The Tree Inclusion Problem: In Optimal Space and Faster

Philip Bille* and Inge Li Gørtz

The IT University of Copenhagen, Department of Theoretical Computer Science, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark. {beetle,inge}@itu.dk

Abstract. Given two rooted, ordered, and labeled trees P and T the tree inclusion problem is to determine if P can be obtained from T by deleting nodes in T. This problem has recently been recognized as an important query primitive in XML databases. Kilpeläinen and Mannila (SIAM J. of Comp. 1995) presented the first polynomial time algorithm using quadratic time and space. Since then several improved results have been obtained for special cases when P and T have a small number of leaves or small depth. However, in the worst case these algorithms still use quadratic time and space. In this paper we present a new approach to the problem which leads to a new algorithm which uses optimal linear space and has subquadratic running time. Our algorithm improves all previous time and space bounds. Most importantly, the space is improved by a linear factor. This will make it possible to query larger XML databases and speed up the query time since more of the computation can be kept in main memory.

1 Introduction

Let T be a rooted tree. We say that T is *labeled* if each node is a assigned a symbol from an alphabet Σ and we say that T is *ordered* if a left-to-right order among siblings in T is given. All trees in this paper are rooted, ordered, and labeled. A tree P is *included* in T, denoted $P \sqsubseteq T$, if P can be obtained from T by deleting nodes of T. Deleting a node v in T means making the children of v children of the parent of v and then removing v. The children are inserted in the place of v in the left-to-right order among the siblings of v. The *tree inclusion problem* is to determine if P can be included in T and if so report all subtrees of T that include P.

Recently, the problem has been recognized as an important query primitive for XML data and has received considerable attention, see e.g., [15, 16, 18, 17]. The key idea is that an XML document can be viewed as an ordered, labeled tree and queries on this tree correspond to a tree inclusion problem. As an example consider Fig. 1. Suppose that we want to maintain a catalog of books for a bookstore. A fragment of the tree, denoted D, corresponding to the catalog

* This work is part of the DSSCV project supported by the IST Programme of the European Union (IST-2001-35443).

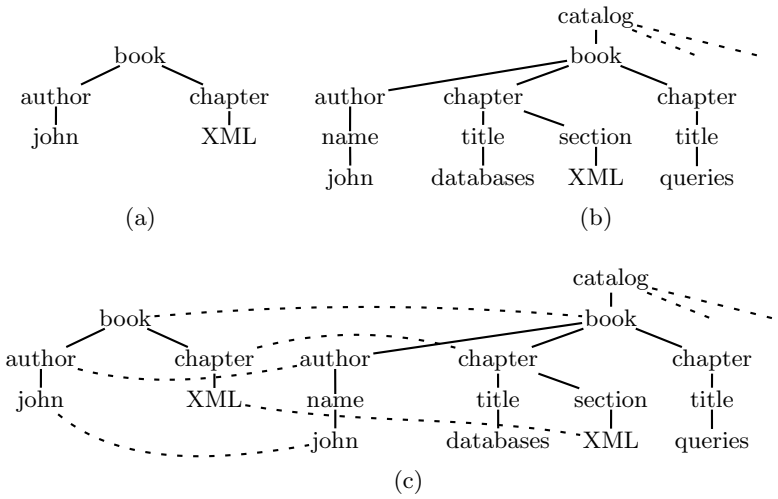

Fig. 1. Can tree (a) be included in tree (b)? Yes. The embedding is given in (c).

is shown in (b). In addition to supporting full-text queries, such as find all documents containing the word "John", we can also utilize the tree structure of the catalog to ask more specific queries, such as "find all books written by John with a chapter that has something to do with XML". We can model this query by constructing the tree, denoted Q, shown in (a) and solve the tree inclusion problem: is $Q \sqsubseteq D$? The answer is yes and a possible way to include Q in D is indicated by the dashed lines in (c). If we delete all the nodes in D not touched by dashed lines the trees Q and D become isomorphic. Such a mapping of the nodes from Q to D given by the dashed lines is called an *embedding* (formally defined in Sec. 3).

The tree inclusion problem was initially introduced by Knuth [11, exercise 2.3.2-22] who gave a sufficient condition for testing inclusion. Motivated by applications in structured databases [9, 12] Kilpeläinen and Mannila [10] presented the first polynomial time algorithm using $O(n_P n_T)$ time and space, where n_P and n_T is the number of nodes in a tree P and T, respectively. During the last decade several improvements of the original algorithm of [10] have been suggested [8, 1, 14, 4]. The previously best known bound is due to Chen [4] who presented an algorithm using $O(l_P n_T)$ time and $O(l_P \min\{d_T, l_T\})$ space. Here, l_S and d_S denotes the number of leaves of and the maximum depth of a tree S, respectively. This algorithm is based on an algorithm of Kilpeläinen [8]. Note that the time and space is still $\Theta(n_P n_T)$ for worst-case input trees.

In this paper we improve all of the previously known time and space bounds. Combining the three algorithms presented in this paper we have:

Theorem 1. *For trees T and P the tree inclusion problem can be solved in $O(\min(\frac{n_P n_T}{\log n_T}, l_P n_T, n_P l_T \log\log n_T))$ time using optimal $O(n_T + n_P)$ space.*

Hence, for worst-case input this improves the previous time and space bounds by a logarithmic and linear factor, respectively. When P has a small number of leaves the running time of our algorithm matches the previously best known time bound of [4] while maintaining linear space. In the context of XML databases the most important feature of our algorithms is the space usage. This will make it possible to query larger trees and speed up the query time since more of the computation can be kept in main memory.

Techniques Most of the previous algorithms, including the best one [4], are essentially based on a simple dynamic programming approach from the original algorithm of [10]. The main idea behind this algorithm is following: Let $v \in V(P)$ and $w \in V(T)$ be nodes with children $v_1, \ldots, v_i$ and $w_1, \ldots, w_j$, respectively. To decide if $P(v)$ can be included $T(w)$ we try to find a sequence of numbers $1 \leq x_1 < x_2 < \cdots < x_i \leq j$ such that $P(v_k)$ can be included in $T(w_{x_k})$ for all k, $1 \leq k \leq i$. If we have already determined whether or not $P(v_s) \sqsubseteq T(w_t)$, for all s and t, $1 \leq s \leq i$, $1 \leq t \leq j$, we can efficiently find such a sequence by scanning the children of v from left to right. Hence, applying this approach in a bottom-up fashion we can determine, if $P(v) \sqsubseteq T(w)$, for all pairs $(v, w) \in V(P) \times V(T)$.

In this paper we take a significantly different approach. The main idea is to construct a data structure on T supporting a small number of procedures, called the *set procedures*, on subsets of nodes of T. We show that any such data structure implies an algorithm for the tree inclusion problem. We consider various implementations of this data structure which all use linear space. The first simple implementation gives an algorithm with $O(l_P n_T)$ running time. As it turns out, the running time depends on a well-studied problem known as the *tree color problem*. We show a general connection between data structures for the tree color problem and the tree inclusion problem. Plugging in a data structure of Dietz [5] we obtain an algorithm with $O(n_P l_T \log\log n_T)$ running time.

Based on the simple algorithms above we show how to improve the worst-case running time of the set procedures by a logarithmic factor. The general idea used to achieve this is to divide T into small trees or forests, called *micro trees* or *clusters* of logarithmic size which overlap with other micro trees in at most 2 nodes. Each micro tree is represented by a constant number of nodes in a *macro tree*. The nodes in the macro tree are then connected according to the overlap of the micro trees they represent. We can efficiently preprocess the micro trees and the macro tree such that the set procedures use constant time for each micro tree. Hence, the worst-case running time is improved by a logarithmic factor to $O(\frac{n_P n_T}{\log n_T})$.

Our results rely on a standard RAM model of computation with word size $\Theta(\log n)$. We use a standard instruction set such as bitwise boolean operations, shifts, and addition. Most of the proofs are omitted due to lack of space. They can be found in the full version of the paper [3].

2 Notation and Definitions

In this section we define the notation and definitions we will use throughout the paper. For a graph G we denote the set of nodes and edges by $V(G)$ and $E(G)$, respectively. Let T be a rooted tree. The root of T is denoted by $\text{root}(T)$. The *size* of T, denoted by n_T, is $|V(T)|$. The *depth* of a node $v \in V(T)$, $\text{depth}(v)$, is the number of edges on the path from v to $\text{root}(T)$ and the depth of T, denoted d_T, is the maximum depth of any node in T. The set of children of a node v is denoted $\text{child}(v)$. A node with no children is a leaf and otherwise an internal node. The set of leaves of T is denoted $L(T)$ and we define $l_T = |L(T)|$. We say that T is *labeled* if each node v is a assigned a symbol, denoted $\text{label}(v)$, from an alphabet Σ and we say that T is *ordered* if a left-to-right order among siblings in T is given. All trees in this paper are rooted, ordered, and labeled.

Let $T(v)$ denote the subtree of T rooted at a node $v \in V(T)$. If $w \in V(T(v))$ then v is an ancestor of w, denoted $v \preceq w$, and if $w \in V(T(v))\backslash\{v\}$ then v is a proper ancestor of w, denoted $v \prec w$. If v is a (proper) ancestor of w then w is a (proper) descendant of v. A node z is a common ancestor of v and w if it is an ancestor of both v and w. The nearest common ancestor of v and w, $\text{nca}(v, w)$, is the common ancestor of v and w of largest depth. The *first ancestor of w labeled α*, denoted $\text{fl}(w, \alpha)$, is the node v such that $v \preceq w$, $\text{label}(v) = \alpha$, and no node on the path between v and w is labeled α. If no such node exists then $\text{fl}(w, \alpha) = \perp$, where $\perp \notin V(T)$ is a special *null node*.

For any set of pairs U, let $U|_1$ and $U|_2$ denote the *projection* of U to the first and second coordinate, that is, if $(u_1, u_2) \in U$ then $u_1 \in U|_1$ and $u_2 \in U|_2$.

Lists A *list*, X, is a finite sequence of objects $X = [v_1, \ldots, v_k]$. The *length* of the list, denoted $|X|$, is the number of objects in X. The ith element of X, $X[i]$, $1 \leq i \leq |X|$ is the object v_i and $v \in X$ iff $v = X[j]$ for some $1 \leq j \leq |X|$. For any two lists $X = [v_1, \ldots, v_k]$ and $Y = [w_1, \ldots, w_k]$, the list obtained by *appending* Y to X is the list $X \circ Y = [v_1, \ldots, v_k, w_1, \ldots, w_k]$. We extend this notation such that for any object u, $X \circ u$ denotes the list $X \circ [u]$. For simplicity in the notation we will sometimes write $[v_i \mid 1 \leq i \leq k]$ to denote the list $[v_1, \ldots, v_k]$. A *pair list* is a list of pairs of object $Y = [(v_1, w_1), \ldots, (v_k, w_k)]$. Here the first and second element in the pair is denoted by $Y[i]_1 = v_i$ and $Y[i]_2 = w_i$. The projection of pair lists is defined by $Y|_1 = [v_1, \ldots, v_k]$ and $Y|_2 = [w_1, \ldots, w_k]$.

Orderings Let T be a tree with root v and let $v_1, \ldots, v_k$ be the children of v from left-to-right. The *preorder traversal* of T is obtained by visiting v and then recursively visiting $T(v_i)$, $1 \leq i \leq k$, in order. Similarly, the *postorder traversal* is obtained by first visiting $T(v_i)$, $1 \leq i \leq k$, and then v. The *preorder number* and *postorder number* of a node $w \in T(v)$, denoted by $\text{pre}(w)$ and $\text{post}(w)$, is the number of nodes preceding w in the preorder and postorder traversal of T, respectively. The nodes to the *left* of w in T is the set of nodes $u \in V(T)$ such that $\text{pre}(u) < \text{pre}(w)$ and $\text{post}(u) < \text{post}(w)$. If u is to the left of w, denoted by $u \lhd w$, then w is to the *right* of u. If $u \lhd w$, $u \preceq w$, or $w \prec u$ we write $u \trianglelefteq w$. The null node $\perp$ is not in the ordering, i.e., $\perp \not\trianglelefteq v$ for all nodes v.

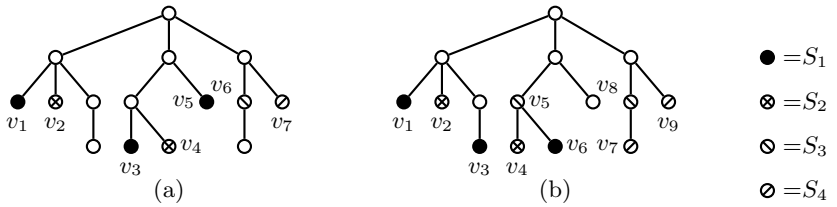

Fig. 2. In (a) we have $\text{mop}(S_1, S_2, S_1, S_3, S_4) = \{(v_3, v_7)\}$ and in (b) we have $\text{mop}(S_1, S_2, S_1, S_3, S_4) = \{(v_1, v_7), (v_3, v_9)\}$.

Deep Sets A set of nodes $V \subseteq V(T)$ is *deep* iff no node in V is a proper ancestor of another node in V.

Minimum Ordered Pair For deep sets of nodes $V_1, \ldots, V_k$ let $\Phi(V_1, \ldots, V_k) \subseteq (V_1 \times \cdots \times V_k)$, be the set such that $(v_1, \ldots, v_k) \in \Phi(V_1, \ldots, V_k)$ iff $v_1 \lhd \cdots \lhd v_k$. If $(v_1, \ldots, v_k) \in \Phi(V_1, \ldots, V_k)$ and there is no $(v'_1, \ldots, v'_k) \in \Phi(V_1, \ldots, V_k)$, where either $v_1 \lhd v'_1 \lhd v'_k \trianglelefteq v_k$ or $v_1 \trianglelefteq v'_1 \lhd v'_k \lhd v_k$ then the pair (v_1, v_k) is a *minimum ordered pair*. The set of minimum ordered pairs for $V_1, \ldots, V_k$ is denoted by $\text{mop}(V_1, \ldots, V_k)$. Fig. 2 illustrates mop on a small example. The following lemma shows that we can compute $\text{mop}(V_1, \ldots, V_k)$ iteratively by first computing $\text{mop}(V_1, V_2)$ and then $\text{mop}(\text{mop}(V_1, V_2)|_2, V_3)$ and so on.

Lemma 1. *For any deep sets of nodes $V_1, \ldots, V_k$:$(v_1, v_k) \in \text{mop}(V_1, \ldots, V_k)$ iff there exists a v_{k-1} such that $(v_1, v_{k-1}) \in \text{mop}(V_1, \ldots, V_{k-1})$ and $(v_{k-1}, v_k) \in \text{mop}(\text{mop}(V_1, \ldots, V_{k-1})|_2, V_k)$.*

3 Computing Deep Embeddings

In this section we present a general framework for answering tree inclusion queries. As in [10] we solve the equivalent *tree embedding problem*. Let P and T be rooted labeled trees. An *embedding* of P in T is an injective function $f : V(P) \to V(T)$ such that for all nodes $v, u \in V(P)$,

(i) $\text{label}(v) = \text{label}(f(v))$. (label preservation condition)
(ii) $v \prec u$ iff $f(v) \prec f(u)$. (ancestor condition)
(iii) $v \lhd u$ iff $f(v) \lhd f(u)$. (order condition)

Lemma 2 ([10]). *For any trees P and T, $P \sqsubseteq T$ iff there exists an embedding of P in T.*

An example of an embedding is given in Fig. 1(c). We say that the embedding f is *deep* if there is no embedding g such that $f(\text{root}(P)) \prec g(\text{root}(P))$. The *deep occurrences* of P in T, denoted $\text{emb}(P, T)$ is the set of nodes,

$$\text{emb}(P, T) = \{f(\text{root}(P)) \mid f \text{ is a deep embedding of } P \text{ in } T\}.$$

Note that $\mathrm{emb}(P,T)$ must be a deep set in T. Furthermore, by definition the set of ancestors of nodes in $\mathrm{emb}(P,T)$ is the set of subtrees $T(u)$ such that $P \sqsubseteq T(u)$. Hence, to solve the tree inclusion problem it is sufficient to compute $\mathrm{emb}(P,T)$ and then, using additional $O(n_T)$ time, report all ancestors (if any) of this set.

The key idea in our algorithm for computing deep embeddings is to construct a data structure that allows a fast implementation of the following procedures, called the *set procedures*. For all $V \subseteq V(T)$, $U \subseteq V(T) \times V(T)$, $\alpha \in \Sigma$ define:

$\textsc{Parent}_T(V)$. Return the set $R := \{\mathrm{parent}(v) \mid v \in V\}$.
$\textsc{Nca}_T(U)$. Return the set $R := \{\mathrm{nca}(u_1,u_2) \mid (u_1,u_2) \in U\}$.
$\textsc{Deep}_T(V)$. Return the set $R := \{v \in V \mid \nexists w \in V \text{ such that } v \prec w\}$.
$\textsc{Mop}_T(U,V)$. Return the set of pairs R such that for any pair $(u_1,u_2) \in U$, $(u_1,v) \in R$ iff $(u_2,v) \in \mathrm{mop}(U|_2, V)$.
$\textsc{Fl}_T(V,\alpha)$. Return the set $R := \{\mathrm{fl}(v,\alpha) \mid v \in V\}$.

With the set procedures we can compute deep embeddings. The following procedure $\textsc{Emb}_T(v)$, $v \in V(P)$, recursively computes the set of deep occurrences of $P(v)$ in T. Fig. 3 illustrates how Emb works on a small example.

$\textsc{Emb}_T(v)$ Let $v_1,\ldots,v_k$ be the sequence of children of v ordered from left to right. There are three cases:

1. $k = 0$ (v is a leaf). Set $R := \textsc{Deep}_T(\textsc{Fl}_T(L(T),\mathrm{label}(v)))$.
2. $k = 1$. Recursively compute $R_1 := \textsc{Emb}_T(v_1)$.
   Set $R := \textsc{Deep}_T(\textsc{Fl}_T(\textsc{Deep}_T(\textsc{Parent}_T(R_1)),\mathrm{label}(v)))$.
3. $k > 1$. Compute $R_1 := \textsc{Emb}_T(v_1)$ and $U_1 := \{(r,r) \mid r \in R_1\}$. For i, $1 \le i \le k$, compute $R_i := \textsc{Emb}_T(v_i)$ and $U_i := \textsc{Mop}_T(U_{i-1}, R_i)$.
   Finally, compute $R := \textsc{Deep}_T(\textsc{Fl}_T(\textsc{Deep}_T(\textsc{Nca}_T(U_k)),\mathrm{label}(v)))$.

If $R = \emptyset$ stop and report that there is no deep embedding of $P(v)$ in T. Otherwise return R.

Lemma 3. *For any two trees T and P, $\textsc{Emb}_T(v)$ computes the set of deep occurrences of $P(v)$ in T.*

Proof. By induction on the size of the subtree $P(v)$. If v is a leaf we immediately have $\mathrm{emb}(v,T) = \textsc{Deep}_T(\textsc{Fl}_T(L(T),\mathrm{label}(v)))$. Suppose that v is an internal node with children $v_1,\ldots,v_k$, $k \ge 1$. We show that $\mathrm{emb}(P(v),T) = \textsc{Emb}_T(v)$.

If $k = 1$, $w \in \textsc{Emb}_T(v)$ implies $\mathrm{label}(w) = \mathrm{label}(v)$ and there is a node $w_1 \in \textsc{Emb}_T(v_1)$ such that $\mathrm{fl}(\mathrm{parent}(w_1),\mathrm{label}(v)) = w$, i.e., no node on the path between w_1 and w is labeled $\mathrm{label}(v)$. By induction $\textsc{Emb}_T(v_1) = \mathrm{emb}(P(v_1),T)$ and thus w is the root of an embedding of $P(v)$ in T. Since $\textsc{Emb}_T(v)$ is the deep set of all such nodes we have $w \in \mathrm{emb}(P(v),T)$. Conversely, if $w \in \mathrm{emb}(P(v),T)$ then $\mathrm{label}(w) = \mathrm{label}(v)$, there is a node $w_1 \in \mathrm{emb}(P(v_1),T)$ such that $w \prec w_1$, and no node on the path between w and w_1 is labeled $\mathrm{label}(v)$, that is, $\mathrm{fl}(w_1,\mathrm{label}(v)) = w$. Hence, $w \in \textsc{Emb}_T(v)$.

Before considering case 3 we show that $U_j = \mathrm{mop}(\textsc{Emb}_T(v_1),\ldots,\textsc{Emb}_T(v_j))$ by induction on j, $2 \le j \le k$. For $j = 2$ it follows from the definition of $\textsc{Mop}_T$ that $U_2 = \mathrm{mop}(\textsc{Emb}_T(v_1),\textsc{Emb}_T(v_2))$. Hence, assume that $j > 2$. We have

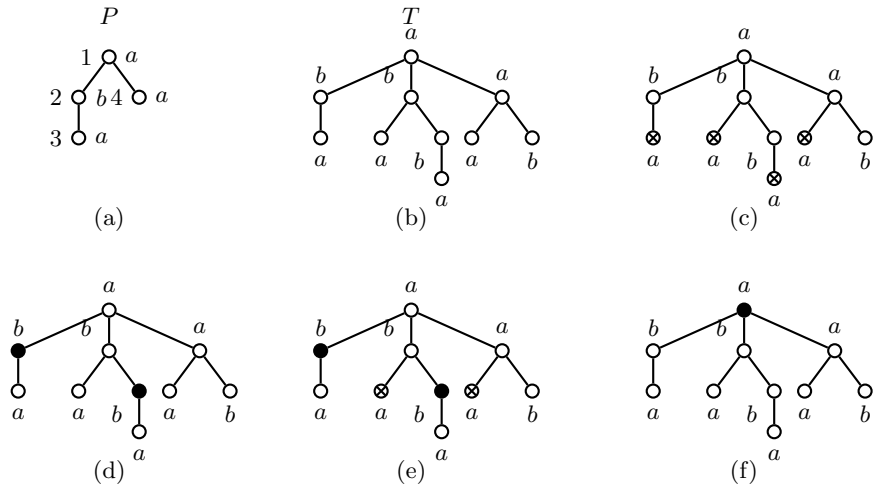

Fig. 3. Computing the deep occurrences of P into T depicted in (a) and (b) respectively. The nodes in P are numbered 1–4 for easy reference. (c) Case 1 of Emb. Since 3 and 4 are leaves and label(3) = label(4) we have $\textsc{Emb}_T(3) = \textsc{Emb}_T(4)$. (d) Case 2 of Emb. Note that the middle child of the root(T) is not in the set since it is not a deep occurrence. (e) Case 3 of Emb: The two minimal ordered pairs of the sets of (d) and (c). (f) nca of the pairs in (e) both give the root node of T which is the only (deep) occurrence of P.

$U_j = \textsc{Mop}_T(U_{j-1}, \textsc{Emb}_T(v_j)) = \textsc{Mop}_T(\mathrm{mop}(\textsc{Emb}_T(v_1), \ldots, \textsc{Emb}_T(v_{j-1})), R_j)$. By definition of $\textsc{Mop}_T$, U_j is the set of pairs such that for any pair $(r_1, r_{j-1}) \in \mathrm{mop}(\textsc{Emb}_T(v_1), \ldots, \textsc{Emb}_T(v_{j-1}))$, we have $(r_1, r_j) \in U_j$ if and only if $(r_{j-1}, r_j) \in \mathrm{mop}(\mathrm{mop}(\textsc{Emb}_T(v_1), \ldots, \textsc{Emb}_T(v_{j-1}))|_2, R_j)$. It now follows from Lemma 1 that $(r_1, r_j) \in U_j$ iff $(r_1, r_j) \in \mathrm{mop}(\textsc{Emb}_T(v_1), \ldots, \textsc{Emb}_T(v_j))$.

Consider case 3. If $k > 1$, $w \in \textsc{Emb}_T(v)$ implies $\mathrm{label}(w) = \mathrm{label}(v)$ and there are nodes $(w_1, w_k) \in \mathrm{mop}(\mathrm{emb}(P(v_1), T), \ldots, \mathrm{emb}(P(v_k), T))$ such that $w = \mathrm{fl}(\mathrm{nca}(w_1, w_k), \mathrm{label}(v))$. Clearly, w is the root of an embedding of $P(v)$ in T. Assume for contradiction that w is not a deep embedding, i.e., $w \prec u$ for some node $u \in \mathrm{emb}(P(v), T)$. Since $w = \mathrm{fl}(\mathrm{nca}(w_1, w_k), \mathrm{label}(v))$ there must be nodes $u_1 \lhd \cdots \lhd u_k$, such that $u_i \in \mathrm{emb}(P(v_i), T)$, $1 \leq i \leq k$, and $u = \mathrm{fl}(\mathrm{nca}(u_1, u_k), \mathrm{label}(v))$. However, this contradicts the fact that $(w_1, w_k) \in \mathrm{mop}(\mathrm{emb}(P(v_1), T), \ldots, \mathrm{emb}(P(v_k), T))$. If $w \in \mathrm{emb}(P(v), T)$ a similar argument implies that $w \in \textsc{Emb}_T(v)$. □

When the tree T is clear from the context we may not write the subscript T in the procedure names. Note that since the $\textsc{Emb}_T(v)$ is a deep set we can assume that Parent, Fl, Nca, and Mop take deep sets as input.

4 A Simple Tree Inclusion Algorithm

In this section we a present a simple implementation of the set procedures which leads to an efficient tree inclusion algorithm. Subsequently, we modify one of the procedures to obtain a family of tree inclusion algorithms where the complexities depend on the solution to a well-studied problem known as the *tree color problem*.

Preprocessing To compute deep embeddings efficiently we require a data structure for T which allows us, for any $v, w \in V(T)$, to compute $\text{nca}_T(v, w)$ and determine if $v \prec w$ or $v \lhd w$. In linear time we can compute $\text{pre}(v)$ and $\text{post}(v)$ for all nodes $v \in V(T)$, and with these it is straightforward to test the two conditions. Using a data structure by Harel and Tarjan [7] we can answer nearest common ancestor queries in $O(1)$ time using $O(n_T)$ space and preprocessing time. Hence, our data structure uses linear preprocessing time and space.

Implementation of the Set Procedures To answer tree inclusion queries we give an efficient implementation of the set procedures. The idea is to represent the node sets in a left-to-right order. For this purpose we introduce some helpful notation. A *node list*, X, is a list of nodes. If $v_i \lhd v_{i+1}$, $1 \leq i < |X|$ then X is *ordered* and if $v_1 \trianglelefteq v_{i+1}$, $1 \leq i < |X|$ then X is *semiordered*. A *node pair list*, Y, is a list of pairs of nodes. We say that Y is ordered if $Y|_1$ and $Y|_2$ are ordered, and semiordered if $Y|_1$ and $Y|_2$ are semiordered.

The set procedures are implemented using node lists and node pair lists. All lists used in the procedures are either ordered or semiordered. As noted in Sec. 3 we may assume that the input to all of the procedures, except Deep, represent a deep set, that is, the corresponding node list or node pair list is ordered. We assume that the input list given to Deep is semiordered. Hence, the output of all the other set procedures must be semiordered.

$\textsc{Parent}_T(X)$. Return the list $Z := [\text{parent}(X[i]) \mid 1 \leq i \leq |X|]$.

$\textsc{Nca}(Y)$. Return the list $Z := [\text{nca}(Y[i]) \mid 1 \leq i \leq |Y|]$.

$\textsc{Deep}_T(X)$. Initially, set $v := X[1]$ and $Z := []$. For each i, $2 \leq i \leq k$, compare v and $X[i]$: If $v \lhd X[i]$ set $Z := Z \circ v$ and $v := X[i]$. If $v \prec X[i]$, set $v := X[i]$ and otherwise $(X[i] \prec v)$ do nothing. Finally, set $Z := Z \circ v$ and return Z.

$\textsc{Mop}_T(X, Y)$. Initially, set $Z := []$. Find the minimum j such that $X[1]_2 \lhd Y[j]$ and set $x := X[1]_1$, $y := Y[j]$, and $h := j$. If no such j exists, stop.
As long as $h \leq |Y|$ do the following: For each i, $2 \leq i \leq |X|$, do: Set $h := h{+}1$ until $X[i]_2 \lhd Y[h]$. Compare $Y[h]$ and y: If $y = Y[h]$ set $x := X[i]_1$. If $y \lhd Y[h]$ set $Z := Z \circ (x, y)$, $x := X[i]_1$, and $y := Y[h]$. Finally, set $Z := Z \circ (x, y)$ and return Z.

$\textsc{Fl}_T(X, \alpha)$. Initially, set $Y := X$, $Z := []$, and $S := []$. Repeat until $Y := []$: For $i = 1, \ldots, |Y|$ if $\text{label}(Y[i]) = \alpha$ set $Z := \textsc{Insert}(Y[i], Z)$ and otherwise set $S := S \circ \text{parent}(Y[i])$. Set $S := \textsc{Deep}_T(S)$, $Y := \textsc{Deep}^*_T(S, Z)$, $S := []$. Return Z.

Procedure Fl calls two auxiliary procedures: Insert(v, Z) takes an ordered list Z and insert the node v such that the resulting list is ordered, and Deep$^*(S, Z)$ takes two ordered lists and returns the ordered list representing the set Deep$(S \cup Z) \cap S$, i.e., Deep$^*(S, Z) = [s \in S | \nexists z \in Z : s \prec z]$. Below we describe the implementation of Fl in more detail.

We use one doubly linked list to represent all the lists Y, S, and Z. For each element in Y we have pointers Pred and Succ pointing to the predecessor and successor in the list, respectively. We also have at each element a pointer Next pointing to the next element in Y. In the beginning Next = Succ for all elements, since all elements in the list are in Y. When going through Y in one iteration we simple follow the Next pointers. When Fl calls Insert$(Y[i], Z)$ we set Next(Pred$(Y[i])$) to Next$(Y[i])$. That is, all nodes in the list not in Y, i.e., nodes not having a Next pointer pointing to them, are in Z. We do not explicitly maintain S. Instead we just save Parent$(Y[i])$ at the position in the list instead of $Y[i]$. Now Deep(S) can be performed following the Next pointers and removing elements from the doubly linked list accordingly to procedure Deep. It remains to show how to calculate Deep$^*(S, Z)$. This can be done by running through S following the Next pointers. At each node s compare Pred(s) and Succ(s) with s. If one of them is a descendant of s then remove s from the doubly linked list.

Using this linked list implementation Deep$^*(S, Z)$ takes time $O(|S|)$, whereas using Deep to calculate this would have used time $O(|S| + |Z|)$.

Complexity of the algorithm For the running time of the node list implementation observe that, given the data structure described above, all set procedures, except Fl, perform a single pass over the input using constant time at each step. Hence we have,

Lemma 4. *For any tree T there is a data structure using $O(n_T)$ space and preprocessing which supports each of the procedures* Parent, Deep, Mop, *and* Nca *in linear time (in the size of their input).*

The running time of a single call to Fl might take time $O(n_T)$. Instead we will divide the calls to Fl into groups and analyze the total time used on such a group of calls. The intuition behind the division is that for a path in P the calls made to Fl by Emb is done bottom up on disjoint lists of node in T.

Lemma 5. *For disjoint ordered node lists $V_1, \ldots, V_k$ and labels $\alpha_1, \ldots, \alpha_k$, such that any node in V_{i+1} is an ancestor of some node in* Deep$(\text{Fl}_T(V_i, \alpha_i))$, $2 \leq i < k$, *all of $\text{Fl}_T(V_1, \alpha_1), \ldots, \text{Fl}_T(V_k, \alpha_k)$ can be computed in $O(n_T)$ time.*

The proof is omitted due to lack of space. The basic idea in the proof is to show that any node in T can be in Y at most twice during all calls to Fl.

Using the node list implementation of the set procedures we get:

Theorem 2. *For trees P and T the tree inclusion problem can be solved in $O(l_P n_T)$ time and $O(n_P + n_T)$ space.*

Proof. By Lemma 4 we can preprocess T in $O(n_T)$ time and space. Let $g(n)$ denote the time used by FL on a list of length n. Consider the time used by $\text{EMB}_T(\text{root}(P))$. We bound the contribution for each node $v \in V(P)$. From Lemma 4 it follows that if v is a leaf the cost of v is at most $O(g(l_T))$. Hence, by Lemma 5, the total cost of all leaves is $O(l_P g(l_T)) = O(l_P n_T)$. If v has a single child w the cost is $O(g(|\text{EMB}_T(w)|))$. If v has more than one child the cost of MOP,NCA, and DEEP is bounded by $\sum_{w \in \text{child}(v)} O(|\text{EMB}_T(w)|)$. Furthermore, since the length of the output of MOP (and thus NCA) is at most $z = \min_{w \in \text{child}(v)} |\text{EMB}_T(w)|$ the cost of FL is $O(g(z))$. Hence, the total cost for internal nodes is,

$$\sum_{v \in V(P) \setminus L(P)} O(g(\min_{w \in \text{child}(v)} |\text{EMB}_T(w)|) + \sum_{w \in \text{child}(v)} |\text{EMB}_T(w)|) \leq \sum_{v \in V(P)} O(g(|\text{EMB}_T(v)|)).$$

Next we bound the sum $\sum_{v \in V(P)} O(g(|\text{EMB}_T(v)|))$. For any $w \in \text{child}(v)$ we have that $\text{EMB}_T(w)$ and $\text{EMB}_T(v)$ are disjoint ordered lists. Furthermore we have that any node in $\text{EMB}_T(v)$ must be an ancestor of some node in $\text{DEEP}_T(\text{FL}_T(\text{EMB}_T(w), \text{label}(v)))$. Hence, by Lemma 5, for any leaf to root path $\delta = v_1, \ldots, v_k$ in P, we have that $\sum_{u \in \delta} g(|\text{EMB}_T(u)|) \leq O(n_T)$. Let Δ denote the set of all root to leaf paths in P. It follows that, $\sum_{v \in V(T)} g(|\text{EMB}_T(v)|) \leq \sum_{p \in \Delta} \sum_{u \in p} g(|\text{EMB}_T(u)|) \leq O(l_P n_T)$.

Since this time dominates the time spent at the leaves the time bound follows. Next consider the space used by $\text{EMB}_T(\text{root}(P))$. The preprocessing of described above uses only $O(n_T)$ space. Furthermore, by induction on the size of the subtree $P(v)$ it follows immediately that at each step in the algorithm at most $O(\max_{v \in V(P)} |\text{EMB}_T(v)|)$ space is needed. Since $\text{EMB}_T(v)$ a deep embedding, it follows that $|\text{EMB}_T(v)| \leq l_T$. □

An Alternative Algorithm In this section we present an alternative algorithm. Since the time complexity of the algorithm in the previous section is dominated by the time used by FL, we present an implementation of this procedure which leads to a different complexity. Define a *firstlabel data structure* as a data structure supporting queries of the form $\text{fl}(v, \alpha)$, $v \in V(T)$, $\alpha \in \Sigma$. Maintaining such a data structure is known as the *tree color problem*, see e.g., [5, 13]. With such a data structure available we can compute $\text{FL}(X, \alpha)$ as the list $[\text{fl}(X[i], \alpha) \mid 1 \leq i \leq |X|]$.

Theorem 3. *Let P and T be trees. Given a firstlabel data structure using $s(n_T)$ space, $p(n_T)$ preprocessing time, and $q(n_T)$ time for queries, the tree inclusion problem can be solved in $O(p(n_T) + n_P l_T \cdot q(n_T))$ time and $O(n_P + s(n_T) + n_T)$ space.*

Proof. Constructing the firstlabel data structures uses $O(s(n_T))$ space and time $O(p(n_T))$. As in the proof of Thm. 2 we have that the total time used by $\text{EMB}_T(\text{root}(P))$ is bounded by $\sum_{v \in V(P)} g(|\text{EMB}_T(v)|)$, where $g(n)$ is the time used by FL on a list of length n. Since $\text{EMB}_T(v)$ is a deep embedding and each

fl takes $q(n_T)$ we have, $\sum_{v \in V(P)} g(|\text{EMB}_T(v)|) \leq \sum_{v \in V(P)} g(l_T) = n_P l_T \cdot q(n_T)$. □

Several firstlabel data structures are available, for instance, if we want to maintain linear space, we can use a data structure by Dietz [5] that supports firstlabel queries in $O(\log \log n_T)$ time using $O(n_T)$ space and $O(n_T)$ expected preprocessing time. Plugging in this data structure we obtain,

Corollary 1. *For trees P and T the tree inclusion problem can be solved in $O(n_P l_T \log \log n_T)$ time and $O(n_P + n_T)$ space.*

Since the preprocessing time $p(n)$ of the firstlabel data structure is expected the running time of the tree inclusion algorithm is also expected. However, the expectation is due to a dictionary using perfect hashing and we can therefore use the deterministic dictionary of [6] with $O(n_T \log n_T)$ worst-case preprocessing time instead. This does not affect the overall complexity of the algorithm.

5 A Faster Tree Inclusion Algorithm

In this section we present a new tree inclusion algorithm which has a worst-case subquadratic running time. Due to lack of space we will only give a rough sketch of the algorithm. A full description of the algorithm can be found in the full version of the paper [3].

The first step is to divide T into small connected subgraphs, called *micro trees* or *clusters*. Using a technique from [2] we can construct in linear time a *cluster partition* of T, consisting of $O(n_T / \log n_T)$ clusters each of size $O(\log n_T)$, with the property that any cluster shares at most two nodes with any other cluster. Each micro tree is represented by a constant number of nodes in a *macro tree*. The nodes in the macro tree are then connected according to the overlap of the micro trees they represent. Note that the total number of nodes in the macro tree is $O(n_T / \log n_T)$.

In linear time of the tree T we preprocess all the micro trees and the macro tree such that the set procedures use constant time for each micro tree. Using a compact node representation we can then implement all the set procedures in $O(n_T / \log n_T)$ time.

Lemma 6. *For any tree T there is a data structure using $O(n_T)$ space and $O(n_T)$ expected preprocessing time which supports all of the set procedures in $O(n_T / \log n_T)$ time.*

The proof of the lemma and all details in the implementation of the set procedures can be found in the full version of the paper. We can now compute the deep occurrences of P in T using the procedure EMB of Sec. 3 and Lemma 6. Since each node $v \in V(P)$ contributes at most a constant number of calls to set procedures it follows that,

Theorem 4. *For trees P and T the tree inclusion problem can be solved in $O(\frac{n_P n_T}{\log n_T})$ time and $O(n_P + n_T)$ space.*

Combining the results in Theorems 2, 4 and Corollary 1 we have the main result of Theorem 1.

Acknowledgments We thank the reviewers for the many insightful comments.

References

1. L. Alonso and R. Schott. On the tree inclusion problem. In *Proc. of Math. Foundations of Computer Science*, pages 211–221, 1993.
2. S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. of Intl. Coll. on Automata, Languages and Programming (ICALP)*, pages 270–280, 1997.
3. P. Bille and I. Gørtz. The tree inclusion problem: In optimal space and faster. Technical Report TR-2005-54, IT University of Copenhagen, January 2005.
4. W. Chen. More efficient algorithm for ordered tree inclusion. *J. Algorithms*, 26:370–385, 1998.
5. P. F. Dietz. Fully persistent arrays. In *Proc. of Workshop on Algorithms and Data Structures (WADS)*, pages 67–74, 1989.
6. T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.
7. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
8. P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, University of Helsinki, Department of Computer Science, 1992.
9. P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proc. of Conf. on Research and Development in Information Retrieval*, pages 214–222, 1993.
10. P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. Comp.*, 24:340–356, 1995.
11. D. E. Knuth. *The Art of Computer Programming, Volume 1.* Addison Wesley, 1969.
12. H. Mannila and K. J. Räihä. On query languages for the p-string data model. *Information Modelling and Knowledge Bases*, pages 469–482, 1990.
13. S. Muthukrishnan and M. Müller. Time and space efficient method-lookup for object-oriented programs. In *Proc. of Symp. on Discrete Algorithms*, pages 42–51, 1996.
14. T. Richter. A new algorithm for the ordered tree inclusion problem. In *Proc. of Symp. on Combinatorial Pattern Matching (CPM)*, pages 150–166, 1997.
15. T. Schlieder and H. Meuss. Querying and ranking XML documents. *J. Am. Soc. Inf. Sci. Technol.*, 53(6):489–503, 2002.
16. T. Schlieder and F. Naumann. Approximate tree embedding for querying XML data. In *Proc. of Workshop On XML and Information Retrieval*, 2000.
17. H. Yang, L. Lee, and W. Hsu. Finding hot query patterns over an xquery stream. *The VLDB Journal*, 13(4):318–332, 2004.
18. L. H. Yang, M. L. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. In *Proc. of Conference on Very Large Databases (VLDB)*, pages 69–80, 2003.