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 $\Sigma$; 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: $\text{startElement}(\text{tag}, \text{level}, \text{id})$ and $\text{endElement}(\text{tag}, \text{level})$, where $\text{tag}$ is the tag of the node being processed, $\text{level}$ is the level at which the node appears, and $\text{id}$ is the unique identifier assigned to the node. A node in $T$ exactly corresponds to a $\text{startElement}$ and (the corresponding $\text{endElement}$ event) in $S$. In addition, if an element $e$ has no subelement, a text is possibly associated with its $\text{startElement}$.

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

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

In this tree structure, a node $v$ is labeled with an element name or a string value, denoted as $\text{label}(v)$. In addition, there are two kinds of edges: child edges ($c$-edges) for parent-child relationships, and descendant edges ($d$-edges) for ancestor-descendant relationships. A $c$-edge from node $v$ to node $u$ is denoted by $v \rightarrow u$ in the text, and represented by a single arc; $u$ is called a $c$-child of $v$. A $d$-edge is denoted $v \Rightarrow u$ in the text, and represented by a double arc; $u$ is called a $d$-child of $v$. 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 $\text{startElement}$ and $\text{node}$ interchangeably since each $\text{startElement}$ 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 $u \in Q$, $\text{label}(u) = \text{label}(f(u))$.

(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 embebed 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 $O(T_d|Q_d|T + O^2|T|)$, where $T_d$ 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 $T_n$, 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, $O(T_dQ_d|Q)$ time is required, leading to a time complexity of $O(T_dQ_d|Q|T + O^2|T|)$. 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 stream of modified SAX events; and an event is either startElement(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.

\[
\text{stack structure: } \ldots \rightarrow p \rightarrow e \rightarrow c \rightarrow \ldots \rightarrow c_1 \rightarrow \ldots
\]

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 \( \phi \). 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 \( q_1, \ldots, q_l \) be the child nodes of \( q \). For each \( q_i \) (\( i = 1, \ldots, k \)), check whether \( \delta(q_i) \) is equal to \( v \).

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);
5. \( E' \cdot p := \) address of \( E' \);
6. \( E := \) E';
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);
13. generate node \( v \) for \( E' \); \( E := E' \cdot p \);
14. append \( v \) to the end of \( \{E' \cdot p \} c \);
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.
AN EFFICIENT STREAMING ALGORITHM FOR EVALUATING XPATH QUERIES

Figure 3: A document tree and its XML stream.

At the beginning, stack is empty.

Step 1: 1st startE into stack

Step 2: 2nd startE into stack

Step 3: 3rd startE into stack

Step 4: meet an endE; pop stack; a node is constructed.

Step 5: 4th startE into stack

Step 6: 5th startE into stack

Step 7: meet an endE; pop stack; a node is constructed.

Step 8: meet an endE; pop stack; a node is constructed.

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 \( \text{containment-check}(v, Q) \), in which another two functions are invoked to do different checkings:

- \( \text{element-check}(u, q) \): \( u \) 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 \( \emptyset \).
- \( \text{bottom-element-check}(u, Q) \): \( u \) is an element containing no subelement. It returns a set of nodes in \( Q^* \) \( \{ q_1, ..., q_k \} \) such that for each \( q_i (1 \leq i \leq k) \) the following conditions are satisfied.
  1. \( \text{label}(u) = \text{label}(q_i) \).
  2. If \( v \cdot c \) is not \( \text{nil} \) then \((\ast v \) has some subelements.\( \ast)\)
     a. \( \{ v_1, ... v_k \} \) be the child nodes of \( v \);
     b. \( \alpha := \alpha(v_1) \cup ... \cup \alpha(v_k) \);
     c. \( \text{for each } q \in \alpha \text{ do} \)
        a. \( \delta(q) := v \cdot C := C \cup \{ q \text{’s parent} \} \);
        b. \( \text{remove all } \alpha(v_i) \text{ for } i = 1, ..., k \);
        c. \( \text{for each } q \text{ in } C \text{ do} \)
           a. \( C_1 := C_1 \cup \text{element-check}(v, q) \);
           b. \( C_2 := \text{bottom-element-check}(v, Q) \);
           c. \( \alpha(v) := \alpha \cup C_1 \cup C_2 \);
     d. \( \text{end} \).

Function \( \text{element-check}(u, q) \)

begin
  1. \( C_1 := \emptyset \);
  2. \( \text{if label}(q) = \text{label}(u) \text{ then} \) (*If \( q \) is \( \ast \ast \), the checking is always successful.\( \ast \ast)\)
    a. \( \{ v_1, ... v_k \} \) be the child nodes of \( v \);
    b. \( \text{for each } q_i (i = 1, ..., k) \text{ do} \)
       a. \( q_i \text{ is root then report } u \};\)
       b. \( \text{report } C_i \);
     c. \( \text{end} \).
  3. \( \text{end} \).

Function \( \text{bottom-element-check}(u, Q) \)

begin
  1. \( C_2 := \emptyset \); \( \text{flag} := \text{false} \);
  2. \( \text{for each leaf node } q \text{ in } Q \text{ do} \) 
     a. \( \text{if } q \text{ is a text then} \) 
        a. \( \text{let } q^* \text{ be the parent of } q \);
        b. \( \text{if } \text{label}(q^*) = \text{label}(u) \text{ and} \)
           a. \( q \text{ matches the text associated with } u \text{ then} \)
              a. \( C_2 := C_2 \cup \{ q \}; \text{flag} := \text{true} \); 
              b. \( \text{end} \).
        c. \( \text{end} \).
     b. \( \text{else} \) 
        a. \( \text{if } \text{label}(q) = \text{label}(u) \text{ then} \)
           a. \( C_2 := C_2 \cup \{ q \}; \text{flag} := \text{true} \);
           b. \( \text{end} \).
     c. \( \text{end} \).
  3. \( \text{if } q \text{ is root and flag := true then report } u \);
12. flag := false;
13. return C;
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 α(v) will be established by checking it against all the leaf nodes of Q. Otherwise, α(v) will be checked for all the child nodes v of v (see lines 3 -6). Concretely, for each q in α(= α(v1) ∪ ... ∪ α(vn)), the value of δ(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 match v.

Finally, we notice that in the execution of element-check(), δ(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.](image)

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

First, v3 in T is constructed. It is a leaf node, matching q3 of the two leaf nodes in Q. Therefore, α(v3) = {q3} (see lines 11). In the same way, we will set α(v1) = {q1}. In a next step, v1 is constructed. It is the parent of v3. In terms of α(v1) = {q1}, δ(q1) is set to be v1 (see Figure 6 for illustration.) After that, element-check(vh, q1) is invoked. (Note that q1 is the parent of q2. See lines 8 -9). Since label(v1) ≠ label(q1), it returns C1 = ∅. bottom-element-check(vh) also returns C2 = ∅. So α(v1) = α(v2) ∪ C1 ∪ C2 = {q1} (see line 12). When v2 is constructed, we will first set δ(q1) = δ(q2) = v2 (in terms of α(v1) = {q1} and α(v2) = {q2}, respectively). Next, we call element-check(v2, q1), in which we will check whether label(v2) = label(q1). It is the case. So we will further check whether δ(q2) (i = 2, 3) is equal to v2. Since both δ(q2) and δ(q3) are equal to v2, we have that T[v2] contains Q(q1). Therefore, C1 = {q1}. Thus, we set α(v2) = α(v1) ∪ α(v2) ∪ C1 ∪ C2 = α(v1) ∪ α(v2) ∪ {q1} ∪ ∅ = {q1, q2, q3}.

In a next step, v3 will be constructed. It is a leaf node, matching q2. Therefore, α(v3) = {q2}. Similarly, we will set α(v4) = {q2}. When v6 is constructed, we will change δ(q2) to v6 (according to α(v1) = α(v6) = {q2}), but δ(q3) (v2) remains not modified. element-check(v6, q1) will return ∅. Thus, α(v6) = α(v1) ∪ α(v6) ∪ C1 ∪ C2 = {q2, q3}. Finally, we will meet v1 and set δ(q1) = v1, δ(q2) = v1, and δ(q3) = v1. Since label(v1) = label(q1), δ(q2) = v1 and δ(q3) = v1, element-check(v1, q1) returns {q1}. So α(v1) is equal to α(v2) ∪ α(v3) ∪ C1 ∪ C2 = {q1, q2, q3}.

![Figure 6: Sample trace.](image)

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 α(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 r0 be the root of Q. Let q1, ..., qk be the child nodes of r0. Then, we have height(Q(qj)) ≤ h (j = 1, ..., k). In terms of the induction hypothesis, for each q in Q(qj) (j = 1, ..., k), it follows from α(v) (where v is a child node of v), we have T[v] contains Q(qj) and δ(qj) will be set to be v. Especially, if T[v] contains Q(qj) (j = 1, ..., k), we must have qj ∈ α(v) and δ(qj) will be set to be v before v is checked against r0. Obviously, if label(v) = label(r0) and for each qj (j = 1, ..., k), δ(qj) is equal to v, Q can be embedded into T[v]. So r0 will be inserted into α(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 \( \alpha(v_i) \), \( \ldots, \alpha(v_k) \), where \( v_i (i = 1, \ldots, k) \) is a child node of some node \( v \) in \( T \). This part of cost is bounded by
   \[
   O\left(\sum_{i} d_i |Q|\right) = O(|T||Q|),
   \]
   where \( d_i \) represents the outdegree of a node \( v_i \) in \( T \).

2. The second part is the time used for generating \( S \) from \( \alpha(\cup \ldots \cup \alpha(v_i)) \). Since the size of \( S \) 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 \( q_i \) in \( S \). This can be estimated by the following sum:
   \[
   O\left(\sum_{j} \sum_{k} c_{jk} \right) \leq O\left(\sum_{j} q_j \right) = O(|T||Q|),
   \]
   where \( c_{jk} \) represents the outdegree 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
   \[
   O\left(\sum_{j} q_j \right) = O(|T||Q|).
   \]

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 \( q \) if \( pre(q') > pre(q) \) 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 \( \alpha_1 \) and \( \alpha_2 \) be two sorted lists with \( |\alpha_1| \leq Q_{leaf} \) and \( |\alpha_2| \leq Q_{leaf} \). The union of \( \alpha_1 \) and \( \alpha_2 \) \( (\alpha_1 \cup \alpha_2) \) can be performed by scanning both \( \alpha_1 \) and \( \alpha_2 \) from left to right and inserting the elements in \( \alpha_2 \) into \( \alpha_1 \) one by one. During this process, any element in \( \alpha_1 \), if it is a descendant of some element in \( \alpha_2 \), will be removed; and any element in \( \alpha_2 \), if it is a descendant of some element in \( \alpha_1 \), will not be inserted into \( \alpha_1 \). The result is stored in \( \alpha_1 \). Obviously, the resulting linked list is still sorted and its size is bounded by \( Q_{leaf} \). We denote this process as \( merge(\alpha_1, \alpha_2) \) and define \( merge(\alpha_1, \ldots, \alpha_k) \) to be \( merge(merge(\alpha_1, \ldots, \alpha_{k-1}), \alpha_k) \). 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 \( q_1, \ldots, q_l \) be the child nodes of \( q \). Let \( v_1, \ldots, v_l \) be the child nodes of \( v \). If \( T[v] \) contains \( Q[q] \), the following two conditions must hold:
- For each \( c \)-edge \( (q, q_j) (1 \leq j \leq l) \), there must exist a \( v_j (1 \leq j \leq l) \) such that \( (v, v_j) \) matches \( (q, q_j) \), and
- \( T[v] \) contains \( Q[q] \).

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

**Algorithm** general-containment-check \((v, Q)\)

---

1. \( C := \emptyset; C_1 := \emptyset; C_2 := \emptyset; \)
2. if \( v \in C \) then (*v has some subelements.*)
3. \{ \begin{align*}
   &\text{let } v_1, \ldots, v_k \text{ be the child nodes of } v; \\
   &\text{for } i = 1 \text{ to } k \text{ do } \}
4. &\text{for } q \in \alpha(v_i) \text{ do } \}
5. &\text{if } (q \text{ is a } d\text{-child}) \text{ or } \}
6. &\text{if } (q \text{ is a } c\text{-child and } q \text{ matches } v_j) \}
7. &\text{then } \delta(q) := v \}
8. 
9. \}
10. \alpha := merge(\alpha(v_1), \ldots, \alpha(v_k));
11. assume that \( \alpha = \{q_1, ..., q_k\} \);
12. for \( i = 1 \) to \( k \) do 
13. if \( (q_i's\ parent \neq q_j, q_j's\ parent) \)
    then \( C := C \cup \{q_i's\ parent\}; \)
14. remove all \( a(v_i) (i = 1, ..., k) \);
15. for each \( q \) in \( C \) do
16. \( C_i := C_i \cup \text{element-check}(v, q); \)
17. \( \}
18. \( S_2 := \text{bottom-element-check}(v); \)
29. \( \alpha(v) := \text{merge}(\alpha, C_1, C_2); \)

The first difference of the above algorithm from the algorithm \( \text{containment-check}( ) \) is that before we set the value for \( \delta(q) \), we will check whether \( q \) is a 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 \( \alpha (= \text{merge}(\alpha(v_1), ..., \alpha(v_k)) \), where \( v_j \) is a child node of the current node \( v \). Since the nodes in a are sorted (according to the nodes' \( \text{pre} \) and \( \text{post} \) values), if there are more than one nodes in \( \alpha \) 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 \( O(Q_{\text{leaf}}) \).

4 CONCLUSIONS

In this paper, an efficient algorithm for the query evaluation in an XML streaming environment is presented. The algorithm runs in \( O(T_{\text{leaf}}Q_{\text{leaf}}) \) time and \( O(T_{\text{leaf}}Q_{\text{leaf}}) \) space, where \( T_{\text{leaf}} \) stands for the number of the leaf nodes in a document tree \( T \) and \( Q_{\text{leaf}} \) 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