Deterministic Regular Expressions in Linear Time

Benoît Groz
Mostrare, INRIA
University of Lille, France

Sebastian Maneth
NICTA and UNSW
Sydney, Australia

Sławek Staworko
Mostrare, INRIA
University of Lille, France

ABSTRACT

Deterministic regular expressions are widely used in XML processing. For instance, all regular expressions in DTDs and XML Schemas are required to be deterministic. In this paper we show that determinism of a regular expression e can be tested in linear time. The best known algorithms, based on the Glushkov automaton, require $O(\sigma|e|)$ time, where σ is the number of distinct symbols in e. We further show that matching a word w against an expression e can be achieved in combined linear time $O(|e| + |w|)$, for a wide range of deterministic regular expressions: (i) star-free (for multiple input words), (ii) bounded-occurrence, i.e., expressions in which each symbol appears a bounded number of times, and (iii) bounded plus-depth, i.e., expressions in which the nesting depth of alternating plus (union) and concatenation symbols is bounded. Our algorithms use a new structural decomposition of the parse tree of e. For matching arbitrary deterministic regular expressions we present an $O(|e| + |w| \log \log |e|)$ time algorithm.

Categories and Subject Descriptors: F.2.2, I.1.1

General Terms: Algorithms

Keywords: DTD, XML Schema, Deterministic Regular Expression, Glushkov Automaton, Linear Time.

1. INTRODUCTION

Deterministic regular expressions are widely used in XML processing. For instance, all regular expressions in DTDs and in XML Schemas are required to be deterministic. The idea stems from the earlier SGML standard where right-hand sides of context-free productions ("content models") are deterministic regular expressions. Such expressions can be parsed more efficiently than unrestricted ones.

Within XML databases and XML processing, the two main tasks performed over regular expressions are (1) testing determinism and (2) matching (= parsing) against (child sequences of) the given input document.

Testing Determinism. The original terminology used in SGML was to restrict content models to be "unambiguous"; it means that at any position in the regular expression (positions are labeled by symbols, such as a or b, but not by operators such as $*$) and for each symbol there may be *at most* one position that follows. For instance, the expression ab^*b is ambiguous because the a-position is followed by two b-positions. Intuitively, the parser upon reading ab has to choose against which b to parse. There exist many translations from regular expressions to finite automata, see e.g. [16]. The particular translation due to Glushkov [12] (see also [2]) associates to each position a state of the automaton. As Brüggemann-Klein shows [8], a regular expression is unambiguous if and only if its Glushkov automaton is deterministic. We therefore use the term "deterministic regular expression." Brüggemann-Klein's result allows to test determinism of an expression e as follows: (i) build the Glushkov automaton A of e and (ii) test determinism of A. The worst-case size of A is $O(\sigma|e|)$, where σ is the number of distinct symbols in e, and A can be built and checked for determinism in this time bound. Thus, this test has *quadratic* time complexity in the size of e. It is a general misconception in the literature that testing determinism of regular expressions can be performed in linear time (cf. e.g., the abstract of [8]). The known algorithms build the Glushkov automaton in quadratic worst-case time. Note that large alphabets appear in practice, and that the quadratic behavior of building the Glushkov automaton is experienced even for very simple expressions such as $E = (a_1 + a_2 + \cdots + a_m)^*$.

For the expression E, determinism can easily be checked in linear time (by checking distinctness of the a_i). The "mixed content" of XML, for instance, is similar to E and some XML validators such as Xerces [11] use specialized linear time procedures for this case. For more complicated expressions, however, it has remained open whether linear time determinism testing is possible. Here we close the problem affirmatively and show that all regular expressions can be tested for determinism in time $O(|e|)$. Our idea is a new decomposition of e's parse tree. For each distinct symbol a of e we build its "skeleton"; roughly speaking, it is a tree consisting of all positions labeled a, plus their iterated LCAs (lowest common ancestors) in e's parse tree; skeleton trees can be obtained in linear time [7] using preprocessing and constant time LCA [1] queries. By adding more nodes and pointers into the skeleton trees, we are able to test determinism in linear time.

Matching. Consider now matching a deterministic regular expression e against an input word w. What makes it

PODS '12, May 21–23, 2012, Scottsdale, Arizona, USA.
Copyright 2012 ACM 978-1-4503-1248-6/12/05 ...$10.00.

difficult to match e against w? We identify several "easy" cases: (i) Star-free: in this case, $|w| \leq |e|$, and we can match easily during one traversal over the parse tree of e. (ii) Bounded number of distinct symbols in e: we simply build the Glushkov automaton. (iii) Bounded number of occurrences of each symbol in e (k-occurrence): Here we use our first technical lemma. It says that testing if two positions follow each other in e (this means the Glushkov automaton has a transition between the positions) can be realized in constant time. This is achieved by preprocessing e's parse tree for LCA [1] and by using LCA queries to realize a structural relationship of follow positions known from [9, 25]. Hence, we do *not* build the Glushkov automaton. Since at most k positions need to be checked for the follow relationship when matching against a k-occurrence expression, the lemma implies linear time $O(|e|+k|w|)$. Note that for real world DTDs it has been reported that a large percentage of regular expressions is k-occurrence for small k [3, 21].

One finding is that plus-symbols (union) play an essential role in the combined complexity of matching. For instance, if no plus-symbols in e are nested, then we show that matching can be done in time $O(|e| + |w|)$. The idea is to annotate particular nodes in e's parse tree with candidates of follow-positions in e. The determinism requirement of e then allows to *amortize* the number of annotated nodes that have to be visited in order to go from one symbol in w to the next.

Our amortization argument fails when the *depth of alternation* of plus and concatenation symbols is not bounded. Such expressions seem the hardest to match, and finding a time $O(|e| + |w|)$ algorithm remains an open problem. Note that the alternation depth is small in practice: Grijzenhout's large collection of real-world DTDs [13] does not contain a single expression with alternation depth larger than 4. We present an algorithm with time $O(|e| + |w| \log\log |e|)$ complexity that works for arbitrary deterministic expressions. It is derived from our linear time determinism test which assigns colors (i.e., labels in Σ) to nodes of e's parse tree. At a position p, the next position labeled a is obtained by a lookup at the lowest ancestor of p with color a. The expression e is preprocessed for lowest colored ancestor queries, using [23] (based on van Emde Boas trees). Note that for arbitrary (nondeterministic) regular expressions, the best known time complexities are time $O(nm/\log n + (n+m)\log n)$ [24] which was improved recently to $O(nm(\log\log n)/(\log n)^{3/2} + n + m)$ [6], where $m = |e|$ and $n = |w|$.

Our results are summarized as follows:

(1) Determinism of a regular expression e can be tested in time $O(|e|)$. This improves previous algorithms requiring quadratic time. Besides a direct proof, we present an alternative one which uses a fixed XPath query, and then applies the result of Bojańczyk and Parys [7].

(2) Deterministic regular expression can be matched in time $O(|e| + |w|)$ against an input word w, if

   (a) each symbol occurs only a bounded number of times in e ("k-occurrence"), or

   (b) the maximal depth of alternating union and concatenation operations in e is bounded.

(3) Star-free deterministic regular expression can be matched against several input words $w_1, \ldots, w_n$, in time $O(|e| + |w_1| + \cdots + |w_n|)$.

(4) Arbitrary deterministic regular expression can be matched in time $O(|e| + |w| \log\log |e|)$.

Recently it was proved that even in the presence of numeric occurrence indicators (as used in XML Schema), determinism of expressions can be tested in time $O(\sigma|e|)$ [18]. We show that our result extends to this case: even in the presence of numeric occurrence indicators we can decide determinism in time $O(|e|)$. We note that all our matching algorithms are *streamable*, i.e., they do not need to store w in memory, but read w in one sequential pass, symbol by symbol. We have implemented all our algorithms and made them available at http://gforge.inria.fr/projects/lire/.

Related Work

The idea of our algorithm (3), and also to a lesser extent of our determinism check (1), is similar to that of Hagenah and Muscholl [14] in their algorithm that computes for any regular expression an ε-free NFA in time $\Omega(|e| \log^2 |e|)$. They decompose the transitions leaving each state into a few sets and group states sharing such sets of outgoing transitions. This decomposition is based on a *heavy path* decomposition of the parse tree of e. We use another decomposition of this parse tree in order to amortize the evaluation cost.

An orthogonal direction of research involves algorithms for the efficient validation of huge documents against a small DTD. Several works [27, 28] focused on obtaining space efficient algorithms in a streaming framework. This is challenging when document trees are deep. Konrad and Magniez [20] provide streaming algorithms in sublinear space for the validation against DTDs. They consider a framework where the algorithm has access to a read-only input stream and several auxiliary read/write streams. The algorithm is allowed to perform read or write passes on the streams. At the beginning of each pass on a stream, the algorithm decides in which direction the stream is processed, and also decides if the pass is a write or a read pass. The authors propose an algorithm that validates a tree t against a constant-size DTD in $O(\log^2 |t|)$ passes, using space $O(\log |t|)$ and 3 auxiliary streams, with $O(\log |t|)$ processing time per symbol. Note that the validator checks the sibling sequences of t against the corresponding deterministic regular expression.

In the context of DTD inference, Bex et al. identify two classes of regular expressions which account for most of the regular expressions in real schemas: the single occurrence regular expressions (1-ORE) and the chain regular expressions (CHARE). An expression is an 1-ORE iff no symbol appears more than once in e, therefore 1-ORE are always deterministic. CHARE are a subclass of 1-ORE, and contain the 1-ORE that consist of a sequence of factors of the form $(a_1 + a_2 + \cdots + a_n)$ where every a_i is a symbol, each factor being possibly extended with a star or a question mark. 1-ORE account for 98% of the regular expressions in real schemas, while CHARE account for 90% of them. Bex, Neven, and van den Bussche [4] also define *simple regular expressions*, which generalize CHARE in that symbols a_i in factors can appear with a star or question mark, and the number of occurrences of a symbol is not restricted.

The class of expressions for which our algorithm (2b) performs in linear time properly contains deterministic simple regular expressions. Moreover although stars are allowed in simple regular expressions, which makes them unfit for algorithm (3), those stars can occur only above a single symbol, or above a union of strings (with possibly a star or question mark above the strings). Therefore, an easy extension of algorithm (3) handles simple deterministic regular expressions.

2. REGULAR EXPRESSIONS

Let Σ be a finite set of symbols. *Regular expressions over* Σ are defined by the following grammar, where $\odot$ represents concatenation, $+$ union, $?$ choice, and $*$ the Kleene star: $e := a(a \in \Sigma) \mid (e)\odot(e) \mid (e)+(e) \mid (e)? \mid (e)^*$. The language $L(e)$ of e is defined as usual [16]. Note that $L((e)?) = L(e) \cup \{\varepsilon\}$, where ε denotes the empty word. We say that e is *nullable* if $\varepsilon \in L(e)$. In expressions, we do not write parentheses around words over Σ and often omit $\odot$ symbols. We require of our regular expressions e that:

(R1) $e = (\#e')\$$ and $\#$ and $\$$ do not appear in e'

(R2) $((e')^*)^*$ does not appear in e

(R3) if $(e')?$ appears in e, then $\varepsilon \notin L(e')$

An arbitrary regular expression can be changed easily (in linear time) into an equivalent one of the required form. Note that # and \$ are tacitly present and required, but, for better readability, are omitted in most examples.

We identify a regular expression with its parse tree (as illustrated in Figure 1), and define the *positions* $Pos(e)$ of e as the leaves of e whereas N_e denotes the set of all nodes from e. For a node $n \in N_e$ we denote by e/n the subexpression of e rooted at n. Every tree t is implemented as a pointer structure, where $Lchild_t(n)$ (resp. $Rchild_t(n)$) returns the left (resp. right) child of node n in t and $parent_t(n)$ returns the parent of n in t. The pointers return *Null* if the respective node does not exist. For unary nodes $Rchild_t(n)$ returns *Null*. We denote by $lab_t(n)$ the label of n in t, and by $\preccurlyeq_t$ the (reflexive) ancestor relationship in t. If $m \preccurlyeq_t n$ then we also say that n is a descendant of m. Thus, each node is ancestor and descendant of itself.

The *size* of a tree t, denoted $|t|$, is the number of nodes in t, whereas the *depth* of t $depth(t)$ is the length of path from the root to the deepest node in t. Our restrictions (R2) and (R3) guarantee that $|e|$ is linear in $|Pos(e)|$. We denote by $\overline{e}$ the regular expression obtained from e by marking the i-th position (from left to right) with subscript i. We denote by $\overline{\Sigma}$ the set of symbols obtained from Σ by adding subscripts below symbols. In particular, $Pos(\overline{e}) = Pos(e)$.

Given a position p of e, $Follow_e(p)$ is the set of positions that may follow p in e:

$$Follow_e(p) = \{q \mid \exists u, v \in \overline{\Sigma}^*, u \cdot lab_{\overline{e}}(p) \cdot lab_{\overline{e}}(q) \cdot v \in L(\overline{e})\}.$$

The expression e is *deterministic* if for all $p, q, q' \in Pos(e)$ with $q, q' \in Follow_e(p)$: $q \neq q'$ implies that $lab_e(q) \neq lab_e(q')$. Whenever the regular expression or the tree is clear from context, we drop the subscript and write *Follow*, *lab*, and $\preccurlyeq$.

Example 2.1. *Let* $e_1 = (ab + b(b?)a)^*$ *and* $e_2 = (a^*ba + bb)^*$. *Denote by* $p_1, \ldots, p_5$ *the positions of* e_1 *in left-to-right order, and by* $q_1, \ldots, q_5$ *those of* e_2. *Then* $\overline{e_1} = (a_1b_2 + b_3(b_4?)a_5)^*$ *and* $Follow_{e_1}(p_3) = \{p_4, p_5\}$. *Similarly,* $\overline{e_2} = (a_1^*b_2a_3 + b_4b_5)^*$, *and* $Follow_{e_2}(q_3) = \{q_1, q_2, q_4\}$. *Expression* e_1 *is deterministic, while* e_2 *is non-deterministic because* $lab_{e_2}(q_2) = lab_{e_2}(q_4) = b$.

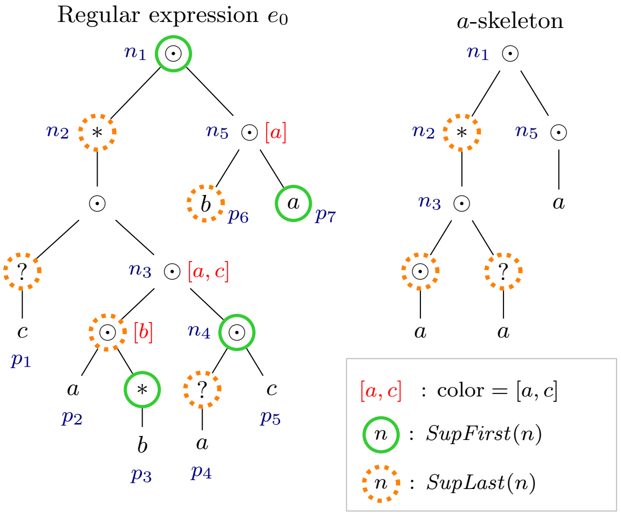

Figure 1: Expression $e_0 = (c?((ab^*)(a?c)))^*(ba)$**.**

Structure of Regular Expressions

The *First* and *Last*-positions of a regular expression e are

$$\begin{aligned} First(e) &= \{p \mid \exists u \in \overline{\Sigma}^*, lab_{\overline{e}}(p) \cdot u \in L(\overline{e})\} \\ Last(e) &= \{p \mid \exists u \in \overline{\Sigma}^*, u \cdot lab_{\overline{e}}(p) \in L(\overline{e})\}. \end{aligned}$$

We also define, for a node n of e, $First(n)$ and $Last(n)$ as $First(e/n)$ and $Last(e/n)$, respectively. Note that $First(n)$ and $Last(n)$ are non-empty for every node n of e. For instance, for the expression e_0 in Figure 1 $First(n_2) = \{p_1, p_2\}$ and $Last(n_2) = \{p_5\}$.

Given two nodes u, v of e, let $LCA(u, v)$ denote the lowest common ancestor of u and v in e. The next lemma was stated before, e.g., in [9, 25], but not in terms of *LCA*.

Lemma 2.2. *Let* $p, q \in Pos(e)$ *and* $n = LCA(p, q)$. *Then* $q \in Follow(p)$ *iff*

(1) $lab(n) = \odot$, $q \in First(Rchild(n))$, $p \in Last(Lchild(n))$, *or*

(2) $q \in First(s)$ *and* $p \in Last(s)$ *where* s *is the lowest* $*$*-labeled ancestor of* n.

Lemma 2.2 says that there are only two ways in which positions follow each other: (1) through a concatenation, or (2) through a star. We write $q \in Follow_e^{\odot}(p)$ if (1) is satisfied, and $q \in Follow_e^*(p)$ if (2) is satisfied. For instance, in e_0 (Figure 1), we have $p_4 \in Follow_{e_0}^{\odot}(p_3)$ and $p_1 \in Follow_{e_0}^*(p_5)$. Note, however, that there may exist some positions p and q that satisfy simultaneously (1) and (2).

It was also observed earlier, e.g., [9, 25, 14], that First and Last-sets (and nullability) can be defined in a syntax-directed way over the parse tree of e. For instance, if $lab(n) = \odot$ and $Lchild(n)$, $Rchild(n)$ are non-nullable then $First(n) = First(Lchild(n))$ and $Last(n) = Last(Rchild(n))$. We define now the Boolean properties *SupFirst* and *SupLast* for every node n, where n' denotes $parent(n)$:

$$SupFirst(n) \text{ iff } lab(n') = \odot, n = Rchild(n'), \text{ and } Lchild(n') \text{ is non-nullable}$$

$$\mathit{SupLast}(n) \text{ iff } \mathit{lab}(n') = \odot, n = \mathit{Lchild}(n'), \text{ and } \mathit{Rchild}(n') \text{ is non-nullable.}$$

If *SupFirst*(n) then the First-set changes at n's parent: $\mathit{First}(\mathit{parent}(n)) \cap \mathit{First}(n) = \varnothing$, and otherwise is a superset: $\mathit{First}(\mathit{parent}(n)) \supseteq \mathit{First}(n)$. For instance, in e_0 (Figure 1), n_4 is a *SupFirst*-node since $\mathit{First}(n_3) = \{p_2\}$ and $\mathit{First}(n_4) = \{p_4, p_5\}$. This explains the name "*SupFirst*": a node with this property is "maximal" with respect the First-sets of its direct descendants (without the property). The same holds for *SupLast* and *Last*. We define for any node n, the pointers $\mathit{pSupFirst}(n)$ and $\mathit{pSupLast}(n)$ as the lowest ancestors x of n such that $\mathit{SupFirst}(x)$ and $\mathit{SupLast}(x)$, respectively. Recall that by (R1), $e = (\#e')\$$; this implies that for every node of e', both $\mathit{pSupFirst}(n)$ and $\mathit{pSupLast}(n)$ are defined. These definitions will never be applied to the "help nodes" of e that are not in e' (such as the root node of e); note however, that the root n_1 in Figure 1 is a *SupFirst*-node (because of the phantom position # not shown in the figure). We can check membership in First and Last, using *pSupFirst* and *pSupLast*.

Lemma 2.3. *Let $p \in \mathit{Pos}(e)$ and $n \in N_e$.*

(1) $p \in \mathit{First}(n)$ iff $\mathit{pSupFirst}(p) \preccurlyeq n \preccurlyeq p$, and

(2) $p \in \mathit{Last}(n)$ iff $\mathit{pSupLast}(p) \preccurlyeq n \preccurlyeq p$.

It is well-known, see [15, 1], that arbitrary LCA queries on a tree t can be answered in constant time, after preprocessing of t in linear time. For positions p and q, define the Boolean $\mathit{checkIfFollow}(p, q)$ as true iff $q \in \mathit{Follow}(p)$.

Theorem 2.4. *After preprocessing of e in $O(|e|)$ time, $\mathit{checkIfFollow}(p, q)$ can be answered in constant time for every $p, q \in \mathit{Pos}(e)$.*

Proof. First preprocess e for LCA queries. Next, add to each node n of e the pointers $\mathit{pSupLast}(n)$, $\mathit{pSupFirst}(n)$, and $\mathit{pStar}(n)$. The latter points to the lowest $*$-labeled ancestor of n. Clearly, this preprocessing can be carried out in time $O(|e|)$. We are ready to compute $\mathit{checkIfFollow}(p, q)$ in constant time: first obtain $n = \mathit{LCA}(p, q)$. By Lemmas 2.2 and 2.3 we return *true* (1) if $\mathit{lab}(n) = \odot$, $\mathit{pSupFirst}(q) \preccurlyeq \mathit{Rchild}(n)$, and $\mathit{pSupLast}(p) \preccurlyeq \mathit{Lchild}(n)$. These conditions can be checked in constant time ($n \preccurlyeq n'$ can be realized, e.g., by testing if $\mathit{LCA}(n, n') = n$). If case (1) fails then we compute $n' = \mathit{pStar}(n)$ and check in constant time if $\mathit{pSupFirst}(q) \preccurlyeq n'$ and $\mathit{pSupLast}(p) \preccurlyeq n'$. We return *true* if the checks succeed and *false* otherwise. □

The following technical lemmas state relationships between positions and their *pSupFirst* and *pSupLast* nodes.

Lemma 2.5. *Let $p, q \in \mathit{Pos}(e)$ and $q \in \mathit{Follow}_e(p)$. Then*

(1) $\mathit{parent}(\mathit{pSupFirst}(q)) \preccurlyeq p$ and

(2) $\mathit{parent}(\mathit{pSupLast}(p)) \preccurlyeq q$.

Proof. To show (1), assume that $\mathit{parent}(\mathit{pSupFirst}(q))$ is not an ancestor of p. Then $n = \mathit{LCA}(p, q)$ is an ancestor of $\mathit{parent}(\mathit{pSupFirst}(q))$, hence $\mathit{pSupFirst}(q) \not\preccurlyeq n$. By Lemma 2.3(1) we obtain $q \notin \mathit{First}(n)$ and therefore, by Lemma 2.2, q does not follow p. Point (2) can be proved similarly. □

Lemma 2.6. *Let p and q be two positions of e such that q follows p. If $\mathit{pSupLast}(p) \preccurlyeq \mathit{parent}(\mathit{pSupFirst}(q))$ then $\mathit{pSupFirst}(q)$ is nullable.*

Proof. Let $p, q \in \mathit{Pos}(e)$ such that $q \in \mathit{Follow}(p)$ and $\mathit{pSupLast}(p) \preccurlyeq \mathit{parent}(\mathit{pSupFirst}(q))$, and let $x = \mathit{LCA}(p, q)$. Assume first that $q \in \mathit{Follow}^{\odot}(p)$. Then $\mathit{lab}(x) = \odot$ and there are no *SupLast* nodes between p and $\mathit{pSupLast}(p)$ except $\mathit{pSupLast}(p)$. It means that in particular $\mathit{Rchild}(x)$ is nullable. Hence $\mathit{pSupFirst}(q)$ is nullable if it is the right-child of x. Otherwise $\mathit{pSupFirst}(q)$ is an ancestor of x. In that case, there are no *SupFirst* nodes between q and $\mathit{pSupFirst}(q)$, except $\mathit{pSupFirst}(q)$, so that $\mathit{Lchild}(x)$ is nullable. Consequently, x is nullable, and there are no *SupFirst* nor *SupLast* nodes between x and $\mathit{pSupFirst}(q)$, except the node $\mathit{pSupFirst}(q)$. Therefore, $\mathit{pSupFirst}(q)$ is nullable. The case $q \in \mathit{Follow}^*(p)$ is handled similarly: $\mathit{pStar}(x)$ is nullable and satisfies $\mathit{pSupFirst}(q) \preccurlyeq \mathit{pStar}(x) \preccurlyeq x$. Moreover there are no *SupFirst* nor *SupLast* nodes between x and $\mathit{pSupFirst}(q)$, except $\mathit{pSupFirst}(q)$. Thus, $\mathit{pSupFirst}(q)$ is nullable. □

3. TESTING DETERMINISM

To test determinism we need to check for every $a \in \Sigma$ and positions $q \neq q'$ labeled a whether there exists a p such that q and q' follow p. The challenge of a linear time algorithm is to deal with the quadratically many candidate pairs (q, q').

3.1 Candidate Pair Reduction

We define the following condition:

(P1) for all $q \neq q'$ in $\mathit{Pos}(e)$, $\mathit{pSupFirst}(q) = \mathit{pSupFirst}(q')$ implies $\mathit{lab}(q) \neq \mathit{lab}(q')$.

Clearly, if (P1) is false then e is non-deterministic. To see this, let $q \neq q'$ and $n = \mathit{pSupFirst}(q) = \mathit{pSupFirst}(q')$. Since the First and Last sets of any node are non-empty, there exists a p in $\mathit{Last}(\mathit{Lchild}(\mathit{parent}(n)))$. Note that $\mathit{parent}(n) = \mathit{LCA}(p, q) = \mathit{LCA}(p, q')$. By Lemma 2.2, $q, q' \in \mathit{Follow}_e(p)$, and hence by definition of determinism, $\mathit{lab}(q) \neq \mathit{lab}(q')$. Testing (P1) in linear time is straightforward: during one traversal of e we group the positions with same *pSupFirst*-pointer; for each group we check that all contained positions have distinct labels. This can easily be achieved in linear time, using an adapted bucket sorting algorithm. Therefore we assume from now on that (P1) is true.

According to Lemma 2.5(1) we store information about p in the parent of $\mathit{pSupFirst}(p)$. For each position p labeled a, we

- assign *color* a to the node $\mathit{parent}(\mathit{pSupFirst}(p))$
- say that position p is a *witness* for color a in the node $\mathit{parent}(\mathit{pSupFirst}(p))$.

Observe that each node may be assigned several colors, but, since (P1) holds, each node has at most one witness per color. In Figure 1, node n_3 has colors a and c. The witness for color a (resp. c) in n_3 is p_4 (resp. p_5). Lemma 2.5 states that a position q labeled a that follows p is a witnesses for color a in *some ancestor* of p. Thus, if two positions labeled a follow p, then each of them is witness for color a in ancestors of p.

We say that a node $n \in N_e$ has *class* a if n has color a, or n is a position labeled a, or n is the lowest common ancestor of two nodes of class a. The *a-skeleton* t_a of e consists of all nodes n of class a plus their *pSupLast* and *pStar* nodes (as defined in Section 2). The node labels in t_a are taken over from e, and the tree structure is inherited from e: n' is the left (resp. right) child of n in t_a if (1) n' is in the subtree of the left (resp. right) child of n in e, (2) $n \preccurlyeq n'$, and (3) there is no n'' in t_a with $n \preccurlyeq n'' \preccurlyeq n'$. If a node has no left (resp. right) child defined in this way, then the corresponding pointer is set to *Null*. Note that a node in t_a can be labeled $\odot$ or $+$ and have its left (or right) child point to *Null*. Figure 1 presents a regular expression and its a-skeleton.

Lemma 3.1. *The collection of a-skeleta for all $a \in \Sigma$ can be computed in time $O(|e|)$.*

Proof. The size of the a-skeleton is linear in the number of positions labeled a in e. Hence the size of the collection of a-skeleta is linear in $|e|$. The skeleta can be constructed in linear time by simply applying LCA repeatedly, inserting each position from e in left-to-right order using the linear preprocessing so that the LCA of two nodes of e is obtained in constant time. This construction is detailed in Proposition 4.4 of [7]. □

In the a-skeleton t_a, we equip each node n with three pointers: $\mathit{Witness}(n, a)$, $\mathit{FirstPos}(n, a)$, and $\mathit{Next}(n, a)$. For every node n in t_a,

- if n has color a then $\mathit{Witness}(n, a)$ is the witness for color a in n (and is undefined otherwise)
- $\mathit{FirstPos}(n, a)$ is the position p labeled a such that $p \in \mathit{First}(n)$ if it exists (and is undefined otherwise); note that property (P1) guarantees that there is at most one such position p
- $\mathit{Next}(n, a)$ is the set of all positions in $\mathit{FollowAfter}_e(n)$ labeled a.

The set $\mathit{FollowAfter}_e(n)$ is the extension of *Follow* to internal nodes n of e,

$$\mathit{FollowAfter}_e(n) = \{q \not\succcurlyeq n \mid \exists p \in \mathit{Last}(n), q \in \mathit{Follow}_e(p)\}.$$

Constructing the data structures *FirstPos* and *Witness* is straightforward: *Witness* is built simultaneously with the a-skeleton; *FirstPos* can for instance be computed in a single bottom-up traversal of each a-skeleton, using pointers *pSupFirst* from e and ancestor queries in e. Let n be the root node of the a-skeleton. Then $\mathit{BuildNext}(a, n, \varnothing)$ in Algorithm 1 builds the data structure $\mathit{Next}(n', a)$ for all nodes n' of the a-skeleton.

Lemma 3.2. *Calling $\mathit{BuildNext}(n, a, \varnothing)$ for each $a \in \Sigma$ and root node n of t_a takes in total time $O(|e|)$. If any call returns false then e is non-deterministic. Otherwise, the set $\mathit{Next}(n, a)$ defined during the execution consists of all positions in $\mathit{FollowAfter}_e(n)$ labeled a, for $n \in N_{t_a}$ and $a \in \Sigma$.*

Proof. The $O(|e|)$ time is achieved because (1) *BuildNext* is called at most m-times, where m is the number of nodes of all skeleta, and $m \in O(|e|)$ by Lemma 3.1, and (2) each line of the algorithm runs in constant time because $|Y| \leq 2$ at each call, due to Line 10. To see the correctness consider the execution along a path in t_a. If at Line 7 the current node n has an ancestor u labeled $*$ with no *SupLast*-node on their path, then Y contains $\mathit{FirstPos}(u, a)$; if n is in the left subtree of an ancestor u labeled $\odot$ with no *SupLast*-node on their path, and n has a right sibling n' in t_a, then Y contains $\mathit{FirstPos}(n', a)$. These conditions imply that the set defined in Line 7 holds all a-labeled positions in $\mathit{FollowAfter}_e(n)$. Clearly, e is non-deterministic if $|Y| > 2$ in Line 10. □

Algorithm 1 Computing $\mathit{Next}(n, a)$, if e is deterministic.

```
procedure BuildNext(a : Σ, n : Node, Y : Set(Node)) : Bool
 1  if $SupLast(n)$
 2      then $Y \leftarrow \varnothing$
 3  if $n$ is the left child in $t_a$ of a $\odot$-node and
 4     $n$ has a right sibling $n'$ in $t_a$ and
 5     $(\neg SupLast(n)$ or $parent_{t_a}(n) = parent_e(n))$
 6      then $Y \leftarrow Y \cup \{FirstPos(n', a)\}$
 7  $Next(n, a) \leftarrow \{p \in Y \mid n \not\preccurlyeq p\}$
 8  if $lab(n) = *$
 9      then $Y \leftarrow Y \cup \{FirstPos(n, a)\}$
10  if $|Y| > 2$
11      then return false
12  if $Lchild_{t_a}(n) = Null$
13      then return true
14      else $B \leftarrow BuildNext(a, Lchild_{t_a}(n), Y)$
15  if $Rchild_{t_a}(n) = Null$
16      then return $B$
17      else return $B \wedge BuildNext(a, Rchild_{t_a}(n), Y)$
end procedure
```

We define another condition:

(P2) for every $a \in \Sigma$ and $n \in N_{t_a}$, $\mathit{Next}(n, a)$ contains at most one element.

Clearly, (P2) can be tested in linear time (for instance by incorporating it into Algorithm 1). If (P2) is false, then e is non-deterministic. Thus, from now on we assume that both (P2) and (P1) are true. We identify $\mathit{Next}(n, a)$ with q if $\mathit{Next}(n, a) = \{q\}$, and let it be undefined otherwise.

Lemma 3.3. *Let $p, q \in \mathit{Pos}(e)$ with $\mathit{lab}_e(q) = a$. If $q \in \mathit{Follow}_e(p)$ then the lowest ancestor n of p having color a exists and satisfies $q = \mathit{Witness}(n, a)$ or $q = \mathit{FirstPos}(n, a)$ or $q \in \mathit{Next}(n, a)$.*

Proof. By Lemma 2.2, Lemma 2.5 (1), and Lemma 3.2: $q = \mathit{Witness}(n, a)$ if $\mathit{Rchild}(n) \preccurlyeq_e q$, $q = \mathit{FirstPos}(n, a)$ if $\mathit{Lchild}(n) \preccurlyeq_e q$, and $q = \mathit{Next}(n, a)$ if $n \not\preccurlyeq_e q$. □

From Lemma 3.3 and the definition of (P1) and (P2) we obtain the following result.

Lemma 3.4. *The expression e is non-deterministic iff* (P1) *or* (P2) *is false, or there exist $a \in \Sigma$, $n \in N_{t_a}$ of color a, and $q, q' \in \{\mathit{FirstPos}(n, a), \mathit{Witness}(n, a), \mathit{Next}(n, a)\}$ such that $q \neq q'$ and $\mathit{Follow}_e^{-1}(q) \cap \mathit{Follow}_e^{-1}(q') \neq \varnothing$.*

3.2 Determinism Testing Algorithm

To check determinism using Lemma 3.4 we need to check for $a \in \Sigma$ and $n \in N_{t_a}$ of color a, and for every pair of distinct positions q and q' in $\{\mathit{FirstPos}(n, a), \mathit{Witness}(n, a), \mathit{Next}(n, a)\}$ whether or not

$$\mathit{Follow}_e^{-1}(q) \cap \mathit{Follow}_e^{-1}(q') \neq \varnothing.$$

Three combinations can occur for a position p:

(1) $\mathit{Witness}(n, a)$ and $\mathit{Next}(n, a)$ follow p, or

(2) $\mathit{Witness}(n, a)$ and $\mathit{FirstPos}(n, a)$ follow p, or

(3) $\mathit{FirstPos}(n, a)$ and $\mathit{Next}(n, a)$ follow p.

The third combination, however, reduces to the other two and therefore needs not be considered: let F and N denote the nodes $\mathit{Next}(n, a)$ and $\mathit{FirstPos}(n, a)$, and let n_F and n_N denote the parent of their *SupFirst*-node. We can prove that either $n_F \preccurlyeq n_N \preccurlyeq n$, in which case $F = \mathit{FirstPos}(n_N, a)$ (and $N = \mathit{Witness}(n_N, a)$), or $n_N \preccurlyeq n_F \preccurlyeq n$, in which case N is one of $\mathit{FirstPos}(n_F, a)$ or $\mathit{Next}(n_N, a)$ (and $F = \mathit{Witness}(n_F, a)$).

To understand the first combination, consider the expression $e = (c(b?a?))a$, and let n be the parent of the c node in e. Thus, n is of color a, with the left a in e as witness. Clearly e is non-deterministic: take p as the c position, then both $\mathit{Witness}(n, a)$ and $\mathit{Next}(n, a)$ follow p. The same holds for the expressions $e' = (c(a?b?))a$ and $e'' = (c(b?a)^*)a$. However, expression $e''' = (c(b?a))a$ is deterministic; this is because n's right subtree is non-nullable, which prevents that $\mathit{Next}(n, a)$ and $\mathit{Witness}(n, a)$ both follow a same position p. It is not hard to see, and is formally shown in the proof of Theorem 3.5, that the first combination occurs if and only if the right-child of n is nullable.

Let us now consider combination (2). This combination can only occur if there is a $*$-node $S = \mathit{pStar}(n)$ above n, and $\mathit{pSupLast}(n)$ is above this node S. Let $e = (a(b?a))^*$ and let n be the parent of the first a-position. As we can see, this expression is deterministic. This is for a similar reason as before: because the right child of n is non-nullable. If we consider $e' = (a(b?a?))^*$ then this expression is indeed non-deterministic and it holds that both $\mathit{FirstPos}(n, a)$ and $\mathit{Witness}(n, a)$ follow position p, where p is for instance the b-position. Thus, combination (2) requires that the right child of n is nullable, and also that $\mathit{FirstPos}(S, a) = \mathit{FirstPos}(n, a)$. The latter guarantees that on the path from S to $\mathit{FirstPos}(n, a)$ there is nothing non-nullable "to the left", and hence, that $\mathit{FirstPos}(n, a)$ follows the same position p that $\mathit{Witness}(n, a)$ follows.

To check determinism of e we check (P1), (P2), and then we execute for every $a \in \Sigma$ and every node n with color a, $\mathit{CheckNode}(n, a)$ of Algorithm 2; if any call returns *false*, then e is non-deterministic.

THEOREM 3.5. *Determinism of a regular expression e can be decided in time $O(|e|)$.*

PROOF. Let S, W, N, and F denote the nodes $\mathit{pStar}(n)$, $\mathit{Witness}(n, a)$, $\mathit{Next}(n, a)$, and $\mathit{FirstPos}(n, a)$ respectively. Since (P1) and (P2) can be tested in $O(|e|)$ time, it suffices, by Lemma 3.4, to prove the following two statements.

(i) $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(N) \neq \varnothing$ iff $\mathit{Rchild}_e(n)$ is nullable and $N \neq \mathit{Null}$,

(ii) $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(F) \neq \varnothing$ iff $F \neq \mathit{Null}$, $S \neq \mathit{Null}$, $\mathit{Rchild}_e(n)$ is nullable, $\mathit{FirstPos}(S, a) = F$, and $\mathit{pSupLast}(n) \preccurlyeq S$.

Let us prove statement (i) first. If $N \neq \mathit{Null}$ and $\mathit{Rchild}_e(n)$ is nullable then $\mathit{Lchild}_e(n)$ is not a *SupLast*-node. Therefore any position in $\mathit{Last}(\mathit{Lchild}_e(n))$ belongs to $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(N)$. For the only-if direction, let q be a position in $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(N)$. Then in particular $N \neq \mathit{Null}$. Node n is a strict ancestor of q since $q \in \mathit{Follow}_e^{-1}(W)$ and $n = \mathit{parent}_e(\mathit{pSupFirst}(W))$. As q belongs to $\mathit{Follow}_e^{-1}(N)$, $\mathit{pSupLast}(q)$ is an ancestor of n. This implies that $\mathit{Rchild}(n)$ is nullable according to Lemma 2.6, since $\mathit{Rchild}(n) = \mathit{pSupFirst}(W)$ and W follows q.

Proof of (ii): If $F \neq \mathit{Null}$, $S \neq \mathit{Null}$, $\mathit{Rchild}_e(n)$ is nullable, $\mathit{FirstPos}(S, a) = F$, and $\mathit{pSupLast}(n) \preccurlyeq S$, then any q in $\mathit{Last}(\mathit{Lchild}(n))$ is in $(\mathit{Follow}_e^{\odot})(W) \cap (\mathit{Follow}_e^{*})^{-1}(F)$. Conversely, let q be a position in $\mathit{Follow}_e^{-1}(W) \cap \mathit{Follow}_e^{-1}(F)$. As q belongs to $\mathit{Follow}_e^{-1}(W)$, node n is a strict ancestor of q. If $\mathit{Rchild}_e(n) \preccurlyeq_e q$ then $q \in (\mathit{Follow}_e^{*})^{-1}(F)$, hence $\mathit{FirstPos}(S, a) = F$ and $\mathit{pSupLast}(n) \preccurlyeq S$, and furthermore $\mathit{pSupLast}(q) \preccurlyeq S$, so that $\mathit{Rchild}_e(n)$ is nullable according to Lemma 2.6. Assume now that $\mathit{Lchild}_e(n)$ is an ancestor of q, and let $x = \mathit{LCA}(q, F)$. As an ancestor of both q and F, $\mathit{Lchild}_e(n)$ is an ancestor of x. Furthermore, there is no *SupLast*-node between q and $\mathit{Lchild}_e(n)$, except possibly $\mathit{Lchild}_e(n)$, and there is no *SupFirst*-node between F and $\mathit{Lchild}_e(n)$. Consequently, x is non-nullable because $\mathit{Lchild}_e(n)$ is, and, there is no $*$-labeled node between x and $\mathit{Lchild}_e(n)$. Hence $q \notin (\mathit{Follow}_e^{\odot})^{-1}(F)$, and, more generally, $\mathit{Follow}_e^{-1}(W) \cap (\mathit{Follow}_e^{\odot})^{-1}(F)$ is empty. This means that $q \in (\mathit{Follow}_e^{*})^{-1}(F)$. Thus $S = \mathit{pStar}(x)$ is not *Null*, satisfies $\mathit{FirstPos}(S, a) = F$, and is an ancestor of n since there is no $*$-labeled nodes between x and $\mathit{Lchild}_e(n)$. Accordingly, $\mathit{pSupLast}(q) \preccurlyeq S$ and hence $\mathit{Rchild}_e(n)$ is non-nullable. □

Algorithm 2 Checking determinism.

```
procedure CheckNode($n$ : Node, $a$ : $\Sigma$) : Bool
  $F \leftarrow \mathit{FirstPos}(n, a)$
  $S \leftarrow \mathit{pStar}(n)$
  if $\mathit{Rchild}_e(n)$ is nullable and
     ($\mathit{Next}(n, a) \neq \mathit{Null}$ or
      ($\mathit{FirstPos}(S, a) = F$ and $\mathit{pSupLast}(n) \preccurlyeq S$))
       then return false
  return true
end procedure
```

3.3 Testing Numeric Occurrences

Regular expression occurring in XML Schema may contain numeric occurrence indicators. Following the definitions in [19], regular expressions with numeric occurrence indicators extend regular expressions with $e^{i..j}$ where $i \in \mathbb{N}$, $j \in \mathbb{N} \cup \{\infty\}$, and $i \leq j$. The expression $e^{i..j}$ denotes the union of $L(\underbrace{e \odot e \cdots \odot e}_{k\text{-times}})$ for $i \leq k \leq j$. Also e^i denotes $e^{i..i}$. The definition of determinism in presence of numeric occurrence indicators must take into accounts the iterations. Informally, e is deterministic if for every word w there exists at most one position that can be reached after reading w. For instance, $e = (ab)^{2..2}a(b + d)$ is deterministic, but $e' = (ab)^{1..2}a$ is not, because $w = aba$ can lead to two a-labeled positions in e'. We refer the reader to [19] for the notion of determinism in regular expressions with numeric occurrence indicators. Note that nested iterations can interact with each other: consider the expression $e_5 = ((a^{2..3} + b)^2)^2 b$ from [19]. This expression is non-deterministic because word $w = a^8 b$ can lead to the two b-labeled positions: to the first one if we decompose it into $(a^3)^2 a^2 b$, and to the second one with decomposition $((a^2)^2)^2 b$.

In order to deal with those interactions between iterations, Kilpeläinen and Tukhanen [19] define the *flexibility* of f in e, for every subexpression f of e. They explain how to annotate, in time $O(|e|)$, every node n of e with a Boolean value indicating the flexibility of n. Essentially, flexible iterations are the only ones we have to consider when assessing determinism (in particular $*$ expressions are flexible). The authors give a (more accurate) characterization for determinism of numeric occurrences as Theorem 5.5 in [19]. This characterization can be verified in linear time using a case study similar to the one above (but with flexible iterations instead of $*$ expressions). Therefore, given a regular expression e with numeric occurrence indicators we can decide in time $O(|e|)$ whether or not e is deterministic.

This improves upon the complexity $O(\sigma|e|)$ from [18], where $\sigma = |\Sigma|$. Actually, in Theorem 3.3 from [18], the complexity is stated as $n^2/(\log(n))$, with n representing the size of the binary representation of the regular expression. But with our notations, this translates into a quadratic $O(\sigma|e|)$. Kilpelaïnen obtains this complexity by a merging-based examination of *First* and *Follow* sets, similar to the approach in [19], but relying on a more careful analysis of the *Follow* sets. Interestingly, he observes after his Theorem 3.3 that it seems difficult to go below $O(\sigma n)$ using his approach. We believe that our skeleton-based algorithm offers a good solution to the limitations of the merging-based approach.

3.4 Alternative Determinism Test

Determinism of e can be formulated as follows:

$$\neg(\exists p, p_1, p_2 \in \mathit{Pos}(e).\ \mathit{lab}_e(p_1) = \mathit{lab}_e(p_2) \wedge \\ p_1 \in \mathit{Follow}_e(p) \wedge p_2 \in \mathit{Follow}_e(p)).$$

A natural question arises: Is there a logic that allows to capture determinism, and at the same time, has efficient model checking that yields a procedure for checking determinism in linear time? The answer is positive: It is possible with $\mathcal{X}_{reg}^{=}$, the language of Regular XPath expressions with data equality tests for binary trees with data values as defined in [7].

Trees with data values allow to store with every node its label, drawn form a finite set, and additionally, a data value, drawn form an infinite set. Regular XPath allows to navigate the nodes of the tree using regular expressions of simple steps (e.g., parent to the left child) and filter expressions. Filter expressions with data equality allow essentially to test whether two nodes have the same data value. In [7] Bojańczyk and Parys show that an $\mathcal{X}_{reg}^{=}$-expression φ can be evaluated over a tree t in time $2^{O(|\varphi|)}|t|$.

We wish to construct an $\mathcal{X}_{reg}^{=}$-expression φ_{det} that captures determinism and whose size is constant i.e., does not depend on the regular expression e. The main challenge is to handle position labels of e that can be drawn from an alphabet of arbitrary size. This is accomplished by: 1) storing the labels of positions of e as data values and 2) using data equality to check whether two positions have the same label.

Theorem 3.6. *There exists an $\mathcal{X}_{reg}^{=}$-expression φ_{det} such that for any regular expression e, φ_{det} is satisfied in e if and only if e is deterministic.*

Proof. We present only the construction of φ_{det}. Let *SupFirst* and *SupLast* denote $\mathcal{X}_{reg}^{=}$-expressions that are satisfied only in *SupFirst*- and *SupLast*-nodes, respectively.

$$D = (\text{child}/[\text{not } \mathit{SupFirst}])^*/P \qquad P = [\text{not child}]$$
$$U = ([\text{not } \mathit{SupLast}]/\text{parent})^* \quad F = ([\text{lab}() = \odot])/\text{to-right}/D$$
$$\varphi_{\odot\odot} = \text{child}^*/[\text{not } \mathit{SupLast}]/\text{from-left}/[F = (U/\text{from-left}/F)]$$
$$\varphi_{**} = \text{child}^*/[\text{lab}() = *]/ \\ [D = (U/[\mathit{SupFirst}]/\text{parent}/U/[\text{lab}() = *]/D)]$$
$$\varphi_{\odot *} = \text{child}^*/[\text{not } \mathit{SupLast}]/\text{from-left}/ \\ [(\text{to-right}/[\mathit{SupFirst}]/D) = (\text{parent}/U/[\text{lab}() = *]/D)] \\ \cup\, \text{child}^*/[\text{lab}() = *]/[D = (U/\text{from-left}/F)]$$
$$\varphi_{P_1} = \text{child}^*/[(\text{to-left}/[\text{not } \mathit{SupFirst}]/D) = \\ (\text{to-right}/[\text{not } \mathit{SupFirst}]/D)]$$
$$\varphi_{det} = [\text{not } (\varphi_{P_1} \text{ or } \varphi_{\odot\odot} \text{ or } \varphi_{\odot *} \text{ or } \varphi_{*\odot} \text{ or } \varphi_{**})].$$

Basically, φ_{P_1} checks if (P1) is violated in e and the expression $\varphi_{\ell\ell'}$ for $\{\ell, \ell'\} \subseteq \{*, \odot\}$ checks whether there exist two distinct positions p_1 and p_2 of e such that $\mathit{lab}(p_1) = \mathit{lab}(p_2)$ and $(\mathit{Follow}_e^{\ell})^{-1}(p_1) \cap (\mathit{Follow}_e^{\ell'})^{-1}(p_2) \neq \varnothing$. □

4. MATCHING

In this section we present a collection of algorithms matching a word w against e. First, we present an algorithm for arbitrary deterministic regular expressions that uses the constructions from Section 3 and lowest color ancestor queries to achieve expected time complexity $O(|e| + |w| \log\log|e|)$. Next, we present a matching algorithm for k-occurrence regular expressions in time $O(|e| + k|w|)$, which is linear if k is a constant. The most intricate matching algorithm that we present in this paper is the path-decomposition algorithm. It works in time $O(|e| + c_e|w|)$, where c_e is the maximal depth of alternating union and concatenation operators in e. The three algorithms above perform matching by providing a *transition simulation procedure*: given a position p and a symbol a return the position q labeled a that follows p, or *Null* if no such position exists. If $e = (\#e')\$$, matching a word w against e' is straightforward: begin with position #, use the transition simulation procedure iteratively on subsequent symbols of w, and finally test if the position obtained after processing the last symbol of w is followed by \$.

The algorithms above allow to match multiple input words $w_1, \ldots, w_N$ against one regular expression e: the corresponding running times are obtained by replacing the factor $|w|$ by $|w_1| + \ldots + |w_N|$. We also present an algorithm that runs in time $O(|e| + |w_1| + \ldots + |w_N|)$ for star-free deterministic regular expressions e, a setting in which none of the previously mentioned algorithms guarantee linear complexity.

In the reminder of this section, we fix a deterministic regular expression e, and when talking about positions and nodes, we implicitly mean positions and nodes of e.

4.1 Lowest Colored Ancestor Algorithm

Our previous construction that tests determinism in linear time, provides an efficient procedure for transition simulation. Recall that we color the parent of any *SupFirst*-node n with the labels of the positions that belong to *First*(n). By Lemma 3.3, given a position p and a symbol a the a-labeled position q that follows p is one of: *Witness*(n, a), *FirstPos*(n, a), and *Next*(n, a), where n is the lowest ancestor of p with color a. We use the *checkIfFollow* test (Theorem 2.4) to select the correct following position q among the three candidates.

Example 4.1. *Consider the expression in Figure 1, position p_3, and the symbol c. The lowest ancestor of p_3 with color c is n_3. Here, $\mathit{Witness}(n_3, c) = p_5$, $\mathit{Next}(n_3, c) = p_1$, and $\mathit{FirstPos}(n_3, c) = \mathit{Null}$. Using checkIfFollow we find that it is p_5 that follows p_3. This ends the transition simulation procedure. Now, at position p_5 we read the next symbol a. The lowest ancestor of p_5 with color a is again n_3. This time it is $\mathit{FirstPos}(n_3, a) = p_2$ that follows p_5.*

The basic ingredient of this procedure is an efficient algorithm for answering lowest colored ancestor queries. Recall from [23, 10], that given a tree t with colors assigned to its nodes (some nodes possibly having multiple colors), we can preprocess t in expected time $O(|t| + C)$, where C is the total number of color assignments, so that any lowest colored ancestor query is answered in time $O(\log\log |t|)$. In this way, the transition simulation is accomplished in time $O(\log\log |e|)$, which gives us the following result.

Theorem 4.2. *For any deterministic regular expression e, after preprocessing in expected time $O(|e|)$, we can decide for any word w whether $w \in L(e)$ in time $O(|w| \log\log |e|)$.*

4.2 Bounded Occurrence Algorithm

A regular expressions e is called *k-occurrence* (k-ORE for short) if each symbol $a \in \Sigma$ occurs at most k times in e. While every regular expression is k-ORE for a sufficiently large k, Bex et al. [5] report that the majority of regular expressions in real-life XML schemas are in fact 1-OREs. Given a position p and a symbol a, to find the following a-labeled position q we only need to perform the *checkIfFollow* test (Theorem 2.4) on all a-labeled positions in e, which are gathered into a designated list during preprocessing of e. Thus, transition simulation is performed in time $O(k)$.

Theorem 4.3. *For any deterministic k-ORE e, after preprocessing in time $O(|e|)$, we can decide for any word w whether $w \in L(e)$ in time $O(k|w|)$.*

We note that an analogous technique can be used to match a word w against a nondeterministic k-ORE e: we maintain a set P of at most k positions and when reading symbol a we identify among the a-labeled positions those that follow any of the positions in P. Here, reading one symbol requires $O(k^2)$ time, and thus, the matching can be done in time $O(k^2|w|)$ after $O(|e|)$ preprocessing.

4.3 Path Decomposition Algorithm

Next, we describe an algorithm for matching a word w against a regular expression e in time $O(|e|+c_e|w|)$, where c_e is the maximal depth of alternating union and concatenation operators in e (as mentioned at the end of the Introduction, c_e is bounded by 4 in real-life DTDs [13]).

First, we define the function $h_{\mathit{First}}(n, a)$ that for a node n and a symbol a returns the unique a-labeled position in $\mathit{First}(n)$ and Null if it does not exist. Queries of the form $h_{\mathit{First}}(n, a)$ can be answered in constant time after preprocessing in time $O(|e|)$, but since h_{First} is not used in the final algorithm, we omit the implementation details.

Climbing algorithm. We first present a simple transition simulation procedure that uses h_{First}, and later improve it to obtain the desired evaluation algorithm. Given a position p and a symbol a, it suffices to find an ancestor n of p such that $q = h_{\mathit{First}}(\mathit{Rchild}(n), a)$ follows p (tested with *checkIfFollow*). If such ancestor does not exists, then p has no a-labeled following position. The soundness of this procedure follows from that of *checkIfFollow* and the completeness from Lemma 2.5. A naïve implementation seeks the ancestor in question by climbing up the parse tree starting from p, which yields $O(\mathit{depth}(e))$ time per transition simulation and overall $O(|e| + \mathit{depth}(e) \cdot |w|)$ time for matching.

Path decomposition. Our algorithm speeds up climbing the path using jumps that follow precomputed pointers. The precomputed pointers lead to nodes where we store an aggregation of the values of h_{First} for several nodes skipped during the jump. The pointers are defined using the notion of path decomposition of the parse tree.

Recall that a *path decomposition* of a tree is a set of pairwise disjoint paths covering all nodes of the tree, and here, a path means a sequence of nodes $n_1, \ldots, n_k$ such that n_i is the parent of n_{i+1}. Note that a path decomposition of a tree can be specified by the set of the top-most nodes of the paths, which is how we define the path decomposition of e. A node y of e is the *top-most node* of a path if it is the root of e, or satisfies one of the following conditions:

(i) $\mathit{SupLast}(y)$

(ii) $\mathit{SupFirst}(y)$

(iii) y is the nullable right child of its parent, or

(iv) y is the right child of a $+$-labeled node.

For a position p we define $\mathit{top}(p)$ as the top-most node of the path of the left sibling of $\mathit{pSupFirst}(p)$.

Example 4.4. *Consider the regular expression presented in Figure 2 together with its path decomposition. For this expression $c_e = 4$ because there are at most 4 alternations of union and concatenation operators on any path of the expression, and in particular, it is 4 on the path from p_1 to the root node. Note that $\mathit{top}(p_1) = n_3$ and $\mathit{top}(p_2) = n_1$.*

We now define the function h which is similar to h_{First} but defined for top-most nodes only: $h(n, a)$ points to the a-labeled position p such that $n = \mathit{top}(p)$, i.e., we assign $h(\mathit{top}(p), \mathit{lab}(p)) = p$ for every position p. For instance, in the expression in Figure 2, $h(n_3, a) = p_1$ and $h(n_1, d) = p_2$.

There exists a subtle connection between h and h_{First}. If we consider a top-most node n, then the values of h assigned to n can be viewed as an aggregation of values of h_{First} of several nodes $n_1, \ldots, n_k$, which are gathered from around the path (but not from the path). The decomposition of e ensures that the aggregation is collision-free, i.e., if $h_{\mathit{First}}(n_i, a) \neq \mathit{Null}$ for some i, then $h_{\mathit{First}}(n_j, a) = \mathit{Null}$ for all $j \neq i$. Formally, we state this property as follows.

Lemma 4.5. *For any two different positions p and p', if $\mathit{top}(p) = \mathit{top}(p')$, then p and p' have different labels.*

Proof. Let y denote the lowest node in the path of $\mathit{top}(p)$ and let p_0 denote some position in $\mathit{Last}(y)$. We show that p follows p_0. By definition of $\mathit{top}(p)$, the left sibling of $\mathit{pSupFirst}(p)$ is on the path between y and $\mathit{top}(p)$. Therefore, $\mathit{pSupLast}(p_0) = \mathit{pSupLast}(y)$ is an ancestor of the left sibling of $\mathit{pSupFirst}(p)$ because there is no $\mathit{SupLast}$-node on a path except for the top-most node of the path. Moreover, we observe that the parent of $\mathit{pSupFirst}(p)$ is labeled with $\odot$. Thus, by Lemma 2.2 we get $p \in \mathit{Follow}(p_0)$. Similarly,

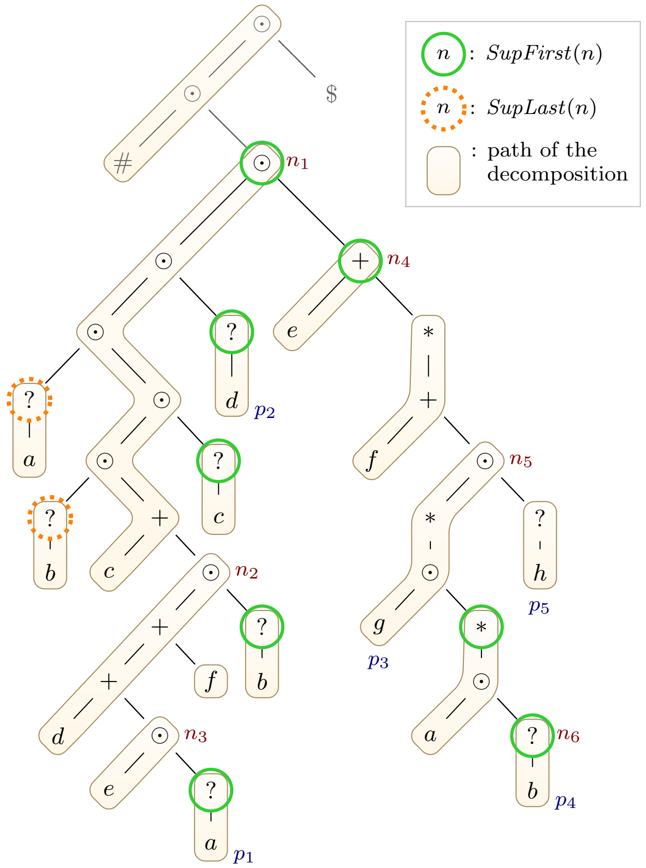
n: $\mathit{SupFirst}(n)$
n: $\mathit{SupLast}(n)$
: path of the decomposition
n_1
n_2
n_3
n_4
n_5
n_6
p_1
p_2
p_3
p_4
p_5
#
$

Figure 2: Path decomposition.

we show that $p' \in \mathit{Follow}(p_0)$. Because e is deterministic, there cannot be two different positions with the same label in $\mathit{Follow}(p_0)$. □

Lazy arrays. To store the values of h we use *lazy arrays*, which we describe in detail next. This interesting data structure, known in programmer's circles [17, 22], provides the functionality of an associative array with constant time initialization, assignment, and lookup operations. The finite set of keys K needs to be known prior to initialization of the data structure. Furthermore, every key needs to be associated with a unique element from a continuous fragment of natural numbers, and here for simplicity, we assume that $K = \{1, \ldots, N\}$ for some $N \geq 1$.

A lazy array consists of an array A that stores the values associated with the keys, a counter C of active keys having a value assigned, and additionally two arrays B and F that store the set of active keys. At initialization, C is set to 0 and uninitialized memory of length N is allocated for each of the arrays A, F, and B (an operation assumed to work in $O(1)$ time). To *assign* value v to key k, we add k to the set of active keys (if k is not in that set already), and assign $A[k] = v$. To *lookup* key k, we return $A[k]$ if k is active and return *Null* otherwise. To add a key k to the set of active keys, we increment C, set $F[C] = k$, and set $B[k] = C$. In this way a key k is active if and only if $1 \leq B[k] \leq C$ and $F[B[k]] = k$. Note that the first condition alone is insufficient to check if a key k is active because B has been allocated with uninitialized memory.

We found out that in practice, hash arrays offer compatible functionality with superior performance while theoretically providing only expected $O(1)$ time for the assignment and lookup operations. As a side note, we point out that lazy arrays stand on their own merit because they allow a constant time reset operation (by simply setting $C = 0$), unmatched by hash arrays (but not needed by our algorithm).

Preprocessing. We construct and fill the lazy-array h in one bottom-up traversal of e. In the same traversal we also compute an additional pointer *nexttop* for every position and every top-most node of a path, defined as follows. We set $\mathit{nexttop}(n)$ to the lowest top-most node y of a path above $\mathit{parent}(n)$ that is either the root of e, or satisfies one of the following conditions:

(1) $\mathit{SupLast}(y)$

(2) $\mathit{SupFirst}(y)$

(3) there exists a non-nullable $\odot$-labeled ancestor of n in the path of y.

For instance, in the expression in Figure 2, $\mathit{nexttop}(p_3) = n_5$, $\mathit{nexttop}(p_4) = n_6$, and $\mathit{nexttop}(p_5) = n_4$. We point out that $\mathit{nexttop}(n)$ is always the top-most node of some path, and furthermore, $\mathit{nexttop}(n)$ is a strict ancestor of n.

Transition simulation. *FindNext* in Algorithm 3 follows *nexttop* pointers on the path from p to the node $\mathit{pSupLast}(p)$ while attempting to find a-labeled follow positions stored in h at the visited nodes. If this does not succeed, then *FindNext* checks in $\mathit{First}(\mathit{parent}(\mathit{pSupLast}(p)))$ (Lines 8–14) to find follow positions. This task would be easy to accomplish with h_{First} through $h_{\mathit{First}}(\mathit{parent}(\mathit{pSupLast}(p)), a)$. Since we wish to use h instead, we need to locate the node n such that $h(n, a)$ returns the position we look for. The location of this node depends on whether or not the node $y = \mathit{parent}(\mathit{pSupLast}(p))$ is nullable. If y is nullable, we perform a single *nexttop* jump from y to reach n. Otherwise, n is the left sibling of y. Finally, we remark that if $h_{\mathit{First}}(\mathit{parent}(\mathit{pSupLast}(p)), a)$ is not *Null*, then $h(n, a)$ returns the same node but the converse needs not be true: even if $q = h(n, a)$ is not *Null*, $h_{\mathit{First}}(\mathit{parent}(\mathit{pSupLast}(p)), a)$

Algorithm 3 Transition simulation.

```
procedure $\mathit{FindNext}(p : \text{Position},\ a : \Sigma) : \text{Position}$
 1  $x \leftarrow p$
 2  while $\mathit{pSupLast}(p) \neq x$
 3      if $\mathit{checkIfFollow}(h(x, a), p)$
 4          then return $h(x, a)$
 5      $x \leftarrow \mathit{nexttop}(x)$
 6  if $\mathit{checkIfFollow}(h(x, a), p)$
 7      then return $h(x, a)$
 8  $y \leftarrow \mathit{pSupFirst}(\mathit{parent}(x))$
 9  if $y$ is nullable
10      then $q \leftarrow h(\mathit{nexttop}(y), a)$
11      else $q \leftarrow h(\mathit{Lchild}(\mathit{parent}(y)), a)$
12  if $\mathit{checkIfFollow}(q, p)$
13      then return $q$
14      else return $\mathit{Null}$
end procedure
```

might be *Null*. Consequently, we verify in Line 12 that the node q indeed follows p.

Example 4.6. *Consider expression in Figure 2, position p_1, and symbol d. The computation of $\mathit{FindNext}(p_1, d)$ follows the jump sequence: $p_1, \mathit{parent}(p_1), n_3, n_2, n_1$. At node n_1, $h(n_1, d)$ yields position p_2, and since p_2 follows p_1, the procedure returns p_2.*

Correctness. To reason about iterations of the main loop of *FindNext*, we introduce this notation: $\mathit{nexttop}^0(n) = n$, and $\mathit{nexttop}^{i+1}(n) = \mathit{nexttop}(\mathit{nexttop}^i(n))$ for $i \geq 0$. Also, the *jump sequence* of p is the sequence

$$\mathit{nexttop}^0(p), \mathit{nexttop}^1(p), \ldots, \mathit{nexttop}^K(p),$$

where K is such that $\mathit{nexttop}^K(p) = \mathit{pSupLast}(p)$. We call K the *length* of the jump sequence of p. We first show that the main loop performs a sufficient number of *nexttop* jumps.

Lemma 4.7. *Let p be a position and K the length of the jump sequence of p. For every position q that follows p, either $\mathit{top}(q) = \mathit{nexttop}^i(p)$ for some $0 \leq i \leq K$ or q belongs to $\mathit{First}(\mathit{parent}(\mathit{pSupLast}(p)))$.*

Proof. By Lemma 2.5 $\mathit{top}(q)$ is an ancestor of p or the left sibling of a non-nullable *SupFirst*-ancestor of p. Furthermore, if $\mathit{pSupFirst}(q)$ is nullable then $\mathit{top}(q)$ is the top of the path containing $\mathit{parent}(\mathit{pSupFirst}(q))$. From the definition of *top* and *nexttop*, the jump sequence of p visits every *SupFirst*- and *SupLast*-ancestor of p, as well as every ancestor y of p such that y is top-most node of a path and there exists some non-nullable $\odot$-labeled ancestor of p on that path.

We assume that $q \notin \mathit{First}(\mathit{parent}(\mathit{pSupLast}(p)))$, and show that in this case no other ancestor of q needs to be visited. Under that assumption, a case analysis for Lemma 2.2 shows that $\mathit{pSupFirst}(q)$ is the right sibling of $\mathit{pSupLast}(p)$, or satisfies $\mathit{pSupLast}(p) \preccurlyeq \mathit{parent}(\mathit{pSupFirst}(q)) \preccurlyeq p$. If $\mathit{pSupFirst}(q)$ is the right sibling of $\mathit{pSupLast}(p)$, then $\mathit{top}(q)$ is equal to $\mathit{pSupLast}(p)$ and is therefore visited by the jump sequence. Otherwise, $\mathit{pSupLast}(p) \preccurlyeq \mathit{parent}(\mathit{pSupFirst}(q))$. By Lemma 2.6, $\mathit{pSupFirst}(q)$ is nullable. Consequently, its parent belongs to the path of $\mathit{top}(q)$. Furthermore, the left sibling of $\mathit{pSupFirst}(q)$, and therefore its parent, are non-nullable. It follows that the parent of $\mathit{pSupFirst}(q)$ is a non-nullable $\odot$-labeled ancestor of p that belongs to the path of $\mathit{top}(q)$ which is thus visited. □

We now show the correctness of *FindNext*.

Lemma 4.8. *For any position p and any symbol a, the procedure $\mathit{FindNext}(p, a)$ returns q iff $q \in \mathit{Follow}(p)$ and $\mathit{lab}(q) = a$.*

Proof. The soundness of *FindNext* follows from the use of *checkIfFollow* prior to returning a position. If a position q is labeled with a and follows p, then *FindNext* returns q by Lemma 4.7 and because the algorithm returns q at Line 13 if q belongs to $\mathit{First}(\mathit{parent}(\mathit{pSupLast}(p)))$. □

Complexity. We show that the amortized running time of the transition simulation procedure in Algorihtm 3, when matching a word w against the deterministic regular expression e, is proportional to c_e, the maximal depth of alternating union and concatenation operators in e.

Lemma 4.9. *Procedure $\mathit{FindNext}(p, a)$ works in amortized time $O(c_e)$, when matching a word against a deterministic regular expression e.*

Proof. We use the potential *pot* of the data structure defined as a function of the current position:

$$\mathit{pot}(p) = |\{v \preccurlyeq p \mid \mathit{SupFirst}(v)\}|.$$

At the phantom position #, the initial potential is set to zero. The potential is decreased by one with every other jump through *nexttop* and is increased by at most one each time the transition simulation procedure is executed.

Now, let q be the position returned by $\mathit{FindNext}(p, q)$, i.e., the a-labeled position that follows p in e. We prove that $\mathit{FindNext}(p, a)$ executes at most $2(\mathit{pot}(q) - \mathit{pot}(p)) + c_e + O(1)$ iterations of the loop (*nexttop* jumps) before returning q.

By definition of *top*, there are no *SupFirst*-nodes between $\mathit{pSupFirst}(q)$ and $\mathit{top}(q)$, hence

$$\mathit{pot}(q) \leq \mathit{pot}(\mathit{top}(q)) + 1. \tag{1}$$

Let K be the length of the jump sequence of p and let $n_i = \mathit{nexttop}^i(p)$ for $0 \leq i \leq K$. Now, from the sequence $n_0, \ldots, n_K$ we remove every node that is the non-nullable right child of a $+$-labeled node and obtain a subsequence $n_{i_0}, n_{i_1}, \ldots, n_{i_{K'}}$. For every $0 \leq j < K' - 1$, if n_{i_j} is the nullable right child of its parent, then $n_{i_{j+1}}$ is a *SupFirst*-node by definition of *nexttop*. Hence, for every $0 \leq j \leq K'$,

$$j \leq 2(\mathit{pot}(n_{i_0}) - \mathit{pot}(n_{i_j})) + 2.$$

Thus, for every $0 \leq j \leq K$,

$$j \leq 2(\mathit{pot}(p) - \mathit{pot}(n_j)) + 2 + K - K'. \tag{2}$$

Let ℓ be the natural number such that $n_\ell = \mathit{top}(q)$. Combining equations (1) and (2), as c_e is an upper bound for $K - K'$, we obtain the result claimed before:

$$\ell \leq 2(\mathit{pot}(p) - \mathit{pot}(q)) + 4 + c_e. \tag{3}$$

From this result, establishing the amortized complexity is straightforward. Given a word $w = a_1 \cdots a_n$, let $p_1, \ldots, p_n$ be the sequence of positions with $p_i = \mathit{FindNext}(p_{i-1}, a_i)$ for $1 \leq i \leq n$ and $p_0 = \#$. Then, the number of iterations through the loop of *FindNext* while matching w against e is at most:

$$\begin{aligned} & n(4 + c_e) + 2\sum_{i=1}^{n}(\mathit{pot}(p_{i-1}) - \mathit{pot}(p_i)) \\ & = n(4 + c_e) + 2(\mathit{pot}(p_0) - \mathit{pot}(p_n)) \\ & \leq n(4 + c_e). \end{aligned}$$

This implies the amortized cost of $O(c_e)$, because each line of *FindNext* runs in constant time. □

Note that in the previous proof it suffices to take a smaller value of c_e, the maximum number of ancestors of a position of e that are labeled with $+$, are non-nullable, and have a parent labeled with $\odot$.

Finally, we formally state the result.

Theorem 4.10. *For any deterministic regular expression e, after preprocessing in time $O(|e|)$, we can decide for any word w whether $w \in L(e)$ in time $O(c_e|w|)$, where c_e is the maximal depth of alternating union and concatenation operators in e.*

4.4 Star-Free Algorithm

Finally, we present an algorithm that matches simultaneously several words $w_1, \dots, w_N$ against a star-free deterministic regular expression e. For a single word this is trivial: in a star-free regular expression, $q \in \mathit{Follow}(p)$ implies that position q is after p in the preorder traversal of e, and therefore, to simulate a transition it suffices to run the *checkIfFollow* test on subsequent positions until a match is found. In fact, the *checkIfFollow* tests can be hard-coded into the traversal to avoid lowest common ancestor queries.

The result is non-trivial when matching several words $w_1, \dots, w_N$. Also this time, the expression is traversed only once and for every word w_i we maintain the current index d_i, indicating the prefix of w_i matched so far. The matching is driven by the preorder traversal of e: with every position visited in the traversal we update the indices $d_1, \dots, d_N$ accordingly. The update process is, however, not straightforward and to perform it efficiently we use a variant of the a-skeleta, constructed dynamically.

First, we define some terminology. We say that the word w_i at index d_i *expects* the symbol a if the symbol of w_i at index $d_i + 1$ is a. We also say that w_i at d_i *reaches* position p if after simulating transitions on the corresponding prefix of w_i we arrive at p (or more precisely, the Glushkov automaton of e reaches p after reading the prefix of w_i). A *dynamic a-skeleton* t_a is essentially a structure containing a subset of positions closed under lowest common ancestors. Additionally, with each position p in t_a we associate a list of (pointers to) words such that if word w_i is associated with p, then the word w_i at index d_i reaches the position p and expects the symbol a.

When processing a position p labeled a, we remove from the dynamic a-skeleton t_a every position q that is followed by p, update indices of the words on the list associated with q, and insert p to some dynamic a-skeleta accordingly. We illustrate the procedure in the following example.

EXAMPLE 4.11. *We consider the deterministic regular expression* $e = (\#(((a + ba)(c?))(d?b)))\$$*, where* $\#$ *and* $\$$ *are two phantom positions that do not need to be matched. The expression* e *has 8 positions:* $\#, p_1, \dots, p_6, \$$. *We match against* e *the words* $w_1 = bcdb$, $w_2 = acdba$, $w_3 = acb$, *and* $w_4 = bada$.

Initially, all indices are $d_1 = d_2 = d_3 = d_4 = 0$. *When describing dynamic a-skeleta, we write* $\langle p, W\rangle$ *to indicate that a position* p *has an associated list of words* W. *Initially,* $t_a = \langle \#, [w_2, w_3]\rangle$, $t_b = \langle \#, [w_1, w_4]\rangle$, *and all other dynamic a-skeleta are empty.*

In the first step, we read the a-labeled position p_1. *Because* p_1 *follows* $\#$, *we remove from* t_a *the position* $\langle \#, [w_2, w_3]\rangle$, *increment* d_2 *and* d_3, *and insert* $\langle p_1, [w_2, w_3]\rangle$ *to* t_c.

Next, we read the b-labeled position p_2. *Because* p_2 *follows* $\#$, *we remove from* t_b *the position* $\langle \#, [w_1, w_4]\rangle$, *increment* d_1 *and* d_4, *and insert* $\langle p_2, [w_4]\rangle$ *to* t_a *and* $\langle p_2, [w_1]\rangle$ *to* t_c. *Because we keep the dynamic a-skeleta closed under lowest common ancestors,* t_c *becomes* $\langle p_1, [w_2, w_3]\rangle + \langle p_2, [w_1]\rangle$, *where* $+$ *is a binary node whose children are* p_1 *and* p_2.

At the position p_3 *labeled with a, because* p_3 *follows* p_2, *we remove* $\langle p_2, [w_4]\rangle$ *from* t_a, *increment* d_4 *and add* $\langle p_3, [w_4]\rangle$ *to* t_d. *At the position* p_4 *labeled with c, because* p_4 *follows* p_1, *we remove from* t_c *the position* $\langle p_1, [w_2, w_3]\rangle$, *increment* d_2 *and* d_3, *and insert* $\langle p_4, [w_2]\rangle$ *to* t_d *and* $\langle p_4, [w_3]\rangle$ *to* t_b. *Although* p_2 *is not followed by* p_4, *we also remove* $\langle p_2, [w_1]\rangle$ *from* t_c *and discard it because we observe that* p_2 *will not be followed by any of the subsequent positions. After this step,* $t_b = \langle p_4, [w_3]\rangle$, t_c *is empty, and* $t_d = \langle p_3, [w_4]\rangle \odot \langle p_4, [w_2]\rangle$.

The next position p_5 *is labeled with d and follows both* p_3 *and* p_4. *Therefore, we remove from* t_d *both* $\langle p_3, [w_4]\rangle$ *and* $\langle p_4, [w_2]\rangle$, *increment* d_2 *and* d_4, *and insert* $\langle p_5, [w_4]\rangle$ *to* t_a *and* $\langle p_5, [w_2]\rangle$ *to* t_b. *This way,* t_b *is* $\langle p_4, [w_3]\rangle \odot \langle p_5, [w_2]\rangle$.

In the last step we move to the position p_6 *labeled with b. Because* p_6 *follows both* p_4 *and* p_5, *we remove* $\langle p_5, [w_2]\rangle$ *and* $\langle p_4, [w_3]\rangle$ *from* t_b *and increment* d_2 *and* d_3. *We insert* $\langle p_6, [w_2]\rangle$ *to* t_a. *Because* $d_3 = |w_3|$ *and* $\$$ *follows* p_6, w_3 *matches* e. *Since there are no further positions to process, the words* w_1, w_2, *and* w_4 *do not match* d.

Details on how to efficiently handle a-skeleta follow. We assume that the positions $p_1, \dots, p_m$ of e are given in the traversal order of e and that e has been preprocessed for LCA and *Last* queries. Every time we process a position $\langle p, W\rangle$, the list W is nonempty and we increment the index of every word in W, which corresponds to consuming one symbol of every word in W. By $|t_a|$ we denote the number of all nodes that are inserted to t_a throughout the execution of the matching algorithm. Note that for every consumed symbol we add to a-skeleta at most one position and at most one additional LCA node. Therefore, the sum of $|t_a|$ over $a \in \Sigma$ is in $O(|w_1| + \dots + |w_N|)$. We shall use this observation when characterizing the total time necessary to identify, remove, and insert positions in the dynamic a-skeleta.

With every dynamic a-skeleton t_a we maintain the right-most position p_a, i.e., the position most recently added to t_a. We also provide a procedure *findLCA*(t_a, p_i) for localizing in t_a the possible position of the lowest common ancestor n_{LCA} of p_a and a new position p_i which follows p_a in the traversal of e. Note that n_{LCA} needs not be present in t_a and *findLCA*(t_a, p_i) returns the top-most descendant of n_{LCA} present in t_a (which may possibly be n_{LCA} itself if t_a contains it). The procedure simply climbs the right-most path in t_a until the desired node is found. Furthermore, our algorithm performs calls to *findLCA* with subsequent positions in the traversal order, i.e., if a call *findLCA*(t_a, p_i) is followed by a call *findLCA*(t_a, p_j), then $i < j$. The procedure *findLCA* takes advantage of this assumption by saving the result of the previous call and beginning to climb the right-most path of t_a from the saved node (if no new nodes have been added in between). This way the cumulative execution time of all *findLCA* calls with t_a is $O(|t_a|)$, which sums over $a \in \Sigma$ to $O(|w_1| + \dots + |w_N|)$.

The *findLCA* procedure is used to insert new positions as well as to identify and to remove relevant positions from the dynamic a-skeleta. Inserting p_i into t_a is straightforward: we find the lowest common ancestor n_{LCA} of n_a and p_i in e, use *findLCA*(t_a, p_i)) to find if and where to insert n_{LCA} to t_a, and then insert p_i. Identifying and retrieving positions in t_a that are followed by p_i is based on Lemma 2.2. We climb the path from *findLCA*(t_a, p_i) to $n_i = \mathit{parent}(\mathit{pSupFirst}(p_i))$ and at every $\odot$-labeled node n we pick the subtree t' rooted at the left child of n. In one traversal of t' we retrieve all of its Last-positions, because they are followed by p_i, and remove all remaining nodes, because none of the remaining positions is followed by any of the subsequent positions $p_{j \geq i}$. Again, because the number of nodes traversed by the procedure in t_a is proportional to $|t_a|$, the overall time necessary to insert and remove positions in all a-skeleta is $O(|w_1| + \dots + |w_N|)$.

THEOREM 4.12. *For any star-free deterministic regular expression e and words $w_1, \ldots, w_N$, we can decide which words belong to $L(e)$ in time $O(|e| + |w_1| + \cdots + |w_N|)$.*

5. CONCLUSIONS

We have presented a linear time algorithm for testing if a regular expression is deterministic, an efficient algorithm for matching words against deterministic regular expressions, and linear time algorithms for matching against k-occurrence, $*$-free (multiple words), and bounded $+$-depth expressions.

It was our original motivation for this work, but remains an open theoretical problem, whether matching for deterministic regular expressions can be carried out in time $O(|e|+|w|)$. We note that our $O(|e| + |w| \log \log |e|)$ matching algorithm is not optimal because of the $O(\log \log |e|)$ cost of lowest color ancestor queries. We plan to find out if the cost of those lowest colored ancestor queries can be amortized and if the particular order of the queries can be used to devise better data structures. Can other approaches solve the problem in $O(|e| + |w|)$ time, e.g., by giving up the the streaming aspect of using transition simulation? Which larger classes of regular expressions, exceeding the deterministic ones, can be matched efficiently? An example of such class is mentioned after Theorem 4.3, the k-OREs. Another interesting and largely open problem is the one of matching under linear time preprocessing of w. Very simple matching problems such as substring search have time $O(|e|)$ solutions; can those be extended to more general regular expressions? Finally, can lower bounds matching the upper bounds be shown? Note that for general regular expressions and NFAs, it is known that no approach relying on constructing an equivalent epsilon-free NFA can achieve linear complexity. This follows from the fact that all epsilon-free NFAs equivalent to $a_1?a_2?\ldots a_m?$ have at least $m \log^2 m$ transitions [26].

Acknowledgments

We are grateful to the PODS reviewers. Their careful reading of the earlier draft and their plentiful comments allowed to largely improve the presentation of the paper. This research has been partially supported by International Initiatives INRIA Associate Teams TRANSDUCE.

6. REFERENCES

[1] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[2] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(3):117–126, 1986.

[3] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *TWEB*, 4(4), 2010.

[4] G. J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML Schema: A practical study. In *WEBDB*, pages 79–84, 2004.

[5] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *TODS*, 35(2), 2010.

[6] P. Bille and M. Thorup. Faster regular expression matching. In *ICALP*, pages 171–182, 2009.

[7] M. Bojanczyk and P. Parys. XPath evaluation in linear time. *J. ACM*, 58(4):17, 2011.

[8] A. Brüggemann-Klein. Regular expressions into finite automata. *TCS*, 120(2):197–213, 1993.

[9] C.-H. Chang and R. Paige. From regular expressions to DFA's using compressed NFA's. *TCS*, 178(1–2):1–36, 1997.

[10] M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression. In *SPAA*, pages 244–253, 1995.

[11] Apache Software Foundation. Xerces-C++ ver. 3.1.1. http://xerces.apache.org/xerces-c/. File MixedContentModel.cpp.

[12] V. M. Glushkov. The abstract theory of automata. *Russ. Math. Surveys*, 16(5):1–53, 1961.

[13] S. Grijzenhout. Quality of the XML web. Master's thesis, University of Amsterdam, July 2010. Draft, see also http://data.politicalmashup.nl/xmlweb/.

[14] C. Hagenah and A. Muscholl. Computing epsilon-free NFA from regular expressions in $O(n \log^2(n))$ time. In *MFCS*, pages 277–285, 1998.

[15] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[16] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation, Second Edition.* Addison-Wesley, 2000.

[17] R. Kecher. Reset an array in constant time. Blog entry cplusplus.co.il, May 2009.

[18] P. Kilpeläinen. Checking determinism of XML schema content models in optimal time. *Inf. Syst.*, 36(3):596–617, 2011.

[19] P. Kilpeläinen and R. Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Inf. Comput.*, 205(6):890–916, 2007.

[20] C. Konrad and F. Magniez. Validating XML documents in the streaming model with external memory. In *ICDT*, 2012.

[21] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML Schema. *TODS*, 31(3):770–813, 2006.

[22] B. Moret and H. Shapiro. *Algorithms from P to NP*. Benjamin/Cummings, 1990.

[23] S. Muthukrishnan and Müller M. Time and space efficient method-lookup for object-oriented programs. In *SODA*, pages 42–51, 1996.

[24] E. W. Myers. A four russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):430–448, 1992.

[25] J.-L. Ponty, D. Ziadi, and J.-M. Champarnaud. A new quadratic algorithm to convert a regular expression into an automaton. In *WIA*, pages 109–119, 1996.

[26] G. Schnitger. Regular expressions and NFAs without *epsilon*-transitions. In *STACS*, pages 432–443, 2006.

[27] L. Segoufin and C. Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *ICDT*, pages 299–313, 2007.

[28] L. Segoufin and V. Vianu. Validating streaming XML documents. In *PODS*, pages 53–64, 2002.