An Efficient Algorithm for Tree Mapping in XML Databases

Yangjun Chen
University of Winnipeg, Winnipeg, Manitoba, Canada R3B 2E9

Abstract: In this article, we discuss an efficient algorithm for tree mapping problem in XML databases. Given a target tree $T$ and a pattern tree $Q$, the algorithm can find all the embeddings of $Q$ in $T$ in $O(|T||Q|)$ time while the existing approaches need exponential time in the worst case.

Key words: Tree mapping, XML databases, query evaluation, tree encoding

INTRODUCTION

XML uses a tree-structured model for representing data. Queries in XML languages (such as XPath\cite{1}, Xquery\cite{2,3}, XML-QL\cite{4} and Quilt\cite{5,6}) also typically specify selection patterns as a kind of tree-structured relations. For instance, the XPath expression:

```
book[title = 'Art of Programming']//author[fn = 'Donald' and ln = 'Knuth']
```

matches `author` elements that (i) have a child subelement `fn` with content ‘Donald’, (ii) have a child subelement `ln` with content ‘Knuth’ and are descendants of `book` elements that have a child `title` subelement with content ‘Art of Programming’. This expression can be represented as a tree structure as shown in Fig. 1.

![Fig. 1: A query tree](image)

In this tree structure, a node $v$ is labeled with an element name or a string value, denoted $\text{label}(v)$. In addition, there are two kinds of edges: child edges ($c$-edges) for parent-child relationships and descendant edges ($d$-edges) for ancestor-descendant relationships. A $c$-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 $c$-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 $v$. Such a query is often called a twig pattern.

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 twig pattern, this path could consist of any sequence of $c$-edges and/or $d$-edges. Based on these concepts, the tree embedding can be defined as follows.

**Definition 1:** An embedding of a twig pattern $Q$ into an XML document $T$ is a mapping $f: Q \rightarrow T$, from the nodes of $Q$ to the nodes of $T$, which satisfies the following conditions:

i. Preserve node label: For each $u \in Q$, $u$ and $f(u)$ are of the same label (or more generally, $u$’s predicate is satisfied by $f(u)$.)

ii. Preserve $c/d$-child relationships: If $u \rightarrow v$ in $Q$, then $f(v)$ is a child of $f(u)$ in $T$; if $u \Rightarrow v$ in $Q$, then $f(v)$ is a descendant of $f(u)$ in $T$.

If there exist a mapping from $Q$ into $T$, we say, $Q$ can be imbedded into $T$, or say, $T$ contains $Q$.

Notice that an embedding could map several nodes of the query (of the same type) to the same node of the database. It also allows a tree mapped to a path. This definition is quite different from the tree matching defined in\cite{7}.

There is much research on how to find such a mapping efficiently and all the proposed methods can be categorized into two groups. By the first group\cite{2,8-17}, a tree pattern is typically decomposed into a set of binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations. Then, an index structure is used to find all the matching pairs that are joined together to form the final result. By the second group\cite{18-23}, a query pattern is decomposed into a set of paths. The final result is constructed by joining all the matching paths together. For all these methods, the join operations involved require exponential time in the worst case. For example, if we decompose a twig pattern into paths to find all the matching paths from a...
To facilitate the checking of reachability (whether a node can be reached from another node through a path), a tree encoding is used\(^{24}\).

Consider a tree \(T\). By traversing \(T\) in preorder, each node \(v\) will obtain a number \(\text{pre}(v)\) to record the order in which the nodes of the tree are visited. In a similar way, by traversing \(T\) in postorder, each node \(v\) will get another number \(\text{post}(v)\). These two numbers can be used to characterize the ancestor-descendant relationships as follows.

Let \(v\) and \(v'\) be two nodes of a tree \(T\). Then, \(v'\) is a descendant of \(v\) iff \(\text{pre}(v') > \text{pre}(v)\) and \(\text{post}(v') < \text{post}(v)\)\(^{24}\).

As an example, have a look at the pairs associated with the nodes of the tree shown in Fig. 2. The first element of each pair is the preorder number of the root nodes in \(T\) and in \(Q\), respectively.

**TREES ENCODING**

**Algorithm for Simple Cases**

Here, we describe an algorithm for simple cases that a twig pattern contains only \(d\)-edges. First, we give a basic algorithm to show the main idea in 3.1. Then, in 3.2, we discuss how this algorithm can be substantially improved. In 3.3, we prove the correctness of the algorithm and analyze its computational complexities.

**Basic algorithm:** The basic algorithm to be given works in a bottom-up way. During the process, two data structures are maintained and computed to facilitate the discovery of subtree matchings.

- Each node \(v\) in \(T\) is associated with a set, denoted \(\alpha(v)\), contains all those nodes \(g\) in \(Q\) such that \(Q[g]\) can be imbedded into \(T[v]\), where \(T[v]\) represents a subtree of \(T\) rooted at \(v\).
- Each \(g\) in \(Q\) is associated with a value \(\delta(q)\), defined as follows:
  - Initially, for each \(g\) in \(Q\), \(\delta(q)\) is set to \(\phi\). During the tree matching process, \(\delta(q)\) is dynamically changed as below:
    1. Let \(v\) be a node in \(T\) with parent node \(u\).
    2. If \(q\) appears in \(\alpha(v)\), change the value of \(\delta(q)\) to \(u\).

    Then, each time before we insert \(q\) into \(\alpha(v)\), we will do the following checkings:
    (i) Check whether \(\text{label}(q) = \text{label}(v)\).
    (ii) Let \(q_1, \ldots, q_k\) be the child nodes of \(q\). For each \(q_i\) (\(i = 1, \ldots, k\)), check whether \(\delta(q_i)\) is equal to \(v\) or to a descendant of \(v\).

    If both (1) and (2) are satisfied, insert \(q\) into \(\alpha(v)\).

    Below is a bottom-up algorithm, working in a recursive way and taking a node \(v\) in \(T\) as the input (which represents \(T[v]\)). Initially, the input is the root of \(T\). The algorithm will mark any node \(u\) in \(T[v]\) if it finds that \(T[u]\) contains \(Q\). In the process, two functions are called:
    - \(\text{node-check}(u, q)\) - It checks whether \(T[u]\) contains \(Q[q]\). If it is the case, return \{\(q\}\}. Otherwise, it returns an empty set \(\emptyset\).
Algorithm tree-matching(v)
input: v - a node of tree T
output: mark any node u in T[v] if T[u] contains Q.
begin
1. \( S := \emptyset; S_1 := \emptyset; S_2 := \emptyset; \)
2. if v is not a leaf node in T then
3. \( \{ \text{let } v_1, ..., v_k \text{ be the child nodes of } v; \} \)
4. for i = 1 to k do call tree-matching(v_i);
5. \( \alpha := \alpha(v_1) \cup ... \cup \alpha(v_k); \)
6. for each q \( \in \alpha \) do
7. \( \{ \delta(q) := v; S := S \cup \{ q's \text{ parent}\}; \} \)
8. remove all \( \alpha(v)(i = 1, ..., k); \)
9. for each q in S do
10. \( S_1 := S_1 \cup \text{node-check}(v, q); \)
11. \} \)
12. \( S_2 := \text{leaf-node-check}(v); \)
13. \( \alpha(v) := \alpha \cup S_1 \cup S_2; \)
end

Function node-check(u, q)
begin
1. \( S_1 := \emptyset; \)
2. if label(q) = label(u) then
3. \( \{ \text{let } q_1, ..., q_k \text{ be the child nodes of } q; \} \)
4. if for each q_i (i = 1, ..., k) \( \delta(q_i) \) is equal to u
5. or to a descendant of u
6. then \( \{ S_1 := S_1 \cup \{ q\}; \} \)
7. if q is root then mark u';
8. return \( S_1; \)
end

Function leaf-node-check(u)
begin
1. \( S_2 := \emptyset; \)
2. for each leaf node q in Q do
3. \( \{ \text{if } type(q) = type(u) \{ S_2 := \{ q\}; \} \)
4. if q is root then mark u;
5. return \( S_2; \)
end

The algorithm tree-matching( ) searches T bottom-up in a recursive way (see line 4). During the process, for each encountered node v in T, we first check whether it is a leaf node (see line 2). If it is a leaf node, the function leaf-node-check( ) is called (see line 12), by which all the matching leaf nodes in Q will be stored in a temporary variable \( S_2 \) that will be added to \( \alpha(v) \) (see line 13). If v is an internal node, lines 3 - 10 are first conducted and then the function leaf-node-check( ) is invoked (see line 12). By executing line 4, tree-matching( ) is recursively called for each child node \( v_i \) of v. After that, for each q appearing in \( \alpha(v_i) \), its \( \delta(v) \) value is set to be v (see line 7). In addition, q's parent is inserted into \( S_1 \), a temporary variable to be used in the next step. Since \( \alpha(v_i) \)'s will not be used any more after this step, they are simply removed (see line 8). By executing lines 9 - 10, we check, for each \( q' \) in \( S_1 \), whether v matches \( q' \) by calling node-check( ), in which the \( \delta(v) \) values of q's child nodes are utilized to facilitate the checkings (see lines 3 - 5 in node-check( )). The following example helps for illustration.

Example 1: Consider \( T \) and \( Q \) shown in Fig. 3.

The algorithm works in a bottom-up way. First, \( v_3 \) in \( T \) is visited. It is a leaf node, matching \( q_3 \) of the two leaf nodes in \( Q \). Therefore, \( \alpha(v_3) = \{ q_3 \} \) (see lines 12). In the same way, we will set \( \alpha(v_5) = \{ q_2 \} \).

\[
\begin{align*}
T: & \quad a \rightarrow v_1 \\
& \quad b \rightarrow v_2 \\
& \quad c \rightarrow v_3 \\
Q: & \quad a \rightarrow q_1 \\
& \quad b \rightarrow q_2 \\
& \quad c \rightarrow q_3
\end{align*}
\]

Fig. 3: A document tree and a query tree

In a next step, \( v_4 \) is encountered. It is the parent of \( v_5 \). In terms of \( \alpha(v_5) = \{ q_2 \} \), \( \delta(q_2) \) is set to be \( v_4 \) (Fig. 4). After that, node-check(\( v_4, q_1 \)) is invoked. (Note that \( q_1 \) is the parent of \( q_2 \).) Since lines 9 - 10.) Since \( \text{label}(v_4) \neq \text{label}(q_1) \), it returns \( S_1 = \emptyset \). leaf-node-check(\( v_4 \)) also returns \( S_2 = \emptyset \). So \( \alpha(v_4) = \alpha(v_3) \cup S_1 \cup S_2 = \{ q_2 \} \) (see line 13). When \( v_2 \) is met, we will first set \( \delta(q_3) = \delta(q_1) = v_2 \) (in terms of \( \alpha(v_4) = \{ q_2 \} \) and \( \alpha(v_3) = \{ q_3 \} \), respectively). Next, we call node-check(\( v_2, q_1 \)), in which we will check whether \( \text{label}(v_2) = \text{label}(q_1) \). It is the case. So we will further check whether \( \delta(q_i) (i = 2, 3) \) is equal to \( v_2 \). Since both \( \delta(q_2) \) and \( d(q_3) \) are equal to \( v_2 \), we have that \( T[v_2] \) contains \( Q[q_1] \). Therefore, \( S_1 = \{ q_1 \} \). Thus, we set \( \alpha(v_2) = \alpha(v_3) \cup \alpha(v_4) \cup S_1 \cup S_2 = \alpha(v_3) \cup \alpha(v_4) \cup \{ q_1 \} \cup \emptyset = \{ q_1, q_2 \} \).

\[
\begin{align*}
T: & \quad a \rightarrow v_1 \\
& \quad b \rightarrow v_2 \\
& \quad c \rightarrow v_3 \\
Q: & \quad a \rightarrow q_1 \\
& \quad b \rightarrow q_2 \\
& \quad c \rightarrow q_3
\end{align*}
\]

Fig. 4: Sample trace
In a next step, we will meet \( v_6 \). It is a leaf node, matching \( q_3 \). Therefore, \( \alpha(v_6) = \{q_3\} \). Finally, we will meet \( v_4 \) and set \( \delta(q_1) = v_1 \) and \( \delta(q_3) = v_3 \). Since \( \text{label}(v_1) = \text{label}(q_1) \), \( \delta(q_2) = \delta(q_3) = v_1 \), we have that \( T[v_1] \) contains \( Q[q_1] \) and \( \alpha(v_1) = \{q_1, q_2, q_3\} \).

**Improvements:** The above algorithm can be substantially improved by elaborating the construction of \( \alpha(v) \)’s.

First, we notice that in the case that \( v \) is a leaf node in \( T \), \( \alpha(v) \) is a set of the leaf nodes in \( Q \), which match \( v \). Such nodes can be stored in a linked list as illustrated below:

```
q_1 --> ... --> q_4
```

Fig. 5: Linked list to store

with the left-most node appearing first and the rightmost node last. Then, for any \( 1 \leq i \leq j \leq k \), we have \( \text{pre}(q_i) < \text{pre}(q_j) \) and \( \text{post}(q_i) < \text{post}(q_j) \). That is, in \( \alpha(v) \), \( q_i \)’s are sorted according to their preorder and postorder values.

Now we consider two \( \alpha \)-lists \( \alpha \) and \( \alpha' \) sorted according to their nodes’ preorder and postorder numbers. Define a merging operation over \( \alpha \) and \( \alpha' \), denoted \( \text{merge}(\alpha, \alpha') \), as follows:

1. Assume that \( \alpha = \{v_1, ..., v_p\} \) and \( \alpha' = \{v'_1, ..., v'_q\} \). We step through both \( \alpha \) and \( \alpha' \) from left to right. Let \( v_i \) and \( v'_j \) be the nodes encountered. We’ll make the following checkings.
2. If \( \text{pre}(v_i) > \text{pre}(v'_j) \) and \( \text{post}(v_i) > \text{post}(v'_j) \), insert \( v'_j \) into a after \( v_i \) and before \( v_j \) and move to \( v_{i+1}' \) (in \( \alpha' \)).
3. If \( \text{pre}(v_i) > \text{pre}(v'_j) \) and \( \text{post}(v_i) < \text{post}(v'_j) \), remove \( v_j \) from \( \alpha' \) and move to \( v_{i+1} \). *(\( v_i \) is subsumed by \( v'_j \)).*
4. If \( \text{pre}(v_i) < \text{pre}(v'_j) \) and \( \text{post}(v_i) > \text{post}(v'_j) \), ignore \( v'_j \) and move to \( v_{j+1} \) (in \( \alpha' \)). *(\( v'_j \) is subsumed by \( v_i \)).*
5. If \( \text{pre}(v_i) < \text{pre}(v'_j) \) and \( \text{post}(v_i) < \text{post}(v'_j) \), ignore \( v_i \) and move to \( v_{i+1} \).
6. If \( \text{pre}(v_i) = \text{pre}(v'_j) \) and \( \text{post}(v_i) = \text{post}(v'_j) \), ignore both \( v_i \) and \( v'_j \) and move to \( v_{i+1} \) and \( v_{j+1} \) in \( \alpha \) and \( \alpha' \), respectively.

The result of \( \text{merge}(\alpha, \alpha') \) is stored in \( \alpha \) and \( \alpha' \) remains unchanged. Especially, the changed \( \alpha \) is still sorted according to their nodes’ preorder and postorder numbers.

In terms of the above discussion, we have the following algorithm to merge two sorted \( \alpha \)-lists together.

**Algorithm merge(\( \alpha, \alpha' \))**

- **Input:** \( \alpha \) and \( \alpha' \) - sorted \( \alpha \)-lists.
- **Output:** modified \( \alpha \), containing all the nodes in \( \alpha \) and \( \alpha' \) with all the subsumed nodes removed.

```
begin
1. \( p \leftarrow \text{first-element}(\alpha) \);
2. \( q \leftarrow \text{first-element}(\alpha') \);
3. while \( p \neq \text{nil} \) do
4. \( \text{while } q \neq \text{nil} \text{ do} \)
5. if \( \text{pre}(p) > \text{pre}(q) \land \text{post}(p) > \text{post}(q) \)
6. then \( \{ \text{insert } q \text{ into } \alpha \text{ before } p; \}
7. \( q \leftarrow \text{next}(q) \); (*\( \text{next}(q) \) represents the node next to \( q \) in \( \alpha' \).*)
8. else if \( \text{pre}(p) > \text{pre}(q) \land \text{post}(p) < \text{post}(q) \)
9. then \( \{ p \leftarrow p; (*p \text{ is subsumed by } q; \}
10. remove \( p \) from \( \alpha \).
11. \( p \leftarrow \text{next}(p) \); (*\( \text{next}(p) \) represents the node next to \( p \) in \( \alpha' \).*
12. else if \( \text{pre}(p) < \text{pre}(q) \land \text{post}(p) > \text{post}(q) \)
13. then \( \{ q \leftarrow \text{next}(q) \}; (*q \text{ is subsumed by } p; \}
14. move to the node next to \( q \).
15. \( p \leftarrow \text{next}(p) \);
16. else if \( \text{pre}(p) = \text{pre}(q) \land \text{post}(p) = \text{post}(q) \)
17. then \( \{ p \leftarrow \text{next}(p); \}
18. \( q \leftarrow \text{next}(q) \}; \}
end
```

We can extend the merging operation over to more than two sorted \( \alpha \)-lists:

\[
\text{merge}(\alpha_1, ..., \alpha_{k-1}, \alpha_k) = \text{merge}(\text{merge}(\alpha_1, ..., \alpha_{k-1}), \alpha_k).
\]

Using this operation, the algorithm \text{tree-matching( )} is rewritten as follows.

**Algorithm tree-matching(\( v \))**

- **Input:** \( v \) - a node of tree \( T \).
- **Output:** mark any node \( u \) in \( T[v] \) if \( T[u] \) contains \( Q \).

```
begin
1. \( S := \emptyset ; \)
2. if \( v \) is not a leaf node in \( T \) then
3. \{\text{let } v_1, ..., v_k \text{ be the child nodes of } v; \}
4. for \( i = 1 \) to \( k \) do call \text{tree-matching}(v_i); 
5. \( \alpha := \text{merge}(\alpha(v_1), ..., \alpha(v_k)); \)
6. assume that \( \alpha = \{q_1, ..., q_j\}; \)
```
This algorithm is almost the same as the previous one, but with the merge operation involved, which effectively reduces the size of each \( \alpha(v) \) from \( O(Q) \) to \( O(Q_{\text{leaf}}) \). Special attention should also be paid to line 7, by which we generate a set \( S \) that contains the parent nodes of all those nodes appearing in \( \alpha(v_i)'s \) \((i = 1, \ldots, k) \), where \( v_i \) is a child node of the current node \( v \). Since the nodes in \( \alpha (\alpha = \text{merge}(\alpha_1, \ldots, \alpha_k)) \) are left-to-
right sorted (according to the nodes’ preorder and postorder numbers), if there are more than one nodes in a sharing the same parent, they must appear consecutively in the list. So each time we insert a parent node \( q' \) (of some \( q \) in \( \alpha \)) into \( S \), we need to check whether it is the same as the previously inserted one. If it is the case, \( q' \) will be ignored. Thus, the size of \( S \) is also bounded by \( O(Q_{\text{leaf}}) \).

**Correctness and computational complexity:** In this subsection, we prove the correctness of the algorithm tree-matching() and analyze its computational complexities.

**Proposition 1:** Let \( v \) be a node in \( T \). Then, for each \( q \) in \( \alpha(v) \) generated by tree-matching(), we have \( T[v] \) contains \( Q[q] \).

**Proof:** We prove the proposition by induction on the height of \( Q \), \( \text{height}(Q) \).

**Basic step:** When \( \text{height}(Q) = 1 \), the proposition trivially holds.

**Induction step:** Assume that the proposition holds for any query tree \( Q' \) with \( \text{height}(Q') \leq h \). We consider a query tree \( Q \) of height \( h + 1 \). Let \( r_0 \) be the root of \( Q \). Let \( q_1, \ldots, q_k \) be the child nodes of \( r_0 \). Then, we have \( \text{height}(Q[q_j]) \leq h \) \((j = 1, \ldots, k) \). In terms of the induction hypothesis, for each \( q \) in \( Q[q_j] \) \((j = 1, \ldots, k) \), if it appears in \( \alpha(v) \) (where \( v_i \) is a child node of \( v \)), we have \( T[v_j] \) contains \( Q[q] \) and \( \delta(q) \) will be set to be \( v \) before \( v \) is checked against \( r_0 \). Obviously, if \( \text{label}(v) = \text{label}(r_0) \) and for each \( q \) \((j = 1, \ldots, k) \), \( \delta(q) \) is equal to \( v \) or a descendant of \( v \), \( Q \) can be embedded into \( T[v] \). So \( r_0 \) is inserted into \( \alpha(v) \).

Now we consider the time complexity of the algorithm, which can be divided into four parts:

1. The first part is the time spent on merging \( \alpha(v_1), \ldots, \alpha(v_k) \), where \( v_i \) \((i = 1, \ldots, k) \) is a child node of some node \( v \) in \( T \). This part of cost is bounded by

   \[
   O(\sum_{j=1}^{k} d_{Q_{\text{leaf}}}) = O(|T|Q_{\text{leaf}}),
   \]

   where \( d \) represents the outdegree of a node \( v_i \) in \( T \).

2. The second part is the time used for generating \( S \) from a merged \( \alpha \)-list. Since the size of the \( \alpha \)-list is bounded by \( O(Q_{\text{leaf}}) \), this part of cost is also bounded by \( O(Q_{\text{leaf}}) \).

3. The third part is the time for checking a node \( v_i \) in \( T \) against each node \( q_j \) in an \( S \). Denote \( S_i \) the set of the nodes in \( Q \), which are checked against \( v_i \). We estimate this part of cost by the following sum:

   \[
   O(\sum_{j=1}^{k} c_j) = O(|T|Q_{\text{leaf}}),
   \]

   where \( c_j \) represents the outdegree of a node \( q_j \) in \( S_i \).

4. The fourth part is the time for checking each node \( v \) in \( T \) against the leaf nodes in \( Q \). Obviously, this part of cost is bounded by

   \[
   O(\sum_{j=1}^{k} Q_{\text{leaf}}) = O(|T|Q_{\text{leaf}}).
   \]

In terms of the above analysis, we have the following proposition.

**Proposition 2:** The time complexity of tree-matching() is bounded by \( O(|T|Q_{\text{leaf}}) \).

**Proof:** See the above discussion.

Since at each time point at most \( T_{\text{leaf}} \) nodes in \( T \) are associated with a \( \alpha \)-list, the space overhead is bounded by \( O(T_{\text{leaf}}Q_{\text{leaf}}) \).

**GENERAL CASES**

The algorithm discussed earlier can be easily extended to general cases that a query tree contains both \( c \)-edges and \( d \)-edges. We only need to make the following changes:

- For each child node \( q \) of \( q \) that is being checked against \( v \), if \( (q, q_i) \) is a \( c \)-edge, we will check whether \( \delta(q) \) is equal to \( v \). If \((q, q_i)\) is a \( d \)-edge, we simply check whether \( \text{pre}(\delta(q)) \geq \text{pre}(v) \) and \( \text{post}(\delta(q)) \leq \text{post}(v) \).

- Accordingly, the algorithm node-check described earlier should be slightly modified.
Function general-node-check(u, q) begin
1. \( S_1 := \emptyset \);
2. if label(q) = label(u) then
3. \{ let \( q_1, \ldots, q_k \) be the child nodes of \( q \);
4. flag := true; \( i := 1 \);
5. while \( i \leq k \land \text{flag} \) do
6. \{ if \( ((q, q_i) \) is a c-edge) \land \( (\delta(q_i) = v) \) \lor
7. \( ((q, q_i) \) is a d-edge) \land
8. \( (\text{pr}e(\delta(q_i)) \geq \text{pre}(u)) \land
9. \( (\text{post}(\delta(q_i)) \leq \text{post}(u))\) \}
10. then flag := false; \}
11. if \( i > g \) then \( \{ S_1 := S_1 \cup \{ q \} \};\)
12. if \( q \) is root then mark u; \}
13. return \( S_1 \);
end

This algorithm is similar to the function node-check(). The only difference is that a general subsumption checking process is used, by which c-edges and d-edges are checked in different ways.

In addition, the lines 5 - 10 in the algorithm tree-matching() given in 3.2 should be replaced with the following segment of code:

for \( i = 1 \) to \( k \) do {
for \( q \in \alpha(v_i) \) do {
let \( q' \) be the parent of \( q \);
if \( ((q', q) \) is a c-edge) \then
\{ \( \delta(q) := v \);
let \( q'' \) be the last element in \( S \);
if \( q'' \) is parent \( \neq q' \) \then
\{ \( S := S \cup \{ q'' \} \};\}
else remove \( q \) from \( \alpha(v_i) \);
\} \}
\( \alpha := \text{merge}(\alpha(v_1), \ldots, \alpha(v_i)) \);

Concerning the correctness of the algorithm, we have to answer a question: whether any c-edge in \( Q \) is correctly checked.

First, we note that any c-edge in \( Q \) cannot be matched to any path with length larger than 1 in \( T \). That is, it can be matched only to a single edge in \( T \). It is exactly what is done by the algorithm.

Each time we check a node \( v \) in \( T \) against some \( q \) in \( Q \), we will first set \( d \) values for any \( q_i \) appearing in \( \alpha(v_j) \)’s, where \( v_j \) is a child node of \( v \). When doing this, for some \( q_i \)'s, their \( \delta \) values are changed (to \( v \)). Assume that the current \( \delta \) value for \( q_i \) is \( v' \) (i.e., \( \delta(q_i) = v' \)). Then, \( v' \) must be a descendant of \( v \) since the algorithm searches \( T \) in a bottom-up way. However, we need to change \( \delta(q_i) \) from \( v' \) to \( v \) since a c-edge can match only a single edge in \( T \) and the fact that \( q_i \) matches \( v \) should be recorded so that the c-edge matching is not missed (see Fig. 6 for illustration).

In Fig. 6, \( v'' \) is a descendant of \( v \) and matches \( q_2 \). So \( \delta(q_2) \) will be set to \( v'' \). However, \( (q_i, q_2) \) is a c-edge. Therefore, the fact that \( v'' \) matches \( q_2 \) makes no contribution to the matching of \( v \) with \( q \). Since \( q_2 \) also matches \( v_2 \), \( \delta(q_2) \) will be changed to \( v \), which enables us to find that \( T[v] \) contains \( Q[q] \).

Proof: See the above discussion.

The time and space complexities for the general cases are the same as for the simple cases.

CONCLUSION

In this article, a new algorithm is proposed for a kind of tree matching, the so-called twig pattern matching. This is a core operation for XML query processing. The main idea of the algorithm is to explore both \( T \) and \( Q \) bottom-up, by which each node \( q \) in \( Q \) is associated with a value (denoted \( \delta(q) \)) to indicate a node \( v \) in \( T \), which has a child node \( v' \) such that \( T[v'] \) contains \( Q[q] \). In this way, the tree embedding can be checked very efficiently. In addition, by using the tree encoding, as well as the subsumption checking mechanism, we are able to minimize the size of the lists of the matching query nodes associated with the nodes in \( T \) to reduce the space overhead. The algorithm runs in \( O(|T| Q_{leaf}) \) time and \( O(T_{leaf} Q_{leaf}) \) space, where \( T_{leaf} \) and \( Q_{leaf} \) represent the numbers of the leaf nodes in \( T \) and in \( Q \), respectively. More importantly, no costly path join operation is necessary.

REFERENCES

1. Florescu, D. and D. Kossman, 1999. Storing and querying XML data using an RDMBS. IEEE Data Eng. Bull., 22: 27-34.
2. World Wide Web Consortium. XML Path Language (XPath), W3C Recommendation, Version 1.0, Nov. 1999, http://www.w3.org/TR/x-path.

3. World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, Dec. 2001, http://www.w3.org/TR/x-query.

4. Dutch, A., M. Fernandez, D. Florescu, A. Levy and D. Suciu, 1999. A query language for XML. Proc. 8th World Wide Web Conf., pp: 77-91.

5. Chamberlin, D.D., J. Clark, D. Florescu and M. Stefanescu. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/query-data-model/.

6. Chamberlin, D.D., J. Robie and D. Florescu, 2000. Quilt: An XML query language for heterogeneous data sources. WebDB 2000.

7. Hoffmann, C.M. and M.J. O'Donnell, 1982. Pattern matching in trees. J. ACM, 29: 68-95.

8. Abiteboul, S., P. Buneman and D. Suciu, 1999. Data on the web: From relations to semistructured data and XML. Morgan Kaufmann Publisher, Los Altos, CA 94022.

9. Chung, C., J. Min and K. Shim, 2002. APEX: An adaptive path index for XML data. ACM SIGMOD.

10. Cooper, B.F., N. Sample, M. Franklin, A.B. Hialtason and M. Shadmon, 2001. A fast index for semistructured data. Proc. VLDB, pp: 341-350.

11. Goldman, R. and J. Widom, 1997. DataGuide: Enable query formulation and optimization in semistructured databases. Proc. VLDB, pp: 436-445.

12. McHugh, J. and J. Widom, 1999. Query optimization for XML. Proc. of VLDB.

13. Li, Q. and B. Moon, 2001. Indexing and querying XML data for regular path expressions. Proc. VLDB, pp: 361-370.

14. Wang, H. and X. Meng, 2005. On the sequencing of tree structures for XML indexing. Proc. Conf. Data Eng., Tokyo, Japan, pp: 372-385.

15. Zhang, C., J. Naughton, D. Dewitt, Q. Luo and G. Lohman, 2001. On supporting containment queries in relational database management systems. Proc. ACM SIGMOD.

16. Kaushik, R., P. Bohannon, J. Naughton and H. Korth, 2002. Covering indexes for branching path queries. ACM SIGMOD.

17. Schmidt, A.R., F. Waas, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey and R. Busse, 2001. The XML benchmark project. Technical Report INS-R0103, Centrum voor Wiskunde en Informatica.

18. Bruno, N., N. Koudas and D. Srivastava, 2002. Holistic twig joins: Optimal XML pattern matching. Proc. SIGMOD Intl. Conf. on Management of Data, Madison, Wisconsin, pp: 310-321.

19. Chen, T., J. Lu and T.W. Ling, 2005. On boosting holism in XML twig pattern matching. Proc. SIGMOD, pp: 455-466.

20. Choi, B., M. Mahoui and D. Wood, 2003. On the optimality of holistic algorithms for twig queries. Proc. DEXA, pp: 235-244.

21. Chen, S. et al., 2006. Twig 2Stack: Bottom-up processing of generalized-tree-pattern queries over XML Documents. Proc. VLDB, Seoul, Korea, pp: 283-323.

22. Lu, J., T.W. Ling, C.Y. Chan and T. Chan, 2005. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. Proc. VLDB, pp: 193-204.

23. Seo, C., S. Lee and H. Kim, 2003. An Efficient Index Technique for XML Documents Using RDBMS. Inform. Software Technol., 45: 11-22, Elsevier Science B.V.

24. Knuth, D.E., 1969. The Art of Computer Programming. Vol.1, Addison-Wesley, Reading.