Querying Streaming XML Using Visibly Pushdown Automata

Robert Clark

October 10, 2008

Abstract

We present a novel solution to streaming XPath query evaluation, capable of supporting forward and reverse axes queries. We show a conversion of an XPath query to a Visibly Pushdown Automaton that preserves the semantics of the query. It was shown in [1] that such a construction results in an automation suitable for efficient streaming query evaluation. We also show that the constructed automaton is polynomial in size of the query when negated predicates are disallowed.

1 Introduction

XML has emerged as the standard format for the exchange of data across the internet. As such, querying XML documents is an important question; that is, we would like to efficiently, both in terms of space and time, be able to search a given XML document for structural and textual patterns.

Xpath is a query language that returns sets of positions in an XML document that answers the query. Xpath query evaluation has been explored in both streaming and non-streaming settings. Traditional Xpath query engines [2] work by first loading an XML document into memory before querying. There are two important cases where this approach is not practical. First, if the document size is very large, there may not be enough memory to hold to whole document while executing the query without experiencing degraded performance due to heavy paging. Second, if the XML data arrives continuously, as with streaming stock market information, loading the whole document into memory is simply not feasible. We would like to be able to evaluate Xpath queries on an XML document in a *streaming* fashion, that is, evaluate the query by inspecting an XML document one tag at a time, and storing the least amount of information necessary to answer the query.

There have been many attempts at solving the streaming XPath problem. Most focus on a construction that captures some subset of the W3C recommendation on XPath [3]. Our approach is similar and we focus on Core XPath, a logical core of XPath, defined in [2]. Previous work in streaming XML has relied on algorithms that use ad hoc data structures to keep track of patterns that have been partially matched by nodes in a streaming XML document [4]. Our approach is novel in that it is an *automata theoretic* approach to the streaming XML problem. The automaton we construct in this paper formalizes the partial matches for an arbitrary XPath query in a uniform fashion.

VPAs are an extension of the well known automation model called Pushdown Automata (PDA). The VPA model was introduced by Madhusudan and Alur in [1]; in this model, the input alphabet is the union of three disjoint alphabets and a character's membership in a particular alphabet determines which stack operations are legal when that character is read by the automation. This model naturally captures the open-closed tag relationship exhibited by an XML, where one may push onto the stack at an open tag event and pop at the arrival of the corresponding close tag event. Madhusudan and Viswanathan have also shown that efficient streaming query algorithms are attainable, if the query is expressed as a query VPA [5].

Our objective is to precisely represent a Core

Xpath (cXPath) expression with a VPA that is suitable for use in streaming XPath query evaluation. The contributions of this paper are as follows:

- We provide a conversion procedure that translates a cXPath expression into a VPA. This construction preserves the semantics of the query, leading to an automaton that captures the exact XPath query.
- Show the worst-case and average case complexity and prove correctness of the above construction.

The rest of the paper will be organized as follows. In section 2 we show that a well-formed XML document can naturally be viewed as a properly nested string. This observation provides the motivation for using VPAs for streaming query evaluation purposes. In section 3 we discuss Xpath and its capabilities of expressing a query and the subset of XPath which will be addressed in this paper called Core Xpath which was introduced in [2]. It is the semantics of cXPath that we will capture with the VPA model of automation. In section 4 we provide the intuition for the conversion process from a cXPath expression to VPA, describe the formal conversion process, and in section 5 we discuss its complexity and correctness.

2 XML String Interperation

In this section we show it is a natural to interpret an XML document as a properly nested string. A detailed discussion of this notion is provided in [6].

2.1 XML to Properly Nested Strings

Suppose we are given a well-formed XML document D. We may write $D = (V, E)$ where V is the set of nodes in D and E is the set of edges in D. We will assume that we have the following function on D, $nodes : D \to V$ which returns the set of nodes in document D.

We will show that there is a natural interpretation of D as a properly nested string. This provides motivation for use of VPAs as our choice of automation to approach this problem since VPAs naturally handle the structure of nested strings by reading the word left to right, pushing on open tags and popping on close tags.

Given an XML document D, let Σ be the collection of the open tags of D and let $\bar{\Sigma}$ be the set of closed tags.

Definition 1. *We say a string over the alphabet $\Sigma \cup \bar{\Sigma}$ is properly nested if it is generated by the grammar:*

$$WM \to aX\bar{a}$$
$$X \to aX\bar{a} \mid XX \mid \epsilon.$$

Figure 1: WM represents the start variable, a is any element in Σ, ϵ is the empty string.

We will denote the language of this grammar as $WM(\Sigma)$ and we will denote the set of all trees with nodes labeled by Σ as $T(\Sigma)$.

Define $\phi : T(\Sigma) \to WM(\Sigma)$ as follows. For a tree $T \in T(\Sigma)$ with a single node labeled a, $\phi(T) = a\bar{a}$. For a tree T with root a and subtrees $T_1, ..., T_k$, $\phi(T) = a\phi(T_1)...\phi(T_k)\bar{a}$. Define $\phi' : WM(\Sigma) \to T(\Sigma)$ as follows. For $a\bar{a} \in WM(\Sigma)$, $\phi'(a\bar{a})$ is the tree consisting of a single node labeled a. And for $w = aw_1...w_k\bar{a} \in WM(\Sigma)$, $\phi'(w)$ is the tree rooted at a with subtrees $\phi'(w_1), ..., \phi'(w_k)$. These mappings are clearly inverse operations.

Proposition 1. *There is a one-to-one, onto map $\phi : T(\Sigma) \to WM(\Sigma)$*

Proof. The mappings described above are injections from $T(\Sigma) \to WM(\Sigma)$ and $WM(\Sigma) \to T(\Sigma)$. □

Thus, given a well-formed XML document, we may without ambiguity speak of the properly nested word the document represents. We adopt this view of an XML document as a properly nested string for the remainder of this paper.

3 XPath

3.1 XPath Introduction

XPath is a means of specifying positions of an XML document. W3C provides an XPath recommendation that contains quite a bit of power, including string manipulation and arithmetical operation support. We will not be discussing the full XPath recommendation in this paper, but rather a logical core of the W3C recommendation that supports the full power to specify location paths in an XML document.

Core XPath was originally introduced in [2]. In their paper, they show that there exist algorithms that evaluate Core Xpath expressions in polynomial time with respect to the document size and query size. They also discuss a main memory implementation of their top-down XPath query processor and show that it performs well in practice. The challenges in efficient XML query processing discussed in the introduction show that a main memory implementation is not a satisfactory solution for all XML documents. We will focus on cXPath expressions and show that a polynomial space VPA construction exists that preserves the meaning of the query and this construction will be suitable for use in efficient *streaming* Xpath query evaluations.

cXPath operates by specifying complex location paths that are the composition of simpler expressions. The simplest expression in cXPath is called a *step*. A step is always of the form $\chi :: a$ or $/\chi :: a$, where χ is an axis and a is the label of every query answer; additionally we may specify an optional *predicate*. Expressions with a predicate specified are of the form $/\chi :: a[\pi]$, where π is an arbitrary cXPath expression. An axis is essentially a binary relation on the XML tree (e.g child, descendent, ancestor etc.). An axis can be either forward or backwards; in a forward axis an answer to the query always occurs *after* the root of a solution's path relative to the document order. An axis that is not forward is said to be *backwards*. A number of streaming XPath engines including XSQ [7] and Vitex [4] provide automation models that capture a subset of XPath containing only forward axes.

3.2 Core XPath

A cXPath expression is generated by the grammar described in figure 2.

$$\begin{gathered} cXPath : path \mid /path \\ \text{path: } step(/step)^* \\ \text{step: } \chi :: a \mid \chi :: a[pred] \\ \text{pred: } cXPath \mid pred\ 'and'\ pred \mid \\ pred\ 'or'\ pred \mid not(pred) \end{gathered}$$

Figure 2: Grammar generating Core XPath. cXPath is the start variable.

The semantics of a cXPath expression are defined by a function $S_{\rightarrow}$. But before we can describe $S_{\rightarrow}$ we must define a few functions. For the following definitions, D is an XML document with nodes labeled by elements of Σ, $N \subseteq nodes(D)$ and $a \in \Sigma$.

The first function we define is a means of passing from a set of nodes in a given document to the set of nodes in the document related to the given set by a specified axis.

Definition 2. *Define a function* $\chi : 2^{nodes(D)} \rightarrow 2^{nodes(D)}$ *where* $\chi(N) = \{n_0 \in nodes(D) \mid \exists n \in N : (n, n_0) \in \chi\}$

Note that χ can be either a binary relation or a function, depending on the context.

Next, we define a function that returns sets of nodes with a specified label.

Definition 3. *Define a function,* $\tau : (\Sigma \cup \{\star\}) \rightarrow 2^{nodes(D)}$ *such that* $\tau(a) = \{n \in nodes(D) \mid$ *node* n *is labeled* $a\}$ *and* $\tau(\star) := nodes(D)$.

We are ready to define the semantics of a cXPath expression, this is described in figure 3. Again, D is an XML document and $N \subseteq nodes(D)$.

Core XPath expressions are all evaluated relative to some *context*, which is essentially a set of nodes from a given document. A context for a step in a location path is the resultant set of nodes from evaluating the step preceding it. For example, consider the cXPath expression $child :: a/parent :: b$. The context for $parent :: b$, is the result of evaluating

$$S_{\rightarrow}[\chi :: a](N) := \chi(N) \cap \tau(a)$$

$$S_{\rightarrow}[/\chi :: a](N) := \chi(\{root\}) \cap \tau(a)$$

$$S_{\rightarrow}[\chi :: a[\pi]](N) := \chi(N) \cap \tau(a) \cap P(\pi)$$

$$S_{\rightarrow}[/\chi :: a[\pi]](N) := \chi(\{root\}) \cap \tau(a) \cap P(\pi)$$

$$S_{\rightarrow}[\pi/\chi :: a[\pi']](N) := \chi(S_{\rightarrow}[\pi](N)) \cap \tau(a) \cap P(\pi')$$

$$S_{\leftarrow}(\chi :: a[\pi]) := \chi^{-1}(\tau(a) \cap P(\pi))$$

$$S_{\leftarrow}(/\pi) = \begin{cases} nodes(D) & \text{if } root \in S_{\leftarrow}(\pi) \\ \varnothing & \text{if } root \notin S_{\leftarrow}(\pi) \end{cases}$$

$$S_{\leftarrow}(\chi :: a[\pi']/\pi) := \chi^{-1}(\tau(a) \cap P(\pi') \cap S_{\leftarrow}(\pi))$$

$$P(\pi \text{ or } \pi') := P(\pi) \cup P(\pi')$$

$$P(\pi \text{ and } \pi') := P(\pi) \cap P(\pi')$$

$$P(not(\pi)) := nodes(D) \cap P(\pi)$$

$$P(\pi) := S_{\leftarrow}(\pi)$$

Figure 3: Semantics defining a cXPath expression.

$child :: a$ which is the set of all a children in the document. For some XML document D, we say that $n \in nodes(D)$ is an *anchor* of the cXPath expression π if $S_{\rightarrow}(\pi, \{n\}) \neq \varnothing$. We say $n \in nodes(D)$ is a *solution* to cXPath expression π if $n \in S_{\rightarrow}(\pi, nodes(D))$

Proposition 2. *Suppose π is a cXPath expression and $N \subseteq nodes(D)$ for an XML document D. And suppose further that $S_{\rightarrow}(\pi, N)$ is non-empty. Then, there is some $n \in N$ that is an anchor for π.*

4 Machine Construction

4.1 Star-Dollar Words

The automation we describe to represent a cXPath expression will accept what we call *star-dollar words*, which we motivate and define in this section.

Consider the simple cXPath expression $/child :: a/child :: b/child :: c$ and XML document D with properly nested string representation w and assume $S_{\rightarrow}(/child :: a/child :: b/child :: c, nodes(D))$ is non empty. The proposition of the last section tells us that there is an anchor $n \in nodes(D)$ to the query. We may conclude that each step returns a non empty set of nodes, for if not there would be a step with an empty context and thus the query would evaluate to the empty set.

This motivates the *dollar* in star-dollar word. While we are processing w the VPA we construct makes a non-deterministic guess that some input character is an anchor for the expression. This guess is the dollar in star-dollar word.

Since we know the label all answer nodes will possess, while processing w we guess that each node with the correct label is a solution to the expression, this guess is the star in star-dollar word.

So each properly nested string corresponding to an XML document will be viewed by the automation as having two distinct marks on the input, a dollar representing a potential anchor and a star representing a potential solution.

Suppose XML document D has nodes labeled by Σ. Then, the *star-dollar alphabet* with respect to document D is defined as $(\Sigma \times \{\star, \_\} \times \{\$, \_\}) \cup \bar{\Sigma}$.

Definition 4. *Suppose w is a properly nested string representing XML document D. We say w is a star-dollar word if there is some position $i \leq length(w)$ such that $w_i \in \Sigma \times \{\star\} \times \{\$, \_\}$ and some position $j \leq length(w)$ such that $w_j \in \Sigma \times \{\star, \_\} \times \{\$\}$.*

In other words, a star-dollar word is a properly nested string with one character marked with a dollar sign and another (possibly the same) position marked with a star. Since we may equivalently talk about trees and properly nested strings, we may view a star-dollar word as a tree with two nodes marked with special characters.

Let us define a few useful functions. Let $w \in WM(\Sigma \times \{\star, \_\} \times \{\$, \_\})$, denote all star-dollar words with respect to properly nested string w as $w(\star, \$)$. Denote the tree represented by w as T_w.

$$\star : w(\star, \$) \rightarrow nodes(T_w)$$

$$\$: w(\star, \$) \rightarrow nodes(T_w)$$

$$\circ : w(\star, \$) \times w(\star, \$) \rightarrow w(\star, \$)$$

The star function returns the node in tree T_w corresponding to the star position in w and the dollar function returns the node corresponding the dollar position. The circle function takes two star-dollar words and returns a star dollar word, where the resultant string has dollar position as in the first component and star position as in the second component and erases other special marks. If the star position from the first component is not equal to the dollar position in the second component the resultant string is empty. For example, if $w_1 = (a, \_, \_)(b, \_, \$)(c, \star, \_)\bar{c}\bar{b}\bar{a}$ and $w_2 = (a, \star, \_)(b, \_, \_)(c, \_, \$)\bar{c}\bar{b}\bar{a}$ then we get $\circ(w_1, w_2) = (a, \star, \_)(b, \_, \$)(c, \_, \_)\bar{c}$ and if $w_1 = w_2 = (a, \star, \_)(b, \_, \$)$, then $\circ(w_1, w_2) = \epsilon$.

4.2 Construction of VPA

Let us describe a formal construction that takes a cXPath expression and produces a VPA that precisely preserves the semantic meaning of the input expression. The idea is to construct VPAs for the simplest cXPath expressions, of the form $\chi :: a$, we will call these automata *simple cXPath automata*. Given a cXPath expression, the *main branch* is that expression without any predicate conditions. The key observation to make is that a solution node to the *main branch* of an expression at each step must serve as an anchor for the predicate expression contained in that step.

We are ready now to provide an example. In this paper the labeling on edges will look something of the form $(\sigma, \_, \_), \sigma$, in the interest of keeping these drawings clear let us agree that if σ is a call character then $(\sigma, \_, \_), \sigma$ means upon reading σ, push σ onto the stack and if σ is a return character than $(\sigma, \_, \_), \sigma$ means upon reading σ pop σ from the stack.

Example 1. Consider the following cXPath expression: $/descendent :: c/child :: a[parent :: b]$, this selects all the a nodes that have a parent b which in turn descend from a c node. We want to parse this expression and from the individual steps, build a VPA that preserves the semantic meaning of expression. The simple cXPath automation for the step $parent :: b$ are described in figure and 7. The automata that captures $/child :: a$ and $descendent :: c$ and are the automaton described in figures 6 and 8 without the loops on the initial states.

Let us focus of the step $/child :: a[parent :: b]$. Take the VPAs that captures $/child :: a$ and $parent :: b$, to created a VPA that captures the whole step, we take a product of the two machines with a modified transition function defined by the following rules: the composite machine may move on a dollar position only if the $/child :: a$ automaton does and we transition on a star position only if the $/child :: a$ automaton does and the $parent :: b$ transitions on a dollar position. Figure 4 captures $/child :: a[parent :: b]$.

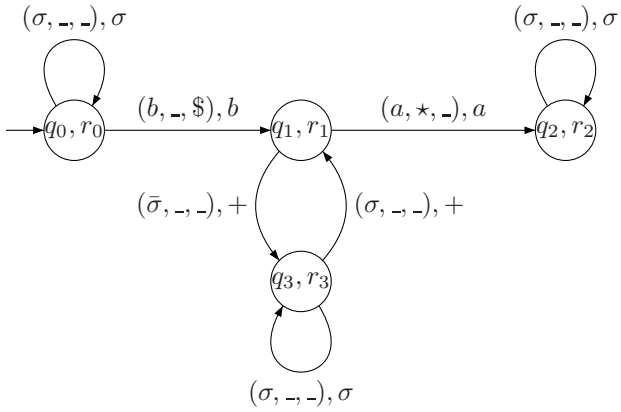

Figure 4: VPA capturing expression $/child :: a[parent : b]$

Let us put it all together and build a VPA that runs successfully if and only the input word corresponds to a tree with the structure described by the given query $/descendent :: c/child :: a[parent :: b]$. Take the VPA representing the step $/descendent :: c$ and $/child :: a[parent :: b]$ take a product and define a new transition function. This transition function will allow transition on a dollar position only if $/descendent :: c$ moves on that dollar position and we transition on a star position only if $/child :: a[parent :: b]$ moves on that star position and we allow moving on unmarked characters only if both machines move on that unmarked character or the $/decsendent :: c$ machine moves on a star position and the $/child :: a[parent :: b]$ machine moves on a dollar position. Figure 5 captures

$/descendent :: c/child :: a[parent :: b]$.

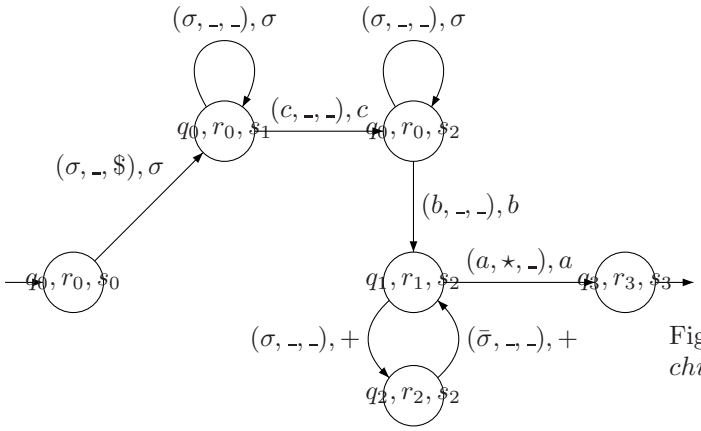

Figure 5: Automata capturing $/descendent :: c/child :: a[parent :: b]$

The construction we provide is inductive. We first provide the *simple cXPath automata* for steps of the form $\chi :: a$. Once given a VPA that captures $\chi :: a$ it is easy to construct a VPA capturing $/\chi :: a$ by simply disallowing looping on an initial state. We will provide only the simple cXPath automation for forward axes. If given a VPA capturing $\chi :: a$ where χ is a forward axis, it is easy to construct the VPA capturing $\chi^{-1} :: a$ by switching the order the VPA looks for an anchor and solution.

Lemma 1. *For each simple cXPath expression, there is a VPA that captures expression with at most 4 states.*

Proof. Figures 6 through 11 provide the constructions. □

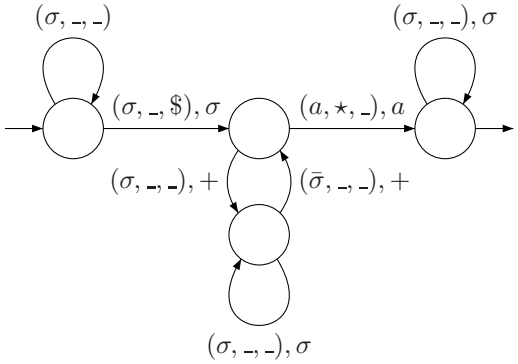

Figure 6: simple cXP automation for the expression $child :: a$

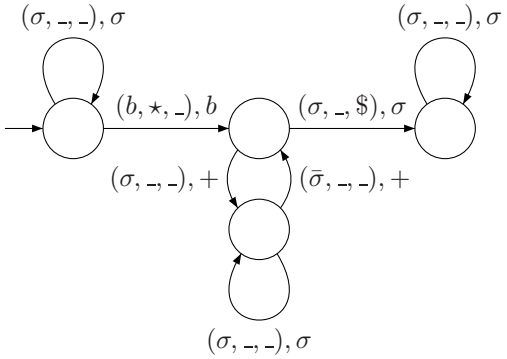

Figure 7: simple cXP automation for the expression $parent :: b$

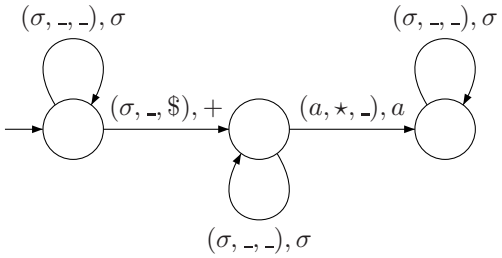

Figure 8: simple cXP automation for the expression $descendent :: a$

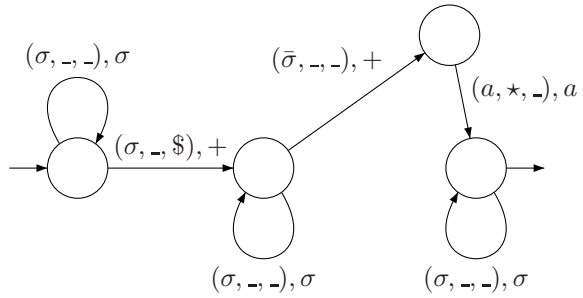

Figure 9: simple cXP automation for the expression $following - sibling :: a$

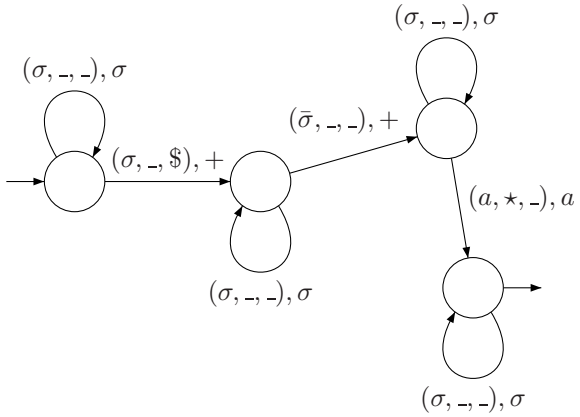

Figure 10: simple cXP automation for the expression $following :: a$

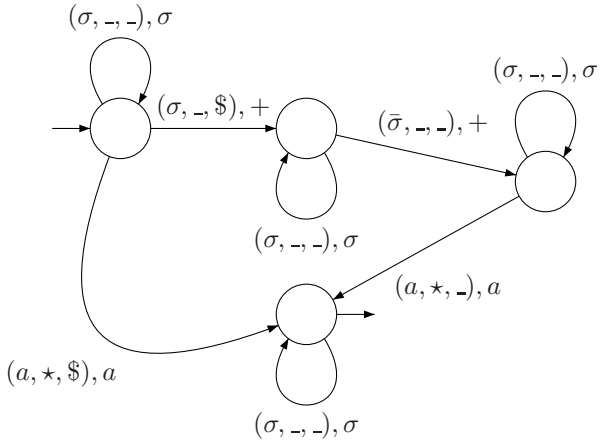

Figure 11: simple cXP automation for the expression $following - or - self :: a$

We now show that for any step, there is a VPA that captures the expression that is the product of the simple cXPath automata.

Lemma 2. *Suppose $\chi_0 :: a_0$ and $\chi_1 :: a_1$ are cXPath expressions captured by A_0 and A_1 respectively, then there is a VPA A that captures $\chi_0 :: a_0[\chi_1 :: a_1]$. Moreover, A is of size $|A_0||A_1|$.*

Proof. We define a new machine A that captures the expression $\chi_0 :: a_0[\chi_1 :: a_1]$, by taking the product of the machines A_0 and A_1 and describe the new transition function where A is allowed to move on a dollar position only if A_0 moves on it, and A is allowed to move on a star position only when A_0 does and A_1 moves on a dollar position. Since this construction is the product of two automata, A must be of size $|A_0||A_1|$. □

Intuitively, this new transition function demands that any solution to $\chi_0 :: a_0$ must act as a anchor for $\chi_1 :: a_1$

Lemma 3. *Suppose π_0, $\chi_1 :: a_1[\pi_1]$ are cXPath expressions captured by VPA A_0 and A_1 respectively, then there is a VPA A that captures $\pi_0/\chi_1 :: a_1[\pi_1]$. Moreover the size of A is $|A_0||A_1|$.*

Proof. Given A_0 and A_1 we construct A by taking the product of A_0 and A_1 and define a new transition function as follows. Move on a blank position if A_0 moves on a star position and A_1 moves on a dollar position or both A_0 and A_1 move on a blank position, move on a dollar position when A_0 does and move on a star position when A_1 does. Since we take the product of two automaton, the resultant automaton has size $|A_0||A_1|$. □

Theorem 1. *For any cXPath expression, there is a VPA that precisely captures the semantics of the expression.*

Proof. Apply the constructions in the above lemmas. □

Predicate expressions containing boolean operators raise no interesting complications since VPAs are closed under all these operations as was shown in [1]. If a cXPath expression does contain a negated predicate the construction we described above yields an exponential growth in state space.

Lemma 4. *Suppose $\chi :: a[not(\pi)]$ is a cXPath expression with A_0 capturing $\chi :: a$ and A_1 capturing $not(\pi)$. The construction above produces an automata A with $2^{|A_1|}|A_0|$ states.*

Proof. This is a consequence of results shown in [1] which states that VPAs are closed under complementation. Essentially we must construct that automata that captures π and then determinize it to create an automata that captures $not(\pi)$ resulting in exponential growth. □

To summarize what we have show: given a cXPath expression without negated predicates, we can construct a polynomial size automaton with the length of the query. If the expression does contain a negated predicate the construction is exponential.

5 Construction Correctness

In this section we show that the construction in the previous section does indeed preserve the semantic meaning of a arbitrary cXPath expression. Suppose that π is a cXPath expression and D is an XML document, and A is VPA described by the construction above. To show that the construction is correct, we want to show that if $w \in L(A)$ is a star-dollar word, then the dollar position is an anchor for a path with the star position as a solution to the query. This idea is expressed precisely in the following theorem.

Theorem 2. $w \in L(A) \iff \star(w) \in S_{\rightarrow}(\pi, \{\$(w)\})$

Proof. Suppose $w \in L(A)$. A is the composition of simple cXPath automation as described in the construction. Suppose $\pi = \pi_0/.../\pi_n$. Denote the VPA capturing each step as A_i and the set of star-dollar words at each step L_{π_i}. There must exist $w_1 \in L(A_1)$ and $w_2 \in L(A_2)$ such that $\circ(w_1, w_2) \in L_{\pi_1/\pi_2}$ for otherwise L_{π_1/π_2} is empty. It is also clear that $\circ(w_1, w_2) \in L_{\pi_1/\pi_2}$ produces a path on D in that $\$(\circ(w_1, w_2)$ is an anchor and $\star(\circ(w_1, w_2))$ is a solution to π_1/π_2. Carrying on, it becomes clear that $\star(w) \in S_{\rightarrow}(\pi, \{\$(w)\})$.

Conversely, suppose $\star(w) \in S_{\rightarrow}(\pi, \{\$(w)\})$. This means that $\$(w)$ is an anchor for the expression π and each $L_{\pi_1}, ..., L_{\pi_n}$ is non-empty. Clearly then, there exist $w_1 \in L_{\pi_1}$ and $w_2 \in L_{\pi_2}$ such that $\circ(w_1, w_2) \in L_{\pi_1/\pi_2}$. Repeating this for each step yields a $w \in L_{\pi_1/.../\pi_n} = L(A)$. □

6 Conclusions

We have presented a novel construction which takes an expression generated by the cXPath grammar and converts that expression into a Visibly Pushdown Automata that accepts a star-dollar word if and only if the dollar position is an anchor of the expression and the star position is a solution to the query. Since cXPath expression have the same path expressing power as XPath, this construction operates on a worthwhile subset of the W3C recommendation. We have exhibited an average case polynomial complexity result for the construct and it is thus suitable for implementation.

7 Future Work

We are currently implementing this construction in Python and are excited so see how it performs in practice.

8 Acknowledgements

I wish to thank Madhusudan Parthasarathy for facilitating my first shot at real live research.

References

[1] R. Alur and P. Madhusudan, Visibly pushdown languages, In Babai [8], pp. 202–211.

[2] G. Gottlob, C. Koch, and R. Pichler, ACM Trans. Database Syst. **30**, 444 (2005).

[3] W3C Report No., , 1999 (unpublished), http://www.w3.org/TR/1999/REC-xpath-19991116.

[4] Y. Chen, S. B. Davidson, and Y. Zheng, An efficient xpath query processor for xml streams, in *ICDE*, edited by L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, p. 79, IEEE Computer Society, 2006.

[5] P. Madhusudan and M. Viswanathan, Query Automata for Nested Words, http://www.faculty.cs.uiuc.edu/∼ madhu/vpquery.pdf.

[6] R. Alur, Marrying words and trees, in *PODS*, edited by L. Libkin, pp. 233–242, ACM, 2007.

[7] F. Peng and S. S. Chawathe, ACM Trans. Database Syst. **30**, 577 (2005).

[8] L. Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, ACM, 2004.