variable length representation. This scheme exploits the fact that in a typical XML file most of the leaves are single children of their parents.

Additional Related work: Peleg [20] considered informative labeling schemes of trees and graphs for several types of problems. Maybe the problem closest to ours among the ones that Peleg studies is finding a labeling scheme that given the labels of two nodes u and v allows finding the identifier of the lowest common ancestor of u and v. Peleg described an $O(\log^2 n)$ labeling scheme for this problem and proves that this bound is tight up to a constant factor if the identifiers are predetermined.

Milo and Kaplan [17] have recently showed how to add the ability to answer parent queries to the theoretically best ancestor schemes with no asymptotic overhead. In another recent work Alstrup et al [4] show a labeling scheme that allows to identify nearest common ancestors.

The structure of the paper is as follows. Section 2 describes the basic prefix labeling scheme and Section 3 presents the compressed version, The implications of full text indexing is considered in Section 4. Section 5 describes the two-level labeling schemes. Finally, the experimental results are presented in Section 6. We conclude in Section 7. For lack of space we omit the proofs of our theoretical results and some details of our implementations from this extended abstract.

2 Simple prefix labeling

We assume in the sequel that all trees are such that each internal node has at least two children[3]. We denote by n the number of nodes in a tree T. We denote the subtree of T rooted at a node v by T_v, and the number of nodes in T_v by $size(v)$.

We call an assignment of binary strings to the edges of the tree, s.t. the collection of strings associated with the outgoing edges from any node is prefix free, a *prefix free assignment.* The *simple prefix scheme* first finds a prefix free assignment to the tree. Then it labels every node v with the concatenation of the strings assigned to the edges on the path from the root to v. It is easy to see that with such an

[1]The worst case running time of our prefix labeling algorithms is $O(n \log n)$. The running time of the interval scheme is $O(n)$.

[2]And then labels them with the label of the parent and an additional bit to distinguish them from their parent.

[3]If this is not the case to begin with we add a child to each internal node having only one child. This transformation at most doubles the number of nodes, and since the bounds on the labels length will be in terms of $\log n$ this only means addition of at most one single bit.

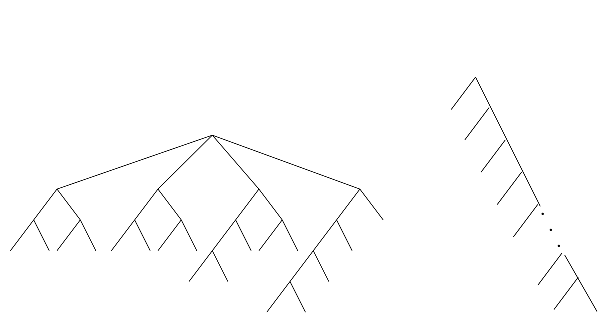


Figure 1: (a) MSL vs. MML (b) long prefix labels (c) Path decomposition

assignment labels are unique, and node u is an ancestor of v, if and only if the label of u is a prefix of the label of v.

For variable-length physical representation, when different labels have different lengths, our goal is to find an assignment that minimizes the sum of the lengths of the labels (or equivalently the average label length). We call this optimization problem the *Minimum Sum of Labels* (MSL) problem. When a fixed-length physical representation is used, namely all labels have the same length (determined by the longest label), our goal is to find an assignment that minimizes the maximum length of a label. We call this optimization problem the *Minimum Maximum Label* (MML) problem.

To see the difference between the two problems consider for example the tree shown in Figure 1(a). To minimize the sum of the lengths of the labels we assign to the edges outgoing from the root the strings $00, 01, 10, 11$ resp. All other nodes have exactly two outgoing edges to which we assign zero and one resp. (It will follow from Theorem 2.1 below that this is indeed the optimum labeling). On the other hand, to minimize the maximum length of a label we will assign to the edges outgoing from the root the strings $000, 001, 01, 1$ resp., which will make the maximum label length 5 rather than the 6 with the previous assignment. (Again, it will follow from Theorem 2.2 below that this is indeed the optimum labeling).

We solve the MSL problem using Huffman's algorithm [16]. Recall that given a set of weights $w_1, \ldots, w_n$, Huffman's algorithm finds a set $b_1, \ldots, b_n$ of prefix-free binary strings that minimizes the sum $\Sigma_{i=1\ldots n}(w_i \times |b_i|)$. To find the strings, the algorithm constructs a binary tree, whose leaves are the given weights, that minimizes $\Sigma_{i=1\ldots n}(w_i \times l_i)$, where l_i is the number of edges on the path from the root to w_i. The corresponding strings are then obtained from the tree by tagging the outgoing edges of each node by zero and one and concatenating the tags along the path from the root to each w_i. To construct the desired binary tree we start with a forest of n singleton nodes of weights $w_1, \ldots, w_n$. At each step the algorithm combines the two lightest trees into one tree (by adding a new node as a parent of the two roots) whose weight is the sum of weights of the two lightest trees. The algorithm ends when there is only one tree in the forest.

The MSL problem breaks into a collection of independent subproblems one for each internal node of the tree. Let v be an internal node whose children are $u_1, \ldots, u_k$. To obtain the labels of the edges outgoing of v we apply Huffman's algorithm to the (multi) set of weights $w_1, \ldots, w_k$ where $w_i = size(u_i)$, $1 \leq i \leq k$. (Recall that $size(u_i)$ is the size of the subtree rooted at u_i.) The following theorem establishes the correctness of this algorithm.

Theorem 2.1 *For a given tree T, the above algorithm computes a prefix free assignment that solves the MSL problem.*

We solve the MML problem using a variant of Huffman's algorithm that computes prefix-free collection of binary strings $b_1, \ldots, b_n$ that minimizes $\max_{i=1\ldots n}(w_i + |b_i|)$ rather than the sum $\Sigma_{i=1\ldots n}(w_i \times |b_i|)$. We call this variant the H-max algorithm. Given a set of weights $w_1, \ldots, w_n$ the H-max algorithm works as Huffman's algorithm merging at each iteration the two lightest trees. The difference is in the weight which H-max assigns to the new tree: Rather than taking the sum of the weights of the two merged trees, the H-max algorithm sets the weight of the new tree to be 1 + the largest weight among the two trees being merged. It is not hard to prove (see [14]) that indeed the H-max algorithm finds a string assignment which minimizes $\max_{i=1\ldots n}(w_i + |b_i|)$

As in the solution of the MSL problem we apply the H-max algorithm to each internal node v in order to label its outgoing edges. In contrast to the MSL algorithm here we have cannot handle each node independently but we have to handle them bottom-up. Let v be an internal node whose children are $u_1, \ldots, u_k$ (Assume we already labeled the subtrees T_{u_i}). To obtain the labels of the edges outgoing of v we apply the H-max algorithm to the (multi) set of weights $w_1, \ldots, w_k$ where w_i be the length of the longest concatenation of strings assigned to the edges of a root to leaf path in T_{u_i}. The following theorem establishes the correctness of this algorithm. The proof is by simple induction on the depth of T_v.

Theorem 2.2 *Given a tree T, the above algorithm computes a prefix free assignment that solves the MML problem.*

Remark: Note that in contrast with the interval scheme the prefix based schemes do not have a logarithmic worst-case upper bound on the maximum label length. As a simple example consider the tree in Figure 1(b). The label of the deepest leaf has $n/2$ bits. The maximum label length cannot in general exceed n since an arbitrary binarization of the tree[4] clearly gives labels of length bounded by n.

3 Compressed prefix labeling

In this section we enhance the prefix labeling with a preprocessing step that first compresses the tree. As a result labels of nodes even in a highly skewed trees cannot get too large. The penalty is a slightly more complicated ancestor test which incorporates another comparison in addition to the prefix decision. We will use the following notation. Given strings b_1, b_2, b_3 such that $b_2 = b_1.b_3$ we denote b_3 by *suffix*(b_2, b_1). We say that s_1 is *smaller* than s_2, denoted $s_1 < s_2$, when s_1 precedes s_2 in the lexicographic order of binary strings.

At a high level the algorithm works as follows. We decompose the tree T into disjoint paths, and construct a (partially ordered) compressed tree $\hat{T}$ by contracting each path into a big virtual node (and defining a partial order on the children of the node). We first label $\hat{T}$ using a scheme similar to the simple prefix labeling scheme of section 2. Once $\hat{T}$ is labeled we label each original node based on the label of the virtual node (path) to which it belongs. The label of a node v in T will be composed of two substrings *code*(v) and *separator*(v). All nodes on the same path will have the same *code* (which is essentially the label of the node corresponding to the path in $\hat{T}$), and will be distinguished by their *separator*. Furthermore, we assign codes and separators s.t. for every two nodes $v, u \in T$, v is an ancestor of u iff the following two conditions hold: (i) *code*(v) is a prefix of *code*(u), and (ii) *separator*$(v) \leq$ *suffix*(*code*(u), *code*(v)).*separator*(u).

We show how to choose the labels such that the length of the code concatenated with the separator of each node is bounded by $2\log n$. One can then encode each label using $2\log n + \log\log n + 2$ bits: The encoding consists of two fixed size fields one of size $2\log n + 1$ and the other of size $\log\log n + 1$. The first field contains the concatenation *code*(v).*separator*(v), padded at the beginning with a sequence of '0's ending with a single '1'. (The '1' denotes where in the field the actual data starts). The second field tells us how to decode the first field, namely, in which position the *code* ends and the *separator* starts.

Compressing the tree: We compress the tree by a quite standard partition of the tree into heavy paths. For a node v we define the *path of* v, denoted by *path*(v), as the set consisting of v and nodes w in T_v such that $size(w) \geq size(v)/2$ (note that these are all nonleaf nodes). Note that this set of nodes indeed form a path since each of v's descendants can have at most one child with weight $\geq size(v)/2$.

[4] Replace the outgoing edges of each internal node v by a arbitrary binary tree with the children of v at its leaves.

We partition the tree into paths recursively as follows. We start from the root v of T and work top down: We first find $path(v)$ and mark it as a new path. Then we continue recursively from every node not on $path(v)$ that is a child of a node on $path(v)$ and decompose all the subtrees rooted at these nodes in the same way. For a path p, we denote by $head(p)$ the node on p closest to the root, and by $parent(p)$ the parent of $head(p)$ in T. Our decomposition algorithm is illustrated in Figure 1(c). The decomposed paths are marked with thick dark lines. The head and parent of the path P are marked by a and b respectively.

We obtain the compressed tree $\hat{T}$ by contracting each path into one single node. Thus, the nodes of $\hat{T}$ are the paths of the path decomposition and a node $p \in \hat{T}$ is a child of a node $q \in \hat{T}$ if and only if $parent(p) \in q$. It is easy to see that the depth of $\hat{T}$ is at most $\log(n)$. (Since if q is a parent of p in $\hat{T}$ then $size(head(p)) < size(head(q))/2$). The *representative* in $\hat{T}$ of a node $v \in T$, denoted by $rep(v)$, is the node corresponding to the path that contains v. The next lemma shows that $\hat{T}$ preserves ancestor relationship among nodes it represents.

Lemma 3.1 *If v_1 is an ancestor of v_2 in T than either v_1 and v_2 are on the same path, so* $\mathrm{rep}(v_1) = \mathrm{rep}(v_2)$*, or they belong to distinct paths, in which case* $\mathrm{rep}(v_1)$ *is an ancestor or* $\mathrm{rep}(v_2)$ *in $\hat{T}$.*

For every internal node $r \in \hat{T}$ we define a partial order $\prec_r$ on the children of r such that $p \prec_r q$ if and only if $parent(p)$ is a proper ancestor of $parent(q)$. We will omit the subscript r from $\prec_r$ without confusion since every node in $\hat{T}$ participates in exactly one such partial order. For example in the compressed tree corresponding to the tree of Figure 1(c), K, L, M, N, O, P are children of Q and the partial order on them is $K \prec \{L, M\} \prec N \prec \{O, P\}$.

Assigning labels: We first label the nodes of $\hat{T}$ with a *prefix free assignment* but in addition we require that this assignment would respect the partial order defined on the edges outgoing out of each node of $\hat{T}$. Specifically, if q and q' are children of p and $q \prec_p q'$, then we require that the string assigned to the edge (p, q) is smaller than the string assigned to (p, q'). We call such an assignment to edges of $\hat{T}$ an *ordered prefix free assignment.*

Once we find an ordered prefix free assignment to the edges of $\hat{T}$ we label each node of $p \in \hat{T}$ by the concatenation of the strings assigned to the edges of $\hat{T}$ along the path from the root to p. Finally, we define the labels of the nodes of T based on the labels of the compressed tree $\hat{T}$. Recall that the label of a node v consists of two parts, $code(v)$ and $separator(v)$. We define $code(v)$ to be the label of $rep(v) \in \hat{T}$, and $separator(v)$ to be the smallest (lexicographically) string assigned to the outgoing edges of $rep(v)$ which point to a representative of a child of v, or the empty string if $rep(v)$ is a leaf node.

Our ancestor test is specified by the following theorem. The proof of this theorem follows from Lemma 3.1 and the fact that the codes and separators were generated based on an ordered prefix free assignment.

Theorem 3.2 *For two nodes v_1, v_2 in T, v_1 is an ancestor of v_2 iff* $\mathrm{code}(v_1)$ *is a prefix of* $\mathrm{code}(v_2)$*, and* $\mathrm{separator}(v_1) \leq \mathrm{suffix}(\mathrm{code}(v_2), \mathrm{code}(v_1)).\mathrm{separator}(v_2)$.

3.1 Finding an ordered prefix free assignment of $\hat{T}$

As in Section 2 the algorithm would be different for variable length representation and for fixed length representation. Notice that for each $v \in T$ the concatenation $code(v).separator(v)$ is a label of a node in $\hat{T}$, and each label of a node in $\hat{T}$ is $code(v)$ for some $v \in T$. Therefore for fixed length representation our strategy is to find an ordered prefix free assignment to $\hat{T}$ that minimizes the maximum length of a label of a node in $\hat{T}$. This also minimizes the maximum length of a label of a node in T. For variable length representation we look for an ordered prefix free assignment that minimizes the sum of the lengths of the code parts of the labels of the nodes of T.

Formally we define optimization problems that we call *Ordered Minimum Sum of Labels (OMSL)* and *Ordered Minimum Maximum Label (OMML)* which are analogous to MSL and MML of Section 2.

The input to each of these problems is a tree together with a partial order defined on the children of every node. The objective is to find an ordered prefix free assignment that either minimizes the sum of the lengths of the code parts of the labels (OMSL) in T, or minimizes the maximum length of a label (OMML), depending on the desired physical representation.

As for MSL and MML (see Section 2) we solve the OMSL and OMML problems by breaking them into little pieces and solving an independent problem for each node. The problem we have to solve at every node $p \in \hat{T}$ for the OMSL problem is the following. Let $w_i = size_T(head(q_i))$, for $1 \leq i \leq k$, where q_i is the ith child of p. Find a binary tree with leaves $w_1, \ldots, w_k$ that minimizes $\sum w_i l_i$ subject to the constraint that w_i must preceded w_j in an inorder traversal of the tree if $q_i \prec_p q_j$. Similarly, for the OMML problem we are looking for a binary tree that minimizes $\max\{l_i + w_i\}$, and satisfies the order constraints. Here w_i is the length of the longest concatenation of the binary strings assigned to a root to leaf path in the (recursive) solution to the problem rooted at q_i. It is straightforward to find the binary string corresponding to each edge outgoing from p from the binary tree. Theorems analogous to Theorem 2.1 and Theorem 2.2 hold for these algorithms, assuming we obtain an optimal solution to the subproblem of each node p.

The only missing link in what we described so far is an algorithm to find the desired binary tree for each node p. We cannot use Huffman's algorithm (or H-max) since these assign binary string for unordered set of weights without any guarantees on the lexicographic order among them. Recently, motivated by this work, Barkan and Kaplan [8] describe an algorithm to find the best tree (according to both measures mentioned above) which is consistent with a partial order defined on the weights. Their algorithm works for a family of partial orders that include those that arise in the OMSL and the OMML problems and is polynomial when the weights are polynomialy bounded[5].

We can prove that both for the OMSL and OMML problems the length of the resulting labels is $O(\log n)$ bits. For OMML we can use the special structure of the partial orders that arise at each node to prove the following stronger bound

Theorem 3.3 *Let T be a tree, and let $\hat{T}$ be the compressed tree of T. For the OMML problem the algorithm finds an ordered prefix free assignment of $\hat{T}$ such that the length of the resulting labels of nodes in T is bounded by $2\log n$.*

The proof of Theorem 3.3 relies on the following special structure of $\prec_p$ for every node $p \in \hat{T}$. We can partition the children of p into blocks $B_1, \ldots, B_{l(p)}$, where $l(p)$ is the length of the path that corresponds to p. The block B_i contains all the children of the i-th node on the path and all nodes in the same block are unrelated. Furthermore, by the definition of the heavy paths, the total weight of nodes in the last block is at least half the total weight. We call a partial order on a set of weights that has these properties a *balanced partial order*. Our proof of Theorem 3.3 is constructive and suggests another algorithm (in addition to the optimal algorithm of [8]) that labels $\hat{T}$ with labels of length at most $2\log n$ bits.

Unfortunately, both the algorithm of [8] and the algorithm suggested by the proof of Theorem 3.3 are quite complicated to implement. Therefore we implemented the following simple heuristics that still guarantee short labels.

Method 1 (Choose arbitrary total order): For the special case where the partial order on the set of weights $w_1, \ldots, w_k$ is in fact a total order, i.e. $w_1 \prec w_2 \prec \ldots \prec w_k$ there are simple polynomial time algorithms [13, 18, 15] for finding an optimal binary tree with $w_1, \ldots, w_k$ at the leaves from left to right that minimizes either $\Sigma_{i=1\ldots n}(w_i \times l_i)$ or $max_{i=1\ldots n}(w_i + l_i)$ (where l_i is the number of edges from the root to the leaf containing w_i). A binary tree containing $w_1 \prec w_2 \prec \ldots \prec w_k$ at the leaves from left to right is called an *alphabetic tree* and the corresponding binary strings are an *alphabetic code*.

Our first heuristic picks an arbitrary total order consistent with $\prec_v$ and finds an optimal alphabetic code consistent with this total order using one of the algorithms mentioned above. We can prove that

[5] The weights here are bounded by n.

both for OMSL and for OMML this heuristic still guarantee $O(\log n)$ size labels. In particular using a result of Mehlhorn [19] we can prove the following theorem.

Theorem 3.4 *The length of the labels obtained by the above scheme adapted to solve the OMML problem is bounded by* $\log(n) + 2d$*, where* d *is the depth of* $\hat{T}$*.*

Since the depth of $\hat{T}$ is at most $\log(n)$ Theorem 3.4 implies a bound of $3\log(n)$ on the maximum label length.

Method 2 (Exploiting the structure of the partial order): In this heuristic we try to exploit the fact that the partial orders are balanced with a simple algorithm. Since the partial order $\prec_p$ is balanced we can split the children of p into two groups, (i) an unordered group consisting of the children in the last partition set $B_{l(p)}$ and (ii) a partially ordered group consisting of all other children. Furthermore, the weight of the first group being larger than that of the second group. Now we can treat each group separately: (i) run Huffman's algorithm to assign strings to nodes in the unordered first group, and (ii) complete the partial order defined on the second group to an arbitrary total order and find the optimal string assignment consistent with this total order using one of the algorithm mentioned above. Finally we have to combine the string assignments of the two parts into one string assignment consistent with the original partial order $\prec_v$. We do that by adding a leftmost zero to the strings of the ordered part and a leading one to the strings of the unordered part. Intuitively we are able to get shorter labels since the weight of the unordered set is large and we avoid imposing unnecessary constraints on its elements. The algorithm which we use for solving the ordered and the unordered problems at each node depends on whether we solve the OMSL problem or the OMML problem.

4 Labeling a weighted tree

A typical XML index contains entries not only for the XML items appearing in the documents (i.e. the nodes in the document tree) but also for the words appearing in the document text. Looking at the document parse tree, this text appears at the leaves. It follows that while an internal node of an XML tree appear in the index just once, a leaf may appear several times, once per word occurring in the text at that leaf.

When the physical representation of labels is variable-length, hence labels have different lengths, we may want to take the number of words occurring in each leaf into account - we would like the labels of leaves containing large number of words to be shorter than those having a smaller number of words (and perhaps even shorter than labels of some internal nodes). In contrast with the interval labeling scheme, it is easy to adapt our prefix schemes to a setup where nodes have different weights reflecting the number of words that they contain: The core of our simple/compressed prefix schemes, tailored for variable length representation, were the algorithms for solving the MSL/OMSL problems, resp. If the weights of nodes vary we can modify these algorithms such that they minimizes the weighted sum of lengths of the labels rather than their unweighted sum. As result nodes with high weights will get smaller labels.

5 Two-level algorithms

As mentioned in the introduction the theoretically best scheme produces labels of length at most $\log n + O(\sqrt{\log n})$. This scheme is recursive. It partition the tree into $\sqrt{\log n}$ levels of forests, labels the forests independently using the interval scheme, and then carefully assembles the label of a node from the labels of its representatives in the different forests. Although theoretically best, this recursive scheme is not expected to improve on the simple interval scheme in practice. This is because the constant multiplying the additive $\sqrt{\log n}$ is large and makes this term be the dominant when n is the current average size of an XML file or even much larger. To check whether some of the ideas behind this scheme can generate significant practical improvements we implemented a simplified version of this recursive scheme that partition the tree into only two levels, rather then $\sqrt{\log n}$ levels.

The schemes of [2, 5] when restricted to two levels works roughly as follows. We prune subtrees of size smaller than $\sqrt{n}$. Then even if the original tree did not have unary nodes, such nodes may exist after pruning. To eliminate these nodes we contract induced paths of length about $\sqrt{n}$ into single vertices. We also add leaves such that each group of pruned subtrees of size at most $2\sqrt{n}$ has a representative leaf in the tree which remains after pruning. We denote by $\overline{T}$ the tree that remains after these transformations. We label the pruned subtrees, and $\overline{T}$, independently, using the interval scheme. We also assign to each node v on each contracted path a binary string whose length is inversely proportional to the size of the pruned subtrees hanging off of v. This is done such that the collection of binary strings associated with each path form an alphabetic code where the order corresponds to the order of the vertices on the path (using the algorithm of [15]). The codeword corresponding to a vertex v is concatenated to all interval labels of nodes in the subtrees hanging off of v. Finally we obtain the label of a pruned node u by concatenating the label of its representative leaf in $\overline{T}$ with the interval of u (with the alphabetic code sticked to it of u is pruned off of a contracted path). The label of node u on a contracted path is the label of the corresponding contracted node on $\overline{T}$ together with u's alphabetic codeword. We will refer to this scheme in the sequel as the *two-level interval scheme.* Although we limited the recurrence to a 2-level partition of the tree one can still prove an upper bound of $\frac{3}{2}\log n + O(1)$ bits on the maximum length of a label. For more details see [2, 5].

When we determine whether v is an ancestor of w there are three possible cases. If v and w are in different subtrees (hence their labels contain different ancestor leaf labels from $\overline{T}$) then they are clearly unrelated. If v and w belong to the same subtree we can determine whether v is an ancestor of w by comparing their labels within the subtree. Finally, if v is in $\overline{T}$ or on a contracted path and w is in a pruned subtree we can decide by comparing the label of w's leaf ancestor in $\overline{T}$ (that is incorporated in the label of w) to the label of v (we also have to use the alphabetic codes in case w is in a pruned subtree and v is on the corresponding contracted path).

Alternatively, we can use the same pruning and contraction process described above to split the tree but instead of using the interval scheme to label the pruned subtrees we can use the compressed prefix scheme (see Section 3). We call this scheme in the sequel the *mixed two-level scheme.* Once Theorem 3.3 is established it is not hard to prove the following performance guarantee on the mixed 2-level scheme (the proof is omitted from this extended abstract).

Theorem 5.1 *The lengths of the labels generated by the mixed 2-level scheme are at most* $\frac{3}{2}\log n + O(\log\log n)$.

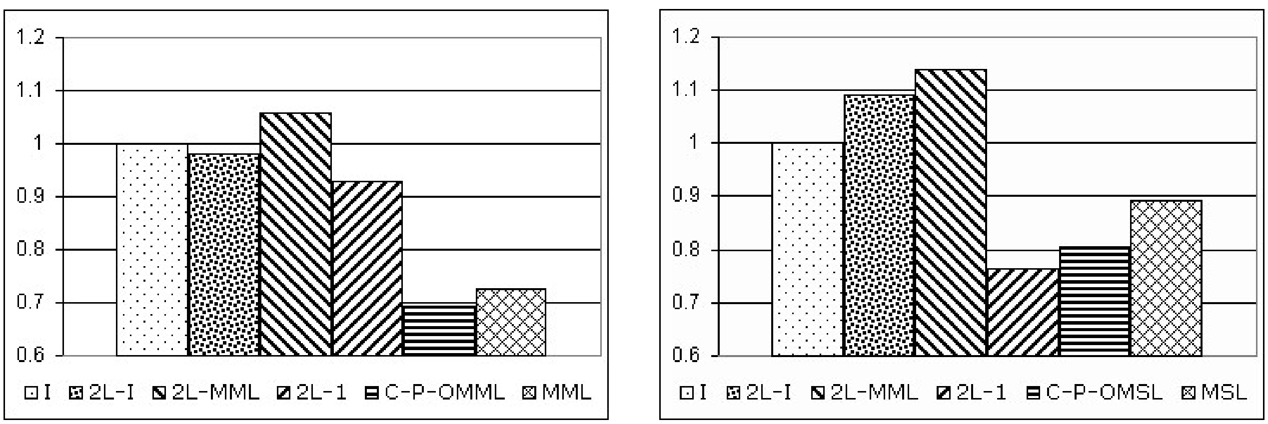
Intuitively, these 2-level schemes exploit the fact that, in the interval scheme, the length of a leaf-label is half the length of a label of an internal node. They obtain the better bound on the maximum label length by balancing out the length of a leaf-label with the length of the label of an internal node.

6 Experimental results

We have implemented the labeling schemes described in this paper and compared their performance on real XML data and random trees. We used two independent sets of XML files picked randomly from files collected by the XML Web-crawler of [26]. The first set contained 1822 files, with an average of 526 nodes per file and has been collected by the crawler by 22/5/2000. The second set contained 509 files, with an average of 338 nodes per file, and was collected by 12/10/2000. At the bottom line, prefix labeling schemes, even simple ones, perform about 20%-30% better than traditional interval labeling on the XML data. As we expected the performance of the prefix labeling schemes in practice is much better than their worst case guarantees. The typical XML file is relatively balanced and does not have the pathological structure required to make our algorithms achieve their worst case bounds. Notice that the average file size in both sets of files was relatively small. We expect that on collections with larger files our algorithms will perform even better, as the contributions of the small additive constants that they incorporate become more negligible. The two level schemes did not perform well on the real XML data.

We compared the labelings produced by various algorithms both when a fixed length representation is used and when a variable length representation is used. When using a fixed length representation we assumed that all labels for a single file (tree) have the same length which is determined by the largest among them. Labels in different files may have different lengths.[6] When using variable length representation, different labels can have different lengths (even within the same tree). Each label then essentially consists of two parts, a fixed length prefix specifying the overall length of the label (in bits), followed by the label value itself. Using variable length representation we can gain for example from allocating smaller labels to leaves when using the interval scheme, or allocating smaller space to shorter labels when using prefix based schemes.

The results which we show for both representations are when the label length could be any integral number of bits. We also compared the performance of the algorithms when each label must occupy an integral number of bytes. The results were similar and are therefore omitted. For variable length representation we also compared the algorithms while giving different weights to the nodes (according to the amount of text they contained) (see Section 4). Again the results were similar and are therefore omitted.



(a) Fix Length

(b) Variable Length

Figure 2: XML Trees, the size of the index relative to the interval scheme

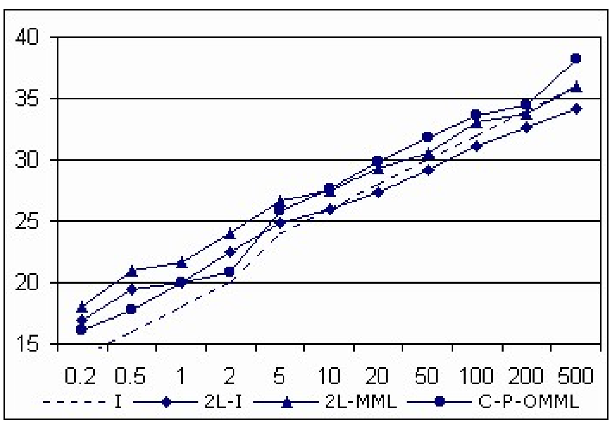
Our results for the first data set of XML files are shown in Figure 2. Figure 2(a) shows the results using fixed length representation and Figure 2(b) shows the results using variable length representation. For fixed length representation we compared the prefix labeling schemes which are tailored to minimize the maximum length of a label (algorithms of the MML family). For variable length representation we used the prefix labeling schemes tailored to minimize the sum of the labels (algorithms of the MSL family). The implementations of the compressed prefix schemes that exploit the partial order (method 2 of Section 3) performed slightly better than the implementations that just impose an arbitrary total order so only the results of former are shown (C-P-OMSL and C-P-OMML). The total size of the index (total size of the labels for all files) is shown relative to the interval scheme (I). Both the 2-level interval scheme (2L-I) and the mixed 2-level scheme (2L-MML), described in Section 5 could not significantly beat the interval scheme. They typically perform much worse due to their large additive constants and

[6] To parse a label we need to know to which type of file it belongs. Recall that the actual identification of a node in the index consists of the file id plus the node label in the file. One possibility is to incorporate into the file id information about the length of its labels. Alternatively, in a distributed environment, we can group the files based on their maximum label length and assign different set of machines to manage each group. Then, all labels stored on a single machine have the same length.

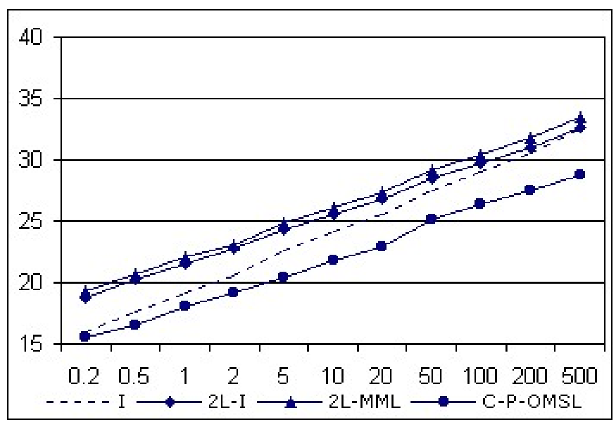
the small average file size. The prefix schemes however both without compression (MML and MSL) and with compression (C-P-OMML and C-P-OMSL) performed better than the interval scheme. The compressed prefix schemes outperform slightly the simple prefix scheme on the first data set, while it is the other way around for fixed length representation on the second data set (not shown in this extended abstract). The reason is probably the larger average tree size in the first data set: As the file gets larger the compression is more likely to have some balancing effect on the tree, which pays for the overhead of a more complicated label structure. Overall we see that prefix labeling is about 20%-30% more compact than the conventional interval scheme.

The last algorithm whose performance is shown in Figure 2 is a degenerate 2-level interval scheme (2l-1). This algorithm first prunes leaves which are single children of their parents and then applies the interval scheme to the remaining tree. Finally it labels the pruned leaves with the label of their parent, and adds an additional bit to all labels to distinguish between pruned leaves and the other nodes. This scheme performed better than the interval scheme, in particular using variable length representation, where it is even slightly better than the prefix-based schemes. This is due to the large fraction of leaves which are only children of their parents in the XML trees.

We also compared the algorithm on two distributions of random trees. The first are trees drawn uniformly at random from the set of ordered full[7] binary trees with n vertices (for varying values of n) [7]. The second are trees drawn uniformly at random from the set of ordered labeled trees with n vertices and no apriori bound on the degree. (Our algorithms of course did not use the labels or the order of the nodes.). We carried out this experiment for two main purposes. The first purpose was to better evaluate the additive constants of the 2-level schemes and check how large need a tree to be for these schemes to win. The second purpose was to check to what extent the relative performance of the algorithms on the XML data depends on the particular structure of the XML files.



(a) Fix Length



(b) Variable Length

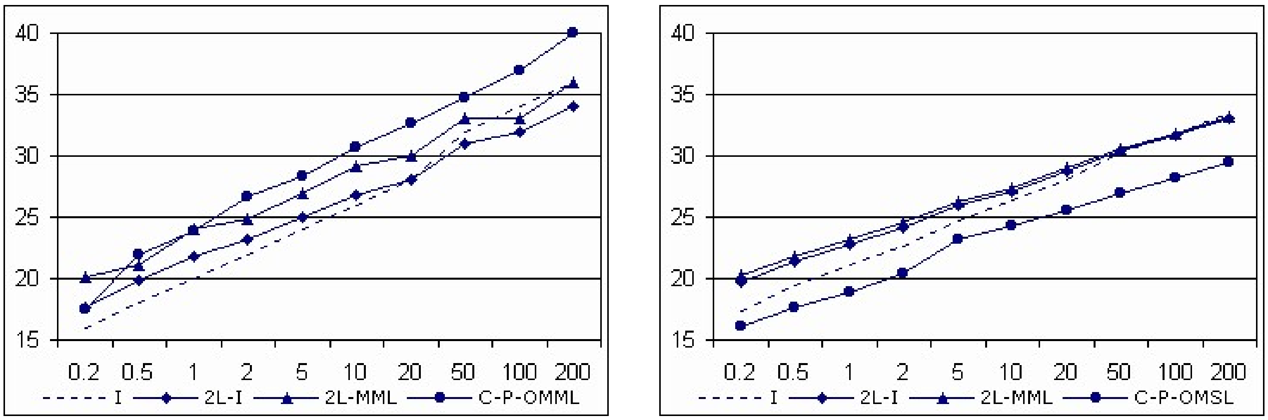
Figure 3: Binary Trees (number of nodes in Thousands vs. label bit length per node)

Our results for the random binary trees are shown in Figure 3, and the results for the order labeled trees are shown in Figure 4. We can see that the 2-level interval scheme performs slightly better than the 2-level mixed scheme on these distributions of trees. The additive constant of the 2-level interval scheme with respect to $\frac{3}{2}\log n$ is about 7. On the both distributions of trees the level schemes become competitive when the tree has over 20K nodes for fixed length representation, and over 200K nodes for variable length representation. The 2-level interval scheme performs slightly better than the 2-level mixed scheme in both representations.

In contrast with the XML files the random trees are often quite deep (for a random labeled tree with about 200K nodes the depth varies from 680 to 2030). Therefore plain prefix schemes without compression

[7]Each node is either a leaf or has both a left child and a right child

do not perform well (results are omitted). The performance of the compressed prefix schemes, however, is more interesting. For both families of random trees the appropriate compressed prefix scheme perform better than the interval scheme when using variable length representation but perform worse than the interval scheme when using fixed length representation. In particular this behavior holds when the average size of the random tree is the same as the average size of the XML files. Thus we conclude that at least with respect to fixed length representation the good performance of the compressed prefix scheme is due to the particular structure of the XML files.



(a) Fix Length

(b) Variable Length

Figure 4: Labeled Trees (number of nodes in Thousands vs. label bit length per node)

7 Conclusions and suggestions for further research

Motivated by the design of a compact index for massive amounts of XML data, we study in this paper the problem of giving short labels to the nodes of XML trees such that, given the labels of two nodes, one can determine whether one node is the ancestor of the other. In contrast to the traditional interval labeling scheme, we introduce a prefix-based approach, demonstrating its adequacy to the labeling of XML data.

One promising line for further research is to adapt the prefix labeling schemes to support updates. When an XML file is modified the search engine would want to attach labels to the new nodes without changing the labels of existing nodes. By doing that it can refrain from deleting labels from its hash table or rebuilding it altogether. Furthermore, by labeling the XML file incrementally through updates, the search engine can potentially index and answer queries about multiple versions simultaneously. One can enhance a prefix labeling scheme to be dynamic by inserting a *place-holder* node as an immediate child of each node that may gain descendants during updates. When insertions indeed occur the labels of these place-holders are expanded to labels of the new nodes, and new place-holders are incorporated in the new subtrees. Note that this dynamic generation of new labels is not possible with the interval scheme – even if we leave gaps in the numbering of the leaves so as to account for new added leaves, an intensive update to one specific part of the document may consume all the vacant ids and no additional nodes could be added without relabeling of existing nodes.

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, October 1999.

[2] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *SODA'01*, January 2001.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1, 1997.

[4] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Identifying nearest common ancestors in a distributed environment. Submitted, july 2001.

[5] Stephen Alstrup and Theis Rauhe, January 2001. private communication at SODA'01.

[6] Altavista. `http://www.altavista.com`.

[7] D. B. Arnold and M. R. Sleep. Uniform random generation of balanced parenthesis strings. *ACM Trans. Program. Lang. Syst.*, 2:122–128, 1980.

[8] A. Barkan and H. Kaplan. Partial alphabetic trees, 2001. Submitted.

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

[10] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[11] D. Butler. Souped-up search engines. *Nature*, 405:112–115, May 2000.

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

[13] E. N. Gilbert and E. F. Moore. Variable length binary encoding. *Bell systems technical journal*, 38:933–968, 1959.

[14] T. C. Hu, D. J. Kleitman, and J. K. Tamaki. Binary trees optimum under various criteria. *SIAM J. Appl. Math.*, 37(2):514–532, October 1979.

[15] T. C. Hu and C. Tucker. Optimum computer search trees. *SIAM J. Appl. Math.*, 21:514–532, 1971.

[16] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc IRE, 40*, pages 1098–1101, 1952.

[17] 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. Springer, August 2001.

[18] D. E. Knuth. Optimum binary search trees. *Acta Infomatica*, 1:14–25, 1971.

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

[20] D. Peleg. Informative labeling schemes for graphs. In *Mathematical Foundations of Computer Science (MFCS)*, pages 579–588, 2000.

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

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

[23] W3C. Extensible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml.

[24] W3C. Extensible stylesheet language (xsl). http://www.w3.org/Style/XSL/.

[25] Xdex. `http://www.xmlindex.com`.

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