AUGMENTED REGULAR EXPRESSIONS: A FORMALISM TO DESCRIBE, RECOGNIZE AND LEARN A CLASS OF CONTEXT-SENSITIVE LANGUAGES.

René Alquézar$^{(*)}$ and Alberto Sanfeliu$^{(**)}$
$^{(*)}$*Dept. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya*
$^{(**)}$*Institut de Cibernètica, Universitat Politècnica de Catalunya - CSIC*
alquezar@lsi.upc.es, sanfeliu@ic.upc.es

Abstract

In order to extend the potential of application of the syntactic approach to pattern recognition, the efficient use of models capable of describing context-sensitive structural relationships is needed. Moreover, the ability to learn such models from examples is interesting to automate as much as possible the development of applications. In this paper, a new formalism that permits to describe a non-trivial class of context-sensitive languages, the Augmented Regular Expressions (AREs), is introduced. AREs augment the descriptive power of regular expressions by including a set of constraints that involve the number of instances of the operands of the star operations in each string of the language. Likewise, algorithms are given to infer AREs from string examples and to recognize language strings by AREs. The method for learning AREs consists of a regular grammatical inference step, aimed at obtaining a regular superset of the target language, followed by a constraint induction process, which reduces the extension of the inferred language transforming it into a context-sensitive one. Hence, this two-step approach avoids the difficulty of learning context-sensitive grammars directly from the data. The method for recognizing language strings is also splitted in two stages: matching the underlying regular expression and checking that the resulting star instances satisfy the constraints.

1. Introduction

A foremost problem in the field of inductive inference is the problem of learning formal languages from partial information. The reported methods can be classified depending on the class of languages they are able to infer and depending on whether the language is presented by examples or by queries [6].

If the input is restricted to just examples, it is usual to distinguish between a *positive* presentation of a language L, this is a set of positive examples $S^+ \subseteq L$, and a (so-called) *complete* presentation[1], where two disjoint finite sets $S^+ \subseteq L$ (examples) and $S^- \subseteq \overline{L}$ (counterexamples) are given. The inference problem associated with the presentation by examples is to find a formal description D (e.g. an acceptor or a grammar) of a language L' such that $S^+ \subseteq L'$ and $S^- \subseteq \overline{L'}$, and D satisfies certain restrictions (e.g. D is the smallest acceptor among all candidates). This problem is traditionally

[1] The term *complete* presentation is somewhat confusing, since obviously, the complete (often infinite) language L is not presented.

referred to as *grammatical inference* (GI) [21, 31]. A GI method is said to *identify L in the limit* if for larger and larger collections of examples, the descriptions D eventually converge to a correct description for L. It is well-known that any enumerable class of recursive languages (context-sensitive and below) can be identified in the limit from complete presentation (both positive and negative data), whereas only certain non-superfinite[2] classes of languages (below regular languages) can be identified from a positive presentation [3, 13].

The presentation of a language L by queries is an oracle that answers, for any query s, whether $s \in L$ or not. The inference methods for this type of presentation yield a description D of L after making some queries about L. A mixed presentation by both examples and queries is also possible, in which the examples are used to infer an initial description to be refined later by queries [5, 23].

The most part of the work in GI has been devoted to the problem of inferring regular grammars (or finite-state automata) [15, 21]. Nevertheless, just a few methods have been proposed for regular GI from complete presentation, either in the classical symbolic approach [1, 14, 22] or in conjunction with other approaches such as genetic search [9] and recurrent neural networks [2, 12, 28]. The most part of the reported regular GI methods are *heuristic* techniques that use only positive examples and some a-priori knowledge (e.g. [18, 25]). In addition, some GI methods have been suggested to learn proper subclasses of context-free languages, such as the even linear languages, from positive data [24, 29], and a few more algorithms have been proposed to infer general context-free grammars from positive structural examples [19, 26] (or together with negative strings [11]).

However, in order to be used in a wider class of problems and applications, there is a need for GI methods that cope with the issue of learning context-sensitive languages. This requirement is specially significant in syntactic pattern recognition problems for computer vision [10], where many objects contain symmetries and structural relationships that are not describable by context-free languages. Nonetheless, work on learning context-sensitive languages is extremely scarce in the literature.

Recently, Takada has shown that a hierarchy of language families that are properly contained in the family of context-sensitive languages can be learned using regular GI algorithms [30]. His approach is based on using control sets on grammars (a type of regulated rewrittings [27]) and establishing a recursive sequence of controlling grammars that starts on a regular grammar. A drawback of Takada's method is that the classes of learnable context-sensitive languages are quite restricted, since the languages are generated through a sequence of (controlled) universal even linear grammars.

On the other hand, an early work by Chou and Fu [8] discussed the matter of inferring transition networks (TNs) [32] from positive examples. An extension of the (heuristic) *k-tails* regular GI method was proposed for learning Basic TNs, thus covering the inference of context-free languages. They gave afterwards a derived trial-and-error scheme (guided by a teacher) for the inference of Augmented TNs representing context-sensitive languages, which required some kind of a-priori knowledge in the form of transformational rules. In general, however, the step form BTN to ATN learning is a hard one, since it is not clear at all how can the test conditions and register-setting actions typical of ATN arcs be inferred from just string examples. Not surprisingly, no other paper following this line of research has been reported since then [21, 31].

Finally, the inductive inference of pattern languages has been studied [4, 20]. Pattern languages are not comparable to the Chomsky's hierarchy of languages, but they provide a limited mechanism to take into account some context influences, basically, the repetition

[2]A superfinite class of languages includes all the finite languages and at least one that is infinite.

of variable substrings along the strings of a language. Again, the descriptive power of such languages appears to be insufficient to deal with non-trivial context-sensitive structures.

In the present paper, a new (yet simple) formalism that allows to describe, recognize and learn a class of non-trivial context-sensitive languages, the Augmented Regular Expression (ARE), is introduced. An overview of the proposed method for inferring AREs is given in Section 2. As in Takada's approach, a regular GI algorithm plays a key role, since the learning process is splitted in two parts: inferring a regular super-set of the target language, and inducing afterwards a set of relations that constrain the regular language to a context-sensitive one. Regular expressions (REs), and their derivation from finite state automata (FSA), are reviewed in Sections 3 and 4, respectively. The *star variables* and *star tree* concepts associated with REs are introduced in Section 5. Two methods to parse strings by REs in order to build a *star instances* data structure (to be used in ARE induction) are presented in Section 6. In Section 7, Augmented Regular Expressions (AREs) are formally defined, and a string recognition algorithm for the language represented by an ARE is provided. The inductive method to infer the constraints of an ARE is described in Section 8. Finally, some conclusions and lines for further research are discussed.

2. Overview of the method for inferring context-sensitive language recognizers from sample strings.

Before proceeding to describe the fundamentals and the details of the proposed method for learning a class of context-sensitive language recognizers, it is worthwhile to give firstly a global picture of it in terms of an actual example, and to present afterwards the overall procedure in algorithmic form.

Figure 1 displays a simple but illustrative example. The problem at hand is to learn a recognizer for the class of contours coming from a frontal view of variable-size cylinders with a fixed-size dent at a variable position along the axis. It is clear that the language associated with such an object is context-sensitive[3], and, consequently, we cannot expect that a regular or even a context-free language learning algorithm return a suitable recognizer for this class of objects. Nevertheless, a human observer would quickly find an adequate description like $a^m c^n bdc^p a^m c^p bdc^n$, if provided with a few examples. This shocking comparison would have to reveal that it should be possible to develop automatic learners capable of inferring syntactic descriptions of this kind. In fact, a rather straight-forward (but, as far as we know, unexplored) approach is suggested here to obtain such a learning method.

Let us say that a sample $S = (S^+, S^-)$ of 16 positive and 48 negative examples was provided, corresponding to some variable-size instances of the contours shown in the top of Fig.1. Even though the target language is context-sensitive, one may try to enter this sample into an FSA learning algorithm and analyse the usefulness of the result[4]. For example, the application of the (regular) active grammatical inference method, recently reported [28], to the given S yielded the deterministic FSA displayed in Fig.1, which accounts for the basic repetitive structure of the model but which over-generalizes a lot, accepting rather arbitrary contours without any length restriction. However, this result was a good starting point to search for a more accurate description that included the context constraints.

[3] The length of the two horizontal segments is the diameter of the cylinder, and the lengths of the vertical segments separated by the dent are obviously the same at each side of the axis

[4] A strategy that resembles "the drunk searching the keys under the lamp".

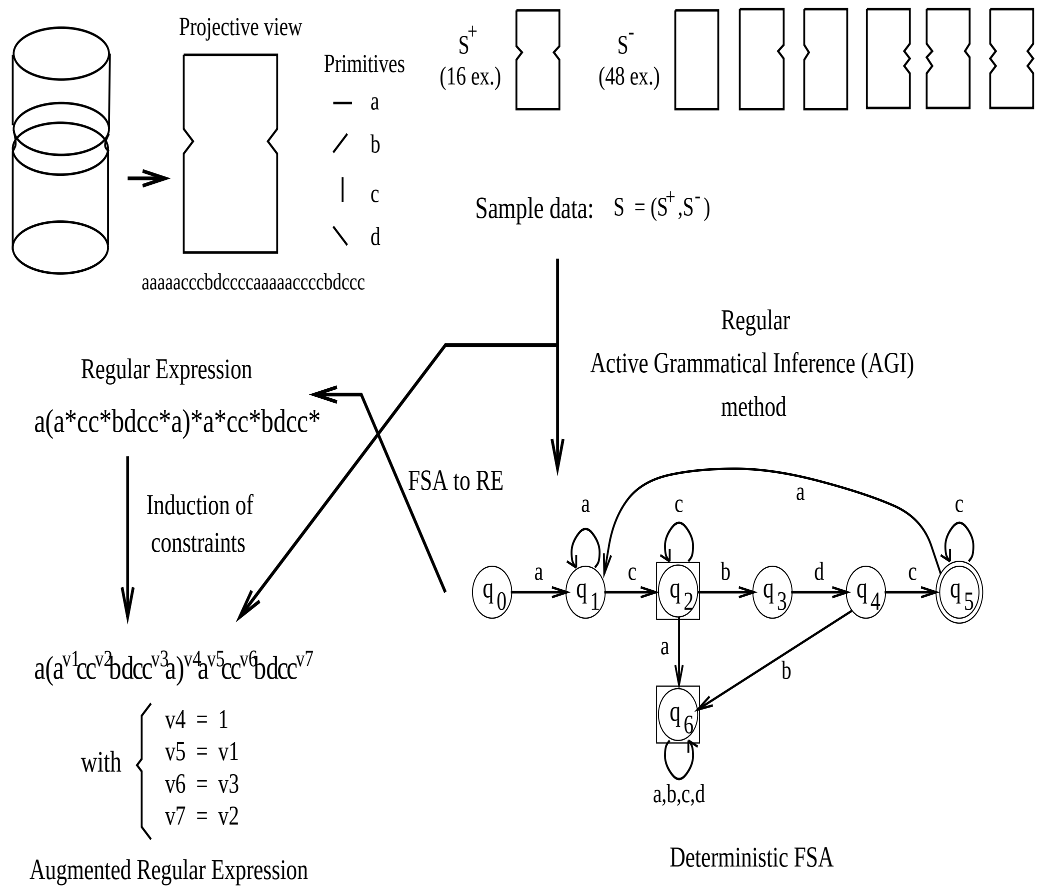


Fig. 1 *An example of application of the proposed method for inferring (some) context-sensitive recognizers.*

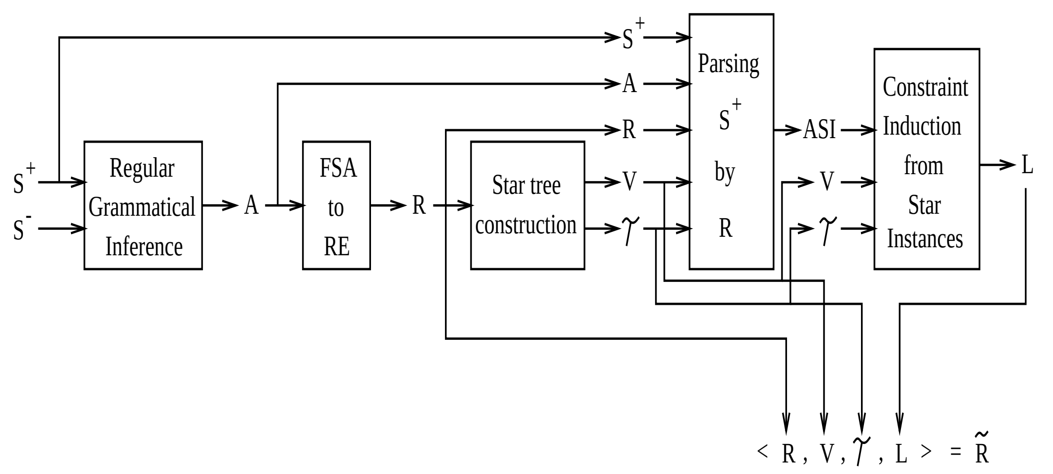


Fig. 2 *Flow diagram of main data in the process of learning Augmented Regular Expressions from string examples.*

The next step is to obtain an equivalent compact representation that facilitates the induction of the constraints, and this turns out to be a regular expression. In our example, the algorithm explained in Section 4 was used yielding the regular expression shown in Fig.1. Finally, from the automatic analysis of the "instances" of the regular expression that were produced by matching (*parsing*) the positive examples, a set of constraints could be derived that, in conjunction with the regular expression, perfectly described the target language (Fig.1). Such a combination of a regular expression and a set of constraints relating its star operators is what we call an Augmented Regular Expression (ARE); a more formal definition will be given later.

The data flow of the process followed to infer an Augmented Regular Expression from examples is depicted in Figure 2. We note that, once the regular expression R is determined from the inferred FSA A, two associated items, the *star variables* V and the *star tree* $\mathcal{T}$ (whose definitions will also be given later), are required both to build an "array of star instances" ASI, containing the information recorded from parsing the examples, and to analyse it in order to induce the set of constraints L. The result of the process can be expressed as the tupla $\tilde{R} = (R, V, \mathcal{T}, L)$.

Method 1 below is a little more detailed description of the learning procedure, with references to the algorithms proposed in the next sections to implement all the processing steps, except the regular grammatical inference step, for which a wide range of algorithms is available [1, 9, 15, 22, 25, 28]. This must not be interpreted, however, as if the choice of the regular GI method were irrelevant. On the contrary, the implicit or explicit biases of the selected method may help or not to reach a "natural" (the simplest in some sense) regular expression for the data, that supports the discovery of the underlying context constraints.

Since some of the variables and parameters mentioned in Method 1 have not been defined yet, we recommend the reader to take a first glance and come back to it when the different parts have been explained and formalized.

METHOD 1: *Induction of augmented regular expressions (for inferring context-sensitive language recognizers)*

Inputs:

A sample $S = (S^+, S^-)$ of a context-sensitive (unknown) language L over an alphabet Σ.

Outputs:

A deterministic FSA $A = (\Sigma, Q, \delta, q_0, F)$, consistent with the sample S (i.e. $S^+ \subseteq L(A) \;\wedge\; S^- \subseteq \Sigma^* - L(A)$), that is obtained from S through a regular grammatical inference algorithm, and which is assumed to accept a regular superset of L (i.e. $L \subseteq L(A)$).

The "canonical" RE R equivalent to A.

The set V of star variables and the star tree $\mathcal{T}$ associated with R.

The array of sets ASI with the lists of instances of the star variables V resulting from the parse of the positive examples ($s \in S^+$) by R.

A set L of linear relations among the star variables in V, that is fully satisfied in the parse of all the positive examples ($s \in S^+$) by R.

The inferred ARE (*Augmented Regular Expression*): $\tilde{R} = (R, V, \mathcal{T}, L)$.

begin

$A :=$ regular_grammatical_inference_algorithm (S^+, S^-)

$n :=$ number_of_states_of_automaton (A)

run_Algorithm_1 $(A, n, R_j, \alpha^l_{ij})$ { returns R_j and α^l_{ij} for $0 \le j < n,\ 1 \le l \le n,\ 0 \le i < l$ }

simplify__equivalent_RE $(A, n, R_j, \alpha^l_{ij}, skelR, R)$ { the "canonical" RE R and its skeleton $skelR$ are obtained }

run_Algorithm_2 $(R, V, \mathcal{T})$ { returns the star variables V and star tree $\mathcal{T}$ associated with R}

$number\_positive\_examples := |S^+|$

for $nexample$ **in** $[1 \,..\, number\_positive\_examples]$ **do**

$s :=$ read_positive_example $(S^+, nexample)$ { where s is a string over Σ }

run_Algorithm_4 $(s, A, skelR, R, V, \mathcal{T}, parsed, path, skelR\_instance, SI)$
{ returns $parsed$ =TRUE, the $path$ of states visited, the skeleton of the example $skelR\_instance$, and the most important, the set SI of lists of star instances of the star variables V resulting from parsing s. }

$ASI[nexample] := SI$

end_for

run_Algorithm_6 $(V, \mathcal{T}, ASI, number\_positive\_examples, L)$ { returns L, the induced set of linear constraints among the star variables }

$\tilde{R} := (R, V, \mathcal{T}, L)$ { simply put the components of the ARE together }

end_method

3. Regular expressions (REs).

Definition. Let $\Sigma = \{a_1, ..., a_m\}$ be an *alphabet* (a finite set of symbols) and let λ denote the *empty string*. The *regular expressions over* Σ and the languages that they describe are defined recursively as follows.

1. $\emptyset$ is a regular expression and describes the empty set.
2. λ is a regular expression and describes the set $\{\lambda\}$.
3. For each $a_i \in \Sigma$ $(1 \leq i \leq m)$, a_i is a regular expression and describes the set $\{a_i\}$.
4. If P and Q are regular expressions describing the languages L_P and L_Q, respectively, then $(P + Q)$, (PQ), and (P^*) are regular expressions that describe the languages $L_P \cup L_Q$ (their *union*), $L_P L_Q$ (their *concatenation*) and L_P^* (the *closure* of L_P), respectively.
5. No other expressions are regular unless they can be generated in a *finite* number of applications of the above rules.

By convention, the precedence of the operations in decreasing order is $*$ (star), (concatenation), $+$ (union). This precedence together with the associativity of the concatenation and union operations allows to omit many parentheses in writing a regular expression.

A language is said to be *regular* if and only if it can be described by a regular expression. We write $L(R)$ for the language described by RE R. Two regular expressions P and Q are said to be *equivalent*, denoted by $P = Q$, if they describe the same language. The following are some basic equivalence rules:

$$
\begin{aligned}
\emptyset + R &= R && (1)\\
\emptyset R &= R\emptyset = \emptyset && (2)\\
\lambda R &= R\lambda = R && (3)\\
\lambda^* &= \lambda && (4)\\
\emptyset^* &= \lambda && (5)\\
R + R &= R && (6)\\
PQ + PR &= P(Q + R) && (7)\\
PQ + RQ &= (P + R)Q && (8)\\
R^* R^* &= R^* && (9)\\
RR^* &= R^* R && (10)\\
(R^*)^* &= R^* && (11)\\
\lambda + RR^* &= R^* && (12)\\
(PQ)^* P &= P(QP)^* && (13)\\
(P + Q)^* &= (P^* Q^*)^* = (P^* + Q^*)^* && (14)\\
(P + Q)^* &= P^*(QP^*)^* = (P^* Q)^* P^* && (15)
\end{aligned}
$$

Although the algebraic manipulation of regular expressions, using the above (and other derived) identities, can be used to simplify them or to demonstrate their equivalence, it does not provide an automatic tool for determining the equivalence of two regular expressions. It is a better approach to convert the given expressions into their equivalent finite-state automata (FSA) and to test these for equivalence (e.g. by minimization) [16].

The following theorem, which proof can be found in [17], is behind the method described in next section to derive regular expressions from FSA.

Theorem 3.1. Let Q, P and R be regular expressions over a finite alphabet. Then, if $L(P)$ does not contain λ, the equation $R = Q + RP$ has a unique solution given by $R = QP^*$.

4. Mapping FSA into equivalent regular expressions.

It is well known (Kleene's theorem [16]) that every language accepted by an FSA can be represented by a regular expression and every language denoted by a regular expression can be recognized by an FSA. Given an FSA A, there can be many equivalent REs R such that $L(A) = L(R)$. Several algorithms have been proposed to find a regular expression that describes the language accepted by a given FSA [16, 17]. By selecting a specific algorithm, a deterministic mapping ψ can be established from FSA to REs, this is, a canonical RE R can be chosen for each FSA A, $R = \psi(A)$. The algorithm that is described next is based on a method due to Arden [7, 17], but a final step has been appended to simplify the resulting regular expression.

4.1. A basic algorithm to find a regular expression equivalent to a given FSA.

Let $A = (\Sigma, Q, \delta, q_0, F)$ be an FSA, where Σ is a finite set of input symbols (alphabet), Q is a finite set of states, $q_0 \epsilon Q$ is the initial state, $F \subseteq Q$ is a set of final states, and $\delta : (Q \times \Sigma) \rightarrow 2^Q$ is a state transition (partial) function. Let us assume that Q has n states and a total order ($<$) is established among them, which can be arbitrary except that the first element is the initial state q_0, i.e. $Q = (q_0, ..., q_{n-1})$. This order can also be applied to the final states, i.e. $F = (q_{f_1}, ..., q_{f_{|F|}})$, where $0 \leq f_1 < ... < f_{|F|} \leq n-1$.

Let α^l_{ij} be the RE that denotes the set of strings (from Σ^*) that take the automaton from state q_i to state q_j without passing through a state q_k with $k < l$; α^n_{ij} will include only the direct transitions from q_i to q_j, whereas α^0_{ij} will be the whole set of strings that lead from q_i to q_j. Let R_j be a synonim for α^0_{0j}, this is the RE that describes the set of strings that take the automaton from the initial state q_0 to state q_j; and finally, let R_{ij} be a synonim for α^j_{ij}.

It is clear that a valid RE equivalent to A is given by

$$R = R_{f_1} + ... + R_{f_{|F|}} \tag{16}$$

Hence, a procedure that determined all the R_j, for $0 \leq j < n$, could be used to yield R. Algorithm 1 is such a procedure [17], which is based on solving the following system of symbolic equations:

$$\begin{array}{lllllllll}
(e^n_0): & R_0 & = & R_0\alpha^n_{00} & + & R_1\alpha^n_{10} & + \;\cdots\; + & R_{n-1}\alpha^n_{(n-1)0} & + \;\lambda \\
(e^n_1): & R_1 & = & R_0\alpha^n_{01} & + & R_1\alpha^n_{11} & + \;\cdots\; + & R_{n-1}\alpha^n_{(n-1)1} & \\
.. & .. & & .. & & .. & \cdots & .. & \\
(e^n_{n-1}): & R_{n-1} & = & R_0\alpha^n_{0(n-1)} & + & R_1\alpha^n_{1(n-1)} & + \;\cdots\; + & R_{n-1}\alpha^n_{(n-1)(n-1)} &
\end{array}$$

where e^n_j are labels to identify each equation. This system can be solved in n steps by substituting orderly each R_j into the other equations, using previously the rule given by Theorem 3.1: $\lambda \notin L(P) \Rightarrow (R = Q + RP \Leftrightarrow R = QP^*)$. Whenever this rule is not applicable, the right hand side of the equation can be directly used to replace R_j.

ALGORITHM 1: *Finds the REs associated with the states of an FSA.*

Inputs: A given FSA $A = (\Sigma, Q, \delta, q_0, F)$ with n states.

Outputs: The associated REs R_j and α^l_{ij} $(1 \leq l \leq n,\ 0 \leq j < n,\ 0 \leq i < l)$.

begin

{ build the initial system of equations }

for j **in** $[0 \,..\, n-1]$ **do**

  for i **in** $[0 \,..\, n-1]$ **do**

    $\alpha^n_{ij} \ := \ \sum a_k$ such that $a_k \in \Sigma$ and $q_j \in \delta(q_i, a_k)$

  end_for

  if $j = 0$ **then**

    Establish eq.(e^n_j) as $R_0 \ = \ R_0\alpha^n_{00} + ... + R_{n-1}\alpha^n_{(n-1)0} + \lambda$

  else

    Establish eq.(e^n_j) as $R_j \ = \ R_0\alpha^n_{0j} + ... + R_{n-1}\alpha^n_{(n-1)j}$

  end_if

end_for

{ solve the equations }

for l **in** $[n-1 \,..\, 1]$ **step** -1 **do**

  { solve eq.(e^{l+1}_l) : $R_l \ = \ R_0\alpha^{l+1}_{0l} + ... + R_{l-1}\alpha^{l+1}_{(l-1)l} + R_l\alpha^{l+1}_{ll}$ }

  if $\alpha^{l+1}_{ll} \neq \emptyset$ **then**

    { Apply the rule of Theorem 3.1. to eq.(e^{l+1}_l),

     i.e., $R_l \ = \ (R_0\alpha^{l+1}_{0l} + ... + R_{l-1}\alpha^{l+1}_{(l-1)l})\ \alpha^{l+1*}_{ll}$ }

    for i **in** $[0 \,..\, l-1]$ **do**

      if $\alpha^{l+1}_{il} \neq \emptyset$ **then** $\alpha^l_{il} \ := \ \alpha^{l+1}_{il}\alpha^{l+1*}_{ll}$

      else $\alpha^l_{il} \ := \ \emptyset$

      end_if

    end_for

---

  else

    for i **in** $[0 \,..\, l-1]$ **do**

      $\alpha^l_{il} \ := \ \alpha^{l+1}_{il}$

    end_for

  end_if

  Establish eq.(e^l_l) as $R_l \ = \ R_0\alpha^l_{0l} + ... + R_{l-1}\alpha^l_{(l-1)l}$

{ substitute the right hand side of eq.(e_l^l) for R_l into the rest of equations }

```
for j in [0 .. n-1] do
    if j ≠ l then
        if α_{lj}^{l+1} ≠ ∅ then
            for i in [0 .. l-1] do
                if α_{il}^l ≠ ∅ then
                    if α_{ij}^{l+1} ≠ ∅ then
                        α_{ij}^l := (α_{ij}^{l+1} + α_{il}^l α_{lj}^{l+1})
                    else
                        α_{ij}^l := α_{il}^l α_{lj}^{l+1}
                    end_if
                else
                    α_{ij}^l := α_{ij}^{l+1}
                end_if
            end_for
        else
            for i in [0 .. l-1] do
                α_{ij}^l := α_{ij}^{l+1}
            end_for
        end_if
        if j = 0 then
            Establish eq.(e_j^l) as   R_0 = R_0 α_{00}^l + ... + R_{l-1} α_{(l-1)0}^l + λ
        else
            Establish eq.(e_j^l) as   R_j = R_0 α_{0j}^l + ... + R_{l-1} α_{(l-1)j}^l
        end_if
    end_if
end_for
end_for
```

{ solve the equation (e_0^1) : $R_0 = R_0\alpha_{00}^1 + \lambda$ }

```
if α_{00}^1 ≠ ∅ then
    R_0 := α_{00}^1*  { = λα_{00}^1* }
else
    R_0 := λ
end_if
```

{ substitute R_0 in the rest of equations }

```
for j in [1 .. n-1] do
    if α_{0j}^1 ≠ ∅ then
        R_j := R_0 α_{0j}^1
    else
        R_j := ∅
    end_if
end_for
end_algorithm
```

4.2. A modification of the basic algorithm to discriminate loops from circuits in the resulting regular expression.

In order to be able to describe (and induce) the greatest number of significant context relations using the Augmented Regular Expression formalism, defined in a later section, the underlying RE should be selected among the REs in its equivalence class according to the following two (somewhat opposite) heuristics:

1. Maximize the number of stars.
2. Preserve unambiguity.

The aim of the first heuristic is to increase the potential for asserting and inferring context relations that involve the instances of the star operations in parsing a string by a regular expression. The aim of the second one is to ease both the RE parsing and constraint induction processes and to avoid the inference of spurious relations due to the multiple possible interpretations of an ambiguous RE.

It can be observed that applying any of the equations (9), (11), (14) or (15) (the former two in reverse) to an RE leads to an equivalent RE with a larger number of stars. However, eqs. (9), (11) and (14) introduce a great deal of ambiguity in the resulting RE. For example, let $R^* = a^*$ and take a^5 as input string, it follows that a^0a^5, a^1a^4, a^2a^3, ... are possible parses by a^*a^*, and $a^1a^1a^1a^1a^1$, $a^2a^0a^1a^2$, $a^1a^3a^1$, ... are possible parses by $(a^*)^*$. On the other hand, equation (15) not only preserves but enforces unambiguity, since an RE containing a union operation is transformed into an RE containing just concatenation and star operations. For instance, let $(P+Q)^* = (a+b)^*$ and take $a^3ba^3ba^3$ as input string, it results that $(a+b)^8$ is the parse by $(a+b)^*$ while $(a^3b)^2a^3$ is the parse by $(a^*b)^*a^*$.

All the stars in the output RE of Algorithm 1 are originated by instantiating the rule of Theorem 3.1 in solving the equations (e_l^{l+1}), $0 \leq l < n$, i.e.
$R_l = (R_0\alpha_{0l}^{l+1} + ... + R_{l-1}\alpha_{(l-1)l}^{l+1}) + R_l\alpha_{ll}^{l+1} \Rightarrow R_l = (R_0\alpha_{0l}^{l+1} + ... + R_{l-1}\alpha_{(l-1)l}^{l+1})\ \alpha_{ll}^{l+1*}$.
Therefore, if the RE α_{ll}^{l+1} is of the form $(P+Q)$, we may apply (15) to the subexpression α_{ll}^{l+1*} to yield a "better" result, in terms of the above heuristics. In the general case, there can be many ways of decomposing α_{ll}^{l+1} into the P and Q union operands. A meaningful decomposition is given by $P = \alpha_{ll}^n$ and $Q = \alpha_{ll}^{l+1} - \alpha_{ll}^n$, where P denotes the direct transitions from q_l to itself (the *loops* of q_l) and Q denotes the *circuits* starting and ending in q_l that only can traverse states q_k with $k > l$. This modification implies the replacement of the part of Algorithm 1 that deals with applying Theorem 3.1 (emphasized with delimiting lines in the algorithmic description) by the following code:

{ Apply the rule of Theorem 3.1. to eq.(e_l^{l+1}),
  i.e., $R_l = (R_0\alpha_{0l}^{l+1} + ... + R_{l-1}\alpha_{(l-1)l}^{l+1})\ \alpha_{ll}^l$ }
if $\alpha_{ll}^{l+1} = \alpha_{ll}^n \ \vee\ \alpha_{ll}^n = \emptyset$ **then**
    $\alpha_{ll}^l := \alpha_{ll}^{l+1*}$ { do not decompose α_{ll}^{l+1} }
else { $\alpha_{ll}^{l+1} = (\alpha_{ll}^n + Q)$ }
    $Q := \alpha_{ll}^{l+1} - \alpha_{ll}^n$ { actually remove the term(s) of α_{ll}^n from the union-type RE α_{ll}^{l+1} }
    $\alpha_{ll}^l := (\alpha_{ll}^{n*}Q)^*\alpha_{ll}^{n*}$ { apply equation (15) }
end_if
for i **in** $[0\ ..\ l-1]$ **do**
    if $\alpha_{il}^{l+1} \neq \emptyset$ **then** $\alpha_{il}^l := \alpha_{il}^{l+1}\alpha_{ll}^l$
    else $\alpha_{il}^l := \emptyset$
    end_if
end_for

In this way, loops can be discriminated from the rest of cycles of the given FSA in the resulting equivalent RE. This is important for pattern recognition tasks, where the loops of a FSA model usually account for (indefinite) length or duration of a basic primitive, a meaningful structure in the pattern (specially if tools are provided to relate the lengths, or durations, of the different parts).

Consider the example of Section 2. The application of Algorithm 1 to the FSA of Fig.1 led to the RE $R = a(a+cc^*bdcc^*a)^*cc^*bdcc^*$, while the above modification on Algorithm 1 yielded $R' = a(a^*cc^*bdcc^*a)^*a^*cc^*bdcc^*$. Although we could further transform R' into the more intuitive description $R'' = aa^*cc^*bdcc^*(aa^*cc^*bdcc^*)^*$ by using eqs. (13) and (10), R' is good enough as underlying RE to enable the inference of all the contextual constraints displayed by the modelled pattern (see Fig. 1). Note that not all the constraints are inferrable from R.

4.3. Simplifying the regular expression obtained by the basic or the modified algorithm.

The RE given by equation (16) can be simplified a lot by determining the common factors (prefixes) in the sum terms. First, let us find a simplified RE for the union of the regular languages R_j for all the states $q_j \in Q$. It is easy to show, by replacing recursively the R_i variables in equations (e_j^j) by the expressions given in equations (e_i^i), for $0 < j < n$ and $0 < i < j$, that the following equivalence holds:

$$\sum_{q_j \in Q} R_j = R_0 + \ldots + R_{n-1} = R_0(\lambda + R_{01}P^Q_{1(n-1)} + \ldots + R_{0(n-1)}P^Q_{(n-1)(n-1)}) = R_0 P^Q_{0(n-1)} \quad (17)$$

where $R_{ij} = \alpha^j_{ij}$, and P^Q_{ik} denotes the set of strings that lead from state q_i to any state $q_j \in Q$ with $i \leq j \leq k$ without passing through a state q_l with $l \leq i$, and it can be defined recursively as

$$P^Q_{ik} = \lambda + \sum_{j=i+1,k}^{R_{ij}\neq\emptyset} R_{ij}\ P^Q_{jk} \quad (18)$$

Next, we define a relation $q_i \sim q_j$ in the following way:

$$q_i \sim q_j \ \Leftrightarrow\ (j < i) \wedge R_{ji} \neq \emptyset$$

Now, let $Q' \subseteq Q$ be any subset of states, and let $C(Q')$ be the transitive closure of Q' with respect to the relation $\sim$. It can be proved that

$$P^{Q'}_{ik} = \emptyset \ \Leftrightarrow\ q_i \notin C(Q'), \quad \forall k \geq i \quad (19)$$

Hence, $P^{Q'}_{ik}$ can be computed recursively using $C(Q')$ for filtering null terms as follows

$$P^{Q'}_{ik} = \begin{cases} \lambda + \displaystyle\sum_{j=i+1,k}^{R_{ij}\neq\emptyset\ \wedge\ q_j\in C(Q')} R_{ij}\ P^{Q'}_{jk} & \text{if } q_i \in Q' \\ \displaystyle\sum_{j=i+1,k}^{R_{ij}\neq\emptyset\ \wedge\ q_j\in C(Q')} R_{ij}\ P^{Q'}_{jk} & \text{otherwise} \end{cases} \quad (20)$$

Then the union of the regular languages associated with the states of Q' is equivalent to a simplified RE with the extracted common factors:

$$\sum_{q_j \in Q'} R_j = R_0 P^{Q'}_{0(n-1)} \quad (21)$$

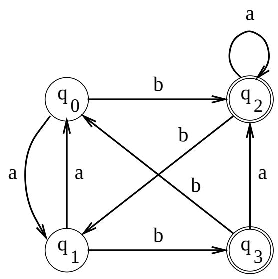
Furthermore, if $Q' \neq \emptyset$ then $P^{Q'}_{0(n-1)} = P^{Q'}_{0z}$, where q_z is the state of Q' with the largest index.

Finally, for the RE equivalent to the given automaton A, we obtain

$$R = \psi(A) = R_0 P^F_{0f_{|F|}} \quad \left(= \sum_{q_f \in F} R_f \right) \tag{22}$$

We refer to the string that is obtained by substituting in R the REs corresponding to R_0 and R_{ij} by the symbols "R0" and "Ri_j" as the *skeleton* of R.

Consider, for example, the 4-state FSA that is displayed in Fig.3a). The two equivalent REs, given by eqs.(16) and (22), after running Algorithm 1, are shown in Fig.3c) and d), respectively. The skeleton of the simplified one is displayed in Fig.3e).



a)

$$\begin{aligned}
R_0 &= ((a+ba^*b)(baa^*b)^*(a+bb))^* \\
R_1 &= ((a+ba^*b)(baa^*b)^*(a+bb))^* \ (a+ba^*b)(baa^*b)^* \\
R_2 &= ((a+ba^*b)(baa^*b)^*(a+bb))^* \ (ba^* + (a+ba^*b)(baa^*b)^*baa^*) \\
R_3 &= ((a+ba^*b)(baa^*b)^*(a+bb))^* \ (a+ba^*b)(baa^*b)^*b
\end{aligned}$$

b)

$$\begin{aligned}
R_2 + R_3 = \ & ((a+ba^*b)(baa^*b)^*(a+bb))^* \ (ba^* + (a+ba^*b)(baa^*b)^*baa^*) \ + \\
& ((a+ba^*b)(baa^*b)^*(a+bb))^* \ (a+ba^*b)(baa^*b)^*b
\end{aligned}$$

c)

$$R = R_0 \ P^{\{q_2,q_3\}}_{03} = ((a+ba^*b)(baa^*b)^*(a+bb))^* \ ((a+ba^*b)(baa^*b)^* \ (baa^* \ + \ b) \ + \ ba^*)$$

d)

$$skel(R) = R0 \ (R0\_1 \ (R1\_2 \ + \ R1\_3) \ + \ R0\_2)$$

e)

Fig. 3 **a)** *An example FSA A with 4 states*
b) *The regular languages associated with its states given by Algorithm 1*
c) *The straightforward RE for A:* $\sum_{q_f \in F} R_f$
d) *The simplified RE for A:* $R = \psi(A) = R_0 \ P^F_{0f_{|F|}}$
e) *The skeleton of R*

5. The star variables and the star tree of a regular expression.

Let V be an ordered set of variables $\{v_1, ..., v_{ns}\}$, which are associated one-to-one with the *star symbols* that appear in a given regular expression R, where ns denotes the number of *star symbols* in R. We refer to the elements of V as *star variables*. Let $pos(V, i)$ be a function that returns the position in R of the star symbol associated with the star variable v_i with index i; moreover, if p is the position in R of a star symbol, then $i = pos^{-1}(V, p)$ gives the index of the corresponding star variable v_i. The function pos can be used to order the set V: $v_i < v_j \Leftrightarrow pos(V, i) < pos(V, j)$.

For $v_i, v_j \in V$, we say that v_i *contains* v_j iff the operand of the star associated with v_i in R includes the star corresponding to v_j; and we say v_i *directly-contains* v_j iff v_i *contains* v_j and there is no $v_k \in V$ such that v_i *contains* v_k and v_k *contains* v_j.

Given any RE R, an associated *star tree* $\mathcal{T} = (N, E, r)$ can be built such that the root node r is a special symbol, the set of nodes is $N = V \cup \{r\}$, and the set of directed edges E (i.e. the structure of the tree) is defined by the containment relationships of the star variables in the following manner:

i) for all the star variables $v_i \in V$ that are not *directly-contained* by other star variable, an edge (r, v_i) is created (v_i is said to be a *son* of r), and therefore v_i is located in the first level of the tree[5];

ii) for all $v_i, v_j \in V$, if v_i *directly-contains* v_j then an edge (v_i, v_j) is created (and v_j is said to be a *son* of v_i).

Furthermore, let us assign an integer identifier to each node of the star tree $\mathcal{T}$: let the identifier of the root r be 0, and let the identifier of any other node be the index i of the star variable v_i corresponding to the node ($1 \leq i \leq ns$).

Algorithm 2 describes how to build the *star tree* $\mathcal{T}$ of a given RE R.

ALGORITHM 2: *Builds the star tree associated with a given RE.*
Inputs: A given RE R over an alphabet Σ.
Outputs: The corresponding set V of star variables ordered by position in R
(which include a representation for the function pos), and
the star tree $\mathcal{T}$ associated with R.

begin

obtain_star_variables_of_RE (R, V, pos) { returns a representation of V and the function pos }
$r :=$ create_startree_node(0) { with 0 as identifier of the node }
concatenation_analysis $(R, 0, V, r)$ { Analyse R as a concatenation of REs,
linking the trees of its outermost star variables as sons of r }
$\mathcal{T} :=$ tree_whose_root_is(r)

end_algorithm

procedure *concatenation_analysis* $(P, p0, V, father)$
input arguments:
P is a RE over Σ
$p0$ is the position of P in the whole RE R
V is the set of star variables of R
input/output arguments:
$father$ is the node of the star tree $\mathcal{T}$ whose subtree is being built

[5]The root r is at level 0

begin

U :=find_factors(P) $\{ U = (u_1 \; ... \; u_m)$ where $P = u_1 ... u_m$ and $m \geq 1 \}$
for i **in** $[1 \; .. \; m]$ **do**
    k := position_of_u_i_in_P (u_i, P)
    l := length(u_i)
    case
        $u_i = a$ $\{$ where $a \in \Sigma$ $\}$
            $\{$ Do nothing $\}$
        $u_i = a^*$ $\{$ where $a \in \Sigma$ $\}$
            $p := p0 + k + 1$ $\{$ is the position of $*$ in R $\}$
            $j := pos^{-1}(V, p)$ $\{$ is the index of the associated star variable v_j $\}$
            $node$:= create_startree_node(j) $\{$ with j as identifier of the node $\}$
            create_edge_in_startree($father, node$)
        $u_i = (P')$ $\{$ where P' is a RE over Σ $\}$
            union_analysis $(u_i, p0 + k, V, father)$ $\{$ Analyse u_i as a union of REs,
                linking the trees of its outermost star variables as sons of $father$ $\}$
        $u_i = (P')^*$ $\{$ where P' is a RE over Σ $\}$
            $p := p0 + k + l - 1$ $\{$ is the position of $*$ in R $\}$
            $j := pos^{-1}(V, p)$ $\{$ is the index of the associated star variable v_j $\}$
            $node$:= create_startree_node(j) $\{$ with j as identifier of the node $\}$
            union_analysis $((P'), p0 + k, V, node)$ $\{$ Analyse (P') as a union of REs,
                linking the trees of its outermost star variables as sons of $node$ $\}$
            create_edge_in_startree($father, node$)
    end_case
end_for

end_procedure

procedure *union_analysis* $(P, p0, V, father)$
input arguments:
    P is a RE over Σ of the form (P')
    $p0$ is the position of P in the whole RE R
    V is the set of star variables of R
input/output arguments:
    $father$ is the node of the star tree $T(R)$ whose subtree is being built

begin

T :=find_terms(P) $\{ T = (t_1, ..., t_n)$ where $P = (t_1 + ... + t_n)$ and $n \geq 1 \}$
for i **in** $[1 \; .. \; n]$ **do**
    k := position_of_t_i_in_P (t_i, P)
    concatenation_analysis $(t_i, p0 + k, V, father)$ $\{$ Analyse t_i as a concatenation of REs,
        linking the trees of its outermost star variables as sons of $father$ $\}$
end_for

end_procedure

6. Parsing strings by regular expressions.

Since any RE R describes a language $L(R)$ of strings (over Σ), it is a natural question to know under what conditions and how can a given string $s \in \Sigma^*$ be parsed by R. The parsing algorithm must return "yes" or "not" depending on whether $s \in L(R)$ or not, and in the first case, it should also return a kind of "instance" of the RE R which would just describe s (something similar to a derivation tree in the case of parsing a string by a grammar).

A RE will be *ambiguous* if there exists a string $s \in L(R)$ for which more than one "instance" of R can be built. The RE parsing algorithms may be classified depending on whether the input RE R is allowed to be *ambiguous*. If it is known that R has been obtained by applying an FSA-to-RE transformation algorithm (such as the one described in Section 4) to a *deterministic* FSA A, then it is sure that R will be *unambiguous*, and hence, a simpler parsing algorithm (with respect to the general case) may be used, since a unique parse of s by R is possible. Furthermore, if the source FSA A is available, the analysis of s by A, i.e. the path of states visited when reading s, can considerably help the task of building the "instance" of R for s.

In next subsection, a data structure is presented which is designed to store the information of the instances of the star variables that occur in parsing a string by an *unambiguous* RE R. This structure can be viewed as a partial representation of what we have called an "instance" of R, since the matched sub-expressions theirselves are not recorded[6]. In subsections 6.2 and 6.3, two *unambiguous* RE parsing algorithms are presented which build the star instances data structure, the first using the RE alone, and the second using the source deterministic FSA as well.

6.1. The star instances data structure.

A star variable v can take as value any natural number, whose meaning is the number of consecutive times (cycles) the operand of the corresponding star (a RE) is matched against a given substring. In such a case, we say that the star variable is instantiated, and sometimes we refer to its value as an "actual exponent". For computational purposes, we will see that it is useful to assign a particular negative integer, say -1, to a star variable v when its father in the star tree $\mathcal{T}$ is instantiated but v is not during a substring parsing.

When a string s is parsed by an *unambiguous* RE R, the associated star variables $v_i \in V$ $(1 \leq i \leq ns)$ will be instantiated zero, one, or more times, depending both on the instances of the star variables that *directly-contain* them and on which terms of the union-type REs in R are selected to parse a substring of s. This is, if the operand of a star in R consists of a union of two or more REs (called terms), it can be traced which term is used for each match of the operand. Consequently, for each cycle, only the star variables that are located in the matched term can be instantiated, while for the rest of star variables that are directly-contained by the external star variable, the negative value -1 can be given to them denoting the lack of instantiation. In this way, all the star variables that are brothers in the star tree $\mathcal{T}$ will have the same structure of potential instances for a given string, either they are actually instantiated or not. Let us put it more formally.

Let $SI_s(V) = \{SI_s(v_1), ...SI_s(v_{ns})\}$ be the set of instances of the star variables in V resulting from the parse of the string s by the RE R (from which V has been defined). Each member of the set is a list of lists containing the instances of a particular star

[6]Some parsing information may be lost if some of the matched sub-expressions belong to a union-type RE and do not contain any star.

variable:

$$\forall i \in [1..ns]: \quad SI_s(v_i) = (l^i_1 \; ... \; l^i_{nlists(i)}) \quad \text{where} \;\; nlists(i) \geq 0$$
$$\forall i \in [1..ns] \; \forall j \in [1..nlists(i)]: \quad l^i_j = (e^i_{j1} \; ... \; e^i_{j(nelems(i,j))}) \quad \text{where} \;\; nelems(i,j) \geq 1$$

The instances stored in the lists are organized according to the containment relationships of the star variables described by the star tree $\mathcal{T}$. This is carried out by defining for each list l^i_j two pointers *father_list*(l^i_j) and *father_elem*(l^i_j) that identify the instance of the father star variable from which the instances of v_i in l^i_j are derived.

For all the star variables that are in the first level of $\mathcal{T}$ (sons of the special root r), the following structure arises:

$$\forall v_i, \;\; (r, v_i) \in \mathcal{T} \quad \Rightarrow \quad SI_s(v_i) = (l^i_1) \;\; \wedge \;\; l^i_1 = (e^i_{11}) \;\; \wedge$$
$$father\_list(l^i_1) = -1 \;\; \wedge \;\; father\_elem(l^i_1) = -1$$

i.e. $nlists(i) = 1$ and $nelems(i, 1) = 1$; furthermore, if v_i is not instantiated in parsing s then $e^i_{11} = -1$ else $e^i_{11} \geq 0$ denotes the number of matches of the star operand in the only instance of v_i.

Otherwise, let v_f be the father of v_i in $\mathcal{T}$. For all the star variables that are in the second level of $\mathcal{T}$, the list of instance lists is either empty (when $e^f_{11} \leq 0$) or its structure is:

$$\forall v_i, \; (r, v_f), (v_f, v_i) \in \mathcal{T} \;\; \wedge \;\; e^f_{11} > 0 \quad \Rightarrow \quad SI_s(v_i) = (l^i_1) \;\; \wedge \;\; l^i_1 = (e^i_{11} \; ... \; e^i_{1(e^f_{11})}) \;\; \wedge$$
$$father\_list(l^i_1) = 1 \;\; \wedge \;\; father\_elem(l^i_1) = 1$$

i.e. $nlists(i) = 1$ and $nelems(i, 1) = e^f_{11}$; furthermore, if v_i is not instantiated in the k-th match of the star operand of v_f then $e^i_{1k} = -1$ else $e^i_{1k} \geq 0$ denotes the number of matches of the star operand of v_i in the k-th cycle.

Finally, for all the star variables v_i that are in the higher levels of $\mathcal{T}$ (with father v_f), we have the following general rule[7]:

$$nlists(i) = \#(e^f_{jk} > 0) \quad \wedge \quad \forall j \in [1..nlists(i)]: \quad nelems(i,j) = e^f_{j'k'} \;\; \wedge$$
$$father\_list(l^i_j) = j' \;\; \wedge \;\; father\_elem(l^i_j) = k'$$

and e^i_{jk} is either a non-negative integer value (the instance of v_i in the k-th cycle of the instance of v_f identified by the pointers {j',k'}) or -1 (if v_i is not instantiated in such cycle).

Figure 4 shows an example of the star instances for a given string and RE, in which the star tree $\mathcal{T}$ has four levels of star variables.

The following operations that deal with the star instance data structure are required in the process of parsing a string by a RE (in what follows $SI[i]$ is a computational representation of $SI_s(v_i)$):

procedure *append_new_list_of_exponents_to_son* $(SI, i, j', k', expo)$
Precondition: $SI[i] = (l^i_1 ... l^i_n)$ where $n = nlists(i) \geq 0$,
$\{j', k'\}$ identify an instance of the father v_f of v_i, and $expo = e^f_{j'k'} > 0$
Postcondition: $SI[i] = (l^i_1 ... l^i_n l^i_{n+1})$ where $nlists(i) = n + 1 \wedge nelems(i, n + 1) = expo$
$\wedge$ *father_list*$(l^i_{n+1}) = j'$ $\wedge$ *father_elem*$(l^i_{n+1}) = k'$ $\wedge$ $\forall k \in [1..expo] : e^i_{(n+1)k} = -1$

[7]This rule is also met, in fact, by the star variables in the second level of $\mathcal{T}$

$$
\begin{array}{lcl}
R & = & (a(b(ce^*c \; + \; df^*d)^*)^*)^* \\
V & = & \{v_1, v_2, v_3, v_4, v_5\} \\
R(V/*) & = & (a(b(ce^{v_1}c \; + \; df^{v_2}d)^{v_3})^{v_4})^{v_5} \\
\mathcal{T} & = & (V \cup r, \{(r, v_5), (v_5, v_4), (v_4, v_3), (v_3, v_1), (v_3, v_2)\}, r) \\
s & = & \text{"}abccdffdcecbddbdfdceecabceeec\text{"} \\
SI_s(v_5) & = & (\ (2)^{\{-1,-1\}}\ ) \\
SI_s(v_4) & = & (\ (3\ 1)^{\{1,1\}}\ ) \\
SI_s(v_3) & = & (\ (3\ 1\ 2)^{\{1,1\}}\ \ (1)^{\{1,2\}}\ ) \\
SI_s(v_1) & = & (\ (0\ \text{-}1\ 1)^{\{1,1\}}\ \ (\text{-}1)^{\{1,2\}}\ \ (\text{-}1\ 2)^{\{1,3\}}\ \ (3)^{\{2,1\}}\ ) \\
SI_s(v_2) & = & (\ (\text{-}1\ 2\ \text{-}1)^{\{1,1\}}\ \ (0)^{\{1,2\}}\ \ (1\ \text{-}1)^{\{1,3\}}\ \ (\text{-}1)^{\{2,1\}}\ )
\end{array}
$$

Fig. 4 *An example of star instances data structure.*

procedure *assign_exponent_to_star_instance* $(expo, SI, i, j, k)$
Precondition: $expo \geq 0$ and $e^i_{jk} = -1$, where e^i_{jk} is the k-th element of the j-th list of instances in $SI[i]$
Postcondition: $e^i_{jk} = expo$

function *read_exponent* (SI, i, j, k) **returns** integer
Precondition: $e^i_{jk} = -1 \ \vee \ e^i_{jk} \geq 0$, where e^i_{jk} is the k-th element of the j-th list of instances in $SI[i]$
Postcondition: e^i_{jk} is returned.

function *number_of_lists* (SI, i) **returns** integer
Precondition: $1 \leq i \leq |V|$, and $nlists(i) \geq 0$ is the number of lists of instances in $SI[i]$
Postcondition: $nlists(i)$ is returned.

function *number_of_elements* (SI, i, j) **returns** integer
Precondition: $1 \leq i \leq |V|$, $1 \leq j \leq nlists(i)$, and $nelems(i, j) \geq 1$ is the number of elements in the j-th list of instances in $SI[i]$
Postcondition: $nelems(i, j)$ is returned.

procedure *initialise_star_instances* $(V, \mathcal{T}, SI)$
Precondition: V is a set of star variables and $\mathcal{T}$ is a star tree over V.
Postcondition: $\forall i \in [1..|V|] : \ v_i \in \text{first\_level}(\mathcal{T}) \ \Rightarrow \ SI[i] = ((-1)^{\{-1,-1\}}) \ \wedge$
$v_i \notin \text{first\_level}(\mathcal{T}) \ \Rightarrow \ SI[i] = ()$

The algorithm for the last of the above operations is described next:

procedure *initialise_star_instances* $(V, \mathcal{T}, SI)$
input arguments: Star variables V and star tree $\mathcal{T}$.
output arguments: The set SI with empty star instances.
begin

$father := r$ { where r is the root node of $\mathcal{T}$ }
$father\_list := -1$
$father\_elem := -1$
{ $father\_list$ and $father\_elem$ point to the instance of the father in SI that generates the list of exponents for a son; in this case, it is a virtual one. }
$father\_exponent := 1$ { $father\_exponent$ determines the length of the next list of exponents to be created for the sons of the current father. }
for i **in** $[1 .. |V|]$ **do**
    $SI[i]$:= create_empty_list()
    $node$:= startree_node_with_identifier_i $(\mathcal{T}, i)$
    if father_of($node$) $=$ r **then**
        { Build a list of one unassigned element for v_i. }
        append_new_list_of_exponents_to_son $(SI, i, father\_list, father\_elem, father\_exponent)$
    end_if
end_for

end_procedure

6.2. An algorithm to parse strings by a regular expression alone.

Algorithm 3 is intended to recognize whether a given string s belongs to the language described by a given unambiguous RE R and, if it belongs, to build the corresponding star instances $SI_s(V)$. The processing of the input string is clearly divided in two phases: the *recognition* and *construction* phases. In the former, only the input string and RE are involved; in the latter, the star variables, the star tree, and the star instances are involved as well. The *construction* phase is a kind of re-run of the *recognition* phase in which it is known in advance that the string will be successfully parsed by the RE, and therefore, the true instances of the star variables can be recorded. To this end, the current star variable that is involved in parsing is tracked, and the exponent of each new instance is computed by counting the number of consecutive matches of the operand of the related star.

The algorithm uses three basic functions (concatenation_parsing, union_parsing and star_parsing) for matching the three types of REs that can appear as subexpressions within the given RE. These functions are recursive, since they call each other to parse shorter and shorter substrings. A concatenation-type RE is regarded as a sequence of factors, each of which is either an alphabet symbol, a union-type RE or a star-type RE. A union-type RE is a sequence of (mutually exclusive) terms, each of them of concatenation-type. Finally, a star-type RE is either an alphabet symbol or a union-type RE which can be matched repeatedly zero or more times.

The worst-time complexity of Algorithm 3 is $O(|s| \cdot |R|)$, where the factor $|R|$ is due to the need of trying to parse a (sub)string by the different terms of a union-type RE in the recognition phase.

ALGORITHM 3: *Parses a string through a RE and builds the lists of instances for its star variables.*

Inputs: A RE R over an alphabet Σ.
The set V of star variables of R.
The star tree $\mathcal{T}$ associated with R.
A given string s from Σ^*.

Outputs: A boolean variable $parsed$ whose value will be TRUE iff s is parsed by R.
The set SI of lists of instances of the star variables V resulting from the parse of s (only if $parsed$ =TRUE).

begin

$build\_SI$:=FALSE { $build\_SI$ is a flag used to signal when it is known in advance that the called parse function will return TRUE, so it only will be run to compute the instances of the star variables. }
{ Try to parse s by R as a concatenation of REs }
$parsed$:= concatenation_parsing $(s, length(s), R, length(R), 0, V, \mathcal{T}, build\_SI,$
$father\_cycle, SI, parsed\_length)$
if $parsed$ **and** $length(s) = parsed\_length$ **then**
initialise_star_instances $(V, \mathcal{T}, SI)$ { Initialise SI by creating one list of one unassigned element for each star variable v_j in the first level of $\mathcal{T}$. }
$build\_SI$:=TRUE
$father := r$ { where r is the root node of $\mathcal{T}$ }
$father\_cycle := 1$ { $father\_cycle$ indicates the next element of the current list to be assigned to an exponent for the star variables that are sons of the current father. }
{ Parse s by R as a concatenation of REs, building the lists of star instances SI }
$parsed$:= concatenation_parsing $(s, length(s), R, length(R), 0, V, \mathcal{T}, build\_SI,$
$father\_cycle, SI, parsed\_length)$
else
$parsed$:=FALSE
end_if

end_algorithm

function *concatenation_parsing* $(s, slength, P, Plength, p0, V, \mathcal{T}, build\_SI,$
$father\_cycle, SI, parsed\_length)$ **returns** boolean
input arguments:
s is a string of length $slength$
P is a RE of length $Plength$ over Σ
$p0$ is the position of P in the whole RE R
V and $\mathcal{T}$ are the star variables and star tree of R, respectively
$build\_SI$ tells whether to build SI or not while parsing
$father\_cycle$ marks the next element to be assigned to an exponent in the current list of the star variables that are sons of the current father (only when $build\_SI$ is TRUE)
input/output arguments:
SI is updated with the star instances resulting from the parse of s by the RE P (only when $build\_SI$ is TRUE)
output arguments:
$parsed\_length$ is the length of the longest prefix of s that is successfully parsed by the RE P
returned value:
if $P \neq \lambda$ then TRUE is returned iff there is a prefix of s that is an instance of the RE P, else ($P = \lambda$) TRUE is returned iff $s = \lambda$.

```
begin
U :=find_factors(P)    { U = (u_1, ..., u_m) where P = u_1...u_m and m ≥ 1 }
if  m = 1 and u_1 = λ then
        parsed_length := 0
        return (slength = 0)
end_if
k := 0    { k is the length of the prefix of s already parsed }
parsed := TRUE;  i := 1    { u_i is the current RE for parsing }
while  parsed and i ≤ m do
        p := position_of_u_i_in_P (u_i, P)
        l := length(u_i)
        case
                u_i = a      { where a ∈ Σ }
                        parsed := (s[k] = a)  { i.e. equivalent to (s[k] = P[p]) }
                        k := k + 1
                u_i = a*     { where a ∈ Σ }
                        exponent := 0
                        while  s[k] = a do
                                k := k + 1;   exponent := exponent + 1
                        end_while
                        if  build_SI then
                                pp := p0 + p + 1    { is the position of * in R }
                                j := pos^{-1}(V, pp)   { is the index of the associated star variable v_j }
                                current_list := number_of_lists (SI, j)
                                current_elem := father_cycle  { current_list and current_elem point
                                    to the element where to assign the exponent for v_j }
                                assign_exponent_to_star_instance (exponent, SI, j, current_list, current_elem)
                        end_if
                u_i = (P')   { where P' is a RE over Σ }
                        substring := string_elements_from_to (s, k, slength)
                        { Try to parse substring by u_i as a union of REs }
                        parsed := union_parsing (substring, slength − k, u_i, l, p0 + p, V, T, build_SI,
                                                        father_cycle, SI, substring_parsed_length)
                        k := k + substring_parsed_length
                u_i = (P')*  { where P' is a RE over Σ }
                        substring := string_elements_from_to (s, k, slength)
                        { Parse substring by u_i as a star RE }
                        parsed := star_parsing (substring, slength − k, u_i, l, p0 + p, V, T, build_SI,
                                                        father_cycle, SI, substring_parsed_length)
                        k := k + substring_parsed_length
        end_case
        i := i + 1
end_while
if parsed then
        parsed_length := k
else
        parsed_length := 0
end_if
return parsed

end_function
```

function *union_parsing* $(s, slength, P, Plength, p0, V, \mathcal{T}, build\_SI,$
$father\_cycle, SI, parsed\_length)$ **returns** boolean

input arguments:

s is a string of length $slength$

P is a RE of length $Plength$ and form (P') over Σ

$p0$ is the position of P in the whole RE R

V and $\mathcal{T}$ are the star variables and star tree of R, respectively

$build\_SI$ tells whether to build SI or not while parsing

$father\_cycle$ marks the next element to be assigned to an exponent in the current list of the star variables that are sons of the current father (only when $build\_SI$ is TRUE)

input/output arguments:

SI is updated with the star instances resulting from the parse of s by the RE P (only when $build\_SI$ is TRUE)

output arguments:

$parsed\_length$ is the length of the longest prefix of s that is successfully parsed by the RE P

returned value:

returns TRUE if there is a prefix of s that is an instance of the RE P, and FALSE otherwise.

begin

T :=find_terms(P) $\{ T = (t_1, ..., t_n)$ where $P = (t_1 + ... + t_n)$ and $n \geq 1$ $\}$
$parsed$:= FALSE; $i := 1$ $\{ t_i$ is the current RE for parsing $\}$
while **not** $parsed$ **and** $i \leq n$ **do**
    p := position_of_t_i_in_P (t_i, P)
    l := length(t_i)
    { Try to parse s by t_i as a concatenation of REs }
    $parsed$:= concatenation_parsing $(s, slength, t_i, l, p0 + p, V, \mathcal{T}, FALSE,$
        $father\_cycle, SI, k)$
    if $parsed$ **then** { the parsed term is t_i }
        if $build\_SI$ **then**
            { Parse again s by t_i now building the star instances SI }
            $parsed$:= concatenation_parsing $(s, slength, t_i, l, p0 + p, V, \mathcal{T}, TRUE,$
                $father\_cycle, SI, k)$
        end_if
    end_if
    $i := i + 1$
end_while
if $parsed$ **then**
    $parsed\_length := k$
else
    $parsed\_length := 0$
end_if
return $parsed$
end_function

function $star\_parsing$ $(s, slength, P, Plength, p0, V, \mathcal{T}, build\_SI,$
$father\_cycle, SI, parsed\_length)$ **returns** boolean

input arguments:

s is a string of length $slength$

P is a RE of length $Plength$ and form $(P')^*$ over Σ

$p0$ is the position of P in the whole RE R

V and $\mathcal{T}$ are the star variables and star tree of R, respectively

$build\_SI$ tells whether to build SI or not while parsing

$father\_cycle$ marks the next element to be assigned to an exponent in the current list of the star variables that are sons of the current father (only when $build\_SI$ is TRUE)

input/output arguments:

SI is updated with the star instances resulting from the parse of s by the RE P (only when $build\_SI$ is TRUE)

output arguments:

$parsed\_length$ is the length of the longest prefix of s that is successfully parsed by the RE P

returned value:

always returns TRUE, since in the worst case $(P')^0$ is matched.

begin

$k := 0$ { k is the length of the prefix of s already parsed }
$parsed :=$ TRUE; $i := 0$ { i is the number of repetitions of (P') found }
while $parsed$ **and** $k < slength$ **do**
    $substring :=$ string_elements_from_to $(s, k, slength)$
    { Try to parse $substring$ by (P') as a union of REs }
    $parsed :=$ union_parsing $(substring, slength - k, (P'), Plength - 1, p0, V, \mathcal{T}, FALSE,$
        $father\_cycle, SI, substring\_parsed\_length)$
    if $parsed$ **then**
        $i := i + 1$
        $s_i :=$ string_elements_from_to $(s, k, k + substring\_parsed\_length)$
        $k := k + substring\_parsed\_length$
    end_if
end_while
$exponent := i$

if $build\_SI$ **then**
    $pp := p0 + Plength - 1$ { is the position of $*$ in R }
    $j := pos^{-1}(V, pp)$ { is the index of the associated star variable v_j }
    $current\_list := number\_of\_lists$ (SI, j)
    $current\_elem := father\_cycle$ { $current\_list$ and $current\_elem$ point to the element where to assign the exponent for v_j }
    assign_exponent_to_star_instance $(exponent, SI, j, current\_list, current\_elem)$

if $exponent > 0$ **then**

$father :=$ startree_node_with_identifier_j $(\mathcal{T}, j)$

$father\_list := current\_list$

$father\_elem := current\_elem$

{ $father\_list$ and $father\_elem$ point to the exponent of the father in SI that generates the list of exponents for a son. }

$father\_exponent := exponent$ { $father\_exponent$ determines the length of the next list of exponents to be created for the sons of the current father }

for $nson$ **in** [1 .. nsons_of($father$)] **do**

$node :=$ son_of $(father, nson)$

$son\_identifier :=$ node_identifier_of $(node)$

{ Create a list of unassigned elements for $v_{son\_identifier}$, whose number is "exponent" }

append_new_list_of_exponents_to_son $(SI, son\_identifier, father\_list,$
$father\_elem, father\_exponent)$

end_for

for i **in** [1 .. $exponent$] **do**

$l :=$ length(s_i)

$new\_father\_cycle := i$

{ Parse s_i by (P') as a union of REs, now building the star instances SI }

$parsed :=$ union_parsing $(s_i, l, (P'), Plength - 1, p0, V, \mathcal{T}, TRUE,$
$new\_father\_cycle, SI, substring\_parsed\_length)$

end_for

end_if

end_if

$parsed\_length := k$

return TRUE

end_function

6.3. An algorithm to parse strings by a regular expression with the help of its equivalent deterministic FSA.

A parsing method more efficient than the previous one is attainable if the FSA A from which the unambiguous RE R has been obtained is given. Algorithm 4 is such a method, which, in addition to the deterministic FSA A, uses some of the REs α_{ij}^{l} yielded by Algorithm 1 and the skeleton of R formed in the subsequent simplification step. The key point is that the automaton (instead of the RE) is used for *recognition*, and the corresponding path of visited states, which is obtained, guides the *construction* of the "instance" of R that matches the input string s (i.e. the star instances data).

There are two achievements in Algorithm 4 that permit to reduce the time complexity of the parsing process. The first one is to locate the substrings of s that are associated with the cycles of the involved star-type REs by finding subpaths of visited states that start and end with the same state without passing through it. The second one is to select directly the term of the involved union-type REs that actually matches the corresponding substring without the need of attempting to parse the non-matched terms.

After recognizing a positive string, the main block of Algorithm 4 splits the path of visited states in a certain number of subpaths, which depend on the final state arrived and on the last visit to the states with lower index. This process can be regarded as finding the instance of the RE skeleton that matches the path of visited states. The subpaths matched to the REs $R_{ij} = \alpha_{ij}^{j}$ are further split in two parts, which correspond respectively to a union-type RE ($R'_{ij} = \alpha_{ij}^{j+1}$) and a star-type RE ($C_j = \alpha_{jj}^{j+1*}$). Obviously, the subpath matched to the RE R_0, if $R_0 \neq \lambda$, correspond to a star-type RE: $C_0 = \alpha_{00}^{1}{}^*$. Then, the input string is split in substrings according to the path segmentation, using the length of each subpath. In this way, each substring so formed is parsed by the corresponding segment of the RE (either a union-type or a star-type RE) and, furthermore, the parsing process is guided by the associated subpath of visited states.

To describe how the guiding mechanism works, the REs R'_{ij} and C_j (for $0 \leq j \leq (n-1), \;\; 0 \leq i < j$), that are used in parsing R, must be decomposed recursively in terms of the REs from which Algorithm 1 has built them. By analysing the behavior of Algorithm 1, the following relations can be established:

$$\begin{array}{rclcl} R'_{ij} & = & \alpha_{ij}^{j+1} & = & \alpha_{ij}^{n} + \alpha_{i(n-1)}^{n}\alpha_{(n-1)(n-1)}^{n}{}^{*}\alpha_{(n-1)j}^{n} + \dots + \alpha_{i(j+1)}^{j+2}\alpha_{(j+1)(j+1)}^{j+2}{}^{*}\alpha_{(j+1)j}^{j+2} \\ C_j & = & \alpha_{jj}^{j+1*} & = & (\alpha_{jj}^{n} + \alpha_{j(n-1)}^{n}\alpha_{(n-1)(n-1)}^{n}{}^{*}\alpha_{(n-1)j}^{n} + \dots + \alpha_{j(j+1)}^{j+2}\alpha_{(j+1)(j+1)}^{j+2}{}^{*}\alpha_{(j+1)j}^{j+2})^{*} \end{array}$$

where some of the above α-labeled REs may represent the empty language $\emptyset$ for a given FSA A.

Symmetrically, a similar relation can be established for the REs α_{ji}^{j+1} (for $1 \leq j \leq (n-1), \;\; 0 \leq i < j$):

$$R''_{ji} = \alpha_{ji}^{j+1} = \alpha_{ji}^{n} + \alpha_{j(n-1)}^{n}\alpha_{(n-1)(n-1)}^{n}{}^{*}\alpha_{(n-1)i}^{n} + \dots + \alpha_{j(j+1)}^{j+2}\alpha_{(j+1)(j+1)}^{j+2}{}^{*}\alpha_{(j+1)i}^{j+2}$$

For clarity purposes, the three preceding relations can be rewritten as follows:

$$R'_{ij} = \alpha_{ij}^{n} + R'_{i(n-1)}C_{n-1}R''_{(n-1)j} + \dots + R'_{i(j+1)}C_{j+1}R''_{(j+1)j} \tag{23}$$

$$C_j = (\alpha_{jj}^{n} + R'_{j(n-1)}C_{n-1}R''_{(n-1)j} + \dots + R'_{j(j+1)}C_{j+1}R''_{(j+1)j})^{*} \tag{24}$$

$$R''_{ji} = \alpha_{ji}^{n} + R'_{j(n-1)}C_{n-1}R''_{(n-1)i} + \dots + R'_{j(j+1)}C_{j+1}R''_{(j+1)i} \tag{25}$$

Consequently, from all the REs α_{ij}^{l} $(1 \leq l \leq n, \; 0 \leq j < n, \; 0 \leq i < l)$ yielded by Algorithm 1, just a subset of at most $2n^2$ α-labeled REs are needed to parse R efficiently, which correspond to the REs denoted by R'_{ij}, R''_{ji}, α^n_{ij}, α^n_{ji} (for $1 \leq j \leq (n-1)$, $0 \leq i < j$), C_j and α^n_{jj} (for $0 \leq j \leq (n-1)$).

Let us explain now how the parsing proceeds when matching a certain substring to a union-type RE of the form R'_{ij} (a similar argument applies to the R''_{ji} REs and to the union-type REs in the C_j REs). Let the subpath of states associated with the substring be $p = p_0 ... p_r$, where we know that $p_0 = i$, $p_r = j$ and $\forall t \in [1, r-1] : (p_t > i) \wedge (p_t > j)$.

If $r = 1$ then the term α^n_{ij} is selected as the matched term. Otherwise, the subpath must be scanned to find the state with the lowest index; let h be such a state, and let $z1$ and $z2$ be the positions of the first and last occurrence of h in the subpath p; then, it is clear that the term $R'_{ih} C_h R''_{hj}$ is the matched term of R'_{ij}, and moreover, the three substrings associated with the subpaths $p_0 ... p_{z1}$, $p_{z1} ... p_{z2}$ and $p_{z2} ... p_r$ must be matched to the REs R'_{ih}, C_h and R''_{hj}, respectively (if $z1 = z2$ then the second substring is empty and, hence, the star-type RE C_h is instantiated zero times).

Since the terms representing empty languages will not appear in the union-type REs included in R, an indexing mechanism is required to access directly to the selected term $R'_{ih} C_h R''_{hj}$ of R'_{ij}, given the state h. For example, an array of n elements, indexed by states, can contain the relative position of the terms that are actually present or a flag denoting absence. Such an array is needed for each one of the non-empty REs R'_{ij}, C_j and R''_{ji}, and it will be referred to as the index of the corresponding subexpression. Note that Algorithm 1 can be modified easily to compute and store these indexes.

Concerning the process of matching a certain substring to a star-type RE of the form C_j, the first step is to split the associated subpath according to the occurrences of its first state. In this way, the substrings that are matched in each cycle of a star instance can be identified quickly. Then each of these substrings must be parsed by the RE that is the operand of the star in C_j (i.e. a union-type RE).

Algorithm 4 uses two guided parsing functions (guided_union_parsing and guided_star_parsing), which implement the irrevocable parsing strategy that has been described. The worst-case time complexity of Algorithm 4 is $O(max\{|skel(R)|, n|s|\})$, where n is the number of states of A, and $|s|$ and $|skel(R)|$ denote the lengths of the input string and the skeleton of R, respectively. This time complexity is due to the *construction* phase, since the *recognition* phase can be run in linear time. The cost $n|s|$ arises from the repeated scanning of subpaths in the search of the state with lowest index (less than n scans for each element of the global path of length $|s|$). The cost $|skel(R)|$ is due to the highest-level segmentation of the input string s, that involves scanning the skeleton of R.

ALGORITHM 4: *If an input string is accepted by a given FSA, the string is then parsed through the equivalent RE, using the path of visited states and building the lists of instances for the star variables.*

Inputs: A deterministic FSA $A = (\Sigma, Q, \delta, q_0, F)$ with n states.
The RE R obtained from A by Algorithm 1 + Simplification step.
The skeleton of R: $skelR$.
The REs R'_{ij}, C_j, R''_{ji}, α^n_{ij}, α^n_{jj}, α^n_{ji} (for $0 \leq j \leq (n-1)$, $0 \leq i < j$), obtained from A by Algorithm 1.
The corresponding indexes IR'_{ij}, IC_j, IR''_{ji} (for $0 \leq j \leq (n-1)$, $0 \leq i < j$).
The set V of star variables of R.
The star tree $\mathcal{T}$ associated with R.
A given string s from Σ^*.

Outputs: A boolean variable $parsed$ whose value will be TRUE iff s is accepted by A (in that case, s is parsed by R).
The path of states $path$ visited by the automaton A during the recognition of the string s.
The instance of the skeleton of R resulting of parsing s: $skelR\_instance$, and the set SI of lists of instances of the star variables V resulting from the parse of s (only if $parsed$ =TRUE).

begin

$parsed :=$ recognize $(s, A, path)$ { this function also returns the path of visited states }

if $parsed$ **then**

$slength := length(s)$
$last\_state := path[slength]$ { $last\_state$ is the index of the final state arrived in the recognition of s }

{ Find the last visit in $path$ of each state lower than q_{last_state} such that q_{last_state} is reachable from it. }
$Q' := \{q_{last\_state}\}$ { $Q' \subseteq F$ is a set whose only member is q_{last_state} }
$G := C(Q')$ { $G \subseteq Q$ is the transitive closure of Q' with respect to the relation $\sim$:
$q_i \sim q_j \Leftrightarrow (j < i) \wedge R_{ji} \neq \emptyset$ }
for i **in** $[0 .. |Q| - 1]$ **do**
if $q_i \in G$ **then**
$last\_visit[i] :=$ find_last_occurrence_of_i_in_path $(i, path)$
end_if
end_for

{ Initialise SI by creating one list of one unassigned element for each star variable v_j in the outermost level. }
initialise_star_instances $(V, \mathcal{T}, SI)$
$father := r$ { where r is the root node of $\mathcal{T}$ }
$father\_cycle := 1$ { $father\_cycle$ marks the next element of the current list to be assigned to an exponent for the star variables that are sons of the current father. }

{ Parse s by R guided by $path$ and $skelR$ }

$p0 := 0$ { $p0$ is the length of the part of R already scanned }
if $C_0 \neq \lambda$ **then**
$substring :=$ string_elements_from_to $(s, 0, last\_visit[0] - 1)$ { always $q_0 \in G$ }
$subpath :=$ path_elements_from_to $(path, 0, last\_visit[0])$
$l := length(substring)$
{ Parse $substring$ by C_0 as a star-type REs, using $subpath$ and building the lists of star instances SI }
guided_star_parsing $(substring, subpath, l, C_0, length(C_0), p0, V, \mathcal{T}, father\_cycle, SI)$
end_if

$p0 := p0 + length(C_0)$
$skelR\_instance :=$"R0" { since $R_0 = C_0$ }
$psk0 := 2$ { $psk0$ is the length of the part of $skelR$ already scanned }

$i := 0$
while $i \neq last\_state$ **do**
$j :=$ Find_minimum_q_j_in_G_visited_later_than_q_i $(i, G, last\_visit)$
$p0 :=$ find_next_Ri-j $(i, j, skelR, R, psk0)$ { scans $skelR$ from position $psk0$ to find the next occurrence of "Ri-j" and returns the position of next R_{ij} in R }
$subpath :=$ path_elements_from_to $(path, last\_visit[i], last\_visit[j])$
$next\_visit\_j :=$ find_first_occurrence_of_j_in_subpath $(j, subpath)$
$substring\_1 :=$ string_elements_from_to $(s, last\_visit[i], next\_visit\_j - 1)$
$subpath\_1 :=$ path_elements_from_to $(path, last\_visit[i], next\_visit\_j)$
$l := length(substring\_1)$
{ Parse $substring\_1$ by R'_{ij} as a union-type REs, using $subpath\_1$ and building the lists of star instances SI }
guided_union_parsing $(substring\_1, subpath\_1, l, R'_{ij}, length(R'_{ij}), p0, V, \mathcal{T},$
$father\_cycle, SI)$
if $C_j \neq \lambda$ **then**
$substring\_2 :=$ string_elements_from_to $(s, next\_visit\_j, last\_visit[j] - 1)$
$subpath\_2 :=$ path_elements_from_to $(path, next\_visit\_j, last\_visit[j])$
$l := length(substring\_2)$
{ Parse $substring\_2$ by C_j as a star-type RE, using $subpath\_2$ and building the lists of star instances SI }
guided_star_parsing $(substring\_2, subpath\_2, l, C_j, length(C_j),$
$p0 + length(R'_{ij}), V, \mathcal{T}, father\_cycle, SI)$
end_if
$p0 := p0 + length(R'_{ij}) + length(C_j)$
$skelR\_instance :=$ append_to_skeleton $(skelR\_instance,$"Ri-j"$, i, j)$
$psk0 := psk0 + length("Ri-j")$ { $psk0$ is the length of the part of $skelR$ already scanned }
$i := j$

end_while

end_if

end_algorithm

action *guided_union_parsing* $(s, path, slength, P, Plength, p0, V, \mathcal{T}, father\_cycle, SI)$
input arguments:
s is a string of length $slength$
$path$ is a state-index array of length $slength + 1$
P is a union-type RE of length $Plength$ over Σ
$p0$ is the position of P in the whole RE R
V and $\mathcal{T}$ are the star variables and star tree of R, respectively
$father\_cycle$ marks the next element to be assigned to an exponent in the current list of the star variables that are sons of the current father
input/output arguments:
SI is updated with the star instances resulting from the parse of s by the RE P

begin

$i := path[0]$; $j := path[slength]$

{ The RE P is of the form $(t_1 + ... + t_k)$ where $k \geq 1$, and it corresponds to R'_{ij} (if $i < j$), R''_{ij} (if $i > j$), or α_{ii}^{i+1} (if $i = j$), where $C_i = \alpha_{ii}^{i+1*}$ }

if $slength = 1$ **then**

{ $t_1 = \alpha_{ij}^n$ is the matched term, and the star instances SI are not affected. }

else

$h :=$ find_lesser_state_in_subpath $(path, slength + 1, z1, z2)$ { always $h > i$ and $h > j$; this function also returns the position of the first and last ocurrence of h in $path$: $z1$ and $z2$, respectively.}

if $i < j$ **then**

$< m, p >:=$ select_matched_term (h, IR'_{ij}) { direct access to the index of R'_{ij} }

else if $i > j$ **then**

$< m, p >:=$ select_matched_term (h, IR''_{ij}) { direct access to the index of R''_{ij} }

else $\{ i = j \}$

$< m, p >:=$ select_matched_term (h, IC_i) { direct access to the index of C_i }

end_if

{ $t_m = R'_{ih} C_h R''_{hj}$ is the matched term and p is the position of t_m in P.}

$substring\_1 :=$ string_elements_from_to $(s, 0, z1 - 1)$
$subpath\_1 :=$ path_elements_from_to $(path, 0, z1)$
$l := length(substring\_1)$
{ Parse $substring\_1$ by R'_{ih} as a union-type REs, using $subpath\_1$ and building the lists of star instances SI }
guided_union_parsing $(substring\_1, subpath\_1, l, R'_{ih}, length(R'_{ih}), p0 + p, V, \mathcal{T}, father\_cycle, SI)$
$p := p + length(R'_{ih})$

if $C_h \neq \lambda$ **then**

$substring\_2 :=$ string_elements_from_to $(s, z1, z2 - 1)$

$subpath\_2 :=$ path_elements_from_to $(path, z1, z2)$

$l := length(substring\_2)$

{ Parse $substring\_2$ by C_h as a star-type RE, using $subpath\_2$ and building the lists of star instances SI }

guided_star_parsing $(substring\_2, subpath\_2, l, C_h, length(C_h), p0 + p, V, \mathcal{T}, father\_cycle, SI)$

$p := p + length(C_h)$

end_if

$substring\_3 :=$ string_elements_from_to $(s, z2, slength - 1)$
$subpath\_3 :=$ path_elements_from_to $(path, z2, slength)$
$l := length(substring\_3)$
{ Parse $substring\_3$ by R''_{hj} as a union-type REs, using $subpath\_3$ and building the lists of star instances SI }
guided_union_parsing $(substring\_3, subpath\_3, l, R''_{hj}, length(R''_{hj}), p0 + p, V, \mathcal{T}, father\_cycle, SI)$

end_if

end_action

action *guided_star_parsing* $(s, path, slength, P, Plength, p0, V, \mathcal{T}, father\_cycle, SI)$

input arguments:

s is a string of length $slength$
$path$ is a state-index array of length $slength + 1$
P is a star-type RE of length $Plength$ over Σ, in fact $P = C_i = \alpha_{ii}^{i+1*}$, where $i = path[0]$.
$p0$ is the position of P in the whole RE R
V and $\mathcal{T}$ are the star variables and star tree of R, respectively
$father\_cycle$ marks the next element to be assigned to an exponent in the current list of the star variables that are sons of the current father

input/output arguments:

SI is updated with the star instances resulting from the parse of s by the RE $P = C_i$

begin

$i := path[0]$ { and it always holds that $i = path[slength]$ }

$S :=$ find_cycle_substrings$(s, path, slength, e)$ { $S = (s_1, ..., s_e)$ where $e \geq 0$ and
$s = s_1...s_e$ such that for $0 < k \leq e$:
$path[start(s_k)] = path[end(s_k)] = i$ and
for $start(s_k) < z < end(s_k)$: $path[z] > i$ }

$exponent := e$ { e is the number of consecutive substrings of s that are matched by α_{ii}^{i+1} }

{ Determine the star variable v_j that has been instantiated }
$pp := p0 + Plength - 1$ { is the position of $*$ in R }
$j := pos^{-1}(V, pp)$ { is the index of the associated star variable v_j }

{ Update the list of star instances of the star variable v_j }
$current\_list := number\_of\_lists\ (SI, j)$
$current\_elem := father\_cycle$ { $current\_list$ and $current\_elem$ point to the element where to assign the exponent for v_j }
assign_exponent_to_star_instance $(exponent, SI, j, current\_list, current\_elem)$

if $exponent > 0$ **then**

{ Establish the pointer to the current instance of v_j in SI }
$father :=$ startree_node_with_identifier_j $(\mathcal{T}, j)$
$father\_list := current\_list$
$father\_elem := current\_elem$
{ $father\_list$ and $father\_elem$ point to the exponent of the father in SI that generates the list of exponents for a son. }
$father\_exponent := exponent$ { $father\_exponent$ determines the length of the next list of exponents to be created for the sons of the current father }

{ Append a virtual list of exponents to the star instances of each star variable that is son of v_j in $\mathcal{T}$. }
for $nson$ **in** [1 .. nsons_of$(father)$] **do**

$node :=$ son_of $(father, nson)$

$son\_identifier :=$ node_identifier_of $(node)$

{ Create a list of unassigned elements for $v_{son\_identifier}$, whose length is the value of $exponent$ }

append_new_list_of_exponents_to_son $(SI, son\_identifier, father\_list,$
$father\_elem, father\_exponent)$

end_for

{ Parse each cycle substring by α_{ii}^{i+1} }

for k **in** [1 .. *exponent*] **do**

s_k := select_$k$th_substring (S, k)

p := position_of_s_k_in_s (s_k)

l := length(s_k)

subpath := path_elements_from_to $(path, p, p + l)$

new_father_cycle := k

{ Parse s_k by α_{ii}^{i+1} as a union-type REs using *subpath* and building the star instances SI }

guided_union_parsing $(s_k, subpath, l, \alpha_{ii}^{i+1}, Plength-1, p0, V, \mathcal{T}, new\_father\_cycle, SI)$

end_for

end_if

end_action

7. Augmented regular expressions (ARE).

7.1. Concept and definition.

An *Augmented Regular Expression* (or ARE for short) is defined as a four-tupla $(R, V, \mathcal{T}, L)$, where R is a regular expression over an alphabet Σ, V is its associated set of star variables, $\mathcal{T}$ is its associated star tree, and L is a set of independent linear relations $\{l_1, ..., l_{nc}\}$, each one involving the variables in V, this is

$$l_i \equiv a_{i1}v_1 + .. + a_{ij}v_j + .. + a_{i(ns)}v_{ns} + a_{i0} = 0, \quad \text{for} \quad 1 \le i \le nc$$

where ns is the number of star variables and nc ($0 \le nc \le ns$) is the number of relations (or constraints). A preferred equivalent formulation of the set L is given by partitioning the set of star variables V into two subsets V^{ind}, V^{dep} of independent and dependent star variables, respectively, and expressing the latter as linear combinations of the former:

$$l_i \equiv v_i^{dep} = a'_{i1}v_1^{ind} + .. + a'_{ij}v_j^{ind} + .. + a'_{i(ni)}v_{ni}^{ind} + a'_{i0}, \quad \text{for} \quad 1 \le i \le nc$$

where ni and nc are the number of independent and dependent star variables, respectively.

The definition of V restricts the allowed values for the star variables to natural numbers, $\forall k \in [1, ns] : v_k \in \mathcal{N}$. Consequently, the set of linear relations L is only well-defined when the involved variables take natural numbers as values. This also implies that some of the star variables, even the independent ones, may be implicitly constrained to a smaller range inside the natural numbers (e.g. $v_k \ge z, z \in \mathcal{N}$; v_k always odd; v_k always even; etc..). Moreover, the coefficients a_{ij} (or a'_{ij}) of the linear relations will always be rational numbers.

Let $\tilde{R} = (R, V, \mathcal{T}, L)$ be an ARE, the language $L(\tilde{R})$ represented by $\tilde{R}$ is defined as the set of strings α from Σ^* such that $\alpha \in L(R)$ and there exists a parse of α by R in which *the star instances $SI_\alpha(V)$ satisfy*[8] *all the linear constraints in L.*

The AREs permit to describe a class of context-sensitive languages by imposing a set of rules that constrain the language of a regular super-set. A very simple example is the language of rectangles $\{ a^m b^n a^m b^n \mid m, n \ge 1 \}$, which is well-known to be context-sensitive (see grammar CSG_1 in Appendix 1), and which is described by the ARE $\tilde{R}_1 = (R_1, V_1, \mathcal{T}_1, L_1)$, where

$$\begin{aligned}
R_1 &= a^*b^*a^*b^* \\
V_1 &= \{v_1, v_2, v_3, v_4\} \\
\mathcal{T}_1 &= (V_1 \cup r, \{(r, v_1), (r, v_2), (r, v_3), (r, v_4)\}, r) \\
L_1 &= \{v_3 = v_1, v_4 = v_2\} \\
&\quad \text{(i.e. } V^{ind} = \{v_1, v_2\}\ V^{dep} = \{v_3, v_4\} \text{ and } nc = 2).
\end{aligned}$$

However, quite more complex languages with an arbitrary level of star embedment and multiple linear constraints (even among stars at different levels of embedment) can be described as well by the ARE formalism. Consider, for instance, the ARE $\tilde{R}_2 = (R_2, V_2, \mathcal{T}_2, L_2)$ with

[8] Let the predicate $satisfy(SI_\alpha(V), L)$ denote this meaning.

$$
\begin{aligned}
R_2(V_2/*) &= (c^{v_1}(d^{v_2}b^{v_3})^{v_4}c^{v_5}a^{v_6}c^{v_7}(b^{v_8}d^{v_9})^{v_{10}}c^{v_{11}}e^{v_{12}})^{v_{13}} \quad \text{and} \\
L_2 &= \{v_{11} = v_1 + v_5 - v_7, \\
&\quad v_{12} = v_6, \\
&\quad v_2 = v_4 - 1, \\
&\quad v_3 = v_4 - 1, \\
&\quad v_8 = 0.5v_{10} + 0.5, \\
&\quad v_9 = 0.5v_{10} + 0.5\}
\end{aligned}
$$

The set of constraints L_2 (besides an unambiguity requirement for parsing R_2) implies that $v_2, v_3, v_8, v_9 \geq 1$; $v_4 \geq 2$; and v_{10} be always odd. Figures 5 and 7 show some examples that belong to the language represented by $\tilde{R}_2$, given an alphabet of graphical primitives $\{a, b, c, d, e\}$. The context-sensitive grammar CSG_2 in Appendix 1 generates[9] $L(\tilde{R}_2)$. The reader is encouraged to compare the compact and descriptive representation provided by the ARE $\tilde{R}_2$ with the obscure context-sensitive grammar CSG_2, that comprises 79 rules.

The following question naturally arises: *Can all the context-sensitive languages be represented by AREs?* The answer is *not*. The context-sensitive grammars CSG_3 and CSG_4, that generate the languages $\{a^k \mid k = 2^i \ \wedge \ i \geq 1\}$ and $\{xx \mid x \in (0+1)^+\}$, respectively, are two counter-examples. The former language is not describable because AREs can only filter the range of values of the star variables through linear relations (and without external parameters, such as i). The latter is not representable due to the fact that the constraints in AREs only involve the number of instances of certain subpatterns in a string but not the relations among the subpatterns theirselves.

In fact, $L(CSG_4)$ corresponds to the pattern language xx over the binary alphabet $\Sigma = \{0, 1\}$, where the variable x stands for any string in Σ^+ [4]. The ARE $(0+1)^{v_1}(0+1)^{v_2}$ with $\{v_2 = v_1\}$ cannot express that the substrings associated with the instances of the operands of the stars denoted by v_1 and v_2 are identical. Even if the equivalence rule $(0+1)^* = (0^*1)^*0^*$ is applied before, the resulting ARE $(0^{v_1}1)^{v_2}0^{v_3}(0^{v_4}1)^{v_5}0^{v_6}$ with $\{v_5 = v_2;\ v_6 = v_3\}$ (which assures that the number of 1's is the same in both substrings) is insufficient to describe the above pattern language, and, furthermore, the hypothetic constraint $v_4 = v_1$ cannot be established using the ARE formalism, because v_4 and v_1 do not share a common housing ancestor (see next subsection). This simple example shows that pattern languages [4] are not covered by AREs. On the other hand, it is obvious that the class of pattern languages does not cover the languages represented by AREs. For example, the language of rectangles $L(\tilde{R}_1)$ and the context-free language $\{0^{v_1}1^{v_2}0^{v_3} \mid v_2 = v_1 + v_3\}$ cannot be described by any pattern language.

[9]In order to exclude the empty string and simplify some rules, the conditions $v_{13}, v_6 \geq 1$ have been added.

7.2. String recognition through augmented regular expressions.

The recognition of a string s as belonging to a language $L(\tilde{R})$ can be clearly divided in two steps: parsing s by R, and if success, checking the satisfaction of constraints L by the star instances $SI_s(V)$ that result from the parse. If R is *unambiguous*, a unique parse and set of star instances $SI_s(V)$ is possible for each $s \in L(R)$ (as given by Algorithms 3 and 4), and therefore a single satisfaction problem must be analysed to test whether $s \in L(\tilde{R})$.

Algorithm 5 below has been designed to evaluate the predicate $satisfy(SI_s(V), L)$ given a set of star instances, previously recorded in a successful parse, and a set of linear constraints among the star variables. It provides the second step in recognizing a string through an ARE. Most of the functions and data structures that are required by Algorithm 5 will also be used by the constraint induction algorithm described in next section, so it is worthwhile to get some insight of them.

First of all, it must be noted that every valid constraint can only involve a set of star variables which share a common structure of instances, i.e. the number of instances of each of them is the same, and these can be grouped, one instance of each variable, in rows, one row for each cycle of the instances of a common *selected* ancestor. In principle, this would mean that the set of related star variables should be brothers in the star tree $\mathcal{T}$ (and their father being the common ancestor).

However, this is not exactly the case. It could happen that the values of the assigned instances of a certain son were constant for each instance of its father. In such a case, one may assume that a unique value is associated with the father's instance and, furthermore, both values (of the father and son instances) might be related by some constraint. In other words, regarding the star instances, the son would be promoted to a lower level in the tree and share with its father the instance structure determined by the instances of its grand-father; we could say that, with respect to the promoted son, the father is a *degenerated* ancestor and the grand-father is the *housing* ancestor.

Moreover, this promotion process could continue until a *non-degenerated* ancestor were found, for which the instances of the promoted variable would not have a common value for each instance of the ancestor, or until a default node were reached as *housing* ancestor. This default node can be the root node r of the star tree or a selected ancestor (e.g. a *housing* ancestor that is shared with other star variables, as it is explained next). Each time a star variable is promoted to a lower level, all of its redundant instances must be collapsed into a single one in order to keep the common structure of instances. See the procedure *determine_ancestor_and_coefficients* which implements the process described.

In summary, the constraints in L can only be defined among (and satisfied by) star variables which share a common *housing* ancestor. Let the constraint l_i in L ($1 \leq i \leq L.nc$) be expressed as

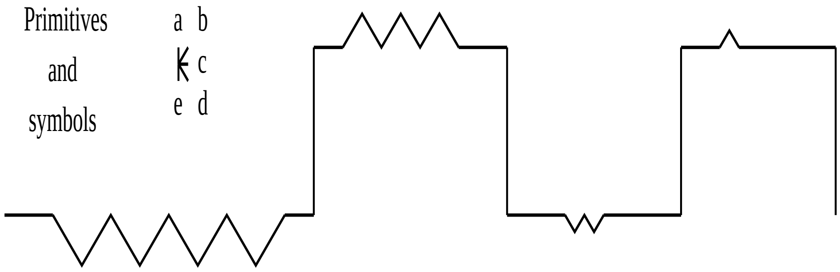
$$v_i^{dep} \;=\; a_{i1}v_1^{ind} + \ldots + a_{il}v_l^{ind} + a_{i0},$$

where the set of independent variables fulfills the restriction that $\forall j \in [1..l]: \; a_{ij} \neq 0$.

In order to test l_i the instances of the dependent star variable v_i^{dep}, obtained in the parse of s, are analysed. If there is no actual exponent of v_i^{dep}, the constraint is considered to be met. Otherwise, its housing ancestor must be the deepest common ancestor of the star variables $\{v_i^{dep}, v_1^{ind}, \ldots, v_l^{ind}\}$, i.e. the first common ancestor going from each of these nodes to the root of $\mathcal{T}$. To check the linear constraint l_i it is mandatory that the instances of all the star variables involved in the relationship can be arranged in the structure of instances caused by the deepest common ancestor. Consequently, if any of them has a housing (non-degenerated) ancestor that is deeper than the common ancestor, it means that a shared structure of instances is not available for the string s, and therefore, the constraint is considered to be violated. In the particular case of a star variable of an ARE having always a constant value ($v_i^{dep} = a_{i0}$), no matter its level in $\mathcal{T}$, its housing ancestor will be the root node, and obviously, all of its actual exponents must be collapsed to the value a_{i0} to verify the constraint.

Finally, when the housing ancestor of all the star variables in l_i is their deepest common ancestor, the constraint is tested on all the actual exponents of v_i^{dep}. To this end, these instances are arranged in a column vector B, whereas the corresponding instances of the involved independent variables are orderly put as columns in a matrix A, together with an all-1's column associated with the constant term of the constraint (see the procedure *build_system_matrix*). Then, it suffices to test $A \cdot X = B$, where X is the vector of coefficients in the right hand side of the constraint.

Consider the example of Figure 5. Given the constraints L_2 and the star instances displayed (for the string $s1$), Algorithm 5 would set v_{13} as housed descendent of the root node r, and the rest of star variables of V_2 as housed descendents of v_{13}. In the main loop, the six constraints of L_2 would be checked. The first one, $v_{11} = v_1 + v_5 - v_7$, would lead to the successful test of the equality $A \cdot X = B$ shown in Figure 6. The rest of constraints would be verified in a similar manner. Therefore, the string s_1 of Fig.5 would be accepted as belonging to $L(\tilde{R}_2)$.



$$
\begin{array}{lcl}
s_1 & = & c^5 d^3 b^3 d^3 b^3 d^3 b^3 d^3 b^3 c^3 a^{10} c^3 b^2 d^2 b^2 d^2 b^2 d^2 c^5 e^{10} c^6 dbdbc^8 a^{10} c^4 bdc^{10} e^{10} \\
R_2(V_2/*) & = & (c^{v_1} (d^{v_2} b^{v_3})^{v_4} c^{v_5} a^{v_6} c^{v_7} (b^{v_8} d^{v_9})^{v_{10}} c^{v_{11}} e^{v_{12}})^{v_{13}} \\
SI_{s_1}(v_{13}) & = & (\ (2)^{\{-1,-1\}}\ )
\end{array}
$$

$$
\begin{array}{lcl lcl}
SI_{s_1}(v_1) & = & (\ (5\ 6)^{\{1,1\}}\ ) & SI_{s_1}(v_7) & = & (\ (3\ 4)^{\{1,1\}}\ ) \\
SI_{s_1}(v_5) & = & (\ (3\ 8)^{\{1,1\}}\ ) & SI_{s_1}(v_{11}) & = & (\ (5\ 10)^{\{1,1\}}\ ) \\
SI_{s_1}(v_6) & = & (\ (10\ 10)^{\{1,1\}}\ ) & SI_{s_1}(v_{12}) & = & (\ (10\ 10)^{\{1,1\}}\ ) \\
SI_{s_1}(v_4) & = & (\ (4\ 2)^{\{1,1\}}\ ) & SI_{s_1}(v_{10}) & = & (\ (3\ 1)^{\{1,1\}}\ ) \\
SI_{s_1}(v_2) & = & (\ (3\ 3\ 3\ 3)^{\{1,1\}}\ \ (1\ 1)^{\{1,2\}} & SI_{s_1}(v_8) & = & (\ (2\ 2\ 2)^{\{1,1\}}\ \ (1)^{\{1,2\}} \\
SI_{s_1}(v_3) & = & (\ (3\ 3\ 3\ 3)^{\{1,1\}}\ \ (1\ 1)^{\{1,2\}} & SI_{s_1}(v_9) & = & (\ (2\ 2\ 2)^{\{1,1\}}\ \ (1)^{\{1,2\}}
\end{array}
$$

Fig. 5 *An example of pattern recognized by the ARE $\tilde{R}_2$ with its corresponding star instances.*

$$
\begin{array}{l}
\\
\text{1st cycle of instance } v_{13} = 2 \text{ in } s_1 \\
\text{2nd cycle of instance } v_{13} = 2 \text{ in } s_1
\end{array}
\quad
\left[\begin{array}{cccc} & v_1 & v_5 & v_7 \\ 1 & 5 & 3 & 3 \\ 1 & 6 & 8 & 4 \end{array} \right]
\left[\begin{array}{c} 0 \\ 1 \\ 1 \\ -1 \end{array} \right]
=
\left[\begin{array}{c} v_{11} \\ 5 \\ 10 \end{array} \right]
$$

Fig. 6 *Verification of the constraint $v_{11} = v_1 + v_5 - v_7$ through the matrix product $A \cdot X = B$ (the top row of the displayed A and B is just for labeling purposes).*

ALGORITHM 5: *Evaluates the predicate* $satisfy(SI_s(V), L)$.

Inputs: A set V of star variables, that are organized in a star tree $\mathcal{T}$.
A set SI containing the lists of instances of the star variables V resulting from the (successful) parse of a string s.
A set L of linear relations among the star variables in V.

Outputs: A boolean variable $satisfy\_constraints$ whose value will be TRUE if and only if the star instances SI satisfy all the constraints in L.

begin

$ASI[1]$:= SI { The set SI is placed in the first position of an array ASI just for compatibility with the arguments of the functions determine_ancestor_and_coefficients and sum_of_star_exponents. }

{ Mark the star variables for which the housing ancestor has not been computed }
for v_id **in** $[1 .. |V|]$ **do**
  $housing\_ancestor\_of\_node\ [v\_id] := -1$
end_for

{ Check the constraints L }
$i := 1;$ $satisfy\_constraints := TRUE$

while $i \leq L.nc$ **and** $satisfy\_constraints$ **do** { check each constraint }

  $vdep\_id :=$ element_of_list $(L.V^{dep}, i)$ { get the i-th dependent variable }
  $linear\_combination[vdep\_id] :=$ element_of_list $(L.right\_hand\_sides, i)$
  $X := linear\_combination[vdep\_id].coefficients$
  $list\_of\_indep\_variables := linear\_combination[vdep\_id].independent\_variables$

  $common\_housing\_anc\_id :=$ deepest_common_ancestor $(\mathcal{T}, vdep\_id,$
    $list\_of\_indep\_variables)$

  $node :=$ startree_node_identified_by $(vdep\_id)$
  determine_ancestor_and_coefficients $(\mathcal{T}, node, vdep\_id, ASI, 1, common\_housing\_anc\_id,$
    $anc\_id, coef[vdep\_id])$ { returns anc_id, the identifier of the housing ancestor of $node$, and $coef[vdep\_id]$, the coefficients given by the exponents of v_{vdep_id} for each instance of v_{anc_id} in $ASI[1]$ }
  $housing\_ancestor\_of\_node\ [vdep\_id] := anc\_id$

  if number_of_actual_exponents $(coef[vdep\_id]) > 0$ **then**

    if $housing\_ancestor\_of\_node\ [vdep\_id] \neq common\_housing\_anc\_id$ **then**
      $satisfy\_constraints := FALSE$

    else
      $j := 1;$ $l := length\ (list\_of\_indep\_variables)$
      while $j \leq l$ **and** $satisfy\_constraints$ **do**
        $v\_id :=$ element_of_list $(list\_of\_indep\_variables, j)$
        if $housing\_ancestor\_of\_node\ [v\_id] < 0$ **then**
          $node :=$ startree_node_identified_by (v_id)
          determine_ancestor_and_coefficients $(\mathcal{T}, node, v\_id, ASI, 1,$
            $common\_housing\_anc\_id, anc\_id, coef[v\_id])$
            { returns anc_id, the identifier of the housing ancestor of $node$, and $coef[v\_id]$, the coefficients given by the exponents of v_{v_id} for each instance of v_{anc_id} in $ASI[1]$ }
          $housing\_ancestor\_of\_node\ [v\_id] := anc\_id$
        end_if

if $housing\_ancestor\_of\_node\ [v\_id] \neq common\_housing\_anc\_id$
then
$satisfy\_constraints := FALSE$
end_if
$j := j + 1$
end_while

if $satisfy\_constraints$ **then**

$row\_max\_number :=$ sum_of_star_exponents $(ASI,$
$common\_housing\_anc\_id, 1)$
$column\_number := l + 1$
build_system_matrix $(vdep\_id, list\_of\_indep\_variables,$
$coef, row\_max\_number, column\_number, row\_actual\_number, B, A)$
{ vector B is given by the star instances of v_{vdep_id}, and the system matrix A is given by the star instances of the independent variables from which v_{vdep_id} depends, and after removing the rows for which the exponent of v_{vdep_id} is unassigned. }

$k := 1$
while $k \leq row\_actual\_number$ **and** $satisfy\_constraints$ **do**
{ check each instance of v_{vdep_id} }
$result :=$ dot_product $(X, A[k], column\_number)$
if $result \neq B[k]$ **then**
$satisfy\_constraints := FALSE$
end_if
$k := k + 1$
end_while

end_if

end_if
end_if

$i := i + 1$
end_while

{ $satisfy\_constraints \equiv satisfy(SI_s(V), L)$ }

end_algorithm

procedure *determine_ancestor_and_coefficients* ($\mathcal{T}$, son, son_id, ASI, $number\_of\_strings$, $default\_housing\_anc\_id$, anc_id, $coefficients$)

input arguments:

$\mathcal{T}$ is the star tree associated with the RE used for parsing.

son is a node of the star tree $\mathcal{T}$.

son_id is the identifier of son.

ASI is an array of sets containing the star instances resulting from the parses of a set of strings by a RE.

$number\_of\_strings$ is the size of the array ASI.

$default\_housing\_anc\_id$ is the identifier of a node of $\mathcal{T}$ to be chosen as housing ancestor of son when its nearer ancestors are found to be degenerated.

output arguments:

anc_id is the node identifier of the housing ancestor of son (according to the star instances in ASI and the limit given by $default\_housing\_anc\_id$).

$coefficients$ is a vector containing the exponents corresponding to the "non-redundant" instances of the star variable v_{son_id} throughout the array ASI.

begin

$node := son$; $node\_id := son\_id$
$current\_ancestor :=$ father_of ($node$)
$anc\_id :=$ node_identifier_of ($current\_ancestor$)
$current_ancestor_is_father :=$TRUE
$found :=$FALSE

while not $found$ **do**
    $redundancy :=$TRUE
    $i := 1$; $i\_prime := 1$

    for $nstring$ **in** $[1 .. number\_of\_strings]$ **do**
        $nlists :=$ number_of_lists ($ASI[nstring], node\_id$)
        for j **in** $[1 .. nlists]$ **do**
            $nelements :=$ number_of_elements ($ASI[nstring], node\_id, j$)
            $still_no_actual_exponent_in_list :=$TRUE
            $common\_exponent := -1$
            for k **in** $[1 .. nelements]$ **do**
                { j and k successively point to the exponents corresponding to the star instances of v_{node_id} during the parse of the string numbered $nstring$}
                $exponent :=$ read_exponent ($ASI[nstring], node\_id, j, k$)
                { determine the corresponding coefficient of v_{son_id} }
                if $current\_ancestor\_is\_father$ **then**
                    $coefficient := exponent$ { the assigned value comes directly from $ASI[nstring][son\_id]$ }

```
                    else    { current ancestor is not the father of son}
                        if exponent > 0 then
                            { find the common exponent of the instances of v_son_id derived
                             from this instance of v_node_id }
                            coefficient := -1
                            for e in [1 .. exponent] do
                              if coefficient < 0 and
                                      coefficients_backup[i_prime] >= 0 then
                                coefficient := coefficients_backup[i_prime]
                              end_if
                              i_prime := i_prime + 1
                            end_for
                        else    { no instance of v_son_id is derived }
                            coefficient := -1
                        end_if
                    end_if
                    coefficients[i] := coefficient
                    i := i + 1
                    { check the redundancy of the coefficients for the current list }
                    if still_no_actual_exponent_in_list then
                        if coefficient >= 0 then
                            still_no_actual_exponent_in_list :=FALSE
                            common_exponent := coefficient
                        end_if
                    else    {already found an exponent >= 0 in current list}
                        if coefficient >= 0 and coefficient ≠ common_exponent then
                            redundancy :=FALSE
                        end_if
                    end_if
                end_for
            end_for
        end_for
        if redundancy and anc_id ≠ default_housing_anc_id then
              { the current ancestor is degenerated }
            copy_integer_vector (coefficients, coefficients_backup, i - 1)
            node := current_ancestor;   node_id := node_identifier_of (node)
            current_ancestor := father_of (node)
            anc_id := node_identifier_of (current_ancestor)
            current_ancestor_is_father :=FALSE
        else    { the housing ancestor of son has been determined }
            found :=TRUE
        end_if
    end_while
    end_procedure
```

function *sum_of_star_exponents* $(ASI, father\_id, number\_of\_strings)$ **returns** integer
input arguments:

ASI is an array of sets containing the star instances resulting from the parses of a set of strings by a RE

$father\_id$ is the identifier of a star variable (or 0 for the root node).

$number\_of\_strings$ is the size of the array ASI

returned value:

returns either $number\_of\_strings$ (when $father\_id = 0$, i.e. the father is the root node r of the star tree) or the sum of the exponents corresponding to the instances of the star variable v_{father_id} throughout the array ASI; the returned value is the potential (maximal) number of star instances of the sons and housed descendents of v_{father_id} (or r, if $father\_id = 0$) resulting from the parses.

begin

if $father\_id = 0$ **then** { $father\_id$ corresponds to the root node }
    $sum := number\_of\_strings$ { count one instance for each string }

else { $father\_id$ is the identifier of a star variable }
    $sum := 0$
    for $nstring$ **in** $[1 \,..\, number\_of\_strings]$ **do**
        $nlists :=$ number_of_lists $(ASI[nstring], father\_id)$
        for j **in** $[1 \,..\, nlists]$ **do**
            $nelements :=$ number_of_elements $(ASI[nstring], father\_id, j)$
            for k **in** $[1 \,..\, nelements]$ **do**
                { j and k successively point to the exponents corresponding to the star instances of v_{father_id} during the parse of the string numbered $nstring$}
                $exponent :=$ read_exponent $(ASI[nstring], father\_id, j, k)$
                if $exponent \geq 0$ **then** { an actual exponent was assigned to this element (i.e. the star variable v_{father_id} was instantiated) }
                    $sum := sum + exponent$
                end_if
            end_for
        end_for
    end_for

end_if

return sum

end_function

procedure *build_system_matrix* ($star\_id, list\_of\_indep\_star\_variables, coef,$
$row\_max\_number, column\_max\_number, row\_actual\_number, B, A$)

input arguments:

$star\_id$ is the identifier of a star variable.

$list\_of\_indep\_star\_variables$ is a list of identifiers of star variables (v_id)

$coef$ is an array containing the vector of coefficients for each star variable.

row_max_number is the size of both $coef[star\_id]$ and $coef[v\_id]$,
for all v_id in the input list.

col_max_number is the number of columns in the returned matrix A.

output arguments:

row_actual_number is the number of rows in the returned matrix A and the number of elements in the returned vector B.

B is a vector which contains the coefficients of the star variable v_{star_id} that correspond to actual exponents.

A is a matrix with a column of 1s and some columns containing the coefficients of the star variables in the input list, but only with the rows for which the coefficient of v_{star_id} is an actual exponent.

begin

$nrow := 0$

for i **in** $[1 .. row\_max\_number]$ **do**
  $coefficient := coef[star\_id][i]$
  { for each instance of star variable v_{star_id} write the corresponding row of coefficients of the independent variables }
  if $coefficient \geq 0$ **then**
    $nrow := nrow + 1$
    $B[nrow] := coefficient$
    $A[nrow, 1] := 1$
    $current\_elem :=$ first_element ($list\_of\_indep\_star\_variables$)
    for j **in** $[2 .. column\_max\_number]$ **do**
      $v\_id :=$ read_value ($list\_of\_indep\_star\_variables, current\_elem$)
      $exponent := coef[v\_id][i]$
      if $exponent < 0$ **then**
        $exponent := 0$ { convert unassigned exponents into zero exponents for seeking the linear relations}
      end_if
      $A[nrow, j] := exponent$
      $current\_elem :=$ next_element ($list\_of\_indep\_star\_variables$)
    end_for
  end_if
end_for

$row\_actual\_number := nrow$

end_procedure

8. Inducing the constraints of an ARE from recorded star instances.

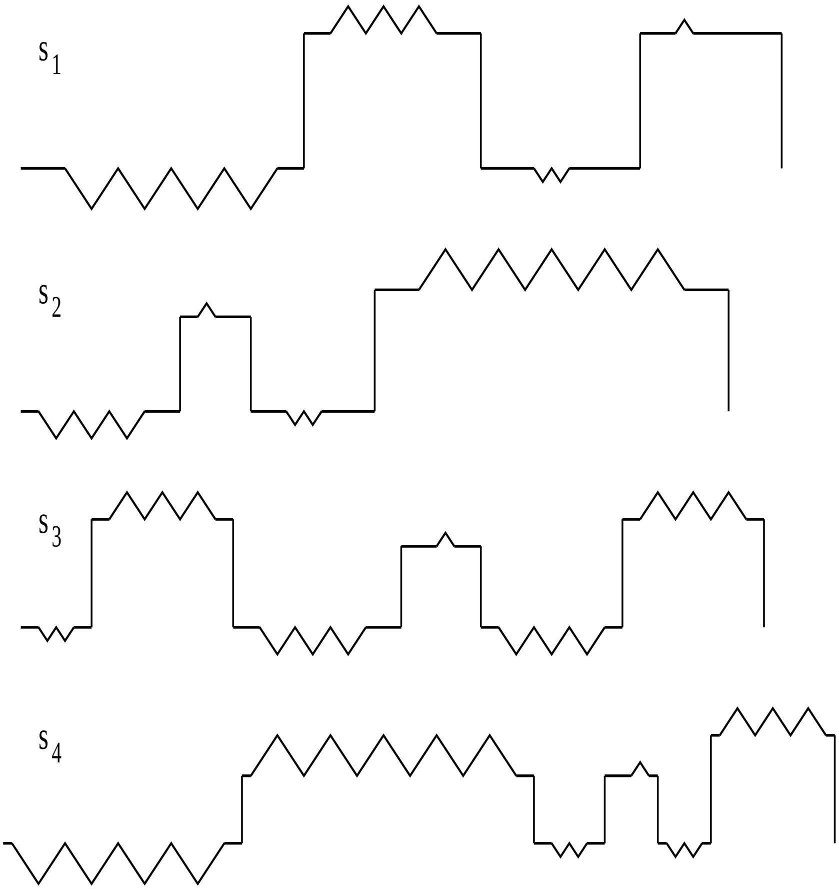
Now, let us consider the problem of inducing or learning AREs. A possible approach is to split the process in two main steps: inferring the underlying RE, and afterwards, inducing the constraints that bear the context sensitivity of the language. For the first step, some regular grammatical inference method is required, followed by the FSA to RE mapping. For the second step, Algorithm 6 in this section is proposed as the basic method, provided that it needs some previous tasks, i.e. building the star tree and parsing a set of example strings.

Let the target ARE be $\tilde{R} = (R, V, \mathcal{T}, L)$. It is assumed as precondition of Algorithm 6 that R, V and $\mathcal{T}$ are already known, and that a certain number of strings have been parsed by R giving rise to an array of sets of star instances ASI. Algorithm 6 tries to discover those regularities that consistently appear throughout the star instances and it expresses them as a set of linear constraints L among the star variables V.

Algorithm 6 is based on establishing a tree of linear systems according to the housing ancestor concept. Each *housing* ancestor will have its own partition of independent and dependent star variables among its *housed* descendents. To construct this partition, each ancestor node of the star tree $\mathcal{T}$ keeps track of its housed descendents that have been determined to be independent. All the variables of the star tree are visited by levels, and for each one (say v_j), its housing ancestor (say v_k) is found and a vector of its non-redundant instances is formed (by calling procedure *determine_ancestor_and_coefficients* with the root node as the default housing ancestor). Then a matrix is built that contains the instances of the independent housed descendents of v_k (procedure *build_system_matrix*). Initially, the number of columns of the matrix is the number of independent housed descendents plus one (an all-1's column) and the number of rows is the number of (non-redundant) actual exponents of v_j, which is bounded by the number of parsed strings if v_k is the root node or by the total sum of exponents of v_k otherwise (see function *sum_of_star_exponents*). Next, the rank of the matrix is evaluated and any linearly dependent column is removed. Finally, it is determined whether the vector of actual exponents of v_j is linearly dependent on the column vectors of the system matrix. If it is, the corresponding linear system can be solved to find the constraint coefficients, and the new constraint is appended to L; otherwise, v_j is included in the list of independent housed descendents of v_k.

Let us illustrate the method with the ARE example $\tilde{R}_2$ of section 7.1. and the set of strings displayed in Figure 7, that belong to $L(\tilde{R}_2)$.

First, the sons of the root node are processed; in this case, v_{13} is the only one and it is found independent (since its instances $[2\ 2\ 3\ 3]^T$ are not constant). Then, the sons of v_{13} are visited. It turns out that v_{13} is the housing ancestor of all of its sons. The star variables v_1, v_4, v_5, v_6, v_7 and v_{10} are successively found independent. At this point, the instances of v_{11} are stored in vector B to be analysed, while the matrix A contains the instances of the independent housing descendents of v_{13} already processed. Figure 8 displays the corresponding linear system. It turns out that vector B is a linear combination of the columns of A, and solving the system yields ($v_{11} = v_1 + v_5 - v_7$). This constraint is put into L. Then, the last son v_{12} is visited and the second constraint ($v_{12} = v_6$) is obtained similarly. Next, v_2 and v_3 (the sons of v_4) are processed, v_{13} is determined as their housing ancestor (since their instances are constant for each instance of v_4), and both are found dependent according to ($v_2 = v_4 - 1$, $v_3 = v_4 - 1$). Finally, v_8 and v_9 (the sons of v_{10}) are also housed by v_{13} and the analysis of their instances gives rise to the last constraints ($v_8 = 0.5v_{10} + 0.5$, $v_9 = 0.5v_{10} + 0.5$). At the end, the inferred set of constraints L coincides with the target L_2 of $\tilde{R}_2$.



$$
\begin{aligned}
s_1 &= c^5d^3b^3d^3b^3d^3b^3d^3b^3c^3a^{10}c^3b^2d^2b^2d^2b^2d^2c^5e^{10}c^6dbdbc^8a^{10}c^4bdc^{10}e^{10} \\
s_2 &= c^2d^2b^2d^2b^2d^2b^2c^4a^7c^2bdc^4e^7c^4dbdbc^6a^9c^5b^3d^3b^3d^3b^3d^3b^3d^3b^3d^3c^5e^9 \\
s_3 &= c^2dbdbc^2a^8c^2b^2d^2b^2d^2b^2d^2c^2e^8c^3d^2b^2d^2b^2d^2b^2c^4a^6c^4bdc^3e^6 \\
& c^2d^2b^2d^2b^2d^2b^2c^2a^8c^2b^2d^2b^2d^2b^2d^2c^2e^8 \\
s_4 &= cd^3b^3d^3b^3d^3b^3d^3b^3c^2a^5cb^3d^3b^3d^3b^3d^3b^3d^3b^3d^3c^2e^5c^2dbdbc^2a^5c^3bdce^5 \\
& cdbdbca^8cb^2d^2b^2d^2b^2d^2ce^8
\end{aligned}
$$

Fig. 7 *Four example strings from $L(\tilde{R}_2)$ which are used to infer the constraints L_2.*

$$
\begin{array}{l}
\\
\text{1st cycle of instance } v_{13} = 2 \text{ in } s_1 \\
\text{2nd cycle} \\
\text{1st cycle of instance } v_{13} = 2 \text{ in } s_2 \\
\text{2nd cycle} \\
\text{1st cycle of instance } v_{13} = 3 \text{ in } s_3 \\
\text{2nd cycle} \\
\text{3rd cycle} \\
\text{1st cycle of instance } v_{13} = 3 \text{ in } s_4 \\
\text{2nd cycle} \\
\text{3rd cycle}
\end{array}
\left[\begin{array}{ccccccc}
 & v_1 & v_4 & v_5 & v_6 & v_7 & v_{10} \\
1 & 5 & 4 & 3 & 10 & 3 & 3 \\
1 & 6 & 2 & 8 & 10 & 4 & 1 \\
1 & 2 & 3 & 4 & 7 & 2 & 1 \\
1 & 4 & 2 & 6 & 9 & 5 & 5 \\
1 & 2 & 2 & 2 & 8 & 2 & 3 \\
1 & 3 & 3 & 4 & 6 & 4 & 1 \\
1 & 2 & 3 & 2 & 8 & 2 & 3 \\
1 & 1 & 4 & 2 & 5 & 1 & 5 \\
1 & 2 & 2 & 2 & 5 & 3 & 1 \\
1 & 1 & 2 & 1 & 8 & 1 & 3
\end{array}\right] X = \left[\begin{array}{c}
v_{11} \\ 5 \\ 10 \\ 4 \\ 5 \\ 2 \\ 3 \\ 2 \\ 2 \\ 1 \\ 1
\end{array}\right]
$$

Fig. 8 *The linear system $A \cdot X = B$ for the star variable v_{11} (the top row of the displayed A and B is just for labeling purposes). The solution is $X = [0\ 1\ 0\ 1\ 0\ \text{-}1\ 0]^T$.*

ALGORITHM 6: *Induces a set of linear relations that hold among the star variables throughout the instances resulting from the parse of a set of strings.*

Inputs: A set V of star variables, that are organized in a star tree $\mathcal{T}$.
An array of sets ASI containing the lists of instances of the star variables V resulting from the (successful) parses of a set of strings.
The size of the array ASI: $number\_of\_strings$.

Outputs: A set L of linear relations among the star variables in V, that is fully satisfied in the parses of all the strings from which ASI has been built.

begin

$L.nc := 0$ { Initialize the number of constraints (dependent star variables) in L }
$L.V^{dep} := \emptyset$ { Create an empty set of dependent star variables V^{dep} }
$L.right\_hand\_sides := \emptyset$ { Create an empty set of linear combinations associated with V^{dep} }
$max\_level :=$ deepest_level_of_tree $(\mathcal{T})$ { this is the level of the deepest node in $\mathcal{T}$ }

for l **in** $[0 \mathrel{..} max\_level - 1]$ **do** { visit the father nodes of $\mathcal{T}$ by levels }
  for i **in** $[1 \mathrel{..} nodes\_of\_level\_l\ (\mathcal{T}, l)]$ **do**
    $father :=$ node_i_of_level_l $(\mathcal{T}, l, i)$
    $father\_id :=$ node_identifier_of $(father)$
    $number\_of\_indep\_housed\_descendents\ [father\_id] := 0$
    $list\_of\_indep\_housed\_descendents\ [father\_id] := \emptyset$
    $max\_number\_of\_rows\_of\_system\_matrix\ [father\_id] :=$
      sum_of_star_exponents $(ASI, father\_id, number\_of\_strings)$

    for $nson$ **in** $[1 \mathrel{..}$ nsons_of $(father)]$ **do**
      $son :=$ son_of $(father, nson)$
      $son\_id :=$ node_identifier_of (son)

      determine_ancestor_and_coefficients $(\mathcal{T}, son, son\_id, ASI, number\_of\_strings, 0, anc\_id, coef[son\_id])$
        { returns anc_id, the identifier of the nearest non-degenerated ancestor of son, and $coef[son\_id]$, the coefficients given by the exponents of v_{son_id} for each instance of v_{anc_id} in ASI }

      $housing\_ancestor\_of\_node\ [son\_id] := anc\_id$
      $row\_max\_number := max\_number\_of\_rows\_of\_system\_matrix\ [anc\_id]$
      $column\_max\_number :=$
        $number\_of\_indep\_housed\_descendents\ [anc\_id] + 1$
      build_system_matrix $(son\_id, list\_of\_indep\_housed\_descendents[anc\_id], coef, row\_max\_number, column\_max\_number, row\_actual\_number, B, A)$
        { vector B is given by the star instances of v_{son_id}, and the system matrix A is given by the star instances of the independent housed descendents of v_{anc_id}, after removing the rows for which the exponent of v_{son_id} is unassigned. }

```
                    if  row_actual_number > 0  then
                        rankA := rank_of_matrix (A, row_actual_number, column_max_number)
                        if  rankA < column_max_number  then
                            remove_dependent_columns_from_system_matrix (A, rankA,
                                row_actual_number, column_actual_number, columns_mask)
                                { matrix A is reduced by removing (in increasing order) those
                                columns that are a linear combination of the previous ones,
                                column_actual_number = rankA, and columns_mask is a
                                binary mask that marks the selected columns }
                        else
                            column_actual_number := column_max_number
                            columns_mask := fully_marked_mask (column_max_number)
                        end_if
                        build_extended_system_matrix (A, B, row_actual_number,
                            column_actual_number, AB)         { matrix AB is built by
                            appending to matrix A a column with vector B }
                        rankAB := rank_of_matrix (AB, row_actual_number,
                                                  column_actual_number + 1)
                        if  rankAB < (column_actual_number + 1) then   { the appended
                            column is a linear combination of the columns of A }
                            solve_linear_system (A, B, row_actual_number,
                                                  column_actual_number, X)   { vector X
                                is obtained by solving the linear system A · X = B }
                            linear_combination[son_id].coefficients := X
                            linear_combination[son_id].independent_variables :=
                                extract_masked_list (list_of_indep_housed_descendents[anc_id],
                                column_max_number, column_actual_number, columns_mask)
                            linear_combination[son_id] := remove_indep_variables_with_coef_zero
                                                         (linear_combination[son_id])
                            L.nc := L.nc + 1   { the number of dependent star variables is
                                increased and the new constraint is appended to L }
                            L.V^dep := append (L.V^dep, son_id)   { i.e. v_son_id ∈ V^dep }
                            L.right_hand_sides := append (L.right_hand_sides,
                                                         linear_combination[son_id])
                        else    { the appended column is linearly independent }
                            number_of_indep_housed_descendents [anc_id] :=
                                number_of_indep_housed_descendents [anc_id] + 1
                            list_of_indep_housed_descendents [anc_id] :=
                                append (list_of_indep_housed_descendents[anc_id], son_id)
                        end_if
                    end_if
                end_for
            end_for
        end_for

        end_algorithm
```

9. Discussion

A powerful new representation class called *Augmented Regular Expressions* (AREs) has been proposed to describe, recognize and learn a class of context-sensitive string languages capable of expressing multiple and complex context constraints. However, it has been demonstrated that not all the context-sensitive languages are covered by AREs. In addition, it has been shown that AREs and pattern languages [4] are not comparable.

In order to represent (context-sensitive) pattern languages such as $L(CSG_4) = \{xx \mid x \in (0+1)^+\}$, the ARE concept should be extended by means of the definition of constraints (e.g. equality) among the substrings that result from instantiating the operands of the star-type subexpressions. This would require that the data structure produced by parsing a string by a RE would register not only the lists of star variable instances but also the associated lists of matched substrings. At first, this adaptation seems possible, and therefore, a more powerful formalism would be achieved.

There are other ways to improve the proposed representation, but at the expense of complicating the algorithms used for recognition and induction. We will mention here two such extensions:

- Using correlation and linear regression, instead of strict linear equations, to establish and test "soft" constraints among the star instances.
- Allowing the definition of (a limited class of) non-linear constraints as well, e.g. quadratic constraints.

The former extension would upgrade the flexibility and robustness of the model, and it would be interesting to cope with most pattern recognition problems, where there is noisy input data and/or an imperfect data segmentation. Alternatively, error-correcting procedures could be designed to recognize noisy instances of the ideal language strings. Concerning the latter extension, the introduction of non-linear constraints would permit to describe quite complex languages. However, inferring such models from string examples would be extremely hard due to the great number of combinations of variables and terms that could arise; moreover, the probability of inducing artificial constraints would grow.

Coming back to the given definition of AREs, it should be noted that the learning scheme proposed is not a method for identifying AREs from examples but a practical approach to infer data-consistent AREs. The effectiveness of the whole procedure depends strongly on the result of the regular grammatical inference step. The algorithm to be used in this step should be biased to return preferrably small deterministic FSA with a high level of generalization with respect to the sample. There are two reasons for this demand: to obtain the simplest consistent regular expression for parsing, and to be able to induce in the following phase the context constraints that limit the extension of the inferred language. The constraint induction could be impeded if the starting language, yielded by the regular GI step, were too restricted to the given examples (i.e. a kind of sample overfitting).

10. References

1. R. Alquezar and A. Sanfeliu, "Incremental grammatical inference from positive and negative data using unbiased finite state automata," in *Proc. of the IAPR Int. Workshop on Structural and Syntactic Pattern Recognition, SSPR'94, Nahariya, Israel, October 1994.*
2. R. Alquezar and A. Sanfeliu, "An algebraic framework to represent finite-state machines in single-layer recurrent neural networks," to appear in *Neural Computation* **7**, 1995.
3. D. Angluin, "Inductive inference of formal languages from positive data," *Information & Control* **45**, pp.117-135, 1980.
4. D. Angluin, "Finding patterns common to a set of strings," *J. Comput. System Science* **21**, pp.46-62, 1980.
5. D. Angluin, "A note on the number of queries needed to identify regular languages," *Information & Control* **51**, pp.76-87, 1981.
6. D. Angluin and C.H. Smith, "Inductive inference: theory and methods," *ACM Computing Survey* **15** (3), pp.237-269, 1983.
7. D.N. Arden, "Delay logic and finite state machines," in *Proc. Second Ann. Symp. on Switching Theory and Logical Design, October 1961*, pp.133-151.
8. S.M. Chou and K.S.Fu, "Inference for transition network grammars," in *Proc. of Int. Joint Conf. on Pattern Recognition, 3*, CA, 1976, pp.79-84.
9. P. Dupont, "Regular grammatical inference from positive and negative samples by genetic search: the GIG method," in *Grammatical Inference and Applications, Proc. of the Second Int. Colloquium, ICGI'94, Alicante, Spain, September 1994*, R.C.Carrasco and J.Oncina (eds.), Springer-Verlag, Lecture Notes in Artificial Intelligence 862, pp.236-245.
10. K.S. Fu, *Syntactic Pattern Recognition and Applications.* Prentice-Hall, New York, 1982.
11. P. Garcia, J. Oncina, "Learning general context-free grammars from positive structural samples and negative strings," *DSIC Research Report*, Universidad Politecnica de Valencia, Spain, 1993.
12. C.L. Giles, C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun, Y.C. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks," *Neural Computation* **4**, pp.393-405, 1992.
13. E.M. Gold, "Language identification in the limit," *Information & Control* **10**, pp.447-474, 1967.
14. E.M. Gold, "Complexity of automaton identification from given data," *Information and Control* **37**, pp.302-320, 1978.
15. J. Gregor, "Data-driven inductive inference of finite-state automata," *Int. J. of Pattern Recognition and Artificial Intelligence* **8** (1), pp.305-322, 1994.
16. J.E. Hopfcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading MA, 1979.
17. Z. Kohavi, *Switching and Finite Automata Theory*, (2nd edition). Tata McGraw-Hill, New Delhi, India, 1978.
18. M. Kudo, M.Shimbo, "Efficient regular grammatical inference techniques by the use of partial similarities and their logical relationships", *Pattern Recognition* **21** (4), pp.401-409, 1988.
19. E. Makinen, "Remarks on the structural grammatical inference problem for context-free grammars," *Information Processing Letters* **44**, pp.125-127, 1992.

20. A. Marron and K. Ko, "Identification of pattern languages from examples and queries," *Information and Computation* **74**, pp.91-112, 1987.
21. L. Miclet, "Grammatical inference," in *Syntatic and Structural Pattern Recognition: Theory and Applications*, H.Bunke and A.Sanfeliu, Eds., World Scientific, 1990.
22. J. Oncina, P. Garcia, "Identifying regular languages in polynomial time," in *Advances in Structural and Syntactic Pattern Recognition*, H. Bunke (ed.), World-Scientific, Singapore, 1992, pp.99-108.
23. T.W. Pao, J.W. Carr, "A solution of the syntactical induction-inference problem for regular languages," *Computer Languages* **3**, pp.53-64, 1978.
24. V. Radhakrishnan, G. Nagaraja, "Inference of even linear grammars and its application to picture description languages," *Pattern Recognition* **21** (1), pp.55-62, 1988.
25. H. Rulot, E. Vidal, "Modelling (sub)string-length-based constraints through a grammatical inference method," in *Pattern Recognition: Theory and Applications*, Devijver and Kittler (eds.), Springer-Verlag, 1987, pp.451-459.
26. Y. Sakakibara, "Efficient learning of context-free grammars from positive structural examples," *Information and Computation* **97**, pp.23-60, 1992.
27. A. Salomaa, *Formal Languages.* Academic Press, New York, 1973.
28. A. Sanfeliu and R. Alquezar, "Active grammatical inference: a new learning methodology," in *Proc. of the IAPR Int. Workshop on Structural and Syntactic Pattern Recognition, SSPR'94, Nahariya, Israel, October 1994.*
29. Y. Takada, "Grammatical inference for even linear languages based on control sets," *Information Processing Letters* **28** (4), pp.193-199, 1988.
30. Y. Takada, "A hierarchy of language families learnable by regular language learners," in *Grammatical Inference and Applications, Proc. of the Second Int. Colloquium, ICGI'94, Alicante, Spain, September 1994*, R.C.Carrasco and J.Oncina (eds.), Springer-Verlag, Lecture Notes in Artificial Intelligence 862, pp.16-24.
31. E. Vidal, "Grammatical inference: an introductory survey," in *Grammatical Inference and Applications, Proc. of the Second Int. Colloquium, ICGI'94, Alicante, Spain, September 1994*, R.C.Carrasco and J.Oncina (eds.), Springer-Verlag, Lecture Notes in Artificial Intelligence 862, pp.1-4.
32. W.A. Woods, "Transition networks grammars for natural language analysis," *CACM* **13**, pp.591-606, 1970.

Appendix 1. List of context-sensitive grammars referenced in the text.

Language $\{\ a^m b^n a^m b^n \mid m, n \geq 1\ \}$.

Grammar CSG_1:

1)	S	$\longrightarrow$	$abab$	2)	S	$\longrightarrow$	$aAbB[aC][bD]$	3)	A	$\longrightarrow$	aAE
4)	A	$\longrightarrow$	aF	5)	B	$\longrightarrow$	bBG	6)	B	$\longrightarrow$	bH
7)	$G[aC]$	$\longrightarrow$	$[aC]b$	8)	Gb	$\longrightarrow$	bb	9)	$G[bD]$	$\longrightarrow$	$b[bD]$
10)	$H[aC]$	$\longrightarrow$	$[aC]H$	11)	Hb	$\longrightarrow$	bH	12)	$H[bD]$	$\longrightarrow$	bb
13)	Eb	$\longrightarrow$	bE	14)	Ea	$\longrightarrow$	aa	15)	$E[aC]$	$\longrightarrow$	$a[aC]$
16)	Fb	$\longrightarrow$	bF	17)	Fa	$\longrightarrow$	aF	18)	$F[aC]$	$\longrightarrow$	aa

Language $\{\ (c^{v_1}(d^{v_2}b^{v_3})^{v_4}c^{v_5}a^{v_6}c^{v_7}(b^{v_8}d^{v_9})^{v_{10}}c^{v_{11}}e^{v_{12}})^+ \mid$
$v_{11} = v_1 + v_5 - v_7;\ v_{12} = v_6;\ v_2 = v_4 - 1;\ v_3 = v_4 - 1;$
$v_8 = 0.5v_{10} + 0.5;\ v_9 = 0.5v_{10} + 0.5;\ v_2, v_3, v_6, v_8, v_9 \geq 1\}$.

Grammar CSG_2:

1)	S	$\rightarrow$	AS	2)	S	$\rightarrow$	A	3)	A	$\rightarrow$	$EF[aG][eH]$
4)	F	$\rightarrow$	aFI	5)	F	$\rightarrow$	J	6)	$I[aG]$	$\rightarrow$	$[aG]I$
7)	$J[aG]$	$\rightarrow$	aG	8)	$I[eH]$	$\rightarrow$	$[eH]e$	9)	$G[eH]$	$\rightarrow$	Ge
10)	E	$\rightarrow$	$[LN]$	11)	E	$\rightarrow$	$[LN][cO]M$	12)	E	$\rightarrow$	$[cK]M[LO]$
13)	$[cK]$	$\rightarrow$	$c[cK]M$	14)	$[cK]$	$\rightarrow$	$[cN]$	15)	$[cO]$	$\rightarrow$	$c[cO]M$
16)	$[cO]$	$\rightarrow$	c	17)	$M[LO]$	$\rightarrow$	$[LO]M$	18)	$[cN][LO]$	$\rightarrow$	$c[LN]$
19)	$[LO]$	$\rightarrow$	$L[cO]M$	20)	$[LN]c$	$\rightarrow$	$L[cN]$	21)	$[cN]c$	$\rightarrow$	$c[cN]$
22)	$[cN]L$	$\rightarrow$	$c[LN]$	23)	Ma	$\rightarrow$	aM	24)	$[cN]a$	$\rightarrow$	$c[aN]$
25)	$[LN]a$	$\rightarrow$	$L[aN]$	26)	$[aN]a$	$\rightarrow$	$a[aN]$	27)	G	$\rightarrow$	$[PQR]$
28)	$M[PQR]$	$\rightarrow$	$M[QR]$	29)	$M[PQR]$	$\rightarrow$	$[cP][QR]$	30)	$[aN][PQR]$	$\rightarrow$	aQ
31)	$M[cP]$	$\rightarrow$	$[cP]M$	32)	$M[cP]$	$\rightarrow$	$[cP]c$	33)	Mc	$\rightarrow$	cM
34)	$[aN][cP]$	$\rightarrow$	$a[cN]$	35)	$[aN]Q$	$\rightarrow$	$a[QN]$	36)	$[cN][QR]$	$\rightarrow$	cQ
37)	$M[QR]$	$\rightarrow$	$Q[Rc]$	38)	MQ	$\rightarrow$	QM	39)	$M[Rc]$	$\rightarrow$	$[Rc]c$
40)	$[cN]Q$	$\rightarrow$	$c[QN]$	41)	$[QN][Rc]$	$\rightarrow$	Qc				
42)	L	$\rightarrow$	$[VT][LW]$	43)	$[LW]$	$\rightarrow$	$T[UW]$	44)	$[LW]$	$\rightarrow$	$TL[UW]$
45)	L	$\rightarrow$	TU	46)	L	$\rightarrow$	TLU	47)	TU	$\rightarrow$	TXb
48)	$T[UW]$	$\rightarrow$	$T[XW]b$	49)	bU	$\rightarrow$	Xbb	50)	bX	$\rightarrow$	Xb
51)	$b[UW]$	$\rightarrow$	$[XW]bb$	52)	$b[XW]$	$\rightarrow$	$[XW]b$	53)	TX	$\rightarrow$	Ud
54)	$T[XW]$	$\rightarrow$	$[UW]d$	55)	dX	$\rightarrow$	Udd	56)	dU	$\rightarrow$	Ud
57)	$d[XW]$	$\rightarrow$	$[UW]dd$	58)	$d[UW]$	$\rightarrow$	$[UW]d$	59)	$[VT]U$	$\rightarrow$	$[VX]b$
60)	$[VT][UW]$	$\rightarrow$	db	61)	$[VX]X$	$\rightarrow$	$[dV]X$	62)	$[VX][XW]$	$\rightarrow$	$[dV][XW]$
63)	$[dV]X$	$\rightarrow$	$d[dV]$	64)	$[dV][XW]$	$\rightarrow$	dd				
65)	Q	$\rightarrow$	$[VT']U'$	66)	Q	$\rightarrow$	$[VT']T'Q'U'$	67)	Q'	$\rightarrow$	$T'T'Q'U'$
68)	Q'	$\rightarrow$	$T'U'$	69)	$T'U'$	$\rightarrow$	$T'X'd$	70)	dU'	$\rightarrow$	$X'dd$
71)	dX'	$\rightarrow$	$X'd$	72)	$T'X'$	$\rightarrow$	$U'b$	73)	bX'	$\rightarrow$	$U'bb$
74)	bU'	$\rightarrow$	$U'b$	75)	$[VT']U'$	$\rightarrow$	$[VX']d$	76)	$[VX']X'$	$\rightarrow$	$[bV']X'$
77)	$[VX']d$	$\rightarrow$	bd	78)	$[bV']X'$	$\rightarrow$	$b[bV']$	79)	$[bV']d$	$\rightarrow$	bd

Rules	$3-9$	implement the constraint	$v_{12} = v_6$
"	$10-41$	"	$v_{11} = v_1 + v_5 - v_7$
"	$42-64$	implement the constraints	$v_2 = v_4 - 1;\ v_3 = v_4 - 1$
"	$65-79$	"	$v_8 = 0.5v_{10} + 0.5;\ v_9 = 0.5v_{10} + 0.5$

Language $\{ a^k \mid k = 2^i \ \wedge \ i \geq 1 \}$.

Grammar CSG_3:

1)	S	$\rightarrow$	$[ACaB]$	2)	$[Ca]a$	$\rightarrow$	$aa[Ca]$	3)	$[Ca][aB]$	$\rightarrow$	$aa[CaB]$
4)	$[ACa]a$	$\rightarrow$	$[Aa]a[Ca]$	5)	$[ACa][aB]$	$\rightarrow$	$[Aa]a[CaB]$	6)	$[ACaB]$	$\rightarrow$	$[Aa][aCB]$
7)	$[CaB]$	$\rightarrow$	$a[aCB]$	8)	$[aCB]$	$\rightarrow$	$[aDB]$	9)	$[aCB]$	$\rightarrow$	$[aE]$
10)	$a[Da]$	$\rightarrow$	$[Da]a$	11)	$[aDB]$	$\rightarrow$	$[DaB]$	12)	$[Aa][Da]$	$\rightarrow$	$[ADa]a$
13)	$a[DaB]$	$\rightarrow$	$[Da][aB]$	14)	$[Aa][DaB]$	$\rightarrow$	$[ADa][aB]$	15)	$[ADa]$	$\rightarrow$	$[ACa]$
16)	$a[Ea]$	$\rightarrow$	$[Ea]a$	17)	$[aE]$	$\rightarrow$	$[Ea]$	18)	$[Aa][Ea]$	$\rightarrow$	$[AEa]a$
19)	$[AEa]$	$\rightarrow$	a								

Language $\{xx \mid x \in (0+1)^+\}$.

Grammar CSG_4:

1)	S	$\rightarrow$	$0E[CB]$	2)	S	$\rightarrow$	$0[FB]$	3)	S	$\rightarrow$	$1E[DB]$
4)	s	$\rightarrow$	$1[GB]$	5)	E	$\rightarrow$	$0EC$	6)	E	$\rightarrow$	$1ED$
7)	E	$\rightarrow$	$0F$	8)	E	$\rightarrow$	$1G$	9)	$C[0B]$	$\rightarrow$	$0[CB]$
10)	$C0$	$\rightarrow$	$0C$	11)	$C[1B]$	$\rightarrow$	$1[CB]$	12)	$C1$	$\rightarrow$	$1C$
13)	$[CB]$	$\rightarrow$	$[0B]$	14)	$D[0B]$	$\rightarrow$	$0[DB]$	15)	$D0$	$\rightarrow$	$0D$
16)	$D[1B]$	$\rightarrow$	$1[DB]$	17)	$D1$	$\rightarrow$	$1D$	18)	$[DB]$	$\rightarrow$	$[1B]$
19)	$F[0B]$	$\rightarrow$	$0[FB]$	20)	$F0$	$\rightarrow$	$0F$	21)	$F[1B]$	$\rightarrow$	$1[FB]$
22)	$F1$	$\rightarrow$	$1F$	23)	$[FB]$	$\rightarrow$	0	24)	$G[0B]$	$\rightarrow$	$0[GB]$
25)	$G0$	$\rightarrow$	$0G$	26)	$G[1B]$	$\rightarrow$	$1[GB]$	27)	$G1$	$\rightarrow$	$1G$
28)	$[GB]$	$\rightarrow$	1								