Projection for Nested Word Automata Speeds up XPath Evaluation on XML Streams

Tom Sebastian, Joachim Niehren

Projection for Nested Word Automata Speeds up XPath Evaluation on XML Streams

Tom Sebastian[1] and Joachim Niehren[2]

[1] Innovimax & Links team of Inria Lille & Cristal lab
[2] Inria & Links team of Inria Lille & Cristal lab

Abstract. We present an evaluator for navigational XPath on Xml streams with projection. The idea is to project away those parts of an Xml stream that are irrelevant for evaluating a given XPath query. This task is relevant for processing Xml streams in general since all Xml standard languages are based on XPath. The best existing streaming algorithm for navigational XPath queries runs nested word automata. Therefore, we develop a projection algorithm for nested word automata, for the first time to the best of our knowledge. It turns out that projection can speed up the evaluation of navigational XPath queries on Xml streams by a factor of 4 in average on the usual XPath benchmarks.

1 Introduction

Projection is most relevant for efficient Xml processing algorithms, as shown for in-memory evaluators for XQuery in [11] and for a fragment of in XPath [10]. The projection algorithm for XQuery runs in Saxon [7], today's most used Xml processing tool. Projection algorithms for the in-memory evaluation of Xslt are missing though.

The objective of the present paper is to initiate the development of projection algorithms for processing Xml streams. Given that a single program written in one of the Xml standards XQuery, Xslt, or XProc contains a collection of XPath queries, we are interested in the evaluation of a collection of XPath queries on a single input stream. The parsing time can be shared between many XPath queries, and thus be should counted seperately. Therefore, we are mainly interested in the parsing-free time for query evaluation. Note however, that the parsing-free time for a single query is often dominated by the parsing time.

We will restrict ourselves to projection for navigational XPath queries, since these are fundamental to all others. For instance, in order to check whether the root of a tree has at least 5 a-children, all other children of the root can be projected. The computation of the projection still requires to read the entire input tree, but the time for this can be shared similarly to the parsing time.

The most efficient evaluation algorithm for navigational XPath queries on Xml streams so far was presented in [2]. Similarly to many recent evaluation algorithms for XPath on Xml streams [9,12,6], it is based on the compilation of navigational XPath queries to nested word automata (Nwas) [1]. Given that

XML streams are nested words, NWAs provide a canonical formalism for defining algorithms on XML streams. This leads to highly efficient algorithms based on first principles as argued in [2]: In particular, one can rely on the nondeterminism of NWAs to express XPATH queries with recursive axes, such as "descendant" or "following", and then use on-the-fly determinization for their evaluation. The evaluation of an XPATH query can then be reduced to running an NWA on all possible answer candidates. Furthermore, the runs of multiple answer candidates in the same state can be shared.

Projection for finite automata is well known [5,10]. It amounts to project away all letters of the input word that do not change the state. Projection for NWAs is more tedious, since such automata have a stack by which they can pass information from opening tags to corresponding closing tags. Therefore, one cannot simply project an opening tag away without taking care of the corresponding closing tag. Our idea is that a projected nested word should contain jump symbols $.^i.$ for projected factors, where the integer i stands for the excess of the factor, i.e., the difference between the number of opening and closing tags. We present *projection nested word automata (*PNWAS*)*, a kind of mixed pushdown and counting automata, that input projected nested words which beside others contain integers as letters. These integers allow the automaton to compute the depth of the current node of the tree at any time, and also the excess of the last jump. Conversely, a projection of a nested word with respect to a given NWA can be computed by any corresponding PNWA. It may be surprising, but it turns out there may exist different PNWAS with maximal projection for the same NWA. Therefore, our projection algorithm has to make its choices.

We then lift NWA projection to the evaluation of navigational XPATH queries on XML streams. It turns out that the parsing-free time for query answering is reduced by a factor of 4 on average on the usual XPathMark benchmark compared to the previously existing algorithm [2].

Outline. In Section 2, we recall NWAs and their usage for XPATH evaluation on XML streams. In Section 3, we introduce PNWAS. In Section 4, we introduce notions of irrelevant labels and prefixes of nested words for states of NWAs. In Section 5, we use them to project NWAs to PNWAS. In Section 6, we present our experimental results for XPATH evaluation on XML streams. The appendix of the present paper at `hal.inria.fr/hal-01182529` contains additional examples of PNWAS, an extension of NWA projection for node selection, and the collection of queries used in our experiments.

2 Nested Word Automata

We recall the definition of NWAs, while pointing out the close relationship between nested words and XML streams.

Let Σ be a finite alphabet. Let P_Σ be the set of parenthesis with labels in Σ, that is the set of opening tags $\langle a\rangle$ and closing tags $\langle /a\rangle$ where $a \in \Sigma$. A nested word over Σ is a word over P_Σ which is well-balanced, so that any opening tag $\langle a\rangle$ can be assigned to a unique corresponding closing tag $\langle /a\rangle$, and vice versa,

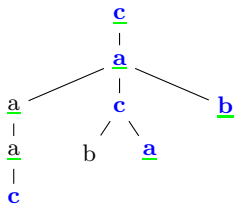

Fig. 1: An unranked tree.

$\langle \mathbf{c} \rangle$ $\langle \mathbf{a} \rangle$ $\langle a \rangle$ $\langle a \rangle$ $\langle \mathbf{c} \rangle$ $\langle /\mathbf{c} \rangle$ $\langle /a \rangle$ $\langle /a \rangle$
$\langle \mathbf{c} \rangle$ $\langle b \rangle$ $\langle /b \rangle$
$\langle \mathbf{a} \rangle$ $\langle /\mathbf{a} \rangle$ $\langle /\mathbf{c} \rangle$
$\langle \mathbf{b} \rangle$ $\langle /b \rangle$ $\langle /a \rangle$ $\langle /c \rangle$

Fig. 2: The corresponding nested word is an XML stream.

and such that the initial opening tag of the word corresponds to the closing tag at its end. Our nested words are more restricted than in the general case [1], in that internal symbols are omitted, corresponding opening and closing tags must have the same label, and initial opening tags cannot be closed before the end.

An XML stream is a nested word that is obtained by linearizing an unranked tree in document order. This is a strong simplification of the XML data model, in that we ignore data values (internal symbols) and the different types of nodes (text, element, attribute, etc). An example of an unranked tree is given in Fig. 1. The XML stream obtained by linearizing this unranked tree into a nested word is given in Fig. 2. It should be mentioned that we cannot assume any a priori knowledge on the set of tags of an XML document in practice (where no XML schemas are available). Instead, the finite alphabet Σ is determined by the tags appearing in the XPATH query of interest [2].

An NWA is a pushdown automaton on nested words [1], whose stack is "visible" in the sense that only a single symbol is pushed at opening events and popped at closing events. Here we assume that NWAs are early [2], so that whenever a final state is reached, any continuation completing the nested word will be accepted. More formally, an (early) NWA is a tuple $A = (\Sigma, Q, I, F, \Gamma, R)$ where Σ is a finite alphabet, Q a finite set of states with subsets $I, F \subseteq Q$ of initial and final states, Γ a finite set of stack symbols, and R is a set of transition rules of the following two types, where $q, q' \in Q, a \in \Sigma$, and $\gamma \in \Gamma$:

Open: $q \xrightarrow{\langle a \rangle \downarrow \gamma} q'$**.** When processing an opening tag $\langle a \rangle$, γ is pushed onto the stack, and the state is changed from q to q'.

Close: $q \xrightarrow{\langle /a \rangle \uparrow \gamma} q'$**.** When processing a closing tag $\langle /a \rangle$, γ is popped from the stack and the state is changed from q to q'.

A configuration of an NWA is a word in $Q\Gamma^*$ consisting of a state $q \in Q$ and a stack $S \in \Gamma^*$. A run of an NWA on a nested word $w \in P_\Sigma^*$ is a function r that maps prefixes w' of w to configurations. The initial configuration must contain an initial state and the empty stack, i.e. $r(\epsilon) \in I$. The NWA then rewrites this configuration: for any prefix $w'p$ of w, $r(w'p)$ is produced from $r(w')$ by applying some rules consuming tag $p \in P_\Sigma$. A run on w is successful if $r(w) \in F$, i.e. if it reaches at the end a final state and the empty stack. Since we assume early NWAs, any run reaching a configuration with a final state on some prefix of a

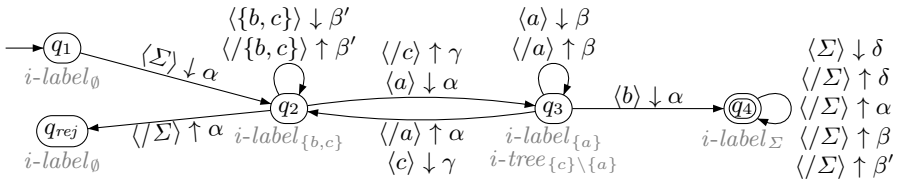

Fig. 3: A deterministic NWA over $\Sigma = \{a, b, c\}$ for XPATH filter `[//a/b]`.

nested word can always be continued into a successful run. The *language* $\mathcal{L}(A)$ of an NWA A is the set of all nested words that permit a successful run by A.

An NWA is called *deterministic* if it is deterministic as a pushdown automaton. Note that NWAs can always be determinized [1] in contrast to more general pushdown automata. Our streaming algorithms will determinize NWAs constructed from XPATH expressions on the fly (as explained in [2]), so that we will only have to project deterministic NWAs but this while creating them on the fly.

An example for a deterministic NWAs is given in Fig. 3. It defines the XPATH filter `[//a/b]` which accepts all XML trees that contain some a-descendant with a b-child. Rules containing label sets represent sets of rules, one for each label. Node selection XPATH queries can be compiled to deterministic NWAs in a similar manner [2] by adding variables to the alphabet. This requires some minor extensions for NWA projection which are out of the scope of the present paper.

3 Projection NWAs

We next introduce projected nested words. Let $\mathbb{N}$ be the set of natural numbers, $\mathbb{N}_0 = \mathbb{N} \uplus \{0\}$, and $\mathbb{Z}$ the set of integers. For any unranked tree, we are interested in the binary node relations child ch, descendant ch^+, n-th grand parents ch^{-n} where $n \in \mathbb{N}$, descendants of n-th grand parents ch^{-n}/ch^+, children of n-th grand parents ch^{-n}/ch, and stay at *self*. So let:

$$Rels = \{ch, ch^+, ch^{-n}, ch^{-n}/ch^+, ch^{-n}/ch, self \mid n \in \mathbb{N}\}.$$

A *projected nested word* is a word whose letters are jump symbols $.^{\overset{i}{:}}.$ where $i \in \mathbb{Z}$ and jump targets $p@r$ where $p \in P_\Sigma$ and $r \in Rels$. We write $P_{\Sigma}^{\overset{\cdot}{:}\cdot}$ for the set of all these letters. We assume that any jump target is proceeded by a jump symbol that indicates the excess of the jump, that is the depth difference in the tree or equivalently, the difference of the numbers of opening and closing tags in the nested word. We also assume that projected nested words are well-nested up to jumping.

Two examples for projected nested words are given in Fig. 4. Both are valid descriptions of the nested word in Fig. 2: pw_1 projects to the letters drawn in blue, while pw_2 projects to the letters drawn in green. As we will see, both projections can be obtained from the NWA in Fig. 3. Note that the initial opening tag is always kept for technical reasons. Except of this, both projections are

pw_1: for all a-nodes without an a-parent and all non-a-children of a-nodes keep the opening and closing tags, until the opening tag of the first match of `//a/b`:

$$\langle \mathbf{c} \rangle \,.^{0}.\, \langle \mathbf{a} \rangle @ ch^{+} \,.^{2}.\, \langle \mathbf{c} \rangle @ ch^{+} \,.^{0}.\, \langle / \mathbf{c} \rangle @ self \,\therefore^{2}\, \langle \mathbf{c} \rangle @ ch^{-3} / ch^{+}$$
$$\therefore^{0}.\, \langle \mathbf{a} \rangle @ ch^{+} \,.^{0}.\, \langle / \mathbf{a} \rangle @ self \,.^{0}.\, \langle / \mathbf{c} \rangle @ ch^{-1} \,\therefore^{0}.\, \langle \mathbf{b} \rangle @ ch^{-1} / ch^{+}$$

pw_2: for all a-nodes and all b-children of a-nodes keep the opening and closing tags, until the opening tag of the first match of `//a/b`:

$$\langle \mathbf{c} \rangle \,.^{0}.\, \langle \mathbf{a} \rangle @ ch^{+} \,.^{0}.\, \langle a \rangle @ ch^{+} \,.^{0}.\, \langle a \rangle @ ch^{+} \,.^{0}.\, \langle / a \rangle @ self$$
$$\therefore^{0}.\, \langle / a \rangle @ ch^{-1} \,\therefore^{1}.\, \langle \mathbf{a} \rangle @ ch^{-1} / ch^{+} \,.^{0}.\, \langle / \mathbf{a} \rangle @ self \,\therefore^{1}\, \langle \mathbf{b} \rangle @ ch^{-2} / ch$$

Fig. 4: Two projected nested words describing the nested word in Fig. 2.

maximal, in that no further tags can be projected away: they just preserve enough information for deciding whether the original nested word satisfies the filter `[//a/b]`. Nevertheless, none of these two projections is more general than the other. The green projection pw_2 has the advantage to keep only tags with letters occurring in the XPath filter `[//a/b]`. The blue projection pw_1, has the advantage to keep fewer of these tags, but therefore, it also keeps some others.

The blue projection pw_1 starts with $\langle \mathbf{c} \rangle$, meaning that any matching nested word must start with $\langle c \rangle$. The next factor $.^{0}.\, \langle \mathbf{a} \rangle @ ch^{+}$ describes a nested word with excess 0 that is followed by $\langle a \rangle$ in descendant position, i.e., by the opening tag of an a-child of the root. The next factor $.^{2}.\, \langle \mathbf{c} \rangle @ ch^{+}$ describes a nested word with excess 2 followed by $\langle c \rangle$ opening a descendant. Then $.^{0}.\, \langle / \mathbf{c} \rangle @ self$ requires to jump with excess 0 to the closing tag $\langle / c \rangle$ of the same node. Next, $\therefore^{2}\, \langle \mathbf{c} \rangle @ ch^{-3} / ch^{+}$ asks to jump with excess -2 to an opening tag $\langle c \rangle$ of a descendant of a grand-grand-grand-parent, etc.

We next introduce PNwas as a mixture of a pushdown and a counting automaton, that receive projected nested words as input. The counting serves for updating the depths of nodes when jumping, so that the depth of the current node can always be deduced from the current stack. Whenever jumping over a projected factor, the excess of this factor is pushed. This is an integer that is popped when trying to close the jump.

Definition 1 *A* PNwa *is a tuple* $A = (\Sigma, Q, I, F, \Gamma, R)$ *like an* Nwa *but with different kinds of transition rules: given* $a \in \Sigma$, $\gamma \in \Gamma$, *and* $q, q' \in Q$, *there are rules of the following types in* R, *for changing the state from* q *to* q'.

Open: $q \xrightarrow{\langle a \rangle \downarrow \gamma} q'$ *Like for* Nwa*s.*

Close: $q \xrightarrow{\langle / a \rangle \uparrow \gamma} q'$**.** *Like for* Nwa*s.*

Jump to a child or a descendant: $q \xrightarrow[\forall z \geq 0]{\therefore^{z}. \langle a \rangle @ r \downarrow z \downarrow \gamma} q'$**, where** $r \in \{ch, ch^{+}\}$**.**

When $r = ch$ *then* z *must be* 0*, and we jump to the opening tag of an* a*-child and push first* 0 *and then* γ *onto the stack. When* $r = ch^{+}$ *then we jump over*

z descendants to the opening tag of an a-descendant, and push first z and then γ onto the stack. For short we denote this transition as $q \xrightarrow{ju(\langle a\rangle,r,\gamma)} q'$.

Rejump to another child or descendant: $q \xrightarrow[\forall z,z'.\ z'\geq 0, z+z'\geq 0]{\dddot{z}.\langle a\rangle @ ch^{-(z'+1)}/r\uparrow z'\downarrow z+z'\downarrow\gamma} q'$, **where** $r \in \{ch, ch^+\}$.

While trying to close a jump from some grand parent to some node one can rejump to another opening a-tag of a child or a descendant of the same grand parent. The excess of the jump to the first node z' on the stack is updated to the excess of the second node $z+z'$. Furthermore, γ is pushed. For short, we write this transition as $q \xrightarrow{reju(\langle a\rangle,r,\gamma)} q'$.

Jump to the closing tag of the self node: $q \xrightarrow{\dddot{0}.\langle/a\rangle @ \textbf{\textit{self}}\uparrow\gamma} q'$. *Jump to the closing tag of the self a-node. In this case, γ is popped from the stack.*

Jump back to the jump's origin: $q \xrightarrow[\forall z\geq 0]{\dddot{z}.\langle/a\rangle @ ch^{-(z+1)}\uparrow z\uparrow\gamma} q'$. *When trying to close a jump, one may jump back to the closing tag of the a-node where the current jump started. The excess of $-z$ is popped from the stack together with the symbol γ which was pushed for the non-jumped a-node. For short we write $q \xrightarrow{ju\text{-}back(\langle/a\rangle,\gamma)} q'$.*

Close last jump step: $q \xrightarrow[\forall z>0]{\langle/a\rangle\uparrow z\downarrow z-1} q'$. *When trying to close a jump, one may read a closing a-tag for which the corresponding opening a-tag was jumped, so that no stack symbol was pushed. In this case the excess of the jump on the stack must be updated from z to $z-1$.*

A configuration of a PNWA is a word in $Q(\Gamma \uplus \mathbb{N}_0)^*$ consisting of a state in Q and a stack in $(\Gamma \uplus \mathbb{N}_0)^*$. A run r of a PNWA A on a projected nested word over Σ is a function that maps any prefix of the projected nested word to a configuration. The run must start in some configuration with some initial state and the empty stack, i.e., $r(\epsilon) \in I$. Furthermore, for any prefix wl where $l \in P_{\dddot{\Sigma}}$, the configuration $r(w)$ must be transformed into $r(wl)$ by applying some rule consuming letter l. A run on a projected nested word w is called successful if it eventually reaches a configuration with a final state, i.e., if $r(w') \in F(\Gamma \uplus \mathbb{N}_0)^*$ for some prefix w' of w. The language $\mathcal{L}(A)$ of a PNWA A is the set of all projected nested words that permit a successful run on A.

In Fig. 5 we present PNWA $\mathbf{A}_1$ that is a projection of the NWA in Fig. 3 for the XPATH filter `[//a/b]`. This automaton accepts the blue projection pw_1 in Fig. 4 of the nested word in Fig. 2. Automaton $\mathbf{A}_1$ visits the opening and closing tags of all a-nodes with no a-parent, and of all non-a-children of these a-nodes, and jumps over all other nodes. Automaton $\mathbf{A}_1$ accepts when the first match of `[//a/b]` arrives. In Fig. 6, we illustrate a successful run of $\mathbf{A}_1$ on pw_1. The states of configurations are placed below the tags, while the stack consists of the labels on the subedges above the state. Edges between tags indicate their correspondence. Furthermore there are edges for jumps to children and descendants, where the excess is pushed, while jumps to the jump origin close the jump, and rejumps update the excess on the stack. The only exceptions are jumps to the

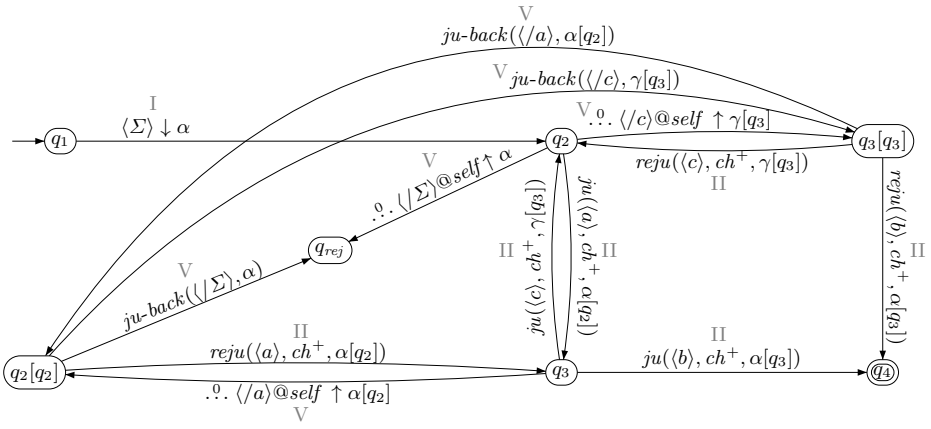

Fig. 5: PNwa $\mathbf{A}_1$ for the XPath filter `[//a/b]`.

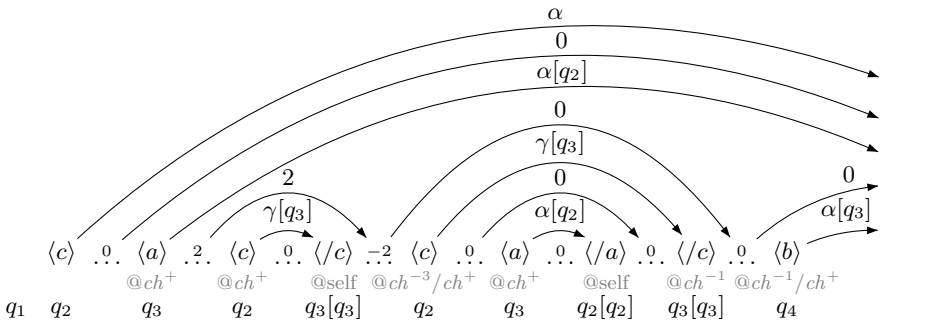

Fig. 6: A successful run of the PNwa $\mathbf{A}_1$ of Fig. 5 on pw_1.

closing tag of self nodes, where no excess is pushed. In general PNwa $\mathbf{A}_1$ works as follows. It starts in the initial state q_1, it opens the root and goes into q_2, where either the root can be closed to q_{rej} or where it can jump over b and c nodes to the opening tag of an a-descendant and go to q_3. There are 3 possibilities depending on what happens first: (1) close the a-node, and go to $q_2[q_2]$, (2) jump down over a sequence of a-nodes to the opening tag of a c-descendant and go to q_2, or (3) jump down over a sequence of a-nodes to the opening tag of a b-descendant and accept in q_4. In state q_2 a c-node with a sequence of a-grand parents can be closed to $q_3[q_3]$. The sequence of a-grand parents consists of a sequence of jumped a-nodes and one not jumped a-node π at the top. Continuing depending on what comes first, the following can happen in $q_3[q_3]$: (1) jump back to the closing tag of the a-grand parent π and go to $q_2[q_2]$, (2) rejump over a sequence of a-nodes, while staying below π, to the opening tag of a c-descendant and go to q_2, or (3) rejump over a sequence of a-nodes, while staying below π, to the opening tag of a b-descendant, and accept in state q_4. In state $q_2[q_2]$ there are 3 possibilities depending on what happens first: (1) rejump over a sequence

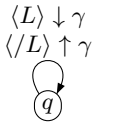

Fig. 7: $q \in i\text{-}label_L$.

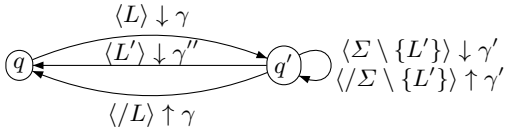

Fig. 8: $q \in i\text{-}tree_{L \setminus L'}$.

of b and c nodes to the opening tag of an a-descendant and goto q_3, while staying below a not-jumped c-grand parent with a sequence of a-grand parents if exists, (2) jump back to the closing tag of a not-jumped c-grand parent with the a-grand parents sequence if exists, or else (3) close the root to q_{rej}.

Next we are interested to evaluate a collection of PNWAS obtained from deterministic NWAs on a single nested word. For this, we need to project the nested word with respect to the PNWAS, and run the PNWAS on the respective projected nested word. Therefore, we have to define how to project a nested word w with respect to a deterministic PNWA. More generally, we define a projection $\pi_q(w)$ for any suffix w of some nested word in P_Σ^* and state q of a PNWA A:

$$\pi_q(w) \;= .^{i}_{!}.\; p@r\,\pi_{q'}(w'')$$

such that $w = w'pw''$ for some $p \in P_\Sigma$ and $w', w'' \in P_\Sigma^*$, where w' is the shortest prefix, so that there exists a rule of A from q to q' consuming letter $p@r$ for some $r \in \mathit{Rels}$, and i is the excess of w'.

4 Irrelevant Labels and Prefixes of Nested Words

In this section, we define properties of NWA states which allow to skip parenthesis with irrelevant labels and irrelevant prefixes of nested words, that is prefixes of linearizations of subtrees.

Definition 2 *An* NWA E *can jump over parenthesis with labels in* L *and incoming state* q *– in formulas* $q \in i\text{-}label_L$ *– if there exists a stack symbol* γ*, such that* E *has all transitions shown in Fig. 7, no other opening transition pushing* γ*, no other* a*-opening transition in* q*, and no other* a*-closing transition with* γ*.*

If $q \in i\text{-}label_L$ then any sequence of letters in P_L is irrelevant in state q, so that it can be removed from the nested word and replaced by a jump symbol. Consider a run of E on a nested word w and assume $q \in i\text{-}label_L$. We next argue, that we can replace all letters in P_L of w with ingoing state q by jump symbols, while "repairing" the run. The first point is that the state is not changed when reading such letters, so that their removal keeps the states correct. But we must also take care of the stack. If an opening tag $\langle a \rangle$ is removed but not the corresponding closing tag, then we have to repair the run, in order to be able to reproduce the missing stack symbol when needed. The idea is to memoize the state before jumping. Since this state does not change while jumping, one

can then recompute the stack symbol that was pushed for any letter that was jumped over. Conversely, it is not possible that a closing tag $\langle /a \rangle$ was removed but not the corresponding opening tag, since the symbol pushed at $\langle a \rangle$ must be γ, and by definition of $q \in i\text{-}label_L$ there is no other opening transition pushing γ than that started in q.

Definition 3 *An* NWA E *in state* q *can jump over prefixes of nested words (subtrees) that start in* $\langle L \rangle$*, do not contain letters in* $P_{L'}$*, and either end with the closing tag of the subtree's root or with a letter in* L'*, if there exist three different stack symbols* $\gamma, \gamma', \gamma''$ *and a state* q' *such that the transitions shown in Fig. 8 exist, but no further opening transitions with* γ*, no further transitions with* γ'*, and no further opening transitions in* q' *for* L'*, and no further closing transition in* q *for* L *popping* γ*. In this case, we write* $q \in i\text{-}tree_{L \setminus L'}$ *and call* q *a state of irrelevant subtrees.*

In the easiest case where $q \in i\text{-}tree_{L \setminus \emptyset}$ one can jump over nested words linearizing subtrees, with incoming state q and labels in L only. When opening the root of the subtree, the state changes to q' and stays there until closing the root and going back to q. So the removal of the subtree does not change the state globally. In this case, the full nested word of the subtree is read, so the stack difference is zero. In the case where $L' \neq \emptyset$ it is more tricky to repair the run, in order to deal with missing stack symbols. But it remains possible, since the state used within the subtree does not change, so that it can be memoized and so that missing stack symbols can be recomputed at closing time.

For illustration, we have annotated the state of the NWA in Fig. 3 with the properties that they satisfy. It turns out that state q_3 satisfies both properties $i\text{-}label_{\{a\}}$ and $i\text{-}tree_{\{c\} \setminus \{a\}}$, but that we cannot perform the two corresponding projections at the same time. When choosing projection with $i\text{-}label_{\{a\}}$ then we obtain the PNWA $\mathbf{A}_1$ from Fig. 5.

5 Projection from Nwas to PNwas

We show how to project deterministic NWAs E to a PNWA A. For any state q of E, we chose a projection property $choice(p)$, which is either $i\text{-}label_L$ or $i\text{-}tree_{L \setminus L'}$ for some sets $L, L' \subseteq \Sigma$. Note that $i\text{-}label_\emptyset$ can always be assigned, so that this assumption can always be satisfied, but not always in a unique manner.

Any state of A is either a state of q of E or a pair of states of E that we write as $q[q']$. Such a pair means that one is in state q and that on the top of the stack is a jump symbol i that was pushed from a jump over i descendants that started in state q'. Any stack symbol of A is either a stack symbol γ of E or a pair written as $\gamma[q]$ of a stack symbol and a state of E. γ serves as the stack symbol that was pushed before at opening tags, while q is the state where a previous jump started. Whenever such a pair $\gamma[q]$ is on the stack then the symbol below is always a jump symbol i that was pushed by a jump over i descendants that started in state q. The sets of initial and final states remain unchanged.

Every transition rule of E gives rise to a possible empty set of transition rules of A, according to rules I–VII in Fig. 9. In PNWA $\mathbf{A}_1$ from Fig. 5 we annotated transitions accordingly. Transitions from an initial state are translated to non-jumping transitions that open the root. If $\mathit{choice}(q) = \mathit{i\text{-}label}_L$, then all looping transitions required by $\mathit{i\text{-}label}_L$ are removed. The other opening transitions starting from q are translated to jumping and rejumping transitions to descendants and descendants of grand parents. If $\mathit{choice}(q) = \mathit{i\text{-}tree}_{L\setminus\emptyset}$ then the opening and closing L transitions, and looping transitions required by $\mathit{i\text{-}tree}_{L\setminus\emptyset}$ are removed. The other opening transitions starting from q are translated to jumping and rejumping rules to children and children of grand parents. If $\mathit{choice}(q) = \mathit{i\text{-}tree}_{L\to L'}$ then the opening and closing L transitions, and looping transitions required by $\mathit{i\text{-}tree}_{L\to L'}$ are removed. All other transitions with opening tag $a \in L'$ departing q are translated to jumping and rejumping rules for descendants. Closing transitions are translated to six rules: Two rules to close self nodes, two rules to jump back to jump's origins, and two last rules that close parents in a state $q[q'']$ for $q \neq q''$. Those states do not allow to rejump, since the previous jump started in a different state q'' than the current state q, and therefore they also do not allow to jump back to the jump's origin. For these states opening and closing transitions are translated as indicated, while recomputing stack symbols, that have not been pushed for jumped grand parents.

Proposition 1 (Soundness) *Let E be a deterministic* NWA *E with initial state q_0 and A be a* PNWA *obtained from E by our projection algorithm. It then holds for any nested word w that $w \in \mathcal{L}(E)$ if and only if $\pi_{q_0}(w) \in \mathcal{L}(A)$.*

6 Experiments

We implemented NWA projection within the QUIXPATH system [3] and tested it on the (revised) XPATHMARK query set [4] for navigational queries. As argued in the introduction, it is most natural to measure the efficiency in *parsing-free time* which can be measured as described in [2].

In a first experiment, we start from the best existing XPATH evaluator on XML streams so far which is based on NWAS [2] (see there for comparisons to alternative tools by [7,12] and others), and enhance it with projection. The results are presented in Fig. 10 for a 559 MB XPATHMARK document. It turns out that projection reduces the parsing-free running time for this query set by a factor of 4.3, which is a major improvement, in particular when evaluating many XPATH queries in parallel as needed for streaming XSLT or XQUERY programs. In our second experiment, we compare the overall running time of our PNWA evaluator of XPATH queries on XML streams with SAXON's in-memory evaluator [7]. For each of our queries, we compare the full running times including parsing, when evaluating the query n-times. The results are given in Fig. 11. It turns out that QUIXPATH with projection for NWAS can answer on average a query up to 12 times in parallel, in no more time than needed by SAXON for the same task.

One observes that running less than 12 queries in parallel with PNWAS is a lot quicker than running them with SAXON, mostly due to the expensive in-memory

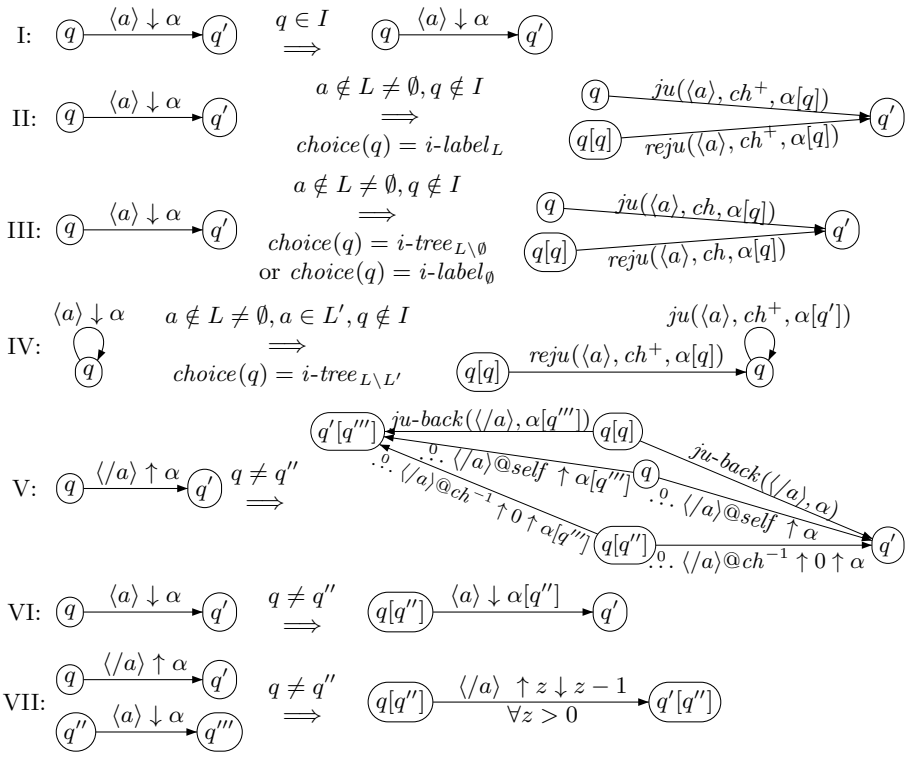

Fig. 9: Rewriting system for rules of a deterministic NWA to rules of the PNWA.

tree creation. But when running more than 12 queries on small documents, the advantage of in-memory evaluation takes over. Indeed, without the time for parsing and in-memory tree construction, SAXON in-memory evaluation is still faster by a factor 20 in average than streaming with PNWAS. With the improvements of the present paper, it now seems possible that stream processing can become more efficient than in-memory evaluation in practice in the future.

Conclusion and Future Work

We have developed a projection algorithm for evaluation navigational XPATH queries on XML streams. The next step will be to lift this algorithm to all of XPATH 3.0. We believe that this can be done by decomposing general XPATH queries into a network of navigational XPATH queries. Such a decomposition underlies the implementation of XPATH 3.0 in our QUIXPATH tool [3], which is unpublished so far. Once this is done, one can hope to lift our XPATH projection to XSLT and XQUERY, by using X-FUN as an intermediate language [8].

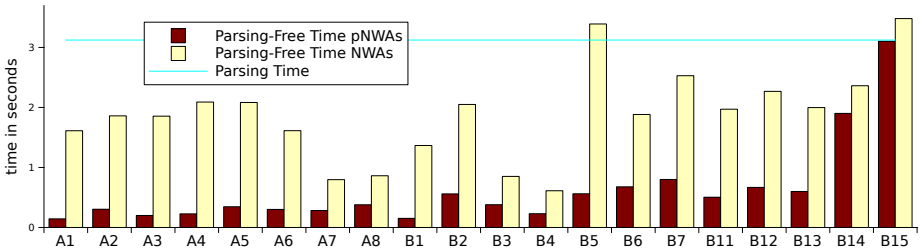

Fig. 10: Improvement by NWA projection of XPATH query evaluation.

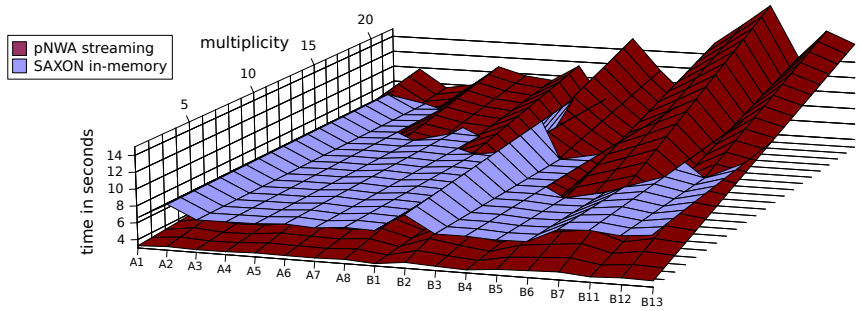

Fig. 11: Streaming wins vs. in-memory evaluation for up to 12 queries in parallel.

References

1. R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009.
2. D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, and M. Zergaoui. Early nested word automata for XPath query answering on XML streams. *TCS 578*: 100-125. 2015.
3. D. Debarbieux, T. Sebastian, M. Zergaoui, and J. Niehren. Quix-tool suite. https://project.inria.fr/quix-tool-suite/, 2014.
4. M. Franceschet. XPathMark: An XPath benchmark for the XMark generated data. In *3rd International XML Database Symposium*, 2005.
5. A. Frisch. Regular tree language recognition with static information. In *IFIP TCS*, pages 661–674, 2004.
6. O. Gauwin and J. Niehren. Streamable fragments of forward XPath. In *CIAA*, volume 6807 of *LNCS*, pages 3–15. 2011.
7. M. Kay. The saxon XSLT and XQuery processor. https://www.saxonica.com.
8. P. Labath and J. Niehren. A uniform programming language for implementing XML standards. In *SOFSEM*, 2015.
9. P. Madhusudan and M. Viswanathan. Query automata for nested words. In *MFCS*, volume 5734 of *LNCS*, pages 561–573. 2009.
10. S. Maneth and K. Nguyen. XPath whole query optimization. *VLPB Journal*, 3(1):882–893, 2010.
11. A. Marian and J. Siméon. Projecting XML documents. In *VLDB*, 213–224, 2003.
12. B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over XML streams. In *SIGMOD Conference*, pages 253–264. ACM, 2012.