STACK ENCODING REVISITED

Yangjun Chen
Dept. Applied Computer Science, University of Winnipeg, Canada

Keywords: XML databases, Trees, Paths, XML pattern matching, Twig joins.

Abstract: The twig join, which is used to find all occurrences of a twig pattern in an XML database, is a core operation for XML query processing. A great many strategies for handling this problem have been proposed and can be roughly classified into two groups. The first group decomposes a twig pattern (a small tree) into a set of binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations; and transforms a tree matching problem into a series of simple relation look-ups. The second group decomposes a twig pattern into a set of paths. Among all this kind of methods, the approach based on the so-called *stack encoding* [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] is very interesting, which can represent in linear space a potentially exponential (in the number of query nodes) number of matching paths. However, the available processes for generating such compressed paths suffer some redundancy and can be significantly improved. In this paper, we analyze this method and show that the time complexities of path generation in its two main procedures: *TwigStack* and *TwigStackXB* can be reduced from $O(m^2 \cdot n)$ to $O(m \cdot n)$, where m and n are the sizes of the query tree and document tree, respectively. Experiments have been done to compare *TwigStackXB* and ours, which shows that using our method much less time is needed to generate matching paths.

1 INTRODUCTION

In XML (World Wide Web Consortium, 1991, 2001), data is represented as a tree; associated with each node of the tree is an element type from a finite alphabet ∑. The children of a node are ordered from left to right, and represent the content (i.e., list of subelements) of that element.

To abstract from existing query languages for XML, e.g. XPath (World Wide Web Consortium, 1991), XQuery (World Wide Web Consortium, 2001), XML-QL (Deutch and *et al*, 1999), and Quilt (Chamberlin and *et al*, 1999; Chamberlin and *et al*, 2000), we express queries as tree patterns where nodes are types from $\sum \cup \{*\}$ (* is a wildcard, matching any node type) and string values, and edges are *parent-child* or *ancestor-descendant* relationships. As an example, consider the query tree shown in Figure 1, which asks for any node of type *b* (node 2) that is a child of some node of type *a* (node 1). In addition, the *b* type (node 2) is the parent of some *c* type (node 4) and an ancestor of some *d* type (node 5). Type *b* (node 3) can also be the parent of some *e* type (node 7). The query corresponds to the following XPath expression:

a[*b*[*c* and //*d*]]/*b*[*c* and *e*//*d*].

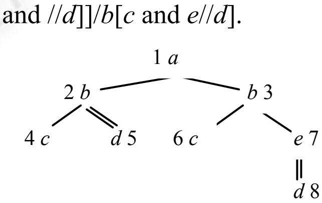

Figure 1: A query tree.

In Figure 1, 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*.

Finding all occurrences of a twig pattern in a database has been considered as a core operation in querying tree structured XML data, both in relational implementation of XML databases, and in native XML databases.

Recently this problem has received much attention in database research community and different strategies have been proposed. Most of

Chen Y. (2007).
STACK ENCODING REVISITED.
In *Proceedings of the Third International Conference on Web Information Systems and Technologies - Internet Technology*, pages 5-14
DOI: 10.5220/0001260800050014

them, for example, all the strategies proposed in (Al-Khalifa and *et al*, 2002; Florescu, Kossman, 1999; McHugh, Widom, 1999; Shanmugasundaram and *et al*, 1999; Tukwila System, 2000; Niagara System, 2000; Zhang and *et al*, 2001), typically decompose a twig pattern into a set of binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations; and the sizes of intermediate relations tend to be very large, even when the input and final result sizes are much more manageable. Another kind of strategies bases on path decomposition, such as those discussed in (Bruno and *et al*, 2002; Wang and *et al*, 2003; Wang and Meng, 2005). In (Wang and *et al*, 2003; Wang and Meng, 2005), all the possible paths of an XML document are explicitly stored and indexed using B+-trees as well as trie structures. In (Bruno and *et al*, 2002), a document is also decomposed, but dynamically depending on the given queries. This method is of special interest since the decomposed paths are not simply stored but compressed by using the so-called *stack encoding*. It reduces the number of intermediate matching paths dramatically. Although the idea of compressing intermediate results is very attractive, the process suggested in (Bruno and *et al*, 2002) for producing compact paths is not so efficient and can be substantially improved.

In this paper, we analyze the method described in (Bruno and *et al*, 2002) and show that the matching paths can be produced in a more efficient way. Particularly, two new algorithms are presented, which improve the two main procedures of this method: *TwigStack* and *TwigStackXB*, by one order of magnitude. In (Bruno and *et al*, 2002), *TwigStack* is utilized to generate matching paths for queries containing only *d*-edges while *TwigStackXB* is for queries containing both *c*- and *d*-edges.

The remainder of the paper is organized as follows. In Section 2, we review the concept of stack encoding and the algorithm *TwigStack* presented in (Bruno and *et al*, 2002), which is necessary for the subsequent discussion. In Section 3, we propose a new algorithm *RefinedTwigStack* to do the same task as *TwigStack*, but using much less time. In Section 4, we extend *RefinedTwigStack* to general cases. Finally, a short conclusion is set forth in Section 5.

2 ON THE TWIGSTACK ALGORITHM

In this section, we review the main procedure *TwigStack* given in (Bruno and *et al*, 2002), which is used to evaluate a special kind of queries that contain only *d*-edges. However, by using a variant structure of B-tree, called *XB-tree*, *TwigStack* can be easily extended to general cases with both *c*-edges and *d*-edges involved.

In the following, we first review what is a stack encoding in 2.1. Then, we describe the *TwigStack* algorithm (Bruno and *et al*, 2002) and analyze its time complexity in 2.2. In (Bruno and *et al*, 2002), a theoretical time analysis is not delivered.

2.1 On the Stack Encoding

Let T be a document tree. Let $q = q_1 \Rightarrow q_2 \ldots \Rightarrow q_{m-1} \Rightarrow q_m$ be a query path containing only *d*-edges. We associate each q_i (i = 1, ..., m) with a stack, denoted $S(q_i)$, in which each entry is a pair (v, p) with v being a node in T and p is a pointer to an entry in $S(parent(q_i))$, where $parent(q_i)$ represents the parent of q_i.

At every point during the computation, all $S(q_i)$'s have the following properties

(i) The entries in $S(q_i)$ (from bottom to top) are guaranteed to lie on a root-to-leaf path in T.

(ii) The set of stacks contains a compact encoding of matching paths.

As an example, consider T and q shown in Figure 2(a).

Obviously, T has four subpaths that match q, as shown in Figure 2(b). By using the stack encoding, they can be stored in a way as shown in Figure 2(c), using much less space.

First, we notice that the matching path $v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6$ is encoded since v_6 points to v_5, v_5 to v_4, and v_4 to v_3. Also, the matching path $v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6$ is encoded since v_1 is below v_3 on the stack $S(q_1)$. For the same reason, $v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_6$ is a matching path since v_2 is below v_4 on the stack $S(q_2)$ and has a pointer to v_1. Finally, since v_3 is below v_5 on the stack $S(q_3)$ and has a pointer to v_2, $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_6$ is also an answer. However, the nodes v_3, v_2, v_5, v_6 do not make up a matching path since v_3 is above v_1 on $S(q_1)$, to which v_2 points.

2.2 Description of *TwigStack*

Now we describe the algorithm *TwigStack*, which stores the intermediate results in a way of stack encoding, and analyze its time complexity. For this purpose, we first show a tree encoding method (Zhang and *et al*, 2001) and define some notations that are used in the description of *TwigStack*.

Let T be a document tree. We associate each node v in T with a quadruple (*DocId*, *LeftPos*, *RightPos*, *LevelNum*), denoted as $\alpha(v)$, where DocId is the document identifier; LeftPos and RightPos are generated by counting word numbers from the beginning of the document until the start and end of the element, respectively; and LevelNum is the nesting depth of the element in the document. (See Figure 3 for illustration.) By using such a data structure, the structural relationship between the nodes in an XML database can be simply determined (Zhang and *et al*, 2001):

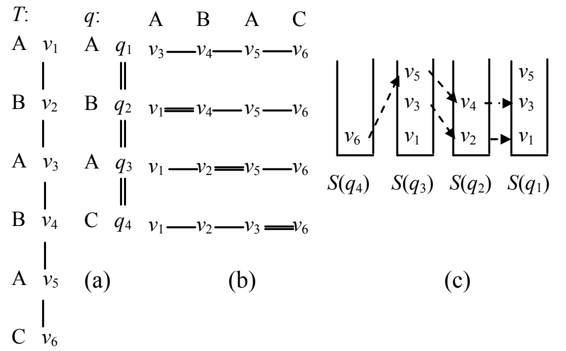

Figure 2: Illustration for stack encoding

(i) *ancestor-descendant*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is an ancestor of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$.

(ii) *parent-child*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is the parent of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, $r_1 > r_2$, and $ln_1 = ln_2 + 1$.

(iii) *from left to right*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is to the left of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $r_1 < l_2$.

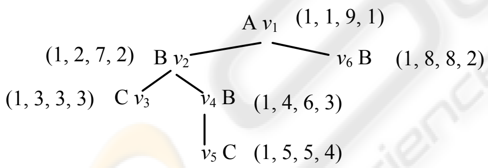

Figure 3: Illustration for tree encoding.

Let q be a query tree containing only *d*-edges. We associate each q_i in q with a data stream $L(q_i)$, which contains the quadruples of the database nodes that match q_i as illustrated in Figure 4. Such a list can be established by using an efficient access mechanism, such as an index structure. In addition, the quadruples in a list are sorted by their (DocId, LeftPos) values.

Finally, we notice that in both $S(q_i)$ and $L(q_i)$, a node v is referenced by $\alpha(v)$. But we will refer to v and $\alpha(v)$ interchangeably in the subsequent discussion if no confusion will be caused.

In terms of the data structure described above, we can now specify some operations that are used in *TwigStack*.

- *next*($L(q_i)$): return the next element in $L(q_i)$. Initially, the pointer is to the position before the first element in $L(q_i)$.
- *advance*($L(q_i)$): move to the next element in $L(q_i)$;
- *LeftPos*(α): return the LeftPost value of α;
- *RightPos*(α): return the RightPost value of α.

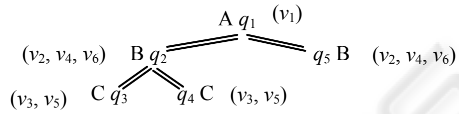
Figure 4: Illustration for for $L(q_i)$'s.

Algorithm *TwigStack* operates in two phases. In the first phase, all paths matching individual query root-to-leaf paths are produced (lines 1 - 14). In the second phase, these matching paths are merge-joined to create the answers to the query twig pattern (line 15).

In order to generate all the matching paths, the query tree q is accessed repeatedly and each time a node q_i, which has in its $L(q_{i_j})$ a node v with the least LeftPos value among all the nodes in all $L(q_j)$'s, is chosen, satisfying the following conditions:

(i) Let q_{i_1}, ..., q_{i_k} be the children of q_i. Let v be the next node in $L(q_i)$ to be handled. Then, for each q_{i_j} $(1 \leq j \leq k)$, v has a descendant u such that $\alpha(u)$ is in $L(q_{i_j})$.

(ii) Each q_{i_j} recursively satisfies the first property.

Such a node is selected by executing a function, called *getNext*(q), which is repeatedly invoked. In this way, each solution to each individual query root-to-leaf path is guaranteed to be merge-joinable with at least one solution to each of other root-to-leaf paths.

Once such a node, denoted q_{act}, is found, the quadruple $\alpha = next(L(q_{act}))$ (which represents a node v in T) will be pushed onto $S(q_{act})$ as follows:

1. If q_{act} is the root of q, remove any $\alpha(u)$ in $S(q_{act})$ with $RightPos(\alpha(u)) < LeftPos(\alpha(q_{act}))$. Then, put $next(L(q_{act}))$ on the top of $S(q_{act})$.

2. If q_{act} is not the root of q, remove any $\alpha(u)$ in $S(parent(q_{act}))$ with $RightPos(\alpha(u)) < LeftPos(next(L(q_{act})))$. If $S(parent(q_{act}))$ remains unempty, put $next(L(q_{act}))$ on the top of $S(q_{act})$ after all the v in $S(q_{act})$ with $RightPos(\alpha(v)) < LeftPos(\alpha(q_{act}))$ are removed.

If q_{act} is a leaf node, store the corresponding matching paths.

For the purpose of self-contentment, we show here the algorithm *TwigStack* in a format slightly different from (Bruno and *et al*, 2002). Then, we conduct a sample trace and analyze its time complexity. (In the original paper (Bruno and *et al*, 2002), the time complexity analysis was not available.)

Algorithm *TwigStack*(q)

(*phase 1*)

1. **while** $\neg$ *end*(q) **do**
2. { $q_{act} \leftarrow$ *getNext*(q);
3. **if** (q_{act} is not the root) **then**
4. *cleanStack*(*S*(*parent*(q_{act})), *LeftPos*(*next*(*L*(q_{act})));
5. **if** (q_{act} is the root of q) $\vee \neg$ empty(*S*(*parent*(q_{act})))
6. **then**
7. { *cleanStack*(*S*(q_{act}), *LeftPos*(*next*(*L*(q_{act})));
8. *moveStreamToStack*(*L*(q_{act}), *S*(q_{act}), pointer to top(*S*(q_{act})));
9. **if** (q_{act} is a leaf node) **then**
10. {output all the matching paths (stored in stacks) in the compact form;
11. pop(*S*(q_{act}));}
12. }
13. **else** *advance*(*L*(q_{act}));
14. }

(*phase 2*)

15. *mergeAllPathSolutions*();

Function *getNext*(q)

1. **if** (q is a leaf node) **then** return q;
2. let q_1, ..., q_k be the children of q;
3. **for** i = 1 to k **do**
4. {$n_i \leftarrow$ *getNext*(q_i);
5. **if** ($n_i \neq q_i$) **then** return n_i;}
6. $n_{min} \leftarrow min\{LeftPos(n_1), ..., LeftPos(n_k)\}$;
7. $n_{max} \leftarrow max\{LeftPos(n_1), ..., LeftPos(n_k)\}$;
8. **while** (*RightPost*(*next*(*L*(q)) < *LeftPost*(*next*(*L*(n_{max})) **do**
9. *advance*(*L*(q));
10. **if** (*LeftPost*(*next*(*L*(q)) < *LeftPost*(*next*(*L*(n_{min}))
11. **then** return q;
12. **else** return n_{min};

Function *end*(q)

1. **if** for any leaf node q_{leaf}, *L*(q_{leaf}) is empty
2. **then** return *true*
3. **else** return *false*;

Procedure *cleanStack*(*S*, *actL*)

1. **while** ($\neg$ empty(*S*) $\wedge$ (*RightPos*(top(*S*) < *actL*) **do**
2. **pop**(*S*);

Procedure *moveStreamToStack*(*L*, *S*, *p*)

1. push(*S*, *next*(*L*), *p*);
2. *advance*(*L*);

By each iteration of the main **while**-loop of *TwigStack*(q), *getNext*(q) is called to find a node q_{act} to handle (see line 2). Then, by executing lines 3 - 8, *next*(*L*(q_{act})) is pushed onto *S*(q_{act}) in the way as described by (1) and (2) above. If q_{act} is a leaf node, all the matching paths (in their stack encoding) will be stored in the compact form (see line 9 - 11). In addition, no matter whether *next*(*L*(q_{act})) can be put onto *S*(q_{act}), the pointer for *L*(q_{act}) will be shifted to the next element (see line 2 in *moveStreamToStack* and line 13 *TwigStack*).

getNext(q) is a recursive algorithm, by which the whole q is searched top-down. In this way, any node returned has always the least preorder number with the conditions (i) and (ii) above satisfied. This can be seen from lines 8 - 9, as well as lines 10 - 12.

Finally, we notice that the algorithm terminates when all *L*(q_{leaf})'s become empty (see Function *end*(q)).

The following example helps for illustration. It is a detailed sample trace, which not only facilitates the analysis of the algorithm's time complexity, but also reveals a possibility of improvements.

Example 1. Consider the document tree T shown in Figure 3 and the query tree q shown in Figure 4. Corresponding to the three leaf nodes in q, we have three paths: P_1: $q_3 \rightarrow q_2 \rightarrow q_1$; P_2: $q_4 \rightarrow q_2 \rightarrow q_1$; and P_3: $q_5 \rightarrow q_1$. When we apply *TwigStack* to T and q, the stacks associated with the nodes in q will be changed as follows.

Step 1 - 3: By the first iteration of the main **while**-loop, q_1 is selected and then v_1 is pushed onto $S(q_1)$. By the second iteration, q_2 will be chosen since after the first iteration $L(q_1)$ becomes empty and so we cannot find a v in $L(q_1)$, which is an ancestor of $next(L(q_2))$ = v_2. (See line 12 in *getNext*.) Therefore, v_2 goes into $S(q_2)$. For the same reason, q_3 will be chosen by the third iteration and $next(L(q_3)) = v_3$ goes into $S(q_3)$. Since q_3 is a leaf node, we get the first matching path (for P_1): $v_3 \rightarrow v_2 \rightarrow v_1$. See Figure 5(a) for illustration.

Step 4: By the fourth iteration, q_4 is selected and $next(L(q_4)) = v_3$ is pushed onto $S(q_4)$) as shown in Figure 5(b). We get the second matching path (for P_2): $v_3 \rightarrow v_2 \rightarrow v_1$.

Step 5: By the fifth iteration, q_2 is chosen again. Then, $next(L(q_2)) = v_4$ is put on the top of $S(q_2)$ as shown in Figure 5(c). Remember that after each iteration, the pointer for the corresponding $L(q_{act})$ is shifted to the next element.

Step 6: By the sixth iteration, q_3 is selected once again and $next(L(q_3)) = v_5$ will be put onto $S(q_3)$. But before that, v_3 is popped out since $RightPos(v_3) < LeftPos(v_3)$ (see line 7 in *TwigStack*.) The stacks are changed as shown in Figure 5(d), which shows another two paths matching P_1: $v_5 \rightarrow v_4 \rightarrow v_1$ and $v_5 \rightarrow v_2 \rightarrow v_1$. They are represented in the compact form.

Step 7: By the seventh iteration, q_4 is selected for the second time and $next(L(q_4)) = v_5$ is pushed onto $S(q_4)$. Before this operation, v_3 is first popped out. The new stacks are shown in Figure 5(e), from which we will get two new matching paths (for P_2): $v_5 \rightarrow v_4 \rightarrow v_1$ and $v_5 \rightarrow v_2 \rightarrow v_1$.

Step 8: By the eighth iteration, q_5 is chosen and $next(L(q_5)) = v_2$ is put on the top of $S(q_5)$ as shown in Figure 9(f). It shows the first matching path for P_3: $v_2 \rightarrow v_1$.

Step 9: By the ninth iteration, q_5 is chosen once again and $next(L(q_5)) = v_4$ is put on the top of $S(q_5)$ as shown in Figure 5(g). It shows the second matching path for P_3: $v_4 \rightarrow v_1$.

Step 10: By the ninth iteration, q_5 is chosen for the third time and $next(L(q_5)) = v_6$ is put on the top of $S(q_5)$ as shown in Figure 5(h). It shows the second matching path for P_3: $v_6 \rightarrow v_1$. We notice that in this step, q_2 will not be selected although $L(q_5) = \{v_6\}$ is not empty. It is because both $L(q_3)$ and $L(q_4)$ are empty and the condition (i) in the previous section cannot be satisfied.

The time complexity of the algorithm can be analyzed as follows.

Let n_i be the size of $L(q_i)$. Then, the main **while**-loop in *TwigStack* will be iterated $\sum_{i=1}^{m} n_i$ times since the termination condition of this **while**-loop is when all the elements in all $L(q_{leaf})$'s are exhausted. In each iteration, the procedure *getNext* will be invoked and all the nodes in the query tree will be accessed. Let λ_{ijk} be the number of elements in $L(q_k)$ checked when node q_k is visited during the (i, j)-th execution of *getNext*. Then, the worst-case cost is bounded by

$$O(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}(1+\lambda_{ijk}))$$

$$= O(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}1) + O(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}\lambda_{ijk})$$

$$= O(m^2 \cdot n) + O(m \cdot n) = O(m^2 \cdot n).$$

Here we should remark that $O(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}\lambda_{ijk})$ cannot be larger than $m \cdot n$ since at most $m \cdot n$ elements may be pushed on to the stacks.

Applying the above method to another algorithm *TwigStackXB* in (Bruno and *et al*, 2002), which is an extension of *TwigStack* for general cases, we get the same time complexity.

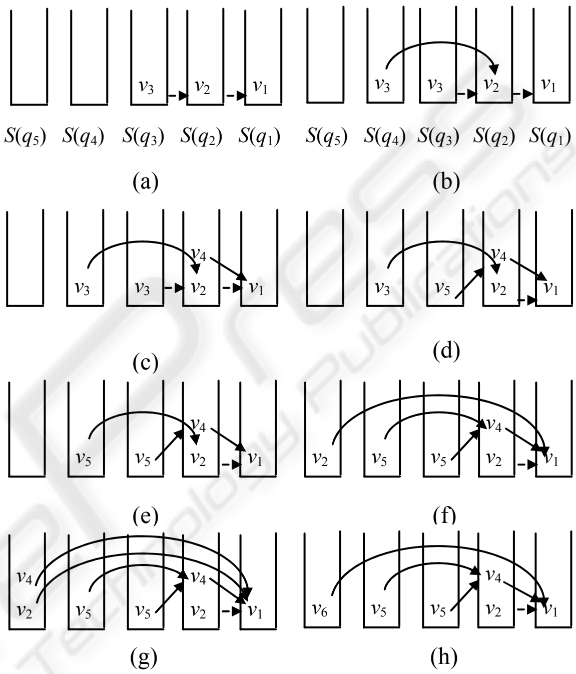

Figure 5: Sample trace.

3 REMOVING REDUNDANCY FORM TWIGSTACK

Now we begin to discuss how *TwigStack* can be improved. As with *TwigStack*, we will associate each node q_i in q with a data stream $L(q_i)$, but with the following conditions satisfied:

(i) For each $v \in L(q_i)$, v matches the predicate at q_i.

(ii) Let q_{i_1}, ..., q_{i_k} be the children of q_i. v has a descendant v' matching q_{i_j} for $j \in \{1, ..., k\}$.

(iii) Each q_{i_j} recursively satisfies (ii).

Obviously, these three conditions correspond to the two properties (i) and (ii) given in the previous section, for any node going onto a stack. Nothing is new. However, not like *getNext* in *TwigStack*, which chooses nodes from q to handle and each time finds a next v in T to be put in some stack (by multiple executions), we generate all $L(q_i)$'s in one scan, which enables us to avoid a great number of repeated accesses to query nodes.

In the following, we will use *T'* to represent a subtree of *T*, which contains only those nodes matching some node in *q*.

We will maintain two $m \times n$ ($m = |q|$, $n = |T'|$) matrices, defined as below.

1. The nodes in both *q* and *T'* are numbered in postorder, and the nodes *v* are then referred to by their postorder numbers, denoted as *post*(*v*).
2. In the first matrix, each entry c_{ij} ($i \in \{1, ..., m\}$, $j \in \{1, ..., n\}$) has value 0 or 1. If $c_{ij} = 1$, it indicates that $i \in L(j)$ and for each child of *i*, *j* has a descendant satisfying the predicate at it. Otherwise, $c_{ij} = 0$. This matrix is denoted by $c(q, T')$.
3. In the second matrix, each entry d_{ij} ($i \in \{1, ..., m\}, j \in \{1, ..., n\}$) is defined as follows. If *j* has a descendant *j'* such that $c_{ij'} = 1$, then $d_{ij} = 1$; otherwise $d_{ij} = 0$. This matrix is denoted by $d(q, T')$. In addition, a node itself is considered to be one of its ancestors.

These two matrices can be established by using an algorithm called *matrixGeneration*(*T'*, *q*), presented below.

Initially, $c_{ij} = 0$ and $d_{ij} = 0$ for all *i* and *j*. During the execution of the algorithm, the values of c_{ij}'s will be changed according to (2) and (3) described above; and d_{ij}'s will be changed to record whether a node *j* in *T'* has a descendant *j'* that matches a certain node *i* in *q*.

Algorithm *matrixGeneration*(*T'*, *q*)
Input: tree *T'* (with nodes 1, ..., *n*) and tree *q* (with nodes 1, ..., *m*)
Output: $c(q, T')$ with values created.
begin
1. **for** *u* := 1, ..., *m* **do** {
2. **for** *v* := 1, ..., *n* **do**
3. {**if** *v* satisfies the predicate at *u* **then**
4. let $u_1, ..., u_k$ be the children of *u*;
5. **if** $d_{u_1v} \wedge ... d_{u_kv} \wedge = 1$ **then** $c_{uv} \leftarrow 1$;
6. }
7. let $v_1, v_2, ..., v_h$ be the nodes such that $c_{uv_p} = 1$ ($1 \le p \le h$);
8. let $\{w_1, ..., w_r\}$ be a set such that each node in it is an ancestor of some v_p ($1 \le p \le h$). Set $d_{uw_l} = 1$ for each w_l ($1 \le l \le r$).
9. }
end

To see how the above algorithm works, we should first notice that both *T'* and *q* are both postorder-numbered. Therefore, the algorithm proceeds in a bottom-up way (see line 1 and 2). For any node *u* in *q* and any node *v* in *T'*, if *v* satisfies the predicate at *u*, we will check each child u_i of *u* to see whether there exists a descendant of *v* that matches u_i (see line 5). If it is the case, c_{uv} will be set to 1.

In line 7 and 8, we change d_{ij}'s according to the newly changed c_{ij}'s.

Example 2. As an example, consider the trees *T* and *q* shown in Figure 3 and 4 once again. Since each node in *T* matches a node in *q*, we have *T'* = *T*. In addition, the nodes of *T* and *q* are postorder numbered as shown in Figure 6(a) and (b), respectively.

When we apply the above algorithm to these two trees, $c(q, T)$ and $d(q, T)$ will be created and changed in the way as illustrated in Figure 7, in which each step corresponds to an execution of the outmost **for**-loop.

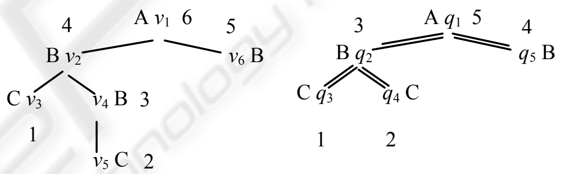

Figure 6: Postorder numbering.

In step 1, we show the values in $c(q, T)$ and $d(q, T)$ after node 1 in *q* is checked against every node in *T*. Since node 1 in *q* matches node 1 and 2 in *T*, c_{11} and c_{12} are all set to 1. Meanwhile, for all those nodes that are an ancestor of 1 or 2 in *T*, the corresponding entries in $d(q, T)$ will be changed. So we have all d_{11}, d_{12}, d_{13}, d_{14}, and d_{16} set to 1 (see line 7 and 8).

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

In step 3, node 3 in *q* will be checked against every node in *T*, but matches only node 4 in *T*. Since it is an internal node, its children will be further checked. For this purpose, we will check both d_{14} and d_{24} since node 3 in *q* has two child nodes postorder-numbered with 1 and 2, respectively. Since $d_{14} = d_{24} = 1$, we set c_{34} to 1. Accordingly, d_{34} and d_{36} are also set to 1.

In step 4, we will set c_{43}, c_{44} and c_{45} to 1 since node 4 in *q* is just a leaf node and matches node 3, 4, and 5 in *T*. So d_{43}, d_{44}, d_{45}, and d_{46} will be accordingly set to 1.

In step 5, since node 5 in *q* matches node 6 in *T*, and both d_{36} and d_{46} are equal to 1 (we remark that node 3 and 4 in q are the child nodes of node 5), we set c_{56} and then d_{56} to 1.

step 1: $c(q, T)$: $d(q, T)$:

$$c(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \qquad d(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

step 2: $c(q, T)$: $d(q, T)$:

$$c(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \qquad d(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 1 & 1 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

step 3: $c(q, T)$: $d(q, T)$:

$$c(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \qquad d(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 1 & 1 & 0 & 1 \\ 3 & 0 & 0 & 1 & 1 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

step 4: $c(q, T)$: $d(q, T)$:

$$c(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \qquad d(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 1 & 1 & 0 & 1 \\ 3 & 0 & 0 & 1 & 1 & 0 & 1 \\ 4 & 0 & 0 & 1 & 1 & 1 & 1 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

step 5: $c(q, T)$: $d(q, T)$:

$$c(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \qquad d(q,T) = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 1 & 1 & 0 & 1 \\ 3 & 0 & 0 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 & 1 & 1 \\ 5 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Figure 7: Sample trace.

In the following discussion, we use T instead of T' to simplify the notation. The reader should notice that it refers to the subtree of T containing only those nodes that match some node in q.

Proposition 1. Algorithm *matrixGeneration*(T, q) computes the values in $c(q, T)$ and $d(q, T)$ correctly.

Proof. We prove the proposition by induction on the sum of the heights of T and q, h. Without loss of generality, assume that height(T) ≥ 1 and height(q) ≥ 1.

Basic step. When $h = 2$, the proposition trivially holds.

Induction hypothesis. Assume that when $h = l$, the proposition holds.

Consider $T = \langle t; T_1, ..., T_k\rangle$ and $q = \langle p; P_1, ..., P_g\rangle$ with height(T) + height(q) = $l + 1$, where t (p) is the root of T (q, resp.) and $T_1, ..., T_k$ ($P_1, ..., P_g$) are the subtrees of t (p, resp.). Obviously, we have height(T_i) + height(q) $\leq l$ and height(T) + height(P_j) $\leq l$. Therefore, in terms of the induction hypothesis, the algorithm correctly computes the values in $c(q, T_i)$ and $d(q, T_i)$, as well as the values in $c(P_j, T)$ and $d(P_j, T)$ ($i = 1, ..., k$; $j = 1, ..., g$). Assume that 1, ..., m are the postorder numbers of the nodes in q, and 1, ..., n are the postorder numbers of the nodes in T. Then, the values for c_{ij} ($i = 1, ..., m - 1$; $j = 1, ..., n - 1$) and d_{ij} ($i = 1, ..., m - 1$; $j = 0, ..., n - 2$) are all correctly generated. Now we will check c_{in} and $d_{i(n-1)}$ ($i = 1, ..., m$), as well as c_{mj} ($j = 1, ..., n$) and d_{mj} ($j = 0, ..., n - 1$) to see whether they can be correctly produced. Let $i_1, ..., i_s$ be the children of i. If i matches n, for each i_f ($1 \leq f \leq s$), $d_{i_f n}$'s will be checked. If we have $d_{i_1 n} \wedge ... \wedge d_{i_s n} = 1$, we set c_{in} to 1; otherwise 0 (see line 5). According to the induction hypothesis, all such $d_{i_f n}$'s are correctly generated. Therefore, c_{in} ($i = 1, ..., m$) is correctly created, so is $d_{i(n-1)}$ ($i = 1, ..., m$). A similar analysis applies to c_{mj} ($j = 1, ..., n$) and d_{mj} ($j = 0, ..., n - 1$).

Proposition 2. Algorithm *matrixGeneration*(T, q) requires $O(n \cdot m)$ time and space, where $n = |T|$ and $m = |q|$.

Proof. During the whole process, against each node u in q, all the nodes v in T is checked and for each v all its children will be examined. Therefore, this part of time is bounded by

$$O(\sum_{u=1}^{m} \sum_{v=1}^{n} d_v) = O(\sum_{u=1}^{m} n) = O(n \cdot m),$$

where d_v represents the outdegree of node v in T.

In addition, after each u in q is checked, for all those nodes in T, which are an ancestor of some node that matches u, the corresponding matrix entries in $d(q, T)$ will be established. But this operation needs only $O(n)$ time if we proceeds as follows. Each time we search T bottom-up from a node v that matches u to find all its ancestors, we mark each node encountered and stop whenever we meet such a mark (made by a previous searching). So at most $O(n)$ nodes will be checked and the total time of this part of operations is bounded by $O(n \cdot m)$.

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

In terms of the matrix $c(q, T)$, it is an easy task to create $L(q_i)$ for each q_i in q as illustrated in Figure 8(a).

Figure 8(b) is the same as Figure 8(a). But in this figure we use node names in $L(q_i)$ instead of their postorder numbers. We will use the node names in the subsequent discussion to avoid any confusion.

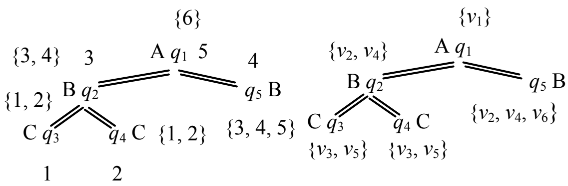

Figure 8: Illustration for $L(q_i)$'s.

Concerning $L(q_i)$, we should pay attention to the following:

(1) The nodes (represented by their quadruple) in $L(q_i)$ are sorted by their (DocId, LeftPos) values (not according to their postorder numbers).

(2) Each node in $L(q_i)$ satisfies the condition (i) and (ii) given in 2.2.

Using such a data structure, the algorithm *TwigStack* can be substantially improved. The main idea is a depth-first searching of q. To this end, we use a stack to control the process. Each entry in the stack is a pair (q_i, v_j), where $q_i \in q$ and $v_j \in T$.

Finally, we notice that *getNext*( ) will not be used since all the values to be produced by executing *getNext*( ) are pre-calculated and incorporated into $L(q_i)$'s. In addition, each node q_i in q is associated with its preorder number, denoted as $pre(q_i)$, which will be used in the following algorithm. In Figure 9, we show the preorder numbering of q.

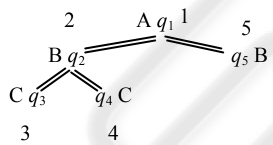

Figure 9: Preorder numbering.

Algorithm *RefinedTwigStack*(q)
(*phase 1*)
1.Repeat the following until all $L(q_i)$ become empty;
2. {let $pre(q_i)$ be the least such that $L(q_i)$ is not empty;
3. push(*stack*, (q_i, *next*($L(q_i)$)));
4. **while** ¬ empty(*stack*) **do**
{(u, v) ← pop(*stack*);
6. **if** (u is not the root) **then**
7. *cleanStack*(S(*parent*(u)), LeftPos(v));
8. **if** (u is the root of q) ∨ ¬ empty(S(*parent*(u)))
9. **then**
10. {*cleanStack*($S(u)$, LeftPos(v));
11. push($S(u)$, v, pointer to top(S(*parent*(u)));
advance($L(u)$);
12. **if** (u is a leaf node) **then**
13. { output all the matching paths (stored in stacks) in the compact form; pop($S(u)$);}
14. }
15. **else** *advance*($L(u)$);
16. let q_1, ..., q_l be the children of u;
17. **for** $j = l$ to 1 **do**
18. {while *next*($L(q_j)$) is not a descendant of v **do**
adavance($L(q_j)$);
19. push(stack, (q_j, *next*($L(q_j)$)));}
20.}}
(*phase 2*)
21. *mergeAllPathSolutions*();

Example 3. Continue with Example 2.

By using our method, we will first generate $L(q_i)$ for each q_i as shown in Figure 8(b). Then, we will search the twig pattern q as follows.

Step 0: At the very beginning, the node q_1 has the least LeftPos value and $L(q_1)$ is not empty. Push (q_1, v_1) into *stack*.

Step 1 - 3: In the following **while**-loop, the whole query tree will be traversed in depth-first fashion.
When we meet q_3, the stacks will be changed as shown in Figure 10(a). Since q_3 is a leaf node, we get the first matching path (for P_1): $v_3 \rightarrow v_2 \rightarrow v_1$.

Step 4: When we meet q_4, another leaf node, the stacks will be changed as shown in Figure 10(b). We get the second matching path (for P_2): $v_3 \rightarrow v_2 \rightarrow v_1$.

Step 5: q_5 is visited. The stacks are changed as shown in Figure 10(c). Since q_4 is a leaf node, we get the third matching path (for P_3): $v_2 \rightarrow v_1$.

Step 6: Now *stack* (used to control the searching of q) is empty. We will try to find another node (in q) with the least LeftPos value and a non-empty list. It is q_2. In $L(q_2)$, we have one element left: {v_4}. Push (q_2, v_4) into *stack*.
q_2 is visited once again. The stacks will be changed as shown in Figure 10(d).

Step 7: q_3 is visited once again. The stacks will be changed as shown in Figure 10(e). (We notice that before v_5 is pushed onto $S(q_3)$, v_3 is popped out.) From this, we get another two paths matching P_1: $v_5 \rightarrow v_4 \rightarrow v_1$ and $v_5 \rightarrow v_2 \rightarrow v_1$. They are represented in the compact form.

Step 8: q_4 is visited once again. The stacks will be changed as shown in Figure 10(f). (We notice that before v_5 is pushed onto $S(q_4)$, v_3 is popped out.) From this, we get another two paths matching P_2: $v_5 \rightarrow v_4 \rightarrow v_1$ and $v_5 \rightarrow v_2 \rightarrow v_1$. They are represented in the compact form.

Step 9: *Stack* becomes empty once again. This time q_5 is chosen and in $L(q_5)$ we still have two elements: {v_4, v_6}. Push (q_5, v_4) into *stack*.

q_5 is visited for the second time. The new status of the stacks is shown in Figure 10(g). From this, we get the second path matching P_3: $v_4 \to v_1$.

Step 10: *Stack* becomes empty for the third time, and q_5 is chosen once again since we still have an element in $L(q_5)$: $\{v_6\}$. Push (q_5, v_6) into *stack*.

q_5 is visited for the second time. The new status of the stacks is shown in Figure 10(h) (before v_6 is pushed onto $S(q_5)$, v_2 and v_4 are popped out.) From this, we get the third path matching P_3: $v_6 \to v_1$. □

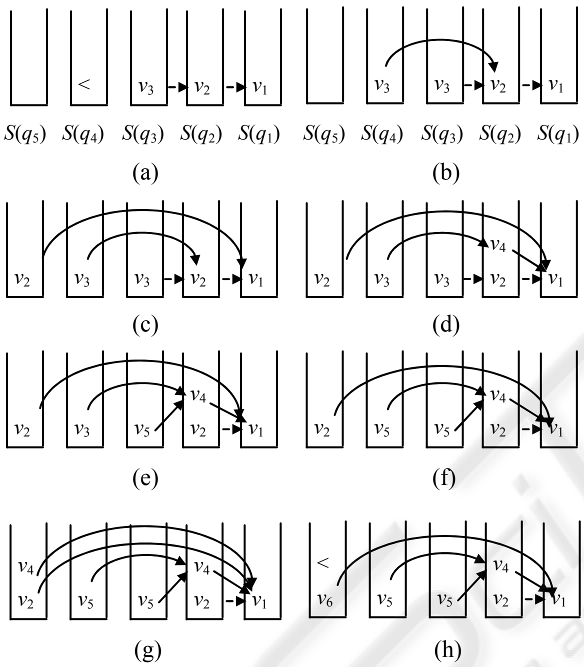

Figure 10: Sample trace.

From the above example, we can see that our algorithm generates the same paths as *TwigStack* although the order of such paths's generation is different. More importantly, in our algorithm *getNext* is not used, which is replaced with *matrixGeneration* that is performed only once.

In the following, we prove the algorithm's correctness and analyze its time complexity.

Proposition 3. Let T and q be the document and query tree, respectively. *RefindTwigStack* generates all the matching paths (in T) for every root-to-leaf path in q.

Proof. In order to prove the proposition, we need to explain

(i) Every path (in T) found by *RefinedTwigStack* must match a root-to-leaf path in q.

(ii) Any path (in T), if it matches a root-to-leaf path in q and each of its nodes satisfies the condition (i) and (ii) given in 2.2, must be found by *RefinedTwigStack*.

Proof of (i). To see that (i) holds, we notice the following two properties of the algorithm:

(1) Any node v in any $L(q_i)$ satisfies the condition (i) and (ii) given in 2.2.

(2) Any node v put in a $S(q_i)$ satisfies the condition below:

Let q' be the parent of q_i. The node on the top of $S(q')$ must be an ancestor of v.

So each time when we meet a leaf node q'', all the paths found must match the path from the root of q to q''.

Proof of (ii). Let $P = q_1 \to q_2 \ldots \to q_m$ be a path in q. Let $\{t_1, t_2, \ldots, t_m\}$ be a set of nodes lying on a path in T, which makes up a matching path of P with each t_i satisfying the condition (i) and (ii) given in 2.2. But this matching path has not been found by *RefinedTwigStack*. Then, there exists a k such that all t_j ($k \le j \le m$) do not have a chance to be put onto the corresponding stacks. First, we notice that $k > 1$ since t_1 must appear in $L(q_1)$ and will be definitely put onto $S(q_1)$ during the computation process. Now we consider t_k with $k > 1$, which does not have a chance to be put onto $S(q_k)$. In terms of line 8 in *RefinedTwigStack*, we must have $S(q_{k+1}) = \{\}$ when we try to put t_k onto $S(q_{k+1})$. This implies that t_{k-1} must have been popped out at a earlier time point when we try to put another node t' onto $S(q_k)$ with $RightPos(t_{k-1}) < LeftPos(t')$. But we have obviously $LeftPos(t') < LeftPos(t_k)$. So we have $RightPos(t_{k-1}) < LeftPos(t_k)$, which contradicts fact that t_{k-1} is an ancestor of t_k. From this analysis, we know that t_k has a chance to be put onto $S(q_k)$. The same analysis applies to $t_{k+1}, \ldots, t_m$. This completes the proof.

The time complexity of *RefinedTwigStack* is easy to analyze. In the whole process, each node v in a $L(q_i)$ is accessed only once. So the total cost is bounded by

$$O\left(\sum_{i=1}^{m} L(q_i)\right) = O(m \cdot n)$$

4 GENERAL CASES

The method discussed in Section 3 can be easily extended to handle general cases that a query tree contains both c-edges and d-edges. For this purpose, we define a third matrix $p(q, T)$ as follows.

An entry $p_{ij} = 1$ indicates that there exists some child k of j, which 'matches' i, *i.e.*, $c_{ik} = 1$; otherwise, $p_{ij} = 0$.

Accordingly, the algorithm *matrixGeneration* should be slightly changed so that the manipulation of $p(q, T)$ is involved.

Algorithm *generalMatrixGeneration*(T, q)
Input: tree T (with nodes 1, ..., n) and tree q (with nodes 1, ..., m)
Output: $c(q, T)$ with values created.
begin
1. **for** u := 1, ..., m **do** {
2. **for** v := 1, ..., n **do**
3. {**if** v satisfies the predicate at u **then**
4. let u_1, ..., u_k be the c-children of u;
5. let u_1', ..., u_g' be the d-children of u;
6. **if** $p_{u_1 v} \wedge ... \wedge p_{u_k v} = 1$ and $d_{u_1' v} \wedge ... \wedge d_{u_k' v} = 1$
7. **then** $c_{uv} \leftarrow 1$;}
8. let v_1, v_2, ..., v_h be the nodes such that $c_{uv_p} = 1$ ($1 \leq p \leq h$);
9. let {w_1, ..., w_r} be a set such that each node in it is an ancestor of some v_p ($1 \leq p \leq h$). Set $d_{uw_l} = 1$ for each w_l ($1 \leq l \leq r$).
10. let {t_1, ..., t_s} be a set such that each node in it is a parent of some v_p ($1 \leq p \leq h$). Set $d_{ut_l} = 1$ for each t_l ($1 \leq l \leq s$).
11. }

end

Since each node u in q may have both c- and d-children, each time when checking it against a node v in T we need to check the corresponding entries in both $d(q, T)$ and $p(q, T)$ (see line 6). In addition, besides the computation of new values for some entries in $d(q, T)$ in each step, we need also to compute new values for the corresponding entries in $p(q, T)$ (see line 10).

5 CONCLUSION

In this paper, a new method is discussed, which substantially improves the method proposed in (Bruno and *et al*, 2002) for doing twig joins that are identified as a core operation for query evaluation in XML databases. Concretely, our method improves the algorithm *TwigStack* and *TwigStackXB* presented in (Bruno and *et al*, 2002) from $O(m^2 \cdot n)$ to $O(m \cdot n)$, where m and n are the sizes of the query tree and document tree, respectively. In addition, a system implementation and experiments are reported, which shows that our method uniformly outperforms *TwigStack*, completely conforming to the conducted theoretic analysis.

ACKNOWLEDGEMENTS

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

REFERENCES

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

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

D. D. Chamberlin, J.Clark, D. Florescu and M. Stefanescu (1999). XQuery1.0: An XML Query Language, http://www.w3.org/ TR/query-datamodel/.

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

A. Deutch, M. Fernandex, D. Florescu, A. Levy, D.Suciu (1999). A Query Language for XML, WWW'99.

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

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

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

The Tukwila System (1999), available from U. of Washington http:/ /data.cs.washington.edu/integration /tukwila/.

The Niagara System (2000), available from U. of Wisconsin http:// www.cs.wisc.edu/niagara/.

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

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

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

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

C. Zhang, J. Naughton, D. Dewitt (2001) Q. Luo, and G. Lohman, on Supporting containment queries in relational database management systems, in *Proc. of ACM SIGMOD*, 2001.