The Tree Inclusion Problem: In Optimal Space and Faster

Philip Bille and Inge Li Gørtz

IT University Technical Report Series

TR-2005-54

ISSN 1600–6100

January 2005

ISSN 1600–6100

ISBN 87-7949-078-6

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone:	+45 72 18 50 00
Telefax:	+45 72 18 50 01
Web	www.itu.dk

The Tree Inclusion Problem: In Optimal Space and Faster

Philip Bille[†][*] Inge Li Gørtz[†]

January 11, 2005

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 use 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. The tree P and T is often called the *pattern* and *target*, respectively.

Recently, the problem has been recognized as an important query primitive for XML data and has received considerable attention, see *e.g.*, [28, 33, 32, 34, 27, 31]. 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 Figure 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 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 Section 3).

The tree inclusion problem was initially introduced by Knuth [21, exercise 2.3.2-22] who gave a sufficient condition for testing inclusion. Motivated by applications in structured databases [18, 23] Kilpeläinen and Mannila [19] 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

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

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

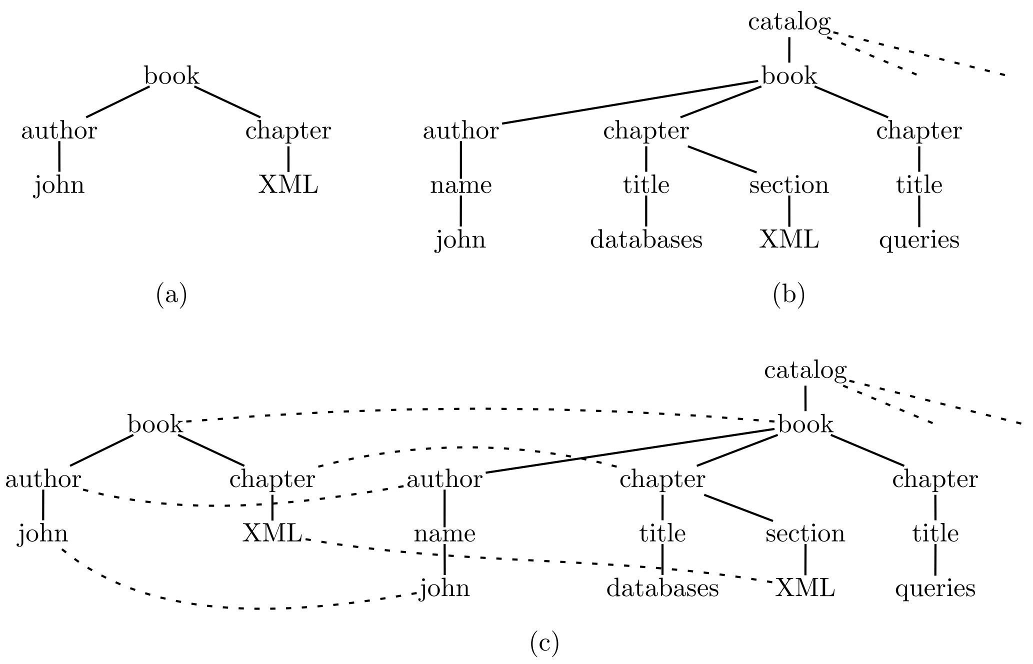

Figure 1: Can the tree (a) be included in the tree (b)? It can and the embedding is given in (c).

the original algorithm of [19] have been suggested [17, 1, 26, 7]. The previously best known bound is due to Chen [7] 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 [17]. 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 P and T 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 linear and logarithmic factor, respectively. When P has a small number of leaves the running time of our algorithm matches the previously best known time bound of [7] 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.

1.1 Techniques

Most of the previous algorithms, including the best one [7], are essentially based on a simple dynamic programming approach from the original algorithm of [19]. 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 a data structure for the tree color problem and the tree inclusion problem. Plugging in a data structure of Dietz [10] 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 worst-case running the 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 show how to 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 $\Omega(\log n)$. We use a standard instruction set such as bitwise boolean operations, shifts, and addition.

1.2 Related Work

For some applications considering *unordered* trees is more natural. However, in [24, 19] this problem was proved to be NP-complete. The tree inclusion problem is closely related to the *tree pattern matching problem* [16, 22, 11, 9]. The goal is here to find an injective mapping f from the nodes of P to the nodes of T such that for every node v in P the ith child of v is mapped to the ith child of $f(v)$. The tree pattern matching problem can be solved in $O(n \log^{O(1)} n)$ time, where $n = n_P + n_T$. Another similar problem is the *subtree isomorphism* problem [8, 29], which is to determine if T has a subgraph which is isomorphic to P. The subtree isomorphism problem can be solved efficiently for ordered and unordered trees. The best algorithms for this problem use $O(n_P^{1.5} n_T / \log n_P)$ for unordered trees and $O(n_P n_T / \log n_P)$ time ordered trees [8, 29]. Both use $O(n_P n_T)$ space. The tree inclusion problem can be considered a special case of the *tree edit distance problem* [30, 35, 20]. Here one wants to find the minimum sequence of insert, delete, and relabel operations needed to transform P into T. The currently best worst-case algorithm for this problem uses $O(n_P^2 n_T \log n_T)$ time. For more details and references see the survey [6].

1.3 Outline

In Section 2 we give notation and definitions used throughout the paper. In Section 3 a common framework for our tree inclusion algorithms is given. Section 4 present two simple algorithms and then, based on these result, we show how to get a faster algorithm in Section 5.

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 $\mathrm{root}(T)$. The *size* of T, denoted by n_T, is $|V(T)|$. The *depth* of a node $v \in V(T)$, $\mathrm{depth}(v)$, is the number of edges on the path from v to $\mathrm{root}(T)$ 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 $\mathrm{child}(v)$. 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 $\mathrm{label}(v)$, 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, $\mathrm{nca}(v, w)$, is the common ancestor of v and w of largest depth. The *first ancestor of w labeled α*, denoted $\mathrm{fl}(w, \alpha)$, is the node v such that $v \preceq w$, $\mathrm{label}(v) = \alpha$, and no node on the path between v and w is labeled α. If no such node exists then $\mathrm{fl}(w, \alpha) = \perp$, 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 $\mathrm{pre}(w)$ and $\mathrm{post}(w)$, 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 $\mathrm{pre}(u) < \mathrm{pre}(w)$ and $\mathrm{post}(u) < \mathrm{post}(w)$. 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 $u \unlhd w$. The null node $\perp$ is not in the ordering, i.e., $\perp \ntriangleleft v$ for all nodes v.

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 Let $V_1, \ldots, V_k$ be deep sets of nodes and 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 $v_1 \lhd v'_1 \lhd v'_k \unlhd v_k$ or $v_1 \unlhd v'_1 \lhd v'_k \lhd v_k$ 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 $\mathrm{mop}(V_1, \ldots, V_k)$. Figure 2 illustrates mop on a small example. We note the following property of minimum ordered pairs.

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

Proof. We start by showing $(v_1, v_k) \in \mathrm{mop}(V_1, \ldots, V_k) \Rightarrow \exists v_{k-1}$ such that $(v_1, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$ and $(v_{k-1}, v_k) \in \mathrm{mop}(\mathrm{mop}(V_1, \ldots, V_{k-1})|_2, V_k)$.

First note that $(w_1, \ldots, w_k) \in \Phi(V_1, \ldots, V_k) \Rightarrow (w_1, \ldots, w_{k-1}) \in \Phi(V_1, \ldots, V_{k-1})$. Since $(v, v_k) \in \mathrm{mop}(V_1, \ldots, V_k)$ there must be a minimum v_{k-1} such that the tuple $(v_1, \ldots, v_{k-1}) \in \Phi(V_1, \ldots, V_{k-1})$. We have $(v, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$. We need to show that $(v_{k-1}, v_k) \in \mathrm{mop}(\mathrm{mop}(V_1, \ldots, V_{k-1})|_2, V_k)$. Since $(v_1, v_k) \in \mathrm{mop}(V_1, \ldots, V_k)$ there exists no $w \in V_k$ such that $v_{k-1} \lhd w \lhd v_k$. Assume there exists a $w \in \mathrm{mop}(V_1, \ldots, V_{k-1})|_2$ such that $v_{k-1} \lhd w \lhd v_k$. Since $(v, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$ this implies that there is a $w' \rhd v_1$ s.t. $(w', w) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$. But this implies that there is a tuple $(w', \ldots, w, v_k) \in \Phi(V_1, \ldots, V_k)$ contradicting that $(v_1, v_k) \in \mathrm{mop}(V_1, \ldots, V_k)$.

We now show that if there exists a v_{k-1} such that $(v_1, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$ and $(v_{k-1}, v_k) \in \mathrm{mop}(\mathrm{mop}(V_1, \ldots, V_{k-1})|_2, V_k)$ then $(v_1, v_k) \in \mathrm{mop}(V_1, \ldots, V_k)$. Clearly, there exists a tuple $(v_1, \ldots, v_{k-1}, v_k) \in \Phi(V_1, \ldots, V_k)$. Assume that there exists a tuple $(w_1, \ldots, w_k) \in \Phi(V_1, \ldots, V_k)$ such that $v_1 \lhd w_1 \lhd w_k \unlhd v_k$. Since $w_{k-1} \unlhd v_{k-1}$ this contradicts that $(v_1, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$. Assume that there exists a tuple

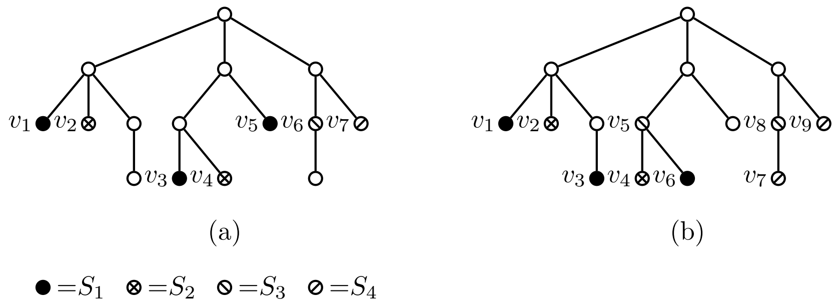

Figure 2: In (a) we have $\{(v_1, v_2, v_3, v_6, v_7), (v_1, v_2, v_5, v_6, v_7), (v_1, v_4, v_5, v_6, v_7), (v_3, v_4, v_5, v_6, v_7)\} = \Phi(S_1, S_2, S_1, S_3, S_4)$ and thus $\mathrm{mop}(S_1, S_2, S_1, S_3, S_4) = \{(v_3, v_7)\}$. In (b) we have $\Phi(S_1, S_2, S_1, S_3, S_4) = \{(v_1, v_2, v_3, v_5, v_7), (v_1, v_2, v_6, v_8, v_9), (v_1, v_2, v_3, v_8, v_9), (v_1, v_2, v_3, v_5, v_9), (v_1, v_4, v_6, v_8, v_9), (v_3, v_4, v_6, v_8, v_9)\}$ and thus $\mathrm{mop}(S_1, S_2, S_1, S_3, S_4) = \{(v_1, v_7), (v_3, v_9)\}$.

$(w_1, \ldots, w_k) \in \Phi(V_1, \ldots, V_k)$ such that $v_1 \trianglelefteq w_1 \lhd w_k \lhd v_k$. Since $(v_1, v_{k-1}) \in \mathrm{mop}(V_1, \ldots, V_{k-1})$ we have $v_{k-1} \trianglelefteq w_{k-1}$ and thus $w_k \rhd v_{k-1}$ contradicting $(v_{k-1}, v_k) \in \mathrm{mop}(\mathrm{mop}(V_1, \ldots, V_{k-1})|_2, V_k)$. □

The lemma shows that we can compute $\mathrm{mop}(V_1, \ldots, V_k)$ iteratively by first computing $\mathrm{mop}(V_1, V_2)$ and then $\mathrm{mop}(\mathrm{mop}(V_1, V_2)|_2, V_3)$ and so on.

3 Computing Deep Embeddings

In this section we present a general framework for answering tree inclusion queries. As in [19] 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) $\mathrm{label}(v) = \mathrm{label}(f(v))$. (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)

An example of an embedding is given in Figure 1(c).

Lemma 2 ([19]) *For any trees P and T. $P \sqsubseteq T$ iff there exists an embedding of P in T.*

We say that the embedding f is *deep* if there is no embedding g such that $f(\mathrm{root}(P)) \prec g(\mathrm{root}(P))$. The *deep occurrences* of P in T, denoted $\mathrm{emb}(P, T)$ is the set of nodes,

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

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.

We show how to compute deep embeddings. The key idea is to construct a data structure that allows a fast implementation of the following procedures. For all $V \in V(T)$, $U \in V(T) \times V(T)$, and $\alpha \in \Sigma$ 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 of nodes in R that are not ancestors of any other node in R.

$\textsc{Mop}_T(U, V)$. Return the set of pairs R such that for any pair $(u_1, u_2) \in U$, $(u_1, v) \in R$ iff $(u_2, v) \in \text{mop}(U|_2, V)$.

$\textsc{Fl}_T(V, \alpha)$. Return the set $R := \{\text{fl}(v, \alpha) \mid v \in V\}$.

Collectively we call these procedures the *set procedures*. 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.

$\textsc{Emb}_T(v)$ 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 $R := \textsc{Deep}_T(\textsc{Fl}_T(L(T), \text{label}(v)))$.
2. $k = 1$. Recursively compute $R_1 := \textsc{Emb}_T(v_1)$.
   Set $R := \textsc{Deep}_T(\textsc{Fl}_T(\textsc{Deep}_T(\textsc{Parent}_T(R_1)), \text{label}(v)))$.
3. $k > 1$. Initially, compute $R_1 := \textsc{Emb}_T(v_1)$, $R_2 := \textsc{Emb}_T(v_2)$. Let $U_1 := \{(r, r) \mid r \in R_1\}$ and compute $U_2 := \textsc{Mop}_T(U_1, R_2)$. Then, in order of increasing i, $3 \le i \le k$, compute $R_i := \textsc{Emb}_T(v_i)$ and $U_i := \textsc{Mop}_T(U, R_i)$. Finally, compute $R := \textsc{Deep}_T(\textsc{Fl}_T(\textsc{Deep}_T(\textsc{Nca}_T(U_k)), \text{label}(v)))$.

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

Figure 3 illustrates how $\textsc{Emb}$ works on a small example.

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 that $\text{emb}(v, T) = \textsc{Deep}_T(\textsc{Fl}_T(L(T), \text{label}(v)))$ and thus case 1 follows. Suppose that v is an internal node with $k \ge 1$ children $v_1, \ldots, v_k$. We show that $\text{emb}(P(v), T) = \textsc{Emb}_T(v)$. Consider cases 2 and 3 of the algorithm.

If $k = 1$ we have that $w \in \textsc{Emb}_T(v)$ implies that $\text{label}(w) = \text{label}(v)$ and there is a node $w_1 \in \textsc{Emb}_T(v_1)$ such that $\text{fl}(\text{parent}(w_1), \text{label}(v)) = w$, that is, no node on the path between w_1 and w is labeled $\text{label}(v)$. By induction $\textsc{Emb}_T(v_1) = \text{emb}(P(v_1), T)$ and therefore 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 it follows that $w \in \text{emb}(P(v), T)$. Conversely, if $w \in \text{emb}(P(v), T)$ then $\text{label}(w) = \text{label}(v)$, there is a node $w_1 \in \text{emb}(P(v_1), T)$ such that $w \prec w_1$, and no node on the path between w and w_1 is labeled $\text{label}(v)$, that is, $\text{fl}(w_1, \text{label}(v)) = w$. Hence, $w \in \textsc{Emb}_T(v)$.

Before considering case 3 we first show that $U_j = \text{mop}(\textsc{Emb}_T(v_1), \ldots, \textsc{Emb}_T(v_j))$ by induction on j, $2 \le j \le k$. For $j = 2$ it follows from the definition of $\textsc{Mop}_T$ that $U_2 = \text{mop}(\textsc{Emb}_T(v_1), \textsc{Emb}_T(v_2))$. Hence, assume that $j > 2$. We have $U_j = \textsc{Mop}_T(U_{j-1}, \textsc{Emb}_T(v_j)) = \textsc{Mop}_T(\text{mop}(\textsc{Emb}_T(v_1), \ldots, \textsc{Emb}_T(v_{j-1})), R_j)$. By definition of $\textsc{Mop}_T$, U_j is the set of pairs such that for any pair $(r_1, r_{j-1}) \in \text{mop}(\textsc{Emb}_T(v_1), \ldots, \textsc{Emb}_T(v_{j-1}))$, $(r_1, r_j) \in U_j$ iff $(r_{j-1}, r_j) \in \text{mop}(\text{mop}(\textsc{Emb}_T(v_1), \ldots, \textsc{Emb}_T(v_{j-1}))|_2, R_j)$. By Lemma 1 it follows that $(r_1, r_j) \in U_j$ iff $(r_1, r_j) \in \text{mop}(\textsc{Emb}_T(v_1), \ldots, \textsc{Emb}_T(v_j))$.

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

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 in an implementation of $\textsc{Emb}$ that $\textsc{Parent}$, $\textsc{Fl}$, $\textsc{Nca}$, and $\textsc{Mop}$ take deep sets as input. We will use this fact in the following sections.

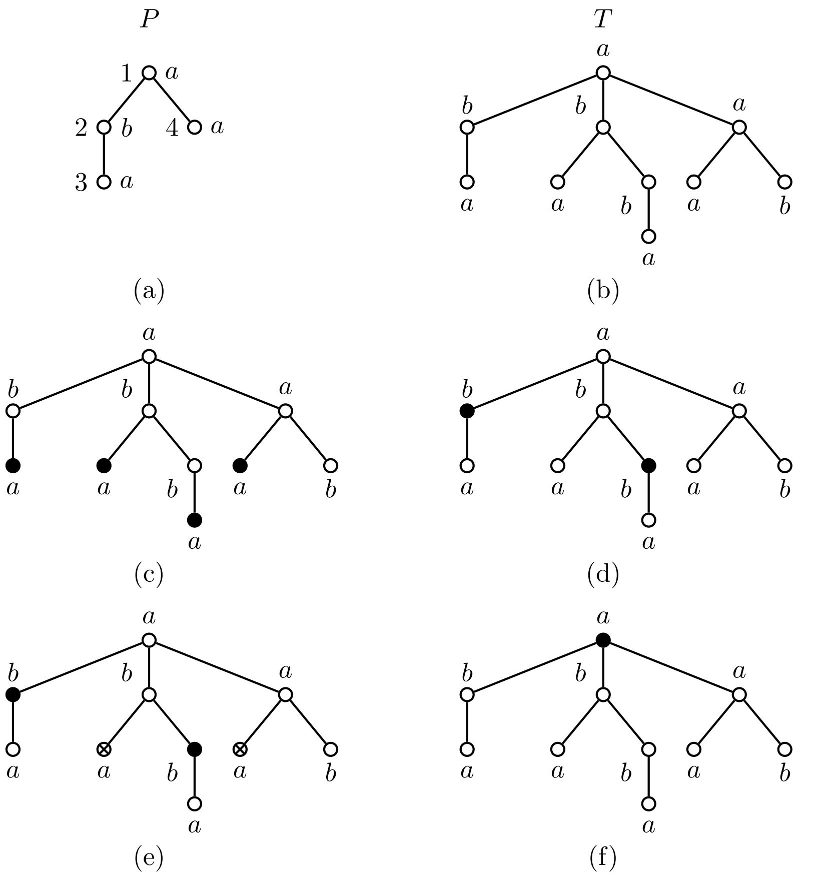

Figure 3: Computing the deep occurrences of P into T depicted in (a) and (b) respectively. The nodes in P are numbered 1–4 for easy reference. (c) Case 1 of $\textsc{Emb}$: The black nodes correspond to the set $\textsc{Emb}_T(1)$. Since 1 and 4 are leaves and label(1) = label(4) we also have that $\textsc{Emb}_T(1)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

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*.

4.1 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. Furthermore,

Lemma 4 ([15]) *For any tree T there is a data structure using $O(n_T)$ space and preprocessing time which supports nearest common ancestor queries in $O(1)$ time.*

Hence, our data structure uses linear preprocessing time and space.

4.2 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 $v_1 \unlhd v_{i+1}$, $1 \leq i < |X|$ 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 below. All lists used in the procedures are either ordered or semiordered. As noted in Section 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 and the output, of course, is ordered. Hence, the output of all the other set procedures must be semiordered.

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

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

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.

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 i 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.

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 $Z := \text{Insert}(Y[i], Z)$ and otherwise set $S := S \circ \text{parent}(Y[i])$.

Set $S := \text{Deep}_T(S)$, $Y := \text{Deep}^*_T(S, Z)$, $S := []$, and $R := []$.

Return Z.

The procedure FL calls two auxiliary procedures: INSERT(v, Z) that takes an ordered list Z and insert the node v such that the resulting list is ordered, and DEEP$^*(S, Z)$ that takes two ordered lists and returns the ordered list representing the set DEEP$(S \cup Z) \cap S$. Below we describe in more detail how to implement FL together with the auxiliary procedures.

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 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(Succ($Y[i]$)) 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 set 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 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 $O(|S| + |Z|)$.

4.3 Correctness of the Set Procedures

Clearly, PARENT and NCA are correct. The following lemmas show that DEEP, FL, and MOP are also correctly implemented.

Lemma 5 *Procedure* DEEP(X) *is correct.*

Proof. Let u be an element in X. We will first prove that if $X \cap V(T(u)) = \emptyset$ then $u \in Z$. Since $X \cap V(T(u)) = \emptyset$ we must at some point during the procedure have $v = u$, and v will not change before u is added to Z. If u occurs several times in X we will have $v = u$ each time we meet a copy of u (except the first) and it follows from the implementation that u will occur exactly once in Z.

We will now prove that if $X \cap V(T(u)) \neq \emptyset$ then $u \notin Z$. Let w be the rightmost and deepest descendant of u in X. There are two cases:

1. u is before w in X. Look at the time in the execution of the procedure when we look at w. There are two cases.
   (a) $v = u$. Since $u \prec w$ we set $v = w$ and proceed. It follows that $u \notin Z$.
   (b) $v = x \neq u$. Since any node to the left of u also is to the left of w and X is an semiordered list we must have $x \in V(T(u))$ and thus $u \notin Z$.
2. u is after w in X. Since w is the rightmost and deepest ancestor of u and X is semiordered we must have $v = w$ at the time in the procedure where we look at u. Therefore $u \notin Z$.

If u occurs several times in X, each copy will be taken care of by either case 1. or 2. □

To show that FL is correct we need the following proposition.

Proposition 1 *Let X be an ordered list and let v be an ancestor of $X[i]$ for some $i \in \{1, \ldots, k\}$. If v is an ancestor of some node in X other than $X[i]$ then v is an ancestor of $X[i-1]$ or $X[i+1]$.*

Proof. Assume for the sake of contradiction that $v \not\prec X[i-1]$, $v \not\prec X[i+1]$, and $v \prec w$, where $w \in X$. Since X is ordered either $w \lhd X[i-1]$ or $X[i+1] \lhd w$. Assume $w \lhd X[i-1]$. Since $v \prec X[i]$ and $X[i-1]$ is the left of $X[i]$, $X[i-1]$ is to the left of v contradicting $v \prec w$. Assume $X[i+1] \lhd w$. Since $v \prec X[i]$ and $X[i+1]$ is the right of $X[i]$, $X[i-1]$ is to the right of v contradicting $v \prec w$. □

Proposition 1 shows that the doubly linked list implementation of DEEP* is correct. Clearly, INSERT is implemented correct be the doubly linked list representation, since the nodes in the list remains in the same order throughout the execution of the procedure.

Lemma 6 *Procedure* $\textsc{Fl}(V, \alpha)$ *is correct.*

Proof. Let $F = \{\mathrm{fl}(v, \alpha) \mid v \in X\}$. It follows immediately from the implementation of the procedure that $\textsc{Fl}(X, \alpha) \subseteq F$. It remains to show that $\textsc{Deep}(F) \subseteq \textsc{Fl}(X, \alpha)$. Let v be a node in $\textsc{Deep}(F))$, let $w \in X$ be the node such that $v = \mathrm{fl}(w, \alpha)$, and let $w = v_1, v_2, \ldots, v_k = v$ be the nodes on the path from w to v. In each iteration of the algorithm we have $v_i \in Y$ for some i unless $v \in Z$. $\square$

Lemma 7 *Procedure* $\textsc{Mop}(X, Y)$ *is correct.*

Proof. We want to show that for $1 \le j < |X|$, $1 \le t < |Y|$, $(X[j]_1, Y[t]) \in Z$ iff $(X[j]_2, Y[t]) \in \mathrm{mop}(X|_2, Y)$. Since $X|_2$ and Y are ordered lists

$$(X[j]_2, Y[t]) \in \mathrm{mop}(X|_2, Y) \Leftrightarrow Y[t-1] \trianglelefteq X[j]_2 \triangleleft Y[t] \trianglelefteq X[j+1]_2. \tag{1}$$

First we show that $(X[j]_1, Y[t]) \in Z \Rightarrow (X[j]_2, Y[t]) \in \mathrm{mop}(X|_2, Y)$. We will break the proof into three parts, each showing one of the inequalities from the right hand side of (1).

- $Y[t-1] \trianglelefteq X[j]_2$: We proceed by induction on j. Base case $j = 1$: Immediately from the implementation of the procedure. $j > 1$: We have $x = X[j-1]_1$ and $y = Y[h]$ for some $h \le t$. By the induction hypothesis $Y[j-1] \trianglelefteq X[j-1]_2$. If $X[j]_2 \triangleleft Y[h]$ then $h = t$ since $Y[h-1] \trianglelefteq X[j-1]_2 \triangleleft X[j]_2$ and thus $Y[t-1] \triangleleft X[j]_2$. If $X[j]_2 \trianglerighteq Y[h]$ then $h \le t-1$ and thus $Y[t-1] \trianglelefteq X[j]_2$.
- $X[j]_2 \triangleleft Y[t]$: Follows immediately from the implementation of the procedure.
- $X[j+1]_2 \trianglerighteq Y[t]$: Assume $X[j+1]_2 \triangleleft Y[t]$. Consider the time in the procedure when we look at $X[j+1]_2$. We have $y = Y[t]$ and thus set $x := X[j+1]_1$ contradicting $(X[j]_1, Y[t]) \in Z$.

It follows immediately from the implementation of the procedure, that if $X[j]_2 \triangleleft Y[t]$, $Y[t-1] \trianglelefteq X[j]_2$, and $X[j+1]_2 \trianglerighteq Y[t]$ then $(X[j]_1, Y[t]) \in Z$. $\square$

4.4 Complexity of the Set Procedures

For the running time of the node list implementation observe that, given the data structure described in Section 4.1, all set procedures, except $\textsc{Fl}$, perform a single pass over the input using constant time at each step. Hence we have,

Lemma 8 *For any tree T there is a data structure using $O(n_T)$ space and preprocessing which supports each of the procedures* $\textsc{Parent}$, $\textsc{Deep}$, $\textsc{Mop}$, *and* $\textsc{Nca}$ *in linear time (in the size of their input).*

The running time of a single call to $\textsc{Fl}$ might take time $O(n_T)$. Instead we will divide the calls to $\textsc{Fl}$ 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 $\textsc{Fl}$ by $\textsc{Emb}$ is done bottom up on disjoint lists of vertices in T.

Lemma 9 *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* $\textsc{Deep}(\textsc{Fl}_T(V_i, \alpha_i))$, $2 \le i < k$, *all of* $\textsc{Fl}_T(V_1, \alpha_1), \ldots, \textsc{Fl}_T(V_k, \alpha_k)$ *can be computed in $O(n_T)$ time.*

Proof. Let Y, Z, and S be as in the implementation of the procedure. Since $\textsc{Deep}$ and $\textsc{Deep}^*$ takes time $O(S)$, we only need to show that the total length of the lists S—summed over all the calls—is $O(n_T)$ to analyze the total time usage of $\textsc{Deep}$ and $\textsc{Deep}^*$. We note that in one iteration $|S| \le |Y|$. $\textsc{Insert}$ takes constant time and it is thus enough to show that any node in T can be in Y at most twice during all calls to $\textsc{Fl}$.

Consider a call to Fl. Note that Y is ordered at all times. Except for the first iteration, a node can be in Y only if one of its children were in Y in the last iteration. Thus in one call to Fl a node can be in Y only once.

Look at a node u the first time it appears in Y. Assume that this is in the call $\textsc{Fl}(V_i, \alpha_i)$. If $u \in X$ then u cannot be in Y in any later calls, since no node in V_j where $j > i$ can be a descendant of a node in V_i. If $u \notin Z$ in this call then u cannot be in Y in any later calls. To see this look at the time when u removed from Y. Since the set $Y \cup Z$ is deep at all times no descendant of u will appear in Y later in this call to Fl, and no node in Z can be a descendant of u. Since any node in V_j, $j > i$, is an ancestor of some node in $\textsc{Deep}(\textsc{Fl}(V_i, \alpha_i))$ neither u or any descendant of u can be in any V_j, $j > i$. Thus u cannot appear in Y in any later calls to Fl. Now if $u \in Z$ then we might have $u \in V_{i+1}$. In that case, u will appear in Y in the first iteration of the procedure call $\textsc{Fl}(V_{i+1}, \alpha_i)$, but not in any later calls since the lists are disjoint, and since no node in V_j where $j > i + 1$ can be a descendant of a node in V_{i+1}. If $u \in Z$ and $u \notin V_{i+1}$ then clearly u cannot appear in Y in any later call. Thus a node in T is in Y at most twice during all the calls. □

4.5 Complexity of the Tree Inclusion Algorithm

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 8 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 $\textsc{Emb}_T(\text{root}(P))$. We bound the contribution for each node $v \in V(P)$. From Lemma 8 it follows that if v is a leaf the cost of v is at most $O(g(l_T))$. Hence, by Lemma 9, 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 $O(g(|\textsc{Emb}_T(w)|))$. If v has more than one child the cost of Mop, which dominates the time for Nca and Deep, is $\sum_{w \in \text{child}(v)} O(|\textsc{Emb}_T(w)|)$. Furthermore, since the length of the output of Mop (and thus Nca) is at most $z = \min_{w \in \text{child}(v)} |\textsc{Emb}_T(w)|$ the cost of Fl is $O(g(z))$. Hence, the total cost for internal nodes is,

$$\sum_{v \in V(P) \setminus L(P)} O\Bigg(g(\min_{w \in \text{child}(v)} |\textsc{Emb}_T(w)|) + \sum_{w \in \text{child}(v)} |\textsc{Emb}_T(w)|\Bigg) \leq \sum_{v \in V(P)} O(g(|\textsc{Emb}_T(v)|)). \qquad (2)$$

Next we bound (2). For any $w \in \text{child}(v)$ we have that $\textsc{Emb}_T(w)$ and $\textsc{Emb}_T(v)$ are disjoint ordered lists. Furthermore we have that any node in $\textsc{Emb}_T(v)$ must be an ancestor of some node in $\textsc{Deep}_T(\textsc{Fl}_T(\textsc{Emb}_T(w), \text{label}(v)))$. Hence, by Lemma 9, for any leaf to root path $\delta = v_1, \ldots, v_k$ in P, we have that $\sum_{u \in \delta} g(|\textsc{Emb}_T(u)|) \leq O(n_T)$. Let Δ denote the set of all root to leaf paths in P. It follows that,

$$\sum_{v \in V(T)} g(|\textsc{Emb}_T(v)|) \leq \sum_{p \in \Delta} \sum_{u \in p} g(|\textsc{Emb}_T(u)|) \leq O(l_P n_T).$$

Since this time dominates the time spent at the leaves the time bound follows. Next consider the space used by $\textsc{Emb}_T(\text{root}(P))$. The preprocessing of Section 4.1 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 $O(\max_{v \in V(P)} |\textsc{Emb}_T(v)|)$ space is needed. Since $\textsc{Emb}_T(v)$ a deep embedding, it follows that $|\textsc{Emb}_T(v)| \leq l_T$. □

4.6 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 $\mathrm{fl}(v, \alpha)$, $v \in V(T)$, $\alpha \in \Sigma$. Maintaining such a data structure is a well-studied problem known as the *tree color problem*. This is a well-studied problem, see e.g. [10, 25, 12, 4]. With such a data structure available we can compute $\textsc{Fl}$ as follows,

$\textsc{Fl}(X, \alpha)$ Return the list $Z := [\mathrm{fl}(X[i], \alpha) \mid 1 \leq i \leq |X|]$.

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 $O(s(n_T))$ and $O(p(n_T))$ time. As in the proof of Theorem 2 we have that the total time used by $\textsc{Emb}_T(\mathrm{root}(P))$ is bounded by $\sum_{v \in V(P)} g(|\textsc{Emb}_T(v)|)$, where $g(n)$ is the time used by $\textsc{Fl}$ on a list of length n. Since $\textsc{Emb}_T(v)$ is a deep embedding and each fl takes $q(n_T)$ we have,

$$\sum_{v \in V(P)} g(|\textsc{Emb}_T(v)|) \leq \sum_{v \in V(P)} g(l_T) = n_P l_T \cdot q(n_T).$$

□

Several firstlabel data structures are available, for instance, if we want to maintain linear space we have,

Lemma 10 (Dietz [10]) *For any tree T there is a data structure using $O(n_T)$ space, $O(n_T)$ expected preprocessing time which supports firstlabel queries in $O(\log \log n_T)$ 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.*

Note that 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 [14] 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. As discussed in the introduction the general idea is cluster T into small trees of logarithmic size which we can efficiently preprocess and then use this to speedup the computation with a logarithmic factor.

5.1 Clustering

In this section we describe how to divide T into micro trees and how the macro tree is created. For simplicity in the presentation we assume that T is a binary tree. If this is not the case it is straightforward to construct a binary tree B, where $n_B \leq 2n_T$, and a mapping $g : V(T) \to V(B)$ such that for any pair of nodes $v, w \in V(T)$, $\mathrm{label}(v) = \mathrm{label}(g(v))$, $v \prec w$ iff $g(v) \prec g(w)$, and $v \lhd w$ iff $g(v) \lhd g(w)$. If the nodes in the set $U = V(B) \backslash \{g(v) \mid v \in V(T)\}$ is assigned a special label $\beta \notin \Sigma$ it follows that for any tree P, $P \sqsubseteq T$ iff $P \sqsubseteq B$.

Let C be a connected subgraph of T. A node in $V(C)$ incident to a node in $V(T) \backslash V(C)$ is a *boundary* node. The boundary nodes of C are denoted by δC. A *cluster* of C is a connected subgraph of C with at most two boundary nodes. A set of clusters CS is a *cluster partition* of T iff $V(T) = \cup_{C \in CS} V(C)$, $E(T) = \cup_{C \in CS} E(C)$, and for any $C_1, C_2 \in CS$, $E(C_1) \cap E(C_2) = \emptyset$, $|E(C_1)| \geq 1$, $\mathrm{root}(T) \in \delta C$ if $\mathrm{root}(T) \in V(C)$. If $|\delta C| = 1$ we call C a *leaf cluster* and otherwise an *internal cluster.*

We use the following recursive procedure $\textsc{Cluster}_T(v,s)$, adopted from [5], which creates a cluster partition CS of the tree $T(v)$ with the property that $|CS| = O(s)$ and $|V(C)| \le \lceil n_T/s \rceil$. A similar cluster partitioning achieving the same result follows from [3, 2, 13].

$\textsc{Cluster}_T(v,s)$. For each child u of v there are two cases:

1. $|V(T(u))| + 1 \le \lceil n_T/s \rceil$. Let the nodes $\{v\} \cup V(T(u))$ be a leaf cluster with boundary node v.
2. $|V(T(u))| > \lceil n_T/s \rceil$. Pick a node $w \in V(T(u))$ of maximum depth such that $|V(T(u))| + 2 - |V(T(w))| \le \lceil n_T/s \rceil$. Let the nodes $V(T(u)) \backslash V(T(w)) \cup \{v,w\}$ be an internal cluster with boundary nodes v and w. Recursively, compute $\textsc{Cluster}_T(w,s)$.

Lemma 11 *Given a tree T with $n_T > 1$ nodes, and a parameter s, where $\lceil n_T/s \rceil \ge 2$, we can build a cluster partition CS in $O(n_T)$ time, such that $|CS| = O(s)$ and $|V(C)| \le \lceil n_T/s \rceil$ for any $C \in CS$.*

Proof. The procedure $\textsc{Cluster}_T(\mathrm{root}(T), s)$ clearly creates a cluster partition of T and it is straightforward to implement in $O(n_T)$ time. Consider the size of the clusters created. There are two cases for u. In case 1, $|V(T(u))| + 1 \le \lceil n_T/s \rceil$ and hence the cluster $C = \{v\} \cup V(T(u))$ has size $|V(C)| \le \lceil n_T/s \rceil$. In case 2, $|V(T(u))| + 2 - |V(T(w))| \le \lceil n_T/s \rceil$ and hence the cluster $C = V(T(u)) \backslash V(T(w)) \cup \{v,w\}$ has size $|V(C)| \le \lceil n_T/s \rceil$.

Next consider the size of the cluster partition. We say that a cluster C is *bad* if $|V(C)| \le c/2$ and *good* otherwise. We will show that at least a constant fraction of the clusters in the cluster partition are good. Let $c = \lceil n_T/s \rceil$. It is easy to verify that the cluster partition created by procedure $\textsc{Cluster}$ has the following properties:

(i) Let C be a bad internal cluster with boundary nodes v and w ($v \prec w$). Then w has two children with at least $c/2$ descendants each.

(ii) Let C be a bad leaf cluster with boundary node v. Then the boundary node v is contained in a good cluster.

By (ii) the number of bad leaf clusters is no larger than twice the number of good internal clusters. By (i) each bad internal cluster C is sharing its lowest boundary node of C with two other clusters, and each of these two clusters are either internal clusters or good leaf clusters. This together with (ii) shows that number of bad clusters is at most a constant fraction of the total number of clusters. Since a good cluster is of size more than $c/2$, there can be at most $2s$ good clusters and thus $|CS| = O(s)$. $\square$

Let $C \in CS$ be an internal cluster $v, w \in \delta C$. The *spine path* of C, $\pi(C)$, is the path between v, w excluding v and w. A node on the spine path is a *spine node*. A node to the left and right of v, w, or any node on $\pi(C)$ is a *left node* and *right node* respectively. If C is a leaf cluster with $v \in \delta C$ then any proper descendant of v is a *leaf node*.

Let CS be a cluster partition of T as described in Lemma 11. We define an ordered *macro tree* T^M. Our definition of T^M may be viewed as an "ordered" version of the macro tree given in [5]. For each internal cluster $C \in CS$, $v, w \in \delta C$, $v \prec w$, we have the node $s(v,w)$ and edges $(v, s(v,w)), (s(v,w), w)$. Furthermore, we have the nodes $l(v,w)$ and $r(v,w)$ and edges $(l(v,w), s(v,w))$ and $(r(v,w), s(v,w))$ ordered such that $l(v,w) \lhd w \lhd r(v,w)$. If C is a leaf cluster and $v \in \delta C$ we have the node $l(v)$ and edge $(l(v), v)$. Since $\mathrm{root}(T)$ is a boundary node T^M is rooted at $\mathrm{root}(T)$. Figure 4 illustrates these definitions.

To each node $v \in V(T)$ we associate a unique macro node denoted $i(v)$. If $u \in V(C)$ and $C \in CS$, then

$$i(u) = \begin{cases} u & \text{If } u \text{ is boundary,} \\ s(v,w) & \text{if } u \text{ is a spine node and } v, w \in \delta C, \\ l(v,w) & \text{if } u \text{ is a left node and } v, w \in \delta C, \\ r(v,w) & \text{if } u \text{ is a right node and } v, w \in \delta C, \\ l(v) & \text{if } u \text{ is a leaf node and } v \in \delta C. \end{cases}$$

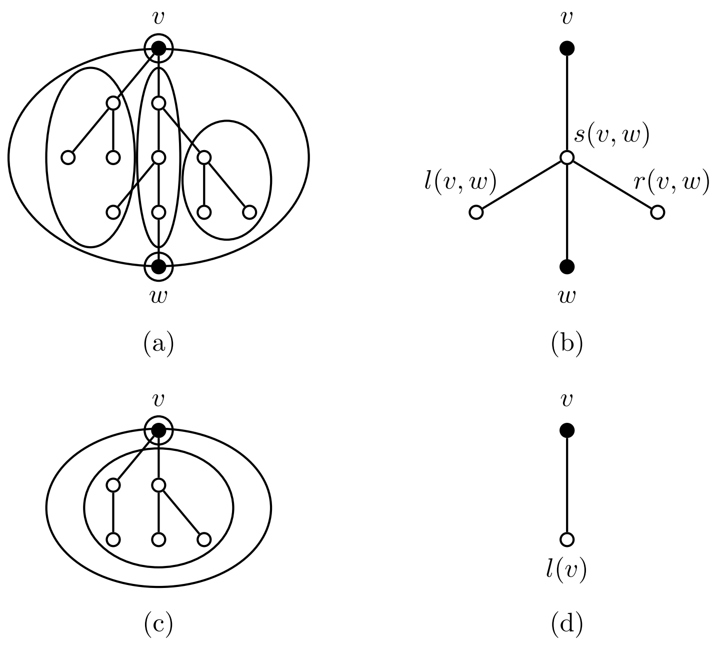

Figure 4: The clustering and the macro tree. (a) An internal cluster. The black nodes are the boundary node and the internal ellipses correspond to the boundary nodes, the right and left nodes, and spine path. (b) The macro tree corresponding to the cluster in (a). (c) A leaf cluster. The internal ellipses are the boundary node and the leaf nodes. (d) The macro tree corresponding to the cluster in (c).

Conversely, for any macro node $x \in V(T^M)$ define the *macro-induced subgraph*, denoted $I(x)$, as the induced subgraph of T of the set of nodes $\{v \mid v \in V(T), x = i(v)\}$. We also assign a *set* of labels to x given by $\text{label}(x) = \{\text{label}(v) \mid v \in V(I(x))\}$. If x is spine node or a boundary node the unique node in $V(I(x))$ of greatest depth is denoted by $\text{first}(x)$. Finally, for any set of nodes $\{x_1, \dots, x_k\} \subseteq V(T^M)$ we define $I(x_1, \dots, x_k)$ as the induced subgraph of the set of nodes $V(I(x_1)) \cup \cdots \cup V(I(x_k))$.

The following propositions states useful properties of ancestors, nearest common ancestor, and the left-to-right ordering in the cluster partition and in T. The propositions follows directly from the definition of the clustering.

Proposition 2 *For any pair of nodes $v, w \in V(T)$, the following hold*

(i) If $i(v) = i(w)$ then $v \prec_T w$ iff $v \prec_{I(i(v))} w$.

(ii) If $i(v) \neq i(w)$, $i(v) \in \{s(v', w'), v'\}$ and $i(w) \in \{l(v', w'), r(v', w')\}$, then $v \prec_T w$ iff $v \prec_{I(i(v), s(v', w'), v')} w$.

(iii) In all other cases, $w \prec_T v$ iff $i(w) \prec_{T^M} i(v)$.

Proposition 3 *For any pair of nodes $v, w \in V(T)$, the following hold*

(i) If $i(v) = i(w)$ then $v \lhd w$ iff $v \lhd_{I(i(v))} w$.

(ii) If $i(v) = l(v', w')$, $i(w) \in \{s(v', w'), v'\}$ then $v \lhd w$ iff $v \lhd_{I(l(v', w'), s(v', w'), v')} w$.

(iii) If $i(v) = r(v', w')$, $i(w) \in \{s(v', w'), v'\}$ then $w \lhd v$ iff $w \lhd_{I(r(v', w'), s(v', w'), v')} v$.

(iv) In all other cases, $v \lhd w$ iff $i(v) \lhd_{T^M} i(w)$.

Proposition 4 *For any pair of nodes $v, w \in V(T)$, the following hold* *(i)* *If* $i(v) = i(w) = l(v')$ *then* $\text{nca}_T(v, w) = \text{nca}_{I(i(v),v')}(v, w)$.

(ii) *If* $i(v) = i(w) \in \{l(v', w'), r(v', w')\}$ *then* $\text{nca}_T(v, w) = \text{nca}_{I(i(v),s(v',w'),v')}(v, w)$.

(iii) *If* $i(v) = i(w) = s(v', w')$ *then* $\text{nca}_T(v, w) = \text{nca}_{I(i(v))}(v, w)$.

(iv) *If* $i(v) \neq i(w)$ *and* $i(v), i(w) \in \{l(v', w'), r(v', w'), s(v', w')\}$ *then* $\text{nca}_T(v, w) = \text{nca}_{I(i(v),i(w),s(v',w'),v')}(v, w)$.

(v) *If* $i(v) \neq i(w)$, $i(v) \in \{l(v', w'), r(v', w'), s(v', w')\}$, *and* $i(w) \preceq_{T^M} w'$ *then* $\text{nca}_T(v, w) = \text{nca}_{I(i(v),s(v',w'),w')}(v, w')$.

(vi) *If* $i(v) \neq i(w)$, $i(w) \in \{l(v', w'), r(v', w'), s(v', w')\}$, *and* $i(v) \preceq_{T^M} w'$ *then* $\text{nca}_T(v, w) = \text{nca}_{I(i(w'),s(v',w'),w')}(w, w')$.

(vii) *In all other cases,* $\text{nca}_T(v, w) = \text{nca}_{T^M}(i(v), i(w))$.

5.2 Preprocessing

In this section we describe how to preprocess T. First we make a cluster partition CS of the tree T with clusters of size s, to be fixed later, and the corresponding macro tree T^M in $O(n_T)$ time. The macro tree is preprocessed as in 4.1. However, since nodes in T^M contain a set of labels, we store for each node $v \in V(T^M)$ a dictionary of label(v). Using perfect hashing the total time to compute all these dictionaries is $O(n_T)$ expected time. Furthermore, we modify the definition of fl such that $\text{fl}_{T^M}(v, \alpha)$ is the nearest ancestor w of v such that $\alpha \in \text{label}(w)$.

Next we show how to preprocess the micro trees. For any labeled, ordered, forest S and $M, N \subseteq V(S)$ we define, in addition, to the set procedures the following useful procedures.

ANCESTOR$_S(M)$. Return the set of all ancestors of nodes in M.

LEFTOF$_S(M, N)$. Return a boolean indicating whether there is at least one node $v \in M$ such that for all nodes $w \in S$, $v \trianglelefteq w$.

LEFT$_S(M)$. Return the leftmost node in M.

RIGHT$_S(M)$. Return the rightmost node in M.

MATCH$_S(M, N, O)$, where $M = \{m_1 \lhd \cdots \lhd m_k\}$, $N = \{v_1 \lhd \cdots \lhd v_k\}$, $O = \{o_1 \lhd \cdots \lhd o_l\}$, and $o_i = v_j$ for some j. Return the set $R := \{m_j \mid o_i = v_j, 1 \leq i \leq l\}$.

MOP$_S(M, N)$ Return the triple $(R_1, R_2, \textsf{bool})$. Where $R_1 = \text{mop}(M, N)|_1$ and $R_2 = \text{mop}(M, N)|_2$, and $\textsf{bool}$ indicates whether there is any node in $v \in M$ such that for all nodes $w \in N$, $v \trianglerighteq w$.

MASK$_S(\alpha)$, $\alpha \in \Sigma$. Return the set of nodes with label α.

We show how to implement these procedures on all macro induced subforest S of each cluster $C \in CS$. Note that a cluster contains at most a constant number of such subforests. For the procedures PARENT$_S$, ANCESTOR$_S$, DEEP$_S$, NCA$_S$, LEFTOF$_S$, LEFT$_S$, RIGHT$_S$, MATCH$_S$, and MOP$_S$ we will simply precompute and store the result of any input for all forests S with at most s nodes. We assume that the input and output node sets of the above procedures is given as a bitstring of length s. Hence, for any forest S the total of number of distinct node sets is at most 2^s. Since at most 3 input sets occur in the procedures and the total number of forest of size at most s is $O(2^{2s})$ it follows that there are $2^{O(s)}$ distinct inputs to each procedure to precompute and store. Furthermore, it is straightforward to compute all results within the same time bound. If c is the constant hidden in the O notation we set $s = \frac{1}{c} \log n_T$ and the total preprocessing time and space used becomes $2^{cs} = 2^{\log n_T} = n_T$. Furthermore, since the size of the input to procedures is logarithmic we can lookup the result of any input in constant time.

Next we show how to compute the remaining procedures $\textsc{Mask}_S$ and $\textsc{Fl}_S$. Note that since the size of the alphabet is potentially $\Omega(n_T)$, we cannot precompute all values for these procedures in $O(n_T)$ time. Instead we implement $\textsc{Mask}_S$ using a dictionary for each subforest S indexed by the labels in S. Again, using perfect hashing we can build all such tables in $O(n_T)$ excepted time using linear space. Hence, we can lookup $\textsc{Mask}_S$ in constant time. Finally, we can compute $\textsc{Fl}$ in constant time with the other procedures since $\textsc{Fl}_S(M, \alpha) = \textsc{Deep}_S(\textsc{Ancestor}_S(M)$ `and` $\textsc{Mask}_S(\alpha))$, where `and` denotes a bitwise and operation.

As discussed in Section 4.6, if we require worst-case running times instead of the expected $O(n_T)$ time above we may instead use a deterministic dictionary without changing the overall running time of our tree inclusion algorithm.

5.3 A Compact Representation of Node Sets

In this section we show how to implement the set procedures in sublinear time using the clustering and preprocessing defined in the previous section.

First we define a compact representation of node sets. A *micro-macro node set* (mm-node set) $\mathcal{V}$ for a tree T with macro tree T^M is a set of pairs $\mathcal{V} = \{(x_1, M(x_k)), \dots, (x_k, M(x_k))\}$, such that for any pair $(x, M(x)) \in \mathcal{V}$:

(i) $x \in V(T^M)$,

(ii) $M(x) \subseteq V(I(x))$,

(iii) $M(x) \neq \emptyset$.

Additionally, if for any pairs $(x, M(x)), (y, M(y)) \in \mathcal{X}$:

(iv) $x \neq y$,

we say that $\mathcal{V}$ is *canonical.* For any mm-node set $\mathcal{V}$ there is a corresponding set of nodes $S(\mathcal{V}) \subseteq V(T)$ given by $S(\mathcal{V}) = \cup_{(x,M(x))\in\mathcal{V}} M(x)$. Conversely, given a set of nodes V there is a unique canonical mm-node set $\mathcal{V}$ given by:

$$\mathcal{V} = \{(x, M(x)) \mid M(x) = V(I(x)) \cap X \neq \emptyset\}.$$

We say that $\mathcal{V}$ is deep iff the set $S(\mathcal{V})$ is deep. Note that by Lemma 2(ii) an mm-node set $\mathcal{V}$ may be deep even though the node set $\mathcal{V}|_1$ is not. Since the size of the macro tree is $O(n_T/\log n_T)$ we have that,

Lemma 12 *For any canonical mm-node set $\mathcal{V}$, $|\mathcal{V}| \leq O(n_T/\log n_T)$.*

As with node lists, we define a *micro-macro node list* (mm-node list), $\mathcal{X} = [(x_1, M(x_k)), \dots, (x_k, M(x_k))]$, as a list where each element is an element of an mm-node set. We say that $\mathcal{X}$ is ordered if $x_1 \lhd_{T^M} \cdots \lhd_{T^M} x_k$ and semiordered if $x_1 \unlhd_{T^M} \cdots \unlhd_{T^M} x_k$.

In the following we show how to implement the set procedures using mm-node lists. As before we assume that the input to each of the procedures is deep. Each of the procedures, except $\textsc{Deep}$, accept as input mm-node lists which are semiordered, canonical, and deep and return as output semiordered mm-node list(s). The input for $\textsc{Deep}$ is semiordered and canonical and the output is semiordered, canonical, and deep. Since the output of the procedures is not necessarily canonical and $\textsc{Deep}$ requires canonical input we need the following additional procedure to make $\textsc{Emb}$ work:

$\textsc{Canonical}(\mathcal{X})$, where $\mathcal{X}$ is a semiordered mm-node list. Return a semiordered canonical mm-list $\mathcal{R}$ such that $S(\mathcal{R}) = S(\mathcal{X})$.

We simply run this procedure on any input mm-node list to $\textsc{Deep}$ immediately before executing $\textsc{Deep}$.

5.4 Implementation of the Set Procedures

The implementation of all set procedures is described in this section.

Parent($\mathcal{X}$). Initially, set $\mathcal{R} := []$. For each i, $2 \leq i \leq |\mathcal{X}|$, set $(x, M(x)) := \mathcal{X}[i]$. There are three cases:

1. $x \in \{l(v,w), r(v,w)\}$. Compute $N = \textsc{Parent}_{I(x,s(v,w),v)}(M(x))$. For each macro node $s \in \{x, s(v,w), v\}$ (in semiorder) set $\mathcal{R} := \mathcal{R} \circ (s, N \cap V(I(s)))$ if $N \cap V(I(s)) \neq \emptyset$.
2. $x = l(v)$. Compute $N = \textsc{Parent}_{I(x,v)}(M(x))$. For each macro node $s \in \{x, v\}$ (in semiorder) set $\mathcal{R} := \mathcal{R} \circ (s, N \cap V(I(s)))$ if $N \cap V(I(s)) \neq \emptyset$.
3. $x \notin \{l(v,w), r(v,w), l(v)\}$. If $N = \textsc{Parent}_{I(x)}(M(x)) \neq \emptyset$ set $\mathcal{R} := \mathcal{R} \circ (x, N)$. Otherwise, if $\text{parent}_{T^M}(x) \neq \bot$ set $\mathcal{R} := \mathcal{R} \circ (\text{parent}_{T^M}(x), \text{first}(\text{parent}_{T^M}(x)))$.

Return $\mathcal{R}$.

Consider the three cases of procedure Parent. Case 1 handles the fact that left and right nodes may have a spine node or a boundary node as parent. Since no left or right node can have a parent outside their cluster there is no need to compute parents in the macro tree. Case 2 handles the fact that the nodes in a leaf node may have the boundary node as parent. Since none of the nodes in the leaf node can have a parent outside their cluster there is no need to compute parents in the macro tree. Case 3 handles boundary and spine nodes. Since the input to Parent is deep there is either a parent within the micro tree or we can use the macro tree to compute the parent of the root of the micro tree.

Nca($\mathcal{X}$). Initially, set $\mathcal{R} := []$. For each i, $1 \leq i \leq |\mathcal{X}|$, set $(x, M(x)) := \mathcal{X}[i]_1$ and $(y, M(y)) := \mathcal{X}[i]_2$ and compare x and y. There are two cases:

1. $x = y$: Let $z := x$. There are two subcases:
   If z is a boundary node then set $\mathcal{R} := \mathcal{R} \circ (z, z)$. Otherwise set
   $$S := \begin{cases} I(z,v), & \text{if } z = l(v), \\ I(z, s(v,w), v), & \text{if } z \in \{l(v,w), r(v,w)\}, \\ I(z), & \text{if } z = s(v,w). \end{cases}$$
   Compute $M := \textsc{Nca}_S(M(x), M(y))$. For each macro node s in S (in semiorder) we set $\mathcal{R} := \mathcal{R} \circ (s, M \cap V(I(s)))$ if $M \cap V(I(s)) \neq \emptyset$.
2. $x \neq y$: Compute $z := \textsc{Nca}_{T^M}(x, y)$. There are two subcases:
   If z is a boundary node then set $\mathcal{R} := \mathcal{R} \circ (z, z)$. Otherwise z must be a spine node $s(v,w)$. There are three cases:
   (a) If $x \in \{l(v,w), s(v,w)\}$ and $y \in \{s(v,w), r(v,w)\}$ compute $M := \textsc{Nca}_{I(x,y,s(v,w),v)}(M(x), M(y))$.
   (b) If $x = l(v,w)$ and $y \preceq_T w$ compute $M := \textsc{Nca}_{I(x,s(v,w),w)}(M(x), w)$.
   (c) If $y = r(v,w)$ and $x \preceq_T w$ compute $M := \textsc{Nca}_{I(y,s(v,w),w)}(w, M(y))$.
   Set $\mathcal{R} := \mathcal{R} \circ (z, M \cap V(I(z)))$.

Return $\mathcal{R}$.

Consider the two cases of procedure Nca. Case 1 handles the cases (i), (ii), and (iii) from Proposition 4. Case 2 handles the cases (iv), (v), (vi) and (vii) from Proposition 4.

Deep($\mathcal{X}$). Initially, set $(x, M(x)) := \mathcal{X}[1]$ and $\mathcal{R} := []$. For each i, $2 \leq i \leq |\mathcal{X}|$, set $(x_i, M(x_i)) := \mathcal{X}[i]$ and compare x and x_i:

1. $x \lhd x_i$: Set $\mathcal{R} := \mathcal{R} \circ (x, \textsc{Deep}_{I(x)} M(x))$, and $(x, M(x)) := (x_i, M(x_i))$.

2. $x \prec x_i$: If $x_i \in \{l(v,w), r(v,w)\}$ and $x = s(v,w)$ compute $N := \text{DEEP}_{I(x_i,s(v,w))}(M(x) \cup M(x_i))$. Then, set $(x, M(x)) := (x, N(x))$ and if $N(x_i) := N \cap I(x_i) \neq \emptyset$ set $\mathcal{R} := \mathcal{R} \circ (x_i, N(x_i))$. Otherwise ($x_i \notin \{l(v,w), r(v,w)\}$ or $x \neq s(v,w)$) set $(x, M(x)) := (x_i, M(x_i))$.
3. $x_i \prec x$: As above, with x and x_i replaced by each other.

Return $\mathcal{R}$.

The above DEEP procedure resembles the previous DEEP procedure implemented on the macro tree. The biggest difference is that a mm-node set $\mathcal{X}$ may be deep even though the set $\mathcal{X}|_1$ is not deep in T^M. However, this can only happen for nodes in the same cluster which is straightforward to handle (see Proposition 2(i) and (ii)).

MOP$(\mathcal{X}, \mathcal{Y})$. Initially, set $\mathcal{R} := []$,$\mathcal{X}' := \mathcal{X}|_1$, $\mathcal{Z} := \mathcal{X}|_2$, $r := \perp$, $s := \perp$, $i := 1$, and $j := 1$. Repeat the following until $i > |\mathcal{X}|$ or $h > |\mathcal{Y}|$:

If $\mathcal{Z}[i]_1 = l(v,w)$ set $j := j+1$ until $\mathcal{Z}[i]_1 \trianglelefteq \mathcal{Y}[j]_1$ or $\mathcal{Y}[j]_1 = s(v,w)$. If $\mathcal{Z}[i]_1 = s(v,w)$ set $j := j+1$ until $\mathcal{Z}[i]_1 \trianglelefteq \mathcal{Y}[j]_1$ or $\mathcal{Y}[j]_1 = r(v,w)$. Otherwise set $j := j+1$ as long as $\mathcal{Z}[i]_1 \triangleright \mathcal{Y}[j]_1$. Set $(x, M(x)) := \mathcal{X}'[i]$, $(z, M(z)) := \mathcal{Z}[i]$, and $(y, M(y)) := \mathcal{Y}[j]$. There are two cases:

1. $z \lhd y$: If $s \lhd y$ set $\mathcal{R} := \mathcal{R} \circ (r,s)$. Set $r := (x, \text{RIGHT}_{I(x)}(M(x)))$, $s := (y, \text{LEFT}_{I(y)}(M(y)))$, and $i := i+1$.
2. Either
   (a) $z = y$,
   (b) $z = l(v,w)$ and $y = s(v,w)$,
   (c) $z = s(v,w)$ and $y = r(v,w)$.
   If $s \lhd y$ then set $\mathcal{R} := \mathcal{R} \circ (r,s)$.
   If $s = y$ and $\text{LEFTOF}_{I(z)}(M(z), M(y)) = \mathsf{true}$ then set $\mathcal{R} := \mathcal{R} \circ (r,s)$.
   Compute $(M_1, M_2, \mathsf{match\_x}) := \text{MOP}_{I(z,y)}(M(z), M(y))$. If $M_1 \neq []$ then compute $M := \text{MATCH}(M(x), M(z), M_1)$, and set $\mathcal{R} := \mathcal{R} \circ ((x, M), (y, M_2))$. Set $r := \perp$, $s := \perp$, and $j := j+1$. If $\mathsf{match\_x} = \mathsf{false}$ set $i := i+1$.

Return $\mathcal{R}$.

The above MOP procedure resembles the previous MOP procedure implemented on the macro tree in one of the cases. Case 1 in the above iteration is almost the same as the previous implementation of the procedure. Case 2(a) are due to the fact that we can have nearest neighbor pairs within a macro-induced subtree $I(x)$. Cases 2(b) and 2(c) takes care of the special cases caused by the spine nodes.

FL$(\mathcal{X}, \alpha)$. Initially, set $\mathcal{R} := []$ and $S := []$. For each $(x, M(x)) := \mathcal{X}[i]$, $1 \leq i \leq |\mathcal{X}|$ there are 2 cases:

1. $x \in \{l(v,w), r(v,w)\}$. Compute $N = \text{FL}_{I(x,s(v,w),v)}(M(x), \alpha)$. If $N \neq \emptyset$, then for each macro node $s \in \{x, s(v,w), v\}$ (in semiorder) set $\mathcal{R} := \mathcal{R} \circ (s, N \cap V(I(s)))$ if $N \cap V(I(s)) \neq \emptyset$. Otherwise, set $U := U \circ \text{parent}(v)$.
2. $x \notin \{l(v,w), r(v,w)\}$. Compute $N = \text{FL}_{I(x)}(M(x), \alpha)$. If $N \neq \emptyset$ set $\mathcal{R} := \mathcal{R} \circ (x, N)$ and otherwise set $U := U \circ \text{parent}(x)$.

Subsequently, compute $S := \text{FL}_{T^M}(U, \alpha)$, and use this result to compute the mm-node list $\mathcal{S} := [(S[i], \text{FL}_{I(S[i])}(\text{first}(S[i]), \alpha)) \mid 1 \leq i \leq |S|]$. Merge the mm-node lists $\mathcal{S}$ and $\mathcal{R}$ with respect to semiorder and return the result.

The FL procedure is similar to PARENT. The cases 1 and 2 compute FL on a micro tree. If the result is within the micro tree we add it to $\mathcal{R}$ and otherwise we store the node in the macro tree which contains parent of the root of the micro tree in a node list S. We then compute FL in the macro tree on the list S and use this to compute the final result.

Finally, we give the trivial CANONICAL procedure.

Canonical($\mathcal{X}$). For each node $x \in V(T^M)$ maintain a set $N(x) \subseteq I(V(x))$ initially empty. For each i, $1 \le i \le |\mathcal{X}|$ set $N(\mathcal{X}[i]_1) := N(\mathcal{X}[i]_1) \cup \mathcal{X}[i]_2$. Then, set $\mathcal{R} := []$ and traverse T^M in any semiordering. For each node $x \in V(T^M)$, if $N(x) \neq \emptyset$ set $\mathcal{R} := \mathcal{R} \circ (x, N(x))$.

Return $\mathcal{R}$.

5.5 Correctness of the Set Procedures

In this section we show the correctness of the mm-node set implementation of the set procedures.

Lemma 13 *Procedure* Parent($\mathcal{X}$) *is correct.*

Proof. Follows immediately by looking at all different kinds of macro nodes, and by the comments below the implementation of the procedure. □

Lemma 14 *Procedure* Nca($\mathcal{X}$) *is correct.*

Proof. Let $(x, M(x)) := \mathcal{X}[i]_1$ and $(y, M(y)) := \mathcal{X}[i]_2$. We will show that $v \in \mathrm{nca}_T(M(x), M(y))$ iff $v \in S(\mathcal{R})$. We first show $v \in \mathrm{nca}_T(M(x), M(y)) \Rightarrow v \in S(\mathcal{R})$. There must exist $u \in M(x)$ and $w \in M(y)$ such that $v = \mathrm{nca}_T(u, w)$. Consider the cases of Proposition 4. In case (i), (ii), and (iii) we have $x = y$. This is Case 1 in the procedure. It follows immediately from the implementation that $v \in \mathcal{R}$. Case (iv)-(vi). This is Case 2(a)-(c) in the procedure since the input is semiordered. Case (vii) is taken care of by both case 1 and 2 in the procedure (z is a boundary node).

That $v \in \mathrm{nca}_T(M(x), M(y)) \Leftarrow v \in S(\mathcal{R})$ follows immediately from the implementation and Proposition 4. □

Lemma 15 *Procedure* Deep($\mathcal{X}$) *is correct.*

Proof. The input to Deep is canonical and semiordered. Let $u \in S(\mathcal{X})$ and $M = S(\mathcal{X}) \cap V(T(u))$. We will show $M = \emptyset$ iff $u \in S(\mathcal{R})$. At some point during the execution of the procedure we have $u \in M(x_i)$.

We first prove $M = \emptyset \Rightarrow u \in S(\mathcal{R})$. Consider the iteration where $u \in M(x_i)$. It is easy to verify that either $u \in S(\mathcal{R})$ after this iteration or $u \in M(x)$. Now assume $u \in M(x)$. It is easy to verify that we have $u \in M(x)$ until $(x, \mathrm{Deep}_{I(x)}(M(x)))$ is appended to $\mathcal{R}$. Since $M = \emptyset$ we have $u \in \mathrm{Deep}_{I(x)}(M(x))$ and thus $u \in S(\mathcal{R})$. □

To prove the correctness of procedure Mop we need the following proposition.

Proposition 5 *Let $\mathcal{R} = [(r_i, M(r_i)) \mid 1 \le i \le k]$ and $\mathcal{S} = [(s_i, M(s_i)) \mid 1 \le i \le l]$ be deep, canonical lists. For any pair of nodes $r \in M(r_i)$, $s \in M(s_j)$ for some i and j, then $(r, s) \in \mathrm{mop}_T(S(\mathcal{R}), S(\mathcal{S}))$ iff one of the following cases are true:*

(i) $r_i = s_j$ *and* $(r, s) \in \mathrm{mop}_{I(r_i)}(M(r_i), M(s_j))$.

(ii) $r_i = l(v, w)$, $s_j = s(v, w)$ *and* $(r, s) \in \mathrm{mop}_{I(r_i, s_j)}(M(r_i), M(s_j))$.

(iii) $r_i = s(v, w)$, $s_j = r(v, w)$ *and* $(r, s) \in \mathrm{mop}_{I(r_i, s_j)}(M(r_i), M(s_j))$.

(iv) $r_i = l(v, w)$, $s_j = r(v, w)$, $r_{i+1} \neq s(v, w)$, $s_{j-1} \neq s(v, w)$, $r = \mathrm{RIGHT}(M(r_i))$, $s = \mathrm{LEFT}(M(s_j))$, *and* $(r_i, s_j) \in \mathrm{mop}_{T^M}(\mathcal{R}|_1, \mathcal{S}|_1)$.

(v) $r_i, s_j \in C \in CS$, $r_i \neq s_j$, *either* r_i *or* s_j *is the bottom boundary node* w *of* C, $r = \mathrm{RIGHT}(M(r_i))$, $s = \mathrm{LEFT}(M(s_j))$, *and* $(r_i, s_j) \in \mathrm{mop}_{T^M}(\mathcal{R}|_1, \mathcal{S}|_1)$.

(vi) $r_i \in C_1 \in CS$, $s_j \in C_2 \in CS$, $C_1 \neq C_2$, $r = \textsc{right}(M(r_i))$, $s = \textsc{left}(M(s_j))$, *and* $(r_i, s_j) \in \mathrm{mop}_{T^M}(\mathcal{R}|_1, \mathcal{S}|_1)$.

The proposition follows immediately, by considering all cases for r_i and s_j, *i.e.*, $r_i = s_j$, r_i and s_j are in the same cluster, and r_i and s_j are not in the same cluster. Using Proposition 5 we get

Lemma 16 *Procedure* $\textsc{Mop}(\mathcal{X}, \mathcal{Y})$ *is correct.*

Proof. Let $(x, M(x)) = \mathcal{X}'[i]$ and $(z, M(z)) = \mathcal{Z}[i]$. We call r, t a corresponding pair in $(M(x), M(z))$ iff r and t are the ith node in the left to right order of $M(x)$ and $M(z)$, respectively. Let

$$S := \{(r, s) \mid r, t \text{ corresponding pair in } (M(x), M(z)), \text{ and } (t, s) \in \mathrm{mop}_T(S(\mathcal{Z}), S(\mathcal{Y}))\}.$$

We first show $(v_x, v_y) \in S \Rightarrow (v_x, v_y)$ is a corresponding pair in $(\mathcal{R}[i]_1, \mathcal{R}[i]_2)$. Let (v_z, v_y) be the pair in $\mathrm{mop}_T(S(\mathcal{Z}), S(\mathcal{Y}))$, where $v_z \in M(z_i)$ and $v_y \in M(y_j)$, and look at each of the cases from Proposition 5.

- Case (i), (ii), and (iii). This is case 2 in the procedure. We have $v_x \in M$ and $v_y \in M_2$, which are both added to $\mathcal{R}$.
- Case (iv), (v), and (vi). This is case 1 in the procedure. Here we set $r := (x, \textsc{right}_{I(x)}(M(x)))$ and $s := (y, \textsc{left}_{I(y)}(M(y)))$, where such $v_x \in M(x)$ and $v_y \in M(y)$. We need to show that (r, s) is added to $\mathcal{R}$ before r and s are changed. If the next case is (i) again then it follows from the fact that $(z_i, y_j) \in \mathrm{mop}_{T^M}(\mathcal{Z}|_1, \mathcal{Y}|_1)$. If the next case is *(ii)* then we must have $s \lhd y$ or $s = y$ and $\textsc{leftof}_{I(z)}(M(z), M(y)) = \mathsf{true}$ since $(z_i, y_j) \in \mathrm{mop}_{T^M}(\mathcal{Z}|_1, \mathcal{Y}|_1)$.

We now show if (v_x, v_y) is a corresponding pair in $(\mathcal{R}[i]_1, \mathcal{R}[i]_2)$ then $(v_x, v_y) \in S$. Look at the two cases from the procedure. In case 1 we set $r := (x, \textsc{right}_{I(x)}(M(x)))$, $s := (y, \textsc{left}_{I(y)}(M(y)))$ because $z \lhd y$. The pair (r, s) is only added to $\mathcal{R}$ if there is no other $z' \in Z|_1$, $z \lhd z'$ such that $z' \lhd y$, or if $z' = \lhd y$ and there are nodes in $M(y)$ to the left of all nodes in $M(z')$. This corresponds to case (iv), (v), or (vi) in Proposition 5. In case 2 it is straightforward to verify that it corresponds to one of the cases (i), (ii), or (iii) in Proposition 5. □

Lemma 17 *Procedure* $\textsc{Fl}(\mathcal{X}, \alpha)$ *is correct.*

Proof. We only need to show that case 1 and 2 correctly computes $\textsc{Fl}$ on a micro tree. That the rest of the procedures is correct follows from case (iii) in Proposition 2 and the comments after the implementation.

That case 1 and 2 are correct follows from Proposition 2. Since we always call $\textsc{Deep}$ on the output from $\textsc{Fl}(\mathcal{X}, \alpha)$ there is no need to compute $\textsc{Fl}$ in the macro tree if N is nonempty. □

Lemma 18 *Procedure* $\textsc{Canonical}(\mathcal{X})$ *is correct.*

Proof. Follows immediately from the implementation of the procedure. □

5.6 Complexity of the Tree Inclusion Algorithm

For the running time of the macro-node list implementation observe that, given the data structure described in Section 5.2, all set procedures, except $\textsc{Fl}$, perform a single pass over the input using constant time at each step. Procedure $\textsc{Fl}(|\mathcal{X}|)$ uses $O(|\mathcal{X}|)$ time to compute $\mathcal{R}$ and U since each step takes constant time. Computing S takes time $O(n_T/\log n_T)$ and computing $\mathcal{S}$ takes time $O(|S|)$. Merging $\mathcal{R}$ and $\mathcal{S}$ takes time linear in the length of the two lists. It follows that $\textsc{Fl}$ runs in $O(n_T/\log n_T)$ time. To summarize we have shown that,

Lemma 19 *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.*

Next consider computing the deep occurrences of P in T using the procedure EMB of Section 3 and Lemma 19. Since each node $v \in V(P)$ contributes at most a constant number of calls to set procedures it follows immediately that,

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

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

References

[1] L. Alonso and R. Schott. On the tree inclusion problem. In *Proceedings of Mathematical Foundations of Computer Science*, pages 211–221, 1993.

[2] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming*, pages 270–280, 1997.

[3] Stephen Alstrup, Jacob Holm, and Mikkel Thorup. Maintaining center and median in dynamic trees. In *Scandinavian Workshop on Algorithm Theory*, pages 46–56, 2000.

[4] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problem. In *Proc. of Foundations of Computer Science (FOCS) 1998*, pages 534–543, 1998.

[5] Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 947–953. Society for Industrial and Applied Mathematics, 2002.

[6] Philip Bille. A survey on tree edit distance and related problems, *Submitted*, 2004.

[7] Weimin Chen. More efficient algorithm for ordered tree inclusion. *Journal of Algorithms*, 26:370–385, 1998.

[8] M. J. Chung. $O(n^{2.5})$ algorithm for the subgraph homeomorphism problem on trees. *J. of Algorithms*, 8(1):106–112, 1987.

[9] Richard Cole, Ramesh Hariharan, and Piotr Indyk. Tree pattern matching and subset matching in deterministic o(n log3 n)-time. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 245–254. Society for Industrial and Applied Mathematics, 1999.

[10] P. F. Dietz. Fully persistent arrays. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proceedings of the Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74, Berlin, 1989. Springer-Verlag.

[11] Moshe Dubiner, Zvi Galil, and Edith Magen. Faster tree pattern matching. In *Proceedings of the 31st IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 145–150, 1990.

[12] P. Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages. In *Proc. of the 4th European Symp. on Algorithms (ESA). Lecture Notes in Computer Science*, pages 107–120, 1996.

[13] Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *IEEE Symposium on Foundations of Computer Science*, pages 632–641, 1991.

[14] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.

[15] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[16] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the Association for Computing Machinery (JACM)*, 29(1):68–95, 1982.

[17] Pekka Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, University of Helsinki, Department of Computer Science, November 1992.

[18] Pekka Kilpeläinen and Heikki Mannila. Retrieval from hierarchical texts by partial patterns. In *Proceedings of the 16th Ann. Int. ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222. ACM Press, 1993.

[19] Pekka Kilpeläinen and Heikki Mannila. Ordered and unordered tree inclusion. *SIAM Journal of Computing*, 24:340–356, 1995.

[20] P.N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th annual European Symposium on Algorithms (ESA) 1998.*, pages 91–102. Springer-Verlag, 1998.

[21] Donald Erwin Knuth. *The Art of Computer Programming, Volume 1.* Addison Wesley, 1969.

[22] S. Rao Kosaraju. Efficient tree pattern matching. In *Proceedings of the 30th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 178–183, 1989.

[23] Heikki Mannila and K. J. Räihä. On query languages for the p-string data model. *Information Modelling and Knowledge Bases*, pages 469–482, 1990.

[24] Jiri Matoušek and R. Thomas. On the complexity of finding iso- and other morphisms for partial k-trees. *Discrete Mathematics*, 108:343–364, 1992.

[25] S. Muthukrishnan and Martin Müller. Time and space efficient method-lookup for object-oriented programs. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 42–51. Society for Industrial and Applied Mathematics, 1996.

[26] Thorsten Richter. A new algorithm for the ordered tree inclusion problem. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM), in Lecture Notes of Computer Science (LNCS), volume 1264*, pages 150–166. Springer, 1997.

[27] T. Schlieder and F. Naumann. Approximate tree embedding for querying XML data. In *ACM SIGIR Workshop On XML and Information Retrieval*, Athens, Greece, 2000.

[28] Torsten Schlieder and Holger Meuss. Querying and ranking XML documents. *J. Am. Soc. Inf. Sci. Technol.*, 53(6):489–503, 2002.

[29] R. Shamir and D. Tsur. Faster subtree isomorphism. *J. of Algorithms*, 33:267–280, 1999.

[30] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery (JACM)*, 26:422–433, 1979.

[31] A. Termier, M. Rousset, and M. Sebag. Treefinder: a first step towards XML data mining. In *IEEE International Conference on Data Mining (ICDM)*, 2002.

[32] Huai Yang, Li Lee, and Wynne Hsu. Finding hot query patterns over an xquery stream. *The VLDB Journal*, 13(4):318–332, 2004.

[33] Liang Huai Yang, Mong Li Lee, and Wynne Hsu. Efficient mining of XML query patterns for caching. In *Proceedings of the 29th VLDB Conference*, pages 69–80, 2003.

[34] P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree signatures for XML querying and navigation. In *LNCS 2824*, pages 149–163, 2003.

[35] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18:1245–1262, 1989.