Using BMH Algorithm to Solve Subset of XPath Queries

David Toth

Dept. of Computer Science and Engineering,
FEE, Czech Technical University,
Karlovo Náměstí 13, 121 35 Praha 2, Czech Republic
tothd1@fel.cvut.cz

Abstract. Boyer-Moore-Horspool (BMH) algorithm is commonly used to solve text searching problems. In this paper is used to solve the constraint subset of XPath queries offering effective algorithm to resolve such queries. XML can be grasp as text file contains tags and its content; that kind of view to XML is used in this work. We constraint XML document content and possible XPath queries. This work focus on key ideas and problems appertaining XPath queries execution using text search algorithm BMH.

1 Introduction

Many papers were published appertaining the XPath query effective execution issues. Many authors concern many aspects of XML and XPath. In this paper we look in detail how to solve part of XPath queries involving only path of nodes in the query but not any functions. Another extensions are possible and in future works can be exploited in more details; see into Conclusions and future works for more information.

Why bother with XML and XPath? XML is special text file format, meta-language, markup language (using tags instead of commands); XML can be represented by a tree where nodes represent tags and connections between nodes represent relationship super-element and embedded element. XML can be used for many reasons. Especially useful is using XML as data interchange format in internet environment. It is conceived as a tool assuring compatibility between different applications possibly running on different platforms and using different inherent data paradigm and formats. Today in B2B applications is XML widely used.

Of course XML has many other purposes. For example in XML documents can be stored content of any magazine or generally paper. Such kind of XML documents are known as document oriented XML; i.e. there is huge amount of text contrary to data oriented XML documents where this is not generally true. In data oriented XML documents are many tags and often as much as data itself. Finally the tag information is itself also valuable information. Document oriented XML are typically intended to use by humans in opposite to data oriented XML documents which are normally designated for computers.

J. Pokorný, V. Snášel, K. Richta (Eds.): Dateso 2007, pp. 81–88, ISBN 80-7378-002-X.

Here we try to prove that whenever data oriented XML documents are supposed we can treat them as plain text files finding queried node(s) using BMH algorithm. Only when the amount of text representing structure information (for simplicity let us consider just tags) in XML file be the same order of magnitude as amount of text representing stored data itself. This is big assumption which has to be preserved to can legally speak about effective XPath query resolution.

The next section introduce basic BMH algorithm principle and show example of using this algorithm. Another section acquaint XPath queries and selected subset further elaborated. After come section named BMH in XPath where are introduced and explained all considered XPath query execution algorithms based basically on BMH from concrete queries to generic ones. Finally in Conclusions and future works section assess gained results and procedures and outline extensions and future works directions.

2 BMH Algorithm

After short introduction here we will treat BMH algorithm basics.

Here we deal with BMH (Boyer-Moore-Horspool) algorithm commonly used to solve text searching problems. The basis of this approach inhere in a little preprocessing of searching pattern and then searching the text forward but backwards from pattern. That is the trick. That is how can be raise effectivity. There can be hopped (or jumped) many irelevant letters (or generally parts of input stream). The number of reading is therefore typically much more lesser than whole length of stream..

2.1 Preprocessing issues

What must be done during preprocessing phase? For any pattern there is a need for create hopping table where must be pre-counted how many letters (or input symbols) can be jumped after input sign and the appropriate sign from pattern has been read and they do not match.

Let us show an example. Suppose input stream as: „Where are you going young man?“ Furthermore assume we trying to find pattern: „you“.

The answer obviously is that the pattern is presented 2 times 1st at position 11 and 2nd at position 21. Naive approach lead us to read whole text i.e. 31 characters and do at least the same number of comparisons.

When we try to use BMH we have to resolve how much can be hopped when some letter is achieved. The preprocessed table should look like this:

y o u x
2 1 3 3

Which means that whenever compare of input letter and pattern letter do not match we should jump ahead of 2, 1 or 3 characters depending on what letter is on input

what is talking about the 1st line in the above table. Note that x mean any other symbol than enumerated before x – in our case whatever symbol but y, o, u.

So the above example will be, using BMH algorithm, take just 16 readings.

2.2 How BMH algorithm operate?

First necessary variables:

I	position in input stream
pat	pattern

2.3 Semiformal BMH description (BMH algorithm)

1. preprocessing – the tab is calculated
2. begin
3. Go to the I where I = len(pat)
   (initial position in the input text is set to the length of pattern)
4. repeat (until in input stream the end is not reached)
5. compare letters backwards from position I with pattern until they fit each other
6. when pattern is matched then
   1. save the position of 1st matched occurence – store I into the array
   2. go to the position of possible next occurence which is: I = I + 2*len(pat)
7. (ELSE) when pattern is not matched then
   1. go ahead to the fartest position possible which is I = I + tab[K], where K is letter of input text where was expected any different symbol
8. end.

Algorithm notes. len means function returning length of its string argument and tab preprocessed table containing length of possible hops for all letters.

Useful source where formal BMH algorithm description is introduced, explained and described using an example is for example [2].

After BMH algorithm basics were explained next section deal with XPath query which we will subject of next investigation.

3 XPath queries

Now after the slim repetition of how BMH algorithm runs we will focus on XPath queries. This section deal with investigated XPath query subset which is briefly introduced.

XPath is derived from XML Path which suggests us it is about determining paths in XML documents. In fact it is more complicated. Functions and other features must be considered furthermore. But for the purpose of this paper we focus on simple queries involving just nodes and paths.

Generally there are two possibilities how to determine path using XPath. So called relative paths and absolute paths. Absolute path always syntactically begins with the sign for root of the XML document which is denoted as '/'. Absolute path is inevitably derived from the root of the document. Result of such query can be none node found, exactly one node in specified path or set of nodes of same name in specified path. Relative queries on the other side mean that somewhere in the query this sequence '//' is placed which means no matter of deep where the node will be found. Such kind of a query is practically eminently useful but it is this kind of queries which means problems with effectivity. XPath queries examples follow.

First absolute queries:

/rootnode/node1/node3 – the result are all nodes appertaining <node3> elements which are descendants of <node1> element which are descendants of <rootnode> element.

/rootnode/node6 – the result of this XPath query are all <node6> elements which are descendants of <rootnode> element.

Let us see now relative queries:

//node2 – such XPath query gives all <node2> elements in valid XML document.

//node4/node5 – the answer is: all <node5> elements whichs parents are <node4>.

And finally we will look at mixed queries:

/rootnode/node7//node8/node9 – the return all <node9> elements whichs parents are <node8> elements placed however deep inside the element <node7> which has to be direct descendant of <rootnode> element.

Many useful properties of XPath were omitted. We will treat about extensions in part Conclusions and future works. XPath query somehow specify the path in tree structure of XML document. We will not need this imagination anymore. In this paper linear conception of XML (XML viewed as a text file) will be sufficient. All about XPath can be found in [4].

After XPath query subset was delimited we will in next section focus on how to this subset can be implemented using BMH algorithm.

4 BMH in XPath

Now after we briefly examined BMH algorithm and supposed XPath queries we will look at how to use BMH when evaluating XPath queries.

First the easiest case. Let us consider the following XPath query: //nodeX. So the result of such a XPath query should be all elements named nodeX wherever inside the XML document.

Of course we suppose the input XML document is well-formed XML document as this term is commonly understand according to [3]. Furthermore we do not consider CDATA section inside the XML document but it is not hard to imagine its subsequent incorporation. But here it would break our deliberations and diverge to another

than key problems. Except CDATA we also do not consider comments in XML. Both CDATA and comments can be easily incorporated to the proposed algorithms but it is not the aim of this work to focus on every aspects. Rather we focus on key principles. See the section Conclusions and future work for summarizing involving issues.

4.1 Semiformal simple algorithm description (simpleBMH)

1. use BMH algorithm to find asked pattern on input stream
2. for all found patterns must be verified the context
    1. just a word in text inside an element but do not denote an element (if it is the case then nothing happen, go further without writing into the results; this test can be done comparing symbols before the found pattern and symbol '/' or '</')
    2. it denotes an element and then there are two options
        1. it is opening tag
           (we should write the result into the output – position when the element starts; opening tag it is if precede symbol '<')
        2. it is closing tag
           (we should write the result into the output – add the closing position to the start position; closing tag it is if precede symbols '</')

The result will create pairs of beginning and ending positions individual element found in the text. Here we can see the power of XML grasped as plain text file.

Now we can search nodeX in XML document using XPath very simple notation: '//nodeX'. How long does this take? In order of magnitude as BMH itself. There is only one or two comparisons more when pattern match and one possible writing into the results (depending on if it is beginning or closing tag). So the performance will be comparable as the performance of BMH itself which is described in detail and with experimental results for example in [1].

Let us see how will the algorithm change when we focus on XPath queries using absolute path in the beginning of the query. We will suppose such a query: /rootnode//nodeX

4.2 Semiformal algorithm description for absolute paths only (apo-algorithm)

1. the beginning of input stream must exactly match '<rootnode'
   (as in the case of previous rumination)
   Note to the following point: (in fact we will use brute force to jump over the rootnode element attributes definitions and the possible content up to the first occurrence of any element inside rootnode)
2. repeat until the end of XML is reached
3. using brute force: find name of following element (second in absolute order in first iteration) in text (in XML document)
    1. if it is the element we looking for then
        1. into the result write the beginning mark

2. apply algorithm BMH (including preprocessing) with elements' end tag as pattern content; algorithm stop when first occurrence was found
3. into the result write the ending mark of this occurrence

2. else: apply algorithm BMH (including preprocessing) with elements' end tag as pattern content; algorithm stop when first occurrence was found
(no writing to the results; only hopping is realized)

Now should be presented some mechanism for generic kind of query. We should use previously elaborated, experienced and tested work. So we will use preceding algorithms. Let us consider more generic query as: /rootnode/node1/node2/node3/nodeX.

4.3 Semiformal generic algorithm for only absolute paths (gab-algorithm)

1. do an extraction of node names into the field of names: nodext
2. let us consider L variable where will be increased the relative level of depth depending on query structure; L = 2
3. let us consider MAX as number of nodes in path in query
4. the beginning of input stream must exactly match '<rootnode'
5. repeat until the end of XML is reached
6. using brute force: find name of following element in text (XML document) or end tag nodext[L-1]
   1. if it is the element we looking for (i.e. nodext[L]) then
      1. L++ (increase L)
      2. if L = MAX then
         1. into the result write the beginning mark
         2. apply algorithm BMH (including preprocessing) with elements' end tag as pattern content; algorithm stop when first occurrence was found
         3. into the result write the ending mark of this occurrence
      3. else:
         1. go to 6.
   2. if it is some other element
      1. apply algorithm BMH (including preprocessing) with elements' end tag as pattern content; algorithm stop when first occurrence was found
      (hopping non requested element)
   3. if it is reached end tag then
      1. L-- (decrease L)

Now we'll try to extend algorithm for long relative queries as for example is this: //node1/node2/nodeX.

4.4 Semiformal generic algorithm for only relative paths (ger-algorithm)

1. make extraction of nodes from query to the field: nodext
2. variable L determine the position in nodext – relative depth; L = 1

3. let MAX denote number of nodes in a query
4. use BMH to find all opening tags: nodext[L] and for every occurrence found run own thread continuing with 5.
5. use gab-algorithm from point 5

Finally we combine absolute path at the beginning and relative path in the end of query. Suppose query like this one: /rootnode/node1/node2//node3/node4/nodeX.

4.5 Semiformal generic algorithm (gen-algorithm)

1. make extraction of nodes from a given query into two fields
   1. nodextabs – containing nodes up to sign '//' and
   2. nodextrel – containing nodes from symbol '//' till the end of a query
2. variable L1 will serve as relative depth for first part
3. variable L2 will serve as relative deptg for second query part
4. MAX1 denotes number of nodes in first part of a query
5. MAX2 denotes number of nodes in second part of a query
6. use gab-algorithm with nodextabs, L1 and MAX1
   1. for every result use ger-algorithm with, nodextrel, L2, MAX2

Author hope this summarizing of thinking can help when inventing new procedures, algorithms and other ruminating related issues. Future works should include also implementation of proposed algorithms. About future works and conclusions concern following section.

5 Conclusions and future works

Most important made assumptions are these:

1. XML document is mainly data-oriented and do not contain CDATA sections.
2. XPath query do not contain anything else but node names and sign '/' or '//'.

We stated relatively strong preconditions. None of those should have great impact on key ideas presented here. All such assumptions should let us concentrate on key and main issues of discussed problem: how can be effectively used BMH algorithm to answer constraint subset of XPath queries. These preconditions are limited but can be step by step eliminated. This should be the aim of future works; choose little pieces of what is not yet implemented and incorporate it to the solution which let grow the picture of whole.

Much work can be done here on this field. Every supposition could be subdue to rational critics and proposed algorithm should be extend to encompass every such a conditions' solution.

All proposed algorithm variants should be experimentally examined as well as all possible following extensions and modifications.

One of future works should respect document oriented XML. There are several ways how to solve this problem. One of such could be invent special object structure a priory allowing jump over parts which should be hopped – variation of state space cutting algorithm.

Consequences of proposed algorithm for execution of XPath queries. Follow most important consequences when using a variant of proposed algorithms.

1. Only data oriented XML documents will be treated truly effectively because of there will be not present long text passages to be searched.
2. When node names containing just plain text are queried and document contain predominantly nodes containing mostly numbers, which is often the case of data-oriented XML documents, then the effectivity will dramatically increase – all numbers will be as a consequence of using BMH algorithm, hopped.
3. With longer queried tag name the proposed algorithm will be more efficient.
4. It has none or just little impact when the length of queried tag name has only 1, 2 or 3 characters.

On one side stated consequences can be grasped as drawbacks but when we focus on special area of data handling here may be especially useful to integrate proposed algorithm as for example when exporting numerical data for XSLT to expose them into the web. Majority of above stated consequences appertain this special case where could be proposed algorithm implemented.

References

1. Lecroq, T.: Experimental Results on String Matching Algorithms. Software—practice and experience, vol. 25(7), John Wiley & Sons, Ltd. (1995) 727–765
2. Lovis, C., Baud, R.H.: Fast Exact String Pattern-matching Algorithms Adapted to the Characteristics of the Medical Language. J Am Med Inform Assoc. 7(4) (2000) 378–391.
3. XML 1.0, W3C Recommendation of Extensible Markup Language (XML) 1.0 (Fourth Edition), http://www.w3.org/TR/REC-xml
4. XPath, W3C Recommendation of XML Path Language (XPath) Version 1.0, http://www.w3.org/TR/xpath