Regular Expressions with Numerical Occurrence Indicators—preliminary results*

Pekka Kilpeläinen and Rauno Tuhkanen

University of Kuopio
Department of Computer Science
P.O. Box 1627, FIN-70211 Kuopio, Finland
{Pekka.Kilpelainen, Rauno.Tuhkanen}@cs.uku.fi

Abstract. Regular expressions with numerical occurrence indicators (#REs) are used in established text manipulation tools like Perl and Unix egrep, and in the recent W3C XML Schema Definition Language. Numerical occurrence indicators do not increase the expressive power of regular expressions, but they do increase the succinctness of expressions by an exponential factor. Therefore methods based on straightforward translation of #REs into corresponding standard regular expressions are computationally infeasible in the general case. We report some preliminary results about computational problems related to efficient matching and comparison of #REs. Matching, or membership testing of languages described by #REs, is shown to be tractable. Simple comparison problems (inclusion and overlap) of #REs are shown to be NP-hard. We also consider *simple* #REs consisting of a single symbol and nested numerical occurrence indicators only, and derive a simple numerical test for the membership of a word in the language described by a simple #RE.

1 Introduction and Related Work

Regular expressions are used as a standard method to describe and to recognize *regular languages*, for example, in text searching [5], in the implementation of programming language compilers [2], and in grammatical document formalisms [14].

Various extensions of regular expressions are used in different applications. We consider *regular expressions with numerical occurrence indicators* (#REs), which allow the number of required and allowed repetitions of sub-expressions to be defined using numerical parameters. A #RE $E^{m..n}$ denotes, intuitively, the catenation of expression E with itself at least m and at most n times. For example, $a^{2..4}$ denotes the repetition of symbol a from 2 to 4 times, that is, $L(a^{2..4}) = \{aa, aaa, aaaa\}$. #REs are used, for example, in egrep [12] and Perl [19] patterns. Also the recent W3C XML Schema Definition Language (XSDL) [17] uses regular expressions with numerical occurrence indicators (`minOccurs` and `maxOccurs`) to describe content types of XML [3] document elements.

* This work has been supported by the Academy of Finland (Grant no. 102270).

In terms of language theory, numerical occurrence indicators are an inessential extension, since exactly the same class of regular languages can be described with standard regular expressions without numerical occurrence indicators. On the other hand, they are a natural and powerful shorthand notation: Expressing an n-fold repetition of an expression E by an explicit catenation of n copies of E lengthens the expression by factor n, which may be impossible to realize in practice[1]. (Notice that the *value* of an integer n is exponential with respect to the *length* of its numerical representation used in an expression like $E^{m..n}$.) Therefore more efficient realizations of numerically controlled repetition are needed. In spite of this, surprisingly little research has been devoted to this topic. Standard literature of regular expression implementation [1, 2, 16, 7] seems to ignore numerical occurrence indicators, and we have not been able to find any documentation of the algorithms used, e.g., in Perl pattern matching.

It would seem promising to implement numerical occurrence indicators via numerical counters attached to automata built of the expressions using some classical construction. (See, e.g., [18, 1, 2, 4]). We are looking at this possibility, but so far we have not found a method to translate a #RE to a deterministic automaton adorned with such counters. (Simulation of a *nondeterministic* execution with counters could lead to a large number of alternative values for the counters, which seems difficult to implement efficiently.) Laurikari's work [15] is related to this approach. He extends the transitions of NFAs provided by the Thompson construction [18] with procedural *tags* in order to record the positions of the input word that are matched by sub-expressions of the pattern. The tagged NFA is then converted to a DFA; possible ambiguities are resolved using *priorities* assigned to the transitions of the NFA. A problem with this approach is that it may not be possible to assign such priorities to the transitions of an NFA when trying to implement the declarative semantics of regular expressions.

The rest of this paper is organized as follows: We present definitions of #REs, and relate them to standard regular expressions in Section 2. Even though #REs are exponentially more succinct than standard regular expressions, testing whether a word is accepted by a #RE is tractable; this is shown in Section 3. In Section 4 we consider the comparison of the languages described by two #REs: whether one is a sub-language of the other, or whether they have any word in common. Both problems are shown to be NP-hard.

In Section 5 we consider *simple* #REs, which consist of a single symbol and a number of nested numerical occurrence indicators. We devise a simple and efficient test for deciding whether a given word is accepted by a simple #RE. Finally, Section 6 is a conclusion and a discussion of further work.

[1] Apache Xerces, one of the most popular open-source XML parsers, uses this approach to implement numerical repetition. Large enough occurrence values crash Xerces.

2 Regular Expressions with Numerical Occurrence Indicators

Regular expressions describe languages, which are subsets of Σ^* for a given finite alphabet Σ. As usual, we define the *closure* L^* of a language L with $L^* = \bigcup_{i\geq 0} L^i$, where $L^0 = \{\epsilon\}$ and $L^{i+1} = LL^i$; here ϵ denotes the empty word, and catenation of languages L_1 and L_2 is defined by $L_1L_2 = \{uv \mid u \in L_1, v \in L_2\}$.

Regular expressions are built of symbols $\emptyset$, ϵ, and $a \in \Sigma$ connected together using sequential catenation, infix operator $|$, postfix operator *, and parentheses for grouping. The language $L(E)$ described by a regular expression E is defined inductively as follows:

$$\begin{array}{ll} L(\emptyset) = \emptyset; & L(\epsilon) = \{\epsilon\}; \\ L(a) = \{a\} \text{ for } a \in \Sigma; & L(FG) = L(F)L(G); \\ L(F|G) = L(F) \cup L(G); & L(F^*) = L(F)^* \ . \end{array}$$

Regular expressions with numerical occurrence indicators (#REs) use, in addition to the above standard constructs, expressions of the form $F^{m..n}$, where m and n are non-negative integers satisfying $m \leq n$. We call m the *minimum bound* and n the *maximum bound* of $F^{m..n}$. The semantics of numerical occurrence indicators is defined as follows:

$$\begin{aligned} L(F^{m..n}) &= \bigcup_{i=m}^{n} L(F)^i \\ &= \{v_1 \dots v_i \mid m \leq i \leq n, v_1, \dots, v_i \in L(F)\} \ . \end{aligned}$$

That is, $L(F^{m..n})$ consists of words formed by concatenating at least m and at most n words of $L(F)$. It is obvious that numerical occurrence indicators do not increase the expressive power of regular expressions, since

$$L(E^{m..n}) = L(E \cdots E(E|\epsilon) \cdots (E|\epsilon)) \ ,$$

where E is repeated m times and $(E|\epsilon)$ is repeated $n - m$ times.

3 Membership Testing

In this section we show that it is tractable to test whether a given word belongs to the language defined by a regular expression with numerical occurrence indicators. This is shown by the dynamic-programming algorithm which we describe below.

Let $w = a_1 \dots a_n$ be an input word and let E be a #RE. The test for $w \in L(E)$ can be implemented by traversing the expression tree of E bottom-up. During the traversal we compute for each sub-expression F of E a Boolean value F.nullable, and a relation $r(F)$. The meaning of these values is as follows:

$$\begin{aligned} F.\text{nullable} = \textbf{true} \text{ iff } \epsilon \in L(F), \text{ and} \\ r(F) = \{(i,j) \mid 1 \leq i \leq j \leq n, a_i \dots a_j \in L(F)\} \end{aligned}$$

That is, F.nullable indicates whether the empty word is accepted by expression F, and $r(F)$ is the *occurrence relation* of F consisting of pairs of start and end positions of the non-empty sub-words $a_i \dots a_j$ of w that match F. The members (i, j) of an occurrence relation are called *occurrence tuples*. Then $w \in L(E)$ if and only if $(1, n) \in r(E)$ or $w = \epsilon$ and E.nullable = **true**.

The base cases for computing the values of F.nullable and $r(F)$ are as follows:

when $F = \emptyset$, F.nullable := **false**; $r(F) := \emptyset$;
when $F = \epsilon$, F.nullable := **true**; $r(F) := \emptyset$;
when $F = a \in \Sigma$, F.nullable := **false**; $r(F) := \{(i, i) \mid 1 \leq i \leq n, a_i = a\}$.

That is, occurrence relation $r(a)$ consists of the single-position occurrence tuples corresponding to the occurrences of symbol a in the input word $a_1 \dots a_n$.

The inductive cases can be implemented using standard operations of relational algebra [6] on the occurrence relations of the sub-expressions. A choice $F = G|H$ can be implemented using union as follows:

F.nullable := G.nullable **or** H.nullable;
$r(F) := r(G) \cup r(H)$;

A catenation $F = GH$ can be implemented by joining contiguous occurrence tuples of sub-expressions together. A join between relations r and r' restricted by a condition θ on matching attributes is denoted by $r \underset{\theta}{\bowtie} r'$, and projection of a relation r on its first and fourth attribute is denoted by $\pi_{1,4}(r)$. We use (i, j) as the schema for all occurrence relations. Then a join condition which requires matching occurrence tuples of r' to occur immediately after the end position j of occurrence tuples of r can be expressed as $r.j = r'.i - 1$. Catenation $F = GH$ can then be implemented as follows:

F.nullable := G.nullable **and** H.nullable;
$r(F) := \pi_{1,4}(r(G) \underset{r(G).j=r(H).i-1}{\bowtie} r(H))$;
if G.nullable **then** $r(F) := r(F) \cup r(H)$;
if H.nullable **then** $r(F) := r(F) \cup r(G)$;

As an example of using join to implement concatenation, consider an input word $w = aaaaa$ of length 5, and expression $F = GH$, where $G = H = (aa)$. Then

$$r(G) = r(H) = \{(1, 2), (2, 3), (3, 4), (4, 5)\} \ ,$$

and

$$\begin{aligned} r(F) &= \pi_{1,4}(r(G) \underset{r(G).j=r(H).i-1}{\bowtie} r(H)) \\ &= \pi_{1,4}(\{(1, 2, 3, 4), (2, 3, 4, 5)\}) = \{(1, 4), (2, 5)\} \ . \end{aligned}$$

The evaluation of iterative expressions is based on repeating a join a number of times. A key observation for efficient execution is that an occurrence relation needs to be joined with itself no more than n times, when n is the length of the

input word. This holds because each join produces either an empty relation or some new tuples whose end positions are strictly larger than the smallest end position of an occurrence tuple produced so far.

For $F = G^*$ we set F.nullable := **true**, and its occurrence relation can be computed as follows:

$r := r(G)$; $r' := r(G)$;
for $k := 2$ **to** n **do**
    $r := r \cup \pi_{1,4}(r \underset{r.j=r'.i-1}{\bowtie} r')$;
$r(F) := r$;

Numerical iteration $F = G^{m_1..n_1}$ can be treated in a similar manner:

F.nullable := $(m_1 = 0)$ **or** G.nullable;
$r := r(G)$; $r' := r(G)$;
if F.nullable **then** // $L(F) = L(G^{0..n_1})$
    for $k := 2$ **to** $\min\{n_1, n\}$ **do**
        $r := r \cup \pi_{1,4}(r \underset{r.j=r'.i-1}{\bowtie} r')$;
else
    for $k := 2$ **to** $\min\{m_1, n\}$ **do** // required matches of G
        $r := \pi_{1,4}(r \underset{r.j=r'.i-1}{\bowtie} r')$;
    for $k := m_1 + 1$ **to** $\min\{n_1, n\}$ **do** // optional matches of G
        $r := r \cup \pi_{1,4}(r \underset{r.j=r'.i-1}{\bowtie} r')$;
endif;
$r(F) := r$;

As an example of iterating a join, consider matching input word $w = ababab$ against expression $F = G^{2..3}$, where $G = ab$. Now $r(G) = \{(1,2),(3,4),(5,6)\}$, which is assigned to both r and r'. The computation of the final value r of $r(F)$ proceeds as follows:

(**for** $k = 2$) $r := \pi_{1,4}(r \underset{r.j=r'.i-1}{\bowtie} r') = \{(1,4),(3,6)\}$, and
(**for** $k = 3$) $r := r \cup \pi_{1,4}(r \underset{r.j=r'.i-1}{\bowtie} r') = \{(1,4),(3,6)\} \cup \{(1,6)\}$

Now $(1,6) \in r(F)$ indicates that $w \in L(F)$.

When n is the length of the input word, the size of an occurrence relation is at most $n + \binom{n}{2} = O(n^2)$. (This upper bound is reached, e.g., when $w = a^n$ and $F = a^*$.) Operations of relational algebra can be implemented in polynomial time with respect to the size of the operand relations. (See, e.g., [8, Chap. 6]). Since at most $3n$ relational operations are performed for each sub-expression of the #RE, we see that the membership test, or matching against a #RE, can be implemented in polynomial time with respect to the length of the input word.

The procedure sketched above is probably mainly of theoretical interest. Its main drawback as a practical matching algorithm is the space required for storing

the occurrence relations. We saw that quadratic space is needed in the worst case. What we would like to have is a practical algorithm for matching #REs, which would run in low-order polynomial time and in constant space.

4 Comparing Expressions with Numerical Occurrence Indicators

In this section we consider the comparison of two #REs E and F. For example, E and F could be two versions of some XSDL content model, and we would like to test whether they are compatible. That is, we would like to know whether they are equivalent, or whether one is a generalization of the other. The problem can be expressed as a *#RE inclusion problem*: Does it hold that $L(E) \subseteq L(F)$, or vice versa? Notice that an algorithm for the inclusion problem gives also a solution to the equivalence problem $L(E) \stackrel{?}{=} L(F)$.

We show that comparison of #REs is NP-hard using reduction from the NP-complete SUBSET SUM problem [9]. First we show the NP-hardness of the #RE inclusion problem.

Theorem 4.1 *The #RE inclusion problem is NP-hard.*

Proof. Let a sequence of positive integers $s_1, \ldots, s_k$ and n form an instance of the SUBSET SUM problem. (That is, "Is $n = \sum_{i \in I} s_i$ for some subset I of $\{1, \ldots, k\}$?"). Form #REs $E = a^{n..n}$ and

$$F = (a^{s_1..s_1}|\epsilon)(a^{s_2..s_2}|\epsilon) \cdots (a^{s_k..s_k}|\epsilon) \ .$$

Then $L(E) \subseteq L(F)$ iff $n = \sum_{i \in I} s_i$ for some $I \subseteq \{1, \ldots, k\}$. □

Notice that expression F in the above reduction is *ambiguous* in the sense that if it is used for matching a word a^n from left to right, it is not possible to know without lookahead which symbols of the expression "match" the current input symbol.[2]

Comparison of #REs seems difficult also for unambiguous expressions. Let us call the testing of whether $L(E) \cap L(F) \neq \emptyset$ for given #REs E and F the *#RE overlap problem*.

Theorem 4.2 *The #RE overlap problem is NP-hard.*

Proof. Let $s_1, \ldots, s_k$ and n form an instance of the SUBSET SUM problem. Form expressions E and F over alphabet $\Sigma = \{a_1, \ldots, a_k\}$ as follows:

$$E = (a_1|a_2|\cdots|a_k)^{n..n} \text{ and}$$
$$F = (a_1^{s_1..s_1}|\epsilon)(a_2^{s_2..s_2}|\epsilon) \cdots (a_k^{s_k..s_k}|\epsilon) \ .$$

[2] This is the way that the SGML standard [13] and the XML and XML Schema recommendations define the notion of *unambiguity*, which is a property required for content-model expressions in these languages.

Both expressions are unambiguous, since each alphabet symbol occurs only once in both of them. Notice that E accepts all words over Σ whose length is exactly n. Now $L(E) \cap L(F) \neq \emptyset$ iff $n = \sum_{i \in I} s_i$ for some $I \subseteq \{1, \ldots, k\}$. □

The above observations suggest that it is probably difficult to realize #REs as deterministic automata, which is standard technology for implementing traditional regular expressions. In contrast, unambiguous (traditional) regular expressions can be translated to a corresponding DFA in linear time [4], and the inclusion and overlapping of the languages accepted by DFAs can be tested in low-order polynomial time, using a well-known cross-product construction. (See, e.g., [11, 20].)

5 Simple #REs

In this section we consider *simple* #REs, which consist of a symbol and nested numerical occurrence indicators only. An example of a simple #RE with two nested occurrence indicators is given below:

$$E = (a^{3..4})^{1..2} \tag{1}$$

Notice that simple #REs describe unary languages only.

Considering even simple #REs gives some insight of possible difficulties in implementing #REs via automata. As an example, expression (1) above defines the language

$$L(E) = L(a^{3..4}) \cup L(a^{6..8}) \ . \tag{2}$$

For example, accepting $a^6 \in L(E)$ would seem difficult using a Thompson-like automaton [18, 1] built directly from expression (1): Since $a^4 \in L((a^{3..4})^{1..1})$ but $a^4 \notin L((a^{3..4})^{2..2})$, a simple-minded implementation might first match the four initial symbols of a^6 by a sub-automaton built of sub-expression $a^{3..4}$; this would leave two symbols that cannot be accepted by iterating this sub-automaton.

On the other hand, unfolding an expression in the style of (2) could produce a very long expansion. We show below that expressing $E = (a^{m_1..n_1})^{m_2..n_2}$ as a choice between sub-expressions of the form $a^{im_1..in_1}$ may require for each $i = m_2, \ldots, n_2$ its own sub-expression which corresponds to an independent range of accepted words. For example,

$$L((a^{5..6})^{1..4}) = L(a^{5..6}) \cup L(a^{10..12}) \cup L(a^{15..18}) \cup L(a^{20..24}) \ .$$

According to the next proposition it is possible that none of the $n_2 - m_2 + 1$ ranges (like $\{5, 6\}$, $\{10, 11, 12\}$, $\{15, \ldots, 18\}$ and $\{20, \ldots, 24\}$ above) either overlap or meet, and thus cannot be expressed as a single repetition of a.

Proposition 5.1 *Let $m_1 \leq n_1$ and $m_2 \leq n_2$. If $n_1 > n_2(n_1 - m_1) + 1$, then*

$$i \cdot n_1 < (i+1)m_1 - 1$$

for all $i = m_2, \ldots, n_2 - 1$.

Proof. For $i = n_2 - 1$ the conclusion is equivalent to the hypothesis. For smaller values of i it is seen true by induction and the assumption of $m_1 \leq n_1$. □

Nested occurrence indicators can be combined under certain conditions. This holds, for example, if the minimum bound of the inner expression is zero:

Proposition 5.2 $L((E^{0..n_1})^{m_2..n_2}) = L(E^{0..n_1 n_2})$.

Proof. Inclusion from left to right holds because each word of $L((E^{0..n_1})^{m_2..n_2})$ is a catenation of at most $n_1 n_2$ words of $L(E)$.

To see that $L(E^{0..n_1 n_2}) \subseteq L((E^{0..n_1})^{m_2..n_2})$ holds, notice that any word $w \in L(E^{0..n_1 n_2})$ is a catenation of i words of $L(E)$ for some $i \in \{0, \ldots, n_1 n_2\}$. If $m_2 \leq i \leq n_1 n_2$, then

$$w \in \bigcup_{i=m_2}^{n_2} \left(\bigcup_{j=1}^{n_1} L(E)^j \right)^i \subseteq \bigcup_{i=m_2}^{n_2} \left(\bigcup_{j=0}^{n_1} L(E)^j \right)^i = L((E^{0..n_1})^{m_2..n_2}) \ .$$

Else, if $i < m_2$, we can pad w with $m_2 - i$ empty words, and thus

$$w \in (L(E)^0)^{m_2 - i} L(E)^i \subseteq (L(E)^0 \cup L(E)^1)^{m_2} \subseteq \bigcup_{i=m_2}^{n_2} \left(\bigcup_{j=0}^{n_1} L(E)^j \right)^i = L((E^{0..n_1})^{m_2..n_2}) \ .$$

□

The above proposition allows us to restrict, without loss of generality, to simple #REs where zero occurs as the minimum bound of the outermost expression only. (If zero occurs as the minimum bound at some other level, we can repeatedly eliminate the next outer level by applying Proposition 5.2.)

Let $w = a^n$. Next we derive a simple test for deciding whether $w \in L(E)$ for a simple #RE E. For this, consider simple #REs with $k \geq 1$ nested occurrence indicators. When $k = 1$, the test is trivial:

$$a^n \in L(a^{m_1..n_1}) \text{ if and only if } m_1 \leq n \leq n_1 \ .$$

In the case of two nested occurrence indicators, notice that

$$L((a^{m_1..n_1})^{m_2..n_2}) = \bigcup_{i=m_2}^{n_2} \{a^{m_1}, a^{m_1+1}, \ldots, a^{n_1}\}^i \ .$$

This means that $w \in L((a^{m_1..n_1})^{m_2..n_2})$ iff we can split w in $i \in \{m_2, \ldots, n_2\}$ sub-words having their lengths in the range $\{m_1, \ldots, n_1\}$. This holds iff

$$i \cdot m_1 \leq n \leq i \cdot n_1 \ ,$$

which is equivalent to $n/n_1 \leq i \leq n/m_1$. It is straightforward to check that such an $i \in \{m_2, \ldots, n_2\}$ exists if and only if

$$\lceil n/n_1 \rceil \leq \lfloor n/m_1 \rfloor \text{ and } \lfloor n/m_1 \rfloor \geq m_2 \text{ and } \lceil n/n_1 \rceil \leq n_2 \ . \tag{3}$$

For nesting-depths $k > 2$, consider an expression

$$E = (\cdots((a^{m_1..n_1})^{m_2..n_2})\cdots)^{m_k..n_k} \ , \tag{4}$$

where the minimum bound m_h is positive for all levels $h = 1, \ldots, k-1$. Now $w \in L(E)$ iff for some $i \in \{m_k, \ldots, n_k\}$ we can divide w in sub-words $w_1, \ldots, w_i$ in such a way that

$$w_j \in L((\cdots((a^{m_1..n_1})^{m_2..n_2})\cdots)^{m_{k-1}..n_{k-1}})$$

for each $j = 1, \ldots, i$. A similar division of each w_j into smaller sub-words continues at lower levels $h = 1, \ldots, k-1$: Each sub-word u at level h has to split, for some $i \in \{m_h, \ldots, n_h\}$, into sub-words $u_1, \ldots, u_i$ of the next lower level $h-1$ so that

$$u_j \in L((\cdots((a^{m_1..n_1})^{m_2..n_2})\cdots)^{m_{h-1}..n_{h-1}})$$

for each $j = 1, \ldots, i$. An example of this kind of level-wise splitting of word a^{36} into sub-words of length 5 or 6 is shown in Fig. 1.

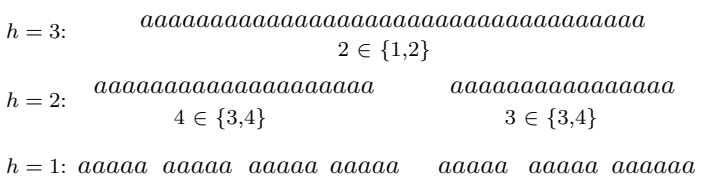

Fig. 1. A level-wise division which shows that $a^{36} \in L(((a^{5..6})^{3..4})^{1..2})$.

Looking at the level-wise construction from the lowest level $h = 1$, we see that $w \in L(E)$ if and only if there is a suitable division of w in $i \in \{l, \ldots, u\}$ sub-words; since these sub-words belong to $\{a^{m_1}, \ldots, a^{n_1}\}$, we have $l = \lceil n/n_1 \rceil$ and $u = \lfloor n/m_1 \rfloor$. Further, $w \in L(E)$ iff the number i of these sub-words is appropriate for the level $h = 2$. That is, we must have

$$b^i \in L((\cdots(b^{m_2..n_2})\cdots)^{m_k..n_k}) \ . \tag{5}$$

(Symbol b acts here as a place-holder for sub-words of this level.) An identical situation holds at the next level: Condition (5) holds iff the following set is non-empty:

$$\{j \in \{\lceil i/n_2 \rceil, \ldots, \lfloor i/m_2 \rfloor\} \mid b^j \in L((\cdots(b^{m_3..n_3})\cdots)^{m_k..n_k})\}$$

That is, a suitable value for $i \in \{l, \ldots, u\}$ exists iff the following set is non-empty:

$$I = \bigcup_{i=l}^{u} \{j \in \{\lceil i/n_2 \rceil, \ldots, \lfloor i/m_2 \rfloor\} \mid b^j \in L((\cdots(b^{m_3..n_3})\cdots)^{m_k..n_k})\} \ .$$

Now $I = \{j \in J \mid b^j \in L((\cdots(b^{m_3..n_3})\cdots)^{m_k..n_k})\}$, where

$$J = \bigcup_{i=l}^{u} \{\lceil i/n_2 \rceil, \ldots, \lfloor i/m_2 \rfloor\} \ .$$

It turns out that set J consists of a single contiguous range of integers:

Proposition 5.3 *Let $l \leq u$ and $1 \leq m_2 \leq n_2$ be integers, and let*

$$J = \bigcup_{i=l}^{u} \{\lceil i/n_2 \rceil, \ldots, \lfloor i/m_2 \rfloor\} \ .$$

Then $J = \{\lceil l/n_2 \rceil, \ldots, \lfloor u/m_2 \rfloor\}$.

Proof. Now

$$J = \{\lceil l/n_2 \rceil, \ldots, \lfloor l/m_2 \rfloor\} \cup \{\lceil (l+1)/n_2 \rceil, \ldots, \lfloor (l+1)/m_2 \rfloor\} \cup \cdots \cup \{\lceil u/n_2 \rceil, \ldots, \lfloor u/m_2 \rfloor\} \ .$$

The boundaries of the constituent sets of J increase by at most one at each step from left to right. Since $m_2 \leq n_2$, the upper bound increases at least once between any two increments of the lower bound, and thus $J = \{\lceil l/n_2 \rceil, \ldots, \lfloor u/m_2 \rfloor\}$. □

Proposition 5.3 means that testing the appropriateness of a range $\{l, \ldots, u\}$ reduces to testing the appropriateness of the range $\{\lceil l/n_2 \rceil, \ldots, \lfloor u/m_2 \rfloor\}$ at the next level.

The above observations are combined together in the following simple algorithm that tests whether $a^n \in L(E)$ for a simple #RE E consisting of k nested numerical occurrence indicators, as given in equation (4). The algorithm computes the boundaries l and u of the range $\{l, \ldots, u\}$ of possible sub-word counts at levels $h = 1, \ldots, k$, and returns **true** iff the range does not get empty (that is, $l \leq u$ holds) and the required number of sub-words at the highest level $h = k$ can be satisfied within the outermost occurrence bounds m_k and n_k.

```
h := 1;
l := n; u := n;
while l ≤ u and h < k do
      l := ⌈l/n_h⌉; u := ⌊u/m_h⌋;
      h := h + 1;
return (l ≤ u and m_k ≤ u and l ≤ n_k);
```

It is easy to check that the algorithm works correctly also when the number of nested occurrence indicators k is one or two; Notice that in the case $k = 2$ the result of the algorithm is the same as the truth value of condition (3) above.

6 Conclusions and Further Work

We have reported some preliminary results about computational problems related to regular expressions with numerical occurrence indicators. #REs are used in established text manipulation tools and in the recent W3C XML Schema Definition Language (XSDL), but we are not aware of them having been treated in the algorithmic literature before.

#REs can be expanded into equivalent regular expressions without numerical occurrence indicators, but this lengthens the expressions by a factor which is exponential with respect to the length of the original expression. We showed in Section 3 that matching of #REs can be performed in polynomial time, without translating them to traditional regular expressions. On the other hand, inclusion and overlap problems between languages described by #REs were shown NP-hard in Section 4.

We also considered so called simple #REs, and derived a simple numerical membership test for languages described by them in Section 5.

We are currently trying to develop an efficient automata-based implementation for #REs. A specific application that we have in mind is the comparison of content model expressions in various XML schema formalisms, especially XML DTD [3] and XSDL [17]. The Glushkov construction [10] seems promising, since when applied to standard regular expressions which satisfy the unambiguity condition of SGML, XML and XSDL, it allows a corresponding DFA to be constructed in linear time [4]. A Glushkov automaton of an expression is also conceptually simple in the sense that there is a one-to-one correspondence between the states of the automaton (except for the initial state) and the occurrences of alphabet symbols in the expression. Especially, all transitions entering a state of a Glushkov automaton are labeled by the same symbol. This regularity gives some promise that we might be able to utilize the results of Section 5 for resolving local ambiguities which were alluded at the beginning of that section.

The NP-hardness results of Section 4 show that it is difficult to implement the comparison of two #REs efficiently. We proved the #RE inclusion problem NP-hard for *ambiguous* expressions only, though, which leaves the possibility of an efficient inclusion test for XSDL content models open.

References

1. A.V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science – Volume A: Algorithms and Complexity*, chapter 5. Elsevier/MIT Press, 1994.
2. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, principles, techniques, and tools.* Addison-Wesley, 1986.
3. T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler, editors. *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation, October 2000.
4. A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:197–213, 1993.

5. C.L.A. Clarke and G.V. Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, 19(3):413–426, May 1997.
6. E.F. Codd. A relational model of data for large shared data banks. *Comm. of the ACM*, 13(6):377–387, June 1970.
7. M. Crochemore and T. Lecroq. Pattern matching and text compression algorithms. In A.B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 8. CRC Press, 2003.
8. H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.
9. M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, New York, 1979.
10. V.M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.
11. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.
12. A. Hume. A tale of two greps. *Software – Practice and Experience*, 18(11):1036–1072, November 1988.
13. International Organization for Standardization. *ISO 8879: Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*, October 1986.
14. P. Kilpeläinen and D. Wood. SGML and XML document grammars and exceptions. *Information and Computation*, 169:230–251, 2001.
15. V. Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Proc. of the Seventh Intl. Symp. on String Processing and Information Retrieval (SPIRE'00)*, pages 181–187. IEEE, 2000.
16. S. Sippu and E. Soisalon-Soininen. *Parsing Theory*, volume I: Languages and Parsing. Springer-Verlag, 1988.
17. H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, editors. *XML Schema Part 1: Structures*. W3C Recommendation, May 2001.
18. K. Thompson. Regular expression search algorithm. *Comm. of the ACM*, 11(6):419–422, June 1968.
19. L. Wall and R.L. Schwartz. *Programming perl*. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.
20. D. Wood. *Theory of Computation*. John Wiley & Sons, Inc., 1987.