On the Query Evaluation in Document DBs

Yangjun Chen
Department of Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, Canada R3B 2E9
ychen2@uwinnipeg.ca

Abstract In this paper, we study the query evaluation in document databases. First, we show that a query represented in an XML language can be generally considered as a labeled tree, and the evaluation of such a query is in fact a tree embedding problem. Then, we propose a strategy to solve this problem, based on dynamic programming. For the ordered tree embedding, the proposed algorithm needs only $O(|T|\cdot|P|)$ time and $O(|T|\cdot|P|)$ space, where $|T|$ and $|P|$ stands for the numbers of the nodes in the target tree T and the pattern tree P, respectively. This computational complexity is better than any existing method on this issue. In addition, how to adapt this method to the general tree embedding is also discussed.

1 Introduction

In XML, data is represented as a tree; associated with each node of the tree is an element type from a finite alphabet $\sum$. The children of a node are ordered from left to right, and represent the content (i.e., list of subelements) of that element. XML queries such as XPath, XQuery, XML-QL and Quilt use tree patterns to extract relevant portions from the input database. A tree pattern query (or called a query tree) that we consider in this paper, denoted by *TPQ* from now on, is defined as follows. The nodes of a tree are labeled by element types from $\sum \cup \{*\}$, where * is a wild card, matching any element type. The type for a node v is denoted $\tau(v)$. There are two kinds of edges: child edges (c-edges) and descendant edges (d-edges). A c-edges 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 *c-child* of v. A d-edge is denoted $v \Rightarrow u$ in the text, and represented by a double arc; u is called a *d-child* of v.

In any DAG (*directed acyclic graph*), a node u is said to be a descendant of a node v if there exists a path (sequence of edges) from v to u. In the case of a TPQ, this path could consist of any sequence of c-edges and/or d-edges.

An embedding of a TPQ P into an XML document T is a mapping f: $P \rightarrow T$, from the nodes of P to the nodes of T, which satisfies the following conditions:

1. Preserve node type: For each $v \in P$, v and $f(v)$ are of the same type.
2. Preserve c/d-child relationships: If $v \rightarrow u$ in P, then $f(u)$ is a child of $f(v)$ in T; if $v \Rightarrow u$ in P, then $f(u)$ is a descendant of $f(v)$ in T.

Any document T, in which P can be embedded, is said to contain P and considered

The author is supported by NSERC 239074-01 (242523) (Natural Science and Engineering Council of Canada).

to be an answer.

To handle all the possible XPath queries, we allow a node u in a TPQ P to be associated with a set of predicates. We distinguish among three different kinds of predicates: *current node related predicates* (called *current*-predicates), *child node related predicates* (called *c*-predicates), and *descendant related predicates* (called *d*-predicates). A *current*-predicate p is just a built-in predicate applied to the current node; i.e., a node v in T, which matches u, must satisfy this predicate associated with u. A *c*-predicate is a built-in predicate applied to the children of the current node. That is, for each node v in T, which matches u, each of its children (or one of its children) must satisfy this predicate. Similarly, a *d*-predicate must be satisfied by all the descendants of the node (or one of its descendants), which matches u. Without loss of generality, we assume that associated with u is a conjunctive-disjunctive normal form: $(p_{11} \vee ... \vee p_{1i_1}) \wedge ... \wedge (p_{k1} \vee ... \vee p_{ki_k})$, where each p_{ij} is a predicate.

For example, the following XPath query:

chapter[section[//paragraph[text() contains ‘informatics’]/following-sibling::*][position() = 3]]/*[self::section or self::chapter-notes]

can be represented by a tree shown in Fig. 1.

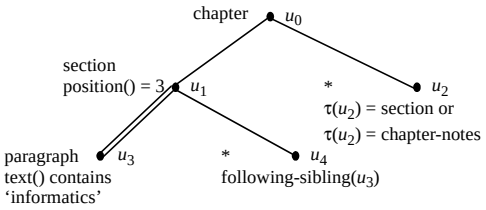

Fig. 1. A sample TPQ

In the query tree shown in Fig. 1, each node is labeled with a type or *, and may or may not be associated with a conjunctive-disjunctive normal form of predicates, which are used to describe the conditions that the node (and/or its children) has to satisfy, or the relationships of the node with some other nodes:

- u_0 - $\tau(u_0)$ = chapter. It matches any node v in T if it is associated with type ‘chapter’.
- u_1 - $\tau(u_1)$ = section; and associated with a current predicate position() = 3. It matches any node v in T if it is a third child of its parent and associated with type ‘section’.
- u_2 - $\tau(u_2)$ = *; and associated with a disjunction of current-predicates: $\tau(u_2)$ = section or $\tau(u_2)$ = chapter-notes. It matches a node v in T if it is associated with type ‘section’ or ‘chapter-notes’.
- u_3 - $\tau(u_3)$ = paragraph; associated with a *c*-predicate: text() contains ‘informatics’. It matches a node v in T if it is associated with type ‘paragraph’ and has a text child that contains word ‘informatics’.

u_4 - $\tau(u_4)$ = *; associated with a current-predicate: following-sibling(v_3), which indicates that if u_4 match a node in T, that node must directly follows any node that matches u_3, i.e., any node with type 'paragraph' and having a text child node that contains word 'informatics'.

Accordingly, the embedding f of a TPQ P into a document T is modified as follows.

1. For each $v \in P$, v and $f(v)$ are of the same type; and $f(v)$ satisfies all the *current*-predicates associated with v.
2. If $v \rightarrow u$ in P, then $f(u)$ is a child of $f(v)$ in T; and $f(u)$ satisfies all the c-predicates associated with v. If $v \Rightarrow u$ in P, then $f(u)$ is a descendant of $f(v)$ in T; and $f(u)$ satisfies all the d-predicates associated with v.

Recently, much research has been conducted on the evaluation of such XML queries [1, 5, 6, 7, 8]. Here, we just mention some of them, which are very closely related to the work to be discussed. The first one is based on *Inversion on elements and words* [8], which needs $O(n^m)$ time in the worst case where n and m are the number of the nodes in T and P, respectively. The second is based on *Inversion on paths and words* [5], which improves the first one by introducing indexes on paths. The time complexity of this method is still exponential and needs $O((n \cdot h)^k)$ time in the worst case, where h is the average height of a document tree and k is the number of joins conducted. The main idea of the third method is to transform a tree embedding into a string matching problem [6, 7]. The time complexity is $O(n \cdot m \cdot h)$. This polynomial time complexity is achieved by imposing an ordering on the siblings in a query tree. That is, the method assumes that the order of sib-lings is significant. If the query tree is ordered differently from the documents, a tree embedding may not be found even though it exists. In this case, the query tree should be reordered and evaluated once again. Another problem of [6] is that the results may be incorrect. That is, a document tree that does not contain the query tree may be designated as one of the answers due the *ambiguity* caused by identical sibling nodes. This problem is removed by the so-called forward prefix check-ing discussed in [7]. Doing so, however, the theoretical time complexity is dra-matically degraded to $O(n^2 \cdot m \cdot h)$. The last one is to represent an XPath query as a parse tree and evaluate such a parse tree bottom-up or top-down [1]. In [1], it is claimed that the bottom-up strategy needs only $O(n^5 \cdot m^2)$ time and $O(n^4 \cdot m^2)$ space, so does its top-down algorithm. But in another paper [2] of the same authors, the same problem is claimed to be NP-complete. It seems to be controversial. In fact, the analysis made in [1] assumes that the query tree is ordered while by the analy-sis conducted in [2] the query tree is considered to be unordered, leading to different analysis results.

In this paper, we present a new algorithm based on the ordered tree embedding. Its time complexity is bounded by $O(n \cdot m)$.

2 A strategy based on Ordered-tree embedding

In this section, we mainly discuss a strategy for the query evaluation based on the

ordered tree embedding, by which the order between siblings is significant. The query evaluation based on the unordered tree embedding is discussed in the next section.

In general, a tree pattern query P can be considered as a labeled tree if we extend the meaning of label matching by including the predicate checking. That is, to check whether a node v in a document T matches a node u in P, we not only compare their types, but also check whether all the predicates associated with u can be satisfied. Such an abstraction enables us to focus on the *hard* part of the problem.

In the following, we first give the basic definitions over the ordered tree embedding in 3.1. Then, we propose an algorithm for solving this problem in 3.2.

2.1 Basic concepts

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 out-degree 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 direct 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 (excluding v) is denoted by $desc(v)$. A leaf is a node with an empty set of descendants. The children of a node v is denoted by $chidren(v)$.

Sometimes we treat a tree T as the forest $<T>$. We also denote the set of nodes in a forest F by $V(F)$. For example, if we speak of functions from a forest F to a forest G, we mean functions mapping $V(F)$ onto $V(G)$. 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 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 checked as follows.

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 2.3.2-2 in [4]. □

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

Lemma 2. Let v and u be nodes in a forest F. Then, v appears on the left side of u if and only if $pre(v) < pre(u)$ and $post(v) < post(u)$.

Proof. The proof is trivial. □

Now we give the definition of ordered tree embeddings. In this definition, we simply use 'label matching' to refer to both type matching and predicate checking.

Definition 1. Let P and T be rooted labeled trees. We define an ordered embedding (f, P, T) as an injective mapping f: $V(P) \rightarrow V(T)$ such that for all nodes $v, u \in V(P)$,

i) label(v) = label($f(v)$); (label preservation condition)
ii) if (v, u) is a c-edge, then $f(v)$ is the parent of $f(u)$; (child condition)
iii) if (v, u) is a d-edge, then $f(v)$ is an ancestor of $f(u)$; (ancestor condition)
iv) v is to the left of u iff $f(v)$ is to the left of $f(u)$. (Sibling condition) □

As an example, we show an ordered tree embedding in Fig. 2.

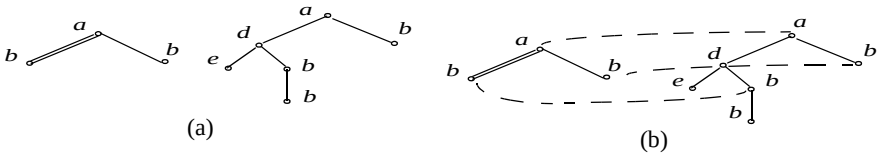

Fig. 2. An example of an ordered tree embeding

In Fig. 2(a), the tree on the left can be embedded in the tree on the right because a mapping as shown in Fig. 2(b) can be recognized, which satisfies all the conditions specified in Definition 1. In addition, Fig. 2(b) shows a special kind of tree embeddings, which is very critic to the design of our algorithm and also quite useful to explain the main idea of our design.

Definition 2. Let P and T be trees. A *root-preserving* embedding of P in T is an embedding f of P in T such that $f(\text{root}(P)) = \text{root}(T)$. If there is a root-preserving embedding of P in T, we say that the root of T is an occurrence of P. □

For example, the tree embedding shown in Fig. 4(b) is a root preserving embedding. Obviously, restricting to root-preserving embedding does not lose generality.

Finally, we use Lemma 2 to define an ordering of the nodes of a forest F given by $v \prec v'$ iff $post(v) < post(v')$ and $pre(v) < pre(v')$. Also, $v \preceq v'$ iff $v \prec v'$ or $v = v'$. The *left relatives*, lr(v), of a node $v \in V(F)$ is the set of nodes that are to the left of v (i.e., all those nodes that precede v both in preorder and postorder), and similarly the *right relatives*, rr(v), is the set of nodes that are to the right of v (i.e., all those nodes that follow v both in preorder and postorder).

Throughout the rest of the paper, we refer to the labeled trees simply as trees since we do not discuss unlabeled trees at all.

2.2 Algorithm description

The algorithm to be given is in fact a dynamic programming solution. During the process, two $m \times n$ ($m = |P|$, $n = |T|$) matrices are maintained and computed to discover tree embeddings. They are described as follows.

1. The nodes in both P and T are numbered in postorder, and the nodes are then referred to by their postorder numbers.
2. The first matrix is used to record subtree embeddings, in which each entry c_{ij} ($i \in \{1, ..., m\}$, $j \in \{1, ..., n\}$) has value 0 or 1. If $c_{ij} = 1$, it indicates that there is a root preserving embedding of the subtree rooted at the node indexed by i (in P) in the subtree rooted at the node indexed by j (in T). Otherwise, $c_{ij} = 0$. This matrix is denoted by $c(P, T)$.
3. In the second matrix, each entry d_{ij} ($i \in \{1, ..., m\}$, $j \in \{0, ..., n - 1\}$) is defined as follows:

   $d_{ij} = \min(\{x \in \text{rr}(j) \mid c_{ix} = 1\} \cup \{\alpha\})$,

   where $\alpha = n + 1$. That is, d_{ij} contains the closest right relative x of node j such that $T[x]$ contains $P[i]$, or $n + 1$, indicating that there exists no right relative x of node j such that $T[x]$ contains $P[i]$. This matrix is denoted by $d(P, T)$.

In the above definitions of matrices, we should notice that the indexes of $d(P, T)$ is slightly different from those of $c(P, T)$. That is, for $d(P, T)$, $j \in \{0, ..., n - 1\}$ (instead of $\{1, ..., n\}$), and $j = 0$ is considered to be a virtual node to the left of any node in T.

The matrix $c(P, T)$ is established by running the following algorithm, called *ordered-tree-embedding* while $d(P, T)$ is employed to facilitate the computation. Initially, $c_{ij} = 0$, and $d_{ij} = 0$ for all i and j. In addition, each node v in T is associated with a quadruple $(\alpha(v), \beta(v), \chi(v), \delta(v))$, where $\alpha(v)$ is v's preorder number, $\beta(v)$ is v's postorder number, $\chi(v)$ is v's level number, and $\delta(v) = \min(desc(v))$. By the level number of v, we mean the number of ancestors of v, excluding v itself. For example, the root of T has the level number 0, its children have the level number 1, and so on. Obviously, for two nodes v_1 and v_2, associated respectively with $(\alpha_1, \beta_1, \chi_1, \delta_1)$ and $(\alpha_2, \beta_2, \chi_2, \delta_2)$, if $\chi_2 = \chi_1 + 1$, $\alpha_1 < \alpha_2$ and $\beta_1 > \beta_2$, we have $v_2 \in children(v_1)$.

In the following algorithm, we assume that for T there exists a virtual node with postorder number 0, which is to the left to any node in T. This is a modified version of the algorithm described in [3], adapted to handling of different kinds of edges (c-edges and d-edges).

Algorithm *ordered-tree-embedding*(T, P)
Input: tree T (with nodes 0, 1, ..., n) and tree P (with nodes 1, ..., m)
Output: $c(P, T)$, which shows the tree embedding.
begin
1. **for** $u := 1, ..., m$ **do**
2. {**for** $v := 0, ..., n - 1$ **do** {$d_{uv} := n + 1$;}

```
3.    l := 0;
4.    for v := 1, ..., n do
5.     {if label(u) = label(v) then
6.       let u_1, ..., u_k be the children of u;
7.       j := δ(v) - 1;
8.       i := 1;
9.       while i ≤ k and j < v do
10.        {j := d_{u_i, j};
11.          if (u, u_i) is a d-edge then
12.            {if j ∈ desc(v) then i := i + 1;
13.          else /*(u, u_i) is a c-edge.*/
14.            {if j ∈ children(v) and j is a c-child then i := i + 1;}
15.        }
16.    if j = k then
17.     {c_uv := 1;
18.      while l ∈ lr(v) do {d_ul ; = v; l := l + 1;}
19.     }
20.   }
end
```

To know how the above algorithm works, we should first notice that both T and P are postorder-numbered. Therefore, the algorithm proceeds in a bottom-up way (see line 1 and 4). For any node u in P and any node v in T, if label(u) = label(v), the children of u will be checked one by one against some nodes in $desc(v)$. The children of u is indexed by i (see line 6); and the nodes in $desc(v)$ is indexed by j (see line 10). Assume that the nodes in $desc(v)$, which are checked during the execution of the while-loop (see lines 9 - 15), are $j_1, ..., j_h$. Then, for each j_g ($1 \le g \le h$), the following conditions are satisfied (see line 10):

(i) $\delta(v) \le j_g < v$ (i.e., $j_g \in desc(v)$),

(ii) There exists u_i such that $j_g = d_{u_i, j_{g-1}}$ with $j_0 = \delta(v) - 1$.

Therefore, for any j_a and $j_b \in \{j_1, ..., j_h\}$, they must be on different paths according to the definition of $d(P, T)$. In addition, in the while-loop, if (u, u_i) is a d-edge, the algorithm checks whether $j \in desc(v)$ (see line 12). If it is the case, u_i has a matching counterpart in $desc(v)$ and i will be increased by 1. Thus, in a next step, the algorithm will check the direct right sibling of u_i against a node in the right relatives of j. If (u, u_i) is a c-edge, we will check whether $j \in children(v)$ and j is c-child (see line 14). If $i = k$ (i.e., $desc(v)$ contains all subtrees $P[u_1], ..., P[u_k]$), we will have a root-preserving embedding of $P[u]$ in $T[v]$. Therefore, c_{uv} is set to 1 (see line 17). Also, for any node l in the left relatives of v, d_{ul} is set to v (see line 18). It is because v must be the closest right relative of any of such nodes in T such that the subtree rooted at it (i.e, $T[v]$) root-preservingly contains $P[u]$.

Example 1. As an example, consider the trees shown in Fig. 3. The nodes in them are postorder numbered.

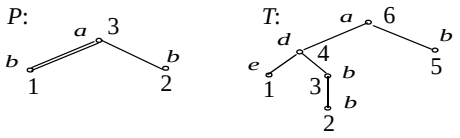

Fig. 3. Labeled trees and postorder numbering

When we apply the algorithm to these two trees, $c(P, T)$ and $d(P, T)$ will be created and changed in the way as illustrated in Fig. 4, in which each step corresponds to an execution of the outmost for-loop.

step 1:

$c(P, T)$:

	1	2	3	4	5	6
1	0	1	1	0	1	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0

$d(P, T)$:

	0	1	2	3	4	5
1	2	2	5	5	5	7
2	0	0	0	0	0	0
3	0	0	0	0	0	0

step 2:

$c(P, T)$:

	1	2	3	4	5	6
1	0	1	1	0	1	0
2	0	1	1	0	1	0
3	0	0	0	0	0	0

$d(P, T)$:

	0	1	2	3	4	5
1	2	2	5	5	5	7
2	2	2	5	5	5	7
3	0	0	0	0	0	0

step 3:

	1	2	3	4	5	6
1	0	1	1	0	1	0
2	0	1	1	0	1	0
3	0	0	0	0	0	1

	0	1	2	3	4	5
1	2	2	5	5	5	7
2	2	2	5	5	5	7
3	6	7	7	7	7	7

Fig. 4. Sample trace

In step 1, we show the values in $c(P, T)$ and $d(P, T)$ after node 1 in P is checked against every node in T. Since node 1 in P matches node 2, 3 and 5 in T, c_{12}, c_{13}, and c_{15} are all set to 1. Furthermore, d_{10} is set to 2 since the closest right relative of node 0 in T, which matches node 1 in P, is node 2 in T. The same analysis applies to d_{11}. Since the closest right relative of node 2, 3, 4 in T, which matches node 1 in P, is node 5 in T, d_{12}, d_{13}, and d_{14} are all set to 5. Finally, we notice that d_{14} is equal to 7, which indicates that there exists no right relative of node 5 that matches node 1 in P.

In step 2, the algorithm generates the matrix entries for node 2 in P, which is done in the same way as for node 1 in P.

In step 3, node 3 in P will be checked against every node in T, but matches only node 6 in T. Since it is an internal node (in fact, it is the root of P), its children will be further checked. First, to check its first child, the algorithm will examine d_{10}, which is equal to 2, showing that node 2 in T is the closest right relative of node 0 that matches node 1 in P. In a next step, the algorithm will check the second child of node 3 in P. To do this, d_{22} is checked. d_{22}'s value is 5, showing that the closest relative of node 2 in T, which matches node 2 in P, is node 5 in T. In addition, since the edge (3, 2) in P is a c-edge, the algorithm will check whether node 5 in T is not only a child of node 6, but also a c-child. Since it is the case, we have a root-pre-

serving embedding of $P[3]$ in $T[6]$. Finally, we notice that when the second child of node 3 in P is checked, the algorithm begins the checking from d_{22} rather than d_{20}. In this way, a lot of useless checkings is avoided. □

Proposition 1. Algorithm *ordered-tree-embedding*(T, P) computes the values in $c(P, T)$ and $d(P, T)$ correctly.

Proof. The proposition can proved by by induction on the sum of the heights of T and P. □

Proposition 2. Algorithm *ordered-tree-embedding*(T, P) requires $O(n \cdot m)$ time and space, where $n = |T|$ and $m = |P|$.

Proof. During the execution of the outermost for-loop, l may increases from 0 to n. Therefore, the time spent on the execution of line 18 in the whole process is bounded by $O(n)$. An execution of the while-loop from line 9 to 15 needs $O(d_u)$ time, where d_u represents the outdegree of node u in P. So the total time is bounded by

$$O(n) + O(\sum_{u=1}^{m}\sum_{v=1}^{n} d_u) = O(n) + O(\sum_{v=1}^{n}\sum_{u=1}^{m} d_u)$$

$$= O(n) + O(\sum_{v=1}^{n} m) = O(n \cdot m).$$

Obviously, to maintain $c(P, T)$ and $d(P, T)$, we need $O(n \cdot m)$ space. □

3 On the evaluation of general tree pattern queries

In this section, we briefly discuss how to use the algorithm for ordered tree embedding to evaluate general tree pattern queries. For this, we need to consider the following problem:

> The ordering of siblings in a pattern (query) tree may be different from that in a target (document) tree.

In order to tackle this problem, we will change the sibling order in a query according to DTD if it is available. If the corresponding DTD does not exist, we store the document trees according to the lexicographical order of the names of the elements/attributes. Whenever a query arrives, the query tree will be constructed according to the same order. However, in the case that a branch has multiple identical child nodes, the tree isomorphism problem cannot be avoided by enforcing sibling order. For example, a query of the form: /X[Y/Z/B]/Y/A can be represented as a tree shown in Fig. 5(a) or a tree shown in Fig. 5(b).

Fig. 5. Tree pattern queries

In this case, a sibling order cannot be specified lexicographically or by DTD schema. In order to find all matches, we have to check these two trees separately and unify their results.

4 Conclusion

In this paper, a new strategy for evaluating XPath queries is discussed. The main idea of the strategy is to handle an XPath query as tree embedding problem, by which the label matching includes both the type matching the predicate satisfaction. A dynamic programming method is proposed to check the ordered tree embedding, by the ordering of siblings is important. The algorithm needs only $O(|T|\cdot|P|)$ time and $O(|T|\cdot|P|)$ space, where $|T|$ and $|P|$ stands for the numbers of the nodes in the target tree *T* and the pattern tree *P*, respectively. Finally, how to adapt this method to the unordered tree embedding is briefly discussed.

References

[1] G. Gottlob, C. Koch, and R. Pichler, Efficient Algorithms for Processing XPath Queries, *ACM Transaction on Database Systems*, Vol. 30, No. 2, June 2005, pp. 444-491.

[2] G. Gottlob, C. Koch, and K.U. Schulz, Conjunctive Queries over Trees, in *Proc. PODS 2004*, June 2004, Paris, France, pp. 189-200.

[3] Pekka Kilpelainen and Heikki Mannila. Ordered and unordered tree inclusion. *SIAM Journal of Computin*g, 24:340-356, 1995.

[4] D.E. Knuth, *The Art of Computer Programming, Vol. 1*, Addison-Wesley, Reading, MA, 1969.

[5] C. Seo, S. Lee, and H. Kim, An Efficient Index Technique for XML Documents Using RDBMS, *Information and Software Technology* 45(2003) 11-22, Elsevier Science B.V.

[6] 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.

[7] H. Wang and X. Meng, On the Sequencing of Tree Structures for XML Indexing, in *Proc. Conf. Data Engineering*, Tokyo, Japan, April, 2005, pp. 372-385.

[8] C. Zhang, J. Naughton, D. DeWitt, Q. Luo and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems, in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, California, USA, 2001