

Query Automata for Nested Words

P. Madhusudan Mahesh Viswanathan

University of Illinois at Urbana-Champaign

Urbana, IL, USA

{madhu, vmahesh}@cs.uiuc.edu

Abstract

We study visibly pushdown automata (VPA) models for expressing and evaluating queries, expressed using MSO formulas, on words with a nesting structure (like XML documents). We define a query VPA model, which is a 2-way deterministic VPA that can mark positions in a document, and show that it is equiexpressive as unary monadic queries. This surprising result parallels a classic result for queries on regular word languages. We also compare our model to query models on unranked trees.

We then consider the algorithmic problem of evaluating, in one pass, the set of all positions satisfying a query in a streaming nested word. We present an algorithm that answers any fixed unary monadic query on a streaming document which uses, at any point, at most space $O(d + I \log n)$, where d is the depth of the document at that point and I is the number of potential answers to the query in the word processed thus far. This algorithm uses space close to the minimal space any streaming algorithm would need, and generalizes to answering n -ary queries.

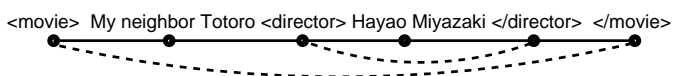
1 Introduction

The general purpose markup language XML (eXtensible Markup Language) represents hierarchically structured data, and has become the standard for data exchange, particularly on the web [27, 1]. An XML document is a text-based linear encoding of tree-structured data. Markups in the text in terms of open- and close-tags are used to create a bracketing structure on the document, and captures the hierarchy of information in the tree. For example, an XML text of the form:

```
<movie> My neighbor Totoro <director> Hayao Miyazaki </director> </movie>
```

encodes a movie title and its director tag is a child node of the movie-tag (since it lies within the `<movie>...</movie>` fragment), indicating that Miyazaki is the director of this particular movie. The XML

document is hence better viewed as a linear structure with a nesting relation:



The study of XML has naturally concentrated on its tree representation; for example, document types are represented using the parse-trees generated by SDTDs, XML query languages like XPath have modalities like “parent” and “child” that refer to the tree edges, and tree automata (over unranked trees) have been used to model and solve decision problems for XML [26, 21, 13, 14, 19].

Nested word automata are a recent invention, and is an automaton that works directly on a linearly structured document with nesting relations as described above [2, 3]. Nested word automata were discovered in the context of *formal verification for software*, where program executions are viewed as nested words, where the nesting relation relates calls to procedures with corresponding returns.

The nested word automaton model works left to right on an XML document, much like a finite automaton on words, but the automaton at a close-tag is allowed to change state by looking at the previous state as well as the state at the open-tag associated with the close-tag. One way to *implement* a nested word automaton on a word is by allowing an automaton access to a stack: the automaton is allowed to store its state at an open-tag and recover the state at the corresponding close-tag. This leads to the definition of *visibly pushdown automata* for XML: a pushdown automaton that pushes onto the stack at open-tags and pops from the stack on close-tags. The class of languages accepted by nested word automata is exactly the same as that accepted by visibly pushdown automata, and we call this the class of *visibly pushdown languages*.

The visibly pushdown automaton model is an appealing formalism for studying and building algorithms for XML. In earlier work [18], we have shown that document type definitions are captured by special classes of visibly pushdown automata, and have used this characterization to build *streaming* algorithms for type-checking and pre-order (and

post-order) typing XML documents. Visibly pushdown automata are particularly effective in building streaming algorithms, as the model naturally captures reading a nested word left-to-right in a streaming fashion, unlike standard tree-automata models that work on the tree represented by an XML document.

In this paper, we study the power of visibly pushdown automaton in expressing and answering *queries* on XML documents. Our first contribution is an automaton model for defining queries, called query visibly pushdown automata (query VPA). A query VPA is a two-way visibly pushdown automaton that can mark positions of a document. When moving right, a query VPA pushes onto the stack when reading open-tags and pops from the stack reading close-tags, like a visibly pushdown automaton. However, when moving left it does the opposite: it pushes onto the stack reading close-tags and pops from the stack reading open-tags. Hence it preserves the invariant that the height of the stack when at a particular position in the document is precisely the number of unmatched calls before that position (which is the number of unmatched returns after that position).

We show that query VPAs have the right expressive power for defining queries by showing that a query is expressible as a unary formula in monadic second-order logic (MSO) if and only if it is implemented by a query VPA. Unary MSO queries form perhaps the largest class of logical queries one can hope to answer by automatic means, and obviously include practical query languages such as XPath. We find it remarkable that the simple definition of two-way VPAs exactly captures all monadic queries. This result parallels a classic result in the theory of automata on finite words, which equates the power of unary monadic queries on (nonnested) words to that of two-way automata on words [10, 22].

While query VPAs are a simple clean model for expressing queries, they do not help answering queries efficiently. They in fact represent one extreme end of the space-time-streaming spectrum: they do not process streaming documents and they are not time-efficient as they reread the document several times, but are incredibly space-efficient, utilizing only space $O(d)$, where d is the depth of the document (with no dependence at all on the length of the document!).

We then turn to streaming algorithms and present our second main result: a streaming algorithm that answers *any* unary monadic query and has efficient time and space data complexity. As far as we know, this is the first time it has been shown that any unary MSO query can be efficiently evaluated on streaming XML.

The algorithm, for any fixed unary monadic query, reads the document in a streaming fashion, and outputs the set of *all* positions that satisfy the query. Note that this is different

from the *filtering* problem that outputs just whether there is *some* position that satisfies the query, which is easy to implement using space $O(d)$, where d is the depth of the document.

When a streaming querying algorithm has read a prefix u of an input document, several positions in u may partially satisfy the query requirement. These potential answers to the query have to be stored as the rest of the document will determine whether these are valid answers to a query. Consequently, we cannot hope for a streaming algorithm that takes constant space (or space $O(d)$). However, a position that has already satisfied the query requirements can be immediately output (to an output tape) and need not be kept in main memory. Formally, let us call a position $i \leq |u|$ an *interesting position* if there is some extension v of u such that i is an answer to the query on uv , but there is another extension v' of u such that i is not an answer to the query on uv' . Summarizing the above argument, a streaming algorithm must keep track of all the interesting positions at u , which would take space (without resort to compression) $O(I \cdot \log n)$, where I is the number of interesting positions at u and $n = |u|$.

The streaming algorithm we present uses only the space argued above as being necessary: after reading u , it uses space $O(d + I \cdot \log n)$ for any *fixed* query (accounting for I pointers into the document), where d is the depth of the document. The algorithm is based on a simulation of a visibly pushdown automaton representing the query on an input document, with care taken to prune away positions that become uninteresting. The notion of basing the space complexity of the algorithm on the set of interesting positions is the algorithmic analog of the *right congruence for finite automata on words* (Myhill-Nerode theorem). We believe that this is an interesting criterion of complexity for streaming query processors, rather than the simplistic linear dependence on the entire length of the document commonly reported (eg. [8]).

The *query complexity* of our algorithm is of course non-elementary for queries expressed in MSO, and is not a fair appraisal of the algorithm. However, if the query is represented by a non-deterministic visibly pushdown automaton with m states, we show the query complexity to be *polynomial* in m . More precisely, the algorithm after reading u uses space $O(m^4 \cdot \log m \cdot d + m^4 \cdot I \cdot \log n)$ where d is the depth of the document at the point u , $n = |u|$, and I is the set of interesting positions at u .

We believe that the streaming algorithm we present here is a very efficient algorithm that holds promise in being useful in practice. Several other schemes of answering XPath queries on streaming XML documents also use some of the aspects of our algorithm: they store partially matched XPath patterns succinctly using a stack, The *states* of the VPA accepting the query forms the natural analog of par-

their matching close-tags. We skip the formal definition, but denote the nested structure associated with a word w as $nw(w) = (\{1, \dots, |w|\}, \{Q_a\}_{a \in \widehat{\Sigma}}, \leq, \nu)$, where the universe is the set of positions in w , each Q_a is a unary predicate that is true at the positions labeled a , the \leq relation encodes the linear order of the word, and ν is a binary relation encoding the nesting edges.

Monadic second-order logic (MSO_ν) over nested structures is defined in the standard manner, with interpreted relations \leq and ν : Formally, fix a countable set of first-order variables FV and a countable set of monadic second-order (set) variables SV . Then the syntax of MSO formulas over $\widehat{\Sigma}$ labeled nested structures is defined as:

$$\begin{aligned} \varphi ::= & x \in X \mid Q_i(x) \mid x \leq y \mid \nu(x, y) \mid \varphi \vee \varphi \mid \neg \varphi \mid \\ & \exists x(\varphi) \mid \exists X(\varphi) \\ & \text{where } x, y \in FV, X \in SV. \end{aligned}$$

Automata on nested words

There are two definitions of automata on nested words which are roughly equivalent: *nested word automata* and *visibly pushdown automata*. In this paper, we prefer the latter formalism. Intuitively, a visibly pushdown automaton is a pushdown automaton that reads a nested word left to right, and pushes a symbol onto the stack when reading open-tags and pops a symbol from the stack when reading a closed tag. Note that a symbol pushed at an open tag is popped at the matching closed tag. Formally,

Definition 1 (VPA). A *visibly pushdown automaton* (VPA) over $(\Sigma, \overline{\Sigma})$ is a tuple $A = (Q, q_0, \Gamma, \delta, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, Γ is a finite stack alphabet, and $\delta = \langle \delta^{\text{open}}, \delta^{\text{close}} \rangle$ is the transition relation, where:

- $\delta^{\text{open}} \subseteq ((Q \times \Sigma) \times (Q \times \Gamma))$;
- $\delta^{\text{close}} \subseteq ((Q \times \overline{\Sigma}) \times \Gamma) \times Q$.

A transition $(q, a, q', \gamma) \in \delta^{\text{open}}$ (denoted $q \xrightarrow{c/\gamma} q'$) is a push-transition, where the automaton reading c changes state from q to q' , pushing γ onto the stack. Similarly, a transition $(q, \bar{a}, \gamma, q') \in \delta^{\text{close}}$ (denoted $q \xrightarrow{\bar{c}/\gamma} q'$) is a pop-transition, allowing the automaton, when in state q reading \bar{c} with γ on the top of the stack, to pop γ off the stack and change state to q' . A *configuration* of a VPA A is a pair $(q, s) \in Q \times \Gamma^*$. If $a \in \Sigma$, we say that $(q_1, s_1) \xrightarrow{a}_A (q_2, s_2)$ if one of the following conditions are true:

- $a = c \in \Sigma$, $s_2 = \gamma.s_1$ and $(q_1, c, q_2, \gamma) \in \delta^{\text{open}}$, or
- $a = \bar{c} \in \overline{\Sigma}$, $s_1 = \gamma.s_2$ and $(q_1, \bar{c}, \gamma, q_2) \in \delta^{\text{close}}$.

Note that the height of the stack after reading a prefix u of a well-matched word w is precisely the number of unmatched calls in u . We extend the definition of \xrightarrow{a}_A to words over $\widehat{\Sigma}^*$ in the natural manner. The language $L(A)$ accepted by VPA A is the set of words $w \in \widehat{\Sigma}^*$ such that $(q_0, \epsilon) \xrightarrow{w}_A (q, \epsilon)$ for some $q \in Q^F$. One important observation about VPAs, made in [2], is that deterministic VPAs are as expressive as non-deterministic VPAs (defined above). Finally, a language L of well-matched words is called a *visibly pushdown language* (VPL) if there some VPA A such that $L = L(A)$.

Monadic queries and automata

A (unary) *query* is a function $f : WM(\widehat{\Sigma}) \rightarrow 2^{\mathbb{N}}$ such that for every $w \in WM(\widehat{\Sigma})$, $f(w) \subseteq [|w|]$. In other words, a query is a function that maps any well-matched word to a set of positions in the word.

A *unary monadic query* is a formula $\varphi(x_0)$ in MSO_ν that has precisely one free variable, the first-order variable x_0 . Such a formula defines a query f_φ : for any word w , $f_\varphi(w)$ is the set of positions i such that the nested structure corresponding to w satisfies $\varphi(x_0)$ when x_0 is interpreted to be the i 'th position. We will say query f is *expressible in MSO_ν* if there is a unary monadic query $\varphi(x_0)$ such that $f = f_\varphi$. We will consider unary monadic queries as the standard way to specify queries on XML documents in this paper (note that core fragments of query languages like XPath are expressible as unary monadic queries).

Any query f over $\widehat{\Sigma}$ -labeled nested structures can be *encoded* as a language of well-matched words over a modified alphabet. If $\widehat{\Sigma} = (\Sigma, \overline{\Sigma})$, then let $\widehat{\Sigma}' = (\Sigma', \overline{\Sigma}')$ where $\Sigma' = \Sigma \cup (\Sigma \times \{*\})$. A *starred-word* is a well-matched word of the form $u(a, *)v$ where $u, v \in \widehat{\Sigma}^*$, i.e. it is a well-matched word over $\widehat{\Sigma}$ where precisely one letter has been annotated with a $*$.

A query f then corresponds to a set of starred-words: $L_*(f) = \{a_1 a_2 \dots a_{i-1} (a_i, *) a_{i+1} \dots a_n \mid i \in f(a_1 \dots a_{i-1} a_i a_{i+1} \dots a_n) \text{ and each } a_j \in \widehat{\Sigma}'\}$. Intuitively, $L_*(f)$ contains the set of all words w where a single position of w is marked with a $*$, and this position is an answer to the query f on w . We refer to $L_*(f)$ as the *starred-language* of f . It is easy to see that the above is a 1-1 correspondence between unary queries and starred-languages.

From results on visibly pushdown automata, in particular the equivalence of MSO_ν formulas and visibly pushdown automata [2], the following lemma follows:

Theorem 1. A query f is expressible in MSO_ν iff $L_*(f)$ is a visibly pushdown language.

Hence we can view unary monadic queries as simply visibly pushdown starred-languages, which will help in many proofs in this paper.

We can now state the main results of this paper:

- We define the automaton model for of *query VPA* for querying XML documents, which is a two-way visibly pushdown automaton that answers unary queries by marking positions in an input word. We show that a unary query is expressible in MSO_ν iff it is computed by some query VPA.
- For any unary monadic query, we show there is an algorithm that works in one pass on a streaming XML document, and outputs the set of positions that form answers to the query. This algorithm works by simulating a visibly pushdown automaton and keeps track of pointers into the streaming document. We show that the space complexity of the algorithm is optimal.

3 Query VPA

The goal of this section is to define an automaton model for querying XML documents called a *query VPA*. A query VPA is a pushdown automaton that can move both left and right over the input string and store information in a stack. The crucial property that ensures tractability of this model is that the stack height of such a machine is pre-determined at any position in the word. More precisely, any query VPA P working over a well-matched input w has the property that, for any partition of w into two strings u and v (i.e., $w = uv$), the stack height of P at the interface of u and v is the same as the number of unmatched open-tags in u (which is the same as the number of unmatched close-tag in v). In order to ensure this invariant, we define the two-way VPA as one that pushes on open-tags and pops on close-tags while moving right, *but pushes on close-tags and pops on open-tags while moving left*. Finally, in addition to the ability to move both left and right over the input, the query VPA can *mark* some positions by entering special *marking states*; intuitively, the positions in a word that are marked will be answers to the unary query that the automaton computes.

We will now define query VPA formally. We will assume that there is a left-end-marker \triangleright and a right-end-marker \triangleleft for the input to ensure that the automaton doesn't fall off its ends; clearly, $\triangleright, \triangleleft \notin \widehat{\Sigma}$.

Definition 2 (Query VPA). *A query VPA (QVPA) over $(\Sigma, \overline{\Sigma})$ is a tuple $P = (Q, q_0, \Gamma, \delta, Q_*, S, C)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $Q_* \subseteq Q$ is a set of marking states, Γ is a finite stack alphabet, (S, C) is a partition of $Q \times \widehat{\Sigma}$, and $\delta = \langle \delta^{open}, \delta^{close}, \delta^{chng} \rangle$ is the transition relation, where:*

- $\delta_{right}^{open} : S \cap (Q \times \Sigma) \rightarrow (Q \times \Gamma)$
- $\delta_{left}^{open} : (S \cap (Q \times \Sigma)) \times \Gamma \rightarrow Q$

- $\delta_{right}^{close} : (S \cap (Q \times \overline{\Sigma})) \times \Gamma \rightarrow Q$
- $\delta_{left}^{close} : S \cap (Q \times \overline{\Sigma}) \rightarrow (Q \times \Gamma)$
- $\delta^{chng} : C \cup (Q \times \{\triangleright, \triangleleft\}) \rightarrow Q$

The δ_{right}^{open} and δ_{left}^{close} functions encode push-transitions of the automaton reading an open-tag and moving right, and reading a close-tag and moving left, respectively. The δ_{left}^{open} and δ_{right}^{close} functions encode pop-transitions when the automaton reads an open-tag and moves left, and reads a close-tag and moves right, respectively. On the other hand, the δ^{chng} function encodes transitions where the automaton changes the direction of its head movement. Observe that we force the automaton to change direction whenever it reads either \triangleright or \triangleleft . Note that the query VPA has no final states. Finally, the definition above describes a deterministic model, which we will focus on in this paper. The non-deterministic version of the above automaton can also be defined, but they do not increase the expressive power.

We will now define the execution of a query VPA on a word $x = \triangleright w \triangleleft$. A *configuration* of the query VPA is a tuple $\langle p, d, q, s \rangle$, where $p \in [|x|]$ is the position in the input currently being scanned, $d \in \{left, right\}$ is the direction in which the head moving currently, $q \in Q$ is the current state, and $s \in \Gamma^*$ is the current stack contents. The *initial configuration* is $\langle 1, right, q_0, \epsilon \rangle$, i.e., initially the automaton is reading the leftmost symbol of w (not \triangleright), it is moving right, in the initial state, with an empty stack. A *run* is a sequence c_0, c_1, \dots, c_n , where c_0 is the initial, and for each i , if $c_i = \langle p_i, d_i, q_i, s_i \rangle$ and $c_{i+1} = \langle p_{i+1}, d_{i+1}, q_{i+1}, s_{i+1} \rangle$, then one of the following holds

- If $(q_i, x[p_i]) \in S \cap (Q \times \Sigma)$, and $d_i = right$ then $d_{i+1} = d_i, p_{i+1} = p_i + 1, q_{i+1} = q$ and $s_{i+1} = \gamma s_i$, where $\delta_{right}^{open}(q_i, x[p_i]) = (q, \gamma)$
- If $(q_i, x[p_i]) \in S \cap (Q \times \overline{\Sigma})$, and $d_i = right$ then $d_{i+1} = d_i, p_{i+1} = p_i + 1, q_{i+1} = q$ and $s_i = \gamma s_{i+1}$, where $\delta_{right}^{close}(q_i, x[p_i], \gamma) = q$
- If $(q_i, x[p_i]) \in S \cap (Q \times \Sigma)$, and $d_i = left$ then $d_{i+1} = d_i, p_{i+1} = p_i - 1, q_{i+1} = q$ and $s_i = \gamma s_{i+1}$, where $\delta_{right}^{open}(q_i, x[p_i], \gamma) = q$
- If $(q_i, x[p_i]) \in S \cap (Q \times \overline{\Sigma})$, and $d_i = left$ then $d_{i+1} = d_i, p_{i+1} = p_i - 1, q_{i+1} = q$ and $s_{i+1} = \gamma s_i$, where $\delta_{right}^{close}(q_i, x[p_i]) = (q, \gamma)$
- If $(q_i, x[p_i]) \in C$ then $s_i = s_{i+1}$, and $q_{i+1} = \delta^{chng}(q_i, x[p_i])$. To define the new position and direction, there are two cases to consider. If $d_i = right$ then $d_{i+1} = left$ and $p_{i+1} = p_i - 1$. On the other hand, if $d_i = left$ then $d_{i+1} = right$ and $p_{i+1} = p_i + 1$.

Observe that the way the run is defined, the stack height in any configuration is determined by the word w . More precisely, in any configuration $c = \langle p, d, q, s \rangle$ of the run with $p \in \{1, \dots, |w|\}$, if $d = \textit{right}$ then $|s|$ is equal to the number of unmatched open-tags in the word $x[1] \cdots x[p-1]$ (which is the same as the number of unmatched close-tags in $x[p] \cdots x[|w|]$). On the other hand, if $d = \textit{left}$ then $|s|$ is equal to the number of unmatched open-tags in the word $x[1] \cdots x[p]$. When scanning the left-end-marker ($p = 0$) or the right-end-marker ($p = |x| - 1$), the stack height is always 0.

Finally, the query VPA P is said to *mark* a position j in a well-matched word w , where $1 \leq j \leq |w|$, if there is some run $c_0 \dots c_n$ of P on the input $\triangleright w \triangleleft$ such that for some i , $c_i = \langle j, d, q, s \rangle$ where $q \in Q_*$. The query implemented by the query VPA P is the function f_P , where $f_P(w)$ is the set of all positions marked by P when executed on $\triangleright w \triangleleft$.

We now prove the main theorem of this section: the set of queries implemented by query VPA is precisely the set of unary monadic queries.

Theorem 2. *A query f is expressible in MSO_v if and only if there is query VPA P such that $f_P = f$.*

Proof.

Translating monadic queries to query VPA:

Let f be a monadic unary query. From Theorem 1, we know that there is a deterministic VPA A such that $L_*(f) = L(A)$. This suggests a very simple algorithm that will mark all the answers to query f by repeatedly simulating A on the document w . First the algorithm will simulate the VPA A on the word w , assuming that the starred position is the rightmost symbol of w ; the algorithm marks position $|w| - 1$ of w only if A accepts. Then the algorithm simulates A assuming that the starred position is $|w| - 2$, and so on, each time marking a position if the run of A on the appropriate starred word is accepting. A naïve implementation of this algorithm will require maintaining the starred position, as well as the current position in the word that the simulation of A is reading, and the ability to update these things. It is unclear how this additional information can be maintained by a query VPA that is constrained to update its stack according to whether it is reading an open-tag or a close-tag. The crux of the proof of this direction is demonstrating that this can indeed be accomplished. While we draw on ideas used in a similar proof for queries on regular word languages (see [22] for a recent exposition), the construction is more involved due to the presence of a stack.

Before giving more details about the construction, we will give two technical constructions involving VPAs. First given any VPA $B = (Q, q_0, \Gamma, \delta, F)$ there is a VPA B' with a canonical stack alphabet that recognizes the same language; the VPA $B' = (Q, q_0, Q \times \Sigma, \delta', F)$ which pushes

(q, c) whenever it reads an open-tag c in state q . Details of this construction can be found in [2]. Next, given any VPA $B = (Q, q_0, \Gamma, \delta, F)$, there is a VPA $B^{\textit{pop}}$ recognizing the same language, which remembers *the symbol last popped since the last unmatched open-tag* in its control state. We can construct this: $B^{\textit{pop}} = (Q \times (\Gamma \cup \{\perp\}), (q_0, \perp), \Gamma, \delta', F \times (\Gamma \cup \{\perp\}))$, where the new transitions are as follows. If $q \xrightarrow{c/\gamma_1} B q'$ then $(q, \gamma) \xrightarrow{c/\gamma_1} B^{\textit{pop}} (q', \perp)$. If $q \xrightarrow{\bar{c}/\gamma_1} B q'$ then $(q, \gamma) \xrightarrow{\bar{c}/\gamma_1} B^{\textit{pop}} (q', \gamma_1)$.

For the rest of this proof, let us fix A to be the deterministic VPA with a canonical stack alphabet recognizing $L_*(f)$, and $A^{\textit{pop}}$ to be the (deterministic) version of A that remembers the last popped symbol in the control state. We will now describe the query VPA P for f . Let us fix the input to be $\triangleright w \triangleleft$, where $w = a_1 a_2 \cdots a_n$.

P will proceed by checking for each i , i starting from n and decremented in each phase till it becomes 1, whether position i is an answer to the query. To do this, it must check if A accepts the starred word where the star is in position i . P will achieve this by maintaining an invariant, which we describe below.

The Invariant. Let $w = a_1 \dots a_n$, and consider a position i in w . Let $w = ua_i v$ where $u = a_1 \dots a_{i-1}$ and $v = a_{i+1} \dots a_n$.

Recall that the suffix from position $i + 1$, v , can be uniquely written as $w_k \bar{c}_k w_{k-1} \bar{c}_{k-1} \cdots w_1 \bar{c}_1 w_0$, where for each j , w_j is a well-matched word, and $\bar{c}_k, \dots, \bar{c}_1$ are the unmatched close-tags in v .

In phase i , the query VPA will navigate to position i with stack σ such that (a) its control has information of a pair (q, γ) such that this state with stack σ is the configuration reached by $A^{\textit{pop}}$ on reading the prefix u , and (b) its control has the set B of states of A that is the precise set of states q' such that A accepts v from the configuration (q', σ) . Hence the automaton has a summary of the way A would have processed the unstarred prefix up to (but not including) position i and a summary of how the unstarred suffix from position $i + 1$ would be processed. Under these circumstances, the query VPA can very easily decide whether position i is an answer to the query — if A on reading $(a_i, *)$ can go from state q to some state in B , then position i must be marked.

Technically, in order to ensure that this invariant can be maintained, the query VPA needs to maintain more information: a set of pairs of states of A , S , that summarizes how A would behave on the first well-matched word in v (i.e. w_k), a stack symbol *StackSym* that accounts for the difference in stack heights, and several components of these in the stack to maintain the computation. We skip these technical details here; the Appendix has a formal description of the invariant and the mechanism to maintain it.

Marking position i . If $a_i \in \Sigma$ then the query VPA P will mark position i iff $q \xrightarrow{(a_i,*)/(q,(a_i,*))}_A q'$ where $q' \in B$. Similarly, if $a_i \in \bar{\Sigma}$ then P marks i iff $q \xrightarrow{(a_i,*)/\gamma'}_A q'$ with $q' \in B$.

Maintaining the Invariant. Initially the query VPA P will simulate the VPA A^{pop} on the word w from left to right. Doing this will allow it to obtain the invariant for position n . So what is left is to describe how the invariant for position $i - 1$ can be obtained from the invariant for position i . Determining the components of the invariant, except for the new control state of A^{pop} , are easy and follow from the definitions; the details are deferred to the appendix. Computing the new state of A^{pop} at position $i - 1$ is interesting, and we describe this below.

Determining the state of A^{pop} . Recall that we know the state of A^{pop} after reading $a_1 \cdots a_{i-1}$, which is (q, γ) . We need to compute the state (q', γ') of A^{pop} after reading $a_1 \cdots a_{i-2}$. The general idea is as follows. The query VPA P will simulate A^{pop} backwards on symbol a_{i-1} , i.e., it will compute $Prev = \{p \mid p \xrightarrow{a_{i-1}}_{A^{pop}} (q, \gamma)\}$. If $|Prev| = 1$ then we are done. On the other hand, suppose $|Prev| = k > 1$. In this case, P will continue simulating A^{pop} backwards on symbols a_{i-2}, a_{i-3} and so on, while maintaining for each state $p \in Prev$ the set of states of A^{pop} that reach p . If at some position j the sets associated with all states $p \in Prev$ become empty except one, or $j = 1$ (we reach the beginning of the word), then we know which state $p \in Prev$ is state of A^{pop} after reading $a_1 \cdots a_{i-2}$ — it is either the unique state whose set is non-empty or it is state whose set includes the initial state. However P now needs to get back to position $i - 1$. This is done by observing the following. We know at position $j + 1$ at least 2 different threads of the backward simulation are still alive. Position $i - 1$ is the unique position where these 2 threads converge if we now simulate A^{pop} forwards. The idea just outlined works for queries on regular word languages, but does not work for query VPA due to one problem. If we go too far back in the backwards simulation (like the beginning of the word), we will lose all the information stored in the stack. Therefore, we must ensure that the backward simulation does not result in the stack height being lower than what it is supposed after reading $a_1 \cdots a_{i-2}$. To do this we use the special properties of the VPA A^{pop} . Observe that if we go backwards on an unmatched open-tag (in $a_1 \cdots a_{i-1}$), the state of the VPA A at the time the open-tag was read is on the stack. Thus, the state of A^{pop} after reading the open-tag is uniquely known. Next if $a_{i-1} \in \bar{\Sigma}$ is a matched close-tag, then since we keep track of the last symbol popped after reading a_{i-1} , we know the symbol that was popped when a_{i-1} was read, which allows us to know

the state of A , when it read the open-tag that matches a_{i-1} . These two observations ensure that we never pop symbols out of the stack. The details are as follows.

$a_{i-1} \in \Sigma$: Simulate backwards until (in the worst case) the rightmost unmatched open-tag symbol in the word $a_1 \cdots a_{i-2}$, and then simulate forward to determine the state (q', γ') as described above.

$a_{i-1} \in \bar{\Sigma}$: γ is symbol that is popped by A^{pop} when it reads a_{i-1} . So γ encodes the state of A when the matching open-tag a_j to a_{i-1} was read. So simulate backwards until a_j is encountered and then simulate forwards.

This completes the description of the query VPA.

Translating query VPA to monadic queries:

We now show the converse direction: For any query VPA A , there is an MSO_ν formula $\varphi(x)$ that defines the same query that A does.

Let A be a query VPA. We translate the query A defines into an MSO_ν formula by translation through several intermediate stages: (a) we first construct a two-way (non-marking) VPA that accepts the starred-language of the query, (b) using a standard mapping of nested words into binary trees called *stack trees* [2], we build a pushdown tree walking automaton that accepts the trees corresponding to the starred language of the query, (c) using the fact that tree-walking pushdown automata define precisely regular tree languages [16], which are equivalent to MSO on trees, we build a unary MSO formula on trees that captures the query, and finally we translate this MSO formula on trees to a unary MSO_ν formula on nested words. Due to interests of space we delegate further details of this proof to the Appendix. \square

4 An algorithm for querying streaming XML

In this section, we exhibit an efficient algorithm for answering any unary monadic query on *streaming XML* documents. A streaming algorithm processing XML is one which reads a document, left to right, letter-by-letter, and cannot look back at any part of its input (unless it stores it), and must output the positions of a document that satisfy the query.

Our optimized streaming algorithm will be presented in stages to clearly outline its development. First we will consider the problem of *filtering XML* documents. The problem of filtering requires one to determine if there is *some* position in a streaming XML document that satisfies a unary monadic query. Note that in filtering, we do not insist that the automaton report which exact position(s) satisfy the

query. We will build a deterministic VPA to solve the filtering problem. Next we will present a (unoptimized) querying algorithm based on the deterministic VPA for filtering. Finally we will outline improvements that optimize the space requirements of the streaming algorithm.

4.1 Filtering

A monadic query $\varphi(x_0)$ implementing a query f can be translated to a visibly pushdown language of starred words $L_*(f)$ (see Theorem 1 above). Let A_f be a VPA that accepts $L_*(f)$. In fact, let us assume that A_f accepts all prefixes u in $L_*(f)$ such that every extension of u to a well-matched word is in $L_*(f)$. The filtering automaton we want to build must accept the set of words $w \in WM(\widehat{\Sigma})$ such that there is some way to annotate one letter of w with a $*$ so that the resulting starred word is accepted by A_f .

The deterministic automaton we build will hence keep track of various runs of A_f for each possibly annotation of w with a $*$. Intuitively, this is easy: we build a non-deterministic VPA that guesses a position of w to be annotated with a $*$, and simulate the VPA A_f on this annotated word. We however go through the details of the construction of this deterministic VPA as it will be the basis of our algorithm.

Lemma 1. *Let $\varphi(x_0)$ be a unary monadic query defining a query f . Then we can effectively compute a deterministic VPA that accepts the language of words w such that $f(w) \neq \emptyset$.*

Proof. Our construction heavily depends on the determinization construction for VPAs [2] (we encourage the reader to see that construction in order to understand the more intricate construction below.)

First, we construct A_f , a nondeterministic VPA accepting $L_*(f)$. Without loss of generality we assume that A_f keeps track in its state whether it has seen a $*$ -ed letter or not, and if it has seen, it will have no move on a $*$ -ed letter (if A_f does not have this form, it can be transformed, at most doubling the number of states, to gain this form). Thus, let $A_f = (Q \cup P, q_0, \Gamma, \delta, F)$, where the states have been partitioned into sets Q and P — Q is the set of states that encode the fact that a $*$ -ed letter has not been seen, and P is the set of states reached if a $*$ -ed letter has been read. Thus, $q_0 \in Q$ and $F \subseteq P$. In what follows, we will adopt the notation convention to refer to states in Q by q, q', q_1, \dots , the states in P by p, p', p_1, \dots , and the states in $Q \cup P$ by r, r', r_1, \dots .

The idea behind the deterministic construction is as follows. Recall that any string u that is a prefix of a well-matched word w can be uniquely written as $w_0 c_1 w_1 c_2 \dots c_k w_k$, where c_1, \dots, c_k are the unmatched open-tags in u and w_i s are well-matched words; thus, the

stack after reading u will have height k . Any execution of the non-deterministic VPA A_f on u will look like

$$q_0 \xrightarrow{w_0} r_1 \xrightarrow{c_1/\gamma_1} r_2 \xrightarrow{w_1} r_3 \xrightarrow{c_2/\gamma_2} \dots \xrightarrow{w_k} r_{2k+1}$$

Hence, the deterministic simulation keeps track of sets of possible *summaries*. A summary is a pair of states (r, r') such that A_f goes from r to r' on reading a well-matched word. Thus, a summary (r, r') captures all non-deterministic runs that start at r (after the last unmatched open-tag) and end in r' .

For the machine A_f there are 4 kinds of relevant summaries: (q, q') -summary that correspond to never having seen a $*$, (q, p) -summary that correspond to having seen a $*$ in the current level, (p, p') -summary that correspond to having seen a $*$ at some previous level, and (p, p') -summary that correspond to having seen a $*$ at the last unmatched open-tag. In the last case, (when we see a $*$ at the last unmatched open-tag) we will denote by $(p, p', *)$. The deterministic machine $B_f = (S, s_0, S \times \Sigma, \delta', F')$ will track all possible summaries.

We defer the formal definition of the construction to the Appendix.

We augment the above automaton so that it accepts only when the stack is empty. Since the set of final states is the set of states that have some summary that takes the automaton from the initial state to a final state, it follows that B_f accepts $L_*(f)$. Note that B_f is deterministic and has at most $O(2^{n^2})$ states, where n is the number of states in A_f . \square

4.2 Querying Algorithm

Our first streaming algorithm simply simulates the machine B_f , constructed in Lemma 1, on the document and at the same time keeps track of $*$ -ed positions. In more detail, the algorithm will associate with every (q, p) -summary (and $(p, p', *)$ -summary) the set $positions(q, p)$ (or $positions(p, p', *)$) of positions x since the last unmatched open-tag (or the position of the last unmatched open-tag), such that the word with $*$ at position x has the summary (q, p) (or $(p, p', *)$). This set of positions will be updated as the summaries are updated. The answers to a query will be all the positions associated with summaries of the form (q_0, p) where p is a final state of A_f . The detailed algorithm is shown in Figure 1 (ignore the line “prune(*Stack*)” for now). Notice that this algorithm does not explicitly construct the machine B_f . Hence its space requirement is not exponential in A_f . The total space used is $O(m^2 \cdot d \cdot \log m + m^2 \cdot n \cdot \log n)$, where m is the size of A_f , d is the depth of the document being read, and n is the length of the document. The space analysis is as follows. The stack has height d , with m^2 summaries in each level taking space $O(m^2 \cdot d \cdot \log m)$ space. Each $*$ -ed position

appears only in summaries corresponding to its level. Moreover, there can be at most n positions, each taking $\log n$ bits, giving as an additional space of $O(m^2 \cdot n \cdot \log n)$. Processing each symbol only take $O(m^2)$ time, giving us a total running time of $O(m^2 \cdot n)$.

4.3 Optimizing the Algorithm

We will now present improvements to the basic algorithm which will slightly increase the per-tag processing time, while provably ensuring that the only positions kept track of at anytime are those that could potentially be answers to the query. We will start by first describing the procedure “prune” (in Figure 1) that prunes the stack after every symbol is read.

Pruning the Stack. As the streaming algorithm simulates the deterministic VPA B_f , the runs on some of the $*$ -ed positions will die. However, the basic algorithm will not necessarily remove these positions from the stack; the algorithm will remove a position i , whose run has died, only when it comes back to the stack level where there is a summary recording position i . The procedure prune in the algorithm in Figure 1 ensures that we remove all positions as soon as their runs end. Here is the idea. Let the current state be the set of summaries s and the top of the stack be (s_1, a) . Consider a summary $(r_1, r_2) \in s_1$. Let $N(r_2, a)$ denote all the states r reached from r_2 on reading the symbol a . Now if there are no summaries of the form $(r, r') \in s$, where $r \in N(r_2, a)$ then we know that all runs that had a summary (r_1, r_2) in the previous level have ended, and so can be removed from the top of the stack (as well as all the positions associated with (r_1, r_2)). After removing all unnecessary summaries from the top of the stack, we can look at the next level and so on. This is what procedure prune does. The total time taken by prune is proportional to the size of the stack which is at most $O(m^2 d)$, where m is the size of automata A_f .

Pruning the stack after every symbol is processed ensures (proved later in this section) that starred positions i , all of whose A_f runs have died are no longer stored. However, the resulting algorithm may still store unnecessary positions. Consider a position i such that the word with $*$ at i has a run in A_f , but that run can never be extended to an accepting run. Such positions will still be maintained by our algorithm. We now present a technical construction involving VPAs that allows us to address this issue.

Lemma 2. *For any VPA A , there is VPA $\text{ValidPref}(A)$ such that $L(A) = L(\text{ValidPref}(A))$ and for any prefix u of a well-matched word, $\text{ValidPref}(A)$ has some execution on u if and only if there exists v such that $uv \in L(A)$. If m is the size of A then $\text{ValidPref}(A)$ has size at most m^2 .*

Proof. See Appendix. \square

Our optimized algorithm is essentially the one in Figure 1 under the provision that the automaton A_f it simulates is the automaton $\text{ValidPref}(A_f)$. In the rest of the section we will prove its optimality. We start by an important definition that allows us to characterize the algorithms optimality.

Definition 3. *Consider a document $w = a_1 a_2 \dots a_n$. A position i is said to be interesting at time k ($i \leq k$) for query f if there is some string v such that $uv \in L_*(f)$, where $u = a_1 a_2 \dots a_{i-1}(a_i, *) \dots a_k$, and there is some string v' such that $uv' \notin L_*(f)$. In other words, there is some extension of the document after k tags such that position i would be an answer to query f , though i may not be an answer for all extensions. The set of all interesting positions at k in w with respect to f will be denoted by $I_k^f(w)$.*

Lemma 3. *Consider the streaming algorithm that performs a deterministic simulation of the automaton $\text{ValidPref}(A_f)$ and prunes the stack after reading every tag. If after reading the string u , there is some summary (q, p) (or $(p, p', *)$) such that $i \in \text{positions}(q, p)$ (or $i \in \text{positions}(p, p', *)$) then i is interesting at that point.*

Proof. Consider the word v which is u with a star at position i , and let $w_0 c_1 w_1 \dots c_k w_k$ its unique decomposition into unmatched open-tags and well-matched words. Let position i appear in the well-matched word w_j (the same ideas can be applied when i is one of the unmatched open-tags). Since (q, p) was not pruned, it means there is a run of the form $q_0 \xrightarrow{w_0} q_1 \xrightarrow{c_1} \dots q \xrightarrow{w_j} p \dots \xrightarrow{w_k} p_n$. Further since the algorithm is simulating $\text{ValidPref}(A_f)$, it follows that there is some v' such that $vv' \in L_*(f)$. Since A_f accepts all prefixes u of $L_*(f)$ such that all extensions of u are in $L_*(f)$, by definition i is interesting. \square

Analysis of the Algorithm. Pruning increases the per-tag processing time to $O(m_1^2 d)$, where m_1 is the size of $\text{ValidPref}(A_f)$. However, the space requirements are provably less. Let i be the maximum over all k of $|I_k^f(w)|$. The total space required is $O(m_1^2 d \log m_1)$ (for stack) plus $O(im_1^2 \log n)$ (to store the positions). We know from Lemma 2 that m_1 is at most m^2 , which gives us the bounds in terms of A_f .

5 Conclusions

An interesting future direction will be to analyze our algorithm’s query complexity for particular fragments of XPath, and compare them with the best known upper bounds in the literature.

Querying XML documents is often followed by outputting parts of the document related to the query positions. The algorithm can be extended to answer n -ary queries, and will help in this regard. We plan to explore XML transducers based on VPAs that produce XML documents from input streams.

There have been some recent results in *minimizing* visibly pushdown automata [17, 9]. Though not efficiently minimizable in general, VPAs are minimizable when some part of its modular structure is fixed in advance (for example, if we decide to process each tag of document using separate modules). We plan to revisit the problem of filtering XML data against a large set of queries, and explore whether the minimization results help in constructing faster algorithms.

In conclusion, we believe that visibly pushdown automata are a simple elegant model for processing XML documents, especially in streaming applications. We believe the insights gained from their theory will result in designing fast algorithms for processing XML. In this vein, an efficient implementation of the algorithm presented here for querying streaming XML would be interesting.

References

- [1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.
- [2] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211. ACM Press, 2004.
- [3] R. Alur and P. Madhusudan. Adding nesting structure to words. In *DLT*, LNCS 4036, pages 1–13, 2006.
- [4] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the memory requirements of xpath evaluation over xml streams. In A. Deutsch, editor, *PODS*, pages 177–188. ACM, 2004.
- [5] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in query evaluation over xml streams. In C. Li, editor, *PODS*, pages 216–227. ACM, 2005.
- [6] C. Barton, P. Charles, D. Goyal, M. Raghavachari, and M. Fontoura. Streaming xpath processing with forward and backward axes. *ICDE*, 00:455, 2003.
- [7] A. Berlea. Online Evaluation of Regular Tree Queries. *Nordic Journal of Computing*, 13(4):1–26, 2006.
- [8] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient xpath query processor for xml streams. In *ICDE '06*, page 79. IEEE Computer Society, 2006.
- [9] P. Chervet and I. Walukiewicz. Minimizing variants of visibly pushdown automata. In L. Kucera and A. Kucera, editors, *MFCS*, volume 4708 of *Lecture Notes in Computer Science*, pages 135–146. Springer, 2007.
- [10] J. Engelfriet and H. J. Hoogeboom. Mso definable string transductions and two-way finite-state transducers. *ACM Trans. Comput. Logic*, 2(2):216–254, 2001.
- [11] J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors. *VLDB 2003*. Morgan Kaufmann, 2003.
- [12] M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees (extended abstract). In *LICS*, pages 188–. IEEE Computer Society, 2003.
- [13] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 173–189, London, UK, 2003. Springer-Verlag.
- [14] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD*, pages 419–430. ACM Press, 2003.
- [15] V. Josifovski, M. Fontoura, and A. Barta. Querying xml streams. *The VLDB Journal*, 14(2):197–210, 2005.
- [16] T. Kamimura and G. Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51, 1981.
- [17] V. Kumar, P. Madhusudan, and M. Viswanathan. Minimization, learning, and conformance testing of boolean programs. In C. Baier and H. Hermans, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2006.
- [18] V. Kumar, P. Madhusudan, and M. Viswanathan. Visibly pushdown automata for streaming XML. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *WWW*, pages 1053–1062. ACM, 2007.
- [19] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *VLDB*, pages 227–238, 2002.
- [20] A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 134–145, London, UK, 1998. Springer-Verlag.
- [21] F. Neven. Automata, logic, and XML. In *CSL*, pages 2–26, London, UK, 2002. Springer-Verlag.
- [22] F. Neven and T. Schwentick. Query automata. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 205–214, New York, NY, USA, 1999. ACM Press.
- [23] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against xml streams. *ICDE*, 00:702, 2003.
- [24] F. Peng and S. S. Chawathe. Xpath queries on streaming data. In *SIGMOD '03*, pages 431–442. ACM Press, 2003.
- [25] P. Ramanan. Evaluating an XPath Query on a Streaming XML Document. In *Proceedings of the International Conference on Management of Data*, pages 41–52, 2005.
- [26] V. Vianu. XML: From practice to theory. In *SBBD*, pages 11–25, 2003.
- [27] World Wide Web Consortium. Extended Markup Language (XML). <http://www.w3.org>.

Appendix

Given: $A_f = (Q \cup P, q_0, \Gamma, \delta, F)$, a VPA accepting $L_*(f)$

Initial State: $State = \{q_0, q_0\}$, and $Stack = \epsilon$

On reading a symbol $a \in \widehat{\Sigma}$

$State' = \emptyset$

case $a = c \in \Sigma$

push $(State, a)$ onto stack

for each $(r_1, r_2) \in State, \gamma \in \Gamma$ such that $r_2 \xrightarrow{c/\gamma} r'$
 $State' = State' \cup \{(r', r')\}$

for each $(q_1, q_2) \in State, \gamma \in \Gamma$ such that $q_2 \xrightarrow{(c,*)/\gamma} p$
 $State' = State' \cup \{(p, p, *)\}$
 $positions(p, p, *) = current_position$

case $a = \bar{c} \in \bar{\Sigma}$

pop $(State_1, c')$; if $c' \neq c$ stop processing and reject

for each $(q, q_1) \in State_1, (q_2, q_3) \in State, \gamma \in \Gamma$ such that $q_1 \xrightarrow{c/\gamma} q_2$ and $q_3 \xrightarrow{\bar{c}/\gamma} q'$
 $State' = State' \cup \{(q, q')\}$

for each $(p, p_1) \in State_1, (p_2, p_3) \in State, \gamma \in \Gamma$ such that $p_1 \xrightarrow{c/\gamma} p_2$ and $p_3 \xrightarrow{\bar{c}/\gamma} p'$
 $State' = State' \cup \{(p, p')\}$

for each $(p, p_1, *) \in State_1, (p_2, p_3) \in State, \gamma \in \Gamma$ such that $p_1 \xrightarrow{c/\gamma} p_2$ and $p_3 \xrightarrow{\bar{c}/\gamma} p'$
 $State' = State' \cup \{(p, p')\}$
 $positions(p, p', *) = positions(p, p_1, *)$

for each $(q, p_1) \in State_1, (p_2, p_3) \in State, \gamma \in \Gamma$ such that $p_1 \xrightarrow{c/\gamma} p_2$ and $p_3 \xrightarrow{\bar{c}/\gamma} p\}$
 $State' = State' \cup \{(q, p)\}$
 $positions(q, p) = positions(q, p) \cup positions(q, p_1)$

for each $(q, q_1) \in State_1, (q_2, p_3) \in State, \gamma \in \Gamma$ such that $q_1 \xrightarrow{c/\gamma} q_2$ and $p_3 \xrightarrow{\bar{c}/\gamma} p\}$
 $State' = State' \cup \{(q, p)\}$
 $positions(q, p) = positions(q, p) \cup positions(q_2, p_3)$

for each $(q, q_1) \in State_1, (p_2, p_3, *) \in State, \gamma \in \Gamma$ such that $q_1 \xrightarrow{(c,*)/\gamma} p_2$ and $p_3 \xrightarrow{\bar{c}/\gamma} p\}$
 $State' = State' \cup \{(q, p)\}$
 $positions(q, p) = positions(q, p) \cup positions(p_2, p_3, *)$

for each $(q, q_1) \in State_1, (q_2, q_3) \in State, \gamma \in \Gamma$ such that $q_1 \xrightarrow{c/\gamma} q_2$ and $q_3 \xrightarrow{(\bar{c},*)/\gamma} p\}$
 $State' = State' \cup \{(q, p)\}$
 $positions(q, p) = positions(q, p) \cup current_position$

$State = State'$

prune($Stack$)

On reading the entire document, for each $(q_0, p) \in State$ with $p \in F$ output $positions(q_0, p)$

Figure 1. Streaming Algorithm for Querying

A Translating Monadic Queries to query VPA

The Invariant. Just before deciding whether to mark position i or not, P will be in configuration $\langle p, d, m, \sigma \rangle$ where $p = i$. Observe that depending on the direction d and on whether a_i is an open-tag or a close-tag, the stack height in this configuration will either be k or $k - 1$ (if $a_i \in \Sigma$) or k or $k + 1$ (if $a_i \in \bar{\Sigma}$). The direction d will be picked to be so that the height is the minimum of the two possibilities for symbol a_i . The control state m will be of the form $\langle (q, \gamma), StackSym, B, S \rangle$. To explain the intuition behind the invariant, let us for now ignore the height difference between the words $a_1 \cdots a_{i-1}$ and $a_{i+1} \cdots a_n$; the information about the height difference will be captured by $StackSym$ precisely. The main idea is to ensure that (q, γ) along with the stack σ is the configuration reached by A^{pop} on reading the word $a_1 \dots a_{i-1}$, while the set B consists of all states q' such that the word $a_{i+1} \cdots a_n$ is accepted from the configuration (q', σ) by the VPA A . Under these circumstances, the query VPA can very easily decide whether position i is an answer to the query — if A on reading $(a_i, *)$ can go from state q to some state in B , then position i must be marked. In order to ensure that this invariant can be updated as position i changes, the query VPA keeps component S in control state m , and some additional information in the stack σ .

The formal invariant maintained by the query VPA when at position i is:

Case $a_i \in \Sigma$: d will be *right*, $StackSym$ (in control state m) will be (B_{k-1}, S_{k-1}) and $\sigma = \langle q_{k-1}, c_{k-1}, B_{k-2}, S_{k-2} \rangle \langle q_{k-2}, c_{k-2}, B_{k-3}, S_{k-3} \rangle \cdots \langle q_1, c_1, B_0, S_0 \rangle$, where

- $((q, \gamma), \langle q_{k-1}, c_{k-1} \rangle \cdots \langle q_1, c_1 \rangle)$ is configuration reached by A^{pop} on reading $a_1 \cdots a_{i-1}$
- $B = \{q' \mid A \text{ accepts } w_k \bar{c}_k \cdots w_0 \text{ from configuration } (q', \langle q, (a_i, *) \rangle \langle q_{k-1}, c_{k-1} \rangle \cdots \langle q_1, c_1 \rangle)\}$
- $S = \{(q_1, q_2) \mid (q_1, \epsilon) \xrightarrow{w_k} (q_2, \epsilon)\}$
- $B_i = \{q' \mid A \text{ accepts } w_i \bar{c}_i \cdots w_0 \text{ from configuration } (q', \langle q_i, c_i \rangle \cdots \langle q_1, c_1 \rangle)\}$, and
- $S_i = \{(q_1, q_2) \mid (q_1, \epsilon) \xrightarrow{w_i} (q_2, \epsilon)\}$

Case $a_i \in \bar{\Sigma}$: d will be *left*, $StackSym$ (in control state m) will be a symbol γ' , and $\sigma = \langle q_k, c_k, B_{k-1}, S_{k-1} \rangle \langle q_{k-1}, c_{k-1}, B_{k-2}, S_{k-2} \rangle \cdots \langle q_1, c_1, B_0, S_0 \rangle$, where

- $((q, \gamma), \gamma' \langle q_k, c_k \rangle \cdots \langle q_1, c_1 \rangle)$ is configuration reached by A^{pop} on reading $a_1 \cdots a_{i-1}$
- $B = \{q' \mid A \text{ accepts } w_k \bar{c}_k \cdots w_0 \text{ from configuration } (q', \langle q_k, c_k \rangle \cdots \langle q_1, c_1 \rangle)\}$

- $S = \{(q_1, q_2) \mid (q_1, \epsilon) \xrightarrow{w_k} (q_2, \epsilon)\}$
- $B_i = \{q' \mid A \text{ accepts } w_i \bar{c}_i \cdots w_0 \text{ from configuration } (q', \langle q_i, c_i \rangle \cdots \langle q_1, c_1 \rangle)\}$, and
- $S_i = \{(q_1, q_2) \mid (q_1, \epsilon) \xrightarrow{w_i} (q_2, \epsilon)\}$

Maintaining the Invariant. Initially the query VPA P will simulate the VPA A^{pop} on the word w from left to right. Doing this will allow it to obtain the invariant for position n . So what is left is to describe how the invariant for position $i - 1$ can be obtained from the invariant for position i . Recall that the configuration of P at position i is $\langle i, d, m, \sigma \rangle$ with $m = \langle (q, \gamma), StackSym, B, S \rangle$ and that the word $a_{i+1} \cdots a_n$ can be written as $w_k \bar{c}_k \cdots \bar{c}_1 w_0$. Let the word $a_i a_{i+1} \cdots a_n$ be uniquely written as $w'_\ell \bar{c}'_\ell \cdots w'_1 \bar{c}'_1 w'_0$ where w'_i 's are well matched words, and \bar{c}'_i 's are the unmatched close-tags. Depending on the symbol a_i , ℓ will either be $k - 1$ or $k + 1$. Let the configuration satisfying the invariant at position $i - 1$ be $\langle i - 1, d', m', \sigma' \rangle$, where $m' = \langle (q', \gamma'), StackSym', B', S' \rangle$. It is very easy to ensure that the direction d' is correct at position $i - 1$. The computation of the new (q', γ') will require P to traverse the word back and forth, and we will describe how this is obtained in the next paragraph. We will now describe the new values of all the remaining components in the configuration. There are 4 cases to consider.

Case $a_i, a_{i-1} \in \Sigma$: In this case $\ell = k - 1$ with $w'_\ell = a_i w_k \bar{c}_k w_{k-1}$, $c'_i = c_i$ and $w'_i = w_i$. Let $StackSym = (B_{k-1}, S_{k-1})$ and $StackSym' = (B'_{k-2}, S'_{k-2})$. We know that $|\sigma| = k - 1$ and $|\sigma'| = k - 2$ in this case. σ' will be obtained from σ by just popping the top of the stack symbol $\langle q_{k-1}, c_{k-1}, B_{k-2}, S_{k-2} \rangle$ of σ . B'_{k-2} and S'_{k-2} in $StackSym'$ are nothing but B_{k-2} and S_{k-2} that are popped from σ . Finally B' and S' are defined as follows:

$$\begin{aligned} S' &= \{(q_1, q_2) \mid q_1 \xrightarrow{a_i/\gamma_1} (q'_1, (q'_1, q'_2)) \in S, \\ &\quad q'_2 \xrightarrow{\bar{c}_k/\gamma_1} (q_3, q_2) \in S_{k-1}\} \\ B' &= \{q_1 \mid (q_1, q_2) \in S', q_2 \xrightarrow{\bar{c}_{k-1}/(q', (a_{i-1}, *))} (q_3, q_2) \in B_{k-2}\} \end{aligned}$$

Above, q' is figured out by back and forth traversal that is described later.

Case $a_i \in \Sigma, a_{i-1} \in \bar{\Sigma}$: In this case $\ell = k - 1$ with $w'_\ell = a_i w_k \bar{c}_k w_{k-1}$, $c'_i = c_i$ and $w'_i = w_i$. Let $StackSym = (B_{k-1}, S_{k-1})$. We know that $|\sigma| = |\sigma'| = k - 1$ in this case and σ' will be same as σ . $StackSym'$ will be symbol γ , the stack symbol in the state of A^{pop} at position i . Finally B' and S' are defined as follows:

$$\begin{aligned} S' &= \{(q_1, q_2) \mid q_1 \xrightarrow{a_i/\gamma_1} (q'_1, (q'_1, q'_2)) \in S, \\ &\quad q'_2 \xrightarrow{\bar{c}_k/\gamma_1} (q_3, q_2) \in S_{k-1}\} \\ B' &= \{q_1 \mid q_1 \xrightarrow{a_i/\gamma_1} (q'_1, (q'_1, q'_2)) \in S, q'_2 \xrightarrow{\bar{c}_k/\gamma_1} (q_3, q_2) \in B_{k-1}\} \end{aligned}$$

Case $a_i, a_{i-1} \in \bar{\Sigma}$: In this case $\ell = k + 1$ with $w'_\ell = \epsilon$, $c'_\ell = a_i$, and $c'_j = c_j$, $w'_j = w_j$ for $j \leq k$. Let $StackSym = \gamma_1$. The new $StackSym'$ will be γ , where (q, γ) is the state reached by A^{pop} on $a_1 \cdots a_{i-1}$. Next, $|\sigma| = k$ and $|\sigma'| = k + 1$; the new additional symbol to be pushed onto the stack (γ_1, B, S) . Finally, $S' = \{(q_1, q_1) \mid q_1 \in Q\}$ and

$$B' = \{q_1 \mid q_1 \xrightarrow{a_i/\gamma}_A q_2 \text{ and } q_2 \in B\}$$

Case $a_i \in \bar{\Sigma}, a_{i-1} \in \Sigma$: In this case $\ell = k + 1$ with $w'_\ell = \epsilon$, $c'_\ell = a_i$, and $c'_j = c_j$, $w'_j = w_j$ for $j \leq k$. Let $StackSym = \gamma_1$. In this case $|\sigma| = |\sigma'| = k$ and so $\sigma' = \sigma$. The new $StackSym'$ will be (B_k, S_k) , where $B_k = B$ and $S_k = S$. Finally, $S' = \{(q_1, q_1) \mid q_1 \in Q\}$ and

$$B' = \{q_1 \mid q_1 \xrightarrow{a_i/(q', (a_{i-1}, *))}_A q_2 \text{ and } q_2 \in B\}$$

where q' is the new state computed by traversing back and forth (described next).

B Translating query VPA to monadic queries

We now show the converse direction: For any query VPA A , there is an MSO_ν formula $\varphi(x)$ which defines the same query that A does.

Let A be a query VPA. We will translate the query A defines into an MSO_ν formula by translation through several intermediate stages.

Let f be the query defined by the query VPA A . We first construct a two-way (non-marking) VPA B that accepts the starred-language of f . B accepts a word w with a $*$ in position i if and only if $i \in f(w)$. Constructing B is easy. B simulates A on a word w with a $*$ in position i and accepts the word if A reaches position A in a marking state. B also ensures in a first run over the word that the word has a unique position that is marked with a $*$. The language accepted by B is $L_*(f)$, the starred-language of f .

Any nested word w can be represented as a tree called a *stack tree*. A stack tree is a $\hat{\Sigma}$ binary tree that has one node for every position in w , and the node corresponding to position i is labeled by $w[i]$. The stack tree corresponding to a word w is defined inductively as follows: (a) if $w = \epsilon$, then the stack tree of w is the empty tree, and (b) if $w = cw_1\bar{c}w_2$, then the stack tree corresponding to w has its root labeled c , has the stack-tree corresponding to w_1 rooted at its left child, the right child is labeled \bar{c} which has no left child, but has a right child which has the stack-tree corresponding to w_2 rooted at it.

We now show that the set of stack-trees corresponding the starred words accepted by B can be accepted using a *pushdown tree-walking automaton* [16]. A pushdown tree-walking automaton works on a tree by starting at the root

and walking up and down the tree, and has access to a stack onto which it always pushes a symbol when going down the tree and pops the stack when coming up an edge. Note that the height of the stack when at a node of the tree is hence always the depth of the node from the root. From B , we can build a tree-walking automaton C that reads the tree corresponding to a starred word, and simulates B on it. C can navigate the tree and effectively simulate moving left or right on the word. When B moves right reading a call symbol, C moves to the left child of the call and pushes the symbol B pushes onto its stack. When B moves right to read the corresponding return, C would go up from the left subtree to this call and pop the symbol from the stack and use it to simulate the move on the return. The backward moves on the word that B makes can also be simulated: for example, when B reads a return and moves left, C would go to the corresponding node on the left-subtree of the call node corresponding to the return, and when doing so push the appropriate symbol that B pushed onto the stack. When C moves down from an internal or return node, or from a call node to the right, it pushes in a dummy symbol onto the stack. In summary, whenever B is in a position i with stack $\gamma_1 \dots \gamma_k$, C would be reading the node corresponding to i in the tree, and the stack would have $\gamma_1 \dots \gamma_k$ when restricted to non-dummy symbols.

It is known that pushdown tree-walking automata precisely accept regular tree languages. Hence we can construct an MSO formula on trees that precisely is true on all trees that correspond to starred words accepted by B . This MSO formula can be translated to MSO_ν ψ on nested words, which is true on precisely the set of starred nested words that B accepts. Assuming x is not a variable in ψ , we replace every atomic formula of the form $Q_{(a,*)}(y)$ (the atomic formula checking whether position y is labeled a and is starred) by the formula $x = yQ_a(y)$, to get a formula $\varphi(x)$, with a free variable x . Intuitively, we replace every check the formula does for a starred label by a check as to whether that position is x . It is easy to see then that the formula $\varphi(x)$ is an MSO_ν formula on $\hat{\Sigma}$ -labeled (unstarred) nested words, which precisely defines the query defined by B , and hence the original query VPA A . This concludes the proof.

C Querying algorithm: Details

Construction of deterministic VPA in Lemma 1

The determinized automata is formally defined as follows.

1. $S = 2^{Q \times Q} \cup 2^{Q \times P} \cup 2^{P \times P} \cup 2^{P \times P \times \{*\}}$
2. $s_0 = \{(q_0, q_0)\}$

3. δ'^{open} is as follows: On reading $a \in \Sigma$ in state s_1 , we will push the pair (s_1, a) and go to state s_2 , where s_2 is the union of sets

- (a) $\{(r', r') \mid \exists(r_1, r_2) \in s_1, \gamma \in \Gamma \text{ s.t. } r_2 \xrightarrow{a/\gamma}_{A_f} r'\}$
- (b) $\{(p, p, *) \mid \exists(q_1, q_2) \in s_1, \gamma \in \Gamma \text{ s.t. } q_2 \xrightarrow{(a,*)/\gamma}_{A_f} p\}$.

4. δ'^{close} is as follows: On reading $\bar{a} \in \bar{\Sigma}$ in state s_1 with (s_2, a) on top of the stack, we go to state s_3 that is the union of the following sets.

- (a) $\{(q, q') \mid \exists(q, q_1) \in s_2, (q_2, q_3) \in s_1, \gamma \in \Gamma \text{ s.t. } q_1 \xrightarrow{a/\gamma}_{A_f} q_2 \text{ and } q_3 \xrightarrow{\bar{a}/\gamma}_{A_f} q'\}$.
- (b) $\{(p, p') \mid \exists(p, p_1) \in s_2, (p_2, p_3) \in s_1, \gamma \in \Gamma \text{ s.t. } p_1 \xrightarrow{a/\gamma}_{A_f} p_2 \text{ and } p_3 \xrightarrow{\bar{a}/\gamma}_{A_f} p'\}$.
- (c) $\{(p, p', *) \mid \exists(p, p_1, *) \in s_2, (p_2, p_3) \in s_1, \gamma \in \Gamma \text{ s.t. } p_1 \xrightarrow{a/\gamma}_{A_f} p_2 \text{ and } p_3 \xrightarrow{\bar{a}/\gamma}_{A_f} p'\}$.
- (d) $\{(q, p) \mid \exists(q, p_1) \in s_2, (p_2, p_3) \in s_1, \gamma \in \Gamma \text{ s.t. } p_1 \xrightarrow{a/\gamma}_{A_f} p_2 \text{ and } p_3 \xrightarrow{\bar{a}/\gamma}_{A_f} p\}$.
- (e) $\{(q, p) \mid \exists(q, q_1) \in s_2, (q_2, p_3) \in s_1, \gamma \in \Gamma \text{ s.t. } q_1 \xrightarrow{a/\gamma}_{A_f} q_2 \text{ and } p_3 \xrightarrow{\bar{a}/\gamma}_{A_f} p\}$.
- (f) $\{(q, p) \mid \exists(q, q_1) \in s_2, (p_2, p_3, *) \in s_1, \gamma \in \Gamma \text{ s.t. } q_1 \xrightarrow{(a,*)/\gamma}_{A_f} p_2 \text{ and } p_3 \xrightarrow{\bar{a}/\gamma}_{A_f} p\}$.
- (g) $\{(q, p) \mid \exists(q, q_1) \in s_2, (q_2, q_3) \in s_1, \gamma \in \Gamma \text{ s.t. } q_1 \xrightarrow{a/\gamma}_{A_f} q_2 \text{ and } q_3 \xrightarrow{(\bar{a},*)/\gamma}_{A_f} p\}$.

5. $F' = \{s \mid \exists(q_0, p) \in s \text{ s.t. } p \in F\}$.

Proof of Lemma2

Lemma2. *For any VPA A , there is VPA $ValidPref(A)$ such that $L(A) = L(ValidPref(A))$ and for any prefix u of a well-matched word, $ValidPref(A)$ has some execution on u if and only if there exists v such that $wv \in L(A)$. If m is the size of A then $ValidPref(A)$ has size at most m^2 .*

Proof. It is well-known that for any PDA P , there is a NFA $Config(P)$ such that $Config(P)$ accepts a string $\gamma_0\gamma_1 \cdots \gamma_kq$ iff P cannot reach an accepting state from the configuration $(q, \gamma_k\gamma_{k-1} \cdots \gamma_0)$ (γ_0 is the bottom of the stack). In addition, $Config(P)$ has the same number of states as P , and can be constructed in cubic time.

The idea behind $ValidPref(A)$ can now be explained easily. $ValidPref(A)$ will simultaneously simulate both A and $Config(A)$ as it is reading the document, ensuring that at any point, the state of $Config(A)$ is the state reached on reading the current stack contents of A . This

can be achieved as follows. When A pushes a symbol, $ValidPref(A)$ will simulate $Config(A)$ on the symbol that A pushes, and at the same time push onto its stack both the symbol that A pushes and the old state of $Config(A)$. When A pops, $ValidPref(A)$ will pop the state of $Config(A)$ on the stack and continue the simulation of $Config(A)$ from the popped state. If at any point the simulation of A reaches a control state q such that $Config(A)$ accepts q from the current state, then $ValidPref(A)$ will terminate that run. The formal definition of $ValidPref(A)$ based on these ideas is skipped in the interests of space. \square

D Comparison with query automata on unranked trees

Neven and Schwentick [22], define an automaton model on trees, called *generalized deterministic two-way tree automata* (G2DTA), as follows. Consider a partition of $Q \times \Sigma$ into two sets U and D , where Q is the set of (finitely many) states of the automaton, and Σ is the alphabet labeling the input tree. Further let U_{up} and U_{stay} be a partition of the set U^* . G2DTA are defined to be tree automata with three types of transitions.

Down transitions $\delta_{\downarrow} : D \times \mathbb{N} \rightarrow Q^*$ such that for each i , $\delta_{\downarrow}(q, a, i)$ (for $(q, a) \in D$) is a string of length i . Thus, when the automaton is in state q at a vertex v labeled a with i children, in the next step the automaton could go down in the tree, and the states at v 's children is given by the string $\delta_{\downarrow}(q, a, i)$. It is assumed that for each $(q, a) \in D$ the language $\{\delta_{\downarrow}(q, a, i) \mid i \in \mathbb{N}\}$ is regular.

Up Transitions $\delta_{\uparrow} : U_{up} \rightarrow Q$ such that for each $q \in Q$, $\{w \in U_{up} \mid \delta_{\uparrow}(w) = q\}$ is regular. When the automaton is at each of the children of a vertex v such that the string formed by state-symbol pairs at the children is $w \in U_{up}$, the automaton moves up to the vertex v in state $\delta_{\uparrow}(w)$.

Stay Transitions $\delta_{\downarrow} : U_{stay} \rightarrow Q^*$ is a length preserving function that is computed by a *two-way finite state transducer*. When the automaton is at each of the children of a vertex v such that the string formed by state-symbol pairs at the children is $w \in U_{stay}$, the automaton simultaneously changes state at each of the children of v to that given by the string $\delta_{\downarrow}(w)$.

In addition, G2DTA is equipped with a selection function that allows it to *select* certain vertices of an unranked tree as answers to a query. We skip the formal definition of these automata, and the reader is referred to [22].

Through a couple of facts, Neven and Schwentick highlight the subtle role stay transitions play. First they show

that if we consider two-way tree automata models without stay transitions then these are not as expressive as MSO queries on unranked trees. Next they show that G2DTA as defined above are much more expressive than MSO. In fact, G2DTA are equivalent in expressive power to linear space bounded Turing Machines.

Thus, in order to achieve MSO expressiveness, Neven and Schwentick define a restricted automaton model called *strong two-way deterministic tree automata* (S2DTA), which are G2DTA with the restriction that on any input tree, at most one stay transition is executed at the children of each node. This is a non-trivial *semantic* restriction and can be shown to EXPTIME-complete to check.

Our proof of that query VPA are exactly equi-expressive to unary MSO query, though it uses similar techniques, does not seem directly obtainable from Neven and Schwentick’s result. We will now outline the challenges in constructing a direct translation between query VPA and S2DTAs. However, before doing so, we first define how an unranked tree can be associated with any well-matched word. Define a *simple well-matched word* to be any word of the form $cw\bar{c}$, where $c \in \Sigma$ and w is any well-matched word. Given a simple well-matched word $w = cw_1w_2 \cdots w_k\bar{c}$, where each w_i is again a simple nested word, $\text{tree}(w)$ is a tree whose root is labeled by (c, \bar{c}) and has k children, where the i -th child is $\text{tree}(w_i)$, defined inductively.

Query VPA to S2DTA

Let us first consider the challenges in translating query VPA to S2DTA. Consider the execution on an input word $\triangleright w \triangleleft$, and let us for simplicity assume that $\text{tree}(w)$ is such that every node (other than the leaves) have at least two children. Suppose the query VPA \mathcal{A} is in configuration $\langle p, d, q, s \rangle$, and let v be the vertex in $\text{tree}(w)$ corresponding to position p . One natural way for the S2DTA \mathcal{T} to encode the state q and stack $s = \gamma_1 \cdots \gamma_k$ is as follows. The state of \mathcal{T} at v would be q , and stack itself would be stored as the “state” of \mathcal{T} at a particular sibling of each of the vertices along the unique path from the root to v ; note that our tree encoding will ensure that v is at depth k , and so this is well defined.

Now let us consider maintaining this invariant as the query VPA takes various transitions. If \mathcal{A} moves right while reading an open-tag, then that corresponds to making a down transition in the S2DTA \mathcal{T} . However, consider moving right while reading a close-tag. Now if the close-tag \bar{c} corresponds to a vertex that is the rightmost child of its parent, then such a right move corresponds to moving up in the tree. On the other hand, if the vertex corresponding to \bar{c} , say v , is not the rightmost child, then in the S2DTA we are required to “move” to v ’s right sibling. There is no way to accomplish this by an up and then a down transition (which

is why Neven and Schwentick needed stay transitions). The only way to achieve this is using a stay transition. However then the translation from \mathcal{A} to \mathcal{T} will result in a G2DTA that is not an S2DTA, as there is no a priori way to bound the number of right moves over non-rightmost close-tags!

S2DTA to Query VPA

Let us now consider the other direction of the translation. First observe that the S2DTA being a tree automaton, allows an “inherent” non-determinism — at a configuration, the automaton can either choose to move down from a vertex or execute an up/stay transition at a set of vertices, if such a transition is enabled. On the other hand, a query VPA has a very deterministic left-right and right-left way to process a word. Thus, for a translation from S2DTA to query VPA to work, we will require to prove that we can always restrict our attention to runs of the S2DTA, where the *left-most enabled transition* is always taken first; in a configuration c of the S2DTA, a left-most enabled transition is either a down transition at a vertex v , such that all vertices to the left of v in the configuration are waiting for an up/stay transition, or is an up/stay transition at v such that all of v ’s siblings are also ready to execute the up/stay transition, and no vertex to the left of v in the configuration has an enabled transition.

Let us assume for now that such a proposition, allowing us to only consider such restricted runs of the S2DTA, can be proved. Let us consider the various transitions of the S2DTA and see how they can be simulated on the query VPA. Suppose the left-most enabled transition is a down transition $\delta_{\downarrow}(q, a, i)$ at v . The VPA cannot simultaneously go to all the children of v , but rather only “moves” to the leftmost subtree of v . Now if the observation about left-most enabled transitions hold, then in the query VPA, we can process the left-most subtree until we need to execute an up/stay transition, and then move to the next sibling and so on. The first challenge to overcome is how and where can we store the string $\delta_{\downarrow}(q, a, i)$ so that after the left-most child is processed, we can proceed with the original down transition at the next sibling. The key idea is this. Note that the string $\delta_{\downarrow}(q, a, i)$ is computed by a *one-way finite automata* (on words) (say $\mathcal{A}_{\delta_{\downarrow}}$). Therefore, we can push the state of $\mathcal{A}_{\delta_{\downarrow}}$ after it produces the first symbol (which is the state at the left-most child of v) onto the stack, process the left-most subtree, and use the pushed state later on to generate the next symbol of $\delta_{\downarrow}(q, a, i)$ and so on. We can do a similar trick for the up transitions. Once again observe that since $\{w \in U_{up} \mid \delta_{\uparrow}(w) = q\}$ is regular, we can assume we have *one-way* automata \mathcal{A}_q for all these languages. So when we need to execute an up transition at v , after processing the subtree rooted at v , we first push the state of all these \mathcal{A}_q automata before processing the subtree rooted at

the next sibling of v . We continue in this fashion until all the siblings of v are ready to execute the up transition. At this point the states of the automata \mathcal{A}_q that we have pushed can be used to determine the state q at the parent of v after the up transition is executed, which corresponds to moving right in the word.

Assuming the ideas outlined in the previous two paragraphs can be carried out technically, we run into the main road-block that does not seem to have any solution, namely, stay transitions. A stay transition $\delta_- : U_{stay} \rightarrow Q^*$ is computed by a two-way transducer, which means that we need to make multiple passes over that state-symbol pairs at each sibling in order to determine the new states at each node. This requires storing the state-symbol string in a tape with two-way access, and there is no such storage available to the query VPA. Our trick of using the stack to dynamically compute as in the case of down and up transitions no longer works here.