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 [13], Xquery [29, 30], XML-QL [12], and Quilt [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.

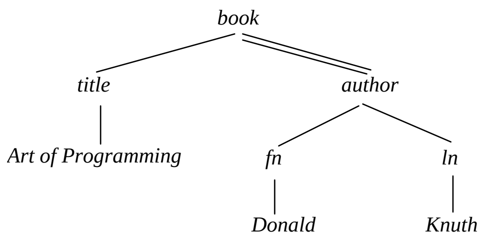

Figure 1. A query tree

In this tree structure, a node *v* is labeled with an element name or a string value, denoted *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 exists 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 [16].

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 [1, 9, 11, 14, 19, 22, 28, 29, 32, 33, 34], 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 [4, 7, 8, 10, 18, 20], 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 database, we need $O(p^{\lambda})$ time to join them together, where p is the largest length of a matching path and λ is the number of all such paths.

In this paper, we proposed a new algorithm with no join operations involved. The algorithm runs in $O(|T|\cdot Q_{leaf})$ time and $O(T_{leaf}\cdot Q_{leaf})$ space, where T_{leaf} and Q_{leaf} represent the numbers of the leaf nodes in T and in Q, respectively.

The remainder of the paper is organized as follows. In Section 2, we restate a kind of tree encoding [17], which facilitates the recognition of different relationships among the nodes of trees. In Section 3, we discuss an algorithm for simple cases that a twig pattern contains only *d*-edges. In Section 4, we extend this algorithm to general cases. Finally, a short conclusion is set forth in Section 5.

TREE ENCODING

To facilitate the checking of reachability (whether a node can be reached from another node through a path), a tree encoding is used [17].

Consider a tree T. By traversing T in *preorder*, each node v will obtain a number $pre(v)$ 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 $post(v)$. 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 $pre(v') > pre(v)$ and $post(v') < post(v)$. See Exercise 2.3.2-20 in [17].

As an example, have a look at the pairs associated with the nodes of the tree shown in Figure 2. The first element of each pair is the preorder number of the corresponding node and the second is its postorder number. Using such labels, the ancestor-descendant relationships can be easily checked.

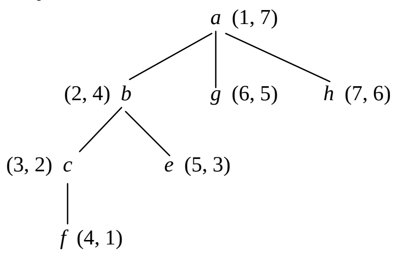

Figure 2. Labeling a tree

For instance, by checking the label associated with b against the label for f, we see that b is an ancestor of f in terms of Proposition 1. Note that b's label is (2, 4) and f's label is (4, 1), and we have $2 < 4$ and $4 > 1$. We also see that since the pairs associated with g and c do not satisfy the condition given in Proposition 1, g must not be an ancestor of c and *vice versa*.

Let (p, q) and (p', q') be two pairs associated with nodes u and v, respectively. We say that (p', q') is subsumed by (p, q), denoted (p', q') (p, q), if $p' > p$ and $q' < q$. Then, u is an ancestor of v if (p', q') is subsumed by (p, q).

In addition, if $p' < p$ and $q' < q$, u is to the left of v.

Finally, we can associate each node v with a level number $l(v)$ (the nesting depth of the element in a document). In conjunction with the tree encoding, this number can be utilized to tell whether a node is the parent of another node. For example, if $pre(v') > pre(v)$, $post(v') < post(v)$ and $l(v) = l(v') + 1$, then v' is a child node of v.

ALGORITHM FOR SIMPLE CASES

In this section, 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 q in Q such that $Q[q]$ can be imbedded into $T[v]$, where $T[v]$ represents a subtree of T rooted at v.
- Each q in Q is associated with a value $\delta(q)$, defined as follows.

Initially, for each $q \in Q$, $\delta(q)$ is set to ϕ. During the tree matching process, $\delta(q)$ is dynamically changed as below.

(i) Let v be a node in T with parent node u.

(ii) 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:

1. Check whether $label(q) = label(v)$.
2. Let $q_1, ..., q_k$ be the child nodes of q. For each q_i ($i = 1, ..., k$), check whether $\delta(q_i)$ is equal to v or to a descendent 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:

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 { }.

leaf-node-check(u) - It returns a set of leaf nodes in Q: $\{q_1, ..., q_k\}$ such that for each q_i ($1 \leq i \leq k$) $label(u) = label(q_i)$.

```
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.  {let $v_1, ..., v_k$ 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 parent};}
8.        remove all $\alpha(v_j)$ ($j = 1, ..., k$);
9.        for each q in S do
10.          $S_1 := S_1 \cup$ node-check(v, q);
11. }
12. $S_2$ := 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.  {let $q_1, ..., q_k$ 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.  {if type(q) = type(u) then {$S_2 := \{q\}$;
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 δ value is set to be v (see line 7). In addition, q's parent is inserted into S, a temporary valuable to be used in a 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, whether v matches q' by calling *node-check*( ), in which the δ 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 Figure. 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 $\alpha(v_5) = \{q_2\}$.

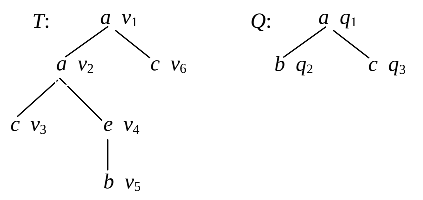

Figure 3. A document tree and a query tree

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 (see Fig. 4 for illustration.) After that, *node-check*(v_4, q_1) is invoked. (Note that q_1 is the parent of q_2. See lines 9 - 10.) Since $label(v_4) \neq label(q_1)$, it returns $S_1 = \emptyset$. *leaf-node-check*(v_4) also returns $S_2 = \emptyset$. So $\alpha(v_4) = \alpha(v_5) \cup S_1 \cup S_2 =$

$\{q_2\}$ (see line 13). When v_2 is met, we will first set $\delta(q_2) = \delta(q_3) = v_2$ (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 *label*(v_2) = *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 $\delta(q_3)$ are equal to v_2, we have that $T[v_2]$ contains $Q[q_1]$. Therefore, $S_1 = \{q_1\}$. Thus, we set $\alpha(v_2) = \alpha(v_3) \cup \alpha(v_4) \cup S_1 \cup S_2 = \alpha(v_3) \cup \alpha(v_4) \cup \{q_1\} \cup \varnothing = \{q_1, q_2, q_3\}$

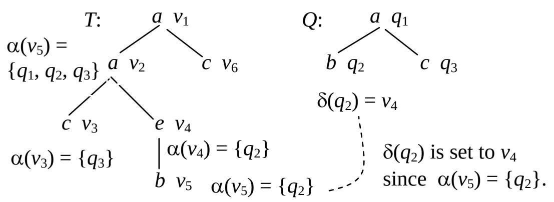

Figure 4. Sample trace

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_1 and set $\delta(q_1) = v_1$ and $\delta(q_3) = v_1$. Since *label*(v_1) = *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:

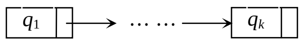

Figure 5. Linked list to store

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

Now we consider two α-lists α and α' sorted according to their nodes' preorder and postorder numbers. Define a merging operation over α and α', denoted *merge*(α, α'), as follows.

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

The result of *merge*(α, α') is stored in α, and α' remains unchanged. Especially, the changed α 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 α-lists together.

Algorithm *merge*(α, α')

Input: α and α' - sorted α-lists.

Output: modified α, containing all the nodes in α and α' with all the subsumed nodes removed.

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

```
    then {attach the rest of $\alpha$' to the end of $\alpha$;}
end
```

We can extend the merging operation over to more than two sorted α-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 *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.  {let $v_1, ..., v_k$ be the child nodes of v;
4.    for i = 1 to k do call tree-matching($v_i$);
5.    $\alpha$ := merge($\alpha(v_1), ..., \alpha(v_k)$);
6.    assume that $\alpha = \{q_1, ..., q_j\}$;
7.    for i = 1 to j do
8.    {$\delta(q_i) := v$;
9.      if ($q_i$'s parent $\neq q_{i-1}$'s parent) then
10.       $S := S \cup \{q_i$'s parent$\}$;}
11.   remove all $\alpha(v_j)$ (j = 1, ..., k);
12.   for each q in S do
13.       $S_1 := S_1 \cup$ node-check(v, q);
14. }
15. $S_2$ := leaf-node-check(v);
16. $\alpha(v)$ := merge($\alpha, S_1, S_2$);
end
```

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_{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 $\alpha(v_j)$'s (j = 1, ..., k), where v_j is a child node of the current node v. Since the nodes in α (α = merge($\alpha_1, ..., \alpha_{k-1}, \alpha_k$)) are left-to-right sorted (according to the nodes' preorder and postorder numbers), if there are more than one nodes in α sharing the same parent, they must appear consecutively in the list. So each time we insert a parent node q' (of some q in α) 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_{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, *height*(Q).

Basic step. When *height*(Q) = 1, the proposition trivially holds.

Induction step. Assume that the proposition holds for any query tree Q' with *height*(Q') $\leq h$. We consider a query tree Q of height h + 1. Let r_Q be the root of Q. Let $q_1, ..., q_k$ be the child nodes of r_Q. Then, we have *height*($Q[q_j]$) $\leq h$ (j = 1, ..., k). In terms of the induction hypothesis, for each q in $Q[q_j]$ (j = 1, ..., k), if it appears in $\alpha(v_i)$ (where v_i is a child node of v), we have $T[v_i]$ contains $Q[q]$ and $\delta(q)$ will be set to be v. Especially, if $T[v_i]$ contains $Q[q_j]$ (j = 1, ..., k), we have $q_j \in$ a(v_i) and $\delta(q_j)$ will be set to be v before v is checked against r_Q. Obviously, if *label*(v) = *label*(r_Q) and for each q_j (j = 1, ..., k), $\delta(q_j)$ is equal to v or a descendant of v, Q can be embedded into $T[v]$. So r_Q 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 $\alpha(v_1), ..., \alpha(v_k)$, where v_i (i = 1, ..., k) is a child node of some node v in T. This part of cost is bounded by

   $$O(\sum_{i}^{|T|} d_i Q_{leaf}) = O(|T|Q_{leaf}),$$

   where d_i represents the ourdegree of a node v_i in T.
2. The second part is the time used for generating S from a merged α-list. Since the size of the α-list is bounded by O(Q_{leaf}), so this part of cost is also bounded by O(Q_{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:

   $$O(\sum_{i}^{|T|}\sum_{j}^{|S_i|} c_j) = O(|T|Q_{leaf}),$$

   where c_j represents the ourdegree of a node q_j in S_i.

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

$$O(\sum_{i}^{|T|} Q_{leaf}) = O(|T|Q_{leaf}).$$

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

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

Proof. See the above discussion. □

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

GENERAL CASES

The algorithm discussed in Section 3 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_i of q that is being checked against v, if (q, q_i) is a *c*-edge, we will check whether $\delta(q_i)$ is equal to v. If (q, q_i) is a *d*-edge, we simply check whether $pre(\delta(q_i) \geq pre(v)$ and $post(\delta(q_i)) \leq post(v)$.

Accordingly, the algorithm *node-check* described in the previous section should be slightly modified.

```
Function general-node-check(u, q)
begin
1.  $S_1 := \varnothing$;
2.  if $label(q) = label(u)$ then
3.  {let $q_1, ..., q_g$ be the child nodes of $q$;
4.    flag := true; $i := 1$;
5.    while ($i \leq g \wedge$ flag) do
6.    {if  $\neg((( q, q_i)$ is a c-edge) $\wedge (\delta(q_i) = v)) \vee$
7.              $((q, q_i)$ is a d-edge) $\wedge$
8.              $(pre(\delta(q_i) \geq pre(u)) \wedge$
9.              $(post(\delta(q_i) \leq post(u))))$
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 d-edge) or
       (($q'$, $q$) is a c-edge and $q$ matches $v_i$))
    then {$\delta(q) := v$;
          let $q''$ be the last element in $S$;
          if ($q$'s parent $\neq q''$)
          then $S := S \cup \{q$'s parent$\}$;}
          else remove $q$ from $\alpha(v_i)$;
        }}
  $\alpha :=$ merge($\alpha(v_1), ..., \alpha(v_k)$);
```

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 δ 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 δ values are changed (to v). Assume that the current δ 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_j should be recorded so that the *c*-edge matching is not missed.

See Fig. 6 for illustration.

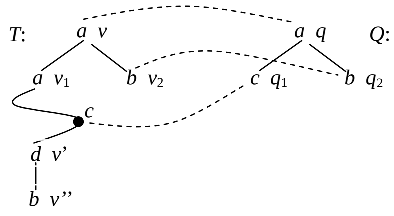

Figure 6. Illustration for *c*-edge checking

In Fig. 6, *v*'' is a descendant of *v* and matches q_2. So $\delta(q_2)$ will be set to *v*'. However, (q, 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*].

In conjunction with Proposition 1, the above analysis shows the correctness of the algorithm. We have the following proposition.

Proposition 3. Let *Q* be a twig pattern containing both *c*-edges and *d*-edges. Let *v* be a node in *T*. For each *q* in $\alpha(v)$ generated by *tree-matching*( ) with *general-node-check*( ), we have *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 paper, 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 $O(|T|\cdot Q_{leaf})$ time and $O(T_{leaf}\cdot Q_{leaf})$ space, where T_{leaf} and Q_{leaf} represent the numbers of the leaf nodes in *T* and in *Q*, respectively. More importantly, no costly join operation is necessary.

REFERENCES

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

2 A. Aghili, H. Li, D. Agrawal, and A.E. Abbadi, TWIX: Twig structure and content matching of selective queries using binary labeling, in: *INFOSCALE*, 2006.

3 S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y. Wu, Structural Joins: A primitive for efficient XML query pattern matching, in *Proc. of IEEE Int. Conf. on Data Engineering*, 2002.

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

5 D. D. Chamberlin, J.Clark, D. Florescu and M. Stefanescu. "XQuery1.0: An XML Query Language," http:/ /www.w3.org/TR/ query-datamodel/.

6 D. D. Chamberlin, J. Robie and D. Florescu. "Quilt: An XML Query Language for Heterogeneous Data Sources," *WebDB 2000*.

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

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

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

10 S. Chen et al., *Twig*^{2}*Stack*: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents, in *Proc. VLDB*, Seoul, Korea, Sept. 2006, pp. 283-323.

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

12 A. Dutch, M. Fernandez, D. Florescu, A. Levy, D.Suciu, A Query Language for

XML, in: *Proc. 8th World Wide Web Conf.*, May 1999, pp. 77-91.

13 D. Florescu and D. Kossman, Storing and Querying XML data using an RDMBS, *IEEE Data Engineering Bulletin*, 22(3):27-34, 1999.

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

15 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.

16 C.M. Hoffmann and M.J. O'Donnell, Pattern matching in trees, *J. ACM*, 29(1):68-95, 1982.

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

18 J. Lu, T.W. Ling, C.Y. Chan, and T. Chan, From Region Encoding to Extended Dewey: on Efficient Processing of XML Twig Pattern Matching, in: *Proc. VLDB*, pp. 193 - 204, 2005.

19 J. McHugh, J. Widom, Query optimization for XML, in *Proc. of VLDB*, 1999.

20 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.

21 G. Miklau and D. Suciu, Containment and Equivalence of a Fragment of XPath, *J. ACM*, 51(1):2-45, 2004.

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

23 J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. Dewitt, and J.F. Naughton, Relational databases for querying XML documents: Limitations and opportunities, in *Proc. of VLDB*, 1999.

24 U. of Washington, The Tukwila System, available from http://data.cs.washington.edu /integration/Tukwila/.

25 U. of Wisconsin, The Niagara System, available from http://www.cs.wisc.edu /niagara/.

26 U of Washington XML Repository, available from http://www.cs.washington. edu/research/xmldatasets.

27 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.

28 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.

29 World Wide Web Consortium. XML Path Language (XPath), W3C Recommendation, Version 1.0, November 1999. See http://www.w3.org/TR/xpath.

30 World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, Dec. 2001. See http://www.w3.org/TR/xquery.

31 XMARK: The XML-benchmark project, http://monetdb.cwi.nl/xml, 2002.

32 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*, 2001.

33 R. Kaushik, P. Bohannon, J. Naughton, and H. Korth, Covering indexes for branching path queries, in: *ACM SIGMOD*, June 2002.

34 A.R. Schmidt, F. Waas, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey, and R. Busse, The XML benchmark project, Technical Report INS-Ro1o3, Centrum voor Wiskunde en Informatica, 2001.