Index-Trees for Descendant Tree Queries in the Comparison Model

Jérémy Barbay

University of British Columbia
jeremy@cs.ubc.ca

[1]

Abstract. Considering indexes and algorithms to answer XPath queries over XML data, we propose an index structure and a related algorithm, both adapted to the comparison model, where elements can be accessed non-sequentially. The indexing scheme uses classical labelling techniques, but structurally represents the ancestor-descendant relationships of nodes of each type, in order to allow exponential searches. The algorithm performs XPath location steps along the `descendant` axis, and it generates few intermediate results. The complexity of the algorithm is proved worst-case optimal in an adaptive comparison model where the index is given, and where the instances are grouped by the number of comparisons needed to check their answer.
Keywords: XPath location steps; holistic algorithm; adaptive analysis.

1 Introduction

1.1 Context

XML is a rapidly emerging format for exchanging data on the web. It standardizes tree structures, so that general tools can be developed and used for the many distinct applications adopting this standard. Among those tools, search engines are prominent, and are strongly based on path navigation as defined by XPath.

XPath [7, 9] is a language for addressing nodes of an XML document by their positions in its tree structure. Each node is specified by the path from the root of the document to it, the path being incrementally described by a sequence of *location steps*. The simplest queries describe a chain of node tests in specified relations, and more complex queries describe a tree structure of node tests. For instance, the expression `//hidden//figure` corresponds to the set of `figure` nodes, descendant of `hidden` nodes, where `//` stands for the `descendant` axis. The query `//hidden[//section]//figure` has a branch, and corresponds to the `figure` nodes of the previous query sharing the `hidden` sub-tree with a `section` node. The XPath 2.0 [7] specifications define two syntaxes (abbreviated, used above, and un-abbreviated), 13 axes, and many expressions for predicates, but only a small subset of them is considered in most applications and studies.

Since data sets can be very large, and can be searched very often, efficiently querying XML data is a major concern. Wise index structures and labelling schemes can be used to simplify the operations [8, 13], in a more or less compact way [1, 2]. For instance, Grust [13] proposes a labelling scheme based on the prefix and postfix rankings of the nodes. He observes that for any node v, this scheme naturally defines a partitioning of the document in 4 parts corresponding to the nodes accessed from v by the four main axis of the location steps of XPath.

As relational database implementations are now quite mature and well optimized, a natural approach to efficiently querying XML documents has been to input XML document in relational databases and to use existing technologies to query them [13, 19]. Bruno *et al.* suggests that "*a limitation of this approach (...) is that intermediate result sizes can get large, even when the input and output sizes are more manageable*" [8], and proposes instead to study holistic algorithms which avoid unnecessarily large intermediate results. This approach supposes a native implementation to treat XML queries, an approach getting more and more interest [14], as opposed to the storage and querying of XML documents in relational databases.

[1] TR-2004-11, Department of Computer Science, University of British Columbia, July 27, 2004

Most studies are restricted to the stream model, where elements of the index are accessed in sequential order only. In practise, this restriction is not always realistic, in particular when the index is implemented in variants of B-trees, as those structures permit to skip parts of the stream. Those structures lead to better performances in practise [8, 13], which somehow contradicts the practicality of the stream model. One motivation to use the stream model seems to be that algorithms restricted to it exhibit pattern accesses with an optimal *miss ratio*. But this measure is not valid to compare the performance of algorithms: an algorithm never accessing the cache will have the worst miss ratio possible, but if it performs sufficiently less accesses than other algorithms, it can still perform better in practise than algorithms with a better miss ratio

When allowed to access data in an arbitrary order, one way to approximate the performance of an algorithm is by counting the number of comparisons it performs: this is the *comparison model*. It is then possible to use *exponential search*, a variant of binary search, often used to search efficiently for several items in a sorted array [3–5, 10, 11]. It permits to search for k ordered elements in a sorted array of size n using $2k \log(1+n/k)$ comparisons. An insufficient analysis would conclude that this algorithm's complexity is linear in n in the worst case, i.e. when $k=n$. A finer analysis distinguishes the two variables k and n and concludes that the algorithm's complexity is $O(k \log(n/k))$. Such finer analyze have been proved useful to analyze adaptive algorithms for sorting problems [12, 16, 18] and for the computation of the intersection of sorted arrays [3, 4, 10, 11], but have not been used yet to analyze the complexity of queries on XML document.

1.2 Contributions

In this context, we make the following contributions:

We consider models where elements of the index can be accessed in any order, e.g. random access memory (RAM). Among those models, we distinguish the *comparison model* where only comparisons are accounted for. This model is motivated by applications where all the information needed is at the same memory level. This model permits in particular to search in sorted arrays by exponential search [6], a variant of binary search. But even in a model allowing arbitrary access to any element, traditional inverted list indexes do not permit to use this algorithm, because the nodes matching a location steps are not always consecutive: this is necessary to reduce the search space by two at each comparison performed.

Hence we propose an index based on a tree structure (index-tree), which structurally represents the ancestor-descendant relationships of nodes, and permits to use exponential search. The impending improvement in performance is similar in a minor way to the one obtained by the use of B^+-trees, in the sense that some node references are not accessed in the index, hence permitting a sub-linear complexity. It is much more important though, as whole parts of the index are skipped, even in the worst case, which permits a sub-linear worst case complexity, and a proved optimality.

We propose a holistic algorithm using index-trees to solve XPath location steps along the `descendant` axis. As Bruno *et al.*'s holistic algorithm [8] for Twig Pattern Matching, the algorithm presented here treats the query as a whole and avoids large intermediate results. It does so by building the answer to the query incrementally, one match at a time, in the document order.

We define an adaptive analysis of XPath location steps along the `descendant` axis, with index-trees. For this analysis we distinguish several variables playing an important role in the relative difficulty of instances: the size k of the query; the number n of nodes in the document; the maximal number h of nodes of same type on a branch of the document; and the minimal number δ of comparisons needed to check the result of the query. This permits to express the worst case complexity $\Theta(\delta hk \log(n/\delta hk))$ of descendant tree queries using index-trees, which is proved by an analysis of the complexity of the algorithm presented, and by the corresponding lower bound.

1.3 Outline

The rest of the paper is organized as follows. We define shortly in Section 2 the labelling scheme; the query language; the index structure; and the exponential search algorithm. In Section 3, we present a new algorithm to solve descendant tree queries, we prove its correctness, and we discuss its memory requirements. We define a measure of the difficulty of an instance, analyze the complexity of the algorithm, and prove its optimality along deterministic algorithms in the comparison model in Section 4. We finally conclude with a summary of results and some perspectives in Section 5.

2 Definitions

2.1 Labelling scheme

Most of the examples of this article are based on a simple XML document describing an article, from which Figure 1 gives the first lines, each line being prefixed by its number. In this sample of the document, the `<article>` tag marks the beginning of the document; the `<title>` and `</title>` tags correspond to the title of an article or of a section; and the `<section>` tag marks the beginning of a section, which ending will be marked by a `</section>` tag. The whole document contain tags of other types, among which the `<figure>` and `</figure>` tags, which correspond to subtrees defining the schemes of the article; and the `<hidden>` and `</hidden>` tags, which correspond to subtrees temporarily excluded from the publication process, for instance because they are still under-work.

```
1:<article>
2: <title>
3:Index-Trees for (...)
4: </title>
5: <section>
6: <title>
7:Introduction
8: </title>
9: <section>
10: <title>
11:Context
12: </title>
13: <para>
(...)
```

Fig. 1. The numbered first lines of the document.

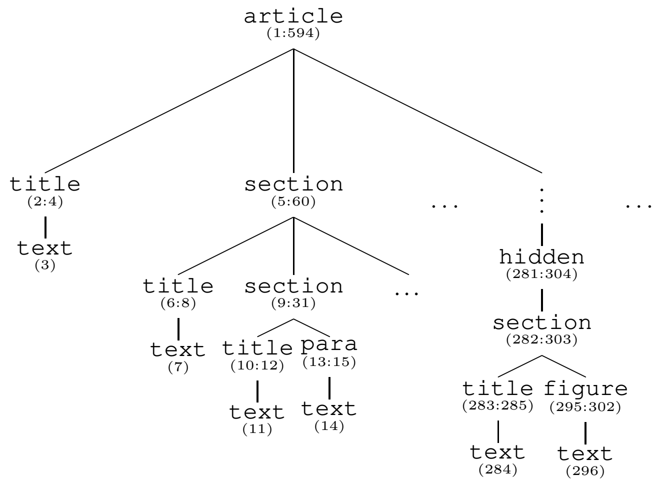


Fig. 2. A subset of the XML tree.

In the index, the nodes of the document are referenced by labels. A wise labelling scheme is mandatory to solve navigation steps efficiently. We borrow the labelling scheme defined by Bruno *et al.* [8], where each node is referenced by an interval of integers. This labelling scheme permits to decide if two nodes are in a descendant relationship with just a few comparisons on their respective labels.

Definition 1 (label). *The* label of an internal node a *of the document is a pair* $s{:}e$ *where* s *and* e *are the respective ranks of the starting and ending tag corresponding to* a *in the document order.*

The label of a leaf b *(e.g. a* `text` *or an attribute node) of the document is the rank* s *(noted* s *or* $s{:}s$*) of* b *in the document order.*

When there is exactly one tag or text per line of the document, the line numbers then correspond to the rank of the corresponding element, making it easier to compute the label of a node. For instance, the first `title` node has label $2{:}4$, because the corresponding `<title>` and `</title>` tags are on lines 2 and 4. Figure 2 presents a part of the tree representation of the same document, each node being annotated with its label.

The interval formed by the labels of a node is a superset of all the labels of its descendant nodes, hence the labels permit to reduce the resolution of each location step to searches in sorted arrays. Given a context node of label $(s{:}e)$, the `descendant` axis corresponds to a search for nodes of label $(s'{:}e')$ such that $s < s' < e$. Location steps along the `ancestor` axis can be solved in exactly the same way using symmetric inequalities, and location steps along `descendant-or-self` and `ancestor-or-self` axes correspond to slight variations of these.

2.2 Query Language

In this paper, we focus on a subset of XPath queries consisting of descendant axis navigation (`//`), branches (`[...]`) and node tests: Figure 3 gives such a query. The answer to this query is the list of `figure` nodes, identified by their labels, such that each node is descendant of a `hidden` node, itself ancestor of a `section` node. The purpose could be for instance to find the figures which have been removed from the publication process together with a whole section.

Q = `//hidden[//section]//figure`

Fig. 3. A simple XPath Query Q

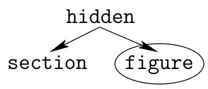


Fig. 4. The query tree of Q

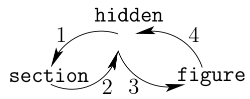


Fig. 5. The iterated tour of Q

Such queries can be represented as trees with a distinguished node, and are called *descendant tree queries*: Figure 4 gives the descendant tree corresponding to the query given Figure 3. There is a node for each tag in the query, and an edge for each navigation step. The nodes are annotated in accordance with the tag of the corresponding node, and each edge corresponds to a location step along the descendant axis. The node specified by the query is circled: the labels matching this node compose the answer to the query.

We will denote by $\texttt{dist}(Q)$ the tag corresponding to the node distinguished by the query; by $\texttt{tour}(Q)$ the iterated tour of the nodes of Q defined by its preorder traversal, and by $|\texttt{tour}(Q)|$ the period of this tour. For instance, the distinguished node of the descendant tree query given in Figure 4 is $\texttt{dist}(Q)$=`figure`; and the tour of this tree is $\texttt{tour}(Q)$=(`hidden, section, hidden, figure, hidden, ...`), of period $|\texttt{tour}(Q)| = 4$, as shown on Figure 5.

2.3 Index-trees

We propose an index based on a tree structure which structurally represents the ancestor-descendant relationships of nodes, and which permits to use exponential search.

Definition 2 (index-tree). *The index-tree corresponding to a given note-type test α for a document D is a tree $I[\alpha]$ containing all the labels of nodes matching α, such that any ancestor-descendant relationship in D corresponds to a parent-child or ancestor-descendant relationship in $I[\alpha]$, and such that the children of each node of an index-tree are consecutive elements in an array.*

The simplest way to encode an index-tree is to create a dynamic tree which for each node lists pointers to the children in a sorted array. Such an encoding has only a constant factor space overhead in comparison with inverted list indexes commonly used for XML documents.

For a node-type α which nodes have no descendants of the same type, the index-tree is simply an inverted list of the labels corresponding to nodes of type α, implemented in an array. In the document of Figure 1, the `figure` and `hidden` nodes are in this case: each index-tree, given Figure 6, consists just of a list of labels in document order.

`hidden`	138:147	209:221	281:304	471:473	503:505	$\cdots$

`figure`	72:77	78:85	87:94	122:129	130:137	139:146	148:155	295:302	319:326	$\cdots$

Fig. 6. A subset of the index-trees for `hidden` and `figure` nodes.

The tree structure of index-trees is apparent only for *recursive* node-types, which nodes can have descendants of same type. In the document of Figure 1, the `section` nodes are in this case: their index-tree, given Figure 7, is a tree structure of arrays of labels, where each array pointed by a label a corresponds to a list of `section` nodes accessible from a by a path containing no `section` node.

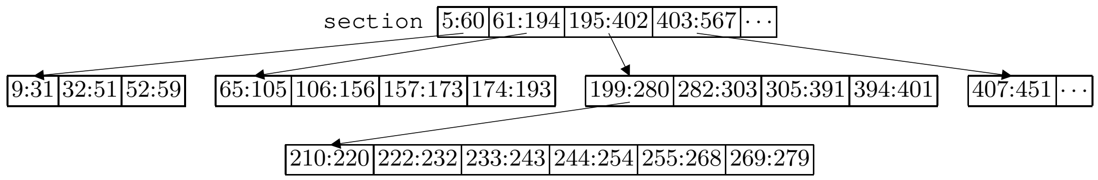


Fig. 7. A subset of the index-tree for `section` nodes.

Additionally, we will denote `parent`(a) the function returning the label of the parent of a in its index-tree, if it has one; `firstChild`(a) the function returning the label of the first child of a in its index-tree; `rightSibling`(a) the function returning the first right sibling of a in its index-tree if there is one (there is at least ∞ if $a \neq \infty$), and returning ∞ if $a = \infty$; `lastRightSibling`(a) the function returning the last right sibling of a, the ∞ label excluded, in its index-tree; and `hasParent`(a), and `hasChild`(a) the boolean functions true if and only if respectively a has a parent or some children in its index-tree.

A minor improvement brought by the use of index-trees is similar to the one obtained by the use of B^+-trees, in the sense that not all labels are accessed in the index, which permits a sub-linear complexity. But the usefulness of index-tree goes far beyond this: as at each level of a subtree, there can be only one ancestor (resp. descendant) b of a given node a, hence b can be searched by an exponential search, with a number of comparisons logarithmic in the size of the array.

2.4 Exponential search

Random Access Memory permits to search in sorted arrays by binary or exponential search. Exponential search permits to look for an element x in a sorted array A of unknown size, starting at position $init$. It returns a value p such that $A[p-1]<x\leq A[p]$, called the *insertion point* of x in A.

This algorithm can be implemented using doubling search and binary search algorithms [4, 10, 17]. Searching for the insertion point p of x in A from position $init$ is then done in two phases: In the first phase, called the *doubling search*, the algorithm performs $i = \lceil \log_2(p - init) \rceil$ comparisons to find the interval $[init + 2^i - 1, init + 2^{i+1} - 1)$, and containing p. To do so, it compares x to the elements occupying positions $(init, init+1, init+3, init+7, \ldots, init + 2^{i+1} - 1)$. In the second phase, a simple binary search on this interval, of size 2^i, permits to find p using i comparisons. The total number of comparisons performed is then $2\lceil \log_2(p - init) \rceil$, but a more sophisticated implementation permits to improve the complexity by a constant factor[6].

For instance, the first `hidden` node of the document, has label $138{:}147$ (see the `hidden` index-tree of Figure 6). An exponential search, implemented by doubling and binary search, for descendants of this `hidden` node in the `figure` index-tree, results in the comparisons shown Figure 8. The first three arrows correspond to the doubling search, the two others correspond to the binary search. The label finally found is $139:146$, and corresponds to a `hidden` node ancestor of the `figure` node given by its label.

This technique applies as well to recursive node-types, when indexed by an index-tree. For instance, the `figure` node of label $122{:}129$ is a descendant of both `section` nodes of label $61{:}194$ and $106{:}156$ (see the `section` index-tree Figure 7. A binary or exponential search in the array at the first level permits to find the label of the first `section` ancestor; and a second search in the array pointed by this node permits to find the label of the second `section` ancestor. The application of the exponential search algorithm to the label search in a sorted array of labels necessitates a few more comparisons which don't change the order of its complexity. It is implemented in function `SearchInArray`(a, b), which is given by Algorithm 5.

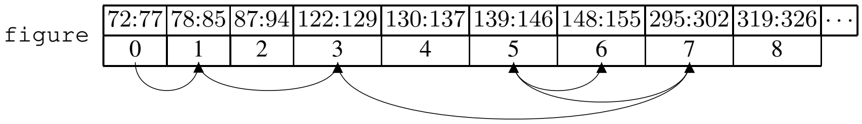


Fig. 8. Comparisons performed during doubling and binary searches in the index-tree for `figure` nodes.

In a list of recursive node labels, a comparison does not permit to divide the search space in two, as it is the case in the sorted arrays of each node of an index-tree. An index based on inverted lists, even if optimized by B^+-tree, do not permit to use binary or exponential search algorithms to find the ancestors or descendants of recursive nodes. For instance, the labels of the `section` nodes, ancestors of the `figure` node in the previous example, are not consecutive in the preordered list of `section` labels (given in Figure 9). From a test on the `section`-node label $65{:}105$, no algorithm can decide to reduce its research to the left or right side of the array, as both sides can (and here do) contain the label of an ancestor of the `figure` node.

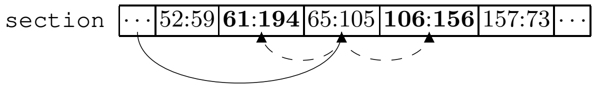


Fig. 9. A subset of the list of `section` node labels, in document order. The labels in bold font correspond to the ancestors of the `figure` node of label 122:129, and are not consecutive.

3 Algorithm

The holistic algorithm proposed is composed of four functions described in Section 3.1. Each function is presented with a short description, its algorithm, a simplified diagram, and an execution example. The correctness of each function, and hence of the whole algorithm, is proved in Section 3.2, and the space complexity is briefly discussed in Section 3.3. The complexity of the algorithm is studied in Section 4.

In general we denote by Q the descendant tree query; by $\alpha, \beta \in Q$ some of its location step; by $A = I[\alpha]$ the index-tree corresponding to the node-test of α (resp. $B = I[\beta]$ for β); and by $a \in A$ a label in A (resp. $b \in B$). Table 1 gives the notations and denominations of some helpful relations between labels. Furthermore, to simplify the algorithm, we signal the end of the arrays by a "fake" label ∞, such that $\forall a$, $a\hat{}\infty$, and such that $\forall r$, $\infty \xrightarrow{r} \infty$.

$a \subset b$	$b.s < a.s$ and $a.e < b.e$	a is a *descendant* of b. b is an *ancestor* of a.
$a < b$	$a.s < b.s$	a is a *preorder predecessor* of b. b is a *preorder successor* of a.
$a\hat{}b$	$a.e < b.s$	a is a *predecessor* of b. b is a *successor* of a.
$a \xrightarrow{r} b$	($r = ad$ and $b \subset a$) or ($r = da$ and $a \subset b$)	a is *in relation r with* b, or b is *in relation r from* a.

Table 1. Notations given a relation $r \in \{ad, da\}$ on labels a, b.

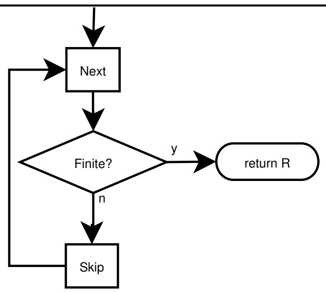
3.1 Functions

The function `Enumerate` (Alg. 1) builds the list of nodes corresponding to the location steps of a descendant tree query Q. To do so, it iteratively calls the functions `Next` and `Skip`; which respectively finds the label of the next node matching Q, and skips it once it is reported; till all elements of one of the index-trees have been considered.

Algorithm 1 $\texttt{Enumerate}(I, Q)$

Given an index I, a query tree Q; the function returns the list of all labels corresponding to Q.

for all $\alpha \in Q$ **do**
  $P[\alpha] \leftarrow$ first element of $I[\alpha]$;
end for
$R \leftarrow$ empty list; $a \leftarrow \texttt{Next}(Q, P)$;
while $a \neq \infty$ **do**
  $R \leftarrow R \cup \{a\}$;
  $P[\texttt{dist}(Q)] \leftarrow \texttt{Skip}(P[\texttt{dist}(Q)])$;
  $a \leftarrow \texttt{Next}(Q, P)$;
end while
return R;



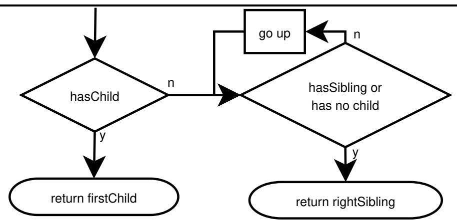
For instance, given the index-trees of Figures 6 and 7, and the descendant tree query given Figure 4, the first call to the function `Next` updates the pointers of P and returns the label $a = 295{:}302$, which corresponds to the first `figure` node, descendant of a `hidden` node, itself ancestor of a `section` node. Entering the loop, the label a is added to the list R and disabled by updating the corresponding value in P to its successor in the `figure` index-tree, $319{:}326$. The loop is then iterated till `Next` returns ∞, when finally the function returns R.

The function `Skip` (Alg. 2) returns the next label in preorder traversal of an index-tree. Its implementation is straightforward: note that there is *always* one label which has a right sibling, if only the ∞ label signalling the end of the index-tree.

Algorithm 2 $\texttt{Skip}(a)$

Given a label $a \neq \infty$, the function returns the next label in the same index-tree in the preorder traversal, or ∞ if none exists.

if $\texttt{hasChild}(a)$ **then return** `firstChild(a)`;**endif**
while $\texttt{rightSibling}(a) = \infty$ and $\texttt{hasParent}(a)$ **do**
  $a \leftarrow \texttt{parent}(a)$;
end while
return $\texttt{rightSibling}(a)$;



For instance, given the `figure` label $295{:}302$, which has no children, but has a sibling, the function `Skip` returns directly its successor $319{:}326$. If given the `section` label $269{:}279$, which has no children and no sibling, the function `Skip` returns directly the first right sibling of its parent, $319{:}326$.

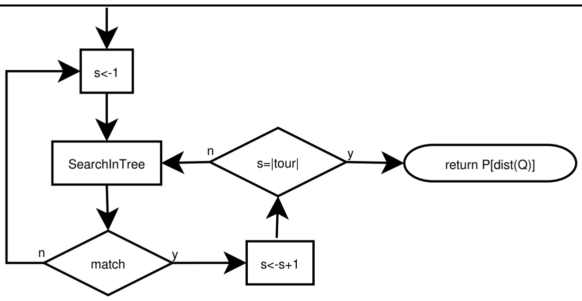
The function `Next` (Alg. 3) searches for the next match of the query Q among preorder successors of the labels given in P. During its search, it updates the values of P, so that the next search ignores the labels already disabled. Note that there is always a match corresponding to Q, if only the set of ∞ labels which terminate the array of the roots of the index-trees. Once the match is found, the function returns the distinguished node of this match.

To perform the search of the next match of Q, each node of Q is successively bound to a label of the corresponding index-tree. The nodes are considered in the order given by the iterated tour of Q. Each node is bound at least once every period of a tour, which is at most $2k$: this permits a better complexity on easy instances (see Section 4).

Algorithm 3 $\texttt{Next}(Q, P)$

Given a query tree Q, and an array P of pointers to labels of the index-trees; the function returns the first label matching the distinguished node of Q, while ignoring all labels preceding the positions indicated by P in the index-trees .

```
α ← any element of tour(Q); s ← 1;
while s < |tour(Q)| do
   β ← the node following α in tour(Q);
   r ← the relation corresponding to (α, β) in Q;
   P[β] ← SearchInTree(P[α], r, P[β]);
   if (P[α] -r-> P[β]) then s ← s + 1; else s ← 1; endif
   α ← β;
end while
return P[dist(Q)];
```



For instance, given the descendant tree query Q of Figure 4 and the array P pointing to the first labels of the index-trees of Figures 6 and 7, the function `Next` successively binds the nodes of Q in the order defined by the tour of Q, as shown in Figure 10. The root of Q (any other node of Q could have been chosen) is first bound to the first `hidden` label available, $138{:}147$ (Fig. 10a). As no label in the `section` index-tree is a descendant of this label, the `hidden` node of Q is unbound, and the `section` node of Q is bound to the label $157{:}173$, which corresponds to the first `section` node possibly a descendant of a preorder successor of the `hidden` node (Fig. 10b). Similarly, no label in the `hidden` index-tree is an ancestor of the node of label $157{:}173$. The `section` node is unbound, and the `hidden` node is bound to the label $209{:}221$, the first node possibly an ancestor of a successor of the `section` node (Fig. 10c).

Following the tour of Q, it is now the turn of the `figure` node to be bound. No `figure` node is a descendant of the node of label $209{:}221$, hence the `hidden` node of Q is unbound, and the `figure` node of Q is bound to the label $295{:}302$ (Fig. 10d). This node is finally part of a match of the query, so the function successfully binds the `hidden` node to the label $282{:}303$ (Fig. 10e) and the `section` node to the label $281{:}304$ (Fig. 10f). Having found a match to the tree descendant query, the function then returns the label $295{:}302$, which corresponds to the distinguished node of the Q, the `figure` node.

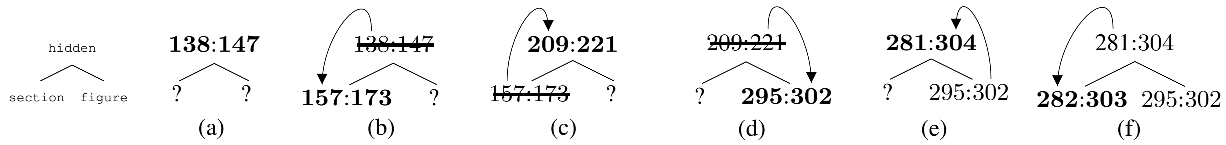


Fig. 10. The successive bindings of the nodes of Q during an execution of the function `Next`. The newly bound labels are in bold, and the newly disabled labels are crossed. The arrows indicate the tour of Q performed by the function.

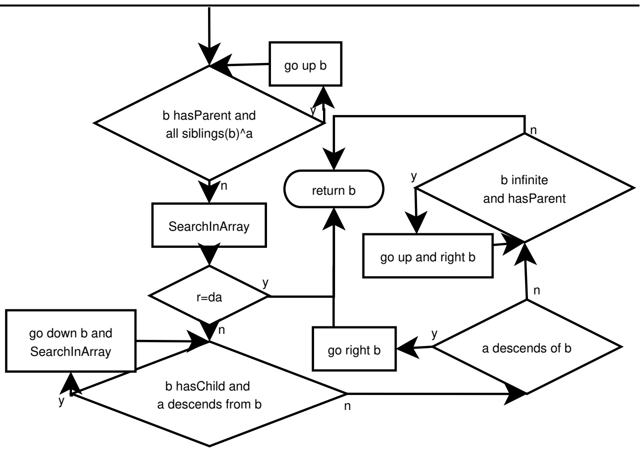
The function `SearchInTree` (Alg. 4) searches, among b_0 and its preorder successors in its index-tree, for a label b in relation r from a. It returns this label if there is one, and otherwise returns the first label which could be in relation r from a successor of a. If even such a label does not exist in the index-tree, it returns the label ∞, the last element of the array at the root of the index-tree.

To do so, the function first moves up from b_0 in the index-tree, as long as all labels of the array containing b are predecessors of a. Once found an array containing at least one label which is either an ancestor, a descendant, or a successor of a, the function returns this label if $r = da$. Otherwise it goes down the index-tree, searching in each array for an ancestor or descendant of a, till it finds a descendant and returns it, or finds that none can exist. If there is no descendant of a in the index-tree of b_0, the function searches the label of the first node possibly a descendant of a preorder successor of a, if necessary going up the index-tree to find a finite label.

Algorithm 4 `SearchInTree`(a, r, b_0)

Given the labels a and b_0, and a relation $r \in \{ad, da\}$; the function returns, if there is one, the first label b, placed after b_0 in the preorder traversal of the index-tree of b_0, such that $a \xrightarrow{r} b$. Otherwise it returns the first label which could be in relation with a successor of a.

$b \leftarrow b_0$;
while `hasParent`(b) and `lastRightSibling`$(b)\hat{}a$ **do**
  $b \leftarrow$ `parent`(b);
end while
$b \leftarrow$ `SearchInArray`(a, b);
if $r = da$ **then return** b; **endif**
while `hasChild`(b) and $a \subset b$ **do**
  $b \leftarrow$ `SearchInArray`$(a,$ `firstChild`$(b))$;
end while
if $a \subset b$ **then** $b \leftarrow$ `rightSibling`(b); **endif**
while $b = \infty$ and `hasParent`(b) **do**
  $b \leftarrow$ `rightSibling`$($`parent`$(b))$;
end while
return b;



For instance, consider the `hidden` label a=138:147, the `section` label b=5:60, and the relation r=ad. In the array containing b, the label 61:194 corresponds to an ancestor of a (see Fig. 11). Any `section` node descendant of a has to be a descendant of this node, hence the function updates b down one level in the `section` index-tree. In this new array the label $b = 106{:}56$ corresponds also to an ancestor of a, which has no `section` descendants: hence there are no `section` nodes among the descendants of a.

Any preorder successor a' of a is either a descendant of b or a successor of it: b cannot be a descendant of a'. Hence the label of the first section node possibly descendant of a preorder successor of a is the sibling of b, of label 157:173, which is returned by the function. Note that if the function is called with $b = 23{:}51$, instead of the first label of the `section` index-tree, the function behaves the same, after checking that no label of the array containing b can be in relation with a (see Fig. 7).

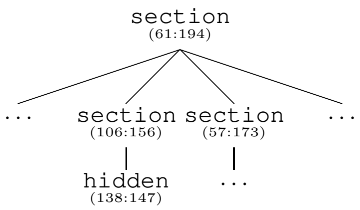


Fig. 11. No `section` node is a descendant of the `hidden` node of label a=138:147. The first `section` node which could be a descendant of a preorder successor of the node of label a has label 157:173.

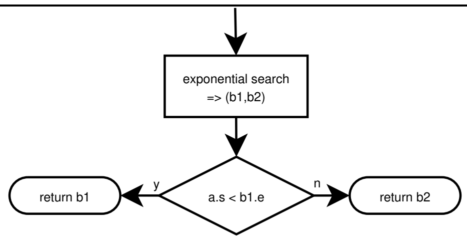
The function `SearchInArray` (Alg. 5) searches, in the part of the array on the right side of b_0, for a label which can be an ancestor or descendant of the label a. It returns this label if there is one, and otherwise returns the first label which could possibly be an ancestor or descendant of a preorder successor of a. To do so, it simply performs an exponential search for the insertion rank of $a.s$ among the $b.s$ components of the labels of the array. Then the relative position of a can be deduced by simply computing $a.s$ with the $b.e$ component of the elements in the array around the insertion rank.

labels on their s component, to find the insertion rank of $a.s$, and decide of the position of a by a comparison of the e components.

Algorithm 5 `SearchInArray`(a, b_0)

Given the labels a and b_0 the function returns the first label b succeeding to b_0 in the same array, such that either $a \subset b$, $b \subset a$, or $a\hat{}b$.

Using exponential search, find b_1 and b_2,
consecutive labels from the same array as b_0 such that
$b_0.s \leq b_1.s < a.s \leq b_2.s$;
if $a.s < b_1.e$ **then return** b_1; **else return** b_2;



For instance, given the `hidden` label 138:147 , and the `figure` label 139:146, the function performs the comparisons described in Figure 8, to find the consecutive labels b_1=130:137 and b_2=139:146. As $138 > 137$, the function finally returns $b_2 = 139{:}146$.

3.2 Correctness proof

We prove the correctness of our algorithm in four steps, the last one being the proof of the correctness of Function `Enumerate`. Lemma 1 states the correctness of Function `SearchInArray`, which is simply adapting the exponential search algorithm to search on arrays of labels. Lemma 2 states the correctness of Function `SearchInTree`, and is used in the proof of Lemma 3 which states the correctness of Function `Next`. Lemma 4 states the correctness of Function `Skip`, and is used with Lemma 3 to prove Theorem 1, which states the correctness of Function `Enumerate` and hence of the general algorithm.

Lemma 1 (Correctness of `SearchInArray`). *Given the labels a and b_0, Algorithm 5 returns the first label b, succeeding to b_0 in its index-tree, such that either $a \subset b$, $b \subset a$, or $a\hat{}b$.*

Proof. By definition of the exponential search, $b_0.s \leq b_1.s < a.s \leq b_2.s$ (see fig. 12). By definition of the labelling

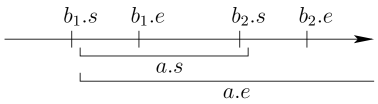


Fig. 12. The possible positions of a in comparison to b_1 and b_2 in Algorithm 5.

scheme, each pair of intervals is related either by an inclusion or by an empty intersection. By definition of the index-tree, there can be only one label b in the array such that $a \subset b$, and all labels b such that $b \subset a$ must be consecutive in the array.

If $a.s < b_1.e$, then $a.e \leq b_1.e$ and b_1 is the unique label of the array such that $a \subset b_1$: that is the correct value to be returned. On the other hand if $a.s > b_1.e$, then $b_1\hat{}a\hat{}b_2$ of $b_2 \subset a$: either way, b_2 is then the label corresponding to the definition of the correct output of the algorithm. $\square$

Lemma 2 (Correctness of `SearchInTree`). *Given the labels a and b_0, and a relation $r \in \{ad, da\}$, Algorithm 4 returns, if there is one, the first label b, preorder successor of b_0 in its index-tree, such that $a \xrightarrow{r} b$. Otherwise it returns the first label which could be in relation with a preorder successor of a.*

Proof. Throughout Algorithm 4, the values taken by variable b delimit which labels are a potential output of the algorithm or not. For conciseness, we call *disabled* the preorder predecessor of b, *available* the others, and b_f the correct return value of Algorithm 4.

The path from the root to b in the index-tree delimits, in each array crossed, the boundary between disabled and available labels. The labels in the arrays not crossed by this path have recursively the same status than their parent. The two successive loops which constitutes the algorithm, disable more labels by updating b with preorder successors, till $b = b_f$ for the relation r.

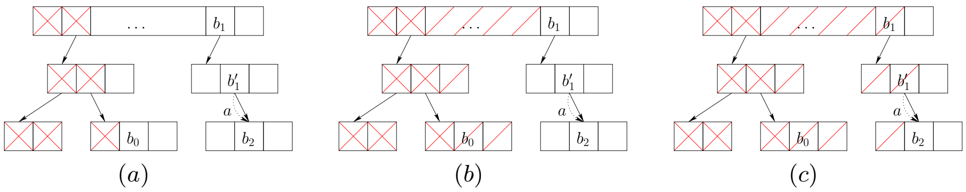


Fig. 13. Schematic representation of the index-tree for b-nodes during the execution of the function `SearchInTree`, at the time of the call (a), after the first loop (b) and after the second loop (c). The crossed locations correspond to disabled labels, and the noted labels are such that $b_0 \hat{} b_1 \supset b_1' \supset a \supset b_2$. The algorithm returns b_1 if $r{=}da$, and b_2 if $r{=}ad$.

Whether $r = da$ or $r = ad$, either a and b_f are on the same root-to-leaf path in the document, or $a \hat{} b_f$. The first loop disables the sub-trees of the index-tree which can't possibly intersect a root-to-leaf path containing a, by updating b up from b_0 when `lastRightSibling`$(b) \hat{} a$. It finds the first array on the path from b_0 up to the root, containing at least one label which is either an ancestor, a descendant, or a successor of a: in Figure 13, this is the array containing b_1. The following exponential search (through the call to `SearchInArray`, proved correct in Lem. 1), finds such a label.

At this point, b is the searched label for $r{=}da$ (descendant-ancestor relation): if b is an ancestor of a, then by construction it is the first one available; otherwise, it is the first available label which could be an ancestor to a successor of a, as all ancestors of b have already been disabled (by being anterior to b_0) and all predecessors of b in the array are such that $b.s < a.s$.

The second loop updates b in the index-tree down the path intersecting the root-to-leaf path containing a, as long as b is a descendant of a: at the end of this loop either $b \subset a$ or $a \hat{} b$: in Figure 13 it corresponds to the path containing b_1, b_1' and b_2. At this point, the ancestors of b are either anterior to b_0 or ancestors of a, and the left siblings of b are not on a root-to-leaf path containing a. If $b \subset a$, then by construction b is the first descendant of a available, it won't be modified anymore by the function and it will be returned. If $a \subset b$ but b has no descendant of the same type, then b cannot be an ancestor of any preorder successor of a, hence the algorithm returns its preorder successor in the index-tree.

If $b = \infty$, a whole array has been disabled. The third loop updates b up the index-tree to find the next available label different from ∞, or ∞ if there is none in the whole index-tree. Hence the correctness of the algorithm, as $a \hat{} \infty$. □

Lemma 3 (Correctness of `Next`). *Given a query tree Q, and an array P of pointers to labels of the index-trees, Algorithm 3 returns the first label matching Q, while ignoring all labels being preorder predecessors of the labels of P.*

Proof. The algorithm counts in variable s the number of consecutive matches in the tour of Q. As the call to the function `SearchInTree` does not change anything if the labels pointed by P are already in relation, any value of s larger than the period $|$`tour`$(Q)|$ of the iterated tour of Q indicates that a match is complete: any label returned by the algorithm corresponds to a match. As Lemma 2 proved that the function `SearchInTree` returns the *first* label b, preorder successor of b_0, such that either $a \xrightarrow{r} b$ or $a\hat{\ }b$, no label part of a match is disabled: the label returned by the algorithms corresponds to the first match available. □

Lemma 4 (Correctness of `Skip`). *Given a label $a \neq \infty$, Algorithm 2 returns the preorder successor of a in its index-tree, or ∞ if none exists.*

Proof. For any node of a tree, the next node in the preorder traversal is its child if it has one, the first right sibling if it has one, or the next node to its parent if it has one. As the array at the root of each index-tree is terminated by a label ∞ which has no children, the algorithm will return the label ∞ at the end of the traversal. □

Theorem 1 (Correctness of `Enumerate`). *Given an index I, a query tree Q, Algorithm 1 returns the list of all labels corresponding to Q.*

Proof. Algorithm 1 is a simple application of the functions `Next` (Alg. 3) and `Skip` (Alg. 2): Iteratively, the function computes the first match constituted of available labels by a call to function `Next`, adds it to the variable R, disables the label matched by the distinguished node of the descendant tree query by a call to function `Skip`, till all the labels are disabled in at least one of the index-trees. The correctness of the functions used is proved by Lemmas 3 and 4: hence Algorithm 1 returns the list of all labels corresponding to Q. □

3.3 Space complexity

As the algorithm outputs sequentially the nodes matching the query, it needs only to maintain a partial matching and the positions in each index delimiting the disabled labels. Hence its memory requirements, apart from the index itself, are small.

If all positions in the index hold in a single memory unit, its needs are linear in the size of the query, and are independent of the input and output sizes. If the document is so large that the positions in the index take more than one unit of memory, it will need $O(k\log(n))$ bytes, where k is the size of the query and n is the size of the document.

4 Adaptive Analysis

4.1 Certificates and non-deterministic algorithms

All correct algorithms have in common that they must check a *certificate* of their result, which can be much larger than the solution of the problem. In particular, queries without a match can have a certificate of arbitrary size, very small or as large as the size of the instance.

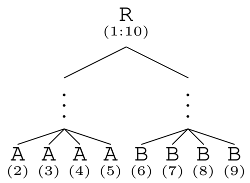


Fig. 14. An easy document for the query $A//B$.

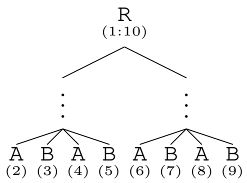


Fig. 15. A difficult document for the query $A//B$.

For instance, the document presented in Figure 14 has no node matching the query $Q = B//A$. A single comparison, between the last label 5 of the A index-tree, and the first label 6 of the B index-tree, permits to certify it: all the A labels are equal or preorder predecessors of 5 and all the B labels are equal or preorder successors of 6, hence $5 < 6$ implies that all the A labels are preorder predecessors of all the B nodes.

On the other hand, the document presented in Figure 15 also has no node matching $Q = A//B$, but this is more difficult to certify: 7 comparisons are needed to check this.

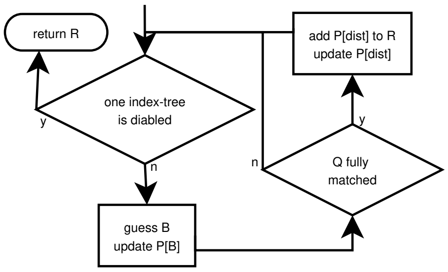
All correct deterministic, probabilistic or non-deterministic algorithms must compute a certificate of some sort, but non-deterministic algorithms can just *guess* a certificate and check it. Then the complexity of the best non-deterministic algorithm corresponds to the size of the certificate, and is called the *non-deterministic complexity* of the instance.

The function `NonDeterministic` (Alg. 6) is a non-deterministic algorithm computing the list of all labels corresponding to Q with the minimum number of comparisons needed.

Algorithm 6 `NonDeterministic`(I, Q)

Given an index I, a descendant tree query Q; the function returns the list of all labels corresponding to Q.

$R \leftarrow \emptyset$;
for all $\alpha \in Q$ **do**
  $P[\alpha] \leftarrow$ first element of $I[\alpha]$;
end for
while $\forall\alpha \in Q,\ I[\alpha] \neq \infty$ **do**
  guess $\beta \in Q$; $\alpha \leftarrow$`Parent`(β) ;
  $a \leftarrow P[\alpha]$;
  $P[\beta] \leftarrow \min\{b \in I[\beta],\ b \geq P[\beta]$ and $(b \subset P[\alpha]$ or $b\hat{}P[\alpha])\}$;
  if $\forall\beta \in Q,\ P[\beta] \subset P[$`Parent`$(\beta)]$ **then**
    $R \leftarrow R \cup \{P[\text{dist}(Q)]\}$;
    $P[\text{dist}(Q)] \leftarrow \min\{b \in I[\text{dist}(Q)],\ b > P[\text{dist}(Q)]\}$;
  end if
  return R;
end while



Of course this algorithm do not use any binary search, because the non-determinism permits to guess directly the insertion ranks. Beside this point, the algorithm performs all the steps that a randomized or deterministic algorithm would perform: it traverses the index-trees in parallel, looking for either some match or some proof that the label currently considered do not correspond to a part of a match.

Definition 3. *The minimal number of comparisons that a non-deterministic algorithm performs on an instance is called the* Non-Deterministic Complexity *of the instance. It is a lower bound of the complexity of any algorithm on this instance.*

This lower bound is usually weak, but still gives a good measure of the relative difficulty of the instances: when an instance is more difficult for a non-deterministic algorithm, then it is also more difficult for restricted algorithms, such as probabilistic or randomized algorithms.

4.2 Difficulty of an instance

Measures of difficulty permit to distinguish between the instances of same size but distinct difficulties. The obvious measures of difficulty for descendant tree queries on XML documents, provided their index-trees, are the sizes of the constituents of the instance: the size n of the document, and the size k of the tree-query. Some other properties can make some documents more difficult than others, among which is the height of the document, and in particular the maximal height of the index-trees.

Definition 4 (recursivity). *The maximal number h of nodes of same type on a root-to-leaf path in the document is an important characteristic of the difficulty of instances, that we call the* recursivity *of the instance.*

While the recursivity of an instance depends only of the document structure, the relation between the tree-query and the document creates also variation in the difficulty of the instance. One way to capture that difficulty is to observe the performance of a non-deterministic algorithm such as Algorithm 6: the performance of the best non-deterministic algorithm is a natural lower bound of the performance of any probabilistic or deterministic algorithm.

Definition 5 (alternation). *For a given instance composed of a descendant tree query, an XML document and its index-trees, we denote by δ the number of negative tests and whole matches performed by Algorithm 6 on this instance. As it is the number of times when Algorithm 6 starts over the matching process from one single node (and "alternate" its template match), we call this measure of difficulty the* alternation *of the instance.*

For instance the alternation of the instance formed by the query $Q = A//B$ and the document of Figure 14 is $\delta = 1$; and the alternation of the instance formed by the query $Q = A//B$ and the document of Figure 15 is $\delta = 7$;

Note that, for instances without match, the alternation is exactly the number of comparisons performed by the Algorithm 6, i.e. the non-deterministic complexity of the instance. For instances with some matches, the alternation is the number of matches plus the number of comparisons disabling labels. This is different from the non-deterministic complexity, but adequate for a study of the complexity of deterministic algorithms: for instance a document, built by the adversary of a deterministic algorithm A, will maximize the number of comparisons performed by A for each failed match; to the extent where A performs as many comparisons on a successful match than on a failed match.

4.3 Complexity of the algorithm

We prove here the complexity of our algorithm by an adaptive analysis in function of n, k, h, and δ.

Theorem 2. *The function* `Enumerate` *(Alg. 1) performs $O(\delta hk \log(1+n/\delta hk))$ comparisons on an instance of alternation δ composed of a descendant tree query of k nodes; and of a document of size n and recursivity h.*

Note that an instance of recursivity h has no index-tree of height larger than h.

Proof. Let δ be the alternation of the instance, and $(a_i)_{i \leq \delta}$ the corresponding ordered sequence of labels chosen initially or just after a failed or successful match, as guessed by a non-deterministic algorithm. By definition, δ is the minimal length of such a sequence. For commodity, note a_0 a pedigree preceding any other in the document.

Each exponential search performed by the algorithm is said to be "in phase i" if the label a searched is either placed between a_{i-1} and a_i, or equal to a_i, for all $i \in \{0, \ldots, \delta-1\}$. Such a phase is called *positive* if a_i is a match, and *negative* otherwise. There are exactly δ such phases, and as the nodes are considered following the iterated tour of Q, in each phase the algorithm performs at most $2k$ calls to the function `SearchInTree`. During each of these calls, in the worst case b is going up to the root and down to a leaf. As the index-tree is of height at most h, at most h calls to the function `SearchInArray` are performed during a call to the function `SearchInTree`.

For each phase $i \in \{0, \ldots, \delta\}$, for each call $j \in \{1, \ldots, k\}$ to the function `SearchInTree`, and for each of the levels $l \in \{1, \ldots, h\}$ of the corresponding index-tree, let be $n_{(i,j,l)}$ the number of labels skipped by the function `SearchInArray`. Each call to the function `SearchInArray` performs then $O(\log(1 + n_{(i,j,l)}))$ comparisons, summing to $O(\sum_{i=0}^{\delta-1} \sum_{j=1}^{k} \sum_{l=1}^{h} \log(1+n_{(i,j,l)}))$. This is smaller than

$$O\left(\delta hk \log\left(1 + \sum_{i=0}^{\delta-1} \sum_{j=1}^{k} \sum_{l=1}^{h} n_{(i,j,l)}/\delta hk\right)\right),$$

because of the concavity of the function $\log(1 + x)$. As each search starts exactly where the previous one ended, $\sum_{i=0}^{\delta-1} \sum_{j=1}^{k} \sum_{l=1}^{h} n_{(i,j,l)} \leq n$, and the algorithm's complexity is $O(\delta hk \log(1 + n/\delta hk))$. $\square$

4.4 Lower Bound

In this section we prove that the complexity of the algorithm presented in Section 3 is asymptotically optimal.

Theorem 3. *For any $\delta \geq 1$, $h \geq 1$, $k \geq 2$, $n \geq \delta(h + k)$ any deterministic algorithm answering descendant tree queries, on instances formed by a descendant tree query of $O(k)$ nodes, and by a document of $O(n)$ nodes and recursivity $O(h)$; has complexity $\Omega(\delta hk \log(n/\delta hk))$.*

Proof. Here is a summary of the proof: we show first how the lower bound in the case where $(\delta = 1, h = 1, k = 2, n \geq 3)$ is exactly the lower bound for the binary search in a sorted array. Then we successively generalize the lower bound to the cases where the values of h,k and δ are relaxed.

We first consider the descendant tree query $A//B$ of size 2; on documents of $n+2$ nodes, such that the root is of type R, has $n-1$ children of type B and 1 child of type A, which itself has a child of type B (see Fig. 16). This forms instances of alternation $\delta = 1$, as guessing the B-node child of the A-node suffices to certify the match. The index-tree for B is an array of n elements among which exactly one is the solution. As each comparison permits only to divide the search space by two, $\log_2 n$ comparisons are necessary to any deterministic algorithm solving the instance, hence the lower bound for the instances such that $(\delta = 1, h = 1, k = 2, n \geq 3)$.

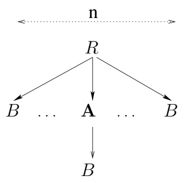


Fig. 16. (δ=1, h=1, k=2, n≥3)

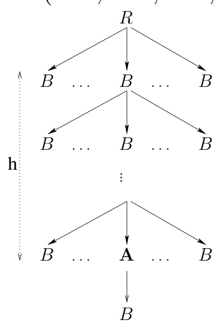


Fig. 17. (δ=1, h≥1, k=2, n≥h+2)

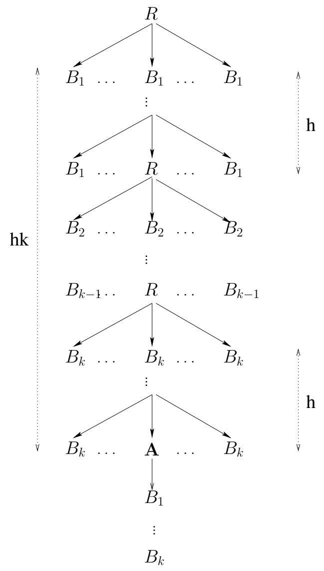


Fig. 18. (δ=1, h≥1, k≥2, n≥h+k)

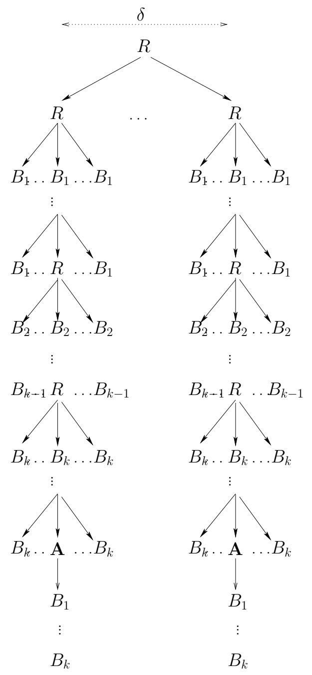


Fig. 19. (δ≥1, h≥1, k≥2, n≥δ(h+k))

Next we relax the recursivity h of the document: this is the maximal height of an index-tree. We consider the query $Q = A//B$ of length 2; on documents of size $n+2$, containing 1 node of type R, 1 node of type A, and n nodes of type B, such that the root and the B nodes form a tree with a unique trunk of B nodes, and such that the A node is parent of a B-node (see Fig. 17). The alternation of this kind of instance is always $\delta = 1$, as guessing the positions of the B-node child of an A-node is sufficient to certify the unique match. An adversary strategy can force any deterministic algorithm to perform $\Omega(h \log n/h)$ comparisons on such instances, by hiding at each level the positions of the B-nodes which are parents, and hiding in the whole tree the A-node. Hence the lower bound, for the instances such that $(\delta = 1, h \geq 1, k = 2, n \geq h+2)$.

Then we relax k, the size of the descendant tree query. We consider the descendant tree query

$$Q = A//B_1//B_2//\ldots//B_k,$$

of length $k+1$; on documents of size $n+k+2$, containing k nodes of type R, 1 node of type A, and n/k nodes of type B_i for all $i \in \{1,\ldots,k\}$; such that it can be constructed recursively from $i=1$ to $i=k$ from the construction $\mathcal{T}$ for $k=1$, by replacing the A-node at each step by a copy of $\mathcal{T}$ where each node of type B is replaced by a node of type B_i (see Fig. 18). Each root-to-leaf path contains no more than h nodes of each type; the alternation of each instance is $\delta = 1$ because guessing non-deterministically the unique descendant of A in each index-tree is

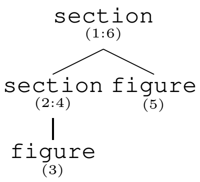


Fig. 20. A document on which holistic algorithm with index-trees cannot solve the query A/B.

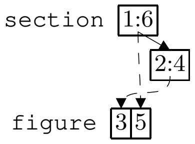


Fig. 21. The corresponding index-trees: dashed arrows link `section` nodes to `figure` children.

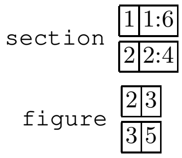


Fig. 22. Levelled indexes would permit exponential search use for parent-child search.

sufficient to certify the only match of this instance. As before, an adversary can make any deterministic algorithm perform $\Omega(hk\log(n/hk))$ comparisons to check the entire match, hence the lower bound, for the instances such that $(\delta = 1, h \geq 1, k \geq 2, n \geq h + k)$.

Finally we relax the difficulty δ of the instances. We consider instances of difficulty δ formed by the same query $Q = A//B_1//B_2//\ldots//B_k$, of length $k+1$; on documents formed by δ documents from the previous case (see Fig. 19). As previously, each root-to-leaf path contains no more than h nodes of each type. The alternation of the instance is exactly δ, as there are δ matches, and guessing the positions of the δ descendants of A-nodes in each index-tree is sufficient to certify the result. Each sub-instance is independent, and an adversary can force any deterministic algorithm to perform $\Omega(\delta hk\log(n/\delta hk))$ comparisons, hence the general lower bound, for the instances such that $(\delta \geq 1, h \geq 1, k = 2, n \geq \delta(h + k))$. □

5 Conclusion

In this paper we showed how the structure of an index-tree permits to take advantage of random access memory to answer XPath location steps along the `descendant` axis. We gave an holistic algorithm using index-trees and showed that, apart from the index, it has small memory requirements. We introduced a measure of the difficulty of the instances formed by a query and a document, and deduced the asymptotic complexity of the problem in the comparison model, provided with the index-trees of the document.

Even with labels augmented with some information about the level of the corresponding node, as done in similar works [8, 13], this holistic approach cannot be directly applied to solve location steps along the parent child axis, or at least not with index-trees; because labels cannot be disabled based solely on their preorder position during a parent-child axis location step. Consider for instance the document of Figure 20: the `figure` child of the first `section` node is a *successor* of the `figure` child of the second `section` node. In such a configuration, disabling the label of the first `figure` label for the first `section` node must not disable it for the second `section` node (see Fig. 21). An index structured by level as in Figure 22 or by *Path Sequences* [15] shouldn't have this problem, and should permit to apply holistic techniques to solve location steps along `parent-child`, `following-sibling` and all forward axes.

Those techniques can be applied with the same data-structure to the more general problem of Twig Pattern Matching. They need to be combined with stack techniques as those used by Bruno *et al.* [8], to represent in a compact way the intermediate results. A related variant that we plan to study, is how to represent in a compact way the possibly exponential set of matches of a twig pattern, and to compute it fast with small memory requirements. Such a representation would permit to avoid to enumerate an exponential list of matches, it would be adapted to the computation of XML projections, and it would be welcome for the *transmission* of results: typically queries are asked by a distant terminal and answered by a main server: returning the answer in a compact way seems an interesting feature.

Finally, the comparison model is realistic for databases where the index can be held partially in random access memory, e.g. if the memory can hold at least the index parts related to the query. The performance in cached RAM models of the exponential search algorithm, which is used throughout our algorithm, is not known. However it seems promising, as simulations have shown similar techniques, such as fast-forwarding [15], to perform well in practise, as compared to binary search and sequential access algorithms.

Acknowledgements: This paper was written while the author was at the University of British Columbia as a postdoctoral fellow. We would like to thank Laks V S Lakshmanan for his introduction to the XML world, Ganesh Ramesh for interesting discussions, Mirela Andronescu and many people from UBC for interesting comments.

References

1. S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 547–556. Society for Industrial and Applied Mathematics, 2001.
2. S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 947–953. Society for Industrial and Applied Mathematics, 2002.
3. J. Barbay. Optimality of randomized algorithms for the intersection problem. In A. Albrecht, editor, *Lecture Notes in Computer Science*, volume 2827 / 2003, pages 26–38. LNCS, Springer-Verlag Heidelberg, November 2003. 3-540-20103-3.
4. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the thirteenth ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 390–399. ACM-SIAM, ACM, January 2002.
5. J. Barbay and C. Kenyon. Deterministic algorithm for the t-threshold set problem. In H. O. Toshihide Ibaraki, Noki Katoh, editor, *Lecture Notes in Computer Science*, pages 575–584. Springer-Verlag, 2003. Proceedings of the 14th Annual International Symposium on Algorithms And Computation (ISAAC).
6. J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.
7. A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath) 2.0. Technical report, W3C Working Draft, November 2003. `http://www.w3.org/TR/xpath20/`.
8. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321. ACM Press, 2002.
9. J. Clark and S. DeRose. Xml path language (xpath). Technical report, W3C Recommendation, November 1999. `http://www.w3.org/TR/xpath/`.
10. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the* 11^{th} *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.
11. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments, Lecture Notes in Computer Science*, pages 5–6, Washington DC, January 2001.
12. V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.
13. T. Grust. Accelerating xpath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120. ACM Press, 2002.
14. H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. Timber: A native xml database. *VLDB*, 11(4):274–291, 2002.
15. I. Manolescu, A. Arion, A. Bonifati, and A. Pugliese. Path sequence-based xml query processing. submitted for publication.
16. K. Mehlhorn. Sorting presorted files. In Springer, editor, *Proceedings of the 4th GI-Conference on Theoretical Computer Science*, volume 67 of *Lecture Notes in Computer Science*, pages 199–212, 1979.
17. K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, chapter 4.2 Nearly Optimal Binary Search Tree, pages 184–185. Springer-Verlag, 1984.
18. O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59:153–179, 1995.
19. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *The VLDB Journal*, pages 302–314, 1999.