Logarithmic Complexity for a class of XML Queries

Jérémy Barbay

University of British Colombia, 201-2366 Main Mall,
Vancouver, B.C. V6T 1Z4 CANADA,
email: jeremy@cs.ubc.ca,
home page: http://www.cs.ubc.ca/~jeremy

Abstract. The index of an XML document typically consists of a set of lists of node references. For each node type, a list gives the references of all nodes of this type, in the order defined by the prefix traversal of the document.
A twig pattern matching query is a labeled tree structure (a "twig pattern"), it is answered by the list of all occurrences of this pattern in the document, and it can be computed faster using an index. While previous results in the field propose index structures and algorithms [7] which answer twig pattern queries with a complexity at best linear in the size of the document, we propose an index $\frac{3}{2}$ times larger which allows to answer twig pattern queries with a number of comparisons logarithmic in the size of the document.
As answering efficiently twig pattern matching queries necessitates a sophisticate encoding of the output to avoid its potentially exponential size [4], we expose our technique on two simpler problems, of output size respectively constant and linear in the size of the document, and we claim that the technique can be applied to answer twig pattern queries using a logarithmic number of comparisons as well.

1 Introduction

XML documents can be stored and indexed either in a relational database, or in a native storage, with different advantages and disadvantages [9]. The index of a native XML database is typically formed by a set of lists of nodes references, such that for each node type (or other predicate specifying nodes), a list gives the `(Id,left:right,level)` references of all nodes of this type, in the order defined by the prefix traversal of the document [7].

The answer to a *Twig Pattern Matching* query is the list of all occurrences of a labeled tree structure (a "twig pattern") in the document. Several algorithms using this kind of indexes have been studied, leading to an optimal algorithm which answers twig pattern queries using a number of comparisons linear in the size of the document [4], by cleverly encoding the potentially exponential output of the query. However, this linear complexity in the worst case is still much

[1] \$Date: 2004/04/05 21:57:44 Revision: 1.33 \$

worse than the results of indexes in relational databases, where a logarithmic complexity is quite common.

We propose an index slightly different from the usual XML index [4, 7], where we encode *in the structure* the ancestor-descendant relationships between nodes, even though this information is already present in the values of the quadruplets, with an overhead factor of $\frac{3}{2}$ in the memory cost. We show that this index permits to answer queries in logarithmic complexity.

As the potentially exponential size of the output is not our concern here, we focus on the *Existential Matching* problem (to decide whether a document contains *at least* one matching of a given twig pattern) and the *Distinguished Twig Pattern Matching* queries (asking for an enumeration of the references of a distinguished predicate in all the matchings of a twig pattern) where the output is respectively constant and linear in the size of the document.

We give an informal method answering existential matching queries using $O(hk \log(1 + n/hk))$ comparisons, for any twig pattern of k predicates, on any document of n nodes for which no branch contain more than h nodes of same type. We give a non-deterministic method answering distinguished twig pattern matching queries, define the non-deterministic complexity for distinguished twig pattern matching queries, and the related difficulty measure δ, called the *alternation*. We finally propose an algorithm answering distinguished twig pattern matching queries with $O(\delta hk \log(1 + n/\delta hk))$ comparisons, on any instance of alternation δ, composed of any distinguished twig pattern query of k predicates, on any document of n nodes for which no branch contains more than h nodes of same type.

We claim that this index structure and the related algorithms can easily be combined with previous results to answer twig pattern matching queries, with a complexity logarithmic in the size of the document.

We proceed as follows: We start by defining our index structure in section 2, discussing several encodings to prove that this index is at most $\frac{3}{2}$ larger than the usual one.

We show how to use this index to answer existential matching queries in section 3. This permits to introduce several useful notations (sec. 3.1), an informal method using binary search to answer existential matching queries on some specific twig patterns (sec. 3.2).

We show how to answer distinguished twig queries in section 4. We define a non-deterministic method to answer queries on general twig patterns (sec. 4.1), which gives the idea of our main algorithm (alg. 3); define the alternation, a measure of difficulty inspired by the non-deterministic complexity of an instance (sec. 4.2); give the functions implementing our algorithm to answer distinguished twig pattern queries (sec. 4.3); and analyze its complexity (sec. 4.4) by giving an upper bound, and comparing it to the complexity of other algorithms.

We finally conclude with some interesting perspectives in section 5.

2 Data Structure

Definition 1. *The* Pedigree *of a node of the document is a tuple* $(Id, left{:}right, level)$ *where* Id *is an identifier for the document;* $left$ *is the rank of the opening of the element in the order defined by the prefix traversal of the document, counting each node twice, once the fist time it is reached, and once after its last descendant has been reached;* $right$ *is the rank of the closing of this element in the same order; and* $level$ *is the number of arcs separating this node from the root of the document plus one. The pedigrees for leaves are denoted by* $(Id, left, level)$ *without loss of information, as in this case* $left = right$.

Note that whereas the Id information is mandatory in a search engine managing several XML documents, it is not relevant in our examples using a single document. The algorithms can be trivially adapted to take the Id values into account, and we will omit this value for the rest of this article to keep notations more concise.

Example 1. For instance, figure 1 represents an XML document for a `book`. It has elements which contain only text, such as `title`s, elements containing other elements of different type, such as `book` but also elements which can contain recursively other elements of same type, such as `sect`s. Below each label of the node is noted its pedigree, without the Id.

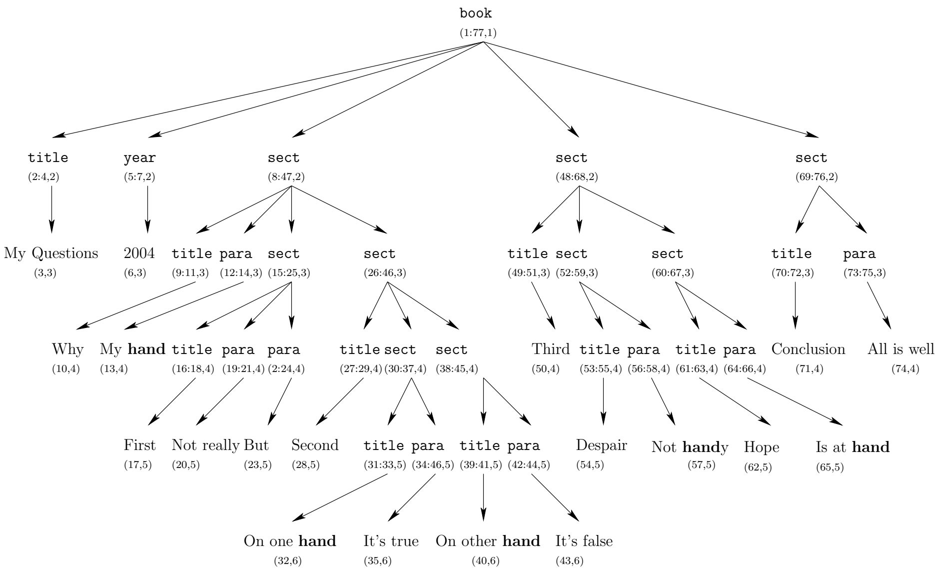

Fig. 1. An example of XML document

Definition 2. *The* Index-Tree *corresponding to a given predicate label* α *in a document* D *is a tree* $I[\alpha]$ *containing all pedigrees of nodes corresponding to this predicate, such that any ancestor-descendant relationship in* D *corresponds to a parent-child or ancestor-descendant relationship in* $I[\alpha]$*. The specific encoding of this tree is free as long as it allows a binary search among each set of children.*

We describe here shortly two distinct encodings of such index-trees. The encoding does not impact the performance of our algorithms as long as it is possible to perform a binary search. Usual structures for index list the pedigrees of the nodes in an array, in the order defined by the prefix traversal of the document. The children of a particular node of the index-tree are not always in consecutive positions, thus rendering a binary search among the children impossible. To allow such a binary search, the children must be placed in consecutive positions in an array, and consecutive sequences of children must be easily distinguished.

One way to achieve this goal is to encode an index-tree in an array, by ordering the nodes of the index-tree in width first traversal, so that all children are in consecutive positions, and by adding to the pedigree of each node two additional values namely *up* and *down*, identifying respectively the index of its parent in the index-tree and the index of its first child, or zero if none exist.

The *down* value trivially encodes the structure of the tree. The *up* value is necessary to easily distinguish the end of a successive sequence of children, and optionally permits an easier traversal up the tree, saving the need of a stack to save the path from the root.

This encoding has a constant factor space overhead of $\frac{6}{4} = \frac{3}{2}$ in comparison with existing XML index structures, which store 4 values for each node, Id, $left$:$right$ and $level$, as our encoding stores two more values *up* and *down*. As we will see, this constant factor overhead in memory cost is well balanced with the exponential speed-up it permits.

Example 2. For instance, here is an array encoding of the index-tree A for the `sect` nodes of the document given figure 1:

i	1	2	3	4	5	6	7	8	9
$left$:$right$	8:47	48:68	69:76	15:25	26:46	30:37	38:45	52:59	60:67
$level$	2	2	2	3	3	4	4	3	2
up	0	0	0	1	1	5	5	2	2
$down$	4	8	0	0	4	0	0	0	0

This particular array is obtained by a traversal of the tree where at each recursive step, when treating a node $A[i]$ at position i (already added in the table),

- $A[i].down$ is set to the index of the first position free in the table;
- the children of $A[i]$ are copied to the table in order, starting at position $A[i].down$, and such that their *up* value is equal to i;
- if any, each children of $A[i]$ is recursively treated.

The traversal order of the tree is not very important, as long as all children from a node n are placed consecutively starting at position $n.down$.

Another way to realize this goal is to encode an index-tree using dynamic allocation to create a sorted array of pointers to the children. This has the advantage to allow easier update, and it makes more readable graphs: figure 2 represents the index-tree for the `sect` nodes of the document of figure 1.

Definition 3. *The* Extended Pedigree *of a node is the tuple* $(left{:}right, level, up, down)$.

Note that the general notation would be $(Id, left{:}right, level, up, down)$ but than we ignore Id values in this paper, and that if the values Id, $left$, $right$ and $level$ are uniquely defined by the document, the values up and $down$ are defined by the *encoding* of the index-tree. To avoid confusion with the nodes of other trees, the nodes of the index-trees are called extended pedigrees for the rest of the paper.

3 Existential Matching

Definition 4. *A* Twig Pattern *is a labeled tree structure whose labels include element tags, attribute-value comparisons, and string values. The edges of the twig patterns are either parent-child edges (pc for short, graphically depicted using a single line), or ancestor-descendant edges (ad for short, graphically depicted using a double line). A* Distinguished Twig Pattern *is just a twig pattern with a distinguished predicate.*

Twig patterns are used for matching XQuery queries to the relevant portions of documents in an XML database. It is necessary to use a clever encoding of the output to counteract the potentially exponential size of the output of such a query [4]. The problem is simpler in the particular case where one simply needs to check whether there is *at least* one matching: the output is then of constant size.

Definition 5. *The answer of an* Existential Twig Matching *query is true if and only if the corresponding twig pattern has at least one matching in the document.*

3.1 Notations

Each comparison between extended pedigrees corresponds to a finite number of integer comparisons. Hence it is sufficient to count extended pedigree comparisons, when estimating the complexity within a constant factor. Table 1 gives the notations that we introduce for extended pedigrees comparison, with their translation in terms of value comparisons. We denote by Q the array encoding a twig pattern; $q = Q[i]$ one of its predicates (resp. q'); $\alpha = q.label$ the corresponding label (resp. $\beta = q'.label$); $A = I[\alpha]$ the corresponding index-tree (resp. $B = I[\beta]$); and $a = A[i]$ or $a \in A$ an extended pedigree (resp. $b = B[i]$ or $a \in B$).

Using those notations, $a \overset{ad}{\to} b$ means that a is an ancestor of b; $a \overset{da}{\to} b$ means that a is a descendant of b; $a \overset{pc}{\to} b$ means that a is a parent of b; $a \overset{cp}{\to} b$ means

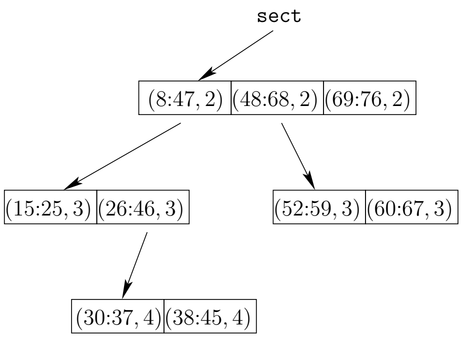

Fig. 2. A dynamic representation of the index-tree for `sect` nodes.

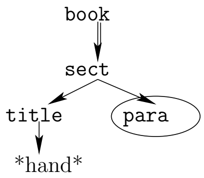

Fig. 3. A Distinguished Twig Pattern. Note the circled distinguished predicate.

$a \subset b$	$b.left < a.left$ and $a.right < b.right$
$a < b$	$a.right < b.left$
$\frac{a}{b}$	$a.level + 1 = b.level$
$a \xrightarrow{r} b$	$(b \subset a$ and $($ $r = ad$ or $(r = pc$ and $\frac{b}{a})))$ or $(a \subset b$ and $($ $r = da$ or $(r = cp$ and $\frac{a}{b})))$
$a \xrightarrow{r?} b$	$(a \subset b$ and $($ $r = ad$ or $r = pc$ $)$ $)$ or $(b \subset a$ and $($ $r = da$ or $r = cp$ $))$

Table 1. Notations given two index-trees A and B, two extended pedigrees a and b in those respective index-trees, and one relation $r \in \{pc, cp, ad, da\}$.

that a is a child of b; $a \subset b$ means that a is in the subtree of root b; $a < b$ means that a's branch is on the left of b's branch; $\frac{a}{b}$ means that a is exactly one level above b (but maybe on a different branch) ; $a \xrightarrow{r} b$ means that a is in relation r with b; and $a \xrightarrow{r?} b$ means that a could be in relation r with a child of b.

The encoding of a distinguished twig pattern is not very important, as they are usually small. Again in an effort to simplify notation, we will assume that they are implemented using an array representation. To avoid confusion with the nodes of other trees, the nodes of the twig pattern are called predicates for the rest of the paper. Each predicate is represented by an column of the array, indicating its *label*, the *relation* with its parent (pc or ad), and the position *down* of its first child. This permits to designate the distinguished predicate by its position in the array. In the graphical representation the distinguished predicate is circled.

Example 3. For instance, here is the array representation of the twig query presented in figure 3. The distinguished predicate is the element `para`, with position

3.

i	0	1	2	3	4
$label$	$book$	$sect$	$title$	$para$	$*hand*$
$relation$	ad	ad	pc	pc	pc
$down$	1	2	4	0	0

Note that in the graphical representation the implicit parent of the root of the query is the root of the document, and that there is an implicit relation ancestor-descendant between those. The array representation nicely explicits those.

For short, distinguished twig patterns which are branches will also be denoted by their XPath equivalent, the distinguished predicate being noted in bold face. For instance, $A//B/\mathbf{C}/D$ stands for the twig pattern whose root is a predicate asking for a node of type A, having a descendant of type B, which have a child of type C, which is distinghished and itself has a child of type D.

An index can be very complicated. The relevant feature to consider here is that it must provide an index-tree for the label of every predicate of a twig pattern. Simplifying it for our discussion, we see it as an array of index-trees, indexed by the labels of the predicates.

Example 4. For instance, the relevant index needed to answer the query of figure 3 for the document in figure 1 would be:

	0	1	2	3	4	5	6	7	8	9
`book`	1:77, 1									
`sect`	8:47, 2	48:68, 2	69:76, 2	15:25, 3	26:46, 3	30:37, 4	38:45, 4	52:59, 3	60:67, 2	
`title`	2:4, 2	9:11, 3	16:18, 4	27:29, 4	31:33, 5	39:41, 5	49:51, 3	53:55, 4	61:63, 4	70:72, 3
`para`	12:14, 3	19:21, 4	22:24, 4	34:36, 5	42:44, 5	56:58, 4	64:66, 4	73:75, 3		
`*hand*`	13, 4	32, 6	40, 6	57, 5	65, 5					

Note that we omitted the *up* and *down* information, as they are relevant only for the `sect` index-tree, and are already given in example 2.

3.2 Binary search

Given a twig pattern query Q, we use an index I to efficiently search a document for matchings. For each predicate $q \in Q$, an index structure like the one in [4] would contain a stream of pedigrees ordered as in the prefix traversal of the tree. These streams, even if implemented in an array, do not permit binary search, because the pedigrees corresponding to a query are not in consecutive positions: a single comparison with a pedigree of the array doesn't yield information about whether the desired matching is on the left or on the right: there might be several, on both sides.

Indeed, the argument is valid for *any total order* on the pedigrees, and there is no possible array encoding which always permits an efficient search. This is directly due to the tree structure of the document. As the problem comes from the tree structure of the document, the natural answer is to introduce a tree structure in the index, in the form of the index-trees defined in section 2.

If there were in total n nodes in the index-tree considered, k free nodes in the pattern, and if no index-tree was higher than h, there is a method answering the existential twig pattern matching query with $O(hk\log(1+n/hk))$ comparisons.

In practice, when looking for several matchings of the same pattern in a document, a direct application of binary search is not efficient. However, algorithms slightly more sophisticated can be used to produce the same result, like doubling search [2] or unbounded search.

4 Distinguished Twig Matching

The existential twig pattern matching queries are not very useful in practice, but their study gives a gentle introduction to the concepts of this paper. Another variant avoiding the potential exponential size of the answer of a twig pattern matching query is when a predicate is distinguished among those of the twig pattern, and one wants to list all nodes in the document matching this particular predicate. The size of the answer is then linear in the size of the document.

Definition 6. *The answer to a* Distinguished Twig Pattern Matching *query is the enumeration of all the references of the nodes in the document matching the distinguished predicate within a context matching the twig pattern.*

4.1 Non-deterministic method

We need to extend our method to answer more general queries, in particular when the twig pattern is not a branch. Note that having one predicate initially bound is *not* a particular case, as adding the root of the document to the twig pattern produces such a bound predicate. We present here an informal non-deterministic method, which gives insights into the behaviour of the algorithm 3 that we propose to answer distinguished twig pattern queries, and enables us to introduce elegantly our measure of difficulty in the next section.

We consider a distinguished twig pattern Q of distinguished predicate d, and an index I. The predicates of Q are initially "free", and will be successively "bound" to various extended pedigree from the index-trees of I, and freed. These extended pedigree are initially all "available" and will progressively be "disabled" as they are considered (being then not available any more).

The method goes as follows, non-deterministically "guessing" predicates in Q and in the index-trees of I. First it guesses an available predicate q in Q, and binds it to the first extended pedigree a from the corresponding index-tree $A = I[\alpha]$ in the order defined by the prefix traversal of A. Then it iterates the following steps, as long as there are still available extended pedigrees in each index-tree:

1. it guesses a free predicate q' in Q, which is connected to a bound predicate q, notes the label α of q and β of q', the relation r from q to q' and the first available extended pedigree a of $A = I[\alpha]$;
2. it guesses an extended pedigree b of the index-tree $B = I[\beta]$, such that b is available and of minimal level among the extended pedigrees susceptible to be in relation r from a, and binds q' to b: all extended pedigrees before b in the prefix order are now disabled;

3. if $a \xrightarrow{r} b$ and all predicates are bound:
   - it outputs the extended pedigree bound to the distinguished predicate of the twig pattern, disables it, and frees all predicates;
   - it guesses a free predicate q in Q, and binds it to the first extended pedigree a from the corresponding index-tree $A = I[\alpha]$ in the order of the prefix traversal of A;
4. if $!(a \xrightarrow{r} b)$, it frees all predicates except q'.

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

The non-determinism permits to accelerate the search of the matchings, but only within some limits. Even if a non-deterministic algorithm can guess the position of each matching it needs to perform a certain number of comparisons to check it. And in order to prove that there is no other matchings, the algorithm must perform several other comparisons.

Definition 7. *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 than another for a non-deterministic algorithm, then it is more difficult for more constrained algorithms too.

4.2 Measure of the difficulty of an instance

For those queries as for the existential twig pattern matching queries, documents and twig patterns can be very different in their difficulty to *any conceivable* algorithm, while very similar in size. The analysis in the worst case among all instances of same size does not take into account this property, which can lead to a contradiction between theoretical and practical results [5, 6]. A measure of difficulty permits to distinguish between the instances of same size but distinct difficulty. We will show later that the complexity of our algorithm depends on it.

Definition 8 (alternation). *For a given instance composed of a distinguished twig pattern, an XML document and its index, we denote by δ the number of matchings of this instance, plus the minimal number of failures of a non-deterministic algorithm answering this query. As it is the number of times a non-deterministic method alternates the basic bound element it is looking for, we call this measure of difficulty the* alternation *of the instance.*

Example 5. For instance the alternation of the instance of the query given figure 3 on the document of figure 1 is at most 5: by successively checking each `*hand*` references, a non-deterministic algorithm can find the 2 matchings with only 3 failures.

Note that the alternation does not count the comparisons performed by the non-deterministic algorithm, nor even the number of non-deterministic guesses, but exactly the minimal number of matching successes and failures on the entire twig pattern: whether the failing is taking one comparison or more is irrelevant.

It is good to count equally successes and failures in a measure of difficulty for the worst case performance analysis: an instance built by an adversary will maximize the number of comparisons for each matching failure, to the extent where an algorithm can perform as many comparisons on a success matching than on a failure matching.

4.3 Algorithms

Unbounded Search The algorithms that we propose here use an `unbounded search` algorithm, which looks 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 has already been studied before, it can be implemented using the doubling search and binary search algorithms [1, 5, 8], and is then of complexity $2\lceil\log_2(p-init)\rceil$, and can be implemented directly [2] to improve the complexity by a constant factor of less than 2. Its adaptation to the pedigree search in a sorted array is trivial: just compare the $left$ components of the pedigrees during the search. We will denote this adaptation `UnboundedSearchInArray`(a, b), which is looking in a sub-array of an index-tree starting from b for the pedigree b' which either can contain or be contained by the pedigree a, or is the first pedigree of the array such that $a < b'$. Note that the value of b' is uniquely defined as by definition of an index-tree the consecutive pedigrees of the sub-arrays are mutually disjoints.

Enumeration of matchings Function `Enumerate` (alg. 1) is an example of how our algorithms can be used to enumerate all matchings of a distinguished twig pattern in a list: The only steps needed are to initialize an array of extended pedigrees to the first element of each index-tree corresponding to a predicate in the twig pattern; and to iteratively call the functions `TwigMatching` and `Next`, which respectively finds the next distinguished matching and skips it once it is reported. Its complexity is the sum of the complexities of the calls to `TwigMatching`, as the cost of the initialization and of the calls to `Next` is negligible. Note that ideally, one wouldn't use the function `Enumerate` but one of his own, treating matchings as they come.

The function `Next` (alg. 2 in the appendix) returns the next extended pedigree in the order defined by the prefix traversal of an index-tree. Its implementation is classical, its complexity bounded by the height of the index-tree.

Algorithm 1 `Enumerate`(I, Q, d, σ)

Given an index I, a query tree Q, a distinguished predicate d, and an order σ on the predicates of Q; the algorithm returns the list of all extended pedigrees corresponding to a node matching the distinguished predicate of the query tree Q.

for all $q \in Q$ **do**
  $\alpha \leftarrow q.label$;
  $P[\alpha] \leftarrow$ first element of $I[\alpha]$;
end for
$R \leftarrow$ empty list; $\alpha \leftarrow Q[d].label$;
while no index-tree is empty **do**
  `Add`(R, `TwigMatching`(I, Q, i, σ, P));
  $P[\alpha] \leftarrow$ `Next`$(P[\alpha])$;
end while
return R;

Algorithm 2 `Next`(a)

Given an extended pedigree a, the algorithm returns the next extended pedigree in the same index-tree, in the order defined by the prefix traversal, or raises an exception if none exists.

if a has a child **then return** `firstChild`(a);**endif**
while a has a parent and no sibling **do**
  $a \leftarrow$ `parent`(a);
end while
if a has no sibling **then raise** an exception "no more elements available";**endif**
return `sibling`(a);
endif

Matching The function `TwigMatching` (alg. 3) is similar in spirit to the non-deterministic method seen for the existential twig pattern matching, in section 4.1. Instead of "guessing" non deterministically extended pedigrees verifying or disproving a matching, it is using the function `UnboundedSearchInTree` to look for them. There is an additional technicality due to the difference between binary search and doubling search: in each index-tree the search starts where the last search stopped. Those positions are memorized and updated in P, an array of extended pedigrees. It takes an arbitrary order σ on the predicates as a parameter, and follows this order when it has to choose a predicate among several equivalent possibilities. The order is not important, in a randomized version it would be redrawn uniformly at each call to the function. Its complexity is the sum of the complexities of the calls to the function `UnboundedSearchInTree`.

The function `UnboundedSearchInTree` (alg. 4) is moving up and down the index-tree, starting from an extended pedigree b, to search for an extended pedigree b' in a given relation r (parent-child, child-parent, ancestor-descendant, or descendant-ancestor) with a specified extended pedigree a. It starts by going up the index-tree till it finds an extended pedigree on the same branch than a, and then goes down, performing an unbounded search at each level of the

Algorithm 3 `TwigMatching`(Q, σ, P)

Given a query tree Q, an order σ on the predicates of Q, and an array of extended pedigrees P; the algorithm returns the first extended pedigree matching the distinguished predicate of the pattern formed by Q, while ignoring all extended pedigrees preceding the positions indicated by P.

mark "free" all predicates of Q;
mark "bound" σ_1, the first predicate of Q in order σ;
while there is a free predicate in Q, and no index-tree is finished **do**
  $\beta \leftarrow$ the first free predicate among those connected to a bound predicate, in the order defined first by the length of time since the last binding and second by σ;
  $(\alpha, r) \leftarrow$ the bound predicate α connected to β, and the relation r by which it is bound $\alpha \xrightarrow{r} \beta$;
  $a \leftarrow P[\alpha]$; $b \leftarrow P[\beta]$;
  $b \leftarrow$ `UnboundedSearchInTree`(a, r, b);
  if $!(a \xrightarrow{r} b)$ **then** mark all predicates "free"; **endif**
  mark β "bound";
end while
if all predicates are bound **then return** b;
else raise an exception "there is no matching"; **endif**

tree, till it finds b' either in the relation r from a, or proving that such a node does not exist. Refer to the table 1 for the translation of $a \xrightarrow{r} b$ and $a \xrightarrow{r?} b'$ in integer comparisons. Its complexity is the sum of the complexities of the calls to `UnBoundedSearchInArray`.

Algorithm 4 `UnboundedSearchInTree`(a, r, b)

Given the extended pedigrees a and b,
and a relation $r \in \{pc, cp, ad, da\}$;
the algorithm returns the first extended pedigree b' after b in the order defined by the prefix traversal of the index-tree of b, such that either $a \xrightarrow{r} b'$, or $a < b'$.

while $b > 0$ and `parent`$(b) < a$ **do**
  $b \leftarrow$ `parent`(b);
end while
$b \leftarrow$ `UnBoundedSearchInArray`(a, b);
while $!(a \xrightarrow{r} b)$ and $(a \xrightarrow{r?} b)$ **do**
  $b \leftarrow$ `firstChild`(b);
  $b \leftarrow$ `UnBoundedSearchInArray`(a, b);
end while
return b;

4.4 Complexity

We prove here formally the claimed complexity of our algorithm.

Theorem 1. *The function* `Enumerate` *(algorithm 1) performs* $O(\delta hk \log(1 + n/\delta hk))$ *comparisons on an instance of alternation* δ *composed of a distinguished twig pattern of* k *predicates and of a document of size* n*, where no branch contain more than* h *nodes of the same type.*

Note that the condition "no branch contain more than h nodes of same type" is just another way to say that no index-tree is 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 extended pedigree chosen initially or just after a failure or matching, as guessed by a non-deterministic algorithm. By definition, δ is the minimal lenght of such a sequence. For commodity, note a_0 a pedigree preceding any other in the document.

Each unbounded search performed by the algorithm is said to be "in phase i" if the extended pedigree 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 matching, and *negative* otherwise.

There are exactly δ such phases, and in each phases the algorithm performs at most $2k$ calls to the function `UnboundSearchInTree`, as the algorithm will traverse each predicate of the twig pattern only once: if the phase is positive it checks the matching in k unbounded searches, and if the phase is negative, choosing in priority the connected predicate untouched for the longest period of time ensures to cover the whole twig pattern every $2k$ unbounded searches. During each call to `UnboundSearchInTree` the algorithm performs at most h calls to the function `UnboundSearchInArray`, as it is at most going once totally up, and once totally down, and as the index-tree is of height at most h.

For each phase $i \in \{0, \ldots, \delta\}$, for each call to `UnBoundedSearchInTree` $j \in \{1, \ldots, k\}$, and for each of the levels $l \in \{1, \ldots, h\}$ of the corresponding index-tree, note $n_{(i,j,l)}$ the number of elements skipped by the `UnboundedSearchInArray`. Each `UnboundedSearchInArray` performs then $O(\log(1 + n_{(i,j,l)}))$ comparisons, which makes a total of $O(\sum_{i=0}^{\delta-1} \sum_{j=1}^{k} \sum_{l=1}^{h} \log(1 + n_{(i,j,l)}))$.

But this is smaller than $O(\delta hk \log(1+\sum_{i=0}^{\delta-1} \sum_{j=1}^{k} \sum_{l=1}^{h} n_{(i,j,l)/\delta hk}))$, because of the concavity of the function $\log(1 + x)$. But then, as each search starts exactly where the previous one started, $\sum_{i=0}^{\delta-1} \sum_{j=1}^{k} \sum_{l=1}^{h} n_{(i,j,l)} \leq n$, hence the algorithm's complexity is $O(\delta hk \log(1 + n/\delta hk))$. □

Of course, on very specific instances, the difficulty δ can be $O(n)$. As $\delta \log_2(1+ n/\delta) = n$ if $\delta = n$, it means that our algorithm is linear in the worst case on δ, exactly as `PathStack` [4] and other algorithms.

But we prove that our algorithm is better by a finer analysis. Among the instances of small difficulty δ, our algorithm's complexity is logarithmic in the size of the document, while the complexity of algorithms such as `PathStack`, or of any algorithm based on a usual index, is still linear in the size of the document.

For instance, on a document of alternation $\delta = 2$, the algorithm `PathStack` would perform $\Omega(n)$ comparisons to answer the distinguished twig pattern query $B_1/B_2/\ldots/\mathbf{B_{k-1}}/A$, when our algorithm performs $O(k \log(n/\delta k))$.

5 Perspectives

Our work presents several perspectives, on which we wish to turn in the future: We would like to implement the algorithm and to test them on random and real data, to corroborate our theoretical result. Should such simulation prove the interest of our algorithm in practice, it would be nice to have an implementation in OCaml, which could be included as a library to be used with XML processing languages such as Cduce [3].

We claim that the index-tree structure and the associated algorithms can be associated with other techniques to answer twig pattern matching queries using a logarithmic number of comparisons: we will definitely give next a general algorithm proving this claim.

Acknowledgements: The author would like to thank infinitely Laks V S Lakshmanan for his inspiring lectures and numerous encouragements, and Ganesh Ramesh for numerous discussions and an incredible proof-reading. This paper was written while the author was at the University of British Columbia as a postdoctoral fellow.

References

1. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the* 13^{th} *ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 390–399. ACM-SIAM, ACM, January 2002.
2. J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.
3. V. Benzaken, G. Castagna, and A. Frisch. Cduce: a white paper, 2002. PLANX Workshop, Pittsburgh.
4. 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.
5. 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.
6. 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.
7. H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. Timber: A native xml database, 2002.
8. K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, chapter 4.2 Nearly Optimal Binary Search Tree, pages 184–185. Springer-Verlag, 1984.
9. 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.

Appendix

A Deterministic Lower Bound without *up* pointers

Theorem 2. *For any $\delta \geq 1$, $h \geq 1$, $k \geq 2$, $n \geq \delta(h+k)$ any deterministic algorithm for the distinguished twig pattern matching, on instances formed by a twig pattern of $O(k)$ predicates and by a document of $O(n)$ nodes, each branch containing $O(h)$ nodes of the same type, 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 $(\delta = 1, h \geq 1, k = 2, n \geq h+2)$, $(\delta = 1, h \geq 1, k \geq 2, n \geq h+k)$ and $(\delta \geq 1, h \geq 1, k \geq 2, n \geq \delta(k+k)$.

We first consider the twig pattern $\mathbf{B}/A$ of size 2, distinguishing the predicate B, on documents of $n+2$ nodes, such that the root is of type R and has n children of type B, among which one has a child of type A. This forms instances of alternation $\delta = 1$, as guessing the B-node parent of the A-node suffices to certify the matching.

The index-tree for B is an array of n elements among which one only 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 to solve the instance, hence the lower bound for the instances such that $(\delta = 1, h = 1, k = 2, n \geq 3)$.

Next we relax h, the maximal number of nodes of the same type on a branch: this is the maximal height of an index-tree. We consider the query $Q = B/B/\ldots/B/A$ of length $k+1$ with k instances of the predicate B, on documents of size $n+2$, containing nodes of type R, B, A, where the root is an R-node with $n/overh$ children of type B, among which only one has a child; there are $h-1$ B-nodes with $n/overh$ children of type B (and $n-h$ B-nodes without children); there is exactly one A-node, its parent is a B-node, and it is at the level $h+2$. The alternation of this kind of document is always $\delta = 1$, as there is only one possible matching, and guessing the positions of the parents B-nodes is sufficient to certify it. Any deterministic algorithm can be forced to perform $\Omega(hlogn/h)$ comparisons on such instances, by hiding in each array the positions of the B-nodes which are parents, 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 twig pattern. We consider the query $Q = B_1/B_2//B_2/\ldots/B_{k-1}//Bk-1/B_k//B_k/\mathbf{A}$, distinguishing the A-predicate, of length $2k = O(k)$; on documents of size $n+2$, containing nodes of type $R, B_1, \ldots, B_{k-1}, A$, where the root is of type R with $\frac{n}{hk}$ children of type B_1; for each $i \leq k$ there are $h-1$ B_i-nodes with $n/overhk$ children of type B_i; for each $i \leq k-1$ there is exactly one B_i-node with $n/overhk$ children of type B_{i+1}; there is exactly one B_k-node with a unique A child. The figure 6 gives an example of such an instance for $h = 1$. As previously, each branch contains no more than h node of each type, and the alternation of each instance is $\delta = 1$ because there is only one matching, and guessing non-deterministically the positions of each B_i-junction node is sufficient to certify it. By choosing dynamically which nodes of type $B_1, \ldots, B_{k-2}$ have a descendance, an adversary can make any algorithm perform $\Omega(hg \log(n/hk)$ comparisons to check the entire matching, 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 $Q = B_1/B_2//B_2/\ldots/B_{k-1}//Bk-1/B_k//B_k/\mathbf{A}$, distinguishing the A-predicate, of length $2k = O(k)$; on documents formed by δ documents from the previous case as on figure 5. As previously, each branch contains no more than h node of each type (the branch are almost the same than in the last case). The alternation of the instance is exactly δ, as there are δ matchings, and guessing the positions of each B_i-junction-node is sufficient to certify it. Each sub-instance is independent, and an adversary can force any algorithm to perform $\Omega(\delta hk \log(n/\delta hk))$ comparisons, the general lower bound for the instances such that $(\delta \geq 1, h \geq 1, k = 2, n \geq \delta(h + k))$. . $\square$

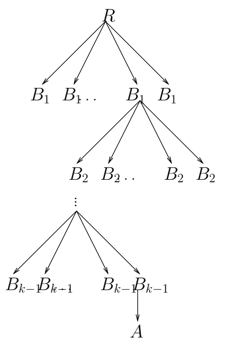

Fig. 4. An instance for the case $(\delta = 1, h = 1, k \geq 2, n \geq k + 1)$.

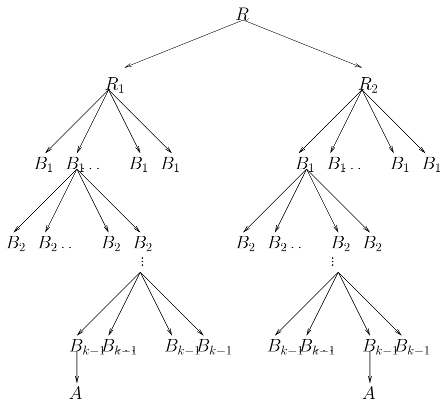

Fig. 5. An instance for the case $(\delta = 2, h = 1, k \geq 2, n \geq 2(k + 1))$.

B Tentative Deterministic Lower Bound with *up* pointers

Proposition 1. *For any $k \geq 2$, $\delta \in \{0, \ldots, \frac{n-1}{2}\}$, any deterministic algorithm for the distinguished twig pattern matching, on instances of alternation at most δ formed by a twig pattern of at most k predicates and by a document of at most n nodes has worst case complexity $\Omega(\delta k \log(n/\delta k)$.*

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

We first consider the twig pattern $\mathbf{B}/A$ of size 2, distinguishing the predicate B; and documents of n nodes, such that the root is of type R, has $n-2$ children of type B, among which one node b, chosen uniformly at random, has a child a, of type A. In these documents each branch contains no more than one node of each type. This forms instances of alternation $\delta = 1$, as guessing the node b, parent of a, suffices to solve each instance. We say that these instances have *parametrisation* $bf\forall n, k = 2, h = 1, \delta = 1$. The index-tree for B is an array of $n-2$ element among which one only is the solution. As each comparison permits only to divide the search space by two, $\log_2(n-2)$ comparisons are necessary to any deterministic algorithm to solve the instance.

Next we relax k, the size of the twig pattern, to consider instances of parametrisation $\forall n, k \geq 2, h = 1, \delta = 1$: . We consider the query $Q = B_1/B_2/\ldots/\mathbf{B_{k-1}}/A$, distinguishing the predicate B_k, and documents of size n, containing nodes of type $R, B_1, \ldots, B_{k-1}, A$, where the root is of type R with $\frac{n-2}{k}$ children of type B_1; exactly one node of type B_i for each $i \in \{1, \ldots, k-2\}$ has $\frac{n-2}{k}$ children of type B_{i+1}; exactly one node of type B_{k-1} has a child of type A. The figure 6 gives an example of such an instance. As previously, in these documents each branch contains no more than one node of each type, hence $h = 1$. Moreover the alternation of the instance is $\delta = 1$, and the twig pattern is of size k. By choosing dynamically which nodes of type $B_1, \ldots, B_{k-2}$ have a descendancy, an adversary can make any algorithm perform as much as $(k-1)\log_2((n-2)/(k-1)$ comparisons to check the entire matching.

Finally we relax the difficulty δ of the instances to get to the parametrisation of our result, $\forall n, k \geq 2, h = 1, \forall\delta$. We consider instances of difficulty δ formed by the same query $Q = B_1/B_2/\ldots/\mathbf{B_{k-1}}/A$, distinguishing the predicate B_k; and documents formed by δ subtrees from the previous paragraph as on figure 5. Each sub-instance is independant, and an adversary can force any algorithm to perform as much as $\delta(k-1)\log_2((n-2\delta-1)/\delta(k-1))$ comparisons.

Hence the lower bound of $\Omega(\delta k \log(n/\delta k))$. □

Note the gap between our lower bound $\Omega(\delta k \log(n/\delta k))$ and our upper bound $O(\delta hk \log(1+n/\delta hk))$. This is due to a surprising consequence of the introduction of *up* pointers in our structure: using those an algorithm can answer a $\mathbf{B}/A$ query on the instance figure 7 using at most $h + \log((n-2)/h)$ comparisons, by picking one leaf among the the last extended pedigree of the array encoding the index-tree, going up the branch till it finds an extended pedigree b of type B, ancestor of the unique node corresponding to the A extended pedigree a, and finding the node parent of the node corresponding to a by a binary search among the $(n-2)/h$ children of b.

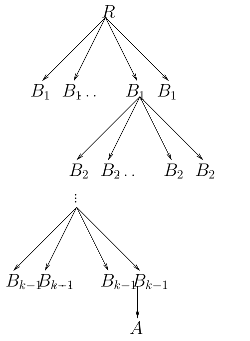

Fig. 6. An instance for the lower bound $\Omega(k \log(n/k))$.

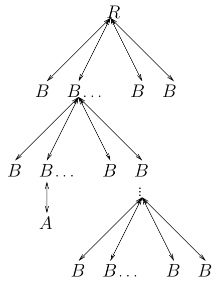

Fig. 7. Using *up* pointers, an algorithm can "cheat".

Without *up* pointers in the index-tree structure, the instances similar to the one of figure 7 would require $(h-1)\log_2(n/(h-1))$ comparisons to be answered, and the complexity of our algorithm would be optimal.

Note that even if the instance presented figure 7 is not standard, it is not clear whether an algorithm cannot take advantage of *up* pointers in more general instances: hence the gap.