Modal logic and navigational XPath: an experimental comparison

Massimo Franceschet[1,2] and Enrico Zimuel[1,2]

[1] Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands
[2] Dipartimento di Scienze, Università "Gabriele D'Annunzio", Pescara, Italy

Abstract. XPath is the core retrieval language of XQuery, the official query language for XML data. We empirically compare three query evaluation strategies for the navigational fragment of XPath known as Core XPath: a bottom-up algorithm based on model checking techniques for multi-modal logic, a first top-down procedure based on a technique to eliminate XPath filters, and a second top-down procedure that takes advantage of the pre/post plane representation of an XML tree. We implement the three methods and we benchmark the resulting XPath processors using a fragment of XPathMark, a recently proposed benchmark for XPath.

1 Introduction

The *Extensible Markup Language* (XML) [1] is a popular representation language for semistructured data [2], which are data that do not necessarily possess a regular schema. The XML Path Language (XPath) [3] is a simple retrieval language for XML data. XPath in the core retrieval fragment of the XML Query Language (XQuery) [4], the standard query language for XML.

XPath and modal logic are similar in many respects. Syntactically, the XPath language contains navigational axes that closely resemble modal logic *modalities*. Semantically, XPath queries are evaluated on XML trees, which are tree-shaped *Kripke structures* whose states (nodes) are labelled with XML tags. Finally, the query evaluation problem for XPath can be reinterpreted as a *model checking problem* for multi-modal logic. XPath queries have the form $q[\alpha]$, where q is called *path* and α is called *filter*. A path is a sequence of axis steps and it is interpreted according to the following *query semantics*: it retrieves those nodes that are reachable from the current one through the axes used in q. A filter is similar to a *modal logic formula* and it is interpreted according to the standard *modal logic semantics*: it selects the current node if it satisfies the filter α. These two semantics are orthogonal, and they are mixed in the semantics of XPath. This orthogonality is the cause of the exponential complexity of a naive implementation of the semantics of XPath [5]. There exist two main strategies to avoid this exponential behavior. The first translates the path q into a modal logic formula α_q and then applies modal logic semantics and methods. The second reduces the filter α to a query q_α and then applies query semantics and tree-search

algorithms. However, it is not clear which of the two contexts, either the modal logic context or the database one, is more appropriate for the implementation of efficient evaluation algorithms for XPath.

In this paper, we neatly isolate two evaluation strategies for the navigational fragment of XPath known as Core XPath [5]. The first algorithm, that we called BottomXPath, translates a Core XPath query into a modal logic formula and then applies model checking procedures in order to retrieve the answer set of the original query. This algorithm works *bottom-up* with respect to the parse tree of the input query: it processes the sub-queries of the input query from the leaves of the parse tree up to the tree root. The second algorithm, that we called TopXPath, replaces the filters present in the input query with query paths and then applies a node retrieval procedure in order to compute the answer set of the original query. This procedure works *top-down* with respect to the parse tree of the input query: it processes the sub-queries of the input query from the root of the parse tree down to the leaves of the tree. The XQuery formal semantics requires that the result of an XPath expression is a sequence of document nodes that is document sorted and duplicate free. The *document order* corresponds to the total order of nodes given by a preorder visit of the nodes of an XML tree. With reference to the time when the sorting of the XPath expression results is performed, we specify two versions of the top-down algorithm. The first version, that we named TopXPath1, does not care about the order of the nodes in the intermediate node sequences and it document sorts the final result only. The second version, that we named TopXPath2, maintains document sorted all the intermediate node sequences and hence it does not need to sort the final result. More importantly, it takes advantage of the hypothesis that the intermediate results are document sorted in order to speed-up the XPath axis evaluation. This happens by pruning the intermediate results as much as possible before starting a new step evaluation.

A theoretical analysis of the worst-case asymptotic computational complexity of the outlined algorithms does not help in evaluating their real-life performance: all the procedures run in *linear time* with respect to the product of the size of the XML tree and the length of the query. In order to better understand the computational differences between the proposed strategies, which is our main goal in this paper, we performed an *experimental analysis*. We implemented the algorithms in standard C language and we used a fragment of the XPath benchmark XPathMark [6] to assess the empirical complexity of the discussed strategies.

This paper relates to previous work as follows. Gottlob et al. [5] propose a bottom-up polynomial-time XPath processing algorithm for full XPath, which runs in linear time for Core XPath. Moreover, they discuss a general mechanism for translating the bottom-up algorithm into a top-down one. The relation between XPath query evaluation and model checking has been investigated in [7, 8], where the authors embed a fragment of Core XPath into temporal logic and use an existing model checker to solve the query evaluation problem. The idea of maintaining document sorted the intermediate answers in order to speed-up

the axis evaluation has been proposed in [9], a work that is mostly inspired by the results in [10]. However, none of these papers has empirically compared the different strategies for XPath query evaluation. This is our main task in this work. Our bottom-up procedure BottomXPath borrows from ideas in [7], while our first top-down algorithm TopXPath1 has been inspired by the work in [5]. Finally, our second top-down algorithm TopXPath2 is an simplified version of the procedure proposed in [9].

The rest of the paper is as follows. In Section 2 we introduce XPath and we relate it to modal logic. In Sections 3 and 4 we describe the bottom-up and top-down evaluation strategies, respectively. In Section 5 we perform the experimental analysis of the proposed algorithms and in Section 6 we show how to rewrite the standard bottom-up model checking algorithm for modal logic into a more efficient top-down procedure. Finally, we sum-up in Section 7.

2 XML Path languages

In this section we introduce three formalisms: the Core XPath query language, the Core XPath calculus, and the Core XPath modal logic. The *Core XPath query language* [5] is defined on a tag set Σ including the special symbol $*$. Let χ be the following set of Core XPath axes: child, parent, descendant, ancestor, descendant-or-self, ancestor-or-self, following-sibling, preceding-sibling, following, preceding, and self. We say that child is the inverse of parent and viceversa, descendant is the inverse of ancestor and viceversa, and so on. Notice that self is the inverse of itself. A *Core XPath query* is defined by the query clause of the following grammar (where $\texttt{axis} \in \chi$ and $\texttt{a} \in \Sigma$):

$$\begin{array}{ll} \texttt{query} & = /\texttt{path} \\ \texttt{path} & = \texttt{step} \mid \texttt{step/path} \\ \texttt{step} & = \texttt{axis} :: \texttt{a} \mid \texttt{axis} :: \texttt{a[filter]} \\ \texttt{filter} & = \texttt{path} \mid \texttt{filter and filter} \mid \texttt{filter or filter} \mid \texttt{not(filter)} \end{array}$$

The *Core XPath calculus* is a calculus of path expressions without filters. A *Core XPath expression* is defined by the exp clause of the following grammar (where $\texttt{axis} \in \chi$ and $\texttt{a} \in \Sigma$):

$$\begin{array}{ll} \texttt{exp} & = /\texttt{path} \mid \texttt{path} \mid \texttt{exp} \cap \texttt{exp} \mid \texttt{exp} \cup \texttt{exp} \mid \overline{\texttt{exp}} \mid \texttt{exp/exp} \\ \texttt{path} & = \texttt{step} \mid \texttt{step/path} \\ \texttt{step} & = \texttt{axis} :: \texttt{a} \end{array}$$

Both Core XPath queries and Core XPath expressions are interpreted over XML trees representing XML documents. Since in the present work we are only interested in the navigational power of XPath, we assume that the XML documents we work with do not contain attributes, namespaces, processing instructions, comments, and parsed character data. An XML tree is a rooted sibling-ordered tree $T = (N, R_{\downarrow}, R_{\rightarrow}, L)$, where:

- N is a set of nodes. We denote by *root* the root node of the tree. A tree node represents an element in the XML document;

- $R_{\downarrow}$ is a binary relation on N such that $(x, y) \in R_{\downarrow}$ iff y is a child of x;
- $R_{\rightarrow}$ is a (functional) binary relation on N such that $(x, y) \in R_{\rightarrow}$ iff y is the right sibling of x;
- L is a function from Σ to the power set of N such that, for $a \in \Sigma \setminus \{*\}$, $L(a)$ is the set of nodes that are labelled with tag a, and $L(*) = N$.

Given an XML tree T, a query q in the Core XPath language, and a context set $C \subseteq N$, the semantics of the Core XPath query language is given by a function $\sigma(T, q, C)$ returning a subset of N. The semantic function σ is inductively defined as follows:

$\sigma(T, \texttt{axis} :: \texttt{a}, C) =$
$\{y \in N \mid \exists x \in C.\ (x, y) \in R^T_{\texttt{axis}} \text{ and } y \in L(\texttt{a})\}$

$\sigma(T, \texttt{axis} :: \texttt{a}[\texttt{filter}], C) =$
$\{y \in N \mid y \in \sigma(T, \texttt{axis} :: \texttt{a}, C) \text{ and } \text{exists}(T, \texttt{filter}, y)\}$

$\sigma(T, \texttt{path}_1/\texttt{path}_2, C) = \sigma(T, \texttt{path}_2, \sigma(T, \texttt{path}_1, C))$

$\sigma(T, /\texttt{path}, C) = \sigma(T, \texttt{path}, \{root\})$

where the predicate exists is as follows:

$\text{exists}(T, \texttt{path}, y)$ iff $\sigma(T, \texttt{path}, \{y\}) \neq \emptyset$

$\text{exists}(T, \texttt{filter}_1 \texttt{ and } \texttt{filter}_2, y)$ iff
both $\text{exists}(T, \texttt{filter}_1, y)$ and $\text{exists}(T, \texttt{filter}_2, y)$ hold

$\text{exists}(T, \texttt{filter}_1 \texttt{ or } \texttt{filter}_2, y)$ iff
either $\text{exists}(T, \texttt{filter}_1, y)$ or $\text{exists}(T, \texttt{filter}_2, y)$ holds

$\text{exists}(T, \texttt{not}(\texttt{filter}), y)$ iff
$\text{exists}(T, \texttt{filter}, y)$ does not hold

The relation $R^T_{\texttt{axis}}$ is a binary relation on N corresponding to the specified axis. For instance, $R^T_{\texttt{child}} = R_{\downarrow}$ and $R^T_{\texttt{descendant}} = (R_{\downarrow})^+$. The *answer set* of the query q with respect to the tree T is equal to $\sigma(T, q, N)$.

Given an XML tree T, an expression e in the Core XPath calculus, and a context set $C \subseteq N$, the semantics of the Core XPath calculus is given by a function $\lambda(T, e, C)$ returning a subset of N, which is defined in terms of the semantics function σ as follows:

$$
\begin{array}{ll}
\lambda(T, /\texttt{path}, C) & = \sigma(T, \texttt{path}, \{root\}) \\
\lambda(T, \texttt{path}, C) & = \sigma(T, \texttt{path}, C) \\
\lambda(T, \texttt{exp}_1 \cap \texttt{exp}_2, C) & = \lambda(T, \texttt{exp}_1, C) \cap \lambda(T, \texttt{exp}_2, C) \\
\lambda(T, \texttt{exp}_1 \cup \texttt{exp}_2, C) & = \lambda(T, \texttt{exp}_1, C) \cup \lambda(T, \texttt{exp}_2, C) \\
\lambda(T, \overline{\texttt{exp}}, C) & = N \setminus \lambda(T, \texttt{exp}, C) \\
\lambda(T, \texttt{exp}_1/\texttt{exp}_2, C) & = \lambda(T, \texttt{exp}_2, \lambda(T, \texttt{exp}_1, C))
\end{array}
$$

The *result set* of an expression e with respect to the tree T is equal to $\lambda(T, e, N)$. It is worth mentioning that the defined semantics for the intersection, union, and complementation operators in the Core XPath calculus differs from the semantics of the XPath 2 [11] operators `intersect`, `union`, and `except`, respectively. For instance, the query:

```
/descendant-or-self::*/(self::* ∩ child::*)
```

selects all nodes except the root, while the result of the query:

```
/descendant-or-self::*/(self::* intersect child::*)
```

is always empty.

Finally, the *Core XPath modal logic* is an instance of multi-modal logic in which there is one modality for each axis in XPath. It is defined over a set of labels Σ including the special symbols denoted by `*` and `root`. A model for Core XPath modal logic is a relational structure corresponding to an XML tree. More precisely, given an XML tree $T = (N, R_{\downarrow}, R_{\rightarrow}, L)$, the corresponding model for Core XPath logic is $M_T = (N, \{R^T_{\texttt{axis}}\}_{\texttt{axis}\in\chi}, L)$, where $L(\texttt{root}) = \{root\}$ and $L(*) = N$. The semantics $\models$ of modal formulas is as usual. We define the *truth set* of a modal formula α with respect to a model M as the set of all states $x \in N$ such that $M, x \models \alpha$.

In Section 3 we will show how to translate Core XPath queries into Core XPath modal formulas, while in Section 4 we will embed Core XPath queries into Core XPath expressions. Finally, in Section 6 we give a translation from Core XPath modal formulas to Core XPath expressions.

3 A bottom-up evaluation strategy

In this section we give an efficient bottom-up algorithm, called BottomXPath, to evaluate a Core XPath query. The algorithm is based on a technique that in the logic context is known as model checking [12]. The *model checking problem* is the following question: given a model M and a formula α, retrieve the truth set of α with respect to M. A *model checker* is an algorithm that solves the model checking problem.

We start by embedding Core XPath queries into Core XPath modal formulas. A *filter expression* in XPath is defined by the filter clause of the Core XPath grammar given in Section 2. We define a translation ω from XPath filter expressions into Core XPath modal formulas as follows (if `filter` is empty in the first two clauses below, then the corresponding conjunct is missing):

$$
\begin{aligned}
\omega(\texttt{axis :: a[filter]}) &= \langle\texttt{axis}\rangle(\texttt{a} \wedge \omega(\texttt{filter})) \\
\omega(\texttt{axis :: a[filter]/path}) &= \langle\texttt{axis}\rangle(\texttt{a} \wedge \omega(\texttt{filter}) \wedge \omega(\texttt{path})) \\
\omega(\texttt{filter}_1\ \texttt{and}\ \texttt{filter}_2) &= \omega(\texttt{filter}_1) \wedge \omega(\texttt{filter}_2) \\
\omega(\texttt{filter}_1\ \texttt{or}\ \texttt{filter}_2) &= \omega(\texttt{filter}_1) \vee \omega(\texttt{filter}_2) \\
\omega(\texttt{not(filter)}) &= \neg\omega(\texttt{filter})
\end{aligned}
$$

We now define a translation τ from Core XPath queries into Core XPath modal formulas as follows (if `filter` is empty in the clauses below, then the corresponding conjunct is missing):

$$\begin{aligned}\tau(/\texttt{axis} :: \texttt{a}[\texttt{filter}]) &= \texttt{a} \wedge \omega(\texttt{filter}) \wedge \langle \texttt{axis}^{-1} \rangle \, \texttt{root} \\ \tau(\texttt{path}/\texttt{axis} :: \texttt{a}[\texttt{filter}]) &= \texttt{a} \wedge \omega(\texttt{filter}) \wedge \langle \texttt{axis}^{-1} \rangle \, \tau(\texttt{path})\end{aligned}$$

where $\texttt{axis}^{-1}$ is the inverse of `axis`. Notice that the length of $\tau(q)$ is *linear* in the length of q and $\tau(q)$ can be computed in linear time. We have the following:

Theorem 1. *Let q be a Core XPath query and T be an XML tree. Then, the answer set of q with respect to T is the truth set of $\tau(q)$ with respect to the corresponding model M_T.*

By virtue of Theorem 1, the answer set for a Core XPath query equals to the truth set for the corresponding Core XPath formula. Hence, we can solve the query evaluation problem in terms of the model checking problem by using a model checker as a query processor. BottomXPath is a bottom-up model checker for Core XPath modal logic. It inputs an XML tree T (and not a multi-modal model) and a Core XPath formula α and processes sub-formulas of α in increasing length order. The algorithm is similar to a model checker for the temporal logic CTL (see, e.g., [12]); instead of CTL temporal operators, BottomXPath checks XPath axes. The tree T is represented as follows: each node is an object composed of a field *pre* containing the order of the node in a preorder visit of the tree, a field p containing a pointer to the parent of the node, or NIL if the node is the root, a field c containing a pointer to the first child of the node, or NIL if the node is a leaf, a field r containing a pointer to the right sibling of the node, or NIL if the node is the last sibling, a field l containing a pointer to the left sibling of the node, or NIL if the node is the first sibling, and a field *tag* containing the tag of the XML element that the node represents.

BottomXPath uses a subprocedure EvalAxis. The latter inputs a tree T, and axis `axis` and a formula β. For each node $x \in N$, the procedure EvalAxis labels x with $\langle \texttt{axis} \rangle \beta$ if, and only if, there exists a node $y \in N$ reachable from x trough the relation induced by `axis` such that y is labelled with β. EvalAxis takes advantage of a Boolean matrix A, where rows represent formulas and columns represent nodes, in order to label nodes with formulas that are true at them. Finally, EvalAxis uses the auxiliary procedure LabelDescendants in order to label the descendant nodes of a given node with a given formula. For space reasons, we only show the implementation of `descendant` and `ancestor` axes of EvalAxis.

```
1: EvalAxis(T, axis, β)
2: case
3: • ...
4: • axis = descendant
5: for all x ∈ N do
6:    if A(β, x) = 1 then
```

```
      y ← p[x]
      while y ≠ NIL and A(⟨descendant⟩β, y) = 0 do
        A(⟨descendant⟩β, y) ← 1
        y ← p[y]
      end while
    end if
  end for
• axis = ancestor
  for all x ∈ N do
    if A(β, x) = 1 and A(⟨ancestor⟩β, x) = 0 then
      y ← c[x]
      while y ≠ NIL do
        LabelDescendant(⟨ancestor⟩β, y)
        y ← r[y]
      end while
    end if
  end for
end case
```

Let n be the size of the input tree T and k the length of the input formula α. The procedure EvalAxis runs in $O(n)$ and hence the cost of BottomXPath amounts to $O(k \cdot n)$. The whole bottom-up evaluation algorithm for Core XPath is as follows:

1. translate q into $\tau(q)$;
2. run BottomXPath on T and $\tau(q)$;
3. sort, in document order, the result of BottomXPath.

The complexity of the translation step is $O(k)$ and the call to BottomXPath costs $O(k \cdot n)$. Since nodes are integers from 1 to n, we can sort the result with a linear-time sorting algorithm like CountingSort. Hence, the overall worst-case complexity is $O(k \cdot n)$.

4 A top-down evaluation strategy

In this section we give two efficient top-down algorithms, called TopXPath1 and TopXPath2, to evaluate Core XPath queries. Both the algorithms first replace the filters present in the input query and then apply a node retrieval procedure in order to compute the answer set of the original query.

We first show how to get rid of filters. The following translation inputs a filter expression f in the Core XPath query language and returns an expression in the Core XPath calculus. It is computed in two steps. First, all the sub-filters in f of the form [`filter`]/`path` are rewritten as [`filter and path`]. The resulting filter

is then processed by the following translation κ:

$$\begin{array}{ll}
\kappa(\texttt{axis :: a}) & = \texttt{self :: a/axis}^{-1}\texttt{ :: *} \\
\kappa(\texttt{axis :: a[filter]}) & = \kappa(\texttt{filter})/\kappa(\texttt{axis :: a}) \\
\kappa(\texttt{step/path}) & = \kappa(\texttt{path})/\kappa(\texttt{step}) \\
\kappa(\texttt{filter}_1\texttt{ and filter}_2) & = \kappa(\texttt{filter}_1) \cap \kappa(\texttt{filter}_2) \\
\kappa(\texttt{filter}_1\texttt{ or filter}_2) & = \underline{\kappa(\texttt{filter}_1)} \cup \kappa(\texttt{filter}_2) \\
\kappa(\texttt{not(filter)}) & = \overline{\kappa(\texttt{filter})}
\end{array}$$

Let us call ι the above defined 2-step translation. We now define a translation υ from Core XPath queries into Core XPath expressions:

$$\begin{array}{ll}
\upsilon(/\texttt{axis :: a[filter]}) & = /\texttt{axis :: a} \cap \\
 & \quad \texttt{/descendant-or-self::*/}\iota(\texttt{filter}) \\
\upsilon(\texttt{path/axis :: a[filter]}) & = \upsilon(\texttt{path})/\texttt{axis :: a} \cap \\
 & \quad \texttt{/descendant-or-self::*/}\iota(\texttt{filter})
\end{array}$$

Notice that the length of $\upsilon(q)$ is linear in the length of q and $\upsilon(q)$ can be computed in linear time. We have the following:

Theorem 2. *Let q be a Core XPath query and T be an XML tree. Then, the answer set of q with respect to T is the result set of $\upsilon(q)$ with respect to T.*

4.1 A first top-down algorithm

In this section we propose a first top-down strategy, called TopXPath1, to evaluate Core XPath queries. With respect to the tree data structure described in Section 3, we assume here that an additional field called *count* is added to the object representation of each node of the tree. The new field is used to record whether the node has been visited or not during a step evaluation. TopXPath1 does not care about the order of the nodes in the intermediate context sets and it sorts the final result only.

TopXPath1 inputs an XML tree T, a Core XPath expression e, and a context set C. It evaluates the expression e in a top-down fashion as defined by its semantics. It uses a procedure ProcessStep1 to evaluate the XPath axis steps. In particular, the procedure ProcessStep1(T, `axis`, `a`, C) elaborates the step `axis :: a` on the tree T with context set C, according to the XPath semantics. In order avoid walking on the same node twice, the procedure checks the *count* field of the node's object. This field is initialized to 0 for each node when TopXPath1 starts. The global variable k is also initialized to 0 and it is incremented by one at each step evaluation performed with ProcessStep1. When a node is visited during a step evaluation, its *count* field is assigned to the value contained in k. Hence, during the k-th step evaluation, all nodes that has been already visited in that step evaluation have their count field set to k, while the count field of the unexplored nodes is less than k. This method avoids the costly resetting of the count field at each step evaluation. Finally, ProcessStep1 uses an auxiliary procedure RetrieveDescendants to retrieve the descendant nodes of a given node

that are labelled with a given tag. We show the implementation of `descendant` and `ancestor` axes of ProcessStep1.

```
 1: ProcessStep1(T, axis, a, C)
 2: k ← k + 1
 3: R ← ∅
 4: case
 5: • ...
 6: • axis = descendant
 7: for x ∈ C do
 8:    if count[x] < k then
 9:       count[x] ← k
10:       y ← c[x]
11:       while y ≠ NIL do
12:          R ← R ∪ RetrieveDescendants(y, a)
13:          y ← r[y]
14:       end while
15:    end if
16: end for
17: • axis = ancestor
18: for x ∈ C do
19:    y ← p[x]
20:    while y ≠ NIL and count[y] < k do
21:       count[y] ← k
22:       if a = ∗ or tag[y] = a then
23:          R ← R ∪ {y}
24:       end if
25:       y ← p[y]
26:    end while
27: end for
28: end case
```

Let n be the size of the input tree T and k the length of the input formula α. The procedure ProcessStep1 costs $O(n)$ and hence the evaluation algorithm TopXPath1 runs in $O(k \cdot n)$. The whole top-down evaluation algorithm for Core XPath is as follows:

1. translate q into $\upsilon(q)$;
2. run TopXPath1 on T and $\upsilon(q)$;
3. sort, in document order, the result of TopXPath1.

The complexity of the translation step is $O(k)$ and the call to TopXPath1 costs $O(k \cdot n)$. The sorting can be performed in linear time. Hence, the overall worst-case complexity is $O(k \cdot n)$.

4.2 A second top-down algorithm

In this section we propose a second top-down strategy, called TopXPath2, to evaluate Core XPath queries. With respect to the tree data structure described in Section 3, we assume here that two additional fields are added to the object representation of each node of the tree: a field called *count* that, as in TopXPath1, is used to record whether the node has been visited or not during a step evaluation, and a field called *post* containing the order of the node in a postorder visit of the tree.

TopXPath2 evaluates a Core XPath expression in a top-down fashion as done by TopXPath1. It uses a sub-procedure ProcessStep2 in order to process the XPath axis steps, which in turn uses an auxiliary procedure Descendant to retrieve the descendant nodes of a given node that are labelled with a given tag. Moreover, it uses the following auxiliary list procedures, where C and L are double-linked lists and x is a node: NewList(), that initializes a new list, DelFirst(C), that deletes and returns the first element of C, DelLast(C), that deletes and returns the last element of C, AddAfter(C, x), that appends x to C, AddListAfter(C, L), that appends L to C, AddBefore(C, x), that adds x in front of C, AddListBefore(C, L), that adds L in front of C, First(C) that returns the first element of C, Last(C) that returns the last element of C. All these procedures can be implemented in constant time. TopXPath2 differs from TopXPath1 since it maintains document sorted the intermediate context sets. Moreover, it exploits the sorted contexts to speed-up the XPath axis evaluation by pruning the context sets as much as possible before starting each step evaluation. By maintaining both the preorder and the postorder ranks for each node, TopXPath2 implicitly represents an XML tree as a bi-dimensional plane, called the *pre/post plane* in [10]. Each node x is encoded by the point $(pre(x), post(x))$. A nice feature of this encoding is that, for each node x, the top-right (respectively, bottom-left) quadrant of x contains all the following (respectively, preceding) nodes of x, and the bottom-right (respectively, top-left) quadrant of x contains all the descendant (respectively, ancestor) nodes of x. Hence, given two arbitrary nodes x and y, we can check in constant time the relative position of y with respect to x. As an example, consider the cases of `following` and `preceding` axes. By exploiting the pre/post plane properties, the context set can always be reduced to a singleton (see the code below). Finally, TopXpath2 takes advantage, when necessary, of the counting technique described in Section 4.1 to avoid the exploration of the same tree zones twice. For space reasons, we only show the implementation of `following` and `preceding` axes of ProcessStep2. Notice that, if the input context set is document sorted, then the results of `following` and `preceding` axes are document sorted as well.

1: ProcessStep2(T, `axis`, `a`, C)
2: $k \leftarrow k + 1$
3: $R \leftarrow \emptyset$
4: **case**
5: • ...

```
• axis = following
L ← NewList()
if C ≠ ∅ then
  x ← DelFirst(C)
  while post[First(C)] < post[x] do
    x ← DelFirst(C)
  end while
  while x ≠ NIL do
    y ← r[x]
    while y ≠ NIL do
      AddListAfter(L, Descendants(y, a))
      y ← r[y]
    end while
    x ← p[x]
  end while
end if
return L
• axis = preceding
L ← NewList()
x ← Last(C)
while x ≠ NIL do
  M ← NewList()
  y ← l[x]
  while y ≠ NIL do
    AddListAfter(M, Descendants(y, a))
    y ← l[y]
  end while
  AddListBefore(L, M)
  x ← p[x]
end while
return L
end case
```

Let n be the size of the input tree T and k the length of the input formula α. ProcessStep2 runs in $O(n)$ and TopXPath2 in $O(k \cdot n)$. In this case, the whole top-down evaluation algorithm for Core XPath is as follows:

1. translate q into $\upsilon(q)$;
2. run TopXPath2 on T and $\upsilon(q)$.

The complexity of the translation step is $O(k)$ and the call to TopXPath2 costs $O(k \cdot n)$. Since TopXPath2 maintains sorted the intermediate context sets, the result of TopXPath2 is already sorted. Hence, the overall worst-case complexity is $O(k \cdot n)$.

5 Experimental analysis

The three proposed algorithms have the same asymptotic worst-case complexity. In order to better understand the computational differences between the proposed strategies, we performed an *experimental analysis*. We implemented the algorithms in standard C language and we used a fragment of the XPath benchmark XPathMark [6] to assess the empirical complexity of the discussed strategies. In this section, we briefly report about this analysis. The source code (released under the GNU General Public License), the executable programs (for Gnu/Linux systems), and additional experimental data and plots (including a comparison with XQuery processor Saxon [13], which is outside of the scope of this paper) are available at the website `http://www.zimuel.it/xpath`.

Our experiments were run on an AMD Sempron 1.7 GHz, with 1 GB RAM, running Debian Gnu/Linux version 2.6.10. All the times are response CPU times in seconds. We ran tests using a variety of XML documents and XPath queries. The XML documents were generated using the XML benchmarking program XMark [14]. We used a document series of 11 XML documents corresponding to the following sizes in MB: 0.116, 0.212, 0.468, 0.909, 1.891, 3.751, 7.303, 15.044, 29.887, 59.489, 116.517. As for the benchmark queries, we took advantage of a fragment of the XPath benchmark XPathMark [6]. The benchmark queries we used are the following (notice that Q7, Q8, Q9, and Q10 are new queries not belonging to XPathMark):

Q1 *The keywords in annotations of closed auctions*

```
/child::site/child::closed_auctions/child::closed_auction
/child::annotation/child::description/child::parlist
/child::listitem/child::text/child::keyword
```

Q2 *All the keywords*

```
/descendant::keyword
```

Q3 *The keywords in a paragraph item*

```
/descendant-or-self::listitem/descendant-or-self::keyword
```

Q4 *The (either North or South) American items*

```
/child::site/child::regions/child::*/child::item
```

```
[parent::namerica or parent::samerica]
```

Q5 *The paragraph items containing a keyword*

```
/descendant::keyword/ancestor::listitem
```

Q6 *The mail containing a keyword*

```
/descendant::keyword/ancestor-or-self::mail
```

Q7 *The last bidder of all open auctions*

```
/child::site/child::open_auctions/child::open_auction
```

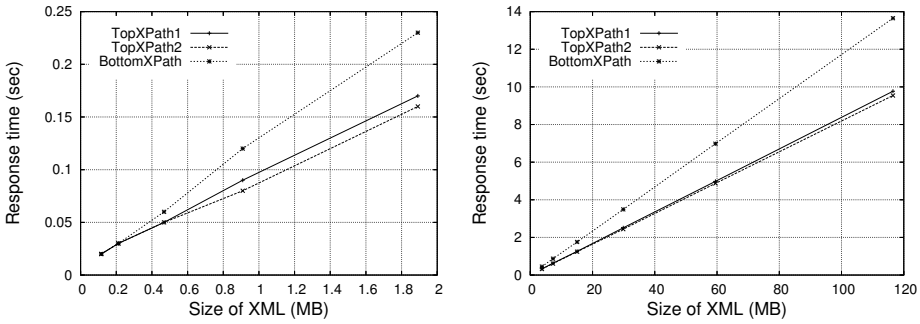

Fig. 1. Query response time (averaged over the benchmark set)

```
/child::bidder[not(following-sibling::bidder)]
```

Q8 *The first bidder of all open auctions*

```
/child::site/child::open_auctions/child::open_auction
/child::bidder[not(preceding-sibling::bidder)]
```

Q9 *The last item of the document*

```
/child::site/child::regions/child::*
/child::item[not(following::item)]
```

Q10 *The first item of the document*

```
/child::site/child::regions/child::*
/child::item[not(preceding::item)]
```

Q11 *People having an address and either a phone or a homepage*

```
/child::site/child::people/child::person
[child::address and (child::phone or child::homepage)]
```

Q12 *People having no homepage*

```
/child::site/child::people/child::person[not(child::homepage)]
```

In order to perform the evaluation, we applied the benchmarking methodology described in [6]. In particular, we used the following measures:

- given a query q and a document d, the *query response time* is the time taken by an algorithm to give the answer for the query q on the document d including all the phases of the elaboration (e.g., parsing of the document, processing of the query, serialization of the results).
- Given a query q and a document d, the *query response speed* is defined as the size of the document d divided by the response time for query q and document d. The measure unit is, for instance, MB/sec.
- Given a query q and two documents d_1 and d_2, where the size of d_2 is bigger than the size of d_1, the *data scalability factor* is defined as v_1/v_2, where v_1

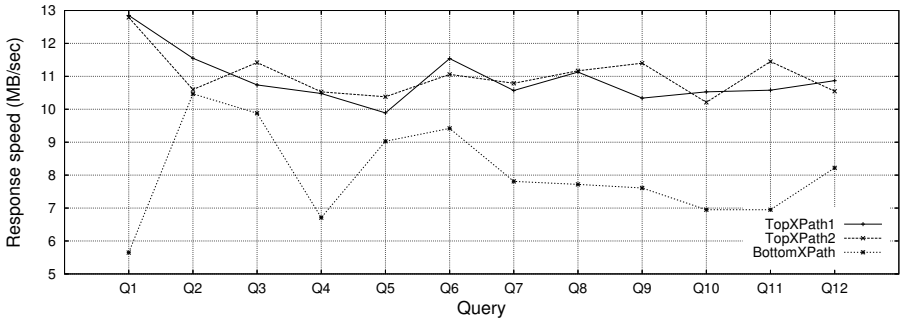

Fig. 2. Query response speed (averaged over the document series)

is the query response speed of q on d_1 and v_2 is the query response speed of q on d_2.

We say that the data scalability is *sub-linear* whenever the data scalability factor is lower than 1, it is *super-linear* whenever the data scalability factor is higher then 1, and it is *linear* whenever the data scalability factor is equal to 1. Usually, sub-linear and linear scalability indicate that the query processor performs well when the size of the database is increased.

Figure 1 depicts, for each document in the series, the average query response time for the entire benchmark (left plot for small files, right plot for bigger ones), while Figure 2 illustrates, for each query in the benchmark, the average query response speed for the entire document series. Finally, Figure 3 plots, for each document in the series, the average data scalability factors for the entire benchmark.

In the following, we comment the outcomes of our experimental evaluation:

- The response times of the two top-down strategies are very close, with TopXPath2 slightly faster than TopXPath1. This tells us that the approach of maintaining the context sequences document sorted at any time does not pay off in terms of response time.
- The top-down strategy is more efficient than the bottom-up one (about 30% faster, and the difference increases as the size of the data increases). The gap is bigger in the case of queries that do not need to explore big portions of the document tree in order to compute the query answer (like Q1 and Q4), while the response times of the two strategies are similar in the case of queries that need to visit the entire document (like Q2) . This phenomenon can be explained as follows: the bottom-up algorithm visits the entire document tree for each sub-query of the main query, while the top-down procedure explores only the tree zones that are relevant for the evaluation of the query.
- All the three XPath processors scale-up linearly with respect to the size of the XML data. This confirms the linear-time complexity of the implemented algorithms.

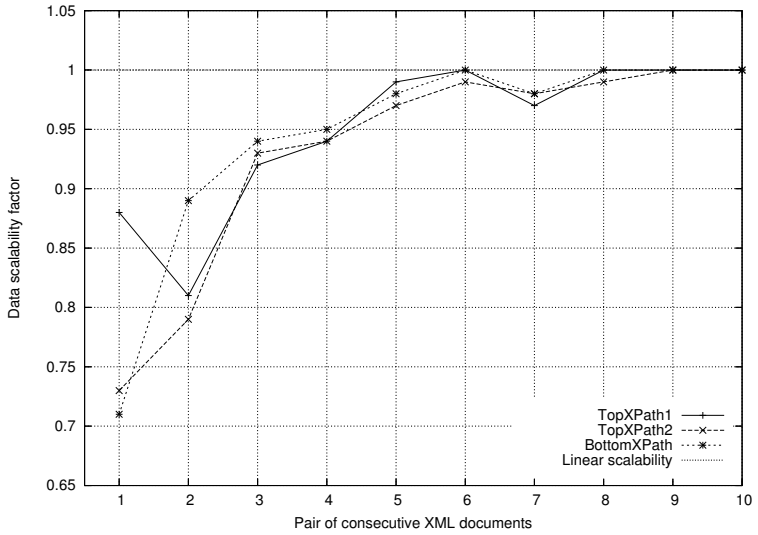

Fig. 3. Data scalability factors (averaged over the benchmark set)

6 Back to modal logic

What do modal logicians get out of this paper at the end of the day? Consider the following translation from Core XPath modal formulas to Core XPath expressions (where `a` is an atomic proposition different from `root`)[3]:

$$\begin{array}{ll}
\chi(\texttt{root}) & = \overline{\texttt{child} :: *} \\
\chi(\texttt{a}) & = /\texttt{descendant} - \texttt{or} - \texttt{self} :: \texttt{a} \\
\chi(\langle \texttt{axis} \rangle \alpha) & = \chi(\alpha)/\texttt{axis}^{-1} :: * \\
\chi(\alpha \wedge \beta) & = \chi(\alpha) \cap \chi(\beta) \\
\chi(\alpha \vee \beta) & = \chi(\alpha) \cup \chi(\beta) \\
\chi(\neg\alpha) & = \overline{\chi(\alpha)}
\end{array}$$

We have the following:

Theorem 3. *Let α be a Core XPath modal formula and T be an XML tree. Then, the truth set of α with respect to the corresponding model M_T is the result set of $\chi(\alpha)$ with respect to T.*

The above theorem gives us the following *top-down* algorithm to solve the model checking problem for Core XPath modal logic: translate a modal formula α into the corresponding expression $\chi(\alpha)$ and then use a top-down strategy as described in Section 4 to retrieve the result set of $\chi(\alpha)$. This paper empirically

[3] A translation from Core XPath modal formulas to Core XPath queries with filters is given in [15].

proves that a top-down evaluation strategy is less expensive than a bottom-up one, because it browses only the relevant parts of the underlying model and not more. However, model checking algorithms for modal and temporal logics typically are bottom-up procedures [12]. The top-down query evaluation strategies described in Section 4 can be turned into an efficient top-down model checking procedure for Core XPath modal logic.

7 Conclusion

We implemented three evaluation strategies for the navigational fragment of XPath and we benchmarked the resulting XPath processors using a fragment of XPathMark, a recently proposed benchmark for XPath. The main outcomes of our investigation are (i) a top-down evaluation approach is faster than a bottom-up one, and (ii) the pre/post plane optimizations for XPath query evaluation are as efficient as tree-search algorithms over the tree modeling the XML document. We also gave a top-down model checker for Core XPath modal logic which empirically performs better than the standard bottom-up one.

It is worth pointing out that a bottom-up strategy outputs much more information than a top-down strategy. In particular, the bottom-up model checking-based procedure computes the answer set for *each* sub-query of the input query, while the top-down routine retrieves only those nodes belonging to the answer of the input query. This feature of the bottom-up approach may in fact become a benefit whenever the answer set for the sub-queries of the input query is relevant. Consider for instance a query processor that is queried many times possibly by different users. It is not unlikely that similar queries are posed at different times. In such a case, a bottom-up strategy may easily reuse the results computed for common sub-queries (in a dynamic programming fashion), while a top-down strategy must re-compute the result for each new query from scratch. As a future work, we would like to compare the performace of the bottom-up and top-down approaches in a multi-query environment.

Acknowledgments

This research was supported by the Netherlands Organization for Scientific Research (NWO) under project number 612.000.207.

References

1. World Wide Web Consortium: Extensible Markup Language (XML). `http://www.w3.org/XML` (1998)
2. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann Publishers (2000)
3. World Wide Web Consortium: XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath` (1999)

4. World Wide Web Consortium: XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery (2005)
5. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. ACM Transactions on Database Systems (TODS) **30**(2) (2005) 444–491
6. Franceschet, M.: XPathMark: an XPath benchmark for the XMark generated data. In: Proceedings of the International XML Database Symposium (XSym). Volume 3671 of LNCS. (2005) 129–143 http://www.science.uva.nl/~francesc/xpathmark.
7. Afanasiev, L., Franceschet, M., de Rijke, M., Marx, M.: CTL model checking for processing simple XPath queries. In: Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME), IEEE Computer Society Press (2004) 117–124
8. Hartel, P.: A trace semantics for positive Core XPath. In: Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME), IEEE Computer Society Press (2005) 103–112
9. Hidders, J., Michiels, P.: Efficient XPath axis evaluation for DOM data structures. In: Workshop on Programming Language Technologies for XML (PLAN-X). (2004) 54–63
10. Grust, T., van Keulen, M., Teubner, J.: Staircase join: Teach a relational DBMS to watch its (axis) steps. In: Proceedings of International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann Publishers (2003) 524–525
11. World Wide Web Consortium: XML Path Language (XPath) Version 2.0. http://www.w3.org/TR/xpath20 (2005)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
13. Kay, M.H.: Saxon. An XSLT and XQuery processor. http://saxon.sourceforge.net (2005)
14. Schmidt, A.R., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In: Proceedings of the International Conference on Very Large Data Bases (VLDB). (2002) 974–985 http://monetdb.cwi.nl/xml/.
15. Marx, M.: XPath with conditional axis relations. In: Proceedings of the International Conference on Extending Database Technology. Volume 2992 of LNCS. (2004) 477–494