UNIVERSITAT POLITÈCNICA DE CATALUNYA

Departament de Llenguatge i Sistemes Informàtics
Ph.D. Programme: Artificial Intelligence

SYMBOLIC AND CONNECTIONIST LEARNING TECHNIQUES FOR GRAMMATICAL INFERENCE

Autor: René Alquézar Mancho
Director: Alberto Sanfeliu Cortés

March 1997

Chapter 9

Inductive inference of augmented regular expressions

As can be seen in the review of GI methods presented in the first part of this thesis, most of the GI research has been devoted to the theory and methods for learning regular and context-free languages (and the associated types of grammars and automata). On the contrary, work on learning of context-sensitive languages (CSLs) has been extremely scarce in the literature of GI and there is a lack of methods to infer context-sensitive grammars (CSGs). Nevertheless, there is a need for GI methods capable of inferring CSLs, specially for pattern recognition tasks in computer vision, where objects usually contain structural relationships that are not describable by CFLs.

In Chapter 3, two isolated works that referred to the matter of CSL learning have been recalled: an old paper by Chou and Fu on the inference of ATNs [ChFu:76], and a rather recent paper by Takada on the inference of a hierarchy of controlled grammars [Taka:94]. In the former, twenty years ago, a semi-automated heuristic procedure was described that relied on the availability of a teacher providing a-priori knowledge of the target grammar in the form of transformational rules; however, no other work on ATN learning has been reported since then (as far as I know). In the latter, a hierarchy of language families that includes the class of ELLs and is properly contained in the class of CSLs was defined for which RGI algorithms can be used; however, the subclass of CSLs learnable by this approach seems to be rather restricted and does not cover most of the typical context-sensitive structures associated with most objects in pattern recognition problems, such as symmetries, relative sizes of different parts, or contour closing restrictions.

On the other hand, the augmented regular expressions (AREs) defined in the previous chapter represent a non-trivial subclass of CSLs that can describe these kind

of context-sensitive structures, and moreover, their parsing is rather efficient to the contrary of CSGs. Hence, learning AREs from examples, which is the topic addressed in this chapter, is an interesting challenge both for theoretical and practical purposes. In principle, it is reasonable to expect that the problem of inferring AREs be not so hard as the problem of inferring general CSGs from string examples.

An ARE permits to describe a CSL by including a set of constraints that reduce the extension of an underlying regular language (see Fig. 9.1). This property can be exploited to face the problem of learning AREs. Thus, a general approach to infer AREs from string examples is presented in this chapter, that is based on a RGI step followed by an inductive process which tries to discover the maximal number of constraints. This approach, which has been reported in a communication to the ICPR'96 conference held in Vienna (Austria) [AlSa:96], avoids the difficulty of learning CSGs for inferring some CSL acceptors. Indeed, two general methods are described depending on whether only positive or both positive and negative examples are given. An algorithm that carries out the step of constraint induction in low-polynomial time is also described, which is applied in both cases. Then, a specific method for learning AREs can be obtained simply by selecting for the former step an RGI algorithm that yields a regular expression.

As an example, a specific method to infer AREs from positive strings is also presented in this chapter, in which the RGI step is carried out by training a second-order ASLRNN for the next-symbol prediction task, extracting afterwards a DFA from the trained network, and finding finally an equivalent RE. This particular method has been implemented and tested using a set of eight target CSLs associated with ideal models of real patterns. The experimental results obtained, which have also been reported in the recent ICGI'96 colloquium held in Montpellier (France) [AlSC:96], show the feasibility of the proposed approach for learning CSLs.

9.1 A general approach to infer AREs from string examples

Two different statements of the problem of learning AREs from examples can be given. A strong statement concerns the *identification problem*: given a complete presentation of a CSL L over Σ described by un unknown *target* ARE $\tilde{R}_T = (R_T, V_T, \mathcal{T}_T, \mathcal{C}_T)$, *identify* $\tilde{R}_T$, or at least an equivalent ARE $\tilde{R}'_T$ describing L, in finite time, using the presented examples (and maybe some other information about $\tilde{R}_T$ or L, if available). An algorithm that solved this problem for every complete presentation of every CSL represented by an ARE would identify in the limit the subclass of CSLs described by AREs. A weaker statement concerns *data compatibility*: given a sample (S^+, S^-) of

an unknown language L over Σ, *infer* an ARE $\tilde{R} = (R, V, \mathcal{T}, \mathcal{C})$ such that $S^+ \subseteq L(\tilde{R})$, $S^- \cap L(\tilde{R}) = \emptyset$, and $\tilde{R}$ is selected through some *heuristic* bias. The methods presented in this chapter approach this second statement of the ARE learning problem, but some aspects related to the identification problem are also discussed in this section.

A possible ARE learning approach is to split the inductive inference in two main stages: inferring the underlying RE R, and afterwards, inducing the constraints $\mathcal{C}$ to be included in the inferred ARE. A nice property of this approach is expressed in the following theorem.

Theorem 9.1. *Let $\mathcal{A}$ be an ARE learning algorithm consisting of an RGI method $\mathcal{A}_R$ that returns a regular expression, followed by a constraint induction method $\mathcal{A}_\mathcal{C}$. Let us assume that the method $\mathcal{A}_\mathcal{C}$ always uses the RE inferred by $\mathcal{A}_R$ and finds the maximal number of linear relations and the largest lower bounds on the associated star variables, that are satisfied by every string in a positive sample. For every target ARE $\tilde{R}_T = (R_T, V_T, \mathcal{T}_T, \mathcal{C}_T)$, where $\mathcal{C}_T = (\mathcal{L}_T, \mathcal{B}_T)$, if $\mathcal{A}_R$ identifies the RE R_T in the limit, then $\mathcal{A}$ identifies the ARE $\tilde{R}_T$ in the limit.*

Proof. If the target underlying RE R_T is identified at some time step t_1, all the AREs proposed by $\mathcal{A}$ from this time step in a sequential presentation of strings will belong to the set of AREs with R_T as underlying RE. After t_1, $\mathcal{A}_\mathcal{C}$ will obtain the maximal number of linear relations $\mathcal{L}$ defined over V_T that are satisfied by the star instances resulting from parsing by R_T each of the strings in the positive sample of $L(\tilde{R}_T)$ seen so far. If this sample is large enough, some actual instances will have been recorded for each one of the dependent star variables determined by $\mathcal{L}_T$, and therefore, all the relations in $\mathcal{L}_T$ will be reflected in the inferred set $\mathcal{L}$, i.e. $\mathcal{L} \supseteq \mathcal{L}_T$. For each relation in $\mathcal{L}$ but not in $\mathcal{L}_T$, a positive string s, such that $SI_s(V_T)$ does not satisfy this relation, will eventually be provided in a positive or a complete presentation of $L(\tilde{R}_T)$. Hence, the sequence of sets $\mathcal{L}$ proposed after t_1 must necessarily converge to $\mathcal{L}_T$. Once $\mathcal{L}_T$ is identified at some time step $t_2 \geq t_1$, the subset of independent star variables in V_T is correctly determined, and $\mathcal{A}_\mathcal{C}$ will return a set of bounds $\mathcal{B}$ that will contain, for each of these variables, the minimal instance value found in the positive sample seen so far. If any of the lower bounds in $\mathcal{B}$, say $v_j \geq c'_j$, is greater than the corresponding one in $\mathcal{B}_T$, $v_j \geq c_j$, i.e. $c'_j > c_j$, a positive string giving rise to an actual instance of v_j with value c_j will eventually be given in a positive or a complete presentation of $L(\tilde{R}_T)$, and thus, the sequence of sets $\mathcal{B}$ proposed after t_2 must necessarily converge to $\mathcal{B}_T$. In summary, if the sequence of REs proposed by $\mathcal{A}_R$ converges to R_T, the sequence of sets of constraints proposed by $\mathcal{A}_\mathcal{C}$ will converge to $\mathcal{C}_T = (\mathcal{L}_T, \mathcal{B}_T)$, and therefore, the target ARE $\tilde{R}_T = (R_T, V_T, \mathcal{T}_T, \mathcal{C}_T)$ will be identified in the limit. □

The constraint induction method $\mathcal{A}_\mathcal{C}$ of Theorem 9.1 always infers the ARE that describes the smallest language containing the given positive sample among those ARE

with the same underlying RE. This is obvious, since the larger the number of dependent star variables, the smaller the extension of the language described by the ARE, and for a fixed set of independent star variables, the larger the lower bounds on these variables, the smaller the extension of the represented language. Angluin demonstrated that, for some non-superfinite classes of formal languages, if the learning algorithm always proposes the smallest language in the class (in the sense of set containment) that covers a positive sample, then that class of languages can be identified in the limit from only positive presentation [Angl:80a,AnSm:83]. Something similar occurs in the case of Theorem 9.1: provided that R_T is identified previously, the target $\tilde{R}_T$ can be identified in the limit within the set of AREs with R_T as underlying RE by $\mathcal{A}_C$ from the presentation of the strings in $L(\tilde{R}_T)$.

Unfortunately, Theorem 9.1 does not help much in practice, since it is not clear at all how can the underlying RE R_T be identified. What is well known is that it cannot be identified from a positive presentation of $L(\tilde{R}_T)$ or $L(R_T)$. A minimal DFA for $L(R_T)$ could be identified in the limit from a complete presentation of $L(R_T)$ [OnGa:92], but not from a complete presentation of $L(\tilde{R}_T)$. Even if the minimal DFA accepting $L(R_T)$ were determined, it would remain the problem of finding R_T in the set of REs equivalent to that DFA.

Hence, a significant drawback of the preceding two-stage approach concerns how to determine the input data to be supplied to the RGI method. Let (S^+, S^-) be a given sample of a target language $L(\tilde{R})$. In order to learn the underlying RE R, or at least some equivalent RE, we should be able firstly to partition the negative sample S^- in the two subsets S_0^- and $(S^- - S_0^-)$, characterized by $S_0^- \cap L(R) = \emptyset$ and $(S^- - S_0^-) \subseteq L(R)$, where S_0^- is the correct negative sample for the RGI step.

Fig. 9.1 displays the situation for a target ARE $\tilde{R}$ with k constraints: starting with the language described by the underlying RE, each time an additional constraint of the ARE is imposed the extension of the represented language disminishes up to reach the target language, and some of the negative examples can be due to the insatisfaction of one or more constraints. Thus, unless a teacher is available to partition the negative examples and determine S_0^- correctly, or it is guaranteed that $S^- = S_0^-$, it is not proper to supply the negative examples to the RGI method, since typically, an overfitting of the positive examples will occur. Consequently, a heuristic method based only on the positive examples may be preferred in some cases for the regular inference step. In fact, two different general methods for inferring AREs, which rely on an RGI technique for the inference of the underlying RE, are presented next for the two distinct cases of knowing or ignoring the part of the sample that must be supplied in the RGI step.

If only positive examples S^+ are given, or if it is known which part of the negative sample corresponds to S_0^-, then the learning process depicted in Fig. 9.2 may be applied. In the first case (only positive examples), either a heuristic or a characterizable method

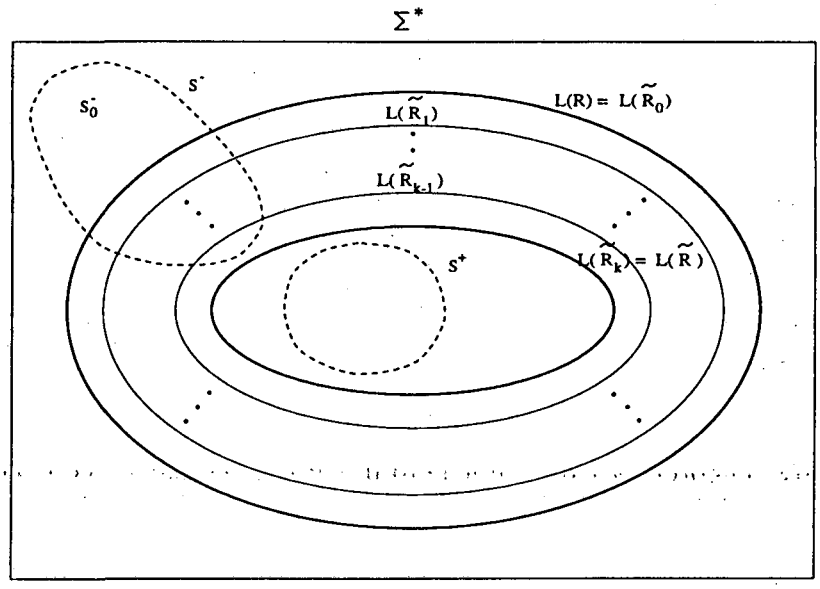


Fig. 9.1 *Representation of the languages involved in the ARE learning problem.*

can be selected for the RGI step. In the second case, the use of an RGI algorithm with the identification in the limit property (e.g. the RPNI algorithm [OnGa:92]) may be preferred. In both cases, if the RGI algorithm returns an FSA, which is the usual case, an FSA-to-RE mapping has to be applied to yield an RE, and the one described in Section 8.1 is recommended. Anyway, the RGI procedure must return an unambiguous RE R, and optionally, it may also yield an FSA A, whenever R has been obtained from A. Note that the identification in the limit of the minimal DFA of the regular language described by a target RE does not imply the identification in the limit of this RE, and therefore, even if the RPNI algorithm is used in the RGI step, this does not guarantee the identification in the limit of the target ARE by the global method.

Once an RE R is inferred, the associated *star variables* V and *star tree* $\mathcal{T}$ are easily determined using Algorithm 8.2. Then, an *array of star instances* ASI is built containing the information recorded from parsing all the positive examples by R. The efficiency of this parsing step is improved if a DFA A equivalent to R is available from the RGI step (remember Section 8.4.1). Finally, the gathered star instances are analysed to induce the set of constraints $\mathcal{C} = (\mathcal{L}, \mathcal{B})$ of the inferred ARE $\tilde{R}$, which includes the maximal number of linear relations and the largest lower bounds satisfied by ASI. Thus, the constraint induction method $\mathcal{A}_\mathcal{C}$ of Theorem 9.1 is actually split in the tasks of building the star tree, parsing a set of example strings, and inferring as many constraints as possible from the collected star instances.

However, in the most general case, a sample $S = (S^+, S^-)$ may be given such that

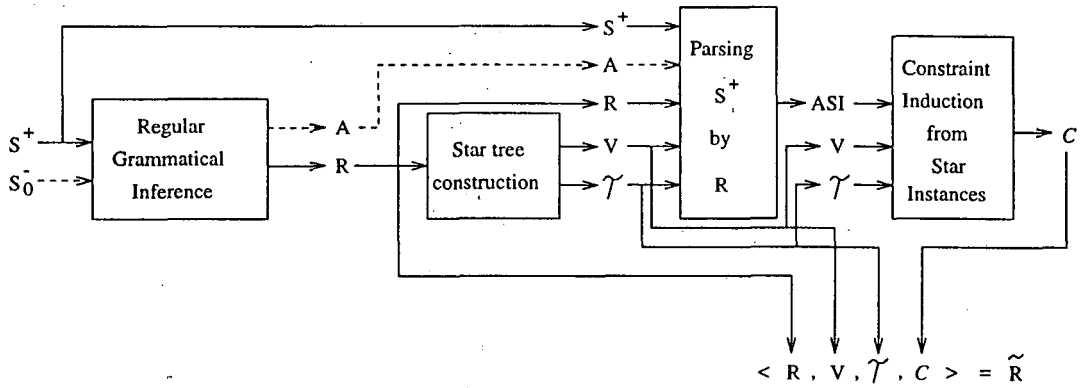


Fig. 9.2 *Block diagram of a general method to learn AREs from positive examples (and maybe negative examples of the underlying regular language).*

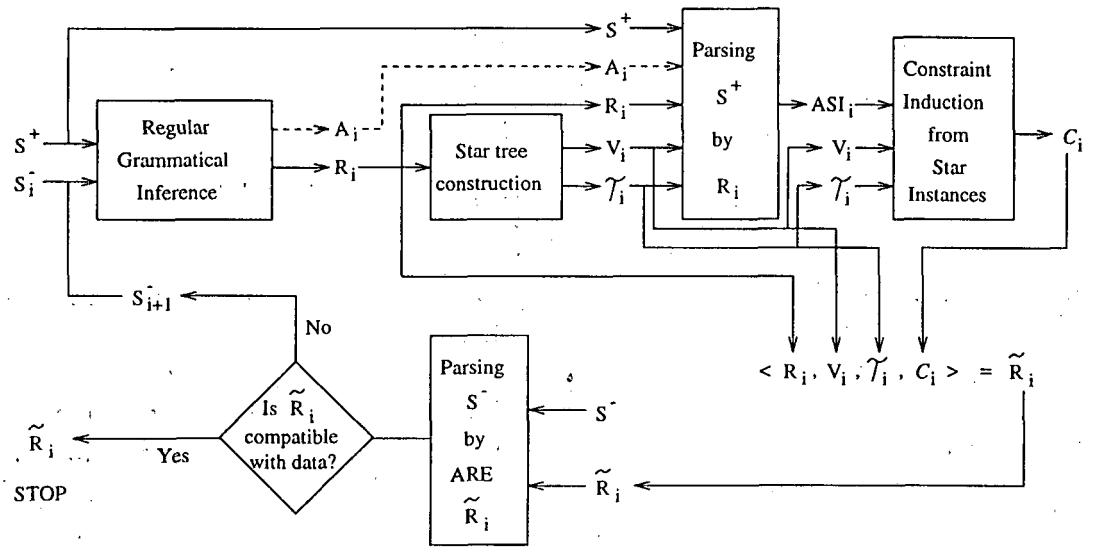


Fig. 9.3 *Block diagram of a general method to learn AREs from positive and negative examples.*

S_0^- is not known. Then, some strategy must be adopted to cope with the negative examples. A possible general method that ensures the data compatibility of the inferred ARE is depicted in Fig. 9.3. Initially, an ARE is inferred from just the positive examples, and if it accepts some string in S^-, then the RGI step is run again incorporating the conflictive negative examples. Several such cycles may be required to infer a consistent ARE, and in the worst case, all the negative sample would be supplied to the RGI method at the last cycle. In that case, the aforementioned overfitting behavior could arise, but the consistency of the inferred ARE with the given sample would be guaranteed.

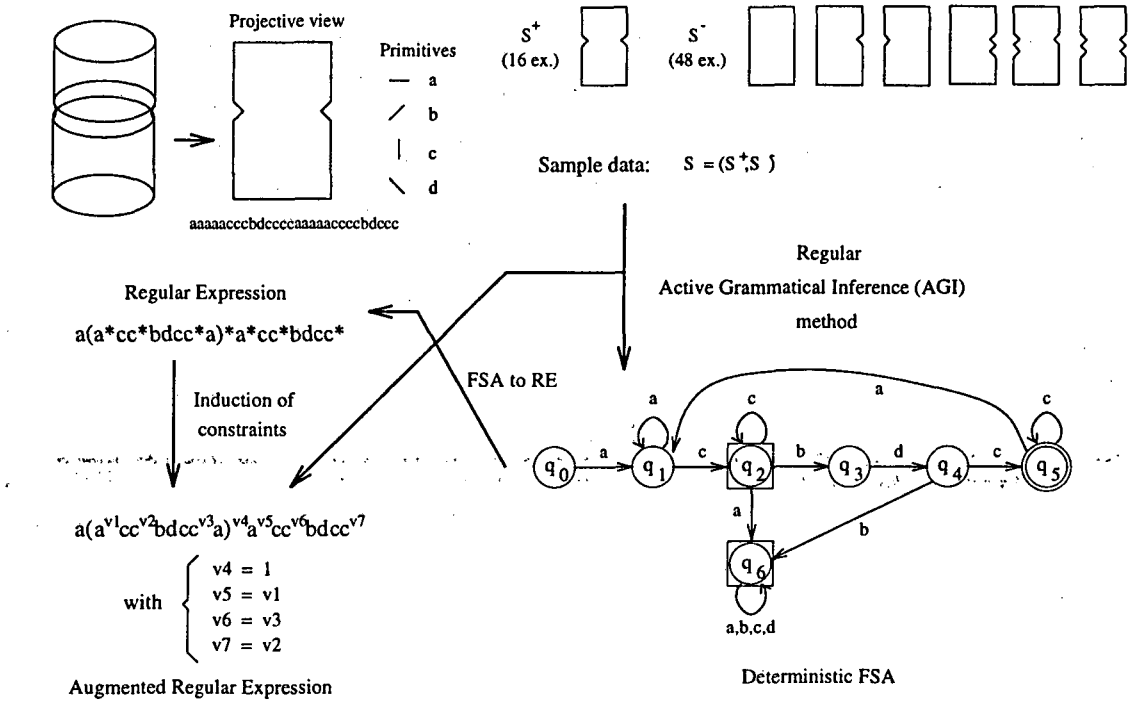


Fig. 9.4 *A case of inference of an ARE from string examples.*

Let us illustrate the ARE learning method depicted in Fig. 9.2 by using an actual example that is shown in Fig. 9.4. The problem was 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 associated language is context-sensitive, and hence, we cannot expect that an RGI or a CFGI algorithm infers a suitable recognizer. Nevertheless, an adequate description like $a^m c^n bdc^p a^m c^p bdc^n$ should be reachable from a few examples. In fact, the ARE learning method is a rather straightforward approach for inferring syntactic descriptions of this kind from examples. In the case of Fig. 9.4, a sample $S = (S^+, S^-)$ of 16 positive and 48 negative examples was given, corresponding to some variable-size instances of the contours shown in the top of the figure. Assuming $S^- = S_0^-$, the *active grammatical inference* method described in Chapter 7 was applied to the entire S, and the DFA displayed in Fig. 9.4 was obtained. This DFA accounts for the basic repetitive structure of the model, but it over-generalizes a lot, accepting many invalid contours without any length restriction. From this DFA, an equivalent RE $R = a(a^*cc^*bdcc^*a)^*a^*cc^*bdcc^*$ was obtained as base of the ARE by applying the FSA-to-RE mapping explained in Section 8.1. Finally, from the automatic analysis of the star instances produced by parsing the positive examples by the RE, a set of constraints could be induced that, in conjunction with the RE, perfectly described the target language. The algorithm used for constraint induction is described in the following section.

It is worthwhile to point that, without the proposed modification of Arden's method, the application of Algorithm 8.1 to the DFA of Fig. 9.4 leads to the RE $R' = a(a + cc^*bdcc^*a)^*cc^*bdcc^*$, from which not all the constraints required to describe the target pattern class are inferrable. On the other hand, although R could be further transformed into the more intuitive description $R'' = aa^*cc^*bdcc^*(aa^*cc^*bdcc^*)^*$ by using some RE equivalence rules (Eqs.(2.13) and (2.10)), the inferred R is good enough as underlying RE to allow the inference of all the contextual constraints involved in the modelled patterns.

In the preceding example, a single cycle of RE inference and constraint induction was enough to obtain an ARE consistent with the given sample, because all the negative examples were forced to be rejected by the underlying RE. As commented before, this is not an adequate strategy in general, because some of the negative examples might belong indeed to the language described by the unknown target RE. In such a case, it would be helpful to have available an informant who partitioned the negative sample in the two subsets of strings to be accepted and rejected respectively by the underlying RE, but this is a rather unrealistic learning condition.

Therefore, several cycles (each yielding a tentative ARE) may be required to reach a consistent ARE, according to the process displayed in Fig. 9.3. Algorithm 9.1 is given to implement the ARE learning method sketched in Fig. 9.3, which always infers a data compatible ARE, but trying to minimize the number of negative strings supplied in the RGI step, with the aim of promoting the maximal generalization of the positive sample by the underlying RE. It can be seen that Algorithm 9.1 uses all the algorithms described in Chapter 8, together with Algorithm 9.2, which is the constraint induction method that will be explained in Section 9.2.

It must be remembered that a wide range of symbolic and connectionist techniques is available to select an RGI algorithm $\mathcal{A}_{RGI}$ for the RGI step in Algorithm 9.1 [Greg:94]. This must not be interpreted, however, as if the choice of the RGI method were irrelevant. On the contrary, the implicit or explicit biases of the selected method may help or not to reach a "suitable" regular expression that allows the discovery of the target context constraints. Moreover, in order to apply the RE parsing methods of Chapter 8, it is mandatory that an unambiguous RE be inferred in the RGI step. This unambiguous RE may be obtained directly by $\mathcal{A}_{RGI}$ or computed from an inferred DFA using the FSA-to-RE mapping of Section 8.1. The method used for parsing the strings by the inferred RE will also depend on whether the FSA-to-RE transformation is performed or not in the RGI step.

ALGORITHM 9.1: *Infers an ARE consistent with a given learning sample.*

Inputs:

$S = (S^+, S^-)$	is a sample of an unknown language L over an alphabet Σ;
$\mathcal{A}_{RGI}$	is an RGI algorithm that guarantees the inference of a DFA or an unambiguous RE that is consistent with the examples supplied in the RGI step;
RGI_provides_RE	is a boolean parameter whose value is TRUE whenever $\mathcal{A}_{RGI}$ directly infers an RE and FALSE otherwise ($\mathcal{A}_{RGI}$ returns a DFA);

Outputs:

$\tilde{R} = (R, V, \mathcal{T}, \mathcal{C})$	is the inferred ARE, where
R	is the underlying RE inferred by the RGI step;
V	is the set of star variables associated with R;
$\mathcal{T}$	is the star tree associated with R;
$\mathcal{C} = (\mathcal{L}, \mathcal{B})$	is the set of induced constraints, where
$\mathcal{L}$	is a set of linear relations among the star variables in V;
$\mathcal{B}$	is a set of lower bounds defined on the star variables that are considered independent in $\mathcal{L}$;

begin_algorithm

$S_0^- := \emptyset$; { set up the negative sample for the RGI step }

repeat { an ARE inference until an ARE consistent with $S = (S^+, S^-)$ is obtained }

if *RGI_provides_RE* **then**

$R :=$ $\mathcal{A}_{RGI}(S^+, S_0^-)$; { apply the RGI algorithm returning an RE }

else

$A :=$ $\mathcal{A}_{RGI}(S^+, S_0^-)$; { apply the RGI algorithm returning a DFA }

$n :=$ number_of_states (A);

modified_Algorithm_8.1 $(A, n, R'_{ij}, C_j, R''_{ji}, \alpha^n_{ij}, \alpha^n_{jj}, \alpha^n_{ji}, IR'_{ij}, IC_j, IR''_{ji})$;
{ returns the REs $R'_{ij}, C_j, R''_{ji}, \alpha^n_{ij}, \alpha^n_{jj}, \alpha^n_{ji}$ and the indices $IR'_{ij}, IC_j, IR''_{ji}$ for $0 \leq j \leq n-1$, $0 \leq i < j$ }

$< R, skelR >:=$ simplify_equivalent_RE (A, n, R'_{ij}, C_j); { the "canonical" RE $R = \psi(A)$ and its skeleton $skelR$ are obtained from the REs C_0 and $R_{ij} = R'_{ij}C_j$, for $1 \leq j \leq n-1$, $0 \leq i < j$ }

end_if

{ $L(R) \supseteq S^+$ and $L(R) \cap S_0^- = \emptyset$ }

$< V, \mathcal{T} >:=$ Algorithm_8.2 (R); { returns the set of star variables V and the star tree $\mathcal{T}$ associated with R}

$k := 0$; reset_list(S^+); { S^+ is considered as a list of strings }

while not end_of_list(S^+) **do**

$s :=$ get_current_element(S^+); { where s is a string over Σ }

```
        if RGI_provides_RE then
            < parsed, SI >:= Algorithm_8.3 (R, V, T, s);   { parse s by RE R }
        else
            < parsed, SI >:= Algorithm_8.4 (A, n, R, skelR, R'_ij, C_j, R''_ji, α^n_ij, α^n_jj, α^n_ji,
                IR'_ij, IC_j, IR''_ji, V, T, s);   { parse s by RE R using the equivalent
                DFA}
        end_if
        { parsed = TRUE, since s ∈ L(R) }
        k := k + 1;
        ASI[k] := SI;   { store the star instances resulting from parsing s }
        move_to_next_element(S+);
    end_while
    number_positive_examples := k;
    < L, B, nd >:= Algorithm_9.2 (V, T, ASI, number_positive_examples);   { infers L,
        a set of nd linear relations, and B, a set of lower bounds, both satisfied by ASI}

    R̃ := (R, V, T, (L, B));   { construct the inferred ARE from its components }
    { L(R̃) ⊇ S+ and L(R̃) ∩ S0− = ∅ }
    consistent_ARE := TRUE;
    S_C− := S− − S0−;   { S_C− contains the negative examples not used in the RGI step }
    reset_list(S_C−);   { S_C− is considered as a list of strings }
    while not end_of_list(S_C−) do   { check the consistency of R̃ }
        s := get_current_element(S_C−);   { where s is a string over Σ }
        if RGI_provides_RE then
            < parsed, SI >:= Algorithm_8.3 (R, V, T, s);   { parse s by RE R }
        else
            < parsed, SI >:= Algorithm_8.4 (A, n, R, skelR, R'_ij, C_j, R''_ji, α^n_ij, α^n_jj, α^n_ji,
                IR'_ij, IC_j, IR''_ji, V, T, s);   { parse s by RE R using the equivalent
                DFA}
        end_if
        if parsed then
            satisfy_constraints := Algorithm_8.5 (V, T, SI, L, B);
            if satisfy_constraints then   { s is recognized by the ARE R̃ }
                S0− := S0− ∪ {s};   { append s to negative sample for next RGI step
                    }
                consistent_ARE := FALSE;   { because L(R̃) ∩ S− ≠ ∅ }
            end_if
        end_if
        move_to_next_element(S_C−);
    end_while
until consistent_ARE;
{ the inferred R̃ = (R, V, T, (L, B)) is consistent with S = (S+, S−) }
end_algorithm
```

9.2 Induction of ARE constraints from recorded star instances

Once an RE R has been inferred by the chosen RGI method, and the associated star variables V and star tree $\mathcal{T}$ have been determined by Algorithm 8.2, the aim is to infer an ARE $\tilde{R} = (R, V, \mathcal{T}, (\mathcal{L}, \mathcal{B}))$ such that $\mathcal{L}$ contains the maximal number of linear relations met by all the provided positive examples and $\mathcal{B}$ specifies the largest lower bounds for the independent star variables. In other words, $\tilde{R}$ should describe the *smallest language* L such that $L \supseteq S^+$ and L is accepted by an ARE with the given R as underlying RE. The theoretical goodness of this scheme has been shown in Theorem 9.1; if R is the correct RE, it will lead to the correct ARE whenever a sufficiently large number of positive examples is given. To this end, firstly, the positive strings in S^+ must be parsed by R (using one of the algorithms presented in Section 8.4.1), thus giving rise to an array of sets of star instances $ASI_{S^+}(V)$.

Algorithm 9.2 implements the desired method for constraint induction, which returns the maximal set of constraints $\mathcal{C} = (\mathcal{L}, \mathcal{B})$ satisfied by all the sets of star instances in $ASI_{S^+}(V)$. This is, $SI_s(V)$ *satisfies* $\mathcal{C} = (\mathcal{L}, \mathcal{B})$ for all $s \in S^+$, there is no $\mathcal{L}'$ with more equations than $\mathcal{L}$ satisfied by $SI_s(V)$ for all $s \in S^+$, and there is no $\mathcal{B}'$ satisfied by $SI_s(V)$ for all $s \in S^+$ such that $\mathcal{B}'$ involves the same star variables than $\mathcal{B}$ and for at least one variable the lower bound in $\mathcal{B}'$ is greater than in $\mathcal{B}$. Before describing Algorithm 9.2, let us introduce some required definitions, which are simple extensions of some counterparts given in Chapter 8.

Definition 9.1. Given a star tree $\mathcal{T}$, an array of sets of star instances $ASI_{S^+}(V)$ for a certain set of strings S^+, and two nodes $v_i, v_j \in V$, we say that v_i is a *degenerated ancestor* of v_j for S^+ iff

i) v_i is an ancestor of v_j in $\mathcal{T}$, and
ii) for each instance of v_i in $ASI_{S^+}(v_i)$, all the values of the instances of v_j in $ASI_{S^+}(v_j)$ that are derived from it are constant.

Again, the root r of $\mathcal{T}$ is considered, by definition, as a non-degenerated ancestor of any other node v_j for every set of strings. An alternative valid definition is the following: v_i is a *degenerated ancestor* of v_j for S^+ iff v_i is a degenerated ancestor of v_j for all $s \in S^+$.

Definition 9.2. Let $v_i \in V \cup \{r\}$, $v_j \in V$; we say that v_i is the *housing ancestor* of v_j for a set of strings S^+ iff v_i is the *nearest* non-degenerated ancestor of v_j for S^+.

Note that the *housing ancestor* of v_j for S^+ is not necessarily the housing ancestor of v_j for every string in S^+. The procedure *determine_housing_ancestor_and_instances*,

that was described in Section 8.4.2, can be used to find the housing ancestor of a star variable v_j for a set of strings and to form the corresponding vector containing the *non-redundant* star instances of v_j.

Algorithm 9.2 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 descendants.* To construct this partition, each ancestor node of $\mathcal{T}$ keeps track of its housed descendants that have been found independent.

Algorithm 9.2 follows the star tree $\mathcal{T}$ by levels and, for each star variable v_j, determines its housing ancestor for S^+, say $h(v_j)$, forms a vector B with its non-redundant star instances in $ASI_{S^+}(V)$, and tries to establish a possible linear relation with respect to the housed descendants of $h(v_j)$ previously labeled as "independent". This is, a linear system $A \cdot X = B$ is built, where matrix A contains the instances of the independent housed descendants of $h(v_j)$. Next, the rank of the matrix A is evaluated and any linearly dependent column is removed from A. Finally, it is determined whether the vector B of actual instances of v_j is linearly dependent on the columns of the matrix A. If the system $A \cdot X = B$ has a solution then X contains the coefficients of a linear relation, and this relation is appended to $\mathcal{L}$; otherwise, v_j is put in the list of independent housed descendants of $h(v_j)$, the associated vector B is stored for further constraint discovery, and the minimal value c_j in B is taken to establish a bound $v_j \geq c_j$, which is appended to $\mathcal{B}$.

ALGORITHM 9.2: *Induces a set of constraints on the values of the star variables (linear relations and bounds) that hold among the instances resulting from parsing a set of strings.*

Inputs:

V	is a set of star variables, that includes, for each variable, the address of the corresponding node in the star tree $\mathcal{T}$;
$\mathcal{T}$	is a star tree containing the variables in V as nodes;
ASI	is an array of sets of star instances $ASI_{S^+}(V)$ that stores the instances resulting from parsing a set of strings S^+ by the RE from which V and $\mathcal{T}$ have been built;
number_of_strings	is the size of the array ASI;

Outputs:

$\mathcal{L}$	is a set of linear relations among the star variables in V, that comprises all the linear relations satisfied by all the sets of star instances in ASI;
$\mathcal{B}$	is a set of lower bounds defined on the star variables that are considered independent in $\mathcal{L}$, and which is directly inferred from the star instances in ASI;
nd	is the number of relations in $\mathcal{L}$;

begin_algorithm

$\mathcal{L}$:= create_empty_list_of_relations(); { initialize $\mathcal{L}$ as an empty set }
$\mathcal{B}$:= create_empty_list_of_bounds(); { initialize $\mathcal{B}$ as an empty set }
$nd := 0$; { initialize the number of linear relations in $\mathcal{L}$ (and dependent star variables) }
max_level := depth_of_startree ($\mathcal{T}$); { this is the level of the deepest node in $\mathcal{T}$ }

for $l := 0$ **to** $max\_level - 1$ **do** { visit the father nodes of $\mathcal{T}$ by levels }

N := startree_nodes_in_level ($\mathcal{T}, l$);
reset_list(N); { N is an ordered list with pointers to the nodes in level l of $\mathcal{T}$ }
while not end_of_list(N) **do**

$father$:= get_current_element(N);
$father\_id$:= node_identifier_of ($father$);
$number\_of\_indep\_housed\_descendants$ [$father\_id$] := 0;
$list\_of\_indep\_housed\_descendants$ [$father\_id$] := create_empty_list();
$max\_number\_of\_rows\_of\_system\_matrix$ [$father\_id$] :=
sum_of_instance_values ($ASI, number\_of\_strings, father\_id$);
{ when $father\_id = 0$ (for the root node), the value returned by the function is $number\_of\_strings$ }
for $k := 1$ **to** nsons_of ($\mathcal{T}, father$) **do**

son := k-th_son_of ($\mathcal{T}, father, k$);
son_id := node_identifier_of (son);
determine_housing_ancestor_and_instances ($\mathcal{T}, son, son\_id, ASI$,
$number\_of\_strings, 0, anc\_id, instances[son\_id]$);
{ returns anc_id, the identifier of the housing ancestor of son, and $instances[son\_id]$, the values of the instances of v_{son_id} derived from the instances of v_{anc_id} in ASI }
row_max_number := $max\_number\_of\_rows\_of\_system\_matrix$ [anc_id];
$column\_max\_number$:=
$number\_of\_indep\_housed\_descendants$ [anc_id] + 1;
build_system_matrix ($son\_id, list\_of\_indep\_housed\_descendants[anc\_id]$,
$instances, row\_max\_number, column\_max\_number, row\_actual\_number$,
B, A); { vector B is given by the actual instances of v_{son_id}, and the system matrix A is given by the star instances of the independent housed descendants of v_{anc_id}, after removing the rows corresponding to dummy instances of v_{son_id} }
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 previ-
                            ous 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
                    AB := build_extended_system_matrix (A, B, row_actual_number,
                        column_actual_number); { 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 }
                        X := solve_linear_system (A, B, row_actual_number, column_actual_number
                            {vector X is obtained by solving A · X = B}
                        right_hand_side.coefficients := X;
                        right_hand_side.independent_variables :=
                            extract_masked_list (list_of_indep_housed_descendants[anc_id],
                            column_max_number, column_actual_number, columns_mask);
                        right_hand_side := remove_indep_variables_with_coef_zero
                                                (right_hand_side);
                        nd := nd + 1; { the number of dependent star variables is
                            increased and the new constraint is appended to L }
                        L := put_element_in_list_of_relations(L, < son_id, right_hand_side >);
                    else { the appended column is linearly independent }
                        number_of_indep_housed_descendants [anc_id] :=
                                number_of_indep_housed_descendants [anc_id] + 1;
                        list_of_indep_housed_descendants [anc_id] := put_element_in_list
                                (list_of_indep_housed_descendants[anc_id], son_id);
                        low_bound := find_minimal_value (B, row_actual_number);
                        B :=put_element_in_list_of_bounds(B, < son_id, low_bound >)
                    end_if
                else
                    B := put_element_in_list_of_bounds (B, < son_id, 0 >);
                end_if
            end_for
            move_to_next_element(N);
        end_while
    end_for
    end_algorithm
```

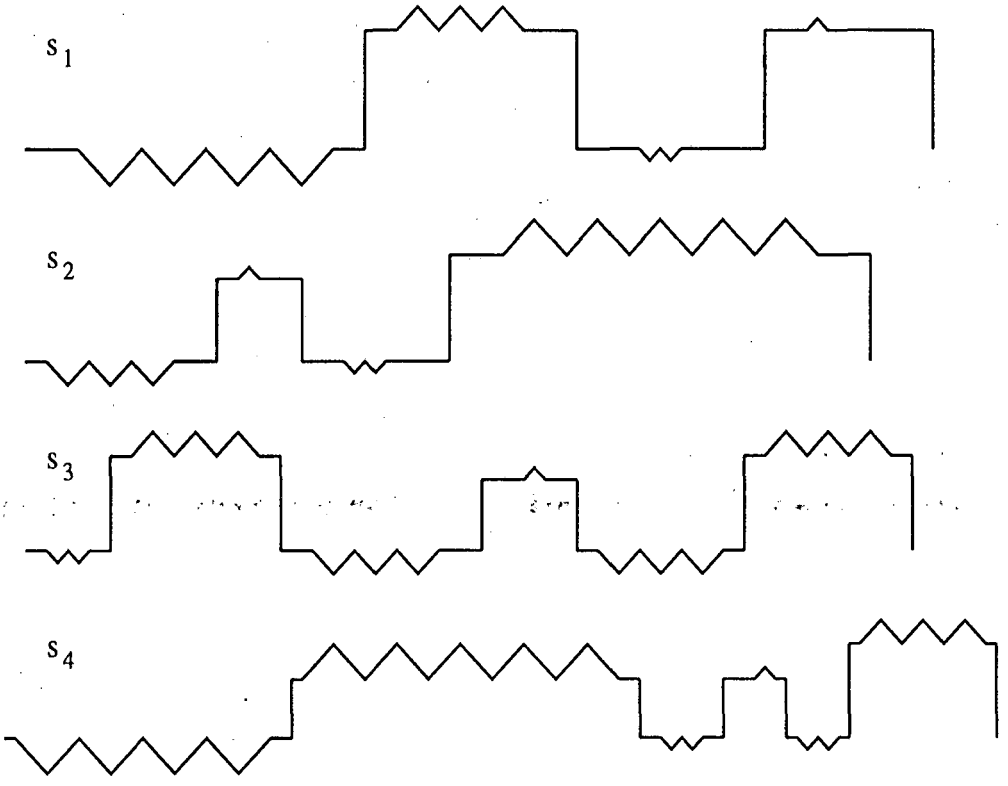
The time complexity of Algorithm 9.2 is $O(|V|^3 \cdot I(ASI_{S^+}(V)))$, where $I(ASI_{S^+}(V))$ is the maximal number of instances of a star variable yielded by parsing the set of strings S^+. This cost comes from a number of calls of order $O(|V|)$ to the functions *rank_of_matrix* and *solve_linear_system*, whose worst-case complexity is $O(|V|^2 \cdot I(ASI_{S^+}(V)))$. The rest of processes carried out in Algorithm 9.2 have a lower complexity. Thus, the global cost of all the calls to the procedure *determine_housing_ancestor_and_instances* is $O(|V| \cdot height(\mathcal{T}) \cdot I(ASI_{S^+}(V)))$, whereas the cost of all the calls to the procedure *build_system_matrix* is $O(|V|^2 \cdot I(ASI_{S^+}(V)))$.

Let us illustrate the constraint induction method using the set of strings displayed in Fig. 9.5, that belong to $L(\tilde{R}_2)$, where the ARE $\tilde{R}_2 = (R_2, V_2, \mathcal{T}_2, (\mathcal{L}_2, \mathcal{B}_2))$ was already defined in Section 8.3 but is recalled herebelow:

$$\begin{aligned} R_2(V_2/*) &\doteq (c^{v_1}(d^{v_2}b^{v_3})^{v_4}c^{v_5}a^{v_6}c^{v_7}(b^{v_8}d^{v_9})^{v_{10}}c^{v_{11}}e^{v_{12}})^{v_{13}} \\ \mathcal{L}_2 &= \{v_{11} = v_1 + v_5 - v_7,\ \ v_{12} = v_6, \\ &\quad v_2 = v_4 - 1,\ \ v_3 = v_4 - 1, \\ &\quad v_8 = 0.5v_{10} + 0.5,\ \ v_9 = 0.5v_{10} + 0.5\} \quad \text{and} \\ \mathcal{B}_2 &= \{v_4 \geq 2;\ v_6, v_{10}, v_{13} \geq 1;\ v_1, v_5, v_7 \geq 0\}. \end{aligned}$$

First, the sons of the root node of $\mathcal{T}_2$ are processed; in this case, v_{13} is the only one and it is found independent by Algorithm 9.2, 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 housed descendants of v_{13} already processed. Fig. 9.6 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 $\mathcal{L}$. Then, the last son v_{12} is visited and the second linear relation, $v_{12} = v_6$, 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 two last linear relations: $v_8 = 0.5v_{10} + 0.5$, $v_9 = 0.5v_{10} + 0.5$. At the end, the inferred set of relations $\mathcal{L}$ coincides with the target $\mathcal{L}_2$ of $\tilde{R}_2$.

On the other hand, the set of bounds inferred from the four strings in Fig. 9.5 is $\mathcal{B} = \{v_1 \geq 1,\ v_4 \geq 2,\ v_5 \geq 1,\ v_6 \geq 5,\ v_7 \geq 1,\ v_{10} \geq 1,\ v_{13} \geq 2\}$, which does not fully coincide with the target set $\mathcal{B}_2$. Some more positive examples would be needed to infer $\mathcal{B} = \mathcal{B}_2$, since for each independent star variable, an actual instance with the value given by the lower bound in $\mathcal{B}_2$ would be required to induce the corresponding bound correctly.



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

Fig. 9.5 *Four example strings from $L(\tilde{R}_2)$ which lead to the inference of the set $\mathcal{L}_2$.*

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

Fig. 9.6 *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 $X = [0\ 1\ 0\ 1\ 0\ \text{-}1\ 0]^T$.*

9.3 Experimental assessment of a specific method for inferring AREs from positive examples

In order to test the general approach proposed in Section 9.1 for inferring AREs from examples, some specific method based on a selected RGI algorithm had to be implemented. The first question that affects the selection of a particular method is to know whether only positive examples or both positive and negative examples are supplied as learning data. If only positive examples are given, the learning scheme shown in Fig. 9.2 is applicable, and this simplifies the global process to infer an ARE, since just one RGI step and one constraint induction step need to be performed for a training sample. If both positive and negative examples are given, the simple scheme of Fig. 9.2 can only be applied if an informant partitions the negative sample properly, and otherwise, the scheme displayed in Fig. 9.3 (and described in Algorithm 9.1) must be followed, possibly involving several cycles of RE inference and constraint induction. In this latter case, the procedure $\mathcal{A}_{RGI}$, which is in charge of the RGI step, should actually call a different RGI method for the first cycle (when only the positive examples are taken into account) than for the rest of cycles (when the consistency with a set of both positive and negative examples is required).

As a consequence, for the sake of simplicity and efficiency in the experimental tests, it was decided to implement and assess a method for inferring AREs from just positive examples, following the process depicted in Fig. 9.2 and using an RGI method based only on a positive sample. Any of the symbolic methods for RGI from a positive sample reviewed in Section 2.3.1 could have been chosen, but it was preferred to try an RNN-based approach for the RGI step, taking profit of the AGI tools that had been developed and used in the tests reported in Section 6.2.2.

9.3.1 A method for inferring AREs from positive examples

The particular method that was selected for the regular inference stage consists of three steps:

i) to train a second-order 2L-ASLRNN (with an antisymmetric logarithm activation function in the recurrent layer and a sigmoid function in the output layer) for the next-symbol prediction task from the given positive examples, using a true gradient-descent learning algorithm;

ii) to extract afterwards a DFA from the net through the use of the FSA extraction method reported in Section 6.1 for the case of only positive examples; and

iii) to perform finally the DFA to RE mapping recommended in Section 8.1.

In the first step, the set of training examples is supplied to the net several times (epochs) up to reach a predetermined number of epochs, which must be large enough to let the network arrive at a minimum of the error function, where the total prediction error on the training set stabilizes. The network input/output requirements and training procedure for the next-symbol prediction task have already been explained in Section 4.2.1, so they will not be repeated here. Likewise, the dynamics of second-order ASLRNNs and the learning algorithm used to train them have also been described previously (see Section 4.1.2).

After the neural training phase, the recurrent layer of the 2L-ASLRNN is supposed to have inferred approximately the state transition function of a DFA, while the output layer is supposed to have learned a function that, for each state, gives the probabilities of each symbol to be the next symbol in a valid string. Thus, it is assumed that the net has developed its own states in the form of clusters of the (recurrent) hidden unit activation vectors. A hierarchical clustering of the recurrent unit activation vectors, based on inter-cluster minimal distance, serves to guide a state merging process from the sample prefix tree, which stops when the inter-cluster minimal distance exceeds a certain threshold d. This threshold is fixed depending on the number of recurrent units N, according to the heuristic rule $d = 2\sqrt{N}/3$, that relates it to the diameter of the N-dimensional activation space.

Then, in the third step, the extracted DFA A is supplied to the proposed modification of Arden's algorithm (see Section 8.1) to obtain an equivalent RE R. The resulting RE R is always unambiguous (thus easing the RE parsing) and such that the star subexpressions due to self-loops are distinguished from those corresponding to the rest of circuits of the DFA. Since self-loops may represent indefinite length or duration of a basic primitive, this separation allows for a later induction of constraints relating the lengths or durations of the different parts of a pattern.

Once the underlying RE R is inferred by the above three-step procedure, the rest of processes shown in Fig. 9.2 are carried out to infer an ARE $\tilde{R} = (R, V, \mathcal{T}, \mathcal{C})$, using the DFA-assisted RE parsing method (Algorithm 8.4) and the constraint induction algorithm described in the preceding section (Algorithm 9.2).

9.3.2 Experimental assessment

The specific method for inferring AREs just described was applied to learn a set of eight test CSLs, associated with ideal models of real patterns, which are shown in Fig. 9.7. Test languages L_1-L_4 are over the alphabet $\Sigma_1 = \{a, b, c, d, e, f, g, h\}$ and test languages L_5-L_8 are over the alphabet $\Sigma_2 = \Sigma_1 \cup \{i, j, k, l\}$.

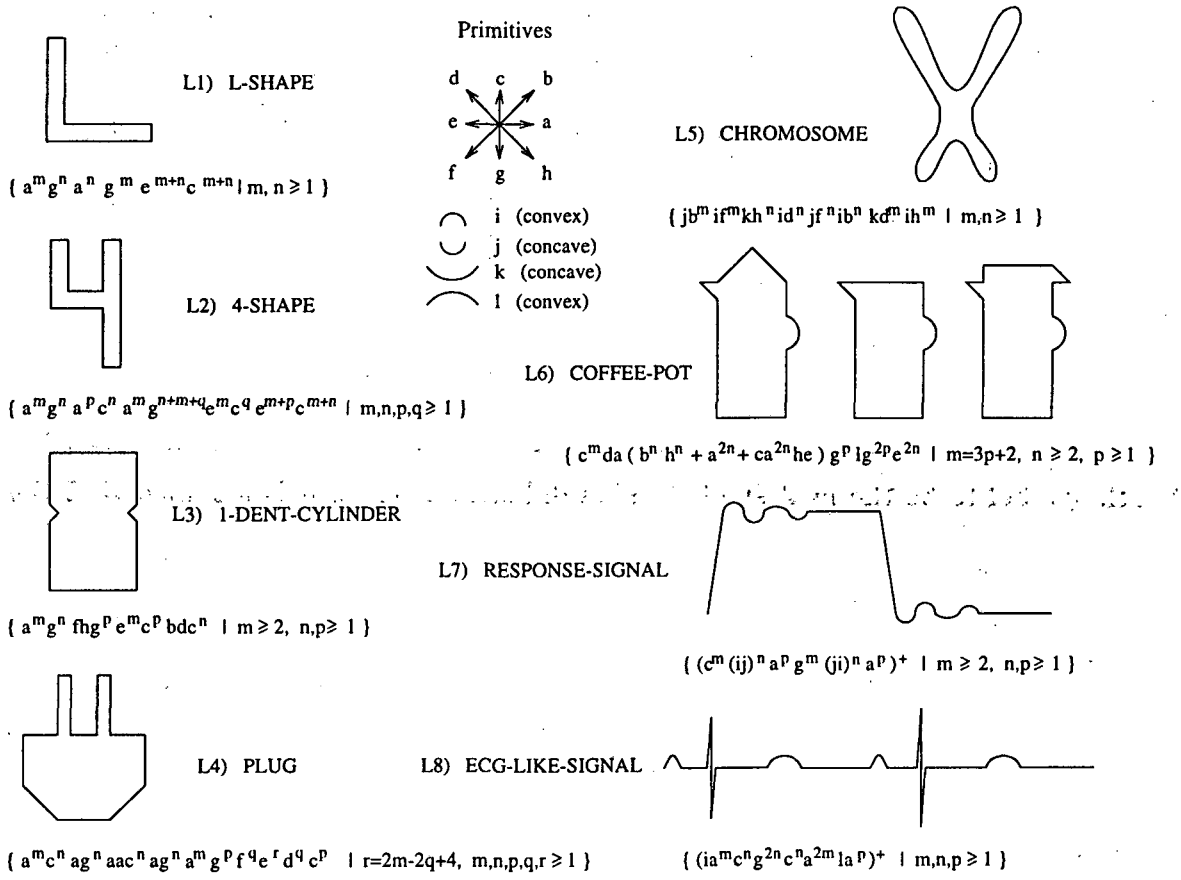


Fig. 9.7 *The eight test CSLs and the patterns described by them.*

The experiment was performed according to the following protocol. For each test language L_i ($1 \leq i \leq 8$), a global sample $SL_i = (SL_i^+, SL_i^-)$ containing 64 positive and 64 negative examples (without repetitions) was generated manually. The positive sample SL_i^+ was written by giving several values to each parameter (degree of freedom) of the language. The negative sample SL_i^- was divided in two groups: 48 strings (75%) corresponded to counterexamples for the target regular language $L(R_i)$ and they were generated from the positive examples by removing, inserting or substituting some subpatterns (i.e. near-misses); 16 strings (25%) corresponded to strings in $L(R_i)$ which did not satisfy at least one constraint in the target CSL L_i. Then, for each language, eight training samples S_{ij}^+ ($1 \leq j \leq 8$), each containing 16 positive examples, were defined such that the whole set SL_i^+ was used and each block of 8 strings was included in two consecutive samples S_{ij}^+ and $S_{i(j+1)}^+$. Every training sample S_{ij}^+ was structurally complete with respect to the minimal DFA describing $L(R_i)$.

For each test language L_i, a second-order 2L-ASLRNN with 4 recurrent units, but different initial random weights, was trained from each sample S_{ij}^+ to learn the next-

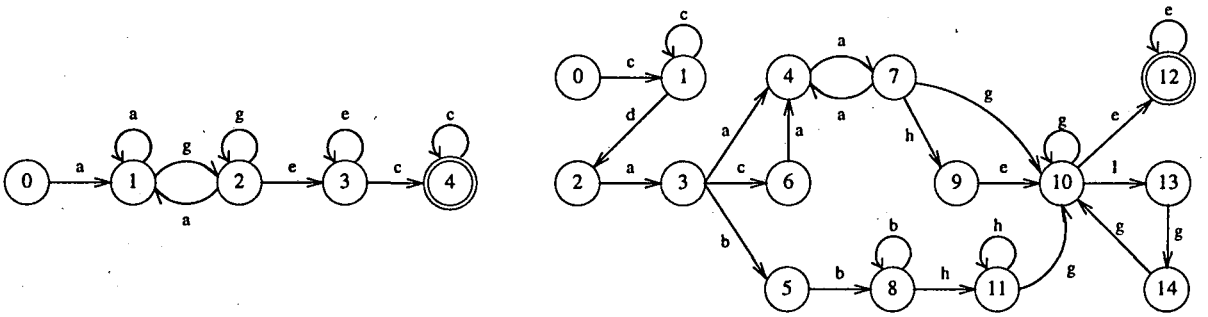
symbol prediction task. Every sample was processed 300 times (training epochs), using a learning rate of 0.005 and momentum of 0.5 in all the runs. A DFA A_{ij} was extracted from each trained net and used to infer an equivalent RE R_{ij}. Finally, an ARE $\tilde{R}_{ij}$ was inferred using R_{ij} to parse the strings in S_{ij}^{+} and the resulting star instances for constraint induction.

	Pos.class		Neg.class		Tot.class		Av.class		Success rate	
	ARE	DFA	ARE	DFA	ARE	DFA	ARE	DFA	ARE	DFA
L_1	100.0	100.0	100.0	75.8	100.0	86.1	100.0	87.9	100.0	-
L_2	87.6	94.0	79.1	49.6	82.6	67.9	83.3	71.8	0.0	-
L_3	100.0	100.0	99.2	44.2	99.5	68.1	99.6	72.1	37.5	-
L_4	85.7	93.5	91.6	48.4	89.0	67.7	88.6	71.0	0.0	-
L_5	100.0	100.0	99.6	72.6	99.8	84.4	99.8	86.3	0.0	-
L_6	100.0	100.0	96.9	59.5	98.2	76.9	98.4	79.7	12.5	-
L_7	94.5	95.6	93.4	70.1	93.9	81.0	93.9	82.8	0.0	-
L_8	93.5	100.0	81.5	59.2	86.6	76.7	87.5	79.6	0.0	-
Mean	95.2	97.9	92.7	59.9	93.7	76.1	93.9	78.9	18.7	-

Table 9.1. *Performance features obtained by the ARE inference method (left) and just the (DFA) regular inference procedure (right) for the test languages.*

To assess the generalization performance of both the inferred ARE $\tilde{R}_{ij}$ and underlying RE R_{ij} (or the previously inferred DFA A_{ij}), an associated test sample was defined as $T_{ij} = (SL_i^{+} - S_{ij}^{+}, SL_i^{-})$ (i.e. the 48 positive examples not used for training and the whole 64 negative examples), and the correct classification rates on T_{ij} were computed. The results of the experiment are summarized in Table 9.1. Five features are displayed for each test language: the former three are averages over the eight learning samples of the correct positive, negative, and total classification rates, respectively; the fourth one refers to the arithmetic mean of the positive and negative classification rates [Dupo:94]; and the fifth one (success rate) is the percentage of times the whole test sample was correctly classified. In such successful cases, and due to the simplicity of the target AREs, it was possible to check that the inferred ARE was equivalent to the target ARE, although an algorithm to check in general the equivalence of two AREs is not available. The last row of Table 9.1 displays the above features averaged over the 8 test languages. For illustrative purposes, Fig. 9.8 shows one of the DFAs and the subsequent ARE inferred for languages L_1 and L_6, together with their classification performance.

Table 9.1 shows the over-generalization carried out by the RGI step, which is indeed desirable to enable the discovery of context constraints afterwards, and the good classification results of the inferred AREs, with average positive, negative and total classification rates above 92%.



$R(V/*) = a(a^{v_1}gg^{v_2}a)^{v_3}a^{v_4}gg^{v_5}ee^{v_6}cc^{v_7}$

$\mathcal{L} = \{v_3 = 1;\ v_4 = v_2;\ v_5 = v_1;$
$v_6 = v_4 + v_5 + 1;\ v_7 = v_4 + v_5 + 1\}$

ARE: Pos. = 100%, Neg. = 100%, Tot. = 100%
DFA: Pos. = 100%, Neg. = 75%, Tot. = 85.7%

(a)

$R(V/*) = cc^{v_1}da((a + ca)(aa)^{v_2}a(g + he)(g^{v_3}lgg)^{v_4}$
$g^{v_5}ee^{v_6} + bbb^{v_7}hh^{v_8}g(g^{v_9}lgg)^{v_{10}}g^{v_{11}}ee^{v_{12}})$

$\mathcal{L} = \{v_4 = 1;\ v_5 = 2/3\ v_1 - 8/3;\ v_6 = 2\ v_2 + 1;$
$v_8 = v_7 + 1;\ v_9 = 1/3\ v_1 - 4/3;\ v_{10} = 1;$
$v_{11} = 2/3\ v_1 - 8/3;\ v_{12} = 2\ v_7 + 3\}$

ARE: Pos. = 100%, Neg. = 95.3%, Tot. = 97.3%
DFA: Pos. = 100%, Neg. = 65.6%, Tot. = 80.4%

(b)

Fig. 9.8 *a) Inferred ARE $\tilde{R}_{11}$ and DFA A_{11} from sample S^+_{11} of L_1 (L-SHAPE).*
b) Inferred ARE $\tilde{R}_{61}$ and DFA A_{61} from sample S^+_{61} of L_6 (COFFEE-POT).

Only for the test languages L_2, L_4 and L_8, the average correct classification rate was below 90%. On the other hand, only L_1 was perfectly learned in all runs, and for the rest of test CSLs, only L_3 and L_6 could be learned in some runs. As it was expected, the inferred AREs notably outperformed the extracted DFA in the correct classification of the negative strings (by a rate difference that approaches to the percentage of "non-regular" negative examples included in the sample), at the expense of a slight impairment in positive string recognition, which is due to the eventual induction of erroneous constraints.

To sum up, the inferred AREs classified quite correctly the test samples of positive and negative strings not used during learning, and a significant improvement in correct classification rate with respect to the inferred DFA or RE was yielded by using the ARE as acceptor. However, although the test CSLs were rather simple, the exact identification of the target language was rarely accomplished. The results of the experiment show that the specific method used for inferring AREs from positive examples could reach a similar generalization performance on simple CSLs than the one displayed by the symbolic RGI methods tested in Chapter 5 on simple regular languages, after processing learning samples of comparable small size. Moreover, the feasibility of the proposed approach for learning CSL acceptors was confirmed.