Comparison of XPath Containment Algorithms

Jorge Guerra[1], Luis Useche[1], Miguel Rivero[1], Imtiaz Syed[1], Hussein Orelus[1], Malek Adjouadi[1], Armando Barreto[1], Bradley Stoute[2], Scott Graham[1], Naphtali Rishe[1]

Florida International University[1] *University of Illinois at Chicago*[2]

Abstract

The complexity of implementing a sound, complete and efficient algorithm to detect containment (and by extension equivalence) in XPath fragments has frustrated previous attempts by computer scientists. Some algorithms have been developed to deal with these problems but these have lacked either in completeness, efficiency, or soundness, working only when restrictions in the type of XPath fragments are enforced.

In this paper we will attempt to evaluate a prominent algorithm in terms of time for different inputs. These inputs try to explore different behaviors of the algorithm such as: the most influential construct for the running time, the behavior of the algorithm in common cases. And the response time of the algorithm for the worst case, which covers the whole tree. We present experiments that allow us to determine the behavior of the algorithm and evaluate it via each of the above questions.

1. Introduction

XPath is ubiquitous in today's computing environment. Technologies such as XSLT, XQuery, XPointer, XLink and others are based on XPath, which may be implemented differently in these technologies but essentially always do the same task: given an input tree (in XML), a context (a point of reference) and an expression (the selection criteria), XPath will return one, many, or zero nodes from the XML input tree.

What are the characteristics of an ideal algorithm? First, we expect it to be sound; that is, in the case of the nodes returned by an XPath expression, the nodes should indeed be part of the expected result set. There should not be erroneous nodes included in the result set. Second, the algorithm should be complete; that is, it should not return false negatives. In other words, the result set of the nodes returned by an XPath expression should include each and all nodes that are expected to be included. Lastly, the algorithm should be efficient, both in its use of space (memory) and in the time it takes to execute. An ideally efficient algorithm would execute with logarithmic growth, which would make the algorithm insensitive to the size of the input tree (XML) and the complexity of the XPath expression after certain threshold. This is the inverse of exponential execution (EXPTIME), the worst case scenario, where after an initial small threshold, every increment in the size of the input tree results in dramatic increases of both space and time requirements.

The subset of XPath fragments that have proved elusive are defined as $XP^{\{[],*,//\}}$, where [] means branching, * means wildcards and // means descendants. Efficient algorithms have been found for fragments that include any two of the three constructs, but finding and algorithm to detect containment in fragments with all three constructs has proven to be coNP-Complete.

Why is the containment problem important? Every time XPath is used to select a set of nodes from an XML document (input tree) the processor implements an algorithm to traverse the input tree and identifies the nodes that must be returned. So far all implementations are considered to be inefficient in that the time and memory it requires XPath to do its work is exponential with respect to the size of the input tree. Finding an efficient algorithm will optimize the speed and memory requirements for applications manipulating XML documents. Also, when two XPath expressions are found to be equivalent, one can be substituted for the other resulting in optimization.

2. Fundamentals

There are some notions that must be understood in order to understand how and why the proposed algorithms are implemented. Some of the terms that must be fully understood are "tree patterns", "arity", "embedding", "boolean patterns", "canonical models", "homomorphism", and "FTA".

2.1. Tree Patterns

Although we usually refer to XPath expressions using a notation such as a//*[b//d][c], this expression can be converted to a tree pattern. In fact, as stated in [4], "Every expression in XP $^{\{[],*,//\}}$ can be translated into a tree pattern of arity one with the same semantics, and, conversely, each pattern of arity one can be translated into an XP $^{\{[],*,//\}}$ expression." By converting two XPath expressions into two tree patterns, it becomes easier to visualize the containment problem, which is trying to identify if a tree pattern is contained in the other one. Likewise, existing algorithms designed to detect containment work on tree structures, not expressions, thus the conversion from expressions to tree patterns becomes necessary.

2.2. Arity

In mathematics, arity refers to number of arguments in the domain of a function. Although in real life programmers typically create functions with many arguments, it is rare in mathematics and sciences to see functions with arity greater than 3, and seldom with more than 1. With tree patterns, arity refers to the number of tuples returned by an expression.

2.3. Boolean Patterns

Boolean patterns are important because they are used in the construction of models and canonical models that would help us identify containment. Boolean patterns are tree patterns with arity 0; in other words, a Boolean pattern only returns a true or false answer; even if the answer is true, it still does not return any tuples (for false it returns the empty set and for true it returns the empty tuple). More importantly, it has been proposed in [4] Proposition 1 that any tree pattern, of any arity, can be translated to an equivalent Boolean pattern, and that proving containment for two Boolean patterns also prove containment for the original tree patterns.

2.4. Canonical Models

First we must define what a model is. Model refers to all variations of possible trees based on the infinite alphabet Σ that evaluate to true when a given Boolean pattern is applied to them [1]. The models of Boolean pattern p are all trees that would return true when compared against p. Models of Boolean pattern p are identified as *Mod(p)*.

Canonical models of p are the subset of *Mod(p)* for trees that have the same shape as p. They are identified as *m(p)*. There is a translation that takes place in the resulting *m(p)* where all wildcards * are replaced by symbols from Σ, but this would make *m(p)* infinite since each * can be replaced by an infinite number of symbols from the infinite alphabet Σ. Thus, to have a working model with a finite set of trees, a constraint is introduced. First, it becomes necessary to identify a symbol *z* from Σ that is not in either tree patterns (remember, we are trying to determine if one tree pattern is contained by another). Then all the * in *m(p)* are then replaced by that one character *z*. The resulting canonical models are identified by $m^z_n(p)$.

3.Algorithm CheckContainment II

The CheckContainment II is the algorithm evaluated in this paper. It was proposed by Miklau and Suciu [4]. It is sound and complete, but is not efficient unless some bounds are imposed on the wildcards, the descendant edges or branching. However, further improvements will consist of approximations or heuristics, as the previous work has proven the problem of checking XPath containment to be in EXPTIME, if no bounds are enforced on the number any particular operand. This algorithm works by dealing with the containment problem of regular tree languages because the containment problem of tree patterns can be reduced to it.

The steps of the algorithm are as follows [4]:

1. Construct the DFTA A accepting $\mathrm{Reg}_{\Omega p}$
2. Construct the AFTA A' accepting $U_p^{-1}(\mathrm{Mod}(p'))$
3. Compute the AFTA B = A X A' (the product automaton)
4. Compute the DFTA C = det(B)
5. If lang(A) is a proper subset lang(C) then return true, else return false.

It is rather simple to determine if a regular tree language is contained within another regular tree language. However, the original input to the algorithm is in a tree pattern format, not a regular tree language, and thus the complexity in this algorithm lies in converting tree patterns into regular tree languages. More specifically, tree patterns are unranked, unordered trees, whereas the tree automata needed to determine if a language is a subset of another works on ordered, ranked trees. "Ranked trees have the property that every node which is not a leaf has the same number of children … On the other hand, in unranked trees different nodes can have different number of children" [3] page 129.

$\mathrm{Reg}_{\Omega p}$ is a regular tree language, which in this case will be accepted by a DFTA (explained below). $\mathrm{Reg}_{\Omega p}$ is

also the resulting ranked and ordered alphabet derived from the infinite alphabet Σ used in the original tree pattern. As part of the algorithm, we must find canonical models based on boolean pattern p. For further information refer to Miklau and Suciu [4].

4. Experimental Evaluation

We now evaluate the proposed algorithms using the implementation made by Haj-Yahya [3] on Java. Given that the original algorithm only tested containment in one direction and we required the algorithm to be more verbose, we added the test on the missing direction. Hence each test we report consist of two checks, given p and p' the program will verify if p' is a subset of p and if p is a subset of p'. However, this does not affect the algorithm's running time order because each test is disjoint, thus the algorithm remains in the same complexity class.

4.1. Experimental Setup

Our testbed is composed of two machines. In the first machine has a Pentium 4 2.00 GHz processor and 1GB of RAM memory running Linux 2.6.17, here we ran the worst case test, that two equal expressions therefore all there space tree has to be explored in order to find a solution. The second machine has a Pentium M 1.60GHz processor with 512MB of RAM and also runs Linux 2.6.17 and was used for the random XPath expressions experiments.

4.2.Covering all the Pattern Tree (worst case)

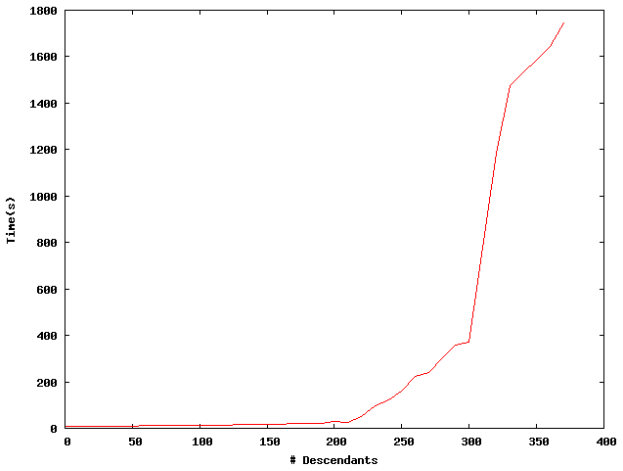


Figure 1: Behavior of the CheckContainment II algorithm in worst case.

The first evaluation is to calculate the time taken by the algorithm in the worst case, that is examining all the tree pattern. To ensure that the algorithm explores all the tree pattern, we introduce the same XPath expression in both parameters, p and q. Note that the XPath expressions used in each of these experiments are generated randomly given a number of descendants, wild-cards and branches. For this experiments we ran the algorithm 36 times, each with 200 wild-cards and 200 branches. The number of descendants was varied from 0 to 370 making increments of 10 in each run. Figure 1 shows the results for this experiment. The x-axis corresponds to the number of descendants in each expression introduced to the algorithm. On the other hand, the y-axis corresponds to the running time measured in seconds of the algorithm. We can see the big impact that the number of descendants has in the running time of the algorithm. Based in this experiment, it is clear that the running time of the pattern tree containment algorithm, in the worst case, increases exponentially with the number of descendants in the XPath expressions. This backs the formal prove presented in [1] that the CheckContainment II algorithm is EXPTIME given an arbitrary number of descendants.

4.3.Running time for different wildcards and descendants

Another interesting evaluation for this algorithm is to measure the difference between the impact of the operands, wild-card and descendant. For this experiments we start from the same expressions to get the time for a base case. Then we increment the corresponding operand, wild-card or descendant, and log the running times in seconds. After plotting the values we obtained the graph in the Figure 2. The x-axis corresponds to the number of the execution in each case and the y-axis is the time in the seconds of the corresponding instance. For each execution we increment the number of the operands by one that were being evaluated. We can see in the graph the big influence that the descendant operands have in the running time of the algorithm. Thus, the highest impact in the running time is directly related to the number of descendants in the evaluated expressions.

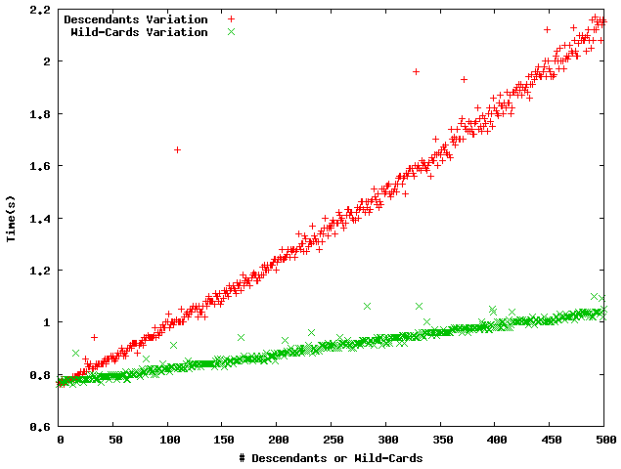
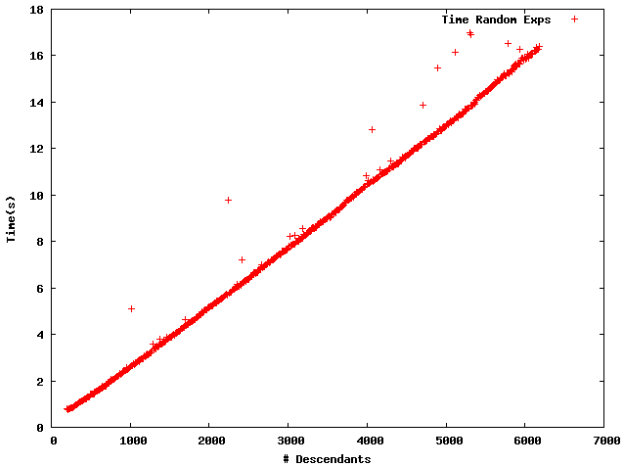


Figure 2: Comparison of CheckContainment II algorithm varying the number of descendants (+) and of wildcards (x).

4.4. Evaluation of random Xpath expressions



The second evaluation of the tree pattern algorithm tests how well it behaves with random expressions as parameter, in other words, how good the performance is without forcing it to cover all the tree. Given that the operand with most influence in the execution time is the descendant, in this case, we introduce random XPath expressions with different number of this operand in both parameters. For this experiment, we ran the algorithm 600 times. For each instance, in the first parameter, we introduce a random XPath expression with 200 wild-cards, 200 branches and descendants starting from 200 until 6200 incrementing each time 10 units; the second parameter was a similar random expression. The above graph shows the result in this experiment. The x-axis represents the number of descendants in the expression and the y-axis corresponds to the running time in seconds of the algorithm. We can see that for random expressions, the algorithm has a polynomial running time even varying the the most relevant operand in the parameters. Thus, this experiment shows that the algorithm in most of the cases is capable to response in a polynomial time and the need of explore the complete tree is not the common case, if we consider this to be the average case for applications checking XPath containment or equivalence.

5. Related Work

Containment for P{[ ],//} was shown in PTIME. The containment can be effectively decided for a large XPATH expression that includes Union, Intersection, path composition, together with all XPATH axes, branching and wild cards[7]. Containment for conjunctive queries is NP complete. In a graph-based data model, it has been showed that a restricted language without wildcard, the containment is NP-Complete.

6. Conclusions

In this paper we presented a simplified version of the XPath containment and equivalence checker proposed by Miklay and Suciu [4,5,6]. We also present three experiments which evaluate different behaviors in the CheckContainment II algorithm. Just measuring the running time of different inputs, we were able to obtain what is the most influence component in the running time algorithm that, based in our experiments result, was the descendant operand. Additionally, this practical experiments corroborate the Theorems 4 and 5 in the Miklau and Sucius paper [4]. However, it also demonstrate that for most of the cases the algorithm has a polynomial time response with the increasing of the descendant operands. Finally, we demostrate that just in the worst cases, where the algorithm needs to cover all the tree, the algorithm has a exponential increase in the running time.

7. Acknowledgements

This research was supported in part by NSF grants HRD-0317692, CNS-0220562, CNS-0320956, and CNS-0426125, and NATO grant SST.NR.CLG:G980822.

8. References

[1] Diez, F. J. and Druzdel, Marek J. 2001. Fundamentals of Canonical Models. In *Ponencia Congreso: IX Conferencia de la Asociacion Espanola para la Inteligencia Artificial (CAEPIA-TTIA 2001).*

[2] Haj-Yahya, Khaled . Das Teilmengenproblem fur eine Untermenge von XPath. Universitat zu Lubeck. Bachelor Thesis. 2005.

[3] Libkin, L. 2004. *Elements of Finite Model Theory (Texts in Theoretical Computer Science. an Eatcs Series)*. SpringerVerlag.

[4] Miklau, G. and Suciu, D. 2002. Containment and equivalence for an XPath fragment. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Madison, Wisconsin, June 03 - 05, 2002). PODS '02. ACM Press, New York, NY, 65-76.

[5] Miklau, G. and Suciu, D. 2003. Containment and equivalence of Tree Patterns. University of Washington Technical Report TR 02-02-03.

[6] Miklau, G. and Suciu, D. 2004. Containment and equivalence for a fragment of XPath. *J. ACM* 51, 1 (Jan. 2004), 2-45.

[7] Pierre Genevès and Nabil Layaïda. 2007. Deciding XPath Containment with MSO. To appear in Data and Knowledge Engineering (DKE), Elsevier.