Query Automata for Nested Words

P. Madhusudan and Mahesh Viswanathan

University of Illinois, Urbana-Champaign
{madhu,vmahesh}@cs.uiuc.edu

Abstract. We study visibly pushdown automata (VPA) models for expressing and evaluating queries on words with a nesting structure. We define a query VPA model, which is a 2-way deterministic VPA that can mark in one run *all* positions in a document that satisfy a query, and show that it is equi-expressive as unary monadic queries. This surprising result parallels a classic result by Hopcroft and Ullman for queries on regular word languages. We also compare our model to query models on unranked trees, and show that our result is fundamentally different from those known for automata on trees.

1 Introduction

A nested word is a word endowed with a nesting structure that captures hierarchically structured segments of the word. Applications of nested words abound in computer science— terms and expressions are naturally nested (the bracketing capturing the nesting), XML/HTML/SGML documents are nested words capturing hierarchically structured data elements (the open and close tags capture the nesting), and even runs of recursive sequential programs can be seen as nested words capturing the nested calling structure (the call to and return from procedures capture the nesting).

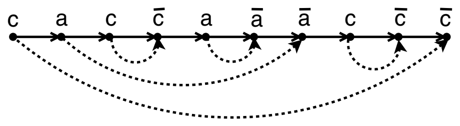

Fig. 1. A nested word

Trees have been the traditional approach to model nested structures. The rich results in the automata theory of trees is a robust theory that captures tractable representations of nested structures. Nested words are an alternative way to describe nested structures, where the linear arrangement of data is emphasized (as is common in a document representing this data, like an XML document). Automata on nested words process a document (word) along this linear order, but also exploit the nesting edges.

The systematic study of nested word structures and finite automata working on them was first done using *visibly pushdown automata* (where the automaton processes the word left to right, and uses a stack to relay information flow along the nesting edges) [1]. An alternative and mathematically equivalent model is that of nested word automata [2], which are *finite state automata* (no stack) and process a nested word linearly, but where the automata are additionally allowed to refer to nested-edge-predecessors in order to update the state.

Visibly pushdown automata and nested word automata were first introduced in the context of formal verification (since runs of recursive programs are nested words). Since its introduction in 2004, this model has become quite popular, and a rich theory of visibly pushdown languages has been developed, ranging from applications to model-checking, monitoring, temporal logics, programming languages, security, XML, and complexity theory[1].

In this paper, we study the power of visibly pushdown automata in expressing and answering *queries* on nested words. We introduce an automaton model for defining queries, called query visibly pushdown automata (query VPA). A query VPA is a *deterministic* two-way (can move left and right) visibly pushdown automaton that can mark positions of a word. 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 word is precisely the number of unmatched calls before that position (which is the number of unmatched returns after that position).

A query VPA runs (deterministically) on a word and *marks* a set of positions; these positions are to be seen as the set of all answers to a query. Note that a query VPA, when given a word, gives all answers to the query in a *single* run.

Our main result is that query VPAs have the right expressive power for defining queries: we show 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. Both directions are non-trivial, and the direction of implementing any MSO query using query VPAs relies on a beautiful observation by Hopcroft and Ullman [4].

We find it remarkable that the simple definition of two-way VPAs exactly captures all monadic queries. This result actually parallels a classic result in the theory of automata on finite *non-nested* words, which equates the power of unary monadic queries on (non-nested) words to that of two-way automata on words [4,10].

A query VPA, when viewed as an algorithm working on a word, traverses the word back and forth, and outputs *all* positions that answer the query, using only space $O(d)$, where d is the depth of the word (the depth of nesting in the word). Our result hence implies that any unary MSO query can be answered on a word of length n using only $O(d)$ space (with *no* dependence on n) and in time $O(n^2)$. As far as we know (and also based on discussions [11]), there is no such parallel result using the theory of tree automata (it is however well-known that given a

[1] See the VPL/Nested word languages page at `http://www.cs.uiuc.edu/~madhu/vpa` for a comprehensive list of papers and references.

particular position, checking whether it is an answer to a query can be done in $O(d)$-space; see also the related work below).

Related Work. Theoretical query models for XML have been studied using tree models. While most tree automata models work by passing states along the edges of the tree, there are tree automaton models that work in the linear order corresponding to its word representation (see [3] and the more recent [6]).

The most closely related work to our result on query VPAs is that of the query automaton model on unranked trees [10]. Query automata (more precisely, S2DTA^u) on unranked trees select positions in a tree and is exactly as powerful as unary monadic queries on trees [10] (which is the same as that on nested words). This model works like a two-way tree automaton on unranked trees, has parallel copies of automata that go up and down the tree, with automata processing children of a node synchronizing when going up to their parent. However, they are complex due to a special class of transitions (called *stay transitions*) that rewrite, using regular relations, the states labeling the children of a node. Further, there is a *semantic* restriction that requires that that the children of any node be processed by a stay transition at most once.

We believe that query VPA are significantly simpler and a different model than S2DTA^u, and it is not easy to convert between the two models (in either direction). Note that we also do not have any semantic restriction of the kind imposed on stay transitions in our setting, which is an important difference between the two models.

In [5], *selecting tree-automata* are defined, which are simpler than query automata and can return positions that satisfy any MSO query. However, these automata are *nondeterministic* in nature, and thus fundamentally different from our model.

Another line of work that is related is the work of Neumann and Seidl: in [9] (see also [8]), it was shown that a single-pass from left-to-right is sufficient to answer all queries that pick an element by referring only to properties of the document that occur before the element; these queries do not handle future predicates and hence can work in one pass using only $O(d)$ space.

Yet another work that is relevant is that reported in Neumann's thesis ([8], Chapter 7), where it is shown that for any query, there is a pushdown automaton that does two passes on a document, the first pass left-to-right, and the second pass right-to-left, such that any node being an answer to a query is determined solely by the states of the automaton on the two passes. Note that this is quite different from ours; an algorithm implementing this scheme would have to store states of the automaton at *all positions* in the first pass, and hence will require $O(n)$-space to output all answers to a query.

2 Preliminaries

For simplicity of exposition, we will assume that every letter in the word is the source or target of a nested edge; extending our results to the general class of nested words is straightforward. Nested words will be modeled as words over an alphabet where a single letter of the alphabet encodes an open/close tag.

Let Σ be a fixed finite alphabet of "open tags", and let $\overline{\Sigma} = \{\overline{c} \mid c \in \Sigma\}$ be the corresponding alphabet of "close tags". Let $\widehat{\Sigma} = \Sigma \cup \overline{\Sigma}$. A *well-matched* word is any word generated by the grammar: $W \rightarrow cW\overline{c}$, $W \rightarrow WW$, $W \rightarrow \epsilon$, where we have a rule $W \rightarrow cW\overline{c}$ for every $c \in \Sigma$. The set of all well-matched words over $\widehat{\Sigma}$ will be denoted by $WM(\widehat{\Sigma})$.

Nested Words, Monadic Second-Order Logic

A well-matched word $w \in WM(\widehat{\Sigma})$ can be seen as a nested structure: a linear labeled structure with nesting edges. For example, the structure corresponding to the word $cac\overline{c}a\overline{a}\overline{a}c\overline{c}\overline{c}$ is shown in Figure 1. The linear skeleton (denoted by solid edges) encodes the word and the nesting edges (denoted by dotted edges) relate open-tags with their matching close-tags. We skip the formal definition, but denote the nested structure associated with a word w as $nw(w) = (\{1, \ldots, |w|\}, \{Q_a\}_{a \in \widehat{\Sigma}}, \leq, \nu)$, 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$_\nu$) 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:

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

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 $\delta = \langle \delta^{open}, \delta^{close} \rangle$ is the transition relation, where:*

– $\delta^{open} \subseteq ((Q \times \Sigma) \times (Q \times \Gamma))$;

– $\delta^{close} \subseteq ((Q \times \overline{\Sigma} \times \Gamma) \times Q)$.

A transition $(q, c, q', \gamma) \in \delta^{open}$ (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 $(q, \overline{c}, \gamma, q')$ (denoted $q \xrightarrow{\overline{c}/\gamma} q'$) is a pop-transition, allowing the automaton, when in state q reading $\overline{c}$ 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 $a \in \widehat{\Sigma}$, 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 $(q_1, c, q_2, \gamma) \in \delta^{open}$, or
– $a = \overline{c} \in \overline{\Sigma}$, $s_1 = \gamma.s_2$ and $(q_1, \overline{c}, \gamma, q_2) \in \delta^{close}$.

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 [1], 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$_\nu$ 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$_\nu$* 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 nested words in this paper.

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 over $\widehat{\Sigma}$ where precisely one letter has been annotated with a $*$.

A query f then corresponds to a set of starred-words:
$L_*(f) = \{a_1 a_2 \ldots a_{i-1}(a_i, *)a_{i+1} \ldots a_n \mid i \in f(a_1 \ldots a_{i-1} a_i a_{i+1} \ldots a_n)$ *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 [1], 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.

The main result of this paper is as follows. We define the automaton model of *query VPA* over nested words, 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. Notice that this result is very different from Theorem 1; the query VPA is a machine that marks *all* positions that are answers to a query, as opposed to a VPA that can check if a *single* marked position is an answer to a query.

3 Query VPA

The goal of this section is to define an automaton model for nested words 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:*

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

- $\delta^{close}_{right} : (S \cap (Q \times \overline{\Sigma})) \times \Gamma) \to Q$
- $\delta^{close}_{left} : S \cap (Q \times \overline{\Sigma}) \to (Q \times \Gamma)$
- $\delta^{chng} : C \cup (Q \times \{\rhd, \lhd\}) \to Q$

The δ^{open}_{right} and δ^{close}_{left} 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^{open}_{left} and δ^{close}_{right} 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 $\rhd$ or $\lhd$. 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 $x = \rhd w \lhd$. 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, $d \in \{\mathit{left}, \mathit{right}\}$ 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 $\langle 1, \mathit{right}, q_0, \epsilon \rangle$, i.e., initially the automaton is reading the leftmost symbol of w (not $\rhd$), it is moving right, in the initial state, with an empty stack. A *run* is a sequence $c_0, c_1, \ldots c_n$, where c_0 is the initial configuration, 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 $d_i = \mathit{right}$ then $d_{i+1} = d_i$, $p_{i+1} = p_i + 1$, $q_{i+1} = q$ and $s_{i+1} = \gamma s_i$, where $\delta^{open}_{right}(q_i, x[p_i]) = (q, \gamma)$
- If $(q_i, x[p_i]) \in S \cap (Q \times \overline{\Sigma})$, and $d_i = \mathit{right}$ then $d_{i+1} = d_i$, $p_{i+1} = p_i + 1$, $q_{i+1} = q$ and $s_i = \gamma s_{i+1}$, where $\delta^{close}_{right}(q_i, x[p_i], \gamma) = q$
- If $(q_i, x[p_i]) \in S \cap (Q \times \Sigma)$, and $d_i = \mathit{left}$ then $d_{i+1} = d_i$, $p_{i+1} = p_i - 1$, $q_{i+1} = q$ and $s_i = \gamma s_{i+1}$, where $\delta^{open}_{left}(q_i, x[p_i], \gamma) = q$
- If $(q_i, x[p_i]) \in S \cap (Q \times \overline{\Sigma})$, and $d_i = \mathit{left}$ then $d_{i+1} = d_i$, $p_{i+1} = p_i - 1$, $q_{i+1} = q$ and $s_{i+1} = \gamma s_i$, where $\delta^{close}_{left}(q_i, x[p_i]) = (q, \gamma)$
- 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 $d_i = \mathit{right}$ then $d_{i+1} = \mathit{left}$ and $p_{i+1} = p_i - 1$. On the other hand, if $d_i = \mathit{left}$ then $d_{i+1} = \mathit{right}$ 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 $p \in \{1, \ldots |w|\}$, if $d = \mathit{right}$ 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 $d = \mathit{left}$ 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 $1 \le j \le |w|$, if the unique run $c_0 \dots c_n$ of P on the input $\triangleright w \triangleleft$ is such that for some i, $c_i = (j, d, q, s)$ 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 state the main result of this paper: 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_νif and only if there is query VPA P such that $f_P = f$.*

The proof of Theorem 2 follows from Lemmas 1 and 2 that are proved in the next two sections.

3.1 Implementing Monadic Queries on Query VPA

In this section we prove one direction of Theorem 2, namely, that every monadic query can be implemented on a query VPA.

Lemma 1. *For any monadic query f, there is a query VPA P such that $f = f_P$.*

Proof. 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 word 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 $|w|$ of $\triangleright w \triangleleft$ 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 [10] 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 [1]. Next, given any VPA $B = (Q, q_0, \Gamma, \delta, F)$, there is a VPA B^{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: $B^{pop} = (Q \times (\Gamma \cup \{\bot\}), (q_0, \bot), \Gamma, \delta', F \times (\Gamma \cup \{\bot\}))$, where the new transitions are as follows. If $q \xrightarrow{c/\gamma_1}_B q'$ then $(q, \gamma) \xrightarrow{c/\gamma_1}_{B^{pop}} (q', \bot)$. If $q \xrightarrow{\overline{c}/\gamma_1}_B q'$ then $(q, \gamma) \xrightarrow{\overline{c}/\gamma_1} (q', \gamma_1)$.

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^{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 $w = u a_i v$ 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 $w_k \overline{c_k} w_{k-1} \overline{c_{k-1}} \cdots w_1 \overline{c_1} w_0$, where for each j, w_j is a well-matched word, and $\overline{c_k}, \dots \overline{c_1}$ 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^{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.

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 $a_i \in \overline{\Sigma}$ 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 two different threads of the backward simulation are still alive. Position $i - 1$ is the unique position where these two 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 $a_{i-1} \in \overline{\Sigma}$ 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 $a_1 \ldots a_{i-2}$, and then simulate forward to determine the state (q', γ') as described above.

$a_{i-1} \in \overline{\Sigma}$: 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.

3.2 Translating Query VPA to Monadic Queries

We now complete the proof of Theorem 2, by showing that any query implemented on a query VPA can be described as a unary monadic query.

Lemma 2. *For any query VPA A, there is an MSO$_\nu$ formula $\varphi(x)$ such that $f_\varphi = f_A$.*

Proof. Let A be a query VPA. The query defined by A will be translated into an MSO$_\nu$ formula 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 i 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 $w = cw_1\overline{c}w_2$, 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 $\overline{c}$ 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* [7]. 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$_\nu$ 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 $x = y \wedge Q_a(y)$, 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 $\widehat{\Sigma}$-labeled (unstarred) nested words, which precisely defines the query defined by B, and hence the original query VPA A. This concludes the proof.

4 Conclusions

The query automaton model we have defined on nested words is an elegant model that answers queries using the least space possible. While our result is theoretical in nature, it may have implications on applications: our model shows that unary queries on XML (like logical XPath queries) can be answered using only $O(d)$-space; we also believe that our model could have applications in verification where, given a run of a sequential program, we can build efficient algorithms that answer queries such as "which positions x of the run satisfy a temporal formula $\varphi(x)$?", with applications to debugging error traces.

Finally, our query automaton model can be adapted to an analogous MSO-complete unary query automata on unranked trees as well: we can define a 2-way pushdown automaton tree-walking automaton that processes it by traversing it according to the linear order (determined by its serialization as a word), pushing onto the stack when going down a tree and popping the stack when coming up; this will essentially be an encoding of the query VPA on the tree.

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC, pp. 202–211. ACM Press, New York (2004)
2. Alur, R., Madhusudan, P.: Adding nesting structure to words. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 1–13. Springer, Heidelberg (2006)
3. Carme, J., Niehren, J., Tommasi, M.: Querying unranked trees with stepwise tree automata. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 105–118. Springer, Heidelberg (2004)
4. Engelfriet, J., Hoogeboom, H.J.: Mso definable string transductions and two-way finite-state transducers. ACM Trans. Comput. Logic 2(2), 216–254 (2001)
5. Frick, M., Grohe, M., Koch, C.: Query evaluation on compressed trees (extended abstract). In: LICS, p. 188. IEEE Computer Society, Los Alamitos (2003)
6. Gauwin, O., Niehren, J., Roos, Y.: Streaming tree automata. Inf. Process. Lett. 109(1), 13–17 (2008)
7. Kamimura, T., Slutzki, G.: Parallel and two-way automata on directed ordered acyclic graphs. Information and Control 49(1), 10–51 (1981)
8. Neumann, A.: Parsing and querying xml documents in sml. Universität Trier, Trier, Germany (1999)

9. Neumann, A., Seidl, H.: Locating matches of tree patterns in forests. In: Arvind, V., Sarukkai, S. (eds.) FST TCS 1998. LNCS, vol. 1530, pp. 134–146. Springer, Heidelberg (1998)
10. Neven, F., Schwentick, T.: Query Automata over Finite Trees. Theoretical Computer Science 275(1-2), 633–674 (2002)
11. Schwentick, T.: Personal communication (2008)