Matching Subsequences in Trees

Philip Bille[1,*] and Inge Li Gørtz[2,**]

[1] IT University of Copenhagen
beetle@itu.dk
[2] Technical University of Denmark
ilg@imm.dtu.dk

Abstract. Given two rooted, labeled trees P and T the tree path subsequence problem is to determine which paths in P are subsequences of which paths in T. Here a path begins at the root and ends at a leaf. In this paper we propose this problem as a useful query primitive for XML data, and provide new algorithms improving the previously best known time and space bounds.

1   Introduction

We say that a tree is *labeled* if each node is assigned a character from an alphabet Σ. Given two sequences of labeled nodes p and t, we say that p is a *subsequence* of t, denoted $p \sqsubseteq t$, if p can be obtained by removing nodes from t. Given two rooted, labeled trees P and T the *tree path subsequence problem* (TPS) is to determine which paths in P are subsequences of which paths in T. Here a path begins at the root and ends at a leaf. That is, for each path p in P we must report all paths t in T such that $p \sqsubseteq t$.

This problem was introduced by Chen [3] who gave an algorithm using $O(\min(l_P n_T, n_P l_T + n_T))$ time and $O(l_P d_T)$ space. Here, n_S, l_S, and d_S denotes the number of nodes, number of leaves, and depth, respectively, of a tree S. Note that in the worst-case this is quadratic time and space. In this paper we show the following result:

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

Hence, if one of the trees has few leaves we match the previous time bounds, while improving the space to linear. The latter bound improves the worst-case time by a logarithmic factor whenever $\log n_P = O(n_T / \log n_T))$. Note that – in the worst-case – the number of pairs consisting of a path from P and a path T is $\Omega(n_P n_T)$, and therefore we need at least as many bits to report the solution to TPS. Hence, on a RAM with logarithmic word size our worst-case bound is optimal.
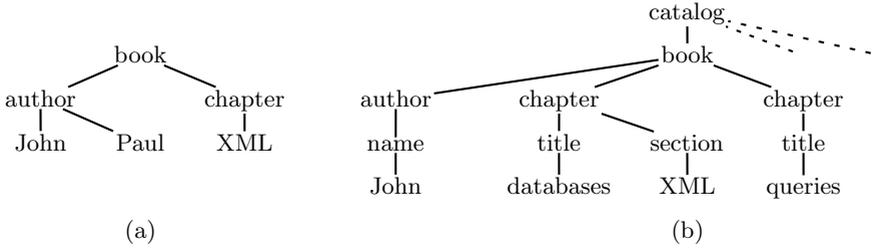
Fig. 1. (a) The trie of queries 1,2,3, or the tree for query 4. (b) A fragment of a catalog of books.

More importantly, all our algorithms use linear space, whereas the previous ones used quadratic space in the worst-case. For practical applications this makes it possible to solve TPS on larger trees and speed up the running time since more of the computation can be kept in main memory.

The first two time bounds are useful when the number of leaves in one of the trees has few leaves. In this case our contribution is the reduction to linear space. If, on the other hand, the number of leaves in both trees are proportional to the number of nodes in the tree the last time bound is the best. In this paper we present the first algorithm with subquadratic worst-case time and space bound.

Applications. We propose TPS as a useful query primitive for XML data. The key idea is that an XML document D may be viewed as a rooted, labeled tree. For example, suppose that we want to maintain a catalog of books for a bookstore. A fragment of a possible XML tree, denoted D, corresponding to the catalog is shown in Fig. 1(b). In addition to supporting full-text queries, such as find all documents containing the word "John", we can also use the tree structure of the catalog to ask more specific queries, such as the following examples:

1. Find all books written by John,
2. find all books written by Paul,
3. find all books with a chapter that has something to do with XML, or
4. find all books written by John and Paul with a chapter that has something to do with XML.

The queries 1,2, and 3 correspond to a *path query* on D, that is, compute which paths in D that contains a specific path as a subsequence. For instance, computing the paths in D that contain the path of three nodes labeled "book", "chapter", and "XML", respectively, effectively answers query 3. Most XML-query languages, such as XPath [4], support such queries.

Using a simple algorithm (a describtion of which we omit due to lack of space) a path query can be solved in linear time. Specifically, if q is a path consisting of n_q nodes, answering the path query on D takes $O(n_q + n_D)$ time. Hence, if we are given path queries $q_1, \ldots, q_k$ we can answer them in $O((n_{q_1} + \cdots + n_{q_k})n_D)$ time. If, however, the paths overlap we can do better by constructing the *trie*, Q, of $q_1, \ldots, q_k$. Answering all paths queries now correspond to solving TPS on

Q and D. As example the queries 1,2, and 3 form the trie shown in Fig. 1(a). Depending on the overlap between $q_1, \ldots, q_k$, n_Q is up to a linear factor smaller than $n_{q_1} + \cdots + n_{q_k}$.

Next consider query 4. This query cannot be answered by solving a TPS problem but is an instance of the *tree inclusion problem* (TI). Here we want to decide if P is *included* in T, that is, if P can be obtained from T by *deleting* nodes of T. Deleting a node y in T means making the children of y children of the parent of y and then removing y. It is straightforward to check that we can answer query 4 by deciding if the tree in Fig. 1(a) can be included in the tree in Fig. 1(b).

Recently, TI has been recognized as an important XML query primitive and has recieved considerable attention, see e.g., [9, 13, 12, 14, 10, 11]. Unfortunately, TI is NP-complete in general [8] and therefore the existing algorithms are based on heuristics. Observe that a necessary condition for P to included in T is that all paths in P are subsequences of paths in T. Hence, we can use TPS to quickly rule out trees that cannot be included T. We believe that in this way TPS can be used as an effective "filter" for many tree inclusion problems that occur in practice.

Technical Overview. Given two strings (or labeled paths) a and b, it is straightforward to determine if a is a subsequence of b by scanning the character from left to right in b. This uses $O(|a| + |b|)$ time. We can solve TPS by applying this algorithm to each of the pair of paths in P and T, however, this may use as much as $O(n_P n_T (n_P + n_T))$ time. Alternatively, Baeza-Yates [2] showed how to preprocess b in $O(|b| \log |b|)$ time such that testing whether a is a subsequence of b can be done in $O(|a| \log |b|)$ time. Using this data structure on each path in T we can solve the TPS problem, however, this may take as much as $O(n_T \log n_T + n_P^2 \log n_T)$. Hence, the availiable subsequence algorithms on strings does not provide an immediate solution.

Inspired by the work of Chen [3] we take another approach. We provide a framework for solving TPS. The main idea is to traverse T while maintaining a subset of nodes in P, called the *state*. When reaching a leaf z in T the state represents the paths in P that are a subsequences of the path from the root to z. At each step the state is updated using a simple procedure defined on subset of nodes. The result of Theorem 1 is obtained by taking the best of two algorithms based on our framework: The first one uses a simple data structure to maintain the state. This leads to an algorithm using $O(\min(l_P n_T, n_P l_T + n_T))$ time. At a high level this algorithm resembles the algorithm of Chen [3] and achieves the same running time. However, we improve the analysis of the algorithm and show a space bound of $O(n_P + n_T)$. This should be compared to the worst-case quadratic space bound of $O(l_P d_T)$ given by Chen [3]. Our second algorithm takes a different approach combining several techniques. Starting with a simple quadratic time and space algorithm, we show how to reduce the space to $O(n_P \log n_T)$ using a decomposition of T into disjoint paths. We then divide P into small subtrees of logarithmic size called *micro trees*. The micro trees are then preprocessed such that subsets of nodes in a micro tree can be maintained in constant time and space. Intuitively, this leads to a logarithmic improvement of the time and space bound.

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 $y \in V(T)$, $\text{depth}(y)$, is the number of edges on the path from y to $\text{root}(T)$ and the depth of T, denoted d_T, is the maximum depth of any node in T. The parent of y is denoted $\text{parent}(y)$. A node with no children is a leaf and otherwise it is an internal node. The number of leaves in T is denoted l_T. Let $T(y)$ denote the subtree of T rooted at a node $y \in V(T)$. If $z \in V(T(y))$ then y is an ancestor of z and if $z \in V(T(y))\backslash\{y\}$ then y is a proper ancestor of z. If y is a (proper) ancestor of z then z is a (proper) descendant of y. We say that T is *labeled* if each node y is a assigned a character, denoted $\text{label}(y)$, from an alphabet Σ. The path from y to $\text{root}(T)$, of nodes $\text{root}(T) = y_1, \ldots, y_k = y$ is denoted $\text{path}(y)$. Hence, we can formally state TPS as follows: Given two rooted tree P and T with leaves $x_1, \ldots, x_r$ and $y_1, \ldots, y_s$, respectively, determine all pairs (i, j) such that $\text{path}(x_i) \sqsubseteq \text{path}(y_j)$. For simplicity we will assume that leaves in P and T are always numbered as above and we identify each of the paths by the number of the corresponding leaf.

Throughout the paper we assume a standard RAM model of computation with logarithmic word size. We use a standard instruction set including bitwise boolean operations, shifts, addition, multiplication, etc.

2   A Framework for Solving TPS

In this section we present a simple general algorithm for the tree path subsequence problem. The key ingredient in our algorithm is the following procedure. For any $X \subseteq V(P)$ and $y \in V(T)$ define:

DOWN(X, y): Return the set CHILD$(\{x \in X \mid \text{label}(x) = \text{label}(y)\}) \cup \{x \in X \mid$
$\qquad\qquad \text{label}(x) \neq \text{label}(y)\}$.

The notation CHILD(X) denotes the set of children of X. Hence, DOWN(X, y) is the set consisting of nodes in X with a different label than y and the children of the nodes X with the same label as y. We will now show how to solve TPS using this procedure.

First assign a unique number in the range $\{1, \ldots, l_P\}$ to each leaf in P. Then, for each i, $1 \leq i \leq l_P$, add a *pseudo-leaf* $\perp_i$ as the single child of the ith leaf. All pseudo-leaves are assigned a special label $\beta \notin \Sigma$. The algorithm traverses T in a depth first order and computes at each node y the set X_y. We call this set the *state* at y. Initially, the state consists of $\{\text{root}(P)\}$. For $z \in \text{child}(y)$, the state X_z can be computed from state X_y as follows: $X_z = \text{DOWN}(X_y, z)$.

If z is a leaf we report the number of each pseudo-leaf in X_z as the paths in P that are subsequences of $\text{path}(z)$. See Fig. 2 for an example. To show the correctness of this approach we need the following lemma.

Lemma 1. *For any node $y \in V(T)$ the state X_y satisfies the following property: If $x \in X_y$ then $\text{path}(\text{parent}(x)) \sqsubseteq \text{path}(y)$.*
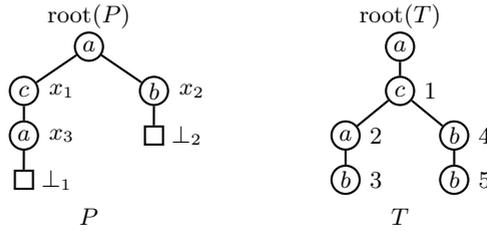
Fig. 2. Letters inside nodes are labels, and the identifier of each node is written outside the node. Initially we have $X = \{\text{root}(P)\}$. Since $\text{label}(\text{root}(P)) = a = \text{label}(\text{root}(T))$ we replace $\text{root}(P)$ with is children and get $X_{\text{root}(T)} = \{x_1, x_2\}$. Since $\text{label}(1) = \text{label}(x_1) \neq \text{label}(x_2)$ we get $X_1 = \{x_3, x_2\}$. Continuing this way we get $X_2 = \{\perp_1, x_2\}$, $X_3 = \{\perp_1, \perp_2\}$, $X_4 = \{x_3, \perp_2\}$, and $X_5 = \{x_3, \perp_2\}$. The nodes 3 and 5 are leaves of T and we thus report paths 1 and 2 after computing X_3 and path 2 after computing X_5.

Proof. By induction on the number of iterations of the procedure. Initially, $X = \{\text{root}(P)\}$ satisfies the property since $\text{root}(P)$ has no parent. Suppose that X_y is the current state and $z \in \text{child}(y)$ is the next node in the depth first traversal of T. By the induction hypothesis X_y satisfies the property, that is, for any $x \in X_y$, $\text{path}(\text{parent}(x)) \sqsubseteq \text{path}(y))$. Then, $X_z = \text{DOWN}(X_y, z) = \text{CHILD}(\{x \in X_y \mid \text{label}(x) = \text{label}(z)\}) \cup \{x \in X_y \mid \text{label}(x) \neq \text{label}(z)\}$.

Let x be a node in X_y. There are two cases. If $\text{label}(x) = \text{label}(z)$ then $\text{path}(x) \sqsubseteq \text{path}(z)$ since $\text{path}(\text{parent}(x)) \sqsubseteq \text{path}(y)$. Hence, for any child x' of x we have $\text{path}(\text{parent}(x')) \sqsubseteq \text{path}(z)$. On the other hand, if $\text{label}(x) \neq \text{label}(z)$ then $x \in X_z$. Since $y = \text{parent}(z)$ we have $\text{path}(y) \sqsubseteq \text{path}(z)$, and hence $\text{path}(\text{parent}(x)) \sqsubseteq \text{path}(y) \sqsubseteq \text{path}(z)$. $\square$

By the above lemma all paths reported at a leaf $z \in V(T)$ are subsequences of $\text{path}(z)$. The following lemma shows that the paths reported at a leaf $z \in V(T)$ are *exactly* the paths in P that are subsequences of $\text{path}(z)$.

Lemma 2. *Let z be a leaf in T and let $\perp_i$ be a pseudo-leaf in P. Then, $\perp_i \in X_z \Leftrightarrow \text{path}(\text{parent}(\perp_i)) \sqsubseteq \text{path}(z)$.*

Proof. It follows from Lemma 1 that $\perp_i \in X_z \Rightarrow \text{path}(\text{parent}(\perp_i)) \sqsubseteq \text{path}(z)$. It remains to show that $\text{path}(\text{parent}(\perp_i)) \sqsubseteq \text{path}(z) \Rightarrow \perp_i \in X_z$. Let $\text{path}(z) = z_1, \ldots, z_k$, where $z_1 = \text{root}(T)$ and $z_k = z$, and let $\text{path}(\text{parent}(\perp_i)) = y_1, \ldots, y_\ell$, where $y_1 = \text{root}(P)$ and $y_\ell = \text{parent}(\perp_i)$. Since $\text{path}(\text{parent}(\perp_i)) \sqsubseteq \text{path}(z)$ there are nodes $z_{j_i} = y_i$ for $1 \leq i \leq k$, such that (i) $j_i < j_{i+1}$ and (ii) there exists no node z_j with $\text{label}(z_j) = \text{label}(y_i)$, where $j_{i-1} < j < j_i$. Initially, $X = \{\text{root}(P)\}$. We have $\text{root}(P) \in X_{z_j}$ for all $j < j_1$, since z_{j_1} is the first node on $\text{path}(z)$ with label $\text{label}(\text{root}(P))$. When we get to z_{j_1}, $\text{root}(P)$ is removed from the state and y_2 is inserted. Similarly, y_i is in all states X_{z_j} for $j_{i-1} \leq j < j_i$. It follows that $\perp_i$ is in all states X_{z_j} where $j \geq j_\ell$ and thus $\perp_i \in X_{z_k} = X_z$. $\square$

The next lemma can be used to give an upper bound on the number of nodes in a state. The proof is omitted due to lack of space.

Lemma 3. *For any $y \in V(T)$ the state X_y has the following property: Let $x \in X_y$. Then no ancestor of x is in X_y.*

It follows from Lemma 3 that $|X_y| \leq l_P$ for any $y \in V(T)$. If we store the state in an unordered linked list each step of the depth-first traversal takes time $O(l_P)$ giving a total $O(l_P n_T)$ time algorithm. Since each state is of size at most l_P the space used is $O(n_P + l_P n_T)$. In the following sections we show how to improve these bounds.

3   A Simple Algorithm

In this section we consider a simple implementation of the above algorithm, which has running time $O(\min(l_P n_T, n_P l_T))$ and uses $O(n_P + n_T)$ space. We assume that the size of the alphabet is $n_T + n_P$ and each character in Σ is represented by an integer in the range $\{1, \ldots, n_T + n_P\}$. If this is not the case we can sort all characters in $V(P) \cup V(T)$ and replace each label by its rank in the sorted order. This does not change the solution to the problem, and assuming at least a logarithmic number of leaves in both trees it does not affect the running time. To get the space usage down to linear we will avoid saving all states. For this purpose we introduce the procedure UP, which reconstructs the state X_z from the state X_y, where $z = \text{parent}(y)$. We can thus save space as we only need to save the current state.

   We use the following data structure to represent the current state X_y: A *node dictionary* consists of two dictionaries denoted X^c and X^p. The dictionary X^c represents the node set corresponding to X_y, and the dictionary X^p represents the node set corresponding to the set $\{x \in X_z \mid x \notin X_y \text{ and } z \text{ is an ancestor of } y\}$. That is, X^c represents the nodes in the current state, and X^p represents the nodes that is in a state X_z, where z is an ancestor of y in T, but not in X_y. We will use X^p to reconstruct previous states. The dictionary X^c is indexed by Σ and X^p is indexed by $V(T)$. The subsets stored at each entry are represented by doubly-linked lists. Furthermore, each node in X^c maintains a pointer to its parent in X^p and each node x' in X^p stores a linked list of pointers to its children in X^p. With this representation the total size of the node dictoinary is $O(n_P + n_T)$.

   Next we show how to solve the tree path subsequence problem in our framework using the node dictionary representation. For simplicity, we add a node $\top$ to P as a the parent of $\text{root}(P)$. Initially, the X^p represents $\top$ and X^c represents $\text{root}(P)$. The DOWN and UP procedures are implemented as follows:

DOWN$((X^p, X^c), y)$:   1.   Set $X := X^c[\text{label}(y)]$ and $X^c[\text{label}(y)] := \emptyset$.
                                    2.   For each $x \in X$ do:
                                        (a)   Set $X^p[y] := X^p[y] \cup \{x\}$.
                                        (b)   For each $x' \in \text{child}(x)$ do:
                                            i.   Set $X^c[\text{label}(x')] := X^c[\text{label}(x')] \cup \{x\}$.
                                              ii.   Create pointers between x' and x.
                                    3.   Return (X^p, X^c).

UP$((X^p, X^c), y)$:   1.   Set $X := X^p[y]$ and $X^p[y] := \emptyset$.
2.   For each $x \in X$ do:
(a)   Set $X^c[\text{label}(x)] := X^c[\text{label}(x)] \cup \{x\}$.
(b)   For each $x' \in \text{child}(x)$ do:
i.   Remove pointers between x' and x.
ii.   Set $X^c[\text{label}(x')] := X^c[\text{label}(x')] \setminus \{x'\}$.
3.   Return (X^p, X^c).

The next lemma shows that UP correctly reconstructs the former state.

Lemma 4. *Let $X_z = (X^c, X^p)$ be a state computed at a node $z \in V(T)$, and let y be a child of z. Then, $X_z = \text{UP}(\text{DOWN}(X_z, y), y)$.*

Proof. Let $(X_1^c, X_1^p) = \text{DOWN}(X_z, y)$ and $(X_2^c, X_2^p) = \text{UP}((X_1^c, X_1^p), y)$. We will first show that $x \in X_z \Rightarrow x \in \text{UP}(\text{DOWN}(X_z, y), y)$.

Let x be a node in X^c. There are two cases. If $x \in X^c[\text{label}(y)]$, then it follows from the implementation of DOWN that $x \in X_1^p[y]$. By the implementation of UP, $x \in X_1^p[y]$ implies $x \in X_2^c$. If $x \notin X^c[\text{label}(y)]$ then $x \in X_1^c$. We need to show parent$(x) \notin X_1^p[y]$. This will imply $x \in X_2^c$, since the only nodes removed from X_1^c when computing X_2^c are the nodes with a parent in $X_1^p[y]$. Since y is unique it follows from the implementation of DOWN that parent$(x) \in X_1^p$ implies $x \in X^c[\text{label}(y)]$.

Let x be a node in X^p. Since y is unique we have $x \in X^p[y']$ for some $y' \neq y$. It follows immediately from the implementation of UP and DOWN that $X^p[y'] = X_1^p[y'] = X_2^p[y']$, when $y' \neq y$, and thus $X^p = X_2^p$.

We will now show $x \in \text{UP}(\text{DOWN}(X_z, y), y) \Rightarrow x \in X_z$. Let x be a node in X_2^c. There are two cases. If $x \notin X_1^c$ then it follows from the implementation of UP that $x \in X_1^p[y]$. By the implementation of DOWN, $x \in X_1^p[y]$ implies $x \in X^c[\text{label}(y)]$, i.e., $x \in X^c$. If $x \in X_1^c$ then by the implementation of UP, $x \in X_2^c$ implies parent$(x) \notin x_1^p[y]$. It follows from the implementation of DOWN that $x \in X^c$. Finally, let x be a node in X_2^p. As argued above $X^p = X_2^p$, and thus $x \in X^p$.   □

From the current state $X_y = (X^c, X^p)$ the next state X_z is computed as follows:

$$X_z = \begin{cases} \text{DOWN}(X_y, z) & \text{if } y = \text{parent}(z), \\ \text{UP}(X_y, y) & \text{if } z = \text{parent}(y). \end{cases}$$

The correctness of the algorithm follows from Lemma 2 and Lemma 4. We will now analyze the running time of the algorithm. The procedures DOWN and UP uses time linear in the size of the current state and the state computed. By Lemma 3 the size of each state is $O(l_P)$. Each step in the depth-first traversal thus takes time $O(l_P)$, which gives a total running time of $O(l_P n_T)$. On the other hand consider a path t in T. We will argue that the computation of all the states along the path takes total time $O(n_P + n_t)$, where n_T is the number of nodes in t. To show this we need the following lemma.

Lemma 5. *Let t be a path in T. During the computation of the states along the path t, any node $x \in V(P)$ is inserted into X^c at most once.*

Proof. Since t is a path we only need to consider the DOWN computations. The only way a node $x \in V(P)$ can be inserted into X^c is if $parent(x) \in X^c$. It thus follows from Lemma 3 that x can be inserted into X^c at most once. □

It follows from Lemma 5 that the computations of the all states when T is a path takes time $O(n_P + n_T)$. Consider a path-decomposition of T. A path-decomposition of T is a decomposition of T into disjoint paths. We can make such a path-decomposition of the tree T consisting of l_T paths. Since the running time of UP and DOWN both are linear in the size of the current and computed state it follows from Lemma 4 that we only need to consider the total cost of the DOWN computations on the paths in the path-decompostion. Thus, the algorithm uses time at most $\sum_{t \in T} O(n_p + n_t) = O(n_P l_T + n_T)$.

Next we consider the space used by the algorithm. Lemma 3 implies that $|X^c| \leq l_P$. Now consider the size of X^p. A node is inserted into X^p when it is removed from X^c. It is removed again when inserted into X^c again. Thus Lemma 5 implies $|X^p| \leq n_P$ at any time. The total space usage is thus $O(n_P + n_T)$. To summarize we have shown,

Theorem 2. *For trees P and T the tree path subsequence problem can be solved in $O(\min(l_P n_T, n_P l_T + n_T))$ time and $O(n_P + n_T)$ space.*

4   A Worst-Case Efficient Algorithm

In this section we consider the worst-case complexity of TPS and present an algorithm using subquadratic running time and linear space. The new algorithm works within our framework but does not use the UP procedure or the node dictionaries from the previous section.

Recall that using a simple linked list to represent the states we immediately get an algorithm using $O(n_P n_T)$ time and space. We first show how to modify the traversal of T and discard states along the way such that at most $O(\log n_T)$ states are stored at any step in the traversal. This improves the space to $O(n_P \log n_T)$. Secondly, we decompose P into small subtrees, called *micro trees*, of size $O(\log n_T)$. Each micro tree can be represented in a single word of memory and this way we can represent a state using only $O(\frac{n_P}{\log n_T})$ space. In total the space used to represent the $O(\log n_T)$ states is $O(\frac{n_P}{\log n_T} \cdot \log n_T) = O(n_P)$. Finally, we show how to preprocess P in linear time and space such that computing the new state can be done in constant time per micro tree. Intuitively, this achieves the $O(\log n_T)$ speedup.

Heavy Path Traversal. In this section we present the modified traversal of T. We first partition T into disjoint paths as follows. For each node $y \in V(T)$ let $size(y) = |V(T(y))|$. We classify each node as either *heavy* or *light* as follows. The root is light. For each internal node y we pick a child z of y of maximum

size among the children of y and classify z as heavy. The remaining children are light. An edge to a light child is a *light edge*, and an edge to a heavy child is a *heavy edge*. The heavy child of a node y is denoted heavy(y). Let lightdepth(y) denote the number of light edges on the path from y to root(T).

Lemma 6 (Harel and Tarjan [7]). *For any tree T and node $y \in V(T)$, lightdepth$(y) \leq \log n_T + O(1)$.*

Removing the light edges, T is partitioned into *heavy paths*. We traverse T according to the heavy paths using the following procedure. For node $y \in V(T)$ define:

VISIT(y): 1. If y is a leaf report all leaves in X_y and return.
 2. Else let $y_1, \ldots, y_k$ be the light children of y and let $z = $ heavy(y).
 3. For $i := 1$ to k do:
   (a) Compute $X_{y_i} := $ DOWN(X_y, y_i)
   (b) Compute VISIT(y_i).
 4. Compute $X_z := $ DOWN(X_y, z).
 5. Discard X_y and compute VISIT(z).

The procedure is called on the root node of T with the initial state $\{\text{root}(P)\}$. The traversal resembles a depth first traversal, however, at each step the light children are visited before the heavy child. We therefore call this a *heavy path traversal*. Furthermore, after the heavy child (and therefore all children) has been visited we discard X_y. At any step we have that before calling VISIT(y) the state X_y is availiable, and therefore the procedure is correct. We have the following property:

Lemma 7. *For any tree T the heavy path traversal stores at most $\log n_T + O(1)$ states.*

Proof. At any node $y \in V(T)$ we store at most one state for each of the light nodes on the path from y to root(T). Hence, by Lemma 6 the result follows. □

Using the heavy-path traversal immediately gives an $O(n_P \log n_T)$ space and $O(n_P n_T)$ time algorithm. In the following section we improve the time and space by an additional $O(\log n_T)$ factor.

Micro Tree Decomposition. A *micro tree* is a connected subgraph of P. A set of micro trees MS is a *micro tree decomposition* iff $V(P) = \cup_{M \in MS} V(M)$ and for any $M, M' \in MS$, $(V(M) \backslash \{\text{root}(M)\}) \cap (V(M') \backslash \{\text{root}(M')\}) = \emptyset$. Hence, two micro trees in a decomposition share at most one node and this node must be the root in at least one of the micro trees. If root$(M') \in V(M)$ then M is the *parent* of M' and M' is the *child* of M. A micro tree with no children is a *leaf* and a micro tree with no parent is a *root*. Note that we may have several root micro trees since they can overlap at the node root(P). We decompose P according to the following classic result:

Lemma 8 (Gabow and Tarjan [5]). *For any tree P and parameter $s > 1$, it is possible to build a micro tree decomposition MS of P in linear time such that $|MS| = O(n_P/s)$ and $|V(M)| \leq s$ for any $M \in MS$.*

Implementing the Algorithm. First decompose P according to Lemma 8 for a parameter s to be chosen later. Hence, each micro tree has at most s nodes and $|MS| = O(n_P/s)$. We represent the state X compactly using a bit vector for each micro tree. Specifically, for any micro tree M we store a bit vector $X_M = [b_1, \ldots, b_s]$, such that $X_M[i] = 1$ iff the ith node in a preorder traversal of M is in X. If $|V(M)| < s$ we leave the remaining values undefined. Later we choose $s = \Theta(\log n_T)$ such that each bit vector can be represented in a single word and the space used by the array is $O(n_P/\log n_T)$.

Next we define a $\textsc{Down}_M$ procedure on each micro tree $M \in MS$. Due to the overlap between micro trees the $\textsc{Down}_M$ procedure takes a bit b which will be used to propagate information between micro trees. For each micro tree $M \in MS$, bit vector X_M, bit b, and $y \in V(T)$ define:

$\textsc{Down}_M(X_M, b, y)$: Compute the state $X'_M := \textsc{Child}(\{x \in X_M \mid \text{label}(x) = \text{label}(y)\}) \cup \{x \in X_M \mid \text{label}(x) \neq \text{label}(y)\}$. If $b = 0$, return X'_M, else return $X'_M \cup \{\text{root}(M)\}$.

Later we will show how to implement $\textsc{Down}_M$ in constant time for $s = \Theta(\log n_T)$. First we show how to use $\textsc{Down}_M$ to simulate $\textsc{Down}$ on P. We define a recursive procedure $\textsc{Down}$ which traverse the hiearchy of micro trees. For micro tree M, state X, bit b, and $y \in V(T)$ define:

$\textsc{Down}(X, M, b, y)$: Let $M_1, \ldots, M_k$ be the children of M.
      1. Compute $X_M := \textsc{Down}_M(X_M, b, y)$.
      2. For $i := 1$ to k do:
        (a) Compute $\textsc{Down}(X, M_i, b_i, y)$, where $b_i = 1$ iff $\text{root}(M_i) \in X_M$.

Intuitively, the $\textsc{Down}$ procedure works in a top-down fashion using the b bit to propagate the new state of the root of micro tree. To solve the problem within our framework we initially construct the state representing $\{\text{root}(P)\}$. Then, at each step we call $\textsc{Down}(R_j, 0, y)$ on each root micro tree R_j.

Lemma 9. *The above algorithm correctly simulates the* $\textsc{Down}$ *procedure on P.*

Proof. Let X be the state and let $X' := \textsc{Down}(X, y)$. For simplicity, assume that there is only one root micro tree R. Since the root micro trees can only overlap at $\text{root}(P)$ it is straightforward to generalize the result to any number of roots. We show that if X is represented by bit vectors at each micro tree then calling $\textsc{Down}(R, 0, y)$ correctly produces the new state X'.

If R is the only micro tree then only line 1 is executed. Since $b = 0$ this produces the correct state by definition of $\textsc{Down}_M$. Otherwise, consider a micro tree M with children $M_1, \ldots, M_k$ and assume that $b = 1$ iff $\text{root}(M) \in X'$. Line 1 computes and stores the new state returned by $\textsc{Down}_M$. If $b = 0$ the correctness follows immediately. If $b = 1$ observe that $\textsc{Down}_M$ first computes the new state and then adds $\text{root}(M)$. Hence, in both cases the state of M is correctly computed. Line 2 recursively computes the new state of the children of M. □

If each micro tree has size at most s and DOWN_M can be computed in constant time it follows that the above algorithm solves TPS in $O(n_P/s)$ time. In the following section we show how to do this for $s = \Theta(\log n_T)$, while maintaining linear space.

Representing Micro Trees. In this section we show how to preprocess all micro trees $M \in MS$ such that DOWN_M can be computed in constant time. This preprocessing may be viewed as a "four russian trick" [1]. To achieve this in linear space we need the following auxiliary procedures on micro trees. For each micro tree M, bit vector X_M, and $\alpha \in \Sigma$ define:

$\text{CHILD}_M(X_M)$: Return the bit vector of nodes in M that are children of nodes in X_M.

$\text{EQ}_M(\alpha)$:   Return the bit vector of nodes in M labeled α.

By definition it follows that:

$$\text{DOWN}_M(X_M, b, y) = \begin{cases} \text{CHILD}_M(X_M \cap \text{EQ}_M(\text{label}(y))) \; \cup \\ \quad (X_M \backslash (X_M \cap \text{EQ}_M(\text{label}(y)))) & \text{if } b = 0, \\ \text{CHILD}_M(X_M \cap \text{EQ}_M(\text{label}(y))) \; \cup \\ \quad (X_M \backslash (X_M \cap \text{EQ}_M(\text{label}(y)))) \cup \{\text{root}(M)\} & \text{if } b = 1. \end{cases}$$

Recall that the bit vectors are represented in a single word. Hence, given CHILD_M and EQ_M we can compute DOWN_M using standard bit-operations in constant time.

Next we show how to efficiently implement the operations. For each micro tree $M \in MS$ we store the value $\text{EQ}_M(\alpha)$ in a hash table indexed by α. Since the total number of different characters in any $M \in MS$ is at most s, the hash table EQ_M contains at most s entries. Hence, the total number of entries in all hash tables is $O(n_P)$. Using perfect hashing we can thus represent EQ_M for all micro trees, $M \in MS$, in $O(n_P/s \cdot s) = O(n_P)$ space and $O(1)$ worst-case lookup time. The preprocessing time is expected $O(n_P)$ w.h.p.. To get a worst-case bound we use the deterministic dictionary of Hagerup et. al. [6] with $O(n_P \log n_P)$ worst-case preprocessing time.

Next consider implementing CHILD_M. Since this procedure is independent of the labeling of M it suffices to precompute it for all *structurally* different rooted trees of size at most s. The total number of such trees is less than 2^{2s} and the number of different states in each tree is at most 2^s. Therefore CHILD_M has to be computed for a total of $2^{2s} \cdot 2^s = 2^{3s}$ different inputs. For any given tree and any given state, the value of CHILD_M can be computed and encoded in $O(s)$ time. In total we can precompute all values of CHILD_M in $O(s2^{3s})$ time. Choosing the largest s such that $3s + \log s \leq n_T$ (hence $s = \Theta(\log n_T)$) this uses $O(n_T)$ time and space. Each of the inputs to CHILD_M are encoded in a single word such that we can look them up in constant time.

Finally, note that we also need to report the leaves of a state efficiently since this is needed in line 1 in the VISIT-procedure. To do this compute the state L

corresponding to all leaves in P. Clearly, the leaves of a state X can be computed by performing a bitwise AND of each pair of bit vectors in L and X. Computing L uses $O(n_P)$ time and the bitwise AND operation uses $O(n_T/s)$ time.

Combining the results, we decompose P, for s as described above, and compute all values of EQ_M and CHILD_M.

Then, we solve TPS using the heavy-path traversal. Since $s = \Theta(\log n_T)$ and from Lemmas 7 and 8 we have the following theorem:

Theorem 3. *For trees P and T the tree path subsequence problem can be solved in $O(\frac{n_P n_T}{\log n_T} + n_P \log n_P)$ time and $O(n_P + n_T)$ space.*

Combining the results of Theorems 2 and 3 this proves Theorem 1.

References

1. V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph (in russian). english translation in soviet math. dokl. 11, 1209-1210, 1975. *Dokl. Acad. Nauk.*, 194:487–488, 1970.
2. R. A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.
3. W. Chen. Multi-subsequence searching. *Inf. Process. Lett.*, 74(5-6):229–233, 2000.
4. J. Clark and S. DeRose. XML path language (XPath), avialiable as http://www.w3.org/TR/xpath, 1999.
5. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proc. of ACM Symp. on Theory of Computing*, pages 246–251, 1983.
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 Journal of Computing*, 13(2):338–355, 1984.
8. P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal of Computing*, 24:340–356, 1995.
9. T. Schlieder and H. Meuss. Querying and ranking XML documents. *J. Am. Soc. Inf. Sci. Technol.*, 53(6):489–503, 2002.
10. T. Schlieder and F. Naumann. Approximate tree embedding for querying XML data. In *ACM SIGIR Workshop On XML and Information Retrieval*, 2000.
11. A. Termier, M. Rousset, and M. Sebag. Treefinder: a first step towards XML data mining. In *IEEE International Conference on Data Mining (ICDM)*, 2002.
12. H. Yang, L. Lee, and W. Hsu. Finding hot query patterns over an xquery stream. *The VLDB Journal*, 13(4):318–332, 2004.
13. L. H. Yang, M. L. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. In *Proceedings of the 29th VLDB Conference*, pages 69–80, 2003.
14. P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree signatures for XML querying and navigation. In *LNCS 2824*, pages 149–163, 2003.