Fast Ordered Tree Matching for XML Query Evaluation

Yangjun Chen, Yibin Chen
Dept. Applied Computer Science, University of Winnipeg
Winnipeg, Manitoba, Canada R3B 2E9
y.chen@uwinnipeg.ca

Abstract– *An XML tree pattern query, represented as a labeled tree, is essentially a complex selection predicate on both structure and content of an XML. Tree pattern matching has been identified as a core operation in querying XML data. We distinguish between two kinds of tree pattern matchings: ordered and unordered tree matching. By the unordered tree matching, only ancestor/descendant and parent/child relationships are considered. By the ordered tree matching, however, the order of siblings has to be taken into account besides ancestor/descendant and parent/child relationships. While different fast algorithms for unordered tree matching are available, no efficient algorithm for ordered tree matching for XML data exists. In this paper, we discuss a new algorithm for processing ordered tree pattern queries, whose time complexity is polynomial.*

Key words: XML documents; tree pattern queries; tree matching; tree encoding; XB-trees

1 Introduction

Xpath [16, 17] is a language for matching paths and, more generally, patterns in tree-structured data and XML documents. These patterns may use either just purely the tree structure of an XML document or data values occurring in the document as well. For example, the XPath expression:

book[title = ‘Art of Programming’]//author[firstName = ‘Donald’ and lastName = ‘Knuth’]

matches *author* elements that (i) have a child subelement *firstName* with content *Knuth*, (ii) have a child subelement *lastName* with content *Donald*, and (iii) are descendants of *book* elements that have a child *title* subelement. It can be represented by a tree structure as shown in Fig. 1.

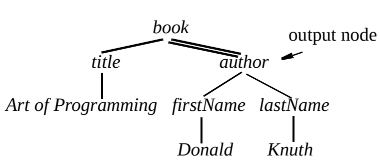

Fig. 1. An Xpath tree

In Fig. 1, there are two kinds of edges: child edges (/-edges for short) for parent-child relationships, and descendant edges (//-edges for short) for ancestor-descendant relationships. A /-edge from node v to node u is denoted by $v \rightarrow u$ in the text, and represented by a single arc; u is called a */-child* of v. A //-edge is denoted by $v \Rightarrow u$ in the text, and represented by a double arc; u is called a *//-child* of v.

Many different strategies have been proposed to efficiently evaluate such kind of queries [1, 3 - 9, 12, 14, 15]. But most of them take only ancestor/descendant and parent/child relationships into consideration. No attention is paid to the left-to-right order of the nodes.

However, in many applications, such as the natural language processing [2], the video content-based retrieval [13], the scene analysis, as well as some problems in the computational biology (such as RNA structure matching [11]) and the data mining (such as tree mining [18]), the order of the nodes is significant. As an example, consider querying grammatical structures as shown in Fig. 2, which is the parse tree of a natural language sentence.

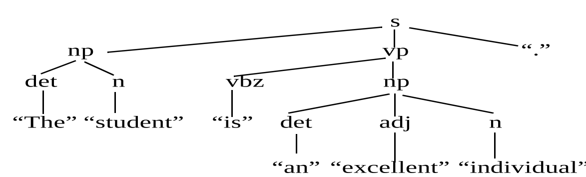

Fig. 2. The parse tree of a sentence

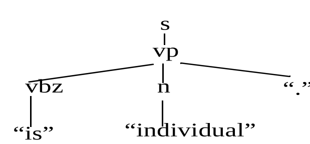

Fig. 3. A query tree which matches a subtree of the parse tree shown in Fig. 2

One might want to locate, say, those sentences that include a verb phrase containing the verb “is” and after it a noun “individual” followed by “.”. This is exactly the sentences whose parse tree can be matched to a subtree of the tree shown in Fig. 2. (See Fig. 3 for illustration.) But the left-to-right ordering must be followed.

In this paper, we discuss an efficient algorithm to solve this kind of problems.

The remainder of the paper is structured as follows. In section 2, we give some basic definitions, which are needed for the subsequent discussion. In Section 3, we present the main algorithm. In Section 4, we analyze the computational complxities. Finally, the paper concludes in Section 5.

2 Basic definitions

We concentrate on labeled trees that are ordered, i.e., the order between siblings is significant. Technically, it is convenient to consider a slight generalization of trees, namely forests. A forest is a finite ordered sequence of disjoint finite trees. A tree T consists of a specially designated node *root*(T) called the root of the tree, and a forest $<T_1, ..., T_k>$, where $k \geq 0$. The trees $T_1, ..., T_k$ are the subtrees of the root of T or the immediate subtrees of tree T, and k is the outdegree of the root of T. A tree with the root t and the subtrees $T_1, ..., T_k$ is denoted by $<t; T_1, ..., T_k>$. The roots of the trees $T_1, ..., T_k$ are the children of t and siblings of each other. Also, we call $T_1, ..., T_k$ the sibling trees of each other. In addition, $T_1, ..., T_{i-1}$ are called the left sibling trees of T_i, and T_{i-1} the immediate left sibling tree of T_i. The root is an ancestor of all the nodes in its subtrees, and the nodes in the subtrees are descendants of the root. The set of descendants of a node v is denoted by *desc*(v). A leaf is a node with an empty set of descendants.

Sometimes we treat a tree T as the forest $<T>$. We may also denote the set of nodes in a forest F by $V(F)$. For example, if we speak of functions from a forest G to a forest F, we mean functions mapping the nodes of G onto the nodes of F. The size of a forest F, denoted by $|F|$, is the number of the nodes in F. The restriction of a forest F to a node v with its descendants *desc*(v) is called a subtree of F rooted at v, denoted by $F[v]$.

Let $F = <T_1, ..., T_k>$ be a forest. The preorder of a forest F is the order of the nodes visited during a preorder traversal. A preorder traversal of a forest $<T_1, ..., T_k>$ is as follows. Traverse the trees $T_1, ..., T_k$ in ascending order of the indices in preorder. To traverse a tree in preorder, first visit the root and then traverse the forest of its subtrees in preorder. The postorder is defined similarly, except that in a postorder traversal the root is visited after traversing the forest of its subtrees in postorder. We denote the preorder and postorder numbers of a node v by *pre*(v) and *post*(v), respectively.

Using preorder and postorder numbers, the ancestorship can be easily checked. If there is path from node u to node v, we say, u is an ancestor of v and v is a descendant of u. In this paper, by 'ancestor' ('descendant'), we mean a proper ancestor (descendant), i.e., $u \neq v$.

Lemma 1 Let v and u be nodes in a forest F. Then, v is an ancestor of u if and only if $pre(v) < pre(u)$ and $post(u) < post(v)$.

Proof. See Exercise 2.3.2-20 in [10] (page 347). □

Similarly, we check the left-to-right ordering as follows.

Lemma 2 Let v and u be nodes in a forest F. The node v is said to be to the left of u if they are not related by the ancestor-descendant relationship and u follows v when we traverse F in preorder. Then, v is to the left of u if and only if $pre(v) < pre(u)$ and $post(v) < post(u)$.

Proof. The proof is trivial. □

In the following, we use the postorder numbers to define an ordering of the nodes of a forest F given by $v \prec v'$ iff $post(v) < post(v')$. Also, $v \preceq v'$ iff $v \prec v'$ or $v = v'$. Furthermore, we extend this ordering with two special nodes $\bot \prec v \prec \top$. The *left relatives*, lr(v), of a node $v \in V(F)$ is the set of nodes that are to the left of v and similarly the *right relatives*, rr(v), are the set of nodes that are to the right of v.

Based on the above concepts, we give the definition of ordered tree matching.

Definition 1 An embedding of a tree pattern P into an XML document T is a mapping φ: $P \rightarrow T$, from the nodes of P to the nodes of T, which satisfies the following conditions:

(i) Preserve node label: For each $u \in P$, *label*(u) = *label*($\varphi(u)$) (or say, u matches $f(u)$).

(ii) Preserve *parent-child*/*ancestor-descendant* relationship: If $u \rightarrow v$ in P, then $\varphi(v)$ is a child of $\varphi(u)$ in T; if $u \Rightarrow v$ in Q, then $\varphi(v)$ is a descendant of $\varphi(u)$ in T.

(iii)Preserve *left-to-right order*: For any two nodes $v_1 \in P$ and $v_2 \in P$, if v_1 is to the left of v_2, then $\varphi(v_1)$ is to the left of $\varphi(v_2)$ in T. □

If there exists such a mapping from P to T we say, T includes P, T contains P, T covers P, or say, P can be embedded in T.

Fig. 4 shows an example of an ordered tree embedding.

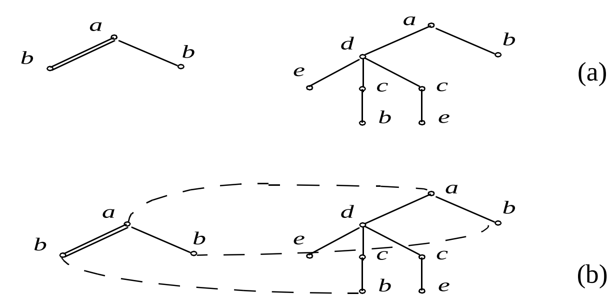

Fig. 4: (a) The tree on the left can be matched to a subtree in the tree on the right. (b) The dashed lines show a tree embedding.

Let P and T be two labeled ordered trees. An embedding φ of P in T is said to be *root-preserving* if $\varphi(root(P)) = root(T)$. If there is a root-preserving embedding of P in T, we say that the root of T is an occurrence of P.

Fig. 4(b) also shows an example of a root preserving embedding. Obviously, restricting to root-preserving embedding does not lose generality. In fact, what can be found by the top-down algorithm to be discussed is a root-preserving tree embedding.

Throughout the rest of the paper, we refer to the labeled ordered trees simply as trees.

3 Algorithm

In this section, we give our algorithm. For simplicity, we consider only the case that a query tree contains only //-

edges. But it is an easy task to extend the algorithm for general cases.

Let $G = <P_1, ..., P_l>$ ($l \geq 1$) be a forest. Consider a node v in G with children $v_1, ..., v_j$, ordered from left to right. We will use $<v_k, i>$ ($1 \leq k \leq j$; $1 \leq i \leq j - k + 1$) to represent an ordered forest containing i subtrees of v: $<G[v_k], ..., G[v_{k+i-1}]>$. Let v be a node on the left-most path in P_1. We call $<v, i>$ a *left corner* of G. Denote by p_j the root of P_j in $G = <P_1, ..., P_l>$ ($j = 1, ..., l$). Then, the left corner $<p_1, i>$ represents the forest $<P_1, ..., P_i>$ ($i \leq l$). In addition, we use $\delta(v)$ to represent a link from a node v in G to the left-most leaf node in $G[v]$, as illustrated in Fig. 5.

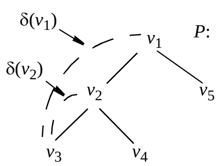

Fig. 5. A pattern tree and illustration for $\delta(v_1)$

Let v' be a leaf node in G. $\delta(v')$ is defined to be a link to v' itself. So in Fig. 5, we have $\delta(v_1) = \delta(v_2) = \delta(v_3) = v_3$. We also denote by $\delta^{-1}(v')$ a set of nodes $\boldsymbol{x}$ such that for each $v \in \boldsymbol{x}$ $\delta(v) = v'$. Therefore, in Fig. 5, $\delta^{-1}(v_3) = \{v_1, v_2, v_3\}$. The out-degree of v in a tree is denoted by $d(v)$ while the height of v is denoted by $h(v)$, defined to be the number of edges on the longest downward path from v to a leaf. The height of a leaf node is set to be 0.

Our algorithm mainly contains two functions: *top-down*(T, G) and *bottom-up*(F, G) to check tree matching, where T is a tree, and F and G are two forests. Each of the two functions returns a left corner $<v, i>$ of G (*i.e.*, v is a node on the left-most path of P_1) such that

- $<G[v_1], ..., G[v_i]>$ can be embedded in T or in F, where $v_1 = v, v_2, ..., v_i$ are consecutive siblings; and
- there is no other left corner $<v', j>$ with v' being an ancestor of v, which can be embedded in T or in F. (In other words, $<v, i>$ is the highest left corner in G such that it can be embedded in T.)

If $v = p_1$ (the root of P_1), it shows that $P_1, ..., P_i$ can be embedded in T or in F.

If the target (a document tree) is a tree and the pattern (a query tree) is a forest, we call the function *top-down*. If both the target and the pattern are forests, we call the function *bottom-up*. But during the computation, they will be called from each other.

In addition, each time a call *top-down*(T, G) returns a pair $<v, i>$, the root t of T is associated with that pair, referred to as $\kappa(t)$. Initially, each $\kappa(t)$ is set to ϕ. $\kappa(t)$ is mainly used in *bottom-up*( ) to avoid redundancy.

Let $T = <t; T_1, ..., T_k>$. Denote by t_s the root of T_s ($s = 1, ..., l$). We use *top-down*(t, $<p_1, l>$) to represent *top-down*(T, G), which is designed to check T and G top-down. For a given G, two cases are recognized:

Case 1: $G = <P_1>$; or $G = <P_1, ..., P_l>$ ($l > 1$), but $|T| \leq |P_1| + |P_2|$. (That is, G is a forest containg only a single tree or a proper forest but the size of the first two subtrees is equal to or larger than the size of T.)

Case 2: $G = <P_1, ..., P_l>$ ($l > 1$), and $|T| > |P_1| + |P_2|$.

In *Case* 1, what we can do is to find a left corner within P_1, which can be embedded in T. This is done as follows:

i) If t is a leaf node, we will check whether label(t) = label($\delta(p_1)$) (note that p_1 is the root of P_1.) If it is the case, set $\kappa(t)$ to be a triplet $[\delta(p_1), 1]$ and return $<\delta(p_1), 1>$. Otherwise, set $\kappa(t)$ to be $[\delta(p_1), 0]$ and return $<\delta(p_1), 0>$.

ii) If $|T| < |P_1|$ or $h(t) < h(p_1)$, we will make a recursive call *top-down*(t, $<p_{11}, j>$), where p_{11} is the left-most child of p_1 and $j = d(p_1)$. So $<p_{11}, j>$ represents a forest of the subtrees of p_1: $<P_{11}, ..., P_{1j}>$. The return value $<v, i>$ of *top-down*(t, $<p_{11}, j>$) is used as the return value of *top-down*(t, $<p_1, l>$).

iii) If $|T| \geq |P_1|$ and $h(t) \geq h(p_1)$, we further distinguish between two cases:
  - label(t) = label(p_1). In this case, we will call *bottom-up*($<t_1, k>$, $<p_{11}, j>$), by which $<P_{11}, ..., P_{1j}>$ will be checked against $<T_1, ..., T_k>$.
  - label(t) ≠ label(p_1). In this case, we will call *bottom-up*($<t_1, k>$, $<p_1, 1>$), by which P_1 will be checked against $<T_1, ..., T_k>$.

In both cases, assume that the return value is $<v, i>$. A further checking needs to be conducted:

- If label(t) = label(v's parent) and $i = d(v$'s parent), the return value should be $<v$'s parent, 1>. Set $\kappa(t)$ to be [v's parent, 1].
- Otherwise, the return value remains $<v, i>$. Set $\kappa(t)$ to be $[v, i]$.

In *Case* 2, we try to find a left corner within $G = <P_1, ..., P_l>$, which can be embedded in T. This is done by calling *bottom-up*($<t_1, k>$, $<p_1, l>$). Assume that the return value is $<v, i>$. The following checkings will be continually conducted.

iv) If $v = p_1$, the return value of *top-down*(t, $<p_1, l>$) is the same as $<v, i>$.

v) If $v \neq p_1$, check whether label(t) = label(v's parent) and $i = d(v)$. If it is the case, the return value will be changed to $<v$'s parent, 1>, and $\kappa(t)$ is set to be [v's parent, 1]. Otherwise, the return value remains $<v, i>$, and $\kappa(t)$ is set to be $[v, i]$.

The following is the formal description of the algorithm *top-down*(t, $<p_1, l>$), in which we assume that each node v has a link to its direct sibling, making a sibling chain. Starting from p_1, we can access $p_1, ..., p_l$ along the sibling chain.

Function *top-down*(t, $<p_1, l>$)

input: t - stands for $T = <t; T_1, ..., T_k>$, $<p_1, l>$ - for $G = <P_1, ..., P_l>$.
output: $<v, i>$ specified above.

```
begin
1.    if (l = 1 or |T[t]| ≤ |G[p1]| + |G[p2]|)
2.    then { let p11 be the left-most child of p1; let j be d(p1);
                                        (*Case 1*)
3.        if t is a leaf then {if label(t) = label(δ(p1))
                                  then i := 1 else i := 0;
4.                                κ(t) := [δ(p1), i]; return <δ(p1), i>;}
5.        if (|T[t]| < |G[p1]| or h(t) < h(p1))
6.        then {<v, i> := top-down(t, <p11, j>); return <v, i>;}
7.        if label(t) = label(p1)  (*|T| ≥ |P1| and h(t) ≥ h(p1)*)
8.        then {if p1 is a leaf then {v := p1; i := 1;}
9.                else {<v, i> := bottom-up(<t1, k>, <p11, j>);
10.                  if label(t) = label(v's parent) and
                         i = d(v's parent)
                     then {v := v's parent; i := 1;}
11.               }
12.               else <v, i> := bottom-up(<t1, k>, <p1, 1>);
                     (*If label(t) ≠ label(p1), call bottom-up( ).*)
13.               κ(t) := [v, i]; return <v, i>;
14.           }
15.       else  {<v, i> := bottom-up(<t1, k>, <p1, l>);
                                        (*Case 2*)
16.               if v ≠ p1 then { p := v's parent;
17.                               if (label(t) = label(p)) and i = d(p)
18.                               then {v := p; i := 1; }
19.                               κ(t) := [v, i];
20.               }
21.               return <i, v>;
22.           }
end
```

The above algorithm mainly consists of two parts: lines 2 - 14 for *Case* 1, and lines 15 - 22 for *Case* 2. In the first part, we first handle the case that T contains only a single node (see lines 3 - 4); and then the case that $|T| < |P_1|$ or $h(t) < h(p_1)$ (see lines 5 - 6). The lines 7 - 14 are devote to the case that $|T| \geq |P_1|$ and $h(t) \geq h(p_1)$. If $\text{label}(t) = \text{label}(p_1)$, we need to check whether p_1 is a leaf node. If it is the case, return $<p_1, 1>$ (see line 8). Otherwise, the bottom-up procedure will be invoked to check $<P_{11}, ..., P_{1j}>$ against $<T_1, ..., T_k>$ (see line 9). If $\text{label}(t) \neq \text{label}(p_1)$, the bottom-up procedure is invoked to check P_1 against $<T_1, ..., T_k>$ (see line 12).

In the second part, the bottom-up procedure is invoked to check $<T_1, ..., T_k>$ against $<P_1, ..., P_l>$ (see line 15). Finally, We notice that each time a node t is checked $\kappa(t)$ is changed to a new value, which is the return value of the current *top-down* execution (see lines 4, 13, and 19).

bottom-up(F, G) is designed to handle the case that both F and G are forests with each containing some subtrees rooted at a set of consecutive siblings in the target and the pattern, respectively. Let $F = <T_1, ..., T_k>$. We use *bottom-up*$(<t_1, k>, <p_1, l>)$ to represent *bottom-up*(F, G). In *bottom-up*$(<t_1, k>, <p_1, l>)$, we will make a series of calls of the form *top-down*$(t_i, <p_{j_i}, l - j_i + 1>)$, where $j_1 = 1$, and $j_1 \leq j_2 \leq ... \leq j_h \leq l$ (for some $h \leq k$), controlled as follows.

1. Two index variables s, j are used to scan $t_1, ..., t_k$ and $p_1, ..., p_l$, respectively. (Initially, s is set to 1, and j is set to 0.) They also indicate that $<P_1, ..., P_j>$ has been successfully embedded in $<T_1, ..., T_s>$.
2. Let $<v_s, i_s>$ be the return value of *top-down*$(t_s, <p_{j+1}, l - j>)$. If $t_s = p_{j+1}$, set j to be $j + i_s$. Otherwise, j is not changed. Set s to be $s + 1$. Go to (2).
3. The loop terminates when all T_s's or all P_j's are examined.

See Fig. 7. for illustration.

If $j > 0$ when the loop terminates, *bottom-up*$(<t_1, k>, <p_1, l>)$ returns $<p_1, j>$.

Otherwise, $j = 0$. In this case, we will continue to search for a left corner $<v, i>$ in G, which can be embedded in F, as described below.

i) Let $<v_1, i_1>, ..., <v_k, i_k>$ be the return values of *top-down*$(t_1, <p_1, l>)$, ..., *top-down*$(t_k, <p_1, l>)$, respectively. Since $j = 0$, each v_f ($f = 1, ..., k$) must be a descendant of p_1 and on the left-most path in P_1.

ii) If each $i_f = 0$ ($f = 1, ..., k$), return $<\delta(p_1), 0>$. Otherwise, there must be some $<v_f, i_f>$'s such that $i_f > 0$. We call such a v_f a *non-zero point*. Find the first non-zero point v_f such that v_f is not a descendant of any other non-zero point. Let $w_1, ..., w_h$ be the right siblings (in this order) of v_f. We will further check $<T_{f+1}, ..., T_k>$ against $<G[w_{i_f}], G[w_{i_f+1}]..., G[w_h]>$. This can be done in the same way as described above. But it is not necessary to record the highest non-zero point. If it is found that $<T_{f+1}, ..., T_k>$ embeds the first q subtrees in $<G[w_{i_f}], G[w_{i_f+1}]..., G[w_h]>$, the return value of *bottom-up*$(<t_1, k>, <p_1, l>)$ is set to be $<v_f, i_f + q>$. Otherwise, the return value is $<v_f, i_f>$.

In this process, a node t in F may be checked multiple times due to the second checking described in (ii). In order to avoid any possible redundancy, we define a simple function as below.

Let v, v' be two nodes in G. Define

$$\beta(v, v') = \begin{cases} \textit{true}, & \text{if } v = v', \text{ or } \delta(v) = \delta(v') \text{ and } v' \text{ is an ancestor of } v; \\ \textit{false}, & \text{otherwise.} \end{cases}$$

During the execution of *bottom-up*( ), this function will be used each time we make a call of the form *top-down*$(t, <p, l>)$ for a node t in F. Let $\kappa(t) = [v, i]$. If $\beta(v, p) = \textit{true}$, we simply set the return value of *top-down*$(t, <p, l>)$ to be $<v, i>$ and *top-down*$(t, <p, l>$ is not actually executed. It is because $<v, i>$ is the highest left corner of some forest in G that can be embedded in $F[t]$, and therefore for any ancestor p of v with $\delta(v) = \delta(p)$ a call of the form *top-down*$(t, <p, l>)$ will definitely return $<v, i>$.

Obviously, if p is a descendant of v and $i > 0$, the return value should be $<p, l>$. But if $i = 0$, the return value is $<p, 0>$.

In terms of the above discussion, we give the following algorithm to implement the bttom-up procedure, in which a subprocedure *td-checking*( ) is invoked to check a T_i against a forest $<P_{j_i}, ..., P_l>$, including the redundancy checking by using $\kappa(t)$'s.

Function *bottom-up*($<t_1, k>, <p_1, l>$)
input: $<t_1, k>$ - stands for $F = <T_1, ..., T_k>$,
$<p_1, l>$ - for $G = <P_1, ..., P_l>$.
output: $<v, i>$ specified above.
begin
1. $s := 1; j := 0; \quad t := t_1; p := p_1; \tau_f := 1; v_f := \phi; i_f := 0;$
(*ϕ is considered to be a descendant of any node.*)
2. **while** ($j < l$ and $s \leq k$) **do**(*first checking*)
3. { $<v, i> :=$ *td-checking*(t, p, j, l);
4. **if** ($v = p$ and $i > 0$) **then** {$j := j + i; p := p_{j+1};$}
(*navigate along the sibling chain to find p_{j+i+1}.*)
5. **else if** v is an ancestor of v_f **then** {$v_f := v; i_f := i; \tau_f := s;$}
(*record the highest non-zero point.*)
6. $s := s + 1; t := t_s;$
(*navigate one step along the sibling chain to find t_{s+1}.*)
7. }
8. **if** $j > 0$ **then** return $<p_1, j>$;
9. **if** $i_f = 0$ **then** return $<\delta(p_1), 0>$
10. let $d(v_f$'s parent$) = c$; find v_f's $(i_f + 1)th$ right sibling w_{i_f};
(*Let $w_1, ..., w_c$ be the right siblings of v_f.*)
11. $x := \tau_f + 1; y := i_f; t := t_{\tau_f + 1}; p := w_{i_f};$
12. **while** ($y < c$ and $x \leq k$) **do**(*second checking*)
13. { $<v, i> :=$ *td-checking*(t, p, y, c);
14. **if** ($v = p$ and $i > 0$) **then** {$y := y + i; p := w_{y+1};$}
15. $x := x + 1; t := t_x;$
16. }
17. **if** $y > 0$ **then** return $<v_f, i_f + y>$ **else** return $<v_f, i_f>$;
18.}
end

Function *td-checking*(t, p, j, l)
input: t - a node in F; p - a node in G; j, l - two integers with $j \leq l$.
output: $<v, i>$ specified above.
begin
1. let $\kappa(t) = [\gamma, \eta]$;
2. **if** $\beta(\gamma, p) = true$ **then** {$v := \gamma; i := \eta;$}
3. **else** {**if** p is a descendant of γ
then {$v := p$; if $\eta = 0$ **then** $i := 0$ else $i := l - j$;}
4. **else** $<v, i> :=$ *top-down*$(t, <p, l - j>)$;
5. }
6. return $<v, i>$;
end

In *bottom-up*( ), the variables s and j are used to scan $T_1, ..., T_k$ and $P_1, ..., P_l$, respectively, while the variables t and p are used to store the roots of the current T_s and P_{j+1} (see line 1). The variables v_f and i_f are for storing the highest non-zero point, and τ_f is for the root of the corresponding T_f.

As described above, the algorithm involves two times of checkings. The first checking is done in lines 2 - 7 while the second checking is conducted in lines 10 - 16. Whether the second checking will be carried out depends on the checking result performed in lines 8 and 9. First, in lines 2 - 7, we do a series of checkings of T_i against $<P_{j_i}, ..., P_l>$ ($i = 1, ..., h$, $1 \leq h \leq k$) and each is done by calling *td-checking*( ) (see line 3), in which $\kappa(t)$'s are checked to eliminate redundancy (see lines 2 - 3 in *td-checking*( )). Line 5 is devoted to the computation of the highest non-zero point $<v_f, i_f>$.

If $j > 0$, the return value of *bottom-up*($<t_1, k>, <p_1, l>$) is $<p_1, j>$ (see line 8). If $j = 0$ and $i_f = 0$, the return value is $<\delta(p_1), 0>$ (see line 9). In both cases, the second checking will not carry on. Therefore, we call the following condition the *second-checking* condition:

$j = 0$ and $i_f > 0$.

If the above condition holds, the second checking will be conducted (see lines 10 - 16). This is almost the same as line 2 - 7. But no computation is arranged to record the highest non-zero point. In line 17, we calculate the return value for the case of $j = 0$.

Example 1 Consider the tree T and the forest G shown in Fig. 6. As indicated by the dashed lines, we have an ordered embedding of a subtree of G in T.

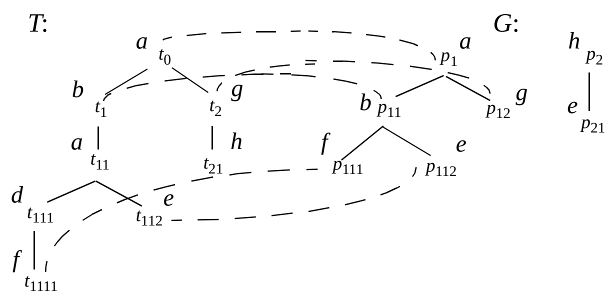

Fig. 6. A target tree and a pattern tree

In Fig. 6, each node in T is identified with t_i, such as t_0, t_1, t_{11}, and so on; and each node in G is identified with p_j. Besides, each subtree rooted at t_i (p_j) is represented by T_i (resp. P_j). In Fig. 7, we trace the computation process when applying the algorithm to T and G. In this figure, a solid arrow represents a subprocedure call while each dashed arrow represents a return value. Associated with a solid arrow is the condition under which the subprocedure is invoked.

The return value of the whole procedure is $<p_1, 1>$, showing that T contains P_1.

From the sample trace, we can see that a node in T can be checked multiple times, but against different nodes in G. For instance, t_{112} is first checked against p_{111}, and then against p_{112}. t_2 is also checked two times, against p_{111} and p_{12}, respectively.

4 Computational complexities

In this section, we analyze the computational complexities of the algorithm.

In the algorithm discussed in the previous section, a node t in F may be involved in multiple calls of the form *top-down*(t, $<p, l>$) due to a possible second checking in *bottom-up*( ).

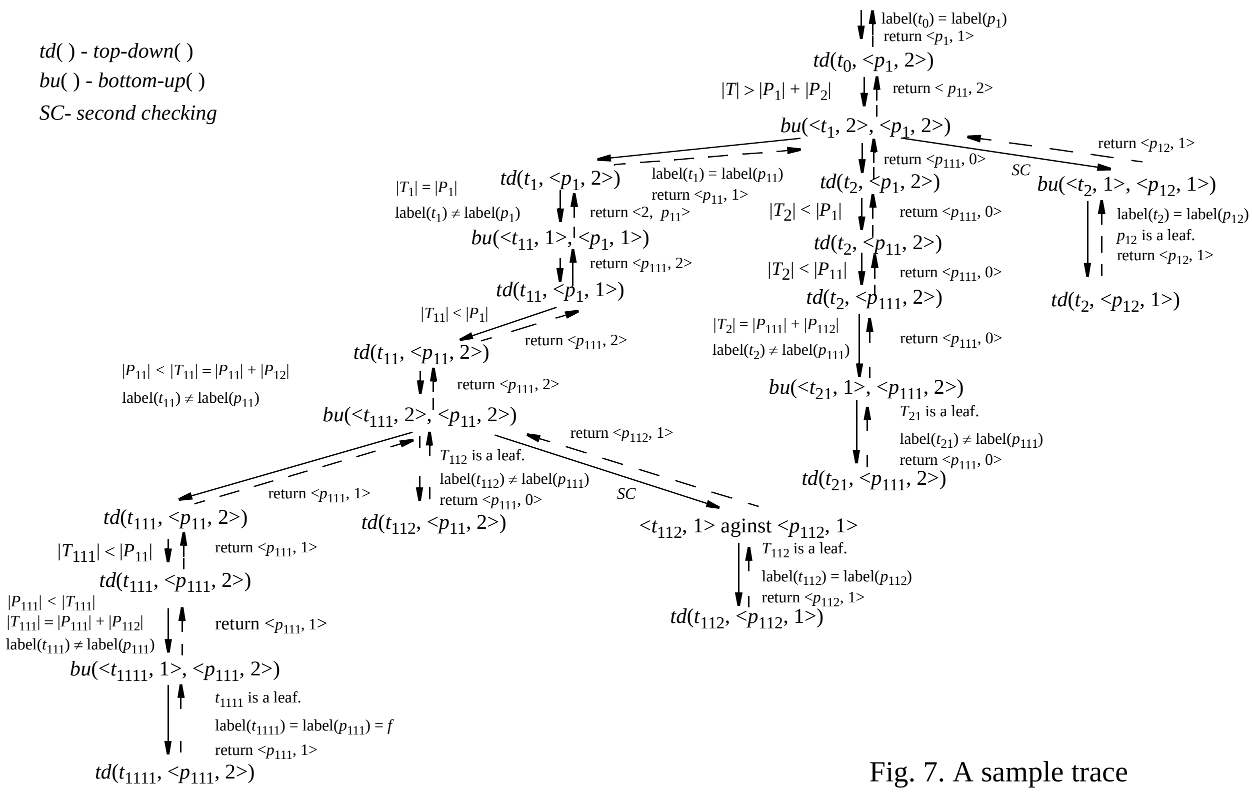

Fig. 7. A sample trace

In the algorithm discussed in the previous section, a node t in F may be involved in multiple calls of the form *top-down*(t, <p, l>) due to a possible second checking in *bottom-up*( ). We denote by [t, p] each of such calls for simplicity. We further distinguish two kinds of [t, p]'s. During a [t, p] of the first kind, t is checked against a node in G, which is done in line 3, line 7, or in line 17 in *top-down*( ).

During a [t, p] of the second kind, we navigate to the left-most child of p if p is not a leaf node (see line 6.)
First, we estimate the number of the calls of the first kind. Without loss of generality, assume that the first [t, p] is invoked by executing line 3 in *bottom-up*( ) to check <P_1, ..., P_l> against <T_1, ..., T_k>. It is possible for t to be involved in a second subprocedure call [t, p'] (see line 13 in *bottom-up*( )). Obviously, p' must be a descendant of p. Also, p' cannot be a node on the left-most path in $G[p]$ due to the second-checking condition: $j = 0$ and $i_f > 0$, where <v_f, i_f> is the first highest non-zero point and $j = 0$ indicates that even P_1 cannot be embedded in <T_1, ..., T_k>.

Since $j = 0$, v_f must be a node on the left-most path in $G[p]$. But its (i_f + 1)*th* right sibling is definitely not on such a path (see line 10 in *bottom-up*( )). So p' is not on the left-most path in $G[p]$.

Now we consider a child t_j of t. Clearly, during the execution of [t, p], t_j can also be involved in two subprocedure calls [t_j, u_1] and [t_j, u_2] while during the execution of [t, p'] t_j can be involved in another two subprocedure calls [t_j, u_1'] and [t_j, u_2']. As discussed above, u_2 cannot be on the left-most path in $G[u_1]$, and u_2' cannot be on the left-most path in $G[u_2]$. Concerning u_2 and u_1', we claim that

u_1' is a node appearing in a subtree to the right of u_2.

Below we show this property.

Consider all the left siblings t_s of t. Let <v_s, i_s> be the return value of the corresponding *top-down*(t_s, <p, l>). Let <v, i> be the return value of *top-down*(t, <p, l>). We distinguish among three cases:

i) For any <v_s, i_s>, v_s is a descendant of v.
ii) There is at least one non-zero point v_s (*i.e.*, $i_s > 0$), which is an ancestor of v and not a descendant of any other non-zero point.
iii) There is at least one non-zero point $v_s = v$, which is not a descendant of any other non-zero point.

In case (i), t will not be checked for a second time at all since by the second checking it must be a forest (or a tree) with the first subtree rooted a node to the right of t against a forest (or a tree) in G.

In case (ii), p' must be a node appearing in a subtree to the right of v_s while u_2 is definitely a node in the subtree rooted at v or at a jth right sibling of v with $j \le i - 1$, and therefore a descendant of v_s. Since u_1' is in the subtree rooted at p', it is to the right of u_2. for illustration.)

In case (iii), we have $v_s = v$. If $i_s \geq i$, p' is definitely to the right of u_2, and so is u_1'. (See Fig. 10(b) for illustration.) In the following, we analyze the case when $i_s < i$.

Let $t_1, ..., t_{j-1}$ be all the left sibling of t_j. Consider $v_1 = v$ and all its right siblings $v_2, ..., v_l$. If u_2 is a node in a subtree rooted at v_q with $q \leq i_s$, u_1' must be a node to the right of u_2. Otherwise, assume that u_2 is a node in a subtree rooted at $v_{q'}$ with $i_s < q' \leq i$. Then, we have $<F[t_1], ..., F[t_{j-1}]>$ embedding $<G[v_1], ..., F[v_{q'-1}]>$. Therefore, $<F[t_1], ..., F[t_{j-1}]>$ must embed $<G[v_{i_s+1}], ..., F[v_{q'-1}]>$. Thus, p' can be $v_{q'}$ or to the right of $v_{q'}$. If p' is $v_{q'}$, u_1' can be an ancestor of u_2, equal to u_2, or a descendant of u_2. (Also, see Fig. 10(c) for illustration.) In any case, the corresponding checking is skipped by using $\kappa(t_j)$. If p' is to the right of $v_{q'}$, u_1' must be to the right of u_2.

The above discussion shows that the claim concerning u_2 and u_1' holds.

Mapping u_1 (u_1') to a node on the left-most path in $G[u_1]$ ($G[u_1']$), we think that t_j is involved in four $[t, v]$'s with each v on a different path in G. So we claim that the number of the first kind of calls is bounded by $O(|T|\cdot|\text{leaves}(G)|)$.

Now we consider the second kind of *top-down* calls. For each t in T, corresponding to a checking of it against a node in G, a downward segment in G may be searched; and for any of its children a segment following that segment may also be searched. So corresponding to a path in T, for all the checkings of the nodes on that path with each checked once, a path in G may be navigated. According to the above analysis, however, a node in T may be checked against different nodes on different paths in G. So the number of the second kind of calls is bounded by $O(|\text{leaves}(T)|\cdot|P|)|$.

Proposition 3 The time complexity of the algorithm is bounded by $O(|T|\cdot|\text{leaves}(G)| + |\text{leaves}(T)|\cdot|P|)|)$.

Proof. See the above analysis. □

Since in the working process no extra data structure is used, we have the following proposition.

Proposition 3 The space complexity of the algorithm is bounded by $O(|T| + |G|)$.

Proof. It is trivially true. □

5 Conclusion

In this paper, a new algorithm is proposed to evaluate XML queries based ordered tree matching, by which not only the ancestor/descendant and parent/child relationships, but also the left-to-right order of nodes are considered. The algorithm mainly contains two functions: *Top-down*( ) and *Bottom-up*( ). Each of them returns a left corner to indicate a subtree (subforest) embedding. This arrangement enables us to use a simple data structure to record intermediate results to avoid redundancy. The time complexity of the new algorithm is bounded by $O(|T|\cdot|\text{leaves}(P)| + |P|\cdot|\text{leaves}(T)|)$ while the space requirement is bounded by $O(|T| + |P|)$, where T and P are a target and a pattern tree, respectively.

References

[1] N. Bruno, N. Koudas, and D. Srivastava, Holistic Twig Joins: Optimal XML Pattern Matching, in *Proc. SIGMOD Int. Conf. on Management of Data*, Madison, Wisconsin, June 2002, pp. 310-321.

[2] B. Catherine and S. Bird, Towards a general model of Interlinear text, in *Proc. of EMELD Workshop*, Lansing, MI, 2003.

[3] T. Chen, J. Lu, and T.W. Ling, On Boosting Holism in XML Twig Pattern Matching, in: *Proc. SIGMOD*, 2005, pp. 455-466.

[4] Y. Chen, A time optimal algorithm for evaluating tree pattern queries, *SAC 2010*, ACM, 1638-1642.

[5] Y. Chen, Donovan Cooke: XPath query evaluation based on the stack encoding, *C3S2E 2009*, IEEE, 43-57

[6] Y. Chen: Unordered Tree Matching and Tree Pattern Queries in XML Databases, *ICSOFT (2) 2009*: 191-198.

[7] Y. Chen, An Efficient Streaming Algorithm for Evaluating XPath Queries. in *Proc. WEBIST*, 2008, pp. 190-196.

[8] Y. Chen, S.B. Davison, Y. Zheng, An Efficient XPath Query Processor for XML Streams, in *Proc. ICDE*, Atlanta, USA, April 3-8, 2006.

[9] Sayyed Kamyar Izadi, Theo Härder, Mostafa S. Haghjoo, S^3: Evaluation of Tree-Pattern Queries Supported by Structural Summaries, *Data & Knowledge Engineering*, 68, pp. 126-145, Elsevier, Sept. 2008.

[10] D.E. Knuth, *The Art of Computer Programming, Vol. 1 (1st edition)*, Addison-Wesley, Reading, MA, 1969.

[11] R.B. Lyngs, M. Zuker & C.N.S. Pedersen, Internal loops in RNA secondary structure prediction, in *Proceedings of the 3rd annual international conference on computational molecular biology (RECOMB)*, 260-267 (1999).

[12] Q. Li and B. Moon, Indexing and Querying XML data for regular path expressions, in: *Proc. VLDB*, Sept. 2001, pp. 361-370.

[13] Y. Rui, T.S. Huang, and S. Mehrotra, Constructing table-of-content for videos, *ACM Multimedia Systems Journal, Special Issue Multimedia Systems on Video Libraries*, 7(5):359-368, Sept 1999.

[14] L. Qin, J.X. Yu, and B. Ding, “TwigList: Make Twig Pattern Matching Fast,” In *Proc. 12th Int’l Conf. on Database Systems for Advanced Applications (DASFAA)*, pp. 850-862, Apr. 2007.

[15] H. Wang, S. Park, W. Fan, and P.S. Yu, ViST: A Dynamic Index Method for Querying XML Data by Tree Structures, *SIGMOD Int. Conf. on Management of Data*, San Diego, CA., June 2003.

[16] World Wide Web Consortium. XML Path Language (XPath), W3C Recommendation, 2007. See http://www.w3.org/TR/xpath20.

[17] World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, Jan. 2007. See http://www.w3.org/TR/xquery.

[18] M. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of KDD*, 2002.