Bit-Parallel Tree Pattern Matching Algorithms for Unordered Labeled Trees *

Hiroaki Yamamoto[1] and Daichi Takenouchi[2]

[1] Department of Information Engineering, Shinshu University,
4-17-1 Wakasato, Nagano-shi, 380-8553 Japan.
yamamoto@cs.shinshu-u.ac.jp
[2] NTT Advanced Technology Corporation

Abstract. The following tree pattern matching problem is considered: Given two unordered labeled trees P and T, find all occurrences of P in T. Here P and T are called a *pattern tree* and a *target tree*, respectively. We first introduce a new problem called *the pseudo-tree pattern matching problem*. Then we show two efficient bit-parallel algorithms for the pseudo-tree pattern matching problem. One runs in $O(L_P \cdot n \cdot l \cdot \lceil \frac{h}{W} \rceil)$ time and $O(n \cdot l \cdot \lceil \frac{h}{W} \rceil)$ space, and another one runs in $O((L_P \cdot n + h \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ time and $O((n + h \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ space, where n is the number of nodes in T, h and l are the height of P and the number of leaves of P, respectively, and W is the length of a computer-word. The parameter L_P, called a *recursive level of* P, is defined to be the number of occurrences of the same label on a path from the root to a leaf. Hence we have $L_P \leq h$. Finally, we give an algorithm to extract all occurrences from pseud-occurrences in $O(n \cdot L_P \cdot l^{3/2})$ time and $O(n \cdot L_P \cdot l)$ space.

1 Introduction

In recent years, XML has been recognized as a common data format for data storages and exchanging data over the Internet, and has been widely spread. The tree pattern matching problem is a central part of XML query problems. In addition, this problem has a number of applications in the fields of computer science. Therefore, many researches have been done on developing an efficient tree pattern matching algorithm. The tree pattern matching problem is as follows: Given two labeled trees P and T, find all occurrences of P in T. Here P and T are called a pattern tree and a target tree, respectively. For this problem, ordered trees and unordered trees have been considered. An ordered tree is a tree such that the left-to-right order among siblings is significant. Hence, the order must usually be preserved in the tree pattern matching problem. On the other hand, an unordered tree is a tree such that any order among siblings is not defined, and hence the order is not significant in the tree pattern matching problem.

* This research was supported by the Ministry of Education, Sports, Culture, Science and Technology, Grant-in-Aid for Scientific Research (C).

For ordered trees, the tree pattern matching problem under the matching condition preserving parent-child relationship and the position of a child has been studied. The obvious algorithm runs in $O(n \cdot m)$ time, where n and m are the number of nodes of a target tree and a pattern tree, respectively. Hoffman and Donnell [7] proposed several algorithms. Dubiner, Galil and Magen [5] improved $O(n \cdot m)$ time and presented an $O(n \cdot \sqrt{m} \cdot polylog(m))$ time algorithm. Cole and Hariharan [3, 4] presented $O(n \log^2 m)$ time tree pattern matching algorithm by introducing a subset matching problem. Chauve [2] consider a more general matching condition, and has given an $O(n \cdot l)$ time algorithm, where l is the number of leaves of a pattern tree.

Several researches on unordered trees have also been done. Kilpeläinen and Mannila [9] studied the tree inclusion problem, which can be regarded as the unordered tree pattern matching problem with an ancestor-descendant relationship. They presented $O(n \cdot m)$ time algorithm for ordered trees and showed NP-completeness for unordered trees. Shamir and Tsur [11] gave an $O(n \cdot \frac{m^{3/2}}{\log m})$ time algorithm to solve the subtree isomorphism problem, in which unrooted and unlabeled trees are considered. This algorithm can solve the unordered tree pattern matching problem with a parent-child relationship. Furthermore several researches on XML query problems have been done (for example, see [6, 12, 13]) because the order among siblings is not significant in many practical applications for querying XML.

In this paper, we are concerned with a tree pattern matching problem on unordered labeled trees. We here introduce two new notions of *a pseudo-tree pattern matching problem* and *the recursive level of a labeled tree*. A pseudo-tree pattern matching problem is defined by allowing a many-to-one mapping from nodes of P to nodes of T. Note that tree pattern matching problems are normally defined based on a one-to-one mapping. Then the pseudo-tree pattern matching problem is to find out all pseud-occurrences of P in T. Götz, Koch and Martens [6] have studied on the tree homeomorphism problem for searching XML data. This problem can be regarded as a pseudo-tree pattern matching problem with ancestor-descendant relationship. They gave an $O(n \cdot m \cdot h)$ time algorithm. *The recursive level of a labeled tree* is defined to be the maximum number of occurrences of the same label over a path from the root to a leaf. In XML applications, a labeled tree with the recursive level 1, called a *non-recursive labeled tree*, is well studied (for example see [6, 12]). We present two efficient bit-parallel algorithms for solving the pseudo-tree pattern matching problem as follows. Here h and l are the height and the number of leaves of P, respectively, and W is the length of a computer-word, and L_P is the recursive level of P. Our algorithms make use of the Shift-OR technique which has been developed on the string matching problem [1].

- One algorithm runs in $O(L_P \cdot n \cdot l \cdot \lceil \frac{h}{W} \rceil)$ time and $O(n \cdot l \cdot \lceil \frac{h}{W} \rceil)$ space.
- Another one runs in $O((L_P \cdot n + h \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ time and $O((n + h \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ space. This algorithm consists of two parts *a preprocessing part* and *a matching part*. The preprocessing part, which generates bit-masks from a pattern tree P, takes $O(h \cdot 2^l \cdot \lceil \frac{h \cdot l}{W} \rceil)$ time and the matching part takes $O(L_P \cdot n \cdot \lceil \frac{h \cdot l}{W} \rceil)$ time.

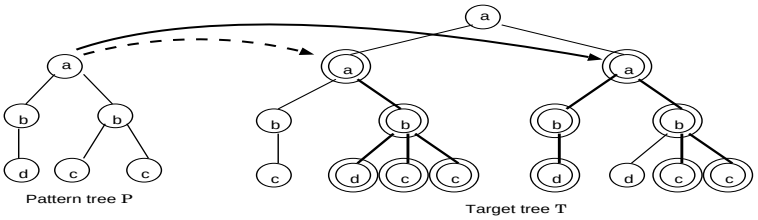

Fig. 1. A pseud-occurrence and an occurrence. The dotted arrow indicates a pseud-occurrence and the solid arrow indicates an occurrence (an exact occurrence).

In general, W is defined as $W = O(\log n)$ on conventional computing models. Hence, if h is at most $\log n$, then the first algorithm runs in $O(L_P \cdot n \cdot l)$ time, and if $h \cdot l$ is at most $\log n$, then the second algorithm runs in $O(L_P \cdot n)$ time. This time, if $L_P = O(1)$, then the second algorithm solves the pseudo-tree pattern matching problem in $O(n)$ time. Thus our algorithms run faster for pattern trees with small size.

Finally we give an algorithm to extract occurrences from pseud-occurrences for the tree pattern matching problem. If there are not any nodes with the same label among siblings in P, then a pseud-occurrence of P is identical to an occurrence of P, and hence the bit-parallel algorithms for the pseudo-tree pattern matching problem solve the tree pattern matching problem. If there are nodes with the same label among siblings in P, then a pseud-occurrence does not always become an occurrence. For this case, we can show an algorithm to obtain all occurrences of P from pseud-occurrences using an algorithm finding a maximum matching on bipartite graphs. Our algorithm runs in $O(n \cdot L_P \cdot l^{3/2})$ time and $O(n \cdot L_P \cdot l)$ space.

2 Tree Pattern Matching Problem and Related Definitions

Let Σ be an alphabet. Then we concentrate on a labeled tree such that each node of the tree is labeled by a symbol of Σ. Let T a labeled tree. For any node v of T, the children of node v are *siblings* of each other. If the order among siblings is significant, then the tree is said to be *ordered*; otherwise it is said to be *unordered*. The height of T is defined as follows. The depth of the root is defined to be 1. For any node v of T, the depth of v is defined to be the depth of the parent plus 1. Then the height of T is defined to be the maximum depth over all nodes of T. We introduce a notion of a *recursive level of* T. For any node v of T, *the recursive level of* v is defined to be the number of occurrences of the same symbol as v over the path from the root to v. The recursive level of T is defined to be the maximum recursive level over all nodes of T. In addition, for any $\sigma \in \Sigma$, we define *the recursive level of* σ to be the maximum recursive

level over all nodes with label σ. We use two orders when traversing over the nodes of T. One is *preorder*, which is recursively defined as follows: First the root of T is visited. Let $T_1, \ldots, T_t$ be subtrees rooted by children of the root in the left-to-right order. Then each T_j is visited in the order from T_1 to T_t. Another one is *postorder*, in which the leftmost leaf is first visited, and then each node is visited after having visited all the children of the node.

Let P and T be unordered labeled trees, which are called a *pattern tree* and a *target tree*, respectively. We first define a notion of pseud-occurrence of P in T and a pseudo-tree pattern matching problem.

Definition 1 (a pseud-occurrence). *We say that P* nearly *matches T at a node d of T if there is a mapping ϕ from nodes of P into nodes of T such that*

1. *the root of P is mapped to d,*
2. *for any node u of P, there is a node $\phi(u)$ of T such that the label of u is equal to the label of $\phi(u)$,*
3. *for any nodes u, v of P, u is the parent of v if and only if $\phi(u)$ is the parent of $\phi(v)$.*

We say that d is a pseud-occurrence of P in T.

We give an example of a pseud-occurrence in Fig.1. Note that two nodes with label b of P are mapped to one node of T, that is, the right-hand child of a node a. Thus, in the definition of a pseud-occurrence, a mapping ϕ is allowed to be many-to-one. *The pseudo-tree pattern matching problem* is to find out all pseud-occurrences of P in T. Next we define a tree pattern matching problem, which is defined based on a one-to-one mapping.

Definition 2 (an occurrence). *We say that P matches T at a node d of T if there is a one-to-one mapping ϕ from nodes of P into nodes of T such that*

1. *it satisfies three conditions of the pseud-occurrence,*
2. *for any nodes u, v of P, if $u \neq v$, then $\phi(u) \neq \phi(v)$.*

We say that d is an occurrence (or an exact occurrence) of P in T.

We give an example of an occurrence in Fig.1. Note that the mapping ϕ is required to be a one-to-one mapping. *The tree pattern matching problem* is to find out all occurrences of P in T. It is clear from the definitions that a pseud-occurrence implies an occurrence, but the reverse does not always hold. In this paper, we first discuss the pseudo-tree pattern matching problem and then discuss the tree pattern matching problem.

3 Algorithms for the Pseudo-Tree Pattern Matching Problem

In this section, we give a bit-parallel algorithm to find all pseud-occurrences of a pattern tree P in a target T. We make use of a Shift-OR technique on a

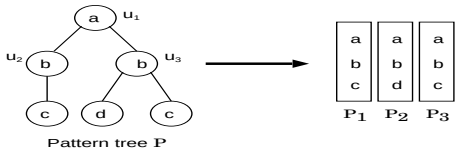

Fig. 2. Decomposition of a pattern tree P into path patterns P_1, P_2 and P_3

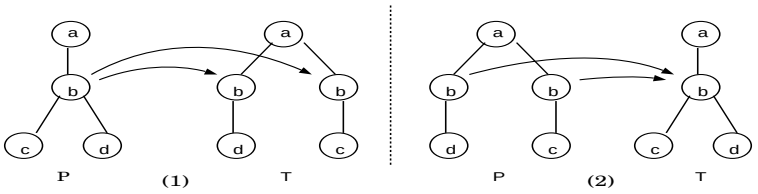

Fig. 3. two problems in BasicTreeMatch

string matching problem, which was developed by Baeza-Yates and Gonnet [1]. Let P_i be the string consisting of labels on a path from the root to a leaf in P. Then we call P_i *a path pattern* and denote by $|P_i|$ the length of path pattern P_i. We decompose P into path patterns for the Shift-OR technique. If P has l leaves, then P is decomposed into l path patterns $P_1, \ldots, P_l$. We say a node v of P appears on a path pattern P_i when the label of v appears on P_i. Fig. 2 illustrates an example of path patterns in which P is decomposed into three path patterns $P_1 = abc$, $P_2 = abd$ and $P_3 = abc$. We say that a path pattern occurs at a node d of T if the path pattern becomes just a prefix of the string consisting of labels on the path from node d to a leaf.

3.1 Bit-Masks for Path Patterns

To make use of the Shift-OR technique, we generate a bit-mask $B[P_i, \sigma]$ of h bits for every path pattern $P_i = p_1^i \cdots p_{|P_i|}^i$ and symbol σ, where h is the height of P. The value of $B[P_i, \sigma]$ is defined to be a bit sequence $b_1 \cdots b_h$ ($b_i \in \{0, 1\}$) such that for $j \leq |P_i|$, $b_j = 0$ if and only if $p_j^i = \sigma$, and for $|P_i| + 1 \leq j \leq h$, $b_j = 0$. For instance, bit-masks for three path patterns P_1, P_2, P_3 in Fig. 2 are defined as follows: $B[P_1, a] = 011$, $B[P_2, a] = 011$, $B[P_3, a] = 011$, $B[P_1, b] = 101$, $B[P_2, b] = 101$, $B[P_3, b] = 101$, $B[P_1, c] = 110$, $B[P_2, c] = 111$, $B[P_3, c] = 110$, $B[P_1, d] = 111$, $B[P_2, d] = 110$, $B[P_3, d] = 111$.

3.2 A Basic Algorithm

In this section, we give a simple algorithm $BasicTreeMatch(P, T)$ using the Shift-OR technique, which is given in Fig. 4. This algorithm finds all nodes in a target

Algorithm BasicTreeMatch(P, T)

Step 0. /* Initialization */
1. set bit-masks $B[P_i, c]$,
2. for all path patterns P_i, set the initial matching state $IM[P_i]$ to $1^{|P_i|}0^{h-|P_i|}$.

Visit each node d of T in postorder and do the following for d.

Step 1. /* This step computes the matching state $M[P_i, d]$ of d from matching states of the children. Let $d_1, \ldots, d_t$ be children of d. If d is a leaf, then we use $IM[P_i]$ instead. */
For all path pattern P_i,
1. if d a leaf, then $M[P_i, d] := IM[P_i]$;
otherwise $M[P_i, d] := M[P_i, d_1] \& \cdots \& M[P_i, d_t]$,
2. $M[P_i, d] := (M[P_i, d] \ << \ 1) \mid B[P_i, \sigma]$.

Step 2. /* This step determines whether all path patterns occur. */
1. $Temp := M[P_1, d] \mid \cdots \mid M[P_l, d]$,
2. the first bit of $Temp$, that is, the bit corresponding to the root of P, is 0, then return d.

Fig. 4. The algorithm *BasicTreeMatch*

tree T at which all path patterns of a pattern tree P occurs. We use an array $M[P_i, d]$ of bit-sequences, called a *matching state*, whose element is an h-bit sequence $b_1 \cdots b_h$. While traversing over nodes of T in postorder, we compute $M[P_i, d]$ for each node d of T. Let $P_i = p_1^i \cdots p_{|P_i|}^i$. This time, for any node d of T, $M[P_i, d] = b_1 \cdots b_h$ satisfies that for any $j \leq |P_i|$, $b_j = 0$ if and only if $p_j^i \cdots p_{|P_i|}^i$ occurs at node d. $M[P_i, d]$ is initially set to $1^{|P_i|}0^{h-|P_i|}$. The operation $M[P_i, d] << 1$ used in the algorithm denotes a shift operation which shifts a bit-sequence in $M[P_i, d]$ one bit to the left and sets the rightmost bit to 0. In addition, the operator "|" denotes a bitwise OR, and the operator "&" denotes a bitwise AND. *BasicTreeMatch* decomposes a pattern tree P into path patterns $P_1, \ldots, P_l$, and searches for a node of T at which all path patterns occur by traversing over T in postorder. The following proposition holds.

Proposition 1. *Let d be a node of T returned by* BasicTreeMatch. *Then all path patterns of P occur at the node d.*

The algorithm *BasicTreeMatch* completely cannot solve the tree pattern matching problem. That is, there are the following two problems: (1) one is that one node of P may be mapped to two or more nodes of T (see (1) in Fig.3); (2) another one is that two or more nodes of P may be mapped to one node of T (see (2) in Fig.3). *BasicTreeMatch* regards these cases as a match. The first case can be solved by introducing a notion of synchronization in a matching stage. Hence the pseudo-tree pattern matching problem can be solved. We will show this in the rest of this section. The second case will be discussed in Section 5.

3.3 A Pseudo-Tree Pattern Matching Algorithm

BasicTreeMatch regards two cases in Fig.3 as a match. In this section, we give a bit-parallel algorithm which does not regard the case (1) as a match. In (1) of Fig.3, a node b of P is mapped to two nodes with b of T. Thus two path patterns

Algorithm PseudoTreeMatch(P, T)

Step 0. /* Initialization */
1. set bit-masks $B[P_i, c]$,
2. for all path patterns P_i, set the initial matching state $IM[P_i]$ to $1^{|P_i|}0^{h-|P_i|}$,
3. for all path patterns P_i and all nodes u of P, set a synchronization bit-mask $Syn[P_i, u]$,
4. for all nodes u of P, set $\mathcal{P}_u = \{P_{i_1}, \ldots, P_{i_e}\}$ such that a path pattern P_{i_j} is in $\mathcal{P}_u$ if and only if u appears on P_{i_j}.

Visit each node d of T in postorder and compute the matching state $M[P_1, d], \ldots, M[P_l, d]$ for d. Here the label of d is σ.

Step 1. /* This step computes the matching state of d from matching states of the children. Let $d_1, \ldots, d_t$ be children of d. If d is a leaf, then we use $IM[P_i]$ instead. */
For all path pattern P_i,
1. if d a leaf, then $M[P_i, d] := IM[P_i]$;
otherwise $M[P_i, d] := M[P_i, d_1] \& \cdots \& M[P_i, d_t]$,
2. $M[P_i, d] := (M[P_i, d] \ << \ 1) \mid B[P_i, \sigma]$.

Step 2. /* Synchronization between path patterns */
For $lev = 1, \ldots, L_\sigma$, /* L_σ denotes the recursive level of symbol σ.*/
for all nodes u other than the root of P such that it has σ and the recursive level lev
1. for $P_i \in \mathcal{P}_u$, $SYN[P_i] := M[P_i, d]$ & $Syn[P_i, u]$,
2. $SynMask := SYN[P_{i_1}] \mid \cdots \mid SYN[P_{i_e}]$, where $\mathcal{P}_u = \{P_{i_1}, \ldots, P_{i_e}\}$,
3. for $P_i \in \mathcal{P}_u$, $M[P_i, d] := M[P_i, d] \mid SynMask$.

Step 3. /* This step determines whether a pseud-occurrence occurs.*/
1. $Temp := M[P_1, d] \mid \cdots \mid M[P_l, d]$,
2. the first bit of $Temp$, that is, the bit corresponding to the root of P, is 0, then return d as a pseud-occurrence.

Fig. 5. The algorithm *PseudoTreeMatch*

bd and bc are separated in T, and hence this does not satisfy the condition of the pseud-occurrence. Therefore, by solving the case (1), we can solve the pseudo-tree pattern matching problem. Our algorithm checks whether bd and bc occur at the same node in T, and if they occur at the same node, then the algorithm regards them as a match; otherwise does not so.

For this purpose, we introduce a new bit-mask called a *synchronization bit-mask* for any node and path pattern of P. Let u be any node of P with the height h. Then, for any path pattern P_i, a synchronization mask $Syn[P_i, u] = b_1 \cdots b_h$ is defined as follows: for any $1 \leq j \leq h$, if the node corresponding to b_j is just u, then $b_j = 1$; otherwise $b_j = 0$. Thus, in $Syn[P_1, u], \ldots, Syn[P_l, u]$, only the bits corresponding to the node u are set to 1; the other bits are set to 0. For instance, we show synchronization bit-masks for the pattern tree given in Fig. 2. For nodes u_1, u_2 and u_3, we have the following: $Syn[P_1, u_1] = 100$, $Syn[P_2, u_1] = 100$, $Syn[P_3, u_1] = 100$, and $Syn[P_1, u_2] = 010$, $Syn[P_2, u_2] = 000$, $Syn[P_3, u_2] = 000$, and $Syn[P_1, u_3] = 000$, $Syn[P_2, u_3] = 010$, $Syn[P_3, u_3] = 010$.

The algorithm *PseudoTreeMatch* in given Fig. 5 is constructed by adding the synchronization stage of Step 2 to *BasicTreeMatch*; it can find out all pseud-occurrences of P in T. From Proposition 1, we know that *BasicTreeMatch* finds out all nodes in T at which all path patterns occur. We explain how Step 2 works. Let u be a node of P and the depth is j. Then, for any P_i and any node d of T, the j-th bit b_j of $M[P_i, d]$ corresponds to u. Step 2 checks out the bit b_j of $M[P_i, d]$ for all $P_i \in \mathcal{P}_u$, and if the bit b_j of at least one $M[P_i, d]$ is 1, then

$\mathbf{M[P_1,d]}$	$\mathbf{M[P_2,d]}$	. . . . .	$\mathbf{M[P_l,d]}$

Fig. 6. A packed matching state $M[d]$

B[a]	011	011	011
B[b]	101	101	101
B[c]	111	110	110
B[d]	110	111	111

Fig. 7. Packed bit-masks of Fig.2

the bits b_j of all $M[P_i, d]$ are set to 1. To do this, we first check out the value of b_j in (a) of Step 2 using $Syn[P_i, u]$. If b_j of $M[P_i, d]$ is 1, then the j-th bit of $SYN[P_i]$ becomes 1. Hence if there is at least one $SYN[P_i]$ such that the j-th bit is 1, then the j-th bit of *SynMask* becomes 1 in (b) of Step 2. Finally, in this case, for all $P_i \in \mathcal{P}_u$, the j-bit b_j of $M[P_i, d]$ is set to 1 in (c) of Step 2. We have the following theorem.

Theorem 1. *The algorithm* PseudoTreeMatch *finds all pseud-occurrences of P in $O(L_P \cdot n \cdot l \cdot \lceil \frac{h}{W} \rceil)$ time and $O(n \cdot l \cdot \lceil \frac{h}{W} \rceil)$ space, where n is the number of nodes of T, L_P, h, and l are the recursive level, the height, and the number of leaves of P, respectively, and W is the length of a computer-word.*

3.4 Improving the Algorithm by Packing Bit-Sequences

Let n be the number of nodes of T, and let l and h be the number of leaves and height of P, respectively. The algorithm *PseudoTreeMatch* is checking a matching on each path pattern. Therefore it requires at least $n \times l$ time because there are l path patterns. In this section, we improve this matching process by packing path patterns into computer-words. This allows us to carry out matching processes on path patterns simultaneously. We give the improved algorithm *FastPseudoTreeMatch* in Fig. 8. In the algorithm, we pack matching states $M[P_1, d], \ldots, M[P_l, d]$ into one word $M[d]$ as in Fig. 6 (if the packed bit-sequence is long, then multiple words are used). We denote by $(M[P_1, d], \ldots, M[P_l, d])$ such a packed bit-sequence $M[d]$. Similarly, we also pack $B[P_1, \sigma], \ldots, B[P_l, \sigma]$ into $B[\sigma] = (B[P_1, \sigma], \ldots, B[P_l, \sigma])$ for each symbol σ as in Fig. 7. By these packing, we can simultaneously compute the matching state of each node in Step 1. Let $h_i = |P_i|$. We here use an h_i-bit sequence for bit-sequences such as $M[P_i, d]$ and $B[P_i, \sigma]$ to make as compact a packed bit-sequence as possible. In addition, we would like to carry out the synchronization task of Step 2 in *PseudoTreeMatch* simultaneously. To do this, we pack synchronization bit-masks into $PSyn[\sigma, lev]$ as follows, where σ is a symbol and lev is a recursive

Algorithm FastPseudoTreeMatch(P, T)

Step 0. /* Initialization */
1. compute bit-masks $B[P_i, \sigma]$, and set $B[c] := (B[P_1, \sigma], \ldots, B[P_l, \sigma])$ for each symbol σ in P,
2. for all path patterns P_i, set the initial matching state $IM[P_i]$ to 1^{h_i}, and then set the packed initial matching state $IM := (IM[P_1], \ldots, IM[P_l])$,
3. for all path patterns P_i and all nodes u of P, compute a synchronization bit-mask $Syn[P_i, u]$,
4. for all symbols σ and recursive level lev, set the packed synchronization bit-mask $PSyn[\sigma, lev] = (PSyn_1, \ldots, PSyn_l)$, where if $u_j \in N_{(\sigma,lev)}$ appears on P_i, then $PSyn_i = Syn[P_i, u_j]$; otherwise $PSyn_i = 0^{h_i}$.
5. *SetPSynMask*(P), /* set $PSynMask[SYN, lev]$ for at most $h2^l$ distinct values of SYN and a recursive level lev, */
6. set $ZMask := (1^{h_1-1}0, \ldots, 1^{h_l-1}0)$ and $AccCheck := (01^{h_1-1}, \ldots, 01^{h_l-1})$.

Visit each node d of T in postorder and compute the matching state of d as follows.

Step 1. /* Computing the matching state of d from the matching states of the children. Let $d_1, \ldots, d_t$ be children of d. The label of d is σ. */
1. If d is a leaf, then $M[d] := IM$; otherwise $M[d] := M[d_1] \& \cdots \& M[d_t]$,
2. $M[d] := ((M[d] << 1)\ \&\ ZMask) \mid B[\sigma]$.

Step 2. /* Synchronization between path patterns */
For $lev = 1, \ldots, L_\sigma$ do /* L_σ denotes the recursive level of σ. */
1. $SYN := M[d]\ \&\ PSyn[\sigma, lev]$,
2. $M[d] := M[d] \mid PSynMask[SYN, lev]$.

Step 3. /* This step determines whether a pseudo-match occurs. */
1. $Acc := M[d] \mid AccCheck$,
2. if $Acc = AccCheck$, then return d as a pseud-occurrence.

Fig. 8. The algorithm *FastPseudoTreeMatch*

level. We classify nodes of P into subsets $N_{(\sigma,lev)}$ such that $N_{(\sigma,lev)}$ consists of all nodes which are labeled by the symbol σ and have the recursive level lev. Let $N_{(\sigma,lev)} = \{u_1, \ldots, u_s\}$ for any symbol σ and any recursive level lev. Then we define $PSyn[\sigma, lev] = (PSyn_1, \ldots, PSyn_l)$, where if a node u_j $(1 \leq j \leq s)$ appears on P_i, then $PSyn_i = Syn[P_i, u_j]$; otherwise $PSyn_i = 0^{h_i}$.

We introduce an array $PSynMask[SYN, lev]$ of bit-sequences to reflect the result of a synchronization to a matching state, where $SYN = (SYN_1, \ldots, SYN_l)$ and each SYN_i corresponds to $SYN[P_i]$ in *PseudoTreeMatch*. Let us define $\mathcal{P}_u$ to be the set of path patterns P_i such that node u appears on P_i. Then, for any nodes $u, v \in N_{(\sigma,lev)}$, we have $\mathcal{P}_u \cap \mathcal{P}_v = \emptyset$. Hence we can represent $SYN[P_i]$ for all nodes in $N_{(\sigma,lev)}$ by SYN. The algorithm *FastPseudoTreeMatch* computes SYN in one step, while *PseudoTreeMatch* compute $SYN[P_i]$ for each node of $N_{(\sigma,lev)}$. The value of $PSynMask[SYN, lev]$ is defined to be $(PSynMask_1, \ldots, PSynMask_l)$, where each $PSynMask_i$ corresponds to *SynMask* computed for $\mathcal{P}_u$ corresponding to $u \in N_{(\sigma,lev)}$. Hence we can update a matching state $M[d]$ using $PSynMask[SYN, lev]$ in the same way as Step 2 of *PseudoTreeMatch*. *FastPseudoTreeMatch* carries out this task in one step at 2 of Step 2. We compute $PSynMask[SYN, lev]$ in Step 0 by the procedure *SetPSynMask*(P) in given Fig. 9. In *SetPSynMask*(P), $\mathcal{K}_{e^i_j}$ is defined to be a bit sequence $(K_1, \ldots, K_l)$ such that only the $d(u_i)$-th bit from the leftmost bit of $K_{e^i_j}$ is 1 and all other bits are 0, where $\mathcal{P}_{u_i} = \{P_{e^i_1}, \ldots, P_{e^i_{t_i}}\}$ for any $u_i \in N_{(\sigma,lev)}$ and $d(u_i)$ is the depth of

Procedure *SetPSynMask*(P)
For all symbols σ which occurs in P, do the following:

Step 1. /* set $SubMask[SYN, lev]$. */
1. for $lev = 1, \ldots, L_P$ do
2. for $u_i \in \{u_1, \ldots, u_s\}$ $(= N_{(\sigma,lev)})$ other than the root of P, do
3. $TMask := (TMask_1, \ldots, TMask_l)$, where $TMask_k = Syn[P_k, u_i]$,
4. for $SYN = \mathcal{K}_{e^i_1}, \ldots, \mathcal{K}_{e^i_{t_i}}$ do
5. $SubMask[SYN, lev] := TMask$,
6. end-for
7. end-for
8. end-for

Step 2. /* compute $PSynMask[SYN, lev]$. */
1. for $lev = 1, \ldots, L_P$ do
2. $IX := 0$, $Val[0] := 0$ and $PSynMask[0, lev] := (0^{h_1}, \ldots, 0^{h_l})$,
3. for $k_1 = \mathcal{K}_{e^1_1}, \ldots, \mathcal{K}_{e^1_{t_1}}, \ldots, \mathcal{K}_{e^s_1}, \ldots, \mathcal{K}_{e^s_{t_s}}$ do
4. $t := IX$,
5. for $k_2 := 0, \ldots, t$ do
6. $PSynMask[k_1 + Val[k_2], lev] := PSynMask[Val[k_2], lev] \mid SubMask[k_1, lev]$,
7. $IX := IX + 1$,
8. $Val[IX] := k_1 + Val[k_2]$,
9. end-for
10. end-for
11. end-for

Fig. 9. The procedure *SetPSynMask*.

Algorithm ExactTreeMatch(P**,** T**)**

Step 1. Do $PseudoTreeMatch(P, T)$ or $FastPseudoTreeMatch(P, T)$.
Step 2. For all pseud-occurrences d of P in T do
if $CheckMatch(\{v_P\}, d)$ returns $\{v_P\}$, then return d as an exact occurrence, where v_P is the root of P.

Fig. 10. The algorithm *ExactTreeMatch*

u_i. Furthermore we make use of two special bit-masks, *ZMask* and *AccCheck*. *ZMask* is set to $(1^{h_1-1}0, \ldots, 1^{h_l-1}0)$ and is used for clearing the rightmost bit of each matching state in a packed bit-sequence. We need such a clearing process because a shift operation sets the rightmost bit to 0. *AccCheck* is set to $(01^{h_1-1}, \ldots, 01^{h_l-1})$ and is used for checking whether or not a pseud-occurrence occurs. We have the following theorem.

Theorem 2. FastPseudoTreeMatch *can find out all pseud-occurrences of* P *in* T *in* $O((L_P \cdot n + h \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ *time and* $O((n + h \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ *space.*

Let m be the number of nodes in P. Then we have $h \cdot 2^l \leq 2^m$. Hence if $m = \log n$, then *FastPseudoTreeMatch* runs in $O(L_P \cdot n \cdot \lceil \frac{h \cdot l}{W} \rceil)$ time and space. Furthermore, if $h \cdot l = O(W)$ and $L_P = O(1)$, then it runs in $O(n)$ time.

4 A Tree Pattern Matching Algorithm

In the previous section, we have given an algorithm to find all pseud-occurrences of P in T. A pseud-occurrence allows a mapping to map multiple nodes of P

Function CheckMatch(G, d)

Step 1. $Match := \emptyset$,
Step 2. for all $g \in G$ do
1. if g is a leaf of P, then add g to $Match$;
2. otherwise do the following:
   (a) classify children $d_1, \ldots, d_t$ of d into groups $D_{a_1}, \ldots, D_{a_r}$ such that D_{a_j} $(1 \leq j \leq r)$ consists of all nodes with label a_j, and similarly classify children $g_1, \ldots, g_{t'}$ of g into groups $F_{a_1}, \ldots, F_{a_r}$ in the same way.
   (b) for all groups $D_{a_j} = D_{a_1}, \ldots, D_{a_r}$ do
      i. $R := \emptyset$,
      ii. for all nodes $d_k \in D_{a_j}$ do
         A. compute the set G_k^j consisting of all nodes g' of P such that g' is a child of g and d_k becomes a pseud-occurrence of the subtree rooted by node g' using the matching state $M[d_k]$ of d_k.
         B. $F_k^j := CheckMatch(G_k^j, d_k)$,
         C. add all pairs (g', d_k) with $g' \in F_k^j$ to R,
      iii. if $ExistMap(F_{a_j}, D_{a_j}, R) = false$, then go to next node of G.
   (c) add g to $Match$,
Step 3. return $Match$.

Fig. 11. The function *CheckMatch*

to one node of T, but the condition of an occurrence of P does not. Therefore, to find out all occurrences of P, we must check whether a pseud-occurrences satisfies the condition of an occurrence or not.

4.1 A Special Case

Here let us consider a special case; for any node v of a pattern tree P, all children of v have distinct labels. If P is the case, then pseud-occurrences become occurrences of P. Hence, we have the following theorem.

Theorem 3. *Let P be a pattern tree such that for any node of P, labels of any two children of the node are distinct. Then the algorithms* PseudoTreeMatch *and* FastPseudoTreeMatch *can find out all occurrences of P in T.*

4.2 A General Case

Let us consider a general case, that is, there are siblings in P such that they have the same label. The most difficult problem is that two or more nodes of P may be mapped to one node of T. We extract occurrences from pseud-occurrences by checking whether or not there is a one-to-one mapping. As in [11], the algorithm is designed using an algorithm finding a maximum matching on bipartite graphs. The algorithm *ExactTreeMatch* in Fig.10 checks whether a pseud-occurrence of P in T is an occurrence of P using the function *CheckMatch* given in Fig.11.

Given a node d in T and a subset G of nodes of P such that d becomes a pseud-occurrence of the subtree rooted by node $g \in G$, the function $CheckMatch(G, d)$ returns the subset $Match$ of G such that d becomes an occurrence. *CheckMatch* recursively checks whether a pseud-occurrence satisfies a one-to-one mapping in

(b) of Step 2. The function *ExistMap*(F_{a_j}, D_{a_j}, R) returns *true* if there is a subset R' of R such that (1) for any $g \in F_{a_j}$, there is $d \in D_{a_j}$ with $(g, d) \in R'$, and (2) for any $(g_1, d_1), (g_2, d_2) \in R'$, if $g_1 \neq g_2$, then $d_1 \neq d_2$; otherwise returns *false*. We can view (F_{a_j}, D_{a_j}, R) as a bipartite graph having the vertex set $F_{a_j} \cup D_{a_j}$ and the edge set R. This time, *ExistMap*(F_{a_j}, D_{a_j}, R) can be implemented using an algorithm for a maximum matching on bipartite graphs. For a pseud-occurrence d of P in T, let T_d be the subtree of T such that the root is d and the other nodes consists of all descendants of d which are at most h away from d. Then we define N_p to be $\sum_d |T_d|$, where d takes all pseud-occurrences of P and $|T_d|$ denotes the number of nodes in T_d. Then if we use the algorithm by Hopcroft and Karp [8], we have the following theorem. Here note that since we have $N_p \leq n \cdot L_P$ and $L_p \leq h$, the algorithm runs in $O(n \cdot h \cdot l^{3/2})$ time and $O(n \cdot h \cdot l)$ space in the worst case.

Theorem 4. *The algorithm* ExactTreeMatch *can find out all occurrences of P in T in $O(N_P \cdot l^{3/2})$ time and $O(N_P \cdot l)$ space plus the complexity of* PseudoTreeMatch *or* FastPseudoTreeMatch.

Acknowledgments. We are grateful to anonymous referees for many variable comments, which helped to improve algorithms and the presentation.

References

1. R. Baeza-Yates and G.H. Gonnet, A New Approach to Text Searching, Communications of the ACM, 35, 10(1992) 74–82.
2. C. Chauve, Tree pattern matching with a more general notion of occurrence of the pattern, Information Processing Letters, 82(2001) 197–201.
3. R. Cole and R. Hariharan, Verifying Candidate Matches in Sparse and Wildcard Matching , Proc. of the 34th ACM STOC, (2002) 592–601.
4. R. Cole and R. Hariharan, Tree Pattern Matching to Subset Matching in Linear Time, SIAM J. Comput., 32, 4(2003) 1056–1066.
5. M. Dubiner, Z. Galil, and E. Magen, Faster Tree Pattern Matching, Journal of the ACM, 41, 2(1994) 205–213.
6. M. Götz, C. Koch and W. Martens, Efficient Algorithms for the Tree Homeomorphism Problem, Proc. of the DBLP 2007, LNCS 4797, (2007) 17–31.
7. C.M. Hoffman and M.J. O'Donnell, Pattern matching in trees, Journal of the ACM, 29, 1(1982) 68–95.
8. J.E. Hopcroft and R.M. Karp, An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs, SIAM J. Comput., 2, 4(1973) 225–231.
9. P. Kilpeläinen and H. Mannila, Ordered and Unordered Tree Inclusion, SIAM J. Comput., 24, 2(1995) 340–356.
10. J.V. Leeuweno, Graph Algorithms, In J.V. Leeuwen, ed. Handbook of Theoretical Computer Science, Elsevier Science Pub., 1990.
11. R. Shamir and D. Tsur, Faster Subtree Isomorphism, J. of Algorithms, 33(1999) 267–280.
12. J.T. Yao, M. Zhang, A Fast Tree Pattern Matching Algorithm for XML Query, Proc. of the WI'04, (2004) 235–241.
13. P. Zezula, F. Mandreoli, and R. Martoglia, Tree Signatures and Unordered XML Pattern Matching, Proc. of the SOFSEM'04, LNCS 2932, (2004) 122–139.