AN EFFICIENT STREAMING ALGORITHM FOR EVALUATING XPATH QUERIES

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

Keywords: XML databases, Trees, Paths, XML pattern matching, XML streams.

Abstract: With the growing importance of XML in data exchange, much research has been done in providing flexible query mechanisms to extract data from XML documents. In this paper, we focus on the query evaluation in an XML streaming environment, in which data streams arrive continuously and queries have to be evaluated even before all the data of an XML document is available. We will propose an algorithm for this issue, working in $O(|T|\cdot Q_{leaf})$ time and $O(|T|\cdot Q_{leaf})$ space, where T_{leaf} stands for the number of the leaf nodes in a document tree T and Q_{leaf} for the number of the leaf nodes in a query tree Q.

1 INTRODUCTION

There is much current interest in processing streaming XML data, using queries expressed with languages such as XPath (World Wide Web Consortium, 2007) and XQuery (World Wide Web Consortium, 2005). A streaming environment, as found with stock market data, network monitoring, or sensor network, differs from non-streaming XPath query processing in the following aspect. In a streaming environment, data streams, which can be potentially infinite, arrive continuously, and must be processed in a single sequential scan due to the limited storage space available. Query results should be distributed incrementally once they are found, possibly before we have read all the data. In addition, the query processing algorithm should scale well in both time and space. An algorithm that meets such an environment for query evaluation over XML data is called a streaming evaluation algorithm.

In this paper, we propose a new algorithm to evaluate queries in such an environment, which 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 a document tree T and in a query tree Q, respectively.

- Data model and query language

Abstractly, an XML document can be considered as a tree structure with each node standing for an element name from a finite alphabet Σ; and an edge for the element-subelement relationship.

In an XML streaming environment, an XML document tree T is modeled as a stream S of modified SAX events: *startElement*(*tag*, *level*, *id*) and *endElement*(*tag*, *level*), where *tag* is the tag of the node being processed, *level* is the level at which the node appears, and *id* is the unique identifier assigned to the node. A node in T exactly corresponds to a *startElement* and (the corresponding *endElement* event) in S. In addition, if an element e has no subelement, a text is possibly associated with its *startElement*.

These events are the input to our query evaluation processor.

On the other hand, queries in XML query languages, such as XPath (World Wide Web Consortium, 2007), XQuery (World Wide Web Consortium, 2005), XML-QL (Dutch *et al*., 1999), and Quilt (Chamberlin *et al.*, 2002; Chamberlin *et al.*, 2000), typically specify patterns of selection predicates on multiple elements that also have some specified tree structured relations. For instance, the following 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

Chen Y. (2008).
AN EFFICIENT STREAMING ALGORITHM FOR EVALUATING XPATH QUERIES.
In *Proceedings of the Fourth International Conference on Web Information Systems and Technologies*, pages 190-196
DOI: 10.5220/0001531101900196

expression can be represented as a tree structure as shown in Figure 1.

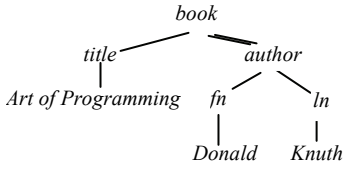


Figure 1: A query tree.

In this tree structure, a node v is labeled with an element name or a string value, denoted as *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. In addition, a node in Q can be a wildcard '*' that matches any element in T. Such a query is often called a twig pattern. In the following discussion, we use *startElement* and *node* interchangeably since each *startElement* event in S exactly corresponds to a node in T.

- *XML query evaluation and tree matching*

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 $u \in Q$, *label*(u) = *label*(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. The purpose of XML query evaluation is to find all the subtrees of T, which contain Q.

Notice that an embedding could map several nodes of the query (of the same label) 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 (Hoffmann and O'Donnell, 1982).

Recently, a great many strategies have been proposed to evaluate XPath queries in an XML streaming environment (Avila *et al.*, 2002; Chen *et al.*, 2006; Ives *et al.*, 2002; Koch *et al.*, 2004; Ludascher *et al.*, 2002; Peng and Chawathe, 2003; Peng *et al.*, 2003). The methods discussed in (Avila *et al.*, 2002; Ives *et al.*, 2002) are based on finite state automata (*FSA*), but only able to handle single path queries, i.e., a query containing branching cannot be processed, as observed in (Peng and Chawathe, 2003). The method proposed in (Peng and Chawathe, 2003) is a general strategy, but requires exponential time ($O(|T| \times 2^{|Q|})$) in the worst case, as analyzed in (Peng *et al.*, 2003). The methods discussed in (Koch *et al.*, 2004; Ludascher *et al.*, 2002) do not support d-edges. If we extend them to general cases, exponential time is required. Up to now, the research culminates in *TwigM* presented in (Chen *et al.*, 2006). It is not only a general-case algorithm, but also works in polynomial time. In the worst case, its time complexity is bounded by $O(T_h Q_d |Q||T| + |Q|^2|T|)$, where T_h is the height of T and Q_d is the largest outdegree of a node in Q. By this method, each node q of Q is associated with a boolean array of length Q_d and a stack of size T_h, in which each element is a node v from T such that its relationship with the nodes in the stack associated with q's parent q' satisfies the relationship between q and q'. Therefore, each time to figure out a stack and push a node into it, $O(T_h Q_d |Q|)$ time is required, leading to a time complexity of $O(T_h Q_d |Q||T| + |Q|^2|T|)$. See Theorem 4.4 in (Chen *et al.*, 2006).

The remainder of the paper is organized as follows. In Section 2, we discuss an algorithm for simple cases that a twig pattern contains only d-edges, as well as wildcards and branches. In Section 3, we extend this algorithm to general cases. Finally, a short conclusion is set forth in Section 4.

2 ALGORITHM FOR SIMPLE CASES

In this section, we describe an algorithm for simple cases that a twig pattern contains only d-edges, wildcards and branches. First, we give a basic algorithm in 2.1. Then, in 2.2, we prove the correctness of the algorithm and analyze its computational complexities.

2.1 Basic Algorithm

Recall that in a streaming environment, the input to the XML query processor is a steam of modified SAX events; and an event is either *s*tartElement(*tag*, *level*, *id*) or *endElement*(*tag*, *level*). In order to evaluate a query Q, we have to scan a stream S from the beginning to the end and report any *startElement* event once the corresponding subtree is found containing Q.

For this purpose, we will maintain a global *stack* structure with each entry in it being a triplet: <*e*, *p*, *c*>, where *e* is a *startElement* event, *p* is a pointer to an entry in *stack* where its parent *startElement* is stored and *c* a pointer to the head of a linked list containing all the nodes constructed for its child elements, as illustrated in Figure 2.

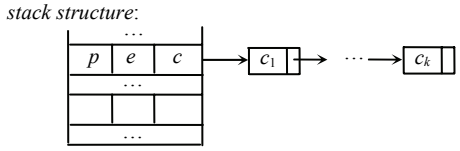


Figure 2: Illustration for *stack* structure.

During the process, two other data structures are also maintained and computed to facilitate the discovery of subtree matchings according to Definition 1.

- Each node v (corresponding to a *startElement* event in S) in a document tree T is associated with a set, denoted $\alpha(v)$, contains all those nodes q in Q such that $Q[q]$ can be imbedded into $T[v]$.
- 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.

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

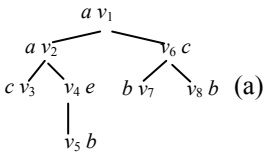
Below is the algorithm, which takes an event stream S and a twig pattern Q as the input. During the process, S is scanned from the beginning to the end and once a *startElement* event is found such that the subtree rooted at the corresponding node contains Q it will be reported.

In the algorithm, a virtual *startElement* event is used, which is considered to be the parent of the first *startElement* event in S (which corresponds to the root of T). The *level* number of the virtual event is set to be -1, and its *tag* and *id* are both set to be *nil*. Two variables E and E' are used. E' is for the current *startElement* event being processed while E is to store the parent of the current *startElement* event. In addition, each time a node v is constructed, a subprocedure *containment-check*(v, Q) is invoked to find all those $q \in Q$ such that $T[v]$ contains $Q[q]$ and store them in $\alpha(v)$.

Algorithm *query-evaluation*(S, Q)
input: S - an XML stream; Q - a twig pattern.
output: report any *startElement* such that for the corresponding node v, $T[v]$ contains Q.
begin
1. *push*(the first element of S, *stack*);
2. E := virtual event;
3. **while** *stack* is not empty **do** {
4. E' := *top*(*stack*);
   (*check the top element in *stack**)
5. $E'.p$:= address of E;
   (*establish parent link for E'*)
6. let e be the next element in S;
7. **if** e is a *startElement* event **then** {
8. $E := E'$;
9. *push*(e, *stack*);
10. }
11. **else** (*e is an *endElement* event.*)
12. {$E'' :=$ *pop*(*stack*);
    (*pop the top element out of *stack**)
13. generate node v for E''; $E := E''.p$;
14. append v to the end of $(E''.p).c$;
15. call *containment-check*(v, Q);
16. }
17. }
end

The above algorithm processes the events in S one by one. Therefore, the corresponding document tree T is searched in the depth-first traversal fashion. Each time a *startElement* event is encountered, it will be pushed into *stack* (see line 1 and lines 6 - 9) and stay there until its corresponding *endElement* is encountered (see lines 11 - 12). In this case, it will be popped out of *stack* and a node v for it will be constructed (see line 13), for which a containment check will be performed (see line 15).

Example 1. Consider the document tree T in Figure 3(a). Its XML stream S is shown in Figure 3(b). Applying the algorithm *query-evaluation*( ) to S, we will regain T if line 15 is not executed. In Figure 4, we trace the first 8 steps of the execution process.



(a)

1. *startE*(*a*, 0, 1) 9. *endE*(*a*, 1)
2. *startE*(*a*, 1, 2) 10. *startE*(*c*, 1, 6)
3. *startE*(*c*, 2, 3) 11. *startE*(*b*, 2, 7)
4. *endE*(*c*, 2) 12. *endE*(*b*, 2)
5. *startE*(*e*, 2, 4) 13. *startE*(*b*, 2, 8)
6. *startE*(*b*, 3, 5) 14. *endE*(*b*, 2)
7. *endE*(*b*, 3) 15. *endE*(*c*, 1)
8. *endE*(*e*, 2) 16. *endE*(*a*, 0)

(b)

Figure 3: A document tree and its XML stream.

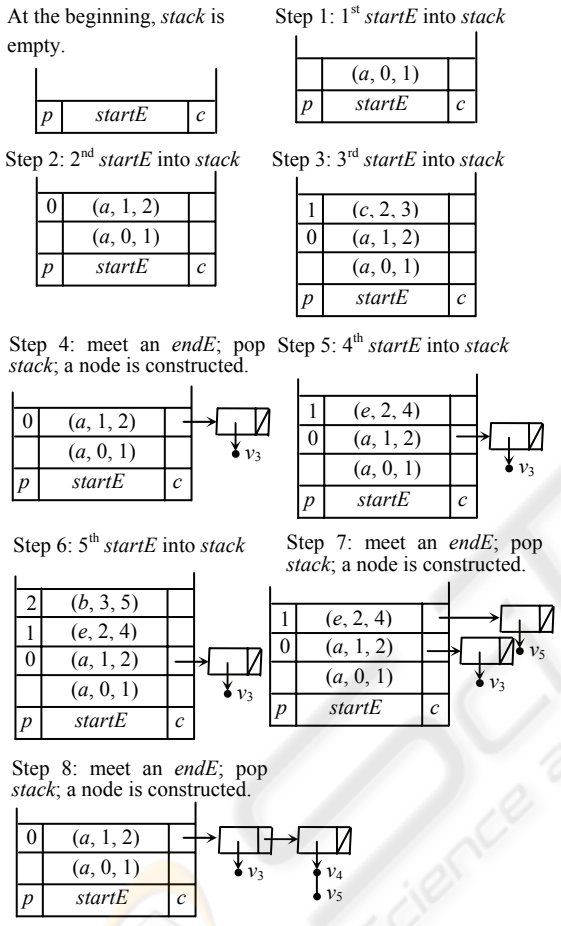


Figure 4: Illustration for for $L(q_i)$'s.

From the above discussion, we can see that a document tree can always be constructed by scanning the corresponding XML stream *S*. For the purpose of query evaluation, however, we have to check the containment each time a node of *T* is constructed. This is done by calling *containment-check*(*v*, *Q*), in which another two functions are invoked to do different checkings:

- *element-check*(*u*, *q*): *u* is an element containing subelements. It checks whether *T*[*u*] contains *Q*[*q*]. If it is the case, return {*q*}. Otherwise, it returns an empty set ∅.
- *bottom-element-check*(*u*, *Q*): *u* is an element containing no subelement. It returns a set of nodes in *Q*: {q_1, ..., q_k} such that for each q_i ($1 \le i \le k$) the following conditions are satisfied.
  (i) *label*(*u*) = *label*(q_i).
  (ii) if q_i has a child, then the child must be a text and matches the text associated with *u*.

Algorithm *containment-check*(*v*, *Q*)
input: *v* - a node in *T*; *Q* - a twig pattern.
output: a(*v*) - a set of query node *q* such that *T*[*v*] contains *Q*[*q*].
begin
1. $C := \emptyset$; $C_1 := \emptyset$; $C_2 := \emptyset$;
2. **if** *v.c* is not *nil* **then** (*v has some subelements.*)
3. {let v_1, ..., v_k be the child nodes of *v*;
4. $\alpha := \alpha(v_1) \cup ... \cup \alpha(v_k)$;
5. **for** each $q \in \alpha$ **do**
6. {$\delta(q) := v$; $C := C \cup$ {*q*'s parent};}
7. remove all $\alpha(v_j)$ (j = 1, ..., k);
8. **for** each *q'* in *C* **do**
9. $C_1 := C_1 \cup$ *element-check*(*v*, *q'*);
10. }
11. C_2 := *bottom-element-check*(*v*, *Q*);
12. $\alpha(v) := \alpha \cup C_1 \cup C_2$;
end

Function *element-check*(*u*, *q*)
begin
1. $C_1 := \emptyset$;
2. **if** *label*(*q*) = *label*(*u*) **then**
(*If *q* is '*', the checking is always successful.*)
3. {let q_1, ..., q_k be the child nodes of *q*;
4. **if** for each q_i (i = 1, ..., k) d(q_i) is equal to *u*
5. **then** {C_1 := {*q*};
6. **if** *q* is *root* **then** report *u*};}
7. return C_1;
end
Function *bottom-element-check*(*u*, *Q*)
begin
1. $C_2 := \emptyset$; *flag* := *false*;
2. **for** each leaf node *q* in *Q* **do** {
3. **if** *q* is a text **then** {
4. let *q'* be the parent of *q*;
5. **if** *label*(*q'*) = *label*(*u*) and
q matches the text associated with *u* **then**
{$C_2 := C_2 \cup$ {*q'*}; *flag* := *true*;
6. }
7. **else** {
8. **if** *label*(*q*) = *label*(*u*) **then** {
9. $C_2 := C_2 \cup$ {*q*}; *flag* := *true*;
10. }
11. **if** *q* is *root* and *flag* := *true* **then** report *u*;

12. *flag* := *false*;
13. }
14. return C_2;

end

One of the inputs to the algorithm *containment-check*( ) is a node v constructed in the execution of *query-evaluation*(S, Q). If v corresponds to an element that has no subelement, the function *bottom-element-check*( ) is called (see line 11), by which a(v) will be established by checking it against all the leaf nodes of Q. Otherwise, $\alpha(v_i)$ will be checked for all the child nodes v_i of v (see lines 3 -6). Concretely, for each q in α (= $\alpha(v_1) \cup ... \cup \alpha(v_k)$), the value of $\delta(q)$ will be changed to v. Meanwhile, q's parent will be stored in a temporary variable C. Then, all the nodes q' in C are the candidates to be further checked. This is done by calling *element-check*(v, q') to see whether $T[v]$ contains $Q[q']$ (see lines 8 - 9). Special attention should be paid to the fact that *bottom-element-check*( ) should also be applied to v to find all the leaf nodes of Q which matche v.

Finally, we notice that in the execution of *element-check*( ), $\delta(q)$'s are utilized to facilitate the checkings (see lines 3 - 5 in *element-check*( )).

The following example helps for illustration.

Example 2. Consider T and S shown in Figure 3 and Q shown in Figure 5.

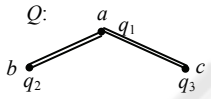


Figure 5: A tree pattern query.

By executing *query-evaluation*(S, Q), the nodes of T will be constructed bottom up.

First, v_3 in T is constructed. It is a leaf node, matching q_3 of the two leaf nodes in Q. Therefore, $\alpha(v_3) = \{q_3\}$ (see lines 11). In the same way, we will set $\alpha(v_5) = \{q_2\}$. In a next step, v_4 is constructed. 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. 6 for illustration.) After that, *element-check*(v_4, q_1) is invoked. (Note that q_1 is the parent of q_2. See lines 8 - 9.) Since *label*(v_4) ≠ *label*(q_1), it returns $C_1 = \varnothing$. *bottom-element-check*(v_4) also returns $C_2 = \varnothing$. So $\alpha(v_4) = \alpha(v_5) \cup C_1 \cup C_2 = \{q_2\}$ (see line 12). When v_2 is constructed, 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 *element-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, $C_1 = \{q_1\}$. Thus, we set $\alpha(v_2) = \alpha(v_3) \cup \alpha(v_4) \cup C_1 \cup C_2 = \alpha(v_3) \cup \alpha(v_4) \cup \{q_1\} \cup \varnothing = \{q_1, q_2, q_3\}$.

In a next step, v_7 will be constructed. It is a leaf node, matching q_2. Therefore, $\alpha(v_7) = \{q_2\}$. Similarly, we will set $\alpha(v_8) = \{q_2\}$. When v_6 is constructed, we will change $\delta(q_2)$ to v_6 (according to $\alpha(v_7) = \alpha(v_8) = \{q_2\}$), but $\delta(q_3)$ (= v_2) remains not modified. *element-check*(v_6, q_1) will return $\varnothing$. Thus, $\alpha(v_6) = \alpha(v_7) \cup \alpha(v_8) \cup C_1 \cup C_2 = \{q_2, q_3\}$. Finally, we will meet v_1 and set $\delta(q_1) = v_1$, $\delta(q_2) = v_1$, and $\delta(q_3) = v_1$. Since *label*(v_1) = *label*(q_1), $\delta(q_2) = v_1$ and $\delta(q_3) = v_1$, *element-check*(v_1, q_1) returns $\{q_1\}$. So $\alpha(v_1)$ is equal to $\alpha(v_2) \cup \alpha(v_6) \cup C_1 \cup C_2 = \{q_1, q_2, q_3\}$.

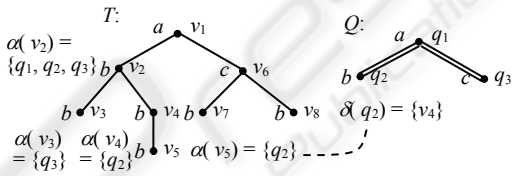


Figure 6: Sample trace.

2.2 Correctness and Computational Complexities

In this subsection, we prove the correctness of *containment-check*( ) and analyze its computational complexities.

Proposition 1. Let v be a node in T. Then, for each q in a(v) generated by *containment-check*( ), we have that $T[v]$ 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') ≤ 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*($Q[q_j]$) ≤ 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 must have $q_j \in \alpha(v_i)$ 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, Q can be embedded into $T[v]$. So r_Q will be 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 unifying $\alpha(v_1)$, ..., $\alpha(v_k)$, where v_i (i = 1, ..., k) is a child node of some node v in T. This part of cost is bounded by
$$O(\sum_{i}^{|T|} d_i \mid Q \mid) = O(|T||Q|),$$
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 α (= $\alpha(v_1) \cup ... \cup \alpha(v_k)$). Since the size of a is bounded by $O(|Q|)$, so this part of cost is also bounded by $O(|Q|)$.
3. The third part is the time for checking a node v_i in T against each node q_j in an S. This can be estimated by the following sum:
$$O(\sum_{i}^{|T|}\sum_{j}^{|C|} c_j) \le O(\sum_{i}^{|T|}\sum_{i}^{|Q|} c_j) = O(|T||Q|),$$
Where c_j represents the ourdegree of a node q_j in S.
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
$$O(\sum_{i}^{|T|} \mid Q \mid) = O(|T||Q|).$$

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

Proposition 2. The time complexity of *containment-check*( ) is bounded by $O(|T||Q|)$.

Proof. See the above discussion.

However, this computational complexity can be improved by reducing the size of each $\alpha(v)$.

For this purpose, we assign each node q in Q a pair of numbers as follows. By traversing Q in *preorder*, each node q will obtain a number *pre*(q) to record the order in which the nodes of the tree are visited. In a similar way, by traversing Q in *postorder*, each node q will get another number *post*(q). These two numbers can be used to characterize the ancestor-descendant relationships as follows.

Let q and q' be two nodes of a tree Q. Then, q' is a descendant of q iff *pre*(q') > *pre*(q) and *post*(q') < *post*(q). See Exercise 2.3.2-20 in [15].

In addition, if *pre*(q') < *pre*(q) and *post*(q') < *post*(q), q' is to the left of q.

Assume that q and q' are two query nodes appearing in $\alpha(v)$. If q' is a descendant of q, then we can remove q' from $\alpha(v)$ since the containment of $Q[q]$ in $T[v]$ implies the containment of $Q[q']$ in $T[v]$. This can be done as follows.

First of all, we notice that the algorithm searches T bottom-up. For a leaf node v in T, $\alpha(v)$ is initialized with all those leaf nodes in Q, which match v. This can be carried out by searching the leaf nodes in Q from left to right. Then, for any two leaf nodes q and q' in $\alpha(v)$, if q' appears before q, we have that *pre*(q') < *pre*(q) and *post*(q') < *post*(q). That is, $\alpha(v)$ is initially sorted by the *pre* and *post* values. We can store $\alpha(v)$ as a linked list. Let α_1 and α_2 be two sorted lists with $|\alpha_1| \le Q_{leaf}$ and $|\alpha_2| \le Q_{leaf}$. The union of α_1 and α_2 ($\alpha_1 \cup \alpha_2$) can be performed by scanning both α_1 and α_2 from left to right and inserting the elements in α_2 into α_1 one by one. During this process, any element in α_1, if it is a descendant of some element in a_2, will be removed; and any element in α_2, if it is a descendant of some element in α_1, will not be inserted into α_1. The result is stored in α_1. Obviously, the resulting linked list is still sorted and its size is bounded by Q_{leaf}. We denote this process as *merge*(α_1, α_2) and define *merge*(α_1, ..., α_{k-1}, α_k) to be *merge*(*merge*((α_1, ..., α_{k-1}), α_k). In this way, the time and space complexities of the algorithm can be improved to $O(|T|Q_{leaf})$ and $O(T_{leaf}Q_{leaf})$, respectively.

3 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, as well as wildcards and branches.

Let q_1, ..., q_k be the child nodes of q. Let v_1, ..., v_l be the child nodes of v. If $T[v]$ contains $Q[q]$, the following two conditions must hold:

- for each *c*-edge (q, q_i) ($1 \le i \le k$), there must exist a v_j ($1 \le j \le l$) such that (v, v_j) matches (q, q_i), and
- $T[v_j]$ contains $Q[q_i]$.

In terms of this analysis, we modify Algorithm *containment-check*( ) as follows.

Algorithm *general-containment-check*(v, Q)
input: v - a node in T; Q - a twig pattern.
output: $\alpha(v)$ - a set of query node q such that $T[v]$ contains $Q[q]$.
begin
1. $C := \varnothing$; $C_1 := \varnothing$; $C_2 := \varnothing$;
2. **if** *v.c* is not *nil* **then**(*v has some subelements.*)
3. {let v_1, ..., v_k be the chi ld nodes of v;
4. **for** i = 1 to k **do** {
5. **for** $q \in \alpha(v_i)$ **do** {
6. **if** ((q is a *d*-child) or
7. (q is a *c*-child and q matches v_i))
8. **then** $\delta(q) := v$
9. }}
10. α := merge($\alpha(v_1)$, ..., $\alpha(v_k)$);

11. assume that $\alpha = \{q_1, ..., q_j\}$;
12. **for** i = 1 to j **do** {
13. **if** (q_i's parent ≠ q_i-1's parent)
then $C := C \cup \{q_i$'s parent$\}$;}
14. remove all a(v_j) (j = 1, ..., k);
15. **for** each q in C **do**
16. $C_1 := C_1 \cup$ *element-check*(v, q);
17. }
18. S_2 := *bottom-element-check*(v);
29. $\alpha(v)$:= merge(α, C_1, C_2);
end

The first difference of the above algorithm from the algorithm *containment-check*( ) is that before we set the value for $\delta(q)$ we will check whether q is a *d*-child or a *c*-child. If q is a *c*-child, we will further check whether it matches v_i (see lines 6 - 8). We notice that q appearing in $\alpha(v_i)$ only indicates that $Q[q]$ can be embedded into $T[v_i]$, but not necessarily means that q matches v_i.

The second difference is line 10 and lines 12 - 13. In line 10, we use the merge operation to union $\alpha(v_1)$, ..., and $\alpha(v_k)$ together. In lines 12 -13, we generate a set C that contains the parent nodes of all those nodes appearing in α (= merge($\alpha(v_1)$, ..., $\alpha(v_k)$), where v_j is a child node of the current node v. Since the nodes in a are sorted (according to the nodes' *pre* and *post* values), 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 a) into C, 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 C is also bounded by $O(Q_{leaf})$.

4 CONCLUSIONS

In this paper, an efficient algorithm for the query evaluation in an XML streaming environment is presented. The algorithm runs in $O(|T|\cdot Q_{leaf})$ time

and $O(|T|\cdot Q_{leaf})$ space, where T_{leaf} stands for the number of the leaf nodes in a document tree T and Q_{leaf} for the number of the leaf nodes in a query tree Q. This computational complexity is much better than any existing strategy for this problem.

ACKNOWLEDGEMENTS

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

REFERENCES

I. Avila-Campillo, T.J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu (2002), XMLTK: An XML Toolkit for Scalable XML Stream Processing, in *Programming Langauge Technologoes for XML(PLAN-X)*, 2002.

D.D. Chamberlin, J.Clark, D. Florescu and M. Stefanescu (2002) 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*.

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

A. Dutch, M. Fernandez, D. Florescu, A. Levy, D. Suciu (1999), A Query Language for XML, in: *Proc. 8th World Wide Web Conf.*, May 1999, pp. 77-91.

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

Z.G. Ives, A.Y. Halevy, and D.S. Weld (2002), An XML query engine for network-bound data, *VLDB Journal*, 11(4), 2002.

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

C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier (2004), Schema-based Scheduling of Event Processor and Buffer Minimization for Queries on Structured Data Stream, in: *Proc. of VLDB*, 2004.

B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou (2002), A Transducer-based XML Query Processor, in: *Proc. of VLDB*, 2002.

F. Peng and S.S. Chawathe (2003), XPath queries on streaming data, in: *Proc. of SIGMOD*, 2003.

F. Peng and S.S. Chawathe (2003), XSQ: A Streaming XPath Engine, Technical Report CS-TR-4493, University of Maryland, 2003.

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

World Wide Web Consortium (2005). XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, 2005. See http://www.w3.org/TR/xquery.