Short and simple labels for small distances and other functions

Haim Kaplan
Tel Aviv University
haimk@math.tau.ac.il

Tova Milo
Tel Aviv University
milo@math.tau.ac.il

Abstract

We present a labeling scheme for rooted trees which allows to compute, from the label of v alone, unique identifiers for the ancestors of v that are at distance at most d from v. For any constant d our labeling scheme produce labels of length $\log n + O(\sqrt{\log n})$, and for $d \in O(\sqrt{\log n})$ the labels are still of length $O(\log n)$.

In particular, given the labels of two nodes u and v we can determine from the labels alone whether u is the parent of v or vice versa, whether u and v are siblings, and whether u and v are at distance at most d from each other.

The need for such labeling scheme arises in several application areas, including in particular communication networks and search engines for large collections of Web XML files. In the latter application XML files are viewed as trees, and typical queries ask for XML files containing a particular set of nodes with specific ancestor, parent, or sibling relationships among them.

1 Introduction

We present an algorithm that given a tree T assigns a unique identifier, $\mathrm{id}(v)$, and a unique label, $\mathrm{lab}(v)$, for every $v \in T$. These identifiers and labels have the property that given the label of a node v we can compute from this label alone the identifiers of all ancestors of v which are at distance at most d from v. The maximum length of a label or identifier that our algorithm produce is $\log n + O(d\sqrt{\log n})$, where n is the number of nodes in the tree. Thus for every constant d our labels are of length $\log n + O(\sqrt{\log n})$. This is close up to a second order additive term to the minimum of $\log n$ bits required to identify the nodes. Given the label of v we can construct the labels of all d ancestors of v starting from v in $O(1)$ time per ancestor.

In particular, our labeling scheme (with $d = 1$) allows parent and sibling queries. I.e. given the labels of two nodes u and v we can determine in constant time whether one is the parent of the other, and whether they are siblings (have a common parent). Furthermore, given the labels of two nodes u and v we can determine whether the distance between u and v is at most d. In case the distance is indeed at most d we can calculate it exactly from the two labels. This latter result is in contrast with a recent lower bound of $\Omega(\log^2 n)$ on the maximum label length if the labeling scheme allows to compute the distance between any pair of nodes [11]. Our result of this paper shows that if we limit ourselves to small distances then much shorter labels suffice.

Our labeling scheme is such that from the label of v alone we can compute the identifiers of the d closest ancestors of v. Since according to our scheme the label of a node v contains its identifier we can use it to determine if a node u is an ancestor of a node v at distance at most d from v from the labels of u and v[1]. In contrast, one may be interested in a labeling scheme that allow to determine whether u is an ancestor of v, using the labels of both v and u, for any pair of nodes u and v without any restriction on their distance. We call a labeling scheme that allows such queries a *labeling scheme for ancestor queries*. Notice that a labeling scheme for ancestor queries does not have any immediate implications for distance queries. The labeling scheme we describe in this paper does not support arbitrary ancestor queries. However, we can combine the technique presented in this paper with a recent labeling scheme for ancestor queries suggested by Abiteboul et al [2] and subsequently improved by Alstrup et al [3]. The resulting labeling scheme will then allow to identify the d closest ancestors of any node, and, additionally, to answer ancestor queries between *every* pair of nodes. Although the combined scheme is more complicated than the one we present here, the maximum label length is still $\log n + O(d\sqrt{\log n})$ (the constant factors are somewhat worse.)

[1] We compute from the label of v the identifiers of its d closest ancestors and check whether one of them is the identifier of u.

Our result builds upon the tree decomposition technique developed by Abiteboul at el in [2]. We decompose the tree by applying the tree decomposition algorithm recursively $\sqrt{\log n}$ times. Based on the resulting decomposition of the tree we assign unique identifiers and labels to the nodes. The identifier of a node v is a concatenation of identifiers of the representatives of v from the $\sqrt{\log n}$ recursive levels. The main technical contributions of the paper are the followings.
1) We separate the notion of identifiers of nodes and labels of nodes, and
2) We develop a technique that exploits the recursive structure of the identifiers and labels to compute the identifiers of the ancestors of v from the label of v.
To simplify the presentation, we demonstrate our technique in this extended abstract using particularly simple identifiers that allows to identify the ancestors of a node v that are at distance at most d from v. But as already mentioned, we can combine our technique with more complicated identifiers derived from the work of Abiteboul et al [2] and Alstrup and Rauhe [3], to obtain a scheme that also supports ancestor queries.

1.1 Related work

Motivated by applications that we will mention in Section 1.2, a lot of work has been devoted recently to the problem of labeling the nodes of a tree or a graph such that given only the labels of a set of nodes, S, one can determine the value of a particular function on S. The main objective of most labeling schemes that have been suggested is to minimize the maximum length of a label.

Early work on labeling schemes for graphs focused on *adjacency labeling schemes* where we want to determine whether u and v are adjacent based solely on their labels. Kannan, Naor, and Rudich [21] suggested $O(\log n)$ adjacency labeling schemes for a number of graph families, including trees, bounded arboricity graphs, and various intersection graph families.

For trees an adjacency labeling scheme is a labeling scheme that allows parent queries. Kannan et al [21] suggested the following simple scheme which we call the *pair scheme*. According to the *pair scheme* we number the nodes of T in some arbitrary order, by consecutive integers starting from 1. Then we label each internal node v by a pair $(n(v), n(p(v)))$, where $n(v)$ and $n(p(v))$ are the numbers of v and its parent respectively. Clearly node v is a parent of node u if v's first attribute equals to u's second attribute. It is easy to see that the length of the labels generated by this scheme is at most $2\log(n)$.

The pair scheme is the only labeling scheme we could find for parent queries in the literature. The labeling scheme we suggest in this paper (with $d = 1$) improves substantially on the $2\log n$ bound of Kannan et al, and produce labels of length $\log n + O(\sqrt{\log n})$. One can view our technique as an extension of the simple pair scheme. At a high level our algorithm partitions the tree into small subtrees and uses the pair scheme within each such subtree with special care of how to glue these independent pair schemes together.

The notion of *adjacency labeling scheme* has been extended by Peleg [20] who studied various *distance labeling schemes*. A distance labeling scheme allows to determine, or at least approximate, the distance between u and v from their labels. Peleg [20] gave a distance labeling scheme for trees producing labels of length $O(\log^2 n)$, and an approximate distance labeling scheme for general graphs. The approximate scheme for general graphs produce distances that are accurate up to a factor of $\sqrt{8k}$ with labels of length $O(\log^2 nkn^{1/k})$. Further extensive work on distance labeling schemes has been done by Gaviolle et al [11] and Peleg et al [16]. These papers establish lower and upper bounds on the label length for several graph families. In particular Gaviolle et al [11] show a lower bound of $\Omega(\log^2 n)$ on the label length of any distance labeling scheme for trees. Our result of this paper shows that if we limit ourselves to small distances then labels much shorter than $O(\log^2 n)$ suffice.

Recent work has also been focused on labeling scheme for ancestor queries. A simple such labeling first suggested by Santoro and Khatib [22] produce labels of length at most $2\log n$. Abiteboul et al [2] have recently presented a labeling scheme for ancestor queries with labels of length $\frac{3}{2}\log n + O(\log\log n)$. Alstrup and Rauhe [3] subsequently showed how to improve the bound of Abiteboul et al, to $\log n + O(\sqrt{\log n})$, by incorporating alphabetic trees [15] into their algorithm. A similar scheme has also been discovered independently by Zwick and Thorup [30] in the context of a routing application. As already mentioned our result builds upon the tree decomposition technique of Abiteboul at el as presented in [2].

We can obtain a labeling scheme for parent queries by taking the ancestor labeling scheme of Alstrup and Rauhe [3] and add to the label of v the height of v in the appropriate subtree that contains it. The label length would still be $\log n + O(\sqrt{\log n})$. This approach, however, does not seem to extend to allow sibling queries or distance queries, even for small distances.

Peleg [19] also considered informative labeling schemes of trees and graphs for functions other than adjacency, distance, or ancestor. One labeling scheme of Peleg allows finding the identifier of the lowest common ancestor of two nodes. Peleg provides 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. Another interesting labeling scheme that Peleg suggests allows to compute the weight of the steiner tree for a subset W of the nodes of a graph G from the labels of the vertices in W.

A more distantly related problem is the problem of preprocessing a tree T such that subsequent lowest common ancestor queries on pairs $v, w \in T$ can be answered quickly. Several algorithms have been developed for this problem [23, 14, 10, 7]. The preprocessing part of these algorithms organize some information on the tree such that one can efficiently find lowest common ancestors and in particular, given v and u determine if v is an ancestor of u. This information however is not organized in a distributed fashion as a labeling of the nodes.

1.2 Motivation

Applications for most of the labeling schemes that we mentioned in Section 1.1, as well as the new schemes suggested in this paper arise from two main areas. The first is routing in communication networks and the second is the design of search engines for large collections of XML files.

Communication Networks. Distance labeling schemes are useful in situations when a router knows the label[2] of the destination but does not know or does not have the time to access the topology of the entire network. Such router may want to choose between several routing options (or routing algorithms) based on the distance to the destination. In case the labels are carefully designed to allow distance or limited distance queries, the router can figure out the distance to the destination without accessing the topology of the network. A concrete application of distance labeling schemes for connection establishment in ATM networks is described in [20].

A distance labeling scheme can also be useful to limit flooding during broadcast or multicast of information. In such context a router may want to switch to unicast or TTL (Time To Live) based broadcast when all destinations reachable from it are relatively close. A close relation between ancestor labeling schemes and routing in trees is suggested by the work of Zwick and Thorup [30].

Note that when the label of the destination (or labels of the destinations) is transmitted with the packet long labels will increase the bandwidth. Each node also stores the labels of all destinations it may need to send data to, so if the labels are long these tables may become too large. Consequently, one would like to use as compact labels as possible.

XML search engines. The Web is constantly growing and contains a huge amount of useful information. To retrieve such data, people typically use search engines like Altavista [4] or Google [5] 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 [24] allows for the development of advanced search engines that support more sophisticated queries. XML allows to describe the semantic nature of the document components, enabling users not only to ask full-text queries but also utilize the document structure to ask for more specific data [18, 8, 26].

The key observation is that Web documents obeying the XML standard can be viewed as trees (basically the parse tree of the document). Typical queries over such documents then amount to testing various inclusion and direct inclusion of document items, which correspond to ancestor, parent, sibling, and similar relationships among the analogous tree nodes [9, 25, 27, 1]. To see an example consider the following query that searches for catalog publications whose title contains the word "'database" and where "Abiteboul" is an editor.[3]

```
SELECT  x
FROM    *.catalog.*  x,
        x.title      y,
        x.editor     z,
WHERE   y contains "database" and z contains "Abiteboul"
```

[2]Labels can also be used to identify the nodes in the network.
[3]The "*" symbol in the query denotes an arbitrary path.

Looking at the XML document (or more precisely, its corresponding parse tree) in Figure 1, the query retrieve all the publication nodes (e.g. `book, journal, proceedings,...` nodes) that are *descendents* of a `catalog` node, and have two sibling `title` and `editor` children whose associated text (which is omitted from the figure) contains the words "databases" and "Abiteboul" resp.

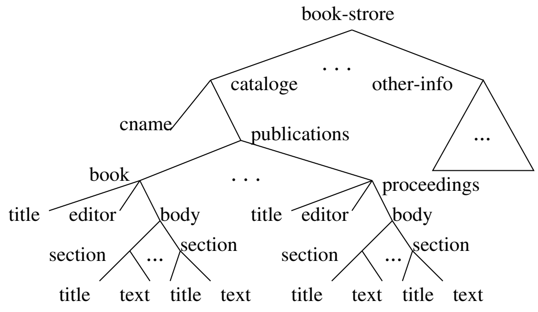

Figure 1: An XML tree

XML Web-search engines process such queries using an index structure that summarizes this structural information [29, 28]. Each node in the document tree is represented in the index by some logical id (label), with the labels being assigned to the nodes such that, given the labels of two nodes, the index can determine fast (by looking only at the labels and without any access to the actual file) whether one node is the parent or the ancestor of the other. To provide 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 huge numbers - thousands of giga bytes of labels [6]. Since a main factor which determines the index size is the length of the labels, reducing this length, even by a constant factor, is a critical issue, contributing both memory cost reduction and performance improvement.

Consider now the actual physical representation of the labels in the index. In a fixed-size representation each label is allocated fixed space, determined by the maximum potential label length. In a variable-size representation labels have different lengths (and thereby consist of a fixed prefix stating the actual length 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 length prefix of each label). We study here the case of fixed-size representation, aiming to improve the bound on the maximum label length. The case of variable-size representation is left for future research.

We use the RAM model of computation, and assume that the size of a computer word is $\Omega(\log n)$ (hence the basic operations on the labels can be performed in constant time). In particular we assume that the RAM model includes the ability to perform bit manipulations like shift, bitwise AND, OR, XOR, and base-two discrete logarithm in constant time.

The paper is organized as follows: We start in section 2 recalling the main principle of the tree decomposition algorithm of [2]. Section 3 demonstrates our techniques by giving a very simple scheme for parent queries using just one iteration of the tree decomposition algorithm. Section 4 shows how to reduce the length of the labels by adapting our technique to recursive pruning of the tree. Finally we show in Section 5 how to extend our ideas so we can compute identifiers for d closest ancestors. We conclude and suggest directions for further research in Section 6.

2 The prune&ccontract algorithm

In this section we give an overview of the tree decomposition algorithm of [2] and its properties. This will serve later as the basis of our new labeling scheme. We refer to this algorithm as the *prune&contract* algorithm. We assume that the input tree is such that each internal node has at least two children [4]. We denote by n

[4]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 labels lengths are in term of $\log n$, adds at most one one bit to the label).

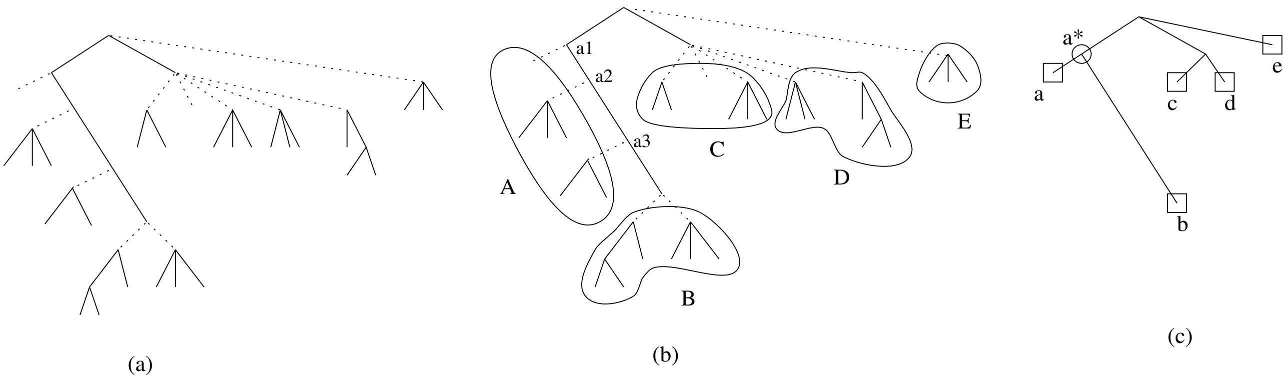

Figure 2: The pruning algorithm

the number of nodes in a tree T and by l the number of leaves. Since we assume that each node in T has at least two outgoing edges $n \leq 2l$.

The algorithm uses a parameter x that controls the size of the pruned subtrees. The algorithm has three phases. In our description below we omit some details that can be found in [2].

- The first phase prunes from T all subtrees having l^x leaves or less, by cutting the edges leading to these subtrees[5]. For example consider the tree in Figure 2(a). The tree has 27 leaves. When $x = \frac{1}{3}$ all subtrees containing three or less leaves are prunes. The dashed lines in Figure 2 indicate the pruned subtrees.
- The second phase groups the pruned subtrees into forests each having between l^x and $2l^x$ leaves[6]. We do the grouping such that the subtrees in each forest are either all children of a single node (as in the case of the forests B-E in Figure 2(b)), or they are the set of children of some path of nodes (as in the case of forest A; the nodes of the path here are denoted by a_1-a_3). The paths are such that their tail node is not a leaf of the remaining tree (in the above example a_3 is not a leaf - it has a non pruned child).
- The third phase adds to the remaining part of the tree a *representative-leaf* per each pruned forest. When the parents of the trees in the forest form a path we first contract the path into a single node and attach the new representative-leaf to that node. Otherwise we attach the leaf to the parent of the forest. An exception is when there is only one single forest below such a parent node. In this case rather than adding a new leaf, the parent itself serves as the representative of the forest. To continue with the above example, in Figure 2(c) the leaves a-e represent the forests A-E resp., and the node a^* corresponds to the contracted path a_1-a_3. The nodes $a, c-e$ are new leaves, while b belonged to the original tree.

Let $\overline{T}$ denotes the tree resulting from the above algorithm after pruning, contraction, and adding representatives. The following Lemma proved in [2] bounds the number of leaves in $\overline{T}$ and the length of the contracted paths. We shall use these lemma to establish the bound on the label length.

Lemma 2.1 *[2] The number of leaves in $\overline{T}$ is at most $3l^{1-x}$ where l is the number of leaves in T.*

Lemma 2.2 *[2] The length of each contracted path is at most $2l^x$.*

We shall use the following definition of the *representative* of a node to define our labeling schemes.

Definition 2.3 *Let v be some node in T. The* representative *node of v in $\overline{T}$, denoted by $r(v)$, is defined as follows. If v was contracted as part of a path into some node c in $\overline{T}$, or belongs to some pruned forest under such path, then $r(v) = c$. If v was pruned from T (but not under a contracted path) then it is represented in $\overline{T}$ by the representative-leaf corresponding to the forest to which v belongs. Otherwise, i.e. if $v \in T \cap \overline{T}$, then $r(v) = v$.*
Finally, for each node $w \in \overline{T}$ that represents pruned vertices we denote by $F(w)$ the forest of nodes of T that w represents.

[5]For some technical reasons we in fact also cut some subtrees having more than l^x but still at most $2l^x$, for details see [2].
[6]Some limited number of forests may in fact contain a smaller number of leaves.

We number each contracted path sequentially starting from 0 such that nodes closer to the root have larger numbers. For a vertex v that belongs to a contracted chain we define the *index of* v as its sequential number on its contracted path.

3 A simple two level parent labeling scheme

First we use the prune&contract algorithm with a parameter x that will be determined later to decompose the tree into pruned subtrees of size about n^x and a remaining tree $\overline{T}$ of size about n^{1-x}. We number the nodes of $\overline{T}$ and the nodes of each pruned forest independently. Using these numbers we define a unique identifier $\mathrm{id}(v)$ for each $v \in T$. The identifier of v consists of two fields. The first field is the number of $r(v)$ in $\overline{T}$. If v is in a pruned subtree then the second field contains the label of v in its pruned subtree. If v is not in a pruned subtree then the second field contains the index of v on its contracted path (or zero in case v is not on a contracted path). The second field also contains a bit which is set iff it contains a number of a pruned node rather than an index or zero.

The label of v, $\mathrm{lab}(v)$, consists of $\mathrm{id}(v)$ together with an additional information that allows us to compute the identifier of the parent of v from the identifier of v. The definition of this additional information follows from the following observation.

Observation 3.1 *During the prune&contract algorithm exactly one of the following cases holds.*

1. *The vertices v and $p(v)$ are on the same contracted path. In this case v and $p(v)$ share the same representative in $\overline{T}$ and the index of $p(v)$ is one greater than the index of v.*

2. *Vertices v and $p(v)$ are in the same pruned subtree. In this case v and $p(v)$ share the same representative and have different numbers in the numbering of their pruned subtree.*

3. *Vertex v is in a pruned subtree and vertex $p(v)$ is on a contracted path. In this case v and $p(v)$ share the same representative.*

4. *Vertex v is in a pruned subtree and is represented by $p(v)$.*

5. *Vertex v is either*
   1) the last node on a contracted path,
   2) a root of a pruned subtree that is represented by a new leaf added to $\overline{T}$,
   3) belongs to $\overline{T}$.
   Vertex $p(v)$ either belongs to $\overline{T}$ or is the first vertex on a contracted path. In this case $r(p(v))$ is the parent of $r(v)$ in $\overline{T}$.

The structure of $\mathrm{lab}(v)$ is thus defined as follows. In addition to $\mathrm{id}(v)$, $\mathrm{lab}(v)$ contains a *merge bit* which is set iff v and $p(v)$ have a common representative in $\overline{T}$ (cases (1-4) above). In case v and $p(v)$ do not have a common representative (case (5)) in $\overline{T}$ we also keep the number of $r(p(v))$ in $\overline{T}$. Otherwise, if v and $p(v)$ do have a common representative we also keep two bits indicating which of cases (1), (2), (3), or (4) occurred. When case (2) occurred we also keep the number of $p(v)$ in $F(r(p(v)))$, and when case (3) occurred we keep the index of $p(v)$ in its contracted path.

It is rather straightforward to check now that given $\mathrm{lab}(v)$ we can compute $\mathrm{id}(p(v))$: If the merge bit is not set we obtain $\mathrm{id}(p(v))$ by storing the number of $r(p(v))$ in $\overline{T}$ in the first field of $\mathrm{id}(p(v))$ and writing zero into the second field. If the merge bit is set then the way we obtain $\mathrm{id}(p(v))$ depends on which of the cases (1), (2), (3), or (4) happens. In case (1) we obtain $\mathrm{id}(p(v))$ by incrementing the index in the second field of $\mathrm{id}(v)$. In case (2) we obtain $\mathrm{id}(p(v))$ by replacing the content of the second field in $\mathrm{id}(v)$ with the number of $p(v)$ in its pruned forest. In case (3) we replace the second field in $\mathrm{id}(v)$ with the index of $p(v)$ on its contracted path and turn off the prune bit. In case (4) we simply write zero in the second field.

Labels length From Lemma 2.1 follows that the length of each field containing a number of a node from $\overline{T}$ is $\lceil \log(6 * n^{1-x}) \rceil$ bits [7]. From Lemma 2.2 follows that the length of a field storing an index is $\lceil \log(2n^x) \rceil$

[7] Recall that all internal nodes had at least two children so the number of nodes in the tree is at most twice the number of the leaves.

bits. Finally the length of a field containing a number of a node in a pruned forest is $\lceil \log(4 * n^x) \rceil$ bits [8].

By inspecting the definition of the labels it is easy to see that the largest labels are in cases (5), (3), and (2).

1. In case (5) the label consists of two numbers of nodes in $\overline{T}$ and either a number of a node in a pruned forest (if v was pruned) or an index (if v is the last node on a contracted path).

2. In case (3) the label consists of a number of a node in $\overline{T}$, a number of a node in a pruned forest, and an index.

3. In case (2) the label consists of a number of a node in $\overline{T}$, and two numbers of nodes in a pruned forest.

To balance out the lengths of these three kinds of labels we pick the pruning parameter to be 1/2. The maximum length of a label is then $\frac{3}{2}\log n + O(1)$.

In summary we proved in this section the following theorem

Theorem 3.2 *The labeling and identification scheme presented in this section produce labels of length $\frac{3}{2}\log n + O(1)$ such that from the label of a node v we can obtain the identifier of $p(v)$ in constant time.*

The following is an immediate corollary.

Corollary 3.3 *Using the labeling scheme described in this section one can determine from the labels of u and v alone whether one is the parent of the other, and whether u and v have a common parent.*

Proof. To determine whether u is the parent of v we produce the identifier of $p(v)$ from the label of v and compare it to the identifier of u that is part of the label of u. By theorem 3.2 these two identifiers are equal iff u is the parent of v. To determine whether v and u have a common parent we produce $\mathrm{id}(p(u))$ from the label of u and $\mathrm{id}(p(v))$ from the label of v and compare them. By Theorem 3.2 these identifiers are equal iff v and u are siblings. □

4 Smaller labels via recursion

In this section we show how to obtain shorter labels for parent queries by applying the prune&contract algorithm recursively. As in Section 3, we define a unique identifier $\mathrm{id}(v)$ for each node v. The label of v consists of the identifier of v together with additional information that allows us to compute the identifier of $p(v)$ from the identifier of v. By Applying the prune&contract algorithm recursively k times, we obtain k levels of pruned forests and a final remaining tree T_k. To identify a node v we need a number from each pruning level. However, taking advantage of the locality of the parent relation, we will see that to produce the identifier of the parent of v it suffices to store *one* additional piece of information from *one* particular pruning level.

Our recursive prune&contract process is similar to the one of [2] and we start by recalling its main principals. Then we explain how to use the various forests generated by the recursive prune&contract process to label the tree for parent queries. By applying the recursion $\sqrt{\log n}$ times, we obtain labels of length at most $\log n + O(\sqrt{\log n})$.

Recursive pruning The recursive prune&contract process works as follows. We denote the number of recursion levels by k. Let $x_1, \ldots, x_k$ be a sequence of numbers that we shall fix later. First we process T using the prune&contract algorithm with x_1 as the pruning threshold. Let $T_1 \equiv \overline{T}$ be the resulting tree. Then we apply the prune&contract algorithm again to T_1 with a threshold x_2 and denote the resulting tree $\overline{T_1}$ by T_2. We continue recursively and at each step define T_i, $1 \le i \le k$, to be the tree $\overline{T_{i-1}}$ obtained by applying the prune&contract algorithm to T_{i-1} with threshold x_i. (With T_0 being the original tree T.) We extend the definition of a representative, $r(v)$, of a node $v \in T$ from Section 2 such that $r^0(v) = v$ and $r^i(v) = r(r^{i-1}(v))$ for any $1 \le i \le k$.

As before we number the nodes of T_k consecutively starting from one. Then we do the same for the pruned forests: For each leaf or contracted node $w \in T_i$ such that $F(w)$ is not empty we number the nodes in $F(w)$

[8]Recall that in the pruning phase we grouped the trees to forests each having at most $2l^x$ leaves, and all the internal nodes had at least two children.

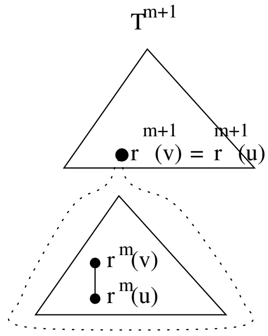

Figure 3: Case 1 of Observation 4.2

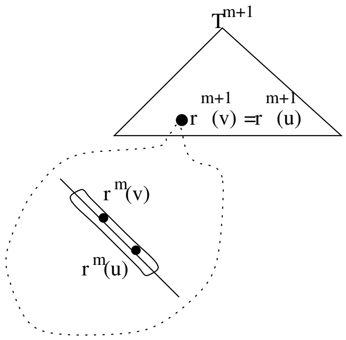

Figure 4: Case 2 of Observation 4.2

consecutively starting from one. Based on these numberings we define the identifiers and the labels of the nodes of T.

We start by defining the identifier of a node v, $\mathrm{id}(v)$. The identifier of v consists of $k+1$ fields ordered from field k down to field 0, and denoted by $f_k, \ldots, f_0$. Field f_i corresponds to the i-th iteration of the prune&contract algorithm. Field f_i starts with a *prune bit* that is set iff $r^i(v)$ was pruned. This bit is followed by the number of $r^i(v)$ in its pruned forest if it was pruned. If $r^i(v)$ was not pruned then the prune bit is followed by the index of $r^i(v)$ on its contracted path or by zero if $r^i(v)$ was not contracted (in which case we can think of v as contracted path of length 1.). It is easy to prove by induction on the number of iterations of the prune&contract algorithm that these identifiers are indeed unique.

The label of v consists of the identifier of v together with additional information that is needed to produce the identifier of the parent of v from the identifier of v. Before to define that additional information we need the following definition of the *merge-level* of a vertex v, and the subsequent observations.

Definition 4.1 *We define the* merge-level *of a vertex $v \in T$ as the maximum j such that $r^j(v) \neq r^j(p(v))$. We denote the merge level of a vertex v by $m(v)$.*

Let $m = m(v)$, the following observation characterizes the possible relations between $r^m(v)$ and $r^m(p(v))$.

Observation 4.2 .
If $m = m(v) < k$ then exactly one of the followings occurred,

1. *Vertices $r^m(v)$ and $r^m(p(v))$ are in the same pruned subtree of T_m. (Illustrated in Figure 3.)*
2. *Vertices $r^m(v)$ and $r^m(p(v))$ are in the same contacted path of T^m. (Illustrated in Figure 4.)*
3. *Vertex $r^m(v)$ is in a pruned subtree of T^m and vertex $r^m(p(v))$ is on a contracted path of T^m. (Illustrated in Figure 5.)*
4. *Vertex $r^m(v)$ is the root of a pruned subtree of T^m and $r^m(p(v)) = r^{m+1}(p(v))$ is a leaf of T_{m+1} representing v. (Illustrated in Figure 6.)*

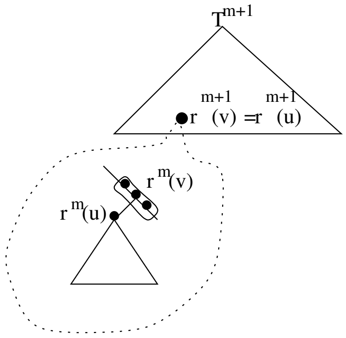

Figure 5: Case 3 of Observation 4.2

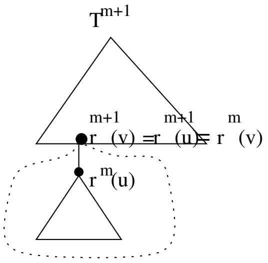

Figure 6: Case 4 of Observation 4.2

If $m(v) = k$ *then* $r^k(p(v))$ *is the parent of* $r^k(v)$ *(Illustrated in Figure 7 (i) and (ii).).*

The following lemma characterizes what can happen to $r^i(v)$ and $r^i(p(v))$ when $i < m(v)$. Its proof follows from the definition of $m(v)$ and the definition of the prune&contract algorithm.

Lemma 4.3 .

(i) For every $i < m(v)$ *either* $r^i(p(v)) = r^{i+1}(p(v))$ *or* $r^i(p(v))$ *is the first node on a contracted path to which* $r^{i+1}(p(v))$ *corresponds.*

(ii) For every $i < m(v)$ *either*
1) $r^i(v) = r^{i+1}(v)$*,*
2) $r^i(v)$ *is the last vertex on a contracted path to which* $r^{i+1}(v)$ *corresponds,*
3) $r^i(v)$ *is a leaf added to* T_i *to represent a pruned forest,* $r^{i-1}(v)$ *is a root of a tree in that forest.*

Now we are ready to define the additional information, denote by addi(v), that we add to id(v) in order to produce the identifier of $p(v)$ from the identifier of v. The label of v consists of both id(v) and addi(v). The first field of addi(v) contains the merge-level of v. We denote this merge level by m. If $m(v) = k$ then the

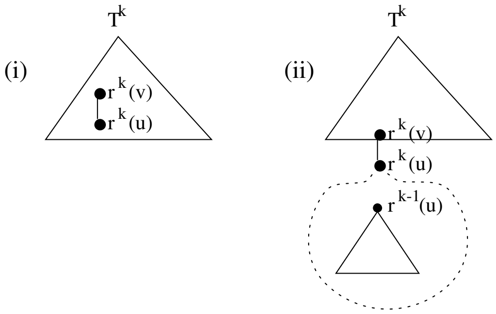

Figure 7: The case when $m(v) = k$ in Observation 4.2

second field contains the number of $r(p(v))$ in T_k. Otherwise the second field contains two bits indicating which of the cases in observation 4.2 occurred. If Case (1) occurred then addi(v) contains also the number of $r^m(p(v))$ in its pruned forest. If Case (3) occurred then addi(v) contains also the index of $r^m(p(v))$ on its contracted path.

Computing the identifier of $p(v)$ Given id(v) and addi(v) we compute id$(p(v))$ as follows. If the merge level of v is k then field f_k of id$(p(v))$ contains the number of $r^k(p(v))$ in T_k which is stored in addi(v), and all fields f_i for $i < k$ of id$(p(v))$ are zero. So let $m = m(v)$ and assume $m < k$. By the definition of the identifiers, and the definition of the merge level, fields f_i for $i > m$ of id$(p(v))$, are the same as the corresponding fields of id(v). By the first part of Lemma 4.3 fields f_i of id$(p(v))$ for $i < m$ are zero. To compute field f_m of id$(p(v))$ we check addi(v) to see which of the cases of Observation 4.2 occurred at iteration m of the prune&contract algorithm. If Case (1) occurred then field f_m of id$(p(v))$ contains the number of $r^m(p(v))$ in its pruned forest. This number is stored in addi(v). This field also has the pruned bit turned on. If Case (2) occurred we obtain field f_m of id$(p(v))$ by adding one to the index stored in field f_m of id(v). If Case (3) occurred field f_m of id$(p(v))$ contains the index of $r^m(p(v))$ in its contracted path that is stored in addi(v), with the pruned bit turned off. If Case (4) occurred then field f_m of id$(p(v))$, including the pruned bit is zero.

We show below the length of a label will be about $\log n$ bits, and we will set the fields f_i to be of the same length. Therefore, assuming that a computer word can hold $\log n$ bits then it is straightforward to check that we can compute id$(p(v))$ from id(v) in constant time.

The length of the labels Each field in the label is of fixed length that is determined by the maximum possible value that can be assigned to that field. Recall that by Lemma 2.1 the number of nodes in T^k is at most $6^k n^{(1-S)}$ where $S = \sum_{i=1}^{k} x_i$. Also, by Lemma 2.2 for every $1 \le i \le k$ the length of the index of $r^{i-1}(v)$ is at most $2 * n^{x_i}$ and the size of the corresponding pruned forest is at most $4 * n^{x_i}$. So, for a vertex v we need

$$\lceil \log(6^k n^{(1-S)}) \rceil + \sum_{i=1}^{k} \lceil \log(4 * n^{x_i}) \rceil + \lceil \log(max(6^k n^{(1-S)}, 4n^{x_1}, \ldots, 4n^{x_k})) \rceil + O(k)$$

bits for the label. This is minimized when $1 - S$ and x_i, $i = 1 \ldots k$, are all equal. When choosing the x_i's accordingly we get a labeling scheme where the maximum label length is

$$(1 + \frac{1}{k+1}) \log n + O(k) \ .$$

By setting $k = \sqrt{\log n}$ we get labels of length $\log n + O(\sqrt{\log n})$.

To summarize the result of this section we have the following theorem

Theorem 4.4 *The identification and labeling scheme we presented in this section assigns identifiers and labels of length $\log n + O(\sqrt{\log n})$ to the nodes of a tree such that given the label of a node v we can compute the identifier of $p(v)$ in constant time.*

The following is an immediate corollary whose proof is similar to the proof of Corollary 3.3.

Corollary 4.5 *One can label the nodes of a tree T with labels of length $\log n + O(\sqrt{\log n})$ such that given the labels of u and v we can determine from the labels alone whether one is the parent of the other, and whether u and v are siblings.*

5 Obtaining the identifiers of the d closest ancestors

Our algorithm from the previous section extends in a straightforward manner to produce labels from which we can compute the identifiers of all ancestors of v which are at distance at most d from v. We use the recursive prune&contract process and the identifiers as defined in Section 4. The label of v contains id(v) and an extended additional field, addi$_d(v)$, that will allow to compute the identifiers of d ancestors of v. The extended label of a vertex lab$_d(v)$ consists of id(v) and addi$_d(v)$.

The field addi$_d(v)$ is a concatenation of the fields addi$(p^i(v))$, for $0 \le i \le d-1$. Given id(v) and addi(v) we can use the algorithm of Section 4 to compute id$(p(v))$. Then we can take id$(p(v))$ and addi$(p(v))$ and compute

$\mathrm{id}(p^2(v))$ using the same algorithm. We continue this way and apply the algorithm of Section 4 d times until we have computed $\mathrm{id}(p^d(v))$. Since each of the fields $\mathrm{addi}(p^i(v))$ is of length $O(\sqrt{\log n})$, the length of $\mathrm{addi}_d(v)$ is $O(d\sqrt{\log n})$, and our labels for computing the identifiers of d ancestors are of length $\log n + O(d\sqrt{\log n})$. We conclude with the main result of this Section.

Theorem 5.1 *There are sets of unique identifiers* $\mathrm{id}(v)$ *and unique labels* $\mathrm{lab}(v)$ *that one can assign to the nodes of the tree* T *such that from* $\mathrm{lab}(v)$ *we can compute* $\mathrm{id}(p^i(v))$ *for* $1 \le i \le d$*. The maximum length of an identifier or a label is* $\log n + O(d\sqrt{\log n})$*.*

Our labeling scheme can be used for distance queries. Given the labels of u and the label of v we can determine whether u and v are at distance at most d from each other. We do that by producing the identifiers of the d closest ancestors of v and the d closest ancestors of u. If there is no intersection among these lists of identifiers then we conclude that the distance between u and v is larger than d. Otherwise we find the identifier, $\mathrm{id}(z)$, of the lowest common ancestor, z, of u and v. By summing up the sequential number of z on the list of the ancestors of v and the sequential number of z on the list of the ancestors of u we obtain the distance between u and v. Notice that if u and v are of distance at most d from each other then their lowest common ancestor is of distance at most d from each of them and therefore it will show up in the list of the d closest ancestors to v and in the list of the d closest ancestors to u. So we have established the following corollary of Theorem 5.1.

Corollary 5.2 *The labeling scheme of Theorem 5.1 allows to determine the distance between* u *and* v *if this distance is less than* d *or figure out that the distance between* u *and* v *is larger than* d*. This labeling scheme also allows to determine whether* u *is an ancestor of* v *and at distance* j *from* v *for any* $j \le d$*.*

Remark: The above scheme is based on the assignment of simple identifiers to the tree nodes, and then attaching to each node a label that enhances the identifier by some additional information required for identifying the d closet ancestors. Using the technique we describe in this paper combined with more complicated identifiers derived from the work of Abiteboul et al [2] and Alstrup and Rauhe [3], we can obtain labels that support arbitrary ancestor queries in addition to identifying the d closest ancestors of any given node. Although the combined scheme is more complicated than the one we present here, the maximum label length is still $\log n + O(d\sqrt{\log n})$ (the constant factors are somewhat worse.) We omit the details for lack of space.

6 Conclusions

We have presented an algorithm that assigns identifiers and labels to the nodes of a tree of length $O(\log n + d\sqrt{\log n})$ such that given a label of a node v one can compute the identifier of the d immediate ancestors of v. In particular our labeling scheme allows to determine the distance between u and v from their labels if this distance is less than d, or figure out that the distance is larger than d.

When d is small then our labels are particularly short but as d gets larger our labels may become even linear in the number of nodes. In contrast, Peleg in [20] shows a labeling scheme for trees whose labels are of length $O(\log^2 n)$ which allows to determine any distance between u and v. So one may try to develop a labeling scheme, that works the same for every d, and is as efficient as ours for small d but degrades to no more than $O(\log^2 n)$ for larger values of d. Another line for further research is to implement our algorithms and test their performance on real XML data.

Index structures for semi-structured data were studied in [12, 17] for the processing of regular path expression. They use an orthogonal approach where all the similar paths in the database are factorized, and the database graph is 'compressed' into a smaller graph. This index graph is then viewed as an automata (a deterministic one in [12] and a non deterministic one in [17]), on which the path expressions are evaluated. While being typically smaller than the original database, the resulting graphs are still rather big [13]. It may be interesting to investigate the incorporation of the compact labeling to get a more efficient and compact representation of the automata graph.

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 *Proc. 12th Symp. Discrete Algorithms (SODA'01)*, January 2001.

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

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

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

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

[7] R. Cole and R. Hariharan. Dynamic LCA queries on trees. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*.

[8] R. Cover. The xml cover pages, extensible markup language - industry news. http://www.oasis-open.org/cover/xmlNews.html.

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

[10] H. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'90)*, pages 434–443, 1990.

[11] Cyril Gavoille, David Peleg, Stephane Perennes, and Ran Raz. Distance labeling in graphs. In *Proc. 12th Symp. Discrete Algorithms (SODA'01)*, January 2001.

[12] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *In Proc. VLDB 97*, 1997.

[13] R. Goldman and J. Widom. Approximate DataGuides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1999.

[14] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–335, May 1984.

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

[16] M. Katz, N. Katz, and D. Peleg. Distance labeling schemes for well-separated graph classes. In S. Tison H. Reichel, editor, *STACS'00*, volume 1170 of *Lecture Notes in Computer Science*, pages 370–381. Springer Verlag, 2000.

[17] T. Milo and D. Suciu. Index structures for path expressions. In *In Proc. ICDT 99*, 1999.

[18] Oasis. Organization for the Advancement of Structured Information Standards. `http://www.oasis-open.org`.

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

[20] David Peleg. Proximity-preserving labeling schemes and their applications. In *Graph-Theoretic Concepts in Computer Science, 25th International Workshop, WG '99*, volume 1665 of *Lecture Notes in Computer Science*, pages 30–41. Springer Verlag, 1999.

[21] S. Rudich S. Kannan, M. Naor. Implicit representation of graphs. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC'88)*, pages 334–343, 1988.

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

[23] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, December 1988.

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

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

[26] W3C. The w3c query languages workshop, dec 1998, boston, massachussets. http://www.w3.org/TandS/QL/QL98/cfp.html.

[27] W3C. Xsl transformations (xslt) specification. http://www.w3.org/TR/WD-xslt.

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

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

[30] U. Zwick and M. Thorup. Compact routing schemes. manuscript, February 2001.