Vol.2010-AL-131 No.4
2010/9/22

拡張擬似木パターンマッチング問題に対するビット並列アルゴリズム

山本博章[†1]　宮崎敬[†2]

本論文では, 無順序木に対し，拡張擬似木パターン照合問題という木パターン照合問題の一種について考える。一般に，木パターン照合問題とは，パターン木 P とターゲット木 T が与えられたとき，T の中で P に一致する部分をすべて見つける問題である。拡張擬似木パターン照合問題は，先祖・子孫関係のみに着目した問題である．この問題は XML データの検索において重要な役割を演じている．我々はこの問題に対し，文字列上のビット並列法を利用した効率的なアルゴリズムを与える。

Bit-Parallel Algorithms for Extended Pseudo-tree Pattern Matching Problem

Hiroaki Yamamoto[†1] and Takashi Miyazaki[†2]

In this paper, we are concerned with a special case of the tree pattern matching problem for XML query evaluation, which is called extended pseudo-tree pattern matching problem. This problem focuses on only ancestor-descendant relationship. Then we show efficient bit-parallel algorithms for this pattern matching problem. Our algorithms runs faster than the existing algorithms for pattern trees of small size.

1. はじめに

近年，XML(eXtended Markup Language) はデータ保存やインターネット上でのデータ交換のための共通のフォーマットとして広く認識され使われるようになってきた。木パターン照合問題は，XML データの検索における中心的な役割を演じている．木パターン照合問題とは，2つのラベル付き木 P と T が与えられたとき，T の中で P に一致する部分をすべて見つける問題である．ここで，P と T をそれぞれパターン木，ターゲット木と呼ぶ．この問題に対しては，木が順序木の場合および無順序木の場合 2 つの場合について研究されてきた．順序木とは，兄弟間に順番が定義された木のことであり，P と T の照合において兄弟間の順番を維持しなければならない．順序木に対するアルゴリズムがいくつか開発されている[3)–6),9)]．一方，無順序木は兄弟間の順番がないものである．したがって，照合において兄弟間の順番を維持しなくてもよい．本論文は無順序木に対するアルゴリズムを扱う．

無順序木に対しても，いくつかの研究が行なわれている．Kilpeläinen と Mannila[10)] は木包含問題について研究をした．これは先祖・子孫関係の下での木パターン照合問題と見ることができる．彼らは，順序木に対しては，$O(n \cdot m)$ 時間アルゴリズムを示し，無順序木に対しては NP 完全性を示した．Shamir と Tsur[12)] は部分木同型問題を解くための $O(n \cdot \frac{m^{3/2}}{\log m})$ 時間アルゴリズムを与えた．この問題では，根およびラベルのない無順序木を考えており親子関係を維持する無順序木パターン照合問題を解くことができる．さらに，XML の検索に関連して無順序木のパターン照合に対する研究も行なわれている[7),8),14),15)]．これは，XML 検索の応用において，兄弟間の順番が重要でない場合が多いからである．

本論文では，無順序木に対するパターン照合問題の一種である拡張擬似木パターン照合問題について議論する．この問題は，パターン照合において，先祖・子孫関係のみに注目した問題で，XML 検索において重要な役割を演じている．この問題に関する研究は，文献 2), 7), 8) で見ることができる．Götz, Koch と Martens[7),8)] は，XML 検索に関連し，この問題を研究し，$O(n \cdot m \cdot \mathit{height}(P))$ 時間かつ $O(\mathit{branch}(T) \cdot \mathit{height}(T))$ 領域のアルゴリズムを与えた．ここで，$\mathit{height}(P)$ は P の高さを示し，$\mathit{branch}(T)$ は T の各ノードから分岐する枝の数の中で最大値を示す．Yamamoto と Takenouchi[13)] は親子関係のみを維持する擬似木パターン照合問題というものを導入し，そこから木パターン照合問題を解くための効率的なビット並列アルゴリズムを開発した．本論文で扱う拡張擬似木パターン照合問題は，この擬似木パターン照合問題を，先祖・子孫関係に拡張したものである．この問題を定義するに当たって，我々は，パターン木を記述するために 2 種類の枝を導入する．一つは，親子枝（pc-edge）と呼ばれるものであり，もう一つは，先祖・子孫枝（ad-edge）と呼ばれるものである．これらはちょうど，XML 検索で使われる子軸（/-軸），子孫軸（//-軸）に対応する．擬似木パターン照合問題は親子枝のみ許された問題である．

†1 信州大学工学部
Faculty of Engineering, Shinshu University
†2 長野工業高等専門学校
Nagano National College of Technology

Vol.2010-AL-131 No.4
2010/9/22

	time	space
Algorithm-1	$O(n \cdot L_P \cdot l \cdot \lceil \frac{h}{W} \rceil)$	$O(branch(T) \cdot l \cdot \lceil \frac{h}{W} \rceil)$
Algorithm-2	$O((n + 2^{h \cdot l}) \cdot \lceil \frac{h \cdot l}{W} \rceil)$	$O((branch(T) + 2^{h \cdot l}) \cdot \lceil \frac{h \cdot l}{W} \rceil)$
Algorithm-3	$O((n \cdot L_P \cdot l + h \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$	$O((branch(T) + \alpha_P \cdot L_P \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$
Götz et al.[7),8)]	$O(n \cdot m \cdot h)$	$O(branch(T) \cdot height(T))$

表 1 Time and space complexities for algorithms.

さて，新たなパラメータとして，ラベル付き木の再帰度というものを定義する．これは，その木の根から葉へのパス上に同じラベルが何回出現するかで定義される．再帰度 1 の木は非再帰的な木と呼ばれており，XML の検索において，よく研究されている (例えば， 7), 8), 14) をみよ)．したがって，再帰度は検索の効率さ計る上で重要なパラメータとなりうる．我々は，拡張擬似木パターン照合問題に対し，以下のように 3 つのビット並列型アルゴリズムを与える．この中で，h と l はパターン木 P の高さと葉数を表す．また，W はコンピュータのワード長を示し，L_P は P の再帰度を示す．我々のアルゴリズムは，文字列照合問題で開発された Shift-OR 法[1)] を応用する．

(1) Algorithm-1: これは，$O(n \cdot L_P \cdot l \cdot \lceil \frac{h}{W} \rceil)$ 時間かつ $O(branch(T) \cdot l \cdot \lceil \frac{h}{W} \rceil)$ 領域で走る．

(2) Algorithm-2: これは，$O((n + 2^{h \cdot l}) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 時間かつ $O((branch(T) + 2^{h \cdot l}) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 領域で走る．このアルゴリズムは 2 つのパート，**前処理部**と**照合部**に分けられる．前処理部は $O(2^{h \cdot l} \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 時間掛かり，照合部は $O(n \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 時間掛かる．したがって，一旦パターン木 P を前処理部で処理すれば，任意のターゲット木 T に対し，P のすべての出現を $O(n \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 時間で見つけることができる．

(3) Algorithm-3: これは，$O((n \cdot L_P \cdot l + h \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 時間かつ $O((branch(T) + \alpha_P \cdot L_P \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 領域で走る．ここで，$L_P \leq h$. パラメータ α_P は P に出現する異なる記号の数である．

表 1 に結果をまとめた．一般に，W は $W = O(\log n)$ として定義される．したがって，もし h が高々$\log n$ ならば Algorithm-1 は $O(L_P \cdot n \cdot l)$ 時間で走る．もし $h \cdot l$ が高々$\log n$ ならば，Algorithm-2 は $O(n)$ 時間で走る．もし l が高々$\log n$ ならば，Algorithm-3 は $O(L_P \cdot n \cdot l)$ 時間で走る．したがって，P の再帰度が小さいほど Algorithm-3 は高速に走る．このように，我々のアルゴリズムはワードに収まるような小さなパターンに対し高速に動作する．

2. 擬似拡張木パターン照合問題と関連定義

今，Σ をアルファベットとし，各ノードが Σ の記号によってラベル付けされた木を考える．T をラベル付き木とする．そのとき，もし兄弟間の順序が定義されていれば T は順序木と呼ばれる．もしこのような順序がなければ，無順序木と呼ばれる．木 T の高さ (height) は次のように定義される：まず，T の根の深さ (depth) を 1 と定義する．T の任意のノード v に対し，v の深さは「その親の深さ+1」と定義する．そのとき，T の高さ $height(T)$ は T のすべてのノードの中で最大の深さである．さらに，T の分岐度 $branch(T)$ を，T のノードから出る枝の数の最大値と定義する．次に，T の再帰度 (recursive level) を定義する．T の任意のノード v に対し，v の再帰度は，根から v へのパス上で v と同じラベルが出現する回数で定義される（v のラベルも含む）．そのとき，T の再帰度は T のすべてのノードの中で最大の再帰度として定義される．さらに，$\sigma \in \Sigma$ に対し，σ の再帰度も定義する．これは σ をラベルに持つノードの中で最大の再帰度として定義される．

今，P と T をラベル付き木とし，それぞれをパターン木とターゲット木と呼ぶ．我々は，パターン木 P に対し，2 種類の枝，すなわち，親子枝（parent-child edge, pc-edge と呼ぶ）および先祖子孫枝（ancestor-descendant edge, ad-edge と呼ぶ）を導入する．これらの枝は，ちょうど XPath における，子軸 (/-軸) および子孫軸 (//-軸) に対応している．拡張擬似木パターン照合問題は，先祖子孫関係のみに焦点を当てた照合問題で以下のように定義される．

定義 1 P のノードから T のノードへの写像 ϕ で，次の条件を満足するもの存在するとき，P は T のノード d で**出現**するという（または，d は P の出現という）．

(1) P の任意のノード u に対し，T のノード $\phi(u)$ が存在し，$\phi(u)$ は u と同じラベルを持つ．

(2) P の根は T のノード d に対応する．

(3) P の任意のノード u，v に対し，u から v へ pc-edge ある必要十分条件は $\phi(u)$ は $\phi(v)$ の親である．

(4) P の任意のノード u，v に対し，u から v へ ad-edge ある必要十分条件は $\phi(u)$ は $\phi(v)$ の先祖である．

このとき，拡張擬似木パターン照合問題は，T の中で P が出現するすべてのノードを求めることである．

親は先祖に含まれるので，ad-edge の条件には pc-edge の条件も含まれていることに注意

Vol.2010-AL-131 No.4
2010/9/22

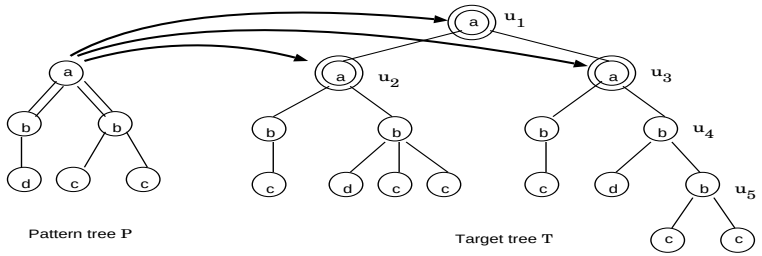

図 1 Occurrences. The arrows indicate occurrences. Double edges in the pattern tree indicate ad-edges and single edges indicate pc-edges.

する．したがって，pc-edge は親子関係に限定したいときにだけ用いられる枝である．出現の例を図 1 に与える．パターン木 P の中で，2 重線の枝が ad-edge を表し，1 重線は pc-edge を表す．T のノード u_1,u_2,u_3 が P の出現になっている．

3. 文字列照合における Shift-OR 法

まず，文字列上のビット並列法である Shift-OR 法について説明する．これは，Baeza-Yates と Gonnet[1] によって開発されたものであり，コンピュータのワード単位で処理を行なうシフト演算と論理和を使ったビット並列文字列照合アルゴリズムである．以下に，アルゴリズムの概略を述べる．

今，$P = p_1 \cdots p_m$ を文字列パターン，$T = t_1 \cdots t_n$ をテキストとする．P に含まれる文字 c に対し，ビットマスク $B[c] = b_1 \cdots b_m$ を定義する．このとき，ビット列 $b_1 \cdots b_m$ は次を満足するように設定される：任意の j に対し，$b_j = 0$ なる必要十分条件は $t_j = c$ のときである．

さて，検索は照合状態を示すビット列 $D = d_1 \cdots d_m$ を使って，t_1 から順番にテキストの文字を読み込みながら，D を以下の式で更新しながら行なう．まず，D の初期値はすべて 1 である．文字 t_j を読んだとき D を以下の式で更新する．

$$D \leftarrow (D \ >> \ 1) \mid B[t_j]$$

ここで，$D \ >> \ 1$ は D を 1 ビット右へシフトする演算を表す．左端には 0 が埋められる．また，演算 "|" はビット毎の論理和を行なう演算である．これを行なったあと，もし $d_m = 0$ ならば，$t_{j-m+1} \cdots t_j$ は P とマッチする文字列となる．

例 1 $P = aabca$，$T = abcaabca$ を考える．そのとき，ビットマスクは $B[a] = 00110$，$B[b] = 11011$，$B[c] = 11101$ となる．$D = 11111$ でスタートする．最後の $T[8] = a$ を計算すると，$D = 01110$ となり，右端のビットが 0 なのでここで P が出現したことが分かる．

4. 拡張擬似木パターン照合問題に対するアルゴリズム

この章では，拡張擬似木パターン照合問題に対するビット並列型アルゴリズムを与える．以下，P をパターン木，T をターゲット木とする．パターン木 P に対し，P_i を根から i 番目の葉へのパス上のラベルの列とする．このとき，P_i をパスパターンと呼び，$|P_i|$ をその長さとする．我々は，Shift-OR 法を利用するため，まず，P をパスパターンに分割する．もし P が l 個の葉を持てば，l 個のパスパターン $P_1, \ldots, P_l$ に分割される．今，パスパターン $P_i = p_1^i \ldots p_{|P_i|}^i$ を考える．T の任意のノード d と P_i の任意のサフィックス $p_j^i \ldots p_{|P_i|}^i$ に対し，d で $p_j^i \ldots p_{|P_i|}^i$ が出現するとは，d から葉へのノードの列 $d_0(= d), d_1, \ldots, d_t$ で次の条件を満足するものが存在するときである．

(1) $d_0, \ldots, d_t$ のラベルの列は $p_j^i \ldots p_{|P_i|}^i$ である．

(2) 任意の e $(j \le e \le |P_i| - 1)$ に対し，

(a) p_e^i から p_{e+1}^i が ad-edge ならば，d_{e-j} は d_{e-j+1} の先祖である．

(b) p_e^i から p_{e+1}^i が pc-edge ならば，d_{e-j} は d_{e-j+1} の親である．

もし P のノード v のラベルが P_i に現れるならば，v が P_i に現れるとも言う．図 2 はパスパターンの例をである．この図で，P は 3 つのパスパターン $P_1 = abc$, $P_2 = abd$ および $P_3 = abc$ に分割される．もしパスパターンが d から葉へのラベル列のプレフィクス (prefix) ならば，そのパスパターンは T のノード d で出現するという．

4.1 パスパターンに対するビットマスク

Shift-OR 法を使うため，h ビットのビットマスク $B[P_i, \sigma]$ を作成する．ここで，h は P の高さ，$P_i = p_1^i \cdots p_{|P_i|}^i$ はパスパターン，記号 σ はラベルを表している．$B[P_i, \sigma]$ の値は

Vol.2010-AL-131 No.4
2010/9/22

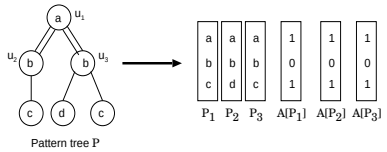

図 2 Decomposition of a pattern tree P into path patterns P_1, P_2 and P_3

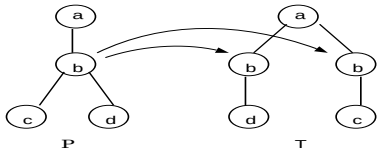

図 3 A bad case in BasicTreeMatch. This case does not satisfy the condition of a occurrence, but the algorithm make an error.

次のように定義される：これは $b_1 \cdots b_h$ ($b_i \in \{0,1\}$) のビット列で，任意の $j \leq |P_i|$ に対し，$b_j = 0$ である必要十分条件は $p_j^i = \sigma$ であり，任意の j ($|P_i|+1 \leq j \leq h$) に対し，$b_j = 0$. 例えば，図 2 の 3 つのパスパターン P_1, P_2, P_3 は以下のように定義される．
$B[P_1, a] = 011$, $B[P_2, a] = 011$, $B[P_3, a] = 011$, $B[P_1, b] = 101$, $B[P_2, b] = 101$, $B[P_3, b] = 101$, $B[P_1, c] = 110$, $B[P_2, c] = 111$, $B[P_3, c] = 110$, $B[P_1, d] = 111$, $B[P_2, d] = 110$, $B[P_3, d] = 111$.

さらに，先祖子孫関係を維持するために，各 P_i に対し，別のビットマスク $AD[P_i]$ を導入する．ビットマスク $AD[P_i] = b_1 \cdots b_h$ は次のように定義される：任意の j に対し，もし p_{j-1}^i から p_j^i への枝が ad-edge ならば $b_j = 0$; さもなければ $b_j = 1$ である．図 2 に例を示す．

4.2 基本アルゴリズム

この章では，Shift-OR 法を利用した基本的なアルゴリズム *BasicTreeMatch*(P,T) を与える．これを図 4 に示す．このアルゴリズムはターゲット木 T のノードで，P のすべてのパスパターンが出現するものすべてを見つけることができる，我々はビット列を要素とする配列 $M[P_i, d]$ を使う．この配列は照合状態と呼ばれ，その要素は長さ h のビット列 $b_1 \cdots b_h$ である．T のノードをポストオーダ (postorder) で巡回しながら，T の各ノード d に対し，$M[P_i, d]$ を計算していく．今，$P_i = p_1^i \cdots p_{|P_i|}^i$ とする．このとき，任意の T のノード d に対し，$M[P_i, d] = b_1 \cdots b_h$ は次の条件を満足するように計算される：任意の $j \leq |P_i|$ に

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 $d_1, \cdots, d_t$ 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) $OLD\_M := M[P_i, d]$,
(3) $M[P_i, d] := (M[P_i, d] \ << \ 1) \mid B[P_i, \sigma]$,
(4) $M[P_i, d] := M[P_i, d] \& (OLD\_M \mid AD[P_i])$.
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.

図 4 The algorithm *BasicTreeMatch*

対し，$b_j = 0$ である必要十分条件は $p_j^i \cdots p_{|P_i|}^i$ がノード d で出現することである．照合状態 $M[P_i, d]$ は最初に $1^{|P_i|}0^{h-|P_i|}$ に設定される．演算 $M[P_i, d] << 1$ は 1 ビット左へシフトするシフト演算を示す．そのとき右端には 0 が入る．さらに演算 "|" はビット毎の OR をとる演算であり，そして "&" はビット毎の AND をとる演算である．アルゴリズム *BasicTreeMatch* は P をパスパターンに分割し，それから T のノードをポストオーダで巡回することにより，すべてのパスパターンが出現するノードを探す．次の命題が成り立つ．

命題 1 ノード d を T のノードで，*BasicTreeMatch* によって返されるものとする．そのとき，P のすべてのパスパターンはノード d で出現する．

アルゴリズム *BasicTreeMatch* は完全に拡張擬似木パターン照合問題を解くことはできない．問題は，P のノードが 2 つ以上の T ノードに対応する可能性がある点である (図 3 を見よ)．次の節で，同期の概念を導入することによりこの問題を解決する．したがってこれにより拡張擬似木パターン照合問題が解かれる．

4.3 拡張擬似木パターン照合アルゴリズム

前に述べたように，アルゴリズム *BasicTreeMatch* には問題点があった．この章では，その問題点，すなわちパターン木の一つのノードをターゲット木の 2 つ以上のノードに対応させてしまう問題を解決する．これによって，拡張擬似木パターン照合問題を解くことがで

Algorithm-1 TreeMatch(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) $OLD\_M := M[P_i, d]$,
(3) $M[P_i, d] := (M[P_i, d] \ << \ 1) \mid B[P_i, \sigma]$.
(4) $M[P_i, d] := M[P_i, d] \& (OLD\_M \mid AD[P_i])$.
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 an 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 an occurrence.

図 5 The algorithm *TreeMatch*

きる．このために，まず，同期ビットマスク呼ぶ新たなビットマスクを導入する．

さて，u をパターン木 P の任意のノードとし，P の高さを h とする．そのとき，任意のパスパターン P_i に対し，同期ビットマスク $Syn[P_i, u] = b_1 \cdots b_h$ を次のように定義する：任意の $1 \leq j \leq h$ に対し，もし b_j に対応するノードがちょうど u ならば，そのとき $b_j = 1$ とする．さもなければ $b_j = 0$ とする．このように，$Syn[P_1, u], \ldots, Syn[P_l, u]$ において，ただ単にノード u に対応するビットだけが 1 にセットされ，その他のビットは 0 である．例えば，図 2 で与えられたパターン木に対する同期ビットマスクは，各ノード u_1, u_2 及び u_3 に対し，次のようになる．

$Syn[P_1, u_1] = 100$, $Syn[P_2, u_1] = 100$, $Syn[P_3, u_1] = 100$, $Syn[P_1, u_2] = 010$, $Syn[P_2, u_2] = 000$, $Syn[P_3, u_2] = 000$, $Syn[P_1, u_3] = 000$, $Syn[P_2, u_3] = 010$, $Syn[P_3, u_3] = 010$.

図 5 に擬似木パターン照合のためのアルゴリズム *TreeMatch* を与える．このアルゴリズムは，*BasicTreeMatch* に同期ステージ（Step2）を加えることによって構成されている．さて，どのように *TreeMatch* の同期ステージ（Step2）が働くか説明する．今，u を P のノードで，その深さを j とする．そのとき，任意の P_i と T の任意のノード d に対し，$M[P_i, d]$ の j 番目のビット b_j は u に対応する．ステップ 2 はすべての $P_i \in \mathcal{P}_u$ に対し，$M[P_i, d]$ の b_j ビットをチェックし，もし少なくとも一つの $M[P_i, d]$ の b_j が 1 ならばすべての $M[P_i, d]$ のビット b_j を 1 にセットする．ここで，$\mathcal{P}_u$ は u が現れるパスパターンからなる集合である．これを行なうため，最初に Step2 の (a) で b_j の値を $Syn[P_i, u]$ を使ってチェックする．もし $M[P_i, d]$ の b_j が 1 ならば $SYN[P_i]$ の j 番目のビットは 1 になる．したがって，もし b_j が 1 であるような $SYN[P_i]$ が少なくとも一つ存在すれば，そのとき $SynMask$ の j 番目のビットは Step2 の (b) において 1 になる．最終的に，すべての $P_i \in \mathcal{P}_u$ に対し，$M[P_i, d]$ の j 番目のビット b_J は，Step2 の (c) において 1 にセットされる．我々は $M[P_i, d_1]$,...,$M[P_i, d_t]$ を計算するための領域は再利用できることに注意する．したがって，照合状態は $branch(T)+1$ の領域があれば計算できる．以上より次の定理が成り立つ．

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

図 6 A matching state $M[d]$ packed sequentially

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

図 7 Packed bit-masks for Fig.2

定理 1 アルゴリズム *TreeMatch* は P のすべての出現を $O(n \cdot L_P \cdot l \cdot \lceil \frac{h}{W} \rceil)$ 時間かつ $O(branch(T) \cdot l \cdot \lceil \frac{h}{W} \rceil)$ 領域で見つける．ここで，n は T のノード数, L_P, h, 及び l はそれぞれ P の再帰度，高さ，葉数である．さらに，W はコンピュータのワード長である．

5. ビット列のパック化によるアルゴリズム

前節で与えた *TreeMatch* は，各パスパターンに対し，一つずつ照合状態を計算している．そのため，パスパターンの数に比例した時間が掛かる．本節では，すべてのパスパターンを一つのビット列に詰め込むことによって，同時に照合状態を計算するアルゴリズムを与える．このとき，詰め込みの方法には 2 通り考えられる．一つは，各パスパターンを順番に並べて一つのビット列とする方法，もう一つは，各パスパターンの同位置にあるビットをグ

Algorithm-2 FastTreeMatch(P, T)
Step 0. /* Initialization */
(1) compute bit-masks $B[P_i, \sigma]$, and set $B[\sigma] := (B[P_1, \sigma], \ldots, B[P_l, \sigma])$ 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) $mathitOLD\_M := M[d]$,
(3) $M[d] := ((M[d] \ << \ 1) \ \& \ ZMask) \mid B[\sigma]$.
(4) $M[d] := M[d]\&(OLD\_M \mid AD[P])$.
Step 2. /* Synchronization between path patterns */
(1) $SYN := M[d] \ \& \ PSyn[\sigma]$,
(2) $M[d] := M[d] \mid PSynMask[SYN]$.
Step 3. /* This step determines whether a match occurs. */
(1) $Acc := M[d] \mid AccCheck$,
(2) if $Acc = AccCheck$, then return d as an occurrence.

図 8 The algorithm *FastTreeMatch*

ループ化することによって一つのビット列を構成する方法である．すなわち，パスパターンをインターリーブすることによってビット列に詰め込む．

5.1 逐次的なビット列の詰め込みによるアルゴリズム

さて，パスパターンを順番に並べて詰め込む方法について述べる．図 8 に改良されたアルゴリズム *FastTreeMatch* を与える．このアルゴリズムでは，図 6 に示すように，照合状態 $M[P_1, d]$, ..., $M[P_l, d]$ を一つのワード $M[d]$ に詰め込んでいる．$(M[P_1, d], \ldots, M[P_l, d])$ によって，このような詰め込まれたビット列 $M[d]$ を表す．同様に，$B[P_1, \sigma], \ldots, B[P_l, \sigma]$ も一つのワードに詰め込み，図 7 のように $B[\sigma] = (B[P_1, \sigma], \ldots, B[P_l, \sigma])$ を作成する．

今，$h_i = |P_i|$ とし，$M[P_i, d]$ や $B[P_i, \sigma]$ のようなビット列に対し，できるだけコンパクトにするため，これらに対し，h_i ビットのビット列を使う．さらに，Step2 の同期ステージを同時に実行するため，同期ビットマスクを次のように $PSyn[\sigma, lev]$ に詰め込む．ここで，σ は記号，lev は再帰度を表す．我々は，P のノードをそのラベルと再帰度によって部分集合 $N_{(\sigma,lev)}$ に分類する．さて，任意の記号 σ と任意の再帰度 lev に対し，$N_{(\sigma,lev)} = \{u_1, \ldots, u_s\}$ とする．そのとき，$PSyn[\sigma, lev] = (PSyn_1, \ldots, PSyn_l)$ と定義する．ここで，もしノード u_j $(1 \le j \le s)$ が P_i 上に現れるならば，そのとき $PSyn_i = Syn[P_i, u_j]$ とし，さもなければ，$PSyn_i = 0^{h_i}$ とする．

我々は，同期の結果を照合状態に反映させるために，ビット列からなる配列 $PSynMask[SYN, lev]$ を導入する．ここで，$SYN = (SYN_1, \ldots, SYN_l)$ でかつ各 SYN_i は *TreeMatch* の中の $SYN[P_i]$ に対応する．さて，$\mathcal{P}_u$ を u が現れるパスパターン P_i からなる集合とする．そのとき，任意のノード $u, v \in N_{(\sigma,lev)}$ に対し，$\mathcal{P}_u \cap \mathcal{P}_v = \emptyset$ が成り立つ．したがって，$N_{(\sigma,lev)}$ のすべてのノードに対し，$SYN[P_i]$ を SYN で表すことができる．アルゴリズム *FastTreeMatch* は SYN を 1 回のステップで計算する．一方，*TreeMatch* は $N_{(\sigma,lev)}$ の各ノードに対し，$SYN[P_i]$ を計算する．$PSynMask[SYN, lev]$ の値は $(PSynMask_1, \ldots, PSynMask_l)$ のなるよう定義される．ここで，$PSynMask_i$ は $u \in N_{(\sigma,lev)}$ に対応した $\mathcal{P}_u$ に対して計算された *SynMask* に対応する．したがって，*TreeMatch* の Step2 と同様に，を使って我々は照合状態 $M[d]$ を更新することができる．*FastTreeMatch* は Step2 の 2 でこのタスクを実行している．我々は，最初に Step0 で $PSynMask[SYN, lev]$ を手続き $SetPSynMask(P)$ を使って計算する．この手続きを図 9 で与える．$SetPSynMask(P)$ において，$\mathcal{K}_{e^i_j}$ は次のようなビット列 $(K_1, \ldots, K_l)$ であると定義する．すなわち，ただ単に $K_{e^i_j}$ の左端から $d(u_i)$ 番目のビットのみが 1 であり，他のすべてのビットは 0 である．ここで，任意の $u_i \in N_{(\sigma,lev)}$ に対し，$\mathcal{P}_{u_i} = \{P_{e^i_1}, \ldots, P_{e^i_{t_i}}\}$ であり，そして $d(u_i)$ は u_i の深さを示す．

さらに，2 つの特別なビットマスク *ZMask* と *AccCheck* を導入する．ビットマスク *ZMask* は $(1^{h_1-1}0, \ldots, 1^{h_l-1}0)$ に設定され，詰め込まれた各照合状態の右端のビットを 0 にするために使われる．ビットマスク *AccCheck* は $(01^{h_1-1}, \ldots, 01^{h_l-1})$ に設定され，出現が発生したかどうかをチェックするために使われる．次の定理が成り立つ．

定理 2 アルゴリズム *FastTreeMatch* は T の中のすべての P の出現を $O((n+2^{h \cdot l}) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 時間かつ $O((branch(T) + 2^{h \cdot l}) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 領域で見つけることができる．

もし $h \cdot l \le \log n$ ならば，*FastTreeMatch* は $O(n \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 時間かつ領域で走る．したがって，$h \cdot l = O(W)$ ならば $O(n)$ 時間で走る．このように，小さなパターンに対してはターゲット木のサイズに比例した時間で拡張擬似木パターン問題を解くことができる．

5.2 パスパターンのインターリーブ的詰め込みによる方法

前章では，コンピュータワードに照合状態を逐次的に詰め込む手法を示した．本章では，照合状態を互いに差し込む形でワードに詰め込む方法について述べる．

アルゴリズム *FastTreeMatchIL* を図 12 に与える．各照合状態 $M[P_i, d]$ に対し，$M[P_i, d][j]$ を $M[P_i, d]$ の j 番目のビットとする．任意の j $(1 \le j \le h)$ に対し，$M[P, d][j] = M[P_1, d][j] \cdots M[P_l, d][j]$ と定義する．そのとき，図 10 に示すように，

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 $SYN = \mathcal{K}_{e_1^i}, \ldots, \mathcal{K}_{e_{t_i}^i}$ 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 $k_1 = \mathcal{K}_{e_1^1}, \ldots, \mathcal{K}_{e_{t_1}^1}, \ldots, \mathcal{K}_{e_1^s}, \ldots, \mathcal{K}_{e_{t_s}^s}$ 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

図 9 The procedure *SetPSynMask*.

M[P$_1$,d][1]	. . .	M[P$_l$,d][1]	. . .	M[P$_1$,d][h]	. . .	M[P$_l$,d][h]

図 10 An interleaved matching state $\overline{M}[d]$

$\overline{M}[d] = (M[P,d][1], \cdots, M[P,d][h])$ と定義する．このように，$\overline{M}[d]$ は，$M[P_1,d]$, $\cdots$, $M[P_l,d]$ に対し，初めにこれらの最初のビットを取り，次に 2 番目のビットを取り，というように順番にビットを取ることによって定義される．同様に，図 11 に示すように，各記号 σ に対し，$\overline{B}[\sigma] = (B[P,\sigma][1], \ldots, B[P,\sigma][h])$ と定義する．ここで，任意の j $(1 \le j \le h)$ に対し，$B[P,\sigma][j] = B[P_1,\sigma][j] \cdots B[P_l,\sigma][j]$ と定義され，$B[P_i,\sigma][j]$ は $B[P_i,\sigma]$ の j 番目のビットである．

さらに，*TreeMatch* の Step 2 の同期作業を同時に実行するために，同期ビットマスク $\overline{PSyn}[\sigma, lev]$ を定義する．ここで，σ は記号，lev は再帰度を示す．我々は，P のノードを部分集合 $N_{(\sigma,lev)}$ に分割する．ここで，$N_{(\sigma,lev)}$ は，ラベルとして記号 σ をもちかつその再帰度が lev であるノードの集合である．今，任意の記号 σ と再帰度 lev に対し，$N_{(\sigma,lev)} = \{u_1, \ldots, u_s\}$ と定義する．さらに，任意の e $(1 \le e \le s)$ に対し，$Syn[P,u_e][j] =$

$\overline{\text{B}}$[σ]	B[P$_1$,σ][1]	. . .	B[P$_l$,σ][1]	. . .	B[P$_1$,σ][h]	. . .	B[P$_l$,σ][h]

$\overline{\text{B}}$[a]	000	111	111
$\overline{\text{B}}$[b]	111	000	111
$\overline{\text{B}}$[c]	111	110	100
$\overline{\text{B}}$[d]	111	111	011

図 11 Interleaved bit-masks for Fig.7

Algorithm-3 FastTreeMatchIL(P, T)
Step 0. /* Initialization */
(1) compute bit-masks $B[P_i,\sigma]$, and set $\overline{B}[\sigma]$ for each symbol σ in P,
(2) for all path patterns P_i, set the initial matching state $IM[i]$ $(1 \le i \le h)$ to 1^l, and then set the packed initial matching state $IM := (IM[1], \ldots, IM[h])$,
(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 synchronization bit-mask $\overline{PSyn}[\sigma, lev]$ which is a bit sequence obtained by interleaving $PSyn[\sigma, lev]$. Here $PSyn[\sigma, lev] = (PSyn_1, \ldots, PSyn_l)$, where if $u_j \in N_{(\sigma,lev)}$ appears on P_i, then $PSyn_i = Syn[P_i, u_j]$; otherwise $PSyn_i = 0^h$.
(5) *SetPSynMaskIL*(P), /* set $\overline{PSynMask}[STATE, lev, \sigma]$. */
(6) set $AccCheck := (0^l, 1^l, \ldots, 1^l)$.
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 $\overline{M}[d] := IM$; otherwise $\overline{M}[d] := \overline{M}[d_1] \& \cdots \& \overline{M}[d_t]$,
(2) $\overline{M}[d] := ((\overline{M}[d] << l)) \mid \overline{B}[\sigma]$.
Step 2. /* Synchronization between path patterns */
For $lev = 1, \ldots, L_\sigma$ do /* L_σ denotes the recursive level of σ. */
(1) $SYN := \overline{M}[d]$ & $\overline{PSyn}[\sigma, lev]$, /* note that $SYN = (SYN_1, \ldots, SYN_h)$. */
(2) $STATE := SYN_{d(u_1)} \mid \cdots \mid SYN_{d(u_s)}$, /* $d(u_i)$ is the depth of u_i for $u_i \in N_{(\sigma,lev)}$. */
(3) $\overline{M}[d] := \overline{M}[d] \mid \overline{PSynMask}[STATE, lev, \sigma]$.
Step 3. /* This step determines whether a match occurs. */
(1) $Acc := \overline{M}[d] \mid AccCheck$,
(2) if $Acc = AccCheck$, then return d as an occurrence.

図 12 The algorithm *FastTreeMatchIL*

$Syn[P_1,u_e][j] \cdots Syn[P_l,u_e][j]$ と定義する．ここで，$Syn[P_i,u_e][j]$ は $Syn[P_i,u_e]$ の j 番目のビットを表す．このとき，$\overline{PSyn}[\sigma, lev] = (PSyn_1, \ldots, PSyn_h)$ と定義する．ここで，任意の j $(1 \le j \le h)$ に対し，$PSyn_j = Syn[P,u_1][j] \mid \cdots \mid Syn[P,u_s][j]$ である．

我々はビット列の配列 $PSynMask[STATE, lev, \sigma]$ を導入する．これは，同期の結果を照合状態へ反省させるために使われる．$\mathcal{P}_u$ を，ノード u が P_i 上に現れるようなパスパターン P_i の集合とする．そのとき，任意のノード $u, v \in N_{(\sigma,lev)}$ に対し，$\mathcal{P}_u \cap \mathcal{P}_v = \emptyset$ が成り立つ．したがって，$N_{(\sigma,lev)}$ のすべてのノードに対し，*TreeMatch* で使われる $SYN[P_i]$ を

Procedure *SetPSynMaskIL*(P)
For all symbols σ which occurs in P, do the following:
Step 1. /* set $SubMask[STATE, lev]$. */
(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) $TMask := (TMask_1, \ldots, TMask_h)$, where $TMask_k = Syn[P, u_i][k]$ for $1 \leq k \leq h$,
(4) for $STATE = I^i_{e^i_1}, \ldots, I^i_{e^i_{t_i}}$ do
(5) $SubMask[STATE, lev] := TMask$,
(6) end-for
(7) end-for
(8) end-for
Step 2. /* compute $\overline{PSynMask}[STATE, lev, \sigma]$. */
(1) for $lev = 1, \ldots, L_P$ do
(2) $IX := 0$, $Val[0] := 0$ and $\overline{PSynMask}[0, lev, \sigma] := (0^l, \ldots, 0^l)$,
(3) for $k_1 = I^1_{e^1_1}, \ldots, I^1_{e^1_{t_1}}, \ldots, I^s_{e^s_1}, \ldots, I^s_{e^s_{t_s}}$ do
(4) $t := IX$,
(5) for $k_2 := 0, \ldots, t$ do
(6) $\overline{PSynMask}[k_1 \mid Val[k_2], lev, \sigma] := \overline{PSynMask}[Val[k_2], lev, \sigma] \mid SubMask[k_1, lev]$,
(7) $IX := IX + 1$,
(8) $Val[IX] := k_1 \mid Val[k_2]$,
(9) end-for
(10) end-for
(11) end-for

図 13 The procedure *SetPSynMaskIL*

l ビット列である $STATE$ で表すことができる．

アルゴリズム *FastTreeMatchIL* は $STATE$ を高々$|N_{(\sigma,lev)}|$ ステップで計算する．$PSynMask[STATE, lev, \sigma]$ の値は，$(PSynMask_1, \ldots, PSynMask_l)$ であると定義される．ここで，各 $PSynMask_i$ は，$u \in N_{(\sigma,lev)}$ に対応する $\mathcal{P}_u$ に対して計算される $SynMask$ に対応する．そのとき，$\overline{PSynMask}[STATE, lev, \sigma]$ の値は，$PSynMask[STATE, lev, \sigma]$ をインターリーブすることによって得られるビット列である．したがって，*TreeMatch* の Step 2 と同様な方法で，$\overline{PSynMask}[SYN, lev, \sigma]$ を使って，照合状態 $\overline{M}[d]$ を更新することができる．アルゴリズム *FastTreeMatchIL* は Step 2 の 2 でこの作業を行なう．我々は，図 13 で与えられた *SetPSynMaskIL*(P) を使って，Step 0 で $\overline{PSynMask}[SYN, lev, \sigma]$ を計算する．*SetPSynMask*(P) の中で，$I^i_{e^i_j}$ は，ただ単に $I^i_{e^i_j}$ の左端から e^i_j 番目のビットだけが 1 で，その他のビットはすべて 0 である l ビット列である．ここで，任意の $u_i \in N_{(\sigma,lev)}$ に対し，$\mathcal{P}_{u_i} = \{P_{e^i_1}, \ldots, P_{e^i_{t_i}}\}$ であり，$d(u_i)$ は u_i の深さを表す．さらに，我々は特別なビットマスク *AccCheck* を導入する．それは $(0^l, 1^l, \ldots, 1^l)$ なるビット列であり，出現が発生したかどうかチェックするために使われる．次の定理が成り立つ．

定理 3 アルゴリズム *FastTreeMatchIL* は T の中のすべての P の出現を $O((n \cdot L_P \cdot l + h \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 時間かつ $O((branch(T) + \alpha_P \cdot L_P \cdot 2^l) \cdot \lceil \frac{h \cdot l}{W} \rceil)$ 領域で見つけることができる．ここで，α_P は P に出現する異なる記号の数である．

参 考 文 献

1) R. Baeza-Yates and G.H. Gonnet, A New Approach to Text Searching, Communications of the ACM, 35, 10(1992) 74–82.
2) Z. Bar-Yossef, M. Fontoura and V. Josifovski, Buffering in Query Evaluation over XML streams, PODS 2005, (2005) 216–227.
3) C. Chauve, Tree pattern matching with a more general notion of occurrence of the pattern, Information Processing Letters, 82(2001) 197–201.
4) R. Cole and R. Hariharan, Verifying Candidate Matches in Sparse and Wildcard Matching , in Proceedings of the 34th ACM Symposium on Theory of Computing, (2002) 592–601.
5) R. Cole and R. Hariharan, Tree Pattern Matching to Subset Matching in Linear Time, SIAM J. Comput., 32, 4(2003) 1056–1066.
6) M. Dubiner, Z. Galil, and E. Magen, Faster Tree Pattern Matching, Journal of the ACM, 41, 2(1994) 205–213.
7) M. Götz, C. Koch and W. Martens, Efficient Algorithms for the Tree Homeomorphism Problem, DBPL 2007, LNCS 4797, (2007) 17–31.
8) M. Götz, C. Koch and W. Martens, Efficient Algorithms for Descendant-Only Tree Pattern Queries, Information Systems, 34, (2009) 602–623.
9) C.M. Hoffman and M.J. O'Donnell, Pattern matching in trees, Journal of the ACM, 29, 1(1982) 68–95.
10) P. Kilpeläinen and H. Mannila, Ordered and Unordered Tree Inclusion, SIAM J. Comput., 24, 2(1995) 340–356.
11) J.V. Leeuweno, Graph Algorithms, In J.V. Leeuwen, ed. Handbook of Theoretical Computer Science, Elsevier Science Pub., 1990.
12) R. Shamir and D. Tsur, Faster Subtree Isomorphism, J. of Algorithms, 33(1999) 267–280.
13) H. Yamamoto and D. Takenouchi, Bit-Parallel Tree Pattern Matching Algorithms for Unordered Labeled Trees, Proc. of WADS'09, LNCS 5664, (2009) 554–565.
14) J.T. Yao, M. Zhang, A Fast Tree Pattern Matching Algorithm for XML Query, Proc. of the WI'04, (2004) 235–241.
15) P. Zezula, F. Mandreoli, and R. Martoglia, Tree Signatures and Unordered XML Pattern Matching, Proc. of the SOFSEM'04, LNCS 2932, (2004) 122–139.