Thresholded Debugging of XPath Queries

Jesús M. Almendros-Jiménez
Department of Languages and Computation
University of Almería, Spain
Email: jalmen@ual.es

Alejandro Luna and Ginés Moreno
Department of Computing Systems
University of Castilla-La Mancha, Spain
Email: Alejandro.Luna@alu.uclm.es Gines.Moreno@uclm.es

***Abstract*—We have recently designed/implemented a method for debugging XPath queries which produces a set of alternative XPath expressions with higher chances for retrieving answers from XML files. In this paper we focus on the scalability of our debugger for dealing with massive XML documents by making use of the new command FILTER which is intended to prematurely disregard those computations leading to non significant solutions (i.e., with a poor "chance degree" according to the user's preferences). The key point is the natural capability for performing "dynamic thresholding" enjoyed by the fuzzy logic language used for implementing the tool, which somehow connects with the so-called «top-k answering problem» very well-known in the fuzzy logic and soft computing arenas.**

***Keywords*—XPath, Debugging, Thresholding, Fuzzy Logic Programming, Software for Soft Computing.**

I. Introduction

The eXtensible Markup Language (XML) was born to represent machine readable data by means of semantic tags, which represent an important novelty w.r.t previous markup languages, and it is widely used in many areas of computer software (visit `http://www.w3.org/XML/`). XML provides a very simple language for representing the structure of data, by making use of tags for labeling pieces of textual content, and a tree structure to describe the content in an hierarchical way. XML emerged as a solution to data exchange between applications thanks to the use of tags that permit to locate the content. XML files are largely used in databases, internet, and so on. Moreover, the XPath language [1] was designed as a query language for XML documents in which the path of the tree is used to describe the query. XPath expressions can be adorned with boolean conditions on nodes and leaves to restrict the number of answers to a given query. XPath is the basis of the more powerful query language XQuery designed to join multiple XML documents and to give format to the answer.

On the other hand, fuzzy logic plays too an important role in information retrieval [2], [3], [4], [5], [6], and the need for providing fuzzy/flexible mechanisms to XML querying has recently motivated the investigation of extensions of the XPath language. We can distinguish those in which the main goal is the introduction of fuzzy information in data (similarity, proximity, vagueness, etc) [7], [8], [9], [10], [11] and the proposals in which the main goal is the handling of crisp information by fuzzy concepts [12], [13], [14], [15], [16], [17]. Our work focuses on the second line of research.

In spite of the simplicity of the XPath language, the programmer usually makes mistakes when (s)he describes the path in which the data are allocated. Tipically, (s)he omits some of the tags of the path, s(he) adds more than necessary, and (s)he also uses similar but wrong tag names. When the query does not match to the tree structure of the XML tree, the answer is empty. However, we can also find the case in which the query matches to the XML tree but the answer does not satisfy the programmer. Due to the inherent flexibility of XML documents, the same tag can occurs at several positions, and the programmer could find answers that do not correspond to her (his) expectations. In other words, (s)he finds a correct path, but a wrong answer. We can also consider the case in which a boolean condition is wrong, expressing a wrong range, and several conditions that do not hold at the same time. When the programmer does not find the answer (s)he is looking for, there is a mechanism that (s)he can try to debug the query. In XPath there exists an operator, denoted by '//', that permits to look for the tag from that position. However, it is useless when the tag is present at several positions, since even though the programmer finds answers, (s)he does not know whether they are close to her (his) expectations.

XPath debugging has to take into account the previous considerations. Particularly, there is an underlying notion of *chance degree*. When the programmer makes mistakes, the number of bugs can be higher or lower, and the chance degree is proportional to them. Moreover, there are several ways on which each bug can be solved, and therefore the chance degree is also dependent from the number of solutions for each bug, and the quality of each solution. The quality of a solution describes the number of changes to be made. Finally, there is a case in which we have also focused our work which occurs when the mistake comes from a similar but wrong tag. Here, the chance degree represents the semantic similarity between the tag expressed in the query and the tag which really appears in the XML document.

Our proposed XPath debugging technique is guided by the programmer that initially establishes a value (i.e., a real value between 0 and 1), used by the debugger to penalize bugs in a proportional way. Additionally, we assume that the debugger is equipped with a table of similarities, that is, a mapping between pairs of similar words with an assigned value in the range $[0, 1]$. It makes possible that chance degrees be computed from similarity degrees. The debugger reports a set of annotated paths by using an extended XPath syntax incorporating three annotations: `JUMP`, `SWAP` and `DELETE`. `JUMP` is used to represent that some tags have been added to the original expression, `SWAP` is used to represent that a tag has been changed by another (similar) one and `DELETE` is used to represent that a tag has been removed. Thus, the reported XPath expressions can be seen as an updated version of the original one, such that: case `JUMP` incorporates '//' at the position in which the bug is found; case `SWAP` includes the

Figure 1. Fuzzy Logical Operators

$$
\begin{array}{lcllcll}
\&_{\mathtt{P}}(x,y) & = & x*y & |_{\mathtt{P}}(x,y) & = & x+y-x*y & \textit{Product: and/or} \\
\&_{\mathtt{G}}(x,y) & = & \min(x,y) & |_{\mathtt{G}}(x,y) & = & \max(x,y) & \textit{Gödel: and+/or-} \\
\&_{\mathtt{L}}(x,y) & = & \max(x+y-1,0) & |_{\mathtt{L}}(x,y) & = & \min(x+y,1) & \textit{Łukasiewicz: and-/or+}
\end{array}
$$

new tag; and finally case DELETE removes the wrong tag. Additionally, our proposal permits the programmer to test the reported XPath expressions. The annotated XPath expressions can be executed in order to obtain a ranked set of answers w.r.t. the chance degree. It facilitates the process of debugging because programmers can visualize answers to each query in a very easy way. We have somehow based our debugger in the proposal proposed in [18], [19], where XPath relaxation is studied by giving some rules for query rewriting (axis relaxation, step deletion and step cloning, among others), even when they do not give chance degrees associated to input XPath expressions.

Although our approach can be applied to standard (crisp) XPath expressions, chance degrees in XPath debugging fits well with our proposed framework. Particularly, XPath debugging annotations can be seen as annotations of XPath expressions similary to the proposed DEEP and DOWN of [20], [21], [22]. DEEP and DOWN serve to annotate XPath expressions and to obtain a ranked set of answers depending on they occur, more deeply and from top to down. Each answer is annotated with a *RSV (Retrieval Status Value)* which describes the degree of satisfaction of the answer. Here JUMP, SWAP and DELETE penalize the answers of annotated XPath expressions. When annotated XPath expressions are executed, we obtain a ranked set of answers with respect to the *RSV* of the programmer. DEEP and JUMP have, in fact, the same behavior: JUMP proportionally penalizes answers as deep as they occur. Finally, and in order to cover with SWAP, we have incorporated to our framework similarity degrees.

Our proposal has been implemented/tested and permits programmers to execute/debug the reported XPath expressions by using our tool freely accessible from `http://dectau.uclm.es/fuzzyXPath/`. Both the interpreter and the debugger manage an extension of XPath [20], [21], [22], [23], [24], which uses *fuzzy logic programming* to provide a fuzzy taste to XPath expressions. The implementation has been coded with the so-called «*Multi-Adjoint Logic Programming*» language (MALP in brief) [25] and developed with the «*Fuzzy LOgic Programming Environment for Research*» $\mathcal{FLOPER}$ designed in our research group [26], [27], [28], which can be freely downloaded from `http://dectau.uclm.es/floper/`.

The main goal of the present paper consists in the introduction of a new fuzzy command inside the *Fuzzy-XPath* debugger which comfortably relies on our implementation based on fuzzy logic programming. So, when «[FILTER=r]» precedes a fuzzy query[1], the debugger *lazily* explores an input XML document for dynamically disregarding as soon as possible those branches of the XML tree leading to irrelevant solutions (i.e., with a CD degraded below r), thus allowing the possibility of efficiently managing large files without reducing the set of answers for which users are mainly interested in. Hence, advice that this dynamic thresholding technique embedded into the core of the *Fuzzy-XPath* debugger has two advantages:

- firstly it permits to concentrate on significant answers (i.e., alternative queries which do not excessively deviate from the original one) without disturbing the attention with useless information, and
- secondly, the computational behavior of the debugging process is highly improved (both in time and space) since a great amount of work is avoided when discriminating useless branches of the XML tree.

The structure of the paper is as follows. After summarizing in Section II our fuzzy extension of XPath [20], [21], [22], in Section III we describe our debugging technique initially presented in [23], [24]. Next, some implementation details are explained in Section IV, paying especial attention to the use of the FILTER command in the *Fuzzy-XPath* interpreter and debugger for performing dynamic thresholding in order to improve the efficiency of both tools. Finally, we conclude in Section V by also sketching some lines of further research.

II. Fuzzy XPath

In this section we summarize the main elements of our proposed fuzzy XPath language described in [20], [21], [22]. We firstly incorporate two structural constraints called DOWN and DEEP to which a certain degree of relevance is associated. So, whereas DOWN provides a ranked set of answers depending on the path they are found from "top to down" in the XML document, DEEP provides a ranked set of answers depending on the path they are found from "left to right" in the XML text. Both structural constraints can be used together, assigning importance's degrees with respect to the distance of nodes to the root XML element.

Secondly, our fuzzy XPath incorporates fuzzy variants of *and* and *or* for XPath conditions. Crisp *and* and *or* operators are used in standard XPath over boolean conditions, and enable to impose boolean requirements on the answers. XPath boolean conditions can be referred to attribute values and node content, in the form of equality and range of literal values, among others. However, the *and* and *or* operators applied to two boolean conditions are not precise enough when the programmer does not give the same value to both conditions. For instance, some answers can be discarded when they could be of interest by the programmer, and accepted when they are not relevant. Besides this, programmers would need to know in which sense a solution is better than another. When several boolean conditions are imposed on a query, each one contributes to satisfy the programmer's preferences in a different way and perhaps, the level of satisfaction is distinct for each solution.

[1]This command was initially implemented inside the *Fuzzy-XPath* interpreter in [29], but it is important to note that in the current paper we focus on the *Fuzzy-XPath* debugger.

We have enriched the arsenal of operators of XPath with fuzzy variants of *and* and *or*. Particularly, we have considered three versions of *and*: *and+*, *and*, *and-* (and the same for *or* : *or+*, *or*, *or-*) which make more flexible the composition of fuzzy conditions. Three versions for each operator that come for free from our adaptation of fuzzy logic to the XPath paradigm. One of the most known elements of fuzzy logic is the introduction of fuzzy versions of classical boolean operators. *Product*, *Łukasiewicz* and *Gödel* fuzzy logics are considered as the most prominent logics and give a suitable semantics to fuzzy operators (see Figure 1). Our contribution is now to give sense to fuzzy operators into the XPath paradigm, and particularly in programmer's preferences. We claim that in our work the fuzzy versions provide a mechanism to force (and debilitate) conditions in the sense that stronger (and weaker) programmer preferences can be modeled with the use of stronger (and weaker) fuzzy conditions. The combination of fuzzy operators in queries permits to specify a ranked set of fuzzy conditions according to programmer's requirements.

Furthermore, we have equipped XPath with an additional operator that is also traditional in fuzzy logic: the average operator *avg*. This operator offers the possibility to explicitly give weight to fuzzy conditions. Rating such conditions by *avg*, solutions increase its weight in a proportional way. However, from the point view of the programmer's preferences, it forces the programmer to quantify his(er) wishes which, in some occasions, can be difficult to measure. For this reason, fuzzy versions of *and* and *or* are better choices in some circumstances. Finally, we have equipped our XPath based query language with a mechanism for thresholding programmer's preferences, in such a way that programmer can request that requirements are satisfied over a certain percentage. The proposed fuzzy XPath is described by the following syntax:

xpath	:=	['**[**'deep-down'**]**']path
path	:=	literal | text() | node | @att | node**/**path | node**//**path
node	:=	QName | QName[cond]
cond	:=	xpath op xpath | xpath num-op number
deep	:=	**DEEP**=number
down	:=	**DOWN**=number
deep-down	:=	deep | down | deep '**;**' down
num-op	:=	**>** | **=** | **<** | **<>**
fuzzy-op	:=	**and** | **and+** | **and-** | **or** | **or+** | **or-** | **avg** | **avg**{num, num}
op	:=	num-op | fuzzy-op

Basically, our proposal extends XPath as follows:

• **Structural constraints**. A given XPath expression can be adorned with «[DEEP = r_1; DOWN = r_2]» which means that the *deepness* of elements is penalized by r_1 and that the *order* of elements is penalized by r_2, and such penalization is proportional to the distance (i.e., the length of the branch and the weight of the tree, respectively). In particular, «[DEEP = 1; DOWN = r_2]» can be used for penalizing only w.r.t. document order. DEEP works for //, that is, the deepness in the XML tree is only computed when descendant nodes are explored, while DOWN works for both / and //. Let us remark that DEEP and DOWN can be used several times on the main *path* expression and/or any other *sub-path* included in conditions.

• **Flexible operators in conditions**. We consider three fuzzy versions for each one of the classical conjunction and disjunction operators (also called connectives or aggregators) describing *pessimistic*, *realistic* and *optimistic* scenarios, see Figure 1. In XPath expressions the fuzzy versions of the connectives make harder to hold boolean conditions, and therefore can be used to debilitate/force boolean conditions. Furthermore, assuming two given *RSV*'s r_1 and r_2, the *avg* operator is obviously defined with a fuzzy taste as $(r_1+r_2)/2$, whereas its *priority-based* variant, i.e. $avg\{p_1, p_2\}$, is defined as $(p_1 * r_1 + p_2 * r_2)/p_1 + p_2$.

Consider now the following XML document for our examples:

```
<bib>
    <name>Classic Literature</name>
    <book year="2001" price="45.95">
        <title>Don Quijote de la Mancha</title>
        <author>Miguel de Cervantes Saavedra</author>
        <references>
            <novel year="1997" price="35.99">
                <name>La Galatea</name>
                <author>Miguel de Cervantes Saavedra</author>
                <references>
                    <book year="1994" price="25.99">
                        <title>Los trabajos de Persiles y Sigismunda</title>
                        <author>Miguel de Cervantes Saavedra</author>
                    </book>
                </references>
            </novel>
        </references>
    </book>
    <novel year="1999" price="25.65">
        <title>La Celestina</title>
        <author>Fernando de Rojas</author>
    </novel>
</bib>
```

In general, a fuzzy XPath expression defines, w.r.t. an XML document, a sequence of subtrees of the XML document where each subtree has an associated RSV. XPath conditions, which are defined as fuzzy operators applied to XPath expressions, compute a new RSV from the RSVs of the involved XPath expressions, which at the same time, provides a RSV to the node.

Let us firstly consider the leftmost query in Figure 2 which requests *title*'s but penalizing the occurrences from the document root by a proportion of 0.8 and 0.9 by nesting and ordering, respectively, and for which we obtain the file listed in Figure 2. In this document we have included as attribute of each subtree, its corresponding RSV. The highest RSVs correspond to the main *books* of the document, and the lowest RSVs represent the *books* occurring in nested positions (those annotated as related *references*). On the other hand, in the right side of Figure 2, it is shown the answer associated to a search of books, possibly referenced *directly or indirectly* from other books, whose publishing year and price are relevant but the year is three times more important than the price. Finally, if we combine both kinds of (structural/conditional) operators in the new query «/bib[DEEP=0.5]//book[@year<2000 avg{3,1} @price<50]/title», we obtain as output this XML document:

```
<result>
  <title rsv="0.25">Don Quijote de la Mancha</title>
  <title rsv="0.0625">Los trabajos de Persiles y Sigismunda</title>
</result>
```

where it is easy to see that the ranked list of solutions is reversed and now "*Don Quijote*" is not penalized with DEEP.

Figure 2. Executing two *Fuzzy-XPath* queries which make use of different fuzzy commands

```
QUERY:          «/bib[DEEP=0.8;DOWN=0.9]//title»

                <result>
                  <title rsv="0.8000">Don Quijote de la Mancha</title>
OUTPUT FILE:      <title rsv="0.7200">La Celestina</title>
                  <title rsv="0.2949">Los trabajos de Persiles y Sigismunda</title>
                </result>\\
```

```
QUERY:          «//book[@year<2000 avg{3,1} @price<50]/title>

                <result>
OUTPUT FILE:      <title rsv="1.00">Los trabajos de Persiles y Sigismunda</title>
                  <title rsv="0.25">Don Quijote de la Mancha</title>
                </result>
```

III. DEBUGGING XPATH

In this section we recast from [23], [24] our proposed technique for debugging XPath expressions. The debugger accepts as inputs a query Q preceded by the [DEBUG=r] command (for instance, «[DEBUG=0.5]/bib/book/title»), where r is a real number in the unit interval. Assuming an input XML document like the one depicted in the previous section, the debugging produces a set of alternative queries $Q_1, ..., Q_n$ packed into an output XML document with the following structure (see also Figure 3):

```
<result>
<query cd="r1" attributes1> Q1 </query>
...
<query cd="rn" attributesn> Qn </query>
</result>
```

where the set of alternatives is ordered with respect to the CD key. This value measures the chance degree of the original query with respect to the new one, in the sense that as more changes are performed on Q_i and as more *traumatic* they are with respect to Q, then the CD value becomes lower.

In the left column of Figure 3, the first alternative, with the highest CD, is just the original query, thus, the CD is *1*, whose further execution should return «Don Quijote de La Mancha». From now on, we show that our debugger runs even when the set of answers is not empty, like in this case. The remaining options give different CD's depending on the chance degree, and provide XPath expressions annotated with JUMP, DELETE and SWAP commands.

In order to explain the way in which our technique generates the attributes and content of each $query$ tag in the output XML document, let us consider a generic path Q of the form: «[DEBUG=r]$/tag_1/.../tag_i/tag_{i+1}/...$», where we say that tag_i is at level i in the original query. So, assume that when exploring the input query Q and the XML document D, we find that tag_i in Q does not occurs at level i in (a branch of) D. Then, we consider the following three situations:

Swapping case: Instead of tag_i, we find tag'_i at level i in the input XML document D, being tag_i and tag'_i two similar terms with similarity degree s. Then, we generate an alternative query by adding the attribute tag_i ="tag'_i" and replacing in the original path the occurrence "$tag_i/$" by "[SWAP=s]$tag'_i/$". The second query proposed in the left column of Figure 3 illustrates this case, that is: « <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query> ». Let us observe that : 1) we have included the attribute «book="novel"» in order to suggest that instead of looking now for a *book*, finding a *novel* should be also a good alternative, 2) in the path we have replaced the tag *book* by *novel* and we have appropriately annotated the exact place where the change has been performed with the annotation [SWAP=0.8] and 3) the CD of the new query has been adjusted with the *similarity degree* 0.8 of the exchanged tags. Now, it is possible to launch with our *Fuzzy-XPath* interpreter mentioned before, the execution of the (fuzzy) XPath queries «/bib/novel/title» and «/bib/[SWAP=0.8]novel/title». In both cases we obtain the same result, i.e., «La Celestina» but with different RSV (or *Retrieval Status Value* [20], [21]) in each case: 1 and 0.8, respectively.

Jumping case: Even when tag_i is not found at level i in the input XML document D, tag_{i+1} appears at a deeper level (i.e., greater than i) in a branch of D. Then, we generate an alternative query by adding the attribute tag_i*="//"*, which means that tag_i has been jumped, and replacing in the path the occurrence *"tag_i/"* by *"[JUMP=r]//"*, being r the value associated to DEBUG. This situation is illustrated by the third and fourth queries in the left column of Figure 3, where we propose to jump tags *book* and *bib*. The execution of the queries returns different results, since for «/bib/[JUMP=0.5]//title» and «/[JUMP=0.5]//book/title» we obtain respectively the following pair of documents:

```
<result>
  <title rsv="0.5">Don Quijote de la Mancha</title>
  <title rsv="0.5">La Celestina</title>
  <title rsv="0.03125">Los trabajos de Persiles y Sigismunda</title>
</result>
```

```
<result>
  <title rsv="0.5">Don Quijote de la Mancha</title>
  <title rsv="0.03125">Los trabajos de Persiles y Sigismunda</title>
</result>
```

where we can see that, as more tags are jumped, their resulting values become lower and lower.

Deletion case: This scenario emerges when at level i in the input XML document D, we found tag_{i+1} instead of tag_i. So, the intuition tell us that tag_i should be removed from the original query Q and hence, we generate an alternative query by adding the attribute tag_i*=""* and replacing in the path the occurrence *"tag_i/"* by *"[DELETE=r]"*, being r the value associated to DEBUG. This situation is illustrated by the fifth query in Figure 3, where the deletion of the tag *book* is followed by a swapping of similar tags *title* and *name*. The CD *0.45* associated to this query is defined as the *product* of the values associated to both DELETE (*0.5*) and SWAP (*0.9*), and hence the chance degree of the original one is lower than the previous examples. So, the execution of query «/bib/[DELETE=0.5][SWAP=0.9]name», should produce the XML-based output: `<result> <name rsv="0.45">Classic Literature</name> </result>` . As we have seen in the previous example, the combined use of one or more debugging commands (SWAP, JUMP and DELETE) is not only allowed but also frequent. In other words, it is possible to find several debugging points. One more example: the execution of query «/[DELETE=0.5][JUMP=0.5]//[SWAP=0.9]name» produces:

```
<result>
  <name rsv="0.225">Classic Literature</name>
  <name rsv="0.028125">La Galatea</name>
</result>
```

Figure 3. Debugging two *Fuzzy-XPath* queries with or without repetitions of the DEBUG command

```
QUERY: «[DEBUG=0.5]/bib/book/title»

OUTPUT FILE:

<result>
  <query cd="1.0">/bib/book/title</query>
  <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query>
  <query cd="0.5" book="//">/bib/[JUMP=0.5]//title</query>
  <query cd="0.5" bib="//">/[JUMP=0.5]//book/title</query>
  <query cd="0.45" book="" title="name">/bib/[DELETE=0.5][SWAP=0.9]
                    name</query>
  <query cd="0.4" bib="//" book="novel">/[JUMP=0.5]//[SWAP=0.8]novel/
                    title</query>
  <query cd="0.25" book="" title="//">/bib/[DELETE=0.5][JUMP=0.5]//
                    title</query>
  <query cd="0.25" book="//" book="">/bib/[JUMP=0.5]//[DELETE=0.5]title</query>
  <query cd="0.25" bib="" book="//">/[DELETE=0.5][JUMP=0.5]//book/title</query>
  <query cd="0.25" bib="//" book="//">/[JUMP=0.5]//[JUMP=0.5]//title</query>
  <query cd="0.25" bib="//" bib="">/[JUMP=0.5]//[DELETE=0.5]book/title</query>
  <query cd="0.225" title="//" title="//" title="name">/bib/book/[JUMP=0.5]//
                              [JUMP=0.5]//[SWAP=0.9]name</query>
  <query cd="0.225" bib="" book="//" title="name">/[DELETE=0.5][JUMP=0.5]//
                              [SWAP=0.9]name</query>
  <query cd="0.225" bib="//" book="" title="name">/[JUMP=0.5]//[DELETE=0.5]
                              [SWAP=0.9]name</query>
  <query cd="0.2" bib="" book="//" book="novel">/[DELETE=0.5][JUMP=0.5]//
                              [SWAP=0.8]novel/title</query>
    .........
</result>
```

```
QUERY: «[DEBUG=0.7]/bib/[DEBUG=0.6]book/[DEBUG=0.5]title»

OUTPUT FILE:


<result>
  <query cd="1.0">/bib/book/title</query>
  <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query>
  <query cd="0.7" bib="//">/[JUMP=0.7]//book/title</query>
  <query cd="0.6" book="//">/bib/[JUMP=0.6]//title</query>
  <query cd="0.56" bib="//" book="novel">/[JUMP=0.7]//[SWAP=0.8]novel/title
                         </query>
  <query cd="0.54" book="" title="name">/bib/[DELETE=0.6][SWAP=0.9]name
                         </query>
  <query cd="0.42" bib="" book="//">/[DELETE=0.7][JUMP=0.6]//book/title
                         </query>
  <query cd="0.42" bib="//" book="//">/[JUMP=0.7]//[JUMP=0.6]//title</query>
  <query cd="0.378" bib="" book="//" title="name">
                         /[DELETE=0.7][JUMP=0.6]//[SWAP=0.9]name</query>
  <query cd="0.378" bib="//" book="" title="name">
                         /[JUMP=0.7]//[DELETE=0.6][SWAP=0.9]name</query>
  <query cd="0.336" bib="" book="//" book="novel">
                         /[DELETE=0.7][JUMP=0.6]//[SWAP=0.8]novel/title</query>
  <query cd="0.3" book="" title="//">/bib/[DELETE=0.6][JUMP=0.5]//title</query>
  <query cd="0.2646" bib="//" bib="" book="" title="name">
                         /[JUMP=0.7]//[DELETE=0.7][DELETE=0.6][SWAP=0.9]name
                         </query>
    .........
</result>
```

This query, which was listed in Figure 3 as <query cd="0.225" bib="" book="//" title="name">/[DELETE=0.5] [JUMP=0.5] [SWAP=0.9] name</query>, contains several changes on its body (w.r.t. the original goal), and hence its final CD *0.225* is quite low since it has been obtained by multiplying the three values associated to the deletion of the tag *bib* (*0.5*), jumping the tag *book* (*0.5*) and the swapping of *title* by *name* (*0.9*).

We would like to remark that even when we have worked with a very simple query with three tags in our examples, our technique works with more complex queries with large paths and connectives in conditions, as well as DEBUG used in several places on the query. For instance, in the right column of Figure 3 (compare it with the left column) we show the result of debugging the following query: «[DEBUG=0.7]/bib/[DEBUG=0.6]book/[DEBUG=0.5]title».

Finally, it is important to note that the wide range of alternatives proposed by our technique (Figure 3 is still incomplete), reveals its high level of flexibility: programmers are free to use the alternative queries to execute them, and to inspect results up to their intended expectations. However, since many of the alternatives have a very low CD, their meanings and shapes have been deviated far away from the original query and they seem to be useless for the user. This fact justifies the technique we propose in the next section in order to avoid the waste of computational resources for generating them.

IV. Dynamic Filters for the Thresholded Debugging of Queries

In [30], [31] we have reported some *thresholding* techniques specially tailored for the MALP language, where the main idea consists in to dynamically create and evaluate filters for prematurely disregarding those superfluous computations leading to non-significant solutions. Somehow inspired by the same guidelines, in [29] we have recently equipped our *Fuzzy-XPath* interpreter with a new command with syntax «[FILTER=r]» (being r a real number between 0 and 1) which can be used just at the beginning of a query for indicating that only those answers with RSV greater of equal than r must be generated and reported. In the present work, we show the benefits of using the same command that we have just implemented into the *Fuzzy-XPath* debugger too (now, only alternative queries to a given one whose CDs are greater of equal than r must be generated during the debugging process).

So, if we execute a *Fuzzy-XPath* query with the following form «[FILTER=0.4]//book[@year<2000 avg @price<50]/title», we obtain nine answers, but only five if we fix «[FILTER=0.8]». Obviously, we would hope that the runtime of the second case should be lower than the first one since, as our approach does, there is no need for computing all solutions and then filtering the best ones. This desired dynamic behaviour when avoiding useless computations is reflected in Figure 4 which considers the effort needed for executing (up) and debugging[2] (down) a query like «[FILTER=r]//book[(@price>25 and @price<30) avg (@year<2000 or @year>2006)]» where each row represents the size of several XML files accomplishing with the same structure of our running example (but considering different nesting levels of tags *book*, *title*, *author* and *references*), and each column refers to a different degree of the FILTER command. Here, the runtime is measured in seconds excluding the extra parsing/compiling time (the benchmarks have been performed using a computer with processor Intel Core Duo, with 2 GB RAM and Windows Vista) and each record in the input file refers to a different *book* (that is, the number of records coincides with the number of occurrences of tag *book*) which might contain other books inside its *references* tag. The size of the files in the figure moves from 323Kb (1000 records) till 3223 Kb (10000 records), but our application works fine even when files of 33Mb (100000 records), which reveals the interest of our results.

[2]In this last case we have used «[DEBUG=0.9]» just after «FILTER».

Figure 4. Performance of the *Fuzzy-XPath* interpreter (up) and debugger (down) by using FILTER on XML files with growing sizes

Records	**FILTER** (*Fuzzy-XPath* interpreter)								
	0.1	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
1000	1.766	1.696	1.734	0.842	0.469	0.268	0.221	0.087	0.056
2000	6.628	6.432	6.998	3.242	1.439	0.677	0.599	0.168	0.122
3000	14.532	14.023	14.059	6.306	2.831	1.257	1.101	0.253	0.179
4000	25.535	24.684	24.722	10.883	4.827	1.918	1.794	0.345	0.242
5000	41.522	37.782	37.166	16.201	7.242	2.993	2.516	0.427	0.281
6000	58.905	55.354	55.596	24.411	10.993	4.207	3.554	0.554	0.373
7000	85.167	85.652	82.733	37.748	14.436	5.083	4.653	0.649	0.460
8000	137.737	102.816	102.763	69.401	26.680	8.273	5.894	0.690	0.481
9000	175.272	131.828	131.021	56.937	22.601	7.869	7.329	0.824	0.549
10000	195.613	185.201	167.676	95.286	26.649	9.516	9.595	0.973	0.742

Records	**FILTER** (*Fuzzy-XPath* debugger)								
	0.1	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
1000	2.857	0.443	0.341	0.381	0.340	0.386	0.349	0.394	0.295
2000	5.833	0.951	0.777	0.794	0.707	0.827	0.716	0.803	0.596
3000	9.422	1.411	1.059	1.243	1.053	1.251	1.100	1.233	0.881
4000	11.742	1.800	1.405	1.597	1.422	1.592	1.463	1.595	1.202
5000	15.646	2.466	1.735	1.786	1.931	1.758	2.143	1.771	1.500
6000	19.315	2.723	2.115	2.522	2.111	2.540	2.112	2.500	1.802
7000	22.599	3.397	2.505	3.025	2.475	2.468	2.783	2.442	2.561
8000	24.234	3.595	2.852	3.115	2.836	3.173	2.857	3.219	2.374
9000	30.305	3.137	4.212	3.184	5.072	3.169	3.174	3.811	2.675
10000	33.329	4.942	3.543	3.573	3.878	3.559	4.211	3.518	2.962

Moreover, in Figure 5 we continue with a similar query to the previous one, but also considering the `DEEP` command[3]. Here, for a large XML document with a fixed size, we express the number of seconds needed for executing such query when varying `FILTER` and `DEEP`, where it is easy to see that the behaviour of the interpreter is more and more improved whenever `FILTER` grows and `DEEP` decreases, as wanted. On the other hand, Figure 6 reflects similar effects that occur in our debugger, where the contrast now is established between the `FILTER` and `DEBUG` commands.

Although the core of our application is written with (fuzzy) MALP rules, our implementation is based on the following items:

(1) We have reused/adapted several modules of our previous PROLOG-based implementation of (crisp) XPath described in [32], [33].

(2) We have used the SWI-PROLOG library for loading XML files, in order to represent a XML document by means of a PROLOG term (the notion of *term*, i.e. data structure, is just the same in MALP and PROLOG).

(3) The parser of XPath has been extended to recognize the new keywords `FILTER`, `DEBUG`, `DEEP`, `DOWN`, `JUMP`, `avg`, etc... with their proper arguments.

(4) Each tag is represented as a data-term of the form: *element(Tag,Attributes,Subelements)*, where *Tag* is the name of the XML tag, *Attributes* is a PROLOG list containing the attributes, and *Subelements* is a PROLOG list containing the sub-elements (i.e. subtrees) of the tag. For instance, the XML document used in our examples is represented in SWI-PROLOG like:

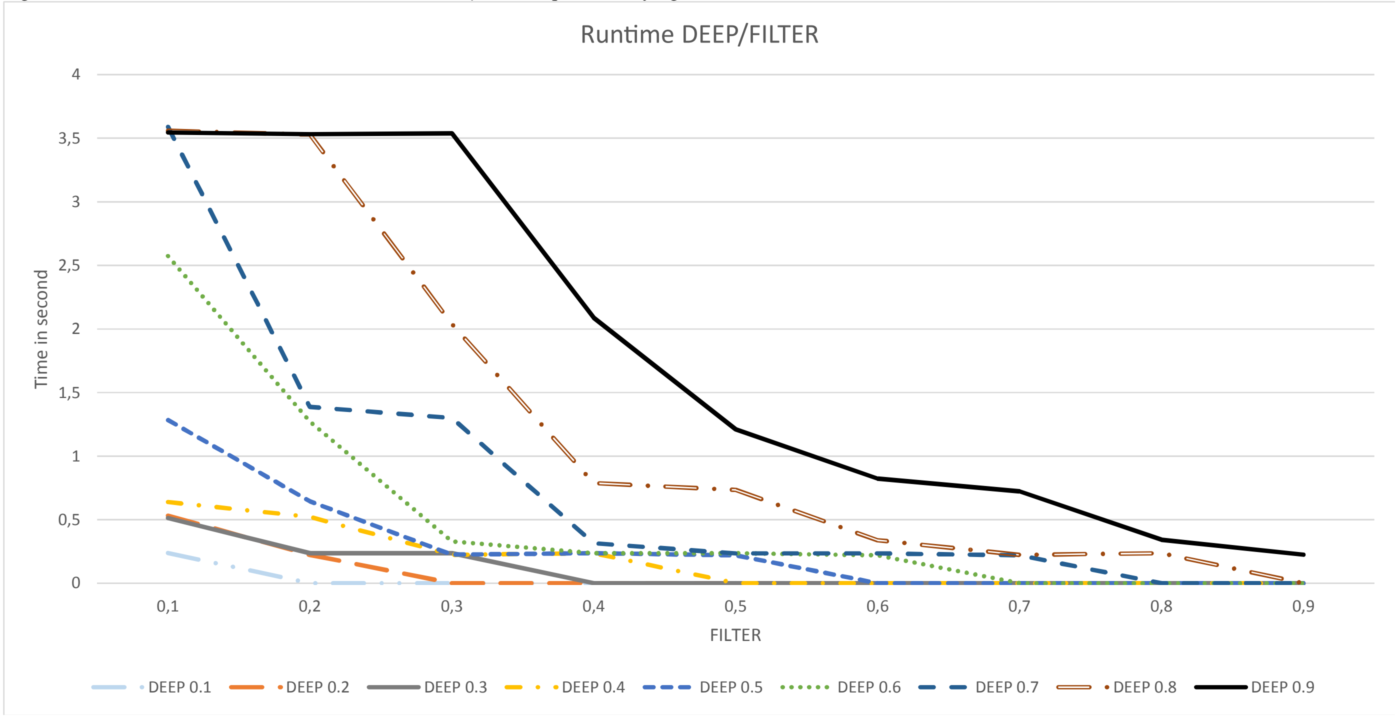
```
[ element(bib,[],
    [ element(name,[],['Classic Literature']),
      element(book,[year='2001',price='45.95'],
        [ element(title,[],['Don Quijote de la Mancha']),
          element(author,[],['Miguel de Cervantes Saavedra']),
          element(references,[],
            [ element(novel,[year='1997',price='35.99'],
                [ element(name,[],['La Galatea']),
                  element(author,[],[ 'Miguel de Cervantes Saavedra']),
                  element(references,[],
                    [ element(book,[ year = '1994', price = '25.99'],
                        [ element(title,[],[ 'Los trabajos de Persiles y
                                 Sigismunda']),
                          element(author,[],[ 'Miguel de Cervantes
                                 Saavedra'])
                        ])
                    ])
                ])
            ])
        ]),
      element(novel, [year='1999',price='25.65'],
        [ element(title,[],['La Celestina']),
          element(author,[],['Fernando de Rojas'])
        ])
    ])
]
```

Loading of documents is achieved by predicate *load_xml(+File,-Term)* and writing by predicate *write_xml(+File,+Term)*.

(5) Predicate *fuzzyXPath(+ListXPath,+Tree,+Deep,+Down,+Filter,+Accum)* receives six arguments: (1) *ListXPath* is the PROLOG representation of a XPath expression; (2) *Tree* is the term representing an input XML document; (3) *Deep/Down/Filter* have the obvious meaning, and finally (4) the last argument *Accum* (which is appropriately updated -maybe decreased- when going deeper in the exploration of the

[3]This kind of statistics can be produced on-line for several XML files and *Fuzzy-XPath* queries via the following couple of URLs that we have just prepared for the interested reader: `http://dectau.uclm.es/fuzzyXPath/?q=FuzzyXPathStatistics` and `http://dectau.uclm.es/fuzzyXPath/?q=DebugStatistics` (regarding the interpreter and the debugger, respectively).

Figure 5. Runtime for the execution of several *Fuzzy-XPath* queries varying DEEP and FILTER



file) accumulates the sequence of penalties produced till reaching a concrete node, and it is very useful for deciding when performing a recursive call to the children of such node whenever the value of *Accum* is better than the one fixed by *Filter*. These actions also directly revert on the new predicate *debugQuery (+ListXPath,+Tree,+Debug,+Filter,+Accum)*, which implements the ideas described in this work and where once again the appropriate management of the two last arguments becomes the keypoint of our dynamic thresholding technique. Both the interpreter and the debugger can be tested on-line via `http://dectau.uclm.es/fuzzyXPath/?q=FuzzyXPathTest` and `http://dectau.uclm.es/fuzzyXPath/?q=DebuggerXPathTest`, respectively (see Figure 7).

(6) The evaluation of the query generates a *truth value* which has the form of a tree, called *tv tree*. The main power of a fuzzy logic programming language like MALP w.r.t. PROLOG, is that instead of answering questions with a simple *true/false* value, solutions are reported in a much more tinged, documented way. Basically, the *fuzzyXPath* predicate traverses the PROLOG tree representing a XML document annotating into the *tv tree* the corresponding DEEP/DOWN values according to the movements performed in the horizontal and vertical axis, respectively. In addition, the *tv tree* is annotated with the values of *and*, *or* and *avg* operators in each node. For instance, the evaluation of the query «/bib[DEEP=0.8]//book[@year<2000 avg{3,1} @price<50]/title» generates the following tv:

```
tv(1.0,[[],
    tv(1.0,[[tv(0.25, [])],
        tv(1.0,[element(title, [], [Don Quijote de la Mancha]), [], []]),
    tv(1.0,[[],
        tv(0.8,[[],
            tv(0.8,[[],
                tv(0.8,[[],
                    tv(0.8,[[tv(1.0, [])],
                        tv(1.0,[element(title, [], [Los trabajos de ...
                            ....
```

For the case of the *debugQuery* predicate, it explores the XML tree in a very similar way than the interpreter, but now the annotations performed on the resulting *tv tree* refer to the corresponding JUMP, DELETE and SWAP values. For instance, the *tv tree* associated to query «[DEBUG=0.5]//book/title» starts as:

```
tv(1.0,[[//, []],
    tv(0.0,[[tag(noExist), []], [],
    tv(0.5,[[[DELETE=0.5], [book=]],
        tv(0.0,[[tag(noExist), []], [],
        tv(0.0,[[tag(noExist), []], [],
        tv(0.0,[[tag(noExist), []], [],
        tv(0.5,[[[JUMP=0.5]//, [title= //]],
            tv(0.9,[[tag(name), [title=name]], [],
            tv(0.0,[[tag(noExist), []], [],
            tv(0.0,[[tag(noExist), []], [], ...
                ....
```

(7) Finally, the *tv tree* is used for computing the output of the query, by multiplying the recorded values. A predicate called *tv_to_elem* has been implemented to output the answers in a sorted, pretty way.

V. Conclusions and Future Work

While in [20], [21], [22], [34] we have enriched XPath with new constructs (both structural -DEEP and DOWN- and constraints -avg and alternative fuzzy versions of classical or/and operators-) in order to flexibly query XML documents,

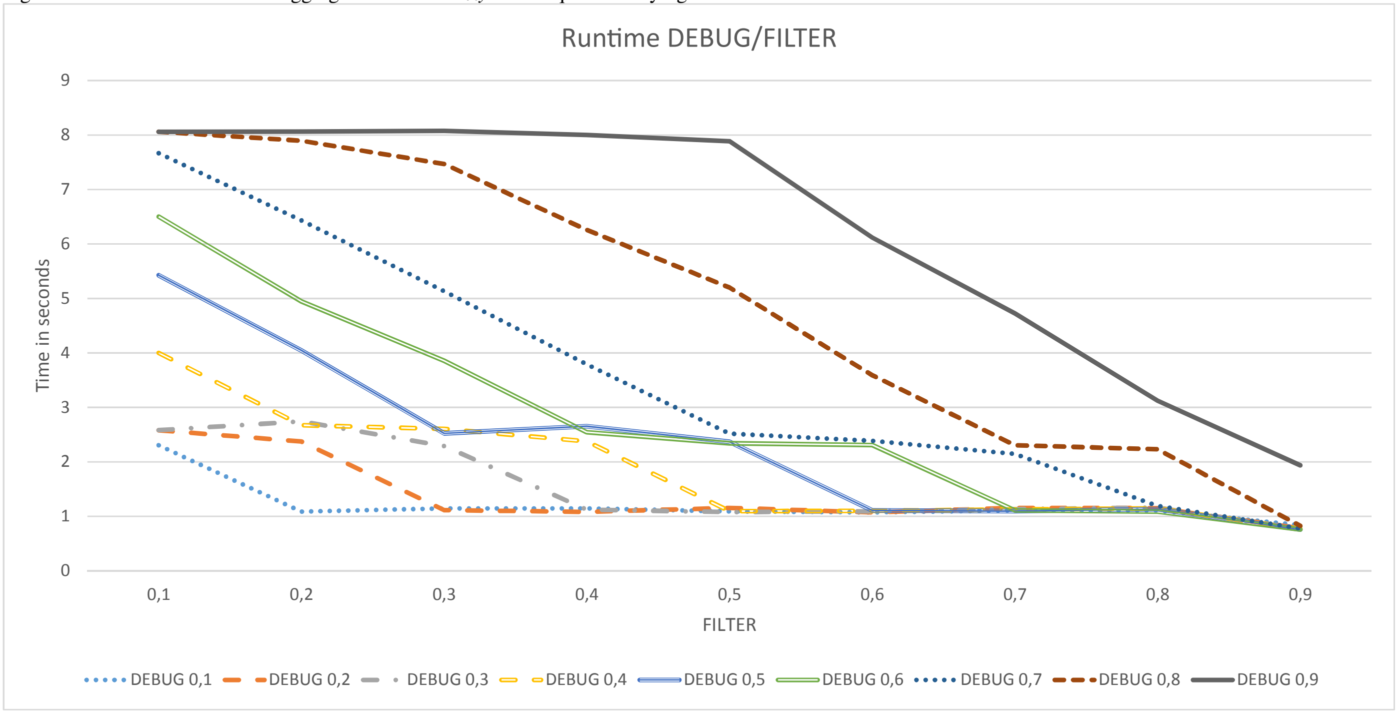
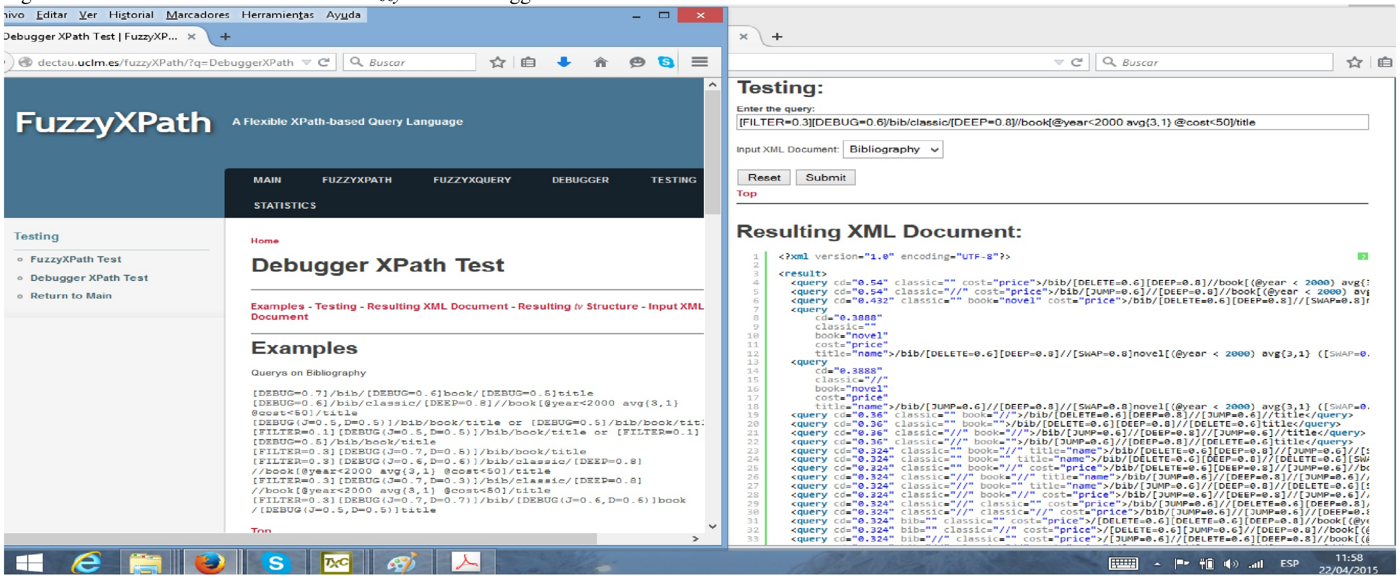


Figure 6. Runtime for the debugging of several *Fuzzy-XPath* queries varying `DEBUG` and `FILTER`


ivo Editar Ver Historial Marcadores Herramientas Ayuda
Debugger XPath Test | FuzzyXP...
dectau.uclm.es/fuzzyXPath/?q=DebuggerXPath
Buscar
MAIN
FUZZYXPATH
FUZZYXQUERY
DEBUGGER
TESTING
STATISTICS
Reset
Submit
Top
ESP
11:58
22/04/2015

FuzzyXPath A Flexible XPath-based Query Language

Testing

- FuzzyXPath Test
- Debugger XPath Test
- Return to Main

Home

Debugger XPath Test

Examples - Testing - Resulting XML Document - Resulting *tv* Structure - Input XML Document

Examples

Querys on Bibliography

```
[DEBUG=0.7]/bib/[DEBUG=0.6]book/[DEBUG=0.5]title
[DEBUG=0.6]/bib/classic/[DEEP=0.8]//book[@year<2000 avg{3,1}
@cost<50]/title
[DEBUG(J=0.5,D=0.5)]/bib/book/title or [DEBUG=0.5]/bib/book/tit
[FILTER=0.1][DEBUG(J=0.5,D=0.5)]/bib/book/title or [FILTER=0.1]
[DEBUG=0.5]/bib/book/title
[FILTER=0.3][DEBUG(J=0.7,D=0.5)]/bib/book/title
[FILTER=0.3][DEBUG(J=0.6,D=0.6)]/bib/classic/[DEEP=0.8]
//book[@year<2000 avg{3,1} @cost<50]/title
[FILTER=0.3][DEBUG(J=0.7,D=0.3)]/bib/classic/[DEEP=0.8]
//book[@year<2000 avg{3,1} @cost<50]/title
[FILTER=0.3][DEBUG(J=0.7,D=0.7)]/bib/[DEBUG(J=0.6,D=0.6)]book
/[DEBUG(J=0.5,D=0.5)]title
```

Testing:

Enter the query:
[FILTER=0.3][DEBUG=0.6]/bib/classic/[DEEP=0.8]//book[@year<2000 avg{3,1} @cost<50]/title

Input XML Document: Bibliography

Resulting XML Document:

```
<?xml version="1.0" encoding="UTF-8"?>

<result>
  <query cd="0.54" classic="" cost="price">/bib/[DELETE=0.6][DEEP=0.8]//book[(@year < 2000) avg{
  <query cd="0.54" classic="//" cost="price">/bib/[JUMP=0.6]//[DEEP=0.8]//book[(@year < 2000) avg
  <query cd="0.432" classic="" book="novel" cost="price">/bib/[DELETE=0.6][DEEP=0.8]//[SWAP=0.8]r
  <query
      cd="0.3888"
      classic=""
      book="novel"
      cost="price"
      title="name">/bib/[DELETE=0.6][DEEP=0.8]//[SWAP=0.8]novel[(@year < 2000) avg{3,1} ([SWAP=0.
  <query
      cd="0.3888"
      classic="//"
      book="novel"
      cost="price"
      title="name">/bib/[JUMP=0.6]//[DEEP=0.8]//[SWAP=0.8]novel[(@year < 2000) avg{3,1} ([SWAP=0.
  <query cd="0.36" classic="" book="//">/bib/[DELETE=0.6][DEEP=0.8]//[JUMP=0.6]//title</query>
  <query cd="0.36" classic="" book="">/bib/[DELETE=0.6][DEEP=0.8]//[DELETE=0.6]title</query>
  <query cd="0.36" classic="//" book="//">/bib/[JUMP=0.6]//[DEEP=0.8]//[JUMP=0.6]//title</query>
  <query cd="0.36" classic="//" book="">/bib/[JUMP=0.6]//[DEEP=0.8]//[DELETE=0.6]title</query>
  <query cd="0.324" classic="" book="//" title="name">/bib/[DELETE=0.6][DEEP=0.8]//[JUMP=0.6]//[
  <query cd="0.324" classic="" book="" title="name">/bib/[DELETE=0.6][DEEP=0.8]//[DELETE=0.6][SW
  <query cd="0.324" classic="" book="//" cost="price">/bib/[DELETE=0.6][DEEP=0.8]//[JUMP=0.6]//b
  <query cd="0.324" classic="//" book="//" title="name">/bib/[JUMP=0.6]//[DEEP=0.8]//[JUMP=0.6]/
  <query cd="0.324" classic="//" book="" title="name">/bib/[JUMP=0.6]//[DEEP=0.8]//[DELETE=0.6][
  <query cd="0.324" classic="//" book="//" cost="price">/bib/[JUMP=0.6]//[DEEP=0.8]//[JUMP=0.6]/
  <query cd="0.324" classic="//" classic="" cost="price">/bib/[JUMP=0.6]//[DELETE=0.6][DEEP=0.8]
  <query cd="0.324" classic="//" classic="//" cost="price">/bib/[JUMP=0.6]//[JUMP=0.6]//[DEEP=0.
  <query cd="0.324" bib="" classic="" cost="price">/[DELETE=0.6][DELETE=0.6][DEEP=0.8]//book[(@ye
  <query cd="0.324" bib="" classic="//" cost="price">/[DELETE=0.6][JUMP=0.6]//[DEEP=0.8]//book[(
  <query cd="0.324" bib="//" classic="" cost="price">/[JUMP=0.6]//[DELETE=0.6][DEEP=0.8]//book[(
```

Figure 7. An on-line session with the *Fuzzy-XPath* debugger

in [23], [24] we have recently presented a flexible approach for XPath debugging. The result of the debugging process of a XPath expression is a set of alternative queries, each one associated to a chance degree. We have proposed `JUMP`, `DELETE` and `SWAP` operators that cover the main cases of programming errors when describing a path about an XML document. Both techniques have been implemented (and can be tested on-line) by enjoying the benefits of using a new fuzzy command for filtering the set of ranked answers in a dynamic way, as we initially conceived for the *Fuzzy-XPath* interpreter in [29].

In this work we have coped with a pending task drawn in [24], [29] devoted to introduce *thresholding* techniques on our XPath debugger in order to increase its performance when dealing with massive XML files. The keypoint idea consists in to create filters for prematurely disregarding those superfluous computations dealing to non-significant solutions. Our approach represents the first real-world application developed with the fuzzy logic language MALP, for which we have recently developed some thresholding tabulation techniques [30], [31] (we are now implementing them into the $\mathcal{FLOPER}$ system). For the near future, we think that all these actions will be very useful for addressing in our framework the well-known "top-k ranking problem" (i.e. determining the top k answers to a query without computing the -usually wider, possibly infinite- whole set of solutions, which is strongly related with the FILTER command reported along this paper) [14], [35].

Acknowledgements

This work was supported by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grants TIN2013-45732-C4-2-P and TIN2013-44742-C4-4-R, as well as by the Andalusian Regional Government (Spain) under Project P10-TIC-6114.

References

[1] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Siméon, "XML path language (XPath) 2.0," *W3C*, 2007.

[2] E. Herrera-Viedma and G. Pasi, "Fuzzy approaches to access information on the Web: recent developments and research trends," in *Proc. International Conference on Fuzzy Logic and Technology (EUSFLAT 2003)*, 2003, pp. 25–31.

[3] S. Schockaert, N. Makarytska, and M. De Cock, "Fuzzy methods on the web: a critical discussion," *35 Years of Fuzzy Set Theory*, pp. 237–266, 2011.

[4] G. Pasi, "Flexible information retrieval: some research trends," *Mathware & soft computing*, vol. 9, no. 1, pp. 107–121, 2008.

[5] ——, "Fuzzy sets in information retrieval: state of the art and research trends," *Fuzzy Sets and Their Extensions: Representation, Aggregation and Models*, pp. 517–535, 2008.

[6] D. Kraft, G. Pasi, and G. Bordogna, "Vagueness and uncertainty in information retrieval: how can fuzzy sets help?" in *Proceedings of the 2006 international workshop on Research issues in digital libraries*. ACM, 2006, p. 3.

[7] P. Buche, J. Dibie-Barthélemy, O. Haemmerlé, and G. Hignette, "Fuzzy semantic tagging and flexible querying of XML documents extracted from the Web," *Journal of Intelligent Information Systems*, vol. 26, no. 1, pp. 25–40, 2006.

[8] A. Gaurav and R. Alhajj, "Incorporating fuzziness in XML and mapping fuzzy relational data into fuzzy XML," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 456–460.

[9] L. Yan, Z. Ma, and J. Liu, "Fuzzy data modeling based on XML schema," in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 1563–1567.

[10] B. Oliboni and G. Pozzani, "Representing fuzzy information by using XML schema," in *Database and Expert Systems Application, 2008. DEXA'08. 19th International Workshop on*. IEEE, 2008, pp. 683–687.

[11] ——, "An XML Schema for Managing Fuzzy Documents," *Soft Computing in XML Data Management*, pp. 3–34, 2010.

[12] A. Campi, E. Damiani, S. Guinea, S. Marrara, G. Pasi, and P. Spoletini, "A fuzzy extension of the XPath query language," *Journal of Intelligent Information Systems*, vol. 33, no. 3, pp. 285–305, 2009.

[13] E. Damiani, S. Marrara, and G. Pasi, "FuzzyXPath: Using fuzzy logic and IR features to approximately query XML documents," *Foundations of Fuzzy Logic and Soft Computing*, pp. 199–208, 2007.

[14] B. Fazzinga, S. Flesca, and A. Pugliese, "Top-k Answers to Fuzzy XPath Queries," in *Database and Expert Systems Applications*. Springer, 2009, pp. 822–829.

[15] B. Fazzinga, S. Flesca, and F. Furfaro, "On the expressiveness of generalization rules for XPath query relaxation," in *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*. ACM, 2010, pp. 157–168.

[16] H. Li, S. Aghili, D. Agrawal, and A. El Abbadi, "FLUX: fuzzy content and structure matching of XML range queries," in *Proceedings of the 15th international conference on World Wide Web*. ACM, 2006, pp. 1081–1082.

[17] E. Damiani, S. Marrara, and G. Pasi, "A flexible extension of XPath to improve XML querying," in *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2008, pp. 849–850.

[18] B. Fazzinga, S. Flesca, and F. Furfaro, "On the expressiveness of generalization rules for XPath query relaxation," in *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*. ACM, 2010, pp. 157–168.

[19] ——, "Xpath query relaxation through rewriting rules," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 23, no. 10, pp. 1583–1600, 2011.

[20] J. Almendros-Jiménez, A. Luna, and G. Moreno, "A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming," in *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11*. Springer Verlag, LNCS 6826, 2011, pp. 186–193.

[21] ——, "Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language," *Electronic Notes on Theoretical Computer Science, ENTCS*, vol. 282, pp. 3–18, 2012.

[22] J. M. Almendros-Jiménez, A. Luna-Tedesqui, and G. Moreno, "Fuzzy xpath through fuzzy logic programming," *New Generation Computing*, vol. 33, no. 2, pp. 173–209, 2015. [Online]. Available: http://dx.doi.org/10.1007/s00354-015-0201-y

[23] J. Almendros-Jiménez, A. Luna, and G. Moreno, "A xpath debugger based on fuzzy chance degrees," in *On the Move to Meaningful Internet Systems: Proceedings OTM 2012 Workshops, Rome, Italy, September 10-14*, P. H. et al., Ed. Springer Verlag, LNCS 7567, 2012, pp. 669–672.

[24] J. Almendros-Jiménez, A. Luna, and G. Moreno, "Annotating Fuzzy Chance Degrees when Debugging Xpath Queries," in *Proc. of the 12th International Work-Conference on Artificial Neural Networks, IWANN'13*. Springer Verlag, LNCS 7903, Part II, 2013, pp. 300–311.

[25] J. Medina, M. Ojeda-Aciego, and P. Vojtáš, "Similarity-based Unification: a multi-adjoint approach," *Fuzzy Sets and Systems*, vol. 146, pp. 43–62, 2004.

[26] P. Morcillo and G. Moreno, "Programming with Fuzzy Logic Rules by using the FLOPER Tool," in *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML'08*, N. B. et al., Ed. Springer Verlag, LNCS 3521, 2008, pp. 119–126.

[27] G. Moreno and C. Vázquez, "Fuzzy logic programming in action with floper," *Journal of Software Engineering and Applications*, vol. 7, pp. 237–298, 2014.

[28] J. Almendros-Jiménez, A. Luna, G. Moreno, and C. Vázquez, "Analyzing Fuzzy Logic Computations with Fuzzy XPath," in *Proc. of PROLE'13*. Universidad Complutense de Madrid, 2013, pp. 136–150.

[29] J. Almendros-Jiménez, A. Luna, and G. Moreno, "Dynamic filtering of ranked answers when evaluating fuzzy xpath queries," in *Rough Sets and Current Trends in Computing - 9th International Conference, RSCTC 2014, Granada and Madrid, Spain, July 9-13*. Springer Verlag, LNCS 8536, 2014, pp. 319–330.

[30] P. Julián, J. Medina, G. Moreno, and M. Ojeda, "Efficient thresholded tabulation for fuzzy query answering," *Studies in Fuzziness and Soft Computing (Foundations of Reasoning under Uncertainty)*, vol. 249, pp. 125–141, 2010.

[31] P. Julián, J. Medina, P. Morcillo, G. Moreno, and M. Ojeda-Aciego, "An unfolding-based preprocess for reinforcing thresholds in fuzzy tabulation," in *Proc. of the 12th International Work-Conference on Artificial Neural Networks, IWANN'13*. Springer Verlag, LNCS 7902, Part I, 2013, pp. 647–655.

[32] J. M. Almendros-Jiménez, "An Encoding of XQuery in Prolog," in *Proc. of the Sixth International XML Database Symposium XSym'09*. Heildelberg,Germany: Springer, LNCS 5679, 2009, pp. 145–155.

[33] J. M. Almendros-Jiménez, A. Becerra-Terón, and F. J. Enciso-Baños, "Querying XML documents in logic programming," *Theory and Practice of Logic Programming*, vol. 8, no. 3, pp. 323–361, 2008.

[34] J. Almendros-Jiménez, A. Luna, and G. Moreno, "Fuzzy xpath queries in xquery," in *On the Move to Meaningful Internet Systems: OTM 2014 Conferences - Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31*. Springer Verlag, LNCS 8841, 2014, pp. 457–472.

[35] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-*k* query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, 2008.