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 3

Non-regular grammatical inference through symbolic approaches

In this chapter, an overview of the different symbolic approaches that have been proposed previously to learn non-regular languages is given. The reader is referred to the original papers describing the methods for more details (see also the GI surveys by Fu [Fu:82] and Miclet [Micl:90]). The matter of non-regular GI is of great concern since, usually, most of the patterns involved either in syntactic pattern recognition problems or natural language processing are not describable by regular languages.

In some of the methods referenced hereinafter a grammar of a certain class is inferred, whereas in some others, the learning algorithm outputs another type of language representation (e.g. a transition network). All these approaches share the property that the inferred hypothesis is able to represent structures which are beyond the expressive power of regular languages; in addition, the learning procedure uses some type of symbolic representation (maybe extended with probabilistic information). In other words, the GI methods included in this chapter are not based on any class of neural network (subsymbolic representation). The few connectionist approaches to non-regular grammatical inference that have been reported will be commented in the next chapter.

Although the contents of this chapter may be regarded to some extent as miscellanea, it is mandatory to establish a certain classification of the specific topics discussed in order to locate precisely each work within the GI field. The following grouping has been chosen:

a) Context-free grammatical inference (CFGI).
b) Inference of controlled grammars.
c) Inference of transition networks.
d) Inference of pattern languages.

Most of the reviewed material belongs to the first cathegory. The second and third points comprise isolated works that have attempted the inductive inference of (at least some family of) context-sensitive languages; they have been set apart since the two approaches are quite distinct, each using its own type of representation. Finally, the topic of pattern language learning has been included because pattern languages, although not comparable to the Chomsky hierarchy, are able to describe some context-sensitive languages.

3.1 Context-free grammatical inference (CFGI)

It is clear that there is an interest of trying to develop learning algorithms for classes of languages more powerful than the regular languages. This is motivated both by theoretical and practical reasons, as remarked in the previous introduction. Therefore, it appears that the class of context-free languages (CFLs), which follows the class of regular languages in expressive power in the Chomsky hierarchy, is a good candidate for the object space of such inductive inference methods.

CFLs can properly account for subtle language constraints and admit compact representations in the form of context-free grammars (CFGs). Remember that a CFG is a grammar $G = (V_N, V_T, P, S)$, such that every production $p \in P$ is of the form $A \rightarrow \alpha$, where $A \in V_N$, $\alpha \in V^*$ (with $V = V_N \cup V_T$). Moreover, some moderately efficient parsers exist for CFGs, thus allowing the use of CFG models in pattern recognition tasks [BuSa:90]. Other alternative descriptions of CFLs are pushdown automata [HoUl:79], *regular-like* expressions [Salo:73], and basic transition networks [Woods:70].

It should be recalled that identification of CFLs in the limit is theoretically possible using both positive and negative examples [Gold:67], though no efficient method is known so far. On the other hand, CFLs cannot be identified in the limit from only positive data, since they form a superfinite class of languages. As in the RGI case, the negative information, that is needed to control overgeneralization, may be replaced by statistical information gathered from the positive data. Indeed, it is known that stochastic CFLs can be identified in the limit with probability 1 from stochastic presentations, which include no explicit negative examples, but positive examples

that are generated by a stochastic CFG [Horn:69]. Other approach is presentation by informant, where a teacher is available, to which the learning algorithm can ask membership and equivalence queries [Angl:88]; the theoretical bounds of this setting are quite related to the case of complete presentation of CFLs (positive and negative examples), i.e. identification in the limit is possible, but no polynomial-time method is known.

Finally, another way of compensating the lack of negative data consists of presenting the positive examples accompanied with their corresponding unlabeled derivation trees (*skeletons*). These structural descriptions can be obtained by adequately adding brackets to the strings; of course, this requires some knowledge of the target grammar. Alternatively, it may be assumed that the structural descriptions are directly available from the data source. It has been shown that certain classes of CFGs can be identified in the limit from positive samples of structural descriptions (bracketed examples) [CrGM:78,Saka:92]. Also, a well-known result states that, for any CFG G, the set of skeletons of all the strings of $L(G)$ is a regular tree language [Saka:92,Maki:92]. This permits to apply learning algorithms for regular tree languages from the skeletons and to infer afterwards an associated CFG.

Hence, the context-free grammatical inference (CFGI) methods can be classified according to the above paradigms in the following groups:

1) Inference methods for (subclasses of) CFGs from positive structural examples.
2) Inference methods for (subclasses of) CFGs from unstructured data.
3) Inference methods for stochastic CFGs.
4) Inference methods for (subclasses of) CFGs from queries.

The techniques reported within each group are reviewed in next subsections.

3.1.1 CFGI from positive structural examples

Let $G = (V_N, V_T, P, S)$ be a CFG. The *parenthesis grammar* of G, denoted by $[G]$, is formed by replacing every production $A \to \alpha$ by $A \to [\alpha]$, where [and] are special symbols not in V_T. Any string s_+ belonging to $L([G])$ is called a *positive structural example* (or *bracketed example*) of G. Then, a *positive structural sample* of G is a subset of $L([G])$.

It can be seen that a bracketed example of a CFG G corresponds to an unlabeled derivation tree of the grammar G, that is, a derivation tree whose internal nodes have no labels. Let us put it more formally through the introduction of some basic definitions about trees.

A *ranked alphabet* V is a finite set of symbols associated with a finite relation called the *rank relation* $r_V \subseteq V \times \mathcal{N}$. V_n denotes the subset $\{f \in V \mid (f,n) \in r_V\} \subseteq V$, that contains the symbols (also called *function symbols*) of *arity* n.

A *tree* over V is a mapping $t : Dom_t \to V$, where the domain Dom_t is a finite subset of $\mathcal{N}^*$ such that

i) If $x \in Dom_t$, then $\forall y \in \mathcal{N}^*$ such that $\exists z \in \mathcal{N}^*,\ x = yz,\ y \in Dom_t$
(i.e. the domain is closed by prefix);

ii) If $yi \in Dom_t\ \ (i \in \mathcal{N})$, then $yj \in Dom_t$, for $1 \leq j < i$;

iii) $\forall x \in Dom_t,\ t(x) \in V_n$, whenever $xi \in Dom_t \iff 1 \leq i \leq n$
(i.e. the arity constraints are fulfilled).

An element of the tree domain Dom_t is called a *node* of t. If $t(x) = A$, then we say that A is the *label* of the node x of t. A node y of t is called a *terminal node* (or *leaf*) iff $\forall x \in Dom_t,\ x \neq y \Rightarrow \neg\exists z \in \mathcal{N}^*,\ x = yz$. A node x of t is an *internal node* iff it is not a leaf. The *frontier* of Dom_t is the set of all terminal nodes of t, and the *interior* of Dom_t is the set $Dom_t - frontier(Dom_t)$.

Now, let $G = (V_N, V_T, P, S)$ be a CFG and let $V = V_N \cup V_T$. For each symbol $A \in V$, the set $D_A(G)$ of trees over V is recursively defined as

$$D_A(G) = \begin{cases} \{a\} & \text{if } A = a \in V_T \\ \{A(t_1, ..., t_k) \mid A \to B_1 ... B_k,\ t_i \in D_{B_i}(G),\ 1 \leq i \leq k\} & \text{if } A \in V_N \end{cases}$$

A tree in $D_A(G)$ is called a *derivation tree of G from A*, and a tree in $D_S(G) = D(G)$ is simply a *derivation tree of G*.

A *skeletal alphabet* Sk is a ranked alphabet consisting of only the special symbol σ with the rank relation $r_{Sk} \subseteq \{\sigma\} \times \{1, 2, ..., r\}$, where r is the maximum rank (arity) of the symbols in Sk. A tree defined over $Sk \cup V_T$, where V_T is a set of terminal symbols (of arity zero), is called a *skeleton*. It follows that all the internal nodes of a skeleton are labeled by the special label σ, whereas the leaves are labeled by symbols in V_T. Let t be a tree, the *skeletal description* of t, denoted $sk(t)$, is a skeleton with $Dom_{sk(t)} = Dom_t$ such that

$$sk(t)(x) = \begin{cases} t(x) & \text{if } x \in frontier(Dom_t) \\ \sigma & \text{if } x \in interior(Dom_t) \end{cases}$$

Let T be a set of trees. The corresponding *skeletal set*, denoted $K(T)$, is $\{sk(t) \mid t \in T\}$.

A skeleton in $K(D(G))$ is called a *(positive) structural description* of G, and $K(D(G))$ is the set of structural descriptions of G. Two CFGs G and G' are said to be *structurally equivalent* iff $K(D(G)) = K(D(G'))$. If G and G' are structurally equivalent, they are equivalent, too.

Finally, given a CFG G, there exists a bijection ϕ of $L([G])$ onto $K(D(G))$ such that, for each bracketed string $s \in L([G])$, there is a skeleton $\phi(s) = sk(t) \in K(D(G))$ (and viceversa), where t is a derivation tree of G which results from any sequence of productions generating the bracketed string s by the associated parenthesis grammar $[G]$ (recall that productions of G and $[G]$ are mapped one-to-one). Hence, bracketed strings and skeletons (unlabeled derivation trees) are alternative representations for the *positive structural examples* of a CFG G.

The following CFGI methods have been proposed to infer CFGs from positive structural examples:

1) The *abstract profiles* method by Crespi-Reghizzi [Cres:72,CrGM:78].
2) The *RC* algorithm [Saka:92].
3) The *k-TTI* algorithm [RuGa:94].

Crespi-Reghizzi [Cres:72] described a method for inferring a CFG G' in the class of *free operator precedence* grammars from a *positive structural information sequence* of $L(G)$ (i.e. a sequence of strings from $L([G])$). Later, the method was extended to the inference of a more general subclass of CFGs, the *k-distinct k-homogeneous* CFGs [CrGM:78]. If we consider the equivalent skeleton representation for the input data, Crespi-Reghizzi's approach may be regarded as the application of a fixed function, which depends on the subtree rooted at a node and/or the environment of this subtree in the skeleton, whose result is used to label the internal nodes. In order to explain the method more precisely, some more definitions are introduced.

Let $G = (V_N, V_T, P, S)$ be a CFG and let $V = V_N \cup V_T$. Given a string $x \in V^+$, the *leftmost* and *rightmost terminal sets* of x, denoted by $L_t(x)$ and $R_t(x)$, are defined respectively as

$$L_t(x) = \{a \mid x \overset{*}{\Rightarrow}_G a\alpha \ \vee \ (x \overset{*}{\Rightarrow}_G A\beta \ \wedge \ a \in L_t(A))\}$$

$$R_t(x) = \{a \mid x \overset{*}{\Rightarrow}_G \alpha a \ \vee \ (x \overset{*}{\Rightarrow}_G \beta A \ \wedge \ a \in R_t(A))\}$$

where $a \in V_T$, $A \in V_N$, $\alpha, \beta \in V^*$. We can say that $L_t(x)$ (respectively $R_t(x)$) consists of the terminals that are the leftmost (rightmost) in some derivation from x. Now, let $V'_T = V_T \cup \{[,]\}$ and let x be a string $\in (V_N \cup V'_T)^+$. The *left* and *right profiles* of order k of x, denoted by $L_k(x)$ and $R_k(x)$, are defined respectively as

$$L_k(x) = \left\{ u \mid x \overset{*}{\Rightarrow}_{[G]} y_1 \ldots y_m,\ y_i \in V'_T,\ u = \left\{ \begin{array}{ll} y_1 \ldots y_k & \text{if } m \geq k \\ y_1 \ldots y_m \$^{k-m} & \text{if } m < k \end{array} \right\} \right\}$$

$$R_k(x) = \left\{ u \mid x \overset{*}{\Rightarrow}_{[G]} y_1 \ldots y_m,\ y_i \in V'_T,\ u = \left\{ \begin{array}{ll} y_{m-k+1} \ldots y_m & \text{if } m \geq k \\ \$^{k-m} y_1 \ldots y_m & \text{if } m < k \end{array} \right\} \right\}$$

where \$ is a special symbol such that $\$ \notin V'_T$. Finally, the k *profile* of a string $x \in (V_N \cup V'_T)^+$, denoted by $P_k(x)$, is a pair formed by the left and right profiles of order k of x, i.e. $P_k(x) = (L_k(x), R_k(x))$.

A CFG $G = (V_N, V_T, P, S)$ is a *free operator precedence grammar* if it has the following properties:

i) every two distinct nonterminals $A, B \in V_N$ cannot have an identical pair of leftmost and rightmost terminal sets, i.e. $(L_t(A), R_t(A)) \neq (L_t(B), R_t(B))$;

ii) for any two distinct productions of the form $A \rightarrow x$ and $A \rightarrow y$, it holds that $(L_t(x), R_t(x)) = (L_t(y), R_t(y))$;

iii) there are no repeated right parts in the productions;

iv) there may be some rules of the type $S \rightarrow B$, where $B \in V_N$.

A CFG G is *k-distinct* if for any two distinct nonterminals A and B, it holds that $P_k(A) \neq P_k(B)$ (they have a distinct k profile). Grammar G is *k-homogeneous* if for any two distinct productions of the form $A \rightarrow x$ and $A \rightarrow y$, it holds that $P_k(x) = P_k(y)$ (i.e. all the right parts of the same nonterminal have identical k profiles). The languages generated by the class of *k-distinct k-homogeneous* CFGs are a subclass of CFLs called *non-counting* CFLs [CrGM:78].

Crespi-Reghizzi's method for inferring a *free operator precedence* grammar [Cres:72] can be summarized in two steps:

1) For each bracketed string s_i in a positive structural information sequence, construct a CFG G_i that exactly generates s_i in the following manner: substitute a nonterminal A_j for each substring x in s_i such that x contains no left or right brackets (a distinct A_j is created when the pair $(L_t(x), R_t(x))$ is new), and create a production $A_j \rightarrow x$; repeat this process until s_i contains a single nonterminal; then define this nonterminal as the start symbol S.
2) Let the inferred grammar be $G = \bigcup_i G_i$.

In order to infer a *k-distinct k-homogeneous* CFG, the method is changed by replacing the step number 2 above by steps 2 and 3 below:

2) For each grammar G_i (obtained from string s_i), combine all nonterminals having the same k profile, thus yielding a grammar G'_i.
3) Let the inferred grammar be $G' = \bigcup_i G'_i$.

Given a positive structural sample S_s^+, Crespi-Reghizzi's method returns the grammar that generates the smallest language among the grammars in the target subclass that are compatible with S_s^+. It can be shown that the target subclass of CFGs (either *free operator precedence* grammars or *k-distinct k-homogeneous* grammars) can be identified in the limit by the corresponding version of the method from a (complete) positive structural information sequence. Moreover, the algorithm runs in polynomial time and produces a characterizable result. Note, however, that only the subclass of *non-counting* CFLs can be identified in the limit using this method.

Sakakibara presented an efficient method, the *RC* algorithm, to infer a grammar in the class of *reversible* CFGs from a set of positive structural descriptions (skeletons) [Saka:92]. A CFG $G = (V_N, V_T, P, S)$ is said to be *reversible* if and only if

i) G is *invertible*, i.e. $A \rightarrow \alpha$ and $B \rightarrow \alpha$ implies $A = B$ (there are no repeated right parts in the productions), and
ii) G is *reset-free*, i.e. for any two nonterminals $B, C \in V_N$, and $\alpha, \beta \in (V_N \cup V_T)^*$, $A \rightarrow \alpha B \beta$ and $A \rightarrow \alpha C \beta$ implies $B = C$.

Sakakibara proved that for any CFL L, there is a *reversible* CFG G such that $L(G) = L$. In other words, reversible CFGs are a normal form for CFGs, since the class of reversible CFGs *can generate all of the* CFLs. To describe Sakakibara's method, we need to introduce firstly some basic definitions about tree automata.

Let V be a ranked alphabet and m be the maximum rank of the symbols in V. A *tree automaton* over V is a quadruple $A = (Q, V, \delta, F)$ such that Q is a finite set of states ($Q \cap V_0 = \emptyset$), $F \subseteq Q$ is a set of final states, and $\delta = (\delta_0, \delta_1, ..., \delta_m)$ is the state transition function of A, which consists of the following maps:

$$\begin{array}{ll} \delta_0(a) = a & \forall a \in V_0 \text{ (the terminal symbols in } V) \\ \delta_k : V_k \times (Q \cup V_0)^k \rightarrow 2^Q & \text{for k=1,...,m.} \end{array}$$

δ can be extended to V^T, the set of trees over V, by letting

$$\delta(f(t_1, ..., t_k)) = \begin{cases} \{f\} & \text{if } k = 0, \\ \bigcup_{q_1 \in \delta(t_1), ..., q_k \in \delta(t_k)} \delta_k(f, q_1, ..., q_k) & \text{if } k > 0 \end{cases}$$

The terminal symbols on the frontier of the tree are taken as "initial" states. A tree t is *accepted* by A iff $\delta(t) \cap F \neq \emptyset$. The set of trees accepted by A, denoted $T(A)$, is defined as $T(A) = \{t \in V^T \mid \delta(t) \cap F \neq \emptyset\}$. A tree automaton is *deterministic* iff for each k-tuple $q_1, ..., q_k \in (Q \cup V_0)^k$ and each symbol $f \in V_k$, there is at most one element in $\delta_k(f, q_1, ..., q_k)$. It is known that both nondeterministic and deterministic tree automata accept the same class of trees, namely, the class of regular tree languages.

A tree automaton over $Sk \cup V_T$, where Sk is a skeletal alphabet and V_T is a set of terminal symbols, is called a *skeletal tree automaton.* Next, it can be shown that for each CFG $G = (V_N, V_T, P, S)$, a corresponding skeletal tree automaton $A(G) = (V_N, Sk \cup V_T, \delta, \{S\})$, with $\delta_0(a) = a \ \ \forall a \in V_T$ and $\delta_k(\sigma, B_1, ..., B_k) \ni A$ whenever $A \rightarrow B_1...B_k$ is in P, can be defined such that $T(A(G)) = K(D(G))$; that is, the set of trees accepted by $A(G)$ is equal to the set of (positive) structural descriptions (or skeletons) of G. Similarly, for each deterministic skeletal tree automaton $A = (Q, Sk \cup V_T, \delta, F)$ a corresponding CFG $G(A)$ can be built such that $K(D(G(A))) = T(A)$; that is, the set of (positive) structural descriptions of $G(A)$ is equal to the set of trees accepted by A [Saka:92]. Therefore, the problem of learning a CFG from positive structural examples can be reduced to the problem of learning a tree automaton (accepting a regular tree language).

A skeletal tree automaton is said to be *reversible* iff it is deterministic, has at most one final state, and is *reset-free* (see [Saka:92] for the definition of the *reset-free* requirement). Basically, the reversible skeletal tree automaton is the extension of the *zero-reversible* FSA defined by Angluin [Angl:82]. It follows that, if G is a reversible CFG, then $A(G)$ is a reversible skeletal tree automaton such that $T(A(G)) = K(D(G))$. Conversely, if A is a reversible skeletal tree automaton, then a corresponding reversible CFG $G'(A)$ can be constructed such that $K(D(G'(A))) = T(A)$ [Saka:92]. Therefore, the problem of structural identification of reversible CFGs is reduced to the problem of identification of reversible skeletal tree automata.

Given a reversible skeletal tree automaton A, a positive sample CS of A (i.e. a finite subset $CS \subseteq T(A)$) is a *characteristic sample* for A if and only if for any reversible skeletal tree automaton A', $T(A') \supseteq CS$ implies $T(A) \subseteq T(A')$. Sakakibara presented the algorithm RT that, from a finite nonempty set of skeletons Sa, it outputs a reversible skeletal tree automaton $A_f = RT(Sa)$ whose characteristic sample is precisely the input sample Sa. The algorithm RT is an extension of Angluin's learning algorithm for zero-reversible FSA [Angl:82] (already mentioned in Chapter 2). RT first constructs A_0, the base tree automaton for Sa, and then generalizes it by merging states, until it finds the finest partition π_f of the set of states of A_0 with the property that $A_f = A_0/\pi_f$ is reversible. The algorithm RT is efficient, since it may be implemented to run in time of $O(n^3)$, where n is the sum of the sizes of the skeletons in Sa, and the size of a skeleton (or tree) is the number of its nodes. Sakakibara also showed that the algorithm RT may be used at the finite stages of an infinite learning process to identify the reversible skeletal tree automata in the limit from positive samples, simply by running RT on the sample at the i-th stage and proposing the result as the i-th guess [Saka:92]. Moreover, RT can be modified to operate in a full incremental mode, such that A_{i+1} may be obtained from A_i and the skeleton s_{i+1}.

Finally, the algorithm RC to learn reversible CFGs from positive structural examples is simply defined as $RC(Sa) = G(RT(Sa))$, this is, RC returns the reversible CFG G_f corresponding to the reversible skeletal tree automaton A_f inferred by algorithm RT from the set of skeletons Sa. In this case, Sa is shown to be a *characteristic structural sample* for G_f, i.e. for any reversible CFG G, $K(D(G)) \supseteq Sa$ implies $K(D(G_f)) \subseteq K(D(G))$. In the same manner, the time complexity of RC is $O(n^3)$, and RC can be proved to identify in the limit any reversible CFG G from a positive structural presentation of G [Saka:92]. Consequently, the full class of CFLs can be learned efficiently whenever a positive structural presentation of a reversible CFG for the target language is available to the learning algorithm RC. It must be noted that this does not imply that all CFGs can be identified in the limit from positive structural examples, because a given CFG may not have any structurally equivalent reversible CFG, and in such a case, the algorithm RC will fail to identify it.

Ruiz and García have proposed a method to learn another family of CFGs from skeletons of their derivation trees, that is based on another algorithm for inferring regular tree languages (tree automata) from positive samples: the k—TTI algorithm, which can be proved to identify in the limit the class of k-TS (*k-Testable in the Strict sense*) tree sets [RuGa:94]. Let V be a ranked alphabet, let V^T be the set of finite trees over V, and $k \geq 2$. For every tree $t \in V^T$, let $ST(t)$ denote the set of its subtrees. The *k-test vector* of t is defined as $Test_k(t) = (r_{k-1}(t), l_{k-1}(t), p_k(t))$, where

$$r_{k-1}(t) = \begin{cases} t & \text{if } depth(t) \leq k-2, \\ t_r & \text{if } depth(t) > k-2 \end{cases}$$

and t_r is a tree formed from t that contains all the nodes of t with a depth $\leq k-2$ and identical labels;

$$l_{k-1}(t) = \{t' \in ST(t) \mid depth(t') \leq k-2\};$$

and

$$p_k(t) = \begin{cases} \emptyset & \text{if } depth(t) \leq k-2, \\ \{r_k(t') \mid t' \in ST(t) \ \wedge \ depth(t') > k-2\} & \text{if } depth(t) > k-2. \end{cases}$$

An equivalence relation $\equiv_k$ can be defined in $V^{\mathcal{T}}$ as:

$$\forall s,t \in V^{\mathcal{T}}: \quad s \equiv_k t \iff Test_k(s) = Test_k(t).$$

A tree language $T \subseteq V^{\mathcal{T}}$ is said to be *k-Testable* (k-T) iff it is the union of some of the equivalence classes defined by $\equiv_k$, and it is said to be *k-Testable in the Strict sense* (k-TS) iff there exist three finite sets R, L, and P, such that $\forall t \in T: \ r_{k-1}(t) \in R$, $l_{k-1} \subseteq L$, $p_k(t) \subseteq P$.

Given a sample $S \subseteq V^{\mathcal{T}}$, let $V(S)$ be the ranked alphabet from S, and let $R_k(S)$, $L_k(S)$, $P_k(S)$ be three finite sets defined constructively as follows

$$R_k(S) = \{r_{k-1}(t) \mid t \in S\}, \quad L_k(S) = \bigcup_{t \in S} l_{k-1}(t), \quad P_k(S) = \bigcup_{t \in S} p_k(t).$$

Now consider that $S \subseteq V^{\mathcal{T}}$ is a set of skeletons, and thus, $V(S) = Sk \cup V_T$, where Sk is a skeletal alphabet and V_T is a set of terminal symbols (of arity zero). Given S and $k \geq 2$, the $k-$TTI algorithm returns a deterministic skeletal tree automaton $A_k(S) = (Q, V(S), \delta, F)$, where the set of states is $Q = R_k(S) \cup L_k(S) \cup P_{k-1}(P_k(S))$, the subset of final states is $F = R_k(S)$, and the transition function δ is defined by
i) $\forall t \in L_k(S): \ \delta_0(t) = t$, and
ii) $\forall \sigma(t_1, ..., t_n) \in P_k(S): \ \delta_n(\sigma, t_1, ..., t_n) = r_{k-1}(\sigma(t_1, ..., t_n))$.

The $k-$TTI inference algorithm satisfies the following properties:

a) $S \subseteq T(A_k(S))$;
b) $T(A_k(S))$ is the smallest k-TS tree set containing S;
c) $S' \subseteq S \implies T(A_k(S')) \subseteq T(A_k(S))$
d) $T(A_{k+1}(S)) \subseteq T(A_k(S))$
e) If $k > 1 + \max_{t \in S}\{depth(t)\}$ then $T(A_k(S)) = S$.

Ruiz and García have presented some preliminary experimental results comparing the behaviour of the $k-$TTI and *RT* algorithms when they are used to infer CFLs from derivation tree skeletons of arbitrary (not necessarily k-TS or reversible) CFGs [RuGa:94]. Algorithm *RT* classified correctly about 90% of the test strings and about 5% more strings than $k-$TTI (for $k = 3$), but the run time of the former was much longer than the latter. For greater values of k, the algorithm $k-$TTI achieved better classification rates, at the expense of less generalization and longer computation time. It was also shown that the CFGs obtained with both algorithms classify correctly all the positive strings of the target language, and they only may fail in over-generalizing it when the grammar is not of the appropriate subclass.

3.1.2 CFGI from unstructured data

3.1.2.1 Inference of even-linear grammars

A well studied subclass of CFLs is the class of even linear languages (ELLs), introduced by Amar and Putzolu [AmPu:64]. The class of ELLs properly contains the class of regular languages and is properly contained in the class of linear languages. The outstanding feature of ELLs is that they possess similar set-theoretic properties as regular sets; namely, the class of ELLs is closed under Boolean operations, and, for two ELLs L_1 and L_2, the questions $L_1 \subset L_2$ and $L_1 = L_2$ are solvable.

A grammar $G = (V_N, V_T, P, S)$ is *linear* iff every production $p \in P$ is of the form $A \rightarrow x$ or $A \rightarrow yBz$, where $A, B \in V_N$, $x, y, z \in V_T^*$. Grammar G is *even-linear* iff it is linear and for all the productions of the form $A \rightarrow yBz$, it holds that $|y| = |z|$. Linear and even linear languages are generated by linear and even linear grammars (ELGs) respectively. Given an ELG, there exists an equivalent ELG where every production is in one of the two forms: i) $A \rightarrow aBb$, where $A, B \in V_N$, $a, b \in V_T$; or ii) $A \rightarrow a$, where $A \in V_N$, $a \in V_T \cup \{\lambda\}$ [SeGa:94]. Another normal form for ELGs is obtained by replacing the above second rule type by productions of the forms $S \rightarrow \lambda$, $A \rightarrow a$, and $A \rightarrow ab$, where $A \in V_N$, $a, b \in V_T$ [Taka:88]; note that $S \rightarrow \lambda$ is only required when $\lambda \in L(G)$. Some works, listed below, have focused on the learning problem of ELLs.

1) The *"skeleton"* method by Radhakrishnan and Nagaraja [RaNa:88].
2) The *regular control set* method by Takada [Taka:88].
3) The *transformation* method by Sempere and García [SeGa:94].

As explained in Chapter 2, Radhakrishnan and Nagaraja proposed a constructive characterizable technique for RGI, that allows the identification of the subclass

of *terminal distinguishable regular languages* (TDRLs) [RaNa:87]. These authors extended this method to the inference of ELLs [RaNa:88]. Their method starts with the construction of the skeletons corresponding to a set of positive examples[1]. The procedure then assigns nonterminals to the nodes of the skeletons. For assigning one nonterminal to more than one node (i.e. to generalize), the method uses an equivalence relation taking into account the context of the subskeleton rooted at these nodes.

The *"skeleton"* learning method for ELLs [RaNa:88] has the following properties:

- It does not identify in the limit the class of ELLs (obviously, since ELLs are a superfinite language class and a positive presentation is given).
- A pseudo-incremental version is available for on-line learning.
- If the sample is from some non-even linear language, the method does not generalize and it infers in the limit a grammar with an infinite number of productions and nonterminals; hence, target non ELGs can be detected.

The skeleton method was applied for the inference of simple objects represented in the Picture Description Language (PDL) [MaRW:82], and the authors showed how to use the method in a hierarchical manner in order to infer CFGs more complex than ELGs [RaNa:88].

Let $\Sigma = \{a_1, ..., a_m\}$ be an alphabet. A *universal* ELG over Σ is an ELG $U = (\{S\}, \Sigma, \Psi, S)$ such that Ψ consists of the following productions:

$$\Psi = \{\psi_k : S \rightarrow aSb, \forall a, b \in \Sigma\} \cup \{\psi_i : S \rightarrow a, \forall a \in \Sigma\} \cup \{\psi_\lambda : S \rightarrow \lambda\}$$

For any alphabet Σ, it follows that $L(U) = \Sigma^*$, U is unambiguous and unique up to renaming of the start symbol S, and U has $m^2 + m + 1$ number of productions.

Let $G = (V_N, \Sigma, P, S)$ be a grammar (e.g. an ELG) over Σ. A subset C of P^* is called a *control set* on G and $L(G, C) = \{\omega \in \Sigma^* \mid S \overset{\alpha}{\Rightarrow}_G \omega \ \wedge \ \alpha \in C\}$ is called the *language generated by G with the control set C*, where $x \overset{\alpha}{\Rightarrow}_G y$ denotes a derivation from x to y ($x, y \in (V_N \cup \Sigma)^*$) by applying a sequence of productions $\alpha = p_1 ... p_j \in P^*$.

Takada demonstrated that every ELL L over an alphabet can be generated by the universal ELG U over the alphabet with a *regular control set* C that regulates the application of its rules (i.e. a regular language over the alphabet of production labels

[1]Note the apparent contradiction of including the *"skeleton"* method within the group of CFGI techniques from unstructured data. This is explained by the fact that the input data do not need to be skeletons but raw positive strings, since a unique skeleton can be determined for each example if target ELGs are in normal form (the same occurs in the case of regular grammars).

of U) [Taka:88]. Furthermore, for any ELL L there exists a unique regular control set C with which a universal ELG U generates L. This permits to reduce the problem of learning ELLs to the RGI problem, since to identify an unknown ELL, a GI algorithm for ELLs has only to identify its corresponding unknown regular control set.

Takada's method [Taka:88] is based on converting the input strings into rule strings by parsing through the universal ELG U, learning the regular control set C using any RGI algorithm, and obtaining the ELG G that generates the same language than U with the control set C. If the RGI method used identifies in the limit the class of regular languages [Angl:87,OnGa:92b], then the GI algorithm for ELLs will also identify in the limit the class of ELLs. To that end, it should be noted that both positive and negative strings [OnGa:92b] or queries answers [Angl:87] must be provided. The whole procedure can be run in polynomial time. Obviously, Takada's method cannot identify ELLs from only positive data.

Let us define a transformation $\sigma : \Sigma^* \rightarrow (\Sigma^2 \cup \Sigma)^*$ recursively as follows: i) $\sigma(a) = a$, $\forall a \in \Sigma \cup \{\lambda\}$, ii) $\sigma(axb) = ab \cdot \sigma(x)$, $\forall a, b \in \Sigma$, $\forall x \in \Sigma^*$, where the operation $\cdot$ denotes concatenation. By extension, given a language L, $\sigma(L) = \{\sigma(x) \mid x \in L\}$. A similar definition and extension is straightforward for the inverse transformation σ^{-1}, and it follows that $\sigma^{-1}(\sigma(L)) = L$. Sempere and García [SeGa:94] demonstrated that if $L \subseteq \Sigma^*$ is an ELL, then $\sigma(L)$ is a regular language (over an alphabet $\Sigma' = \Sigma^2 \cup \Sigma$).

The *transformation* method by Sempere and García [SeGa:94] is also based on reducing the ELL learning problem to the RGI problem. Given a sample S of an ELL, the transformation σ above is applied and a regular language sample $\sigma(S)$ (over alphabet Σ') is obtained. Then, any RGI algorithm can be applied on the transformed sample, and from the inferred FSA A and the inverse transformation σ^{-1}, an ELG G is determined such that $L(G) = \sigma^{-1}(L(A))$. Again, as in Takada's method, ELL (efficient) identification in the limit is possible by using a regular language (efficient) identification method, and this requires the presentation of positive and negative data (either by examples or queries answers). In their work, Sempere and García also proposed a characterization of the class of ELLs, using a relation of finite index, which allows to associate a canonical minimal-size deterministic ELG in a normal form with any given ELL.

3.1.2.2 Inference of other subclasses of CFGs from positive examples

Also within the group of GI techniques that have been proposed to infer CFGs of a certain subclass from unstructured data, we may mention the two following heuristic methods:

4) The *pivot grammar* inference method by Gips [FeGH:69].
5) The *non-recursive CFG* inference method by Chirathamjaree [ChAc:80].

Both methods are provided with just a positive sample S^+ and apply an *ad-hoc* criterion to construct a CFG in the target class.

In an early work, Gips [FeGH:69] described a heuristic method to infer a *pivot grammar* from positive examples. A pivot grammar is a grammar $G = (V_N, V_T, P, S)$, in which the set of terminals V_T can be partitioned into two disjoint sets, denoted by V_{TP} and V_{T0}, and the allowed types of production rules are of the forms: i) $A \rightarrow BaC$, ii) $A \rightarrow Bb$, iii) $A \rightarrow bB$, or iv) $A \rightarrow b$, where $A, B, C \in V_N$, $a \in V_{TP}$ and $b \in V_{T0}$. The class of pivot grammars properly contains the class of linear grammars and is properly contained in the general class of CFGs. Hence, pivot grammars should be more difficult to infer than ELGs.

Gips' method is based on finding the self-embedding in the sample S^+. The method is carried out in two steps. In the former, each string in S^+ is examined to check if it has a proper substring that appears in S^+; if it does not, it becomes a string in the working set, otherwise the longest such substring is replaced by a nonterminal and the resulting string is placed in the working set. In the second step, a simple pivot grammar is built for the working set. It is clear that the method is not well-founded theoretically, and the initial choice of S^+ is quite critical.

Chirathamjaree and Ackroyd have proposed an incremental constructive method to infer a *non-recursive* CFG in Chomsky normal form from a set of positive examples [ChAc:80]. Their method is a kind of error-correcting (conservative) procedure. The first input string s_1 is used to build an initial CFG G_1, which generates only that string. The inference procedure is then applied iteratively, with the $(i+1)$-th input string s_{i+1} being matched against the i-th CFG G_i. The matching process involves the computation of a *minimization matrix* M, that reveals the shortcomings of G_i related to the generation of s_{i+1}. If G_i generates s_{i+1} (a certain element of M is 0) then $G_{i+1} = G_i$, otherwise, the information from the M matrix is used to augment G_i by appending the minimal number of additional terminals, nonterminals and rules, as appropriate, to ensure that the augmented CFG G_{i+1} generates the string s_{i+1}. This process is repeated until all strings in the positive sample have been processed.

Chirathamjaree's method always returns a non-recursive CFG $G = (V_N, V_T, P, S)$, in which nonterminals are hierarchically ordered and the types of production rules are restricted to *terminating* rules $A \rightarrow a$ $(A \in V_N, a \in V_T)$ and *bielement* rules $A \rightarrow BC$ $(A, B, C \in V_N)$ such that B, C, or both, are in the terminating rules. In this way, G only generates a finite set of finite-length strings, that includes the sample strings together with other strings that resemble them. The method is efficient and can be

easily extended to the construction of stochastic CFGs. The authors claimed that their method is appropriate for pattern recognition problems where finite-length strings only are involved (e.g. recognition of isolated spoken words).

3.1.2.3 *Inference of CFGs from positive and negative examples*

Two closely related methods based on the *version space* induction algorithm by Mitchell [Mitc:82] have been reported that approach the problem of inferring CFGs from both positive and negative (unstructured) examples:

6) The *derivational version space* method by Vanlehn and Ball [VaBa:87].
7) The *structural containment version space* method by Giordano [Gior:94].

In both methods, a sample $S = (S^+, S^-)$ is presented incrementally, and the method maintain a set of CFG hypotheses that are compatible with the examples seen so far and that are the most general grammars in a certain class of CFGs, according to a certain generalization partial ordering.

Many induction problems can be stated as the search of one or more hypotheses, that are consistent with a set of positive and negative examples (instances), in a partially ordered set of hypotheses which contains the possible generalizations of the instances. Instead of the simple linear list used in identification by enumeration, the partial ordering organization of the hypotheses space allows the elimination of more hypotheses than just the current one when an incompatibility with the examples is detected. For example, if a grammar G does not generate a string in the target language (positive example), all the grammars *covered* by G (*more-specific-than* G) will not generate it either and they can be discarded. In Chapter 2, we have seen that the problem of regular GI can be characterized as a search in a partially-ordered space by defining a lattice of FSAs derived from a canonical automaton [DuMV:94]. Likewise, many RGI methods have used the inherent pruning technique associated with such an algebraic structure of the hypotheses space [BiFe:72, PaCa:78, Micl:80, Angl:81, Angl:82, OnGa:92, MiGe:94].

The version space formalism developed by Mitchell [Mitc:82] allows a systematic exploration of a set of hypotheses that potentially describe the examples given so far. Indeed, a *version space* is just defined as the set of all the hypotheses (generalizations) that are consistent with a given set of instances in a set partially ordered by a *generalization* predicate. However, the *version space strategy* proposed by Mitchell is a particular induction algorithm, that takes advantage of the mentioned pruning possibility, and which is based on a compact way of representing the version space.

This compact representation consists of keeping only the most general and most specific descriptions at each step of the presentation. Let $M(h, i)$ be a *matching* predicate of two arguments, a hypothesis h and an instance i, that is true if the hypothesis matches the instance. A *generalization* relation can be defined in terms of the matching predicate: a hypothesis h_1 is *less general than* a hypothesis h_2, denoted $h_1 \ll h_2$, if h_1 matches a subset of the elements matched by h_2.

Given a presentation of instances, the version space for that presentation is partially ordered by the $\ll$ relation and it has a subset of minimal elements $\mathcal{S}$ and a subset of maximal elements $\mathcal{G}$. This is, the sets $\mathcal{S}$ and $\mathcal{G}$ contain the most specific and the most general consistent generalizations, respectively. The pair $(\mathcal{S}, \mathcal{G})$ can be used to represent the version space, since it can be proved that, given a presentation, a hypothesis h is contained in the version space for that presentation iff there is some $s \in \mathcal{S}$ and $g \in \mathcal{G}$ such that $s \ll h$ and $h \ll g$. Then, the hypotheses space is explored by means of generalization/specialization operators. These operators permit to maintain the current subsets $\mathcal{S}$ and $\mathcal{G}$ of the version space for the presented instances by producing the most specific generalization/most general specialization of a given hypothesis. A function $Update((\mathcal{S}, \mathcal{G}), i) \rightarrow (\mathcal{S}', \mathcal{G}')$ was given by Mitchell [Mitc:82], that takes the current version space boundaries and an instance i which is marked as either positive or negative, and it returns the boundaries for the new version space. The implementation of the *Update* function in a particular induction problem depends both on the representation language of the hypotheses and on the generalization/specialization operators.

In principle, there are two important problems to apply the version space approach to the inference of CFGs from examples:

a) The size of the version space containing all the consistent CFGs is infinite.
b) The *less general than* relationship, that compares the strings generated by two grammars, also called *weak containment*, is undecidable for CFGs.

The first problem above can be solved by considering just *reduced simple* CFGs in the search space [VaBa:87]. A CFG G is *simple* iff i) no rule has an empty right hand side; ii) if a rule has just one symbol on its r.h.s., then the symbol is a terminal; and iii) every nonterminal appears in a derivation of some string. *Simple* CFGs can generate all the CFLs [HoUl:79]. Given a presentation of examples P, a grammar is *reduced* if it is consistent with P and there is no proper subset of its rules that is consistent with P. Vanlehn and Ball proved that given a finite presentation P, there are finitely many *reduced simple* CFGs consistent with P, and consequently, the $\mathcal{S}$ and $\mathcal{G}$ sets of the *reduced version space* are each finite [VaBa:87]. However, Vanlehn and Ball could not find an *Update* algorithm for some generalization predicate on CFGs that could maintain the boundaries $\mathcal{S}$ and $\mathcal{G}$ of exactly the reduced version space.

Instead, they defined and used the so-called *derivational version space*, that was proved to be a finite superset of the reduced version space for any finite presentation. Given a set of positive examples, the *simple tree product* is defined as the Cartesian product over the sets of *simple* unlabeled derivation trees (i.e. *simple* skeletons) that are possible for each string, where a *simple* skeleton (of a *simple* CFG) does not include any node having a single son that is not a leaf (terminal). Then, given a set of positive and negative strings, the *derivational version space* is the set of simple CFGs corresponding to all possible labelings of each skeleton sequence in the simple tree product for the positive strings minus those CFGs that generate any of the negative strings [VaBa:87].

Next, a partial order for this set was given by the following covering relation [VaBa:87]: given two CFGs G_1 and G_2, G_1 *is fast_covered by* G_2 iff i) both grammars are obtained by labeling the same sequence of skeletons, and ii) the partition of the set of (interior) nodes caused by the labeling of G_1 is a refinement of the partition caused by the labeling of G_2. It can be shown that G_1 *is fast_covered by* G_2 implies G_1 *is less general than* G_2 (i.e. $G_1 \ll G_2$). Moreover, to the contrary of $\ll$, the *fast_cover* relation is computable, thus solving the second problem aforementioned.

The derivational version space under the *fast_cover* relation can be seen as a set of separate partition lattices (slices), one lattice (slice) for each skeleton sequence in the simple tree product, because the *fast_cover* relation can only be true inside the slices, and each slice has one maximal (top) partition, containing just one class for all the nodes, and one minimal (bottom) partition, containing singleton classes. If no negative instance is given, then $\mathcal{G}$ consists of the top partition in each lattice. As negative strings are presented, the $\mathcal{G}$ set expands and the derivational version space shrinks (as the maximal generalizations for each slice may descend in the lattice). The set $\mathcal{S}$ always consists of the bottom partition in each lattice, and it is not affected by the presentation of negative instances. On the other hand, the presentation of positive strings increases both the number of partition lattices (slices) and the sizes of the partition lattices. As new positive strings are presented, the set $\mathcal{S}$ trivially grows, while the set $\mathcal{G}$ may grow or not, since the number of maximal sets grows but the size of the maximal set for each slice may decrease.

Vanlehn and Ball described an *Update* algorithm for the above version space, which is basically aimed at maintaining the set $\mathcal{G}$ (of maximal generalizations), whereas the set of all the positive strings presented is stored instead of the uninteresting set $\mathcal{S}$. In order to find the most general specializations of the members of $\mathcal{G}$, a *splitting* specialization operator is used that consists of dividing an element of a partition in two elements. However, the cost of the proposed *Update* algorithm is extremely high, specially in the process of updating $\mathcal{G}$ when a new positive string is given (which also requires the storage of all the negative strings presented so far).

Despite its theoretical interest, the method described by Vanlehn and Ball is absolutely unpracticable, not only due to the exponential breadth-first search that is inherent in Mitchell's version space approach, but also because of the huge combinatorial explosion of different skeleton sequences and the number of partitions for each sequence, that affect the cost of the *Update* algorithm. It must be noted that if the skeletons of the positive examples are given in the presentation, then just one partition lattice or slice is involved, and the cost may be considerably reduced (though it still is very high).

Giordano has proposed another type of version space for the inference of CFGs from positive and negative examples, which is based on the *structural containment* relation [Gior:94]. This generalization relation is computable in polynomial time on a normal form of CFGs, the *uniquely invertible* CFGs.

We have seen in section 3.1.1 that two CFGs G_1 and G_2 are said to be *structurally equivalent* iff the set of skeletons of the grammar derivation trees is the same for both, i.e. $K(D(G_1)) = K(D(G_2))$. Similarly, a CFG G_1 is *structurally contained in* a CFG G_2 iff the set of skeletons generated by G_1 is contained in the set of skeletons generated by G_2, $K(D(G_1)) \subseteq K(D(G_2))$. In other words, G_2 produces the strings generated by G_1 in the same way (in terms of structure, not necessarily of the involved nonterminals) and possibly more strings. It is clear that *structural containment* implies *weak containment*, i.e. $K(D(G_1)) \subseteq K(D(G_2)) \implies G_1 \ll G_2$.

A CFG G is *uniquely invertible* iff no two productions in G have the same right-hand side. The uniquely invertible CFGs can represent all the CFLs (so they are a normal form for CFGs). Furthermore, a polynomial time algorithm is available [Gior:94] that determines whether, given two uniquely invertible CFGs G_1 and G_2, G_1 is structurally contained in G_2 or not. Therefore, uniquely invertible CFGs can be considered as hypotheses and structural containment as a generalization relation in a version space approach to CFG inference. However, in order to limit the size of the search space, Giordano used uniquely invertible CFGs with no useless symbol and a *fixed number of nonterminals*, which is set a priori arbitrarily.

As in Vanlehn and Ball's method, the *Update* algorithm for Giordano's version space is aimed at maintaining the set $\mathcal{G}$ of maximal generalizations consistent with the given examples, and only specialization operators are defined, which allow to find the most general specializations of the members of $\mathcal{G}$ (to be used when some of the grammars in $\mathcal{G}$ match a negative example). On the other hand, a given positive string just removes from $\mathcal{G}$ the CFGs that do not generate it. Giordano did not explain clearly how the set $\mathcal{G}$ is initialized in his method, although it seems that the set of CFGs in Chomsky normal form that are uniquely invertible and contain the assumed number of nonterminals may be selected as the initial $\mathcal{G}$.

In order to reduce the redundancy in the set of hypotheses, Giordano studied the nature of the equivalence classes induced by structural containment on the search space, and he defined representatives of these classes (though each class may have several representatives indeed), so that his algorithm only processed these grammars. To this end, the notions of *equivalent symbols* and *context of a nonterminal* were used.

Let $G = (N, \Sigma, P, S)$ be a uniquely invertible CFG and $[G] = (N, \Sigma', P', S)$ its associated parenthesis grammar, where $\Sigma' = \Sigma \cup \{[,]\}$ and P' is obtained by enclosing the right hand sides of the productions in P with parentheses. Given a nonterminal $A \in N$, the *context* of A, denoted $C(A)$, is the set $\{(\omega_1, \omega_2) \in \Sigma'^* \times \Sigma'^* \mid S \overset{*}{\Rightarrow}_{[G]} \omega_1 A \omega_2\}$. Two nonterminals $A_1, A_2 \in N$ are *equivalent symbols* iff they have the same contexts, i.e. iff $C(A_1) = C(A_2)$. Also, A_1, A_2 are two equivalent nonterminals of G iff the grammar G' obtained by replacing each occurrence of A_1 and A_2 in P by a nonterminal symbol not in $N - \{A_1, A_2\}$ is structurally equivalent to G. There is an algorithm that determines in polynomial time whether two given nonterminals of a uniquely invertible CFG are equivalent [Gior:94].

If G has no equivalent symbols, then all the most general specializations of G are *reduced uniquely invertible CFGs with no equivalent symbols* that can be obtained by changing left-hand sides of productions or deleting productions of G. If G has some equivalent symbols, then the most general specializations of G are obtained by applying the same specialization operators to each of the representatives of G, where these representatives are taken in principle as a maximal set of grammars equivalent to G and not isomorphic between them. However, the computation of the whole set of representatives of G (by processing equivalent symbols in different ways) can be too costly, and Giordano suggested to apply heuristics to select just a few of them.

Hence, two types of specialization operators were considered: *substitution* and *deletion*. If A, B are two nonterminals of G such that $C(B) \subseteq C(A)$, a grammar structurally contained in G (i.e. a specialization) is obtained by changing A into B in the left-hand side of any production of G. Giordano presented a polynomial algorithm that determines the pairs (A, B) of nonterminals satisfying $C(B) \subseteq C(A)$ for a given grammar G. Then, given a set of possible substitutions in G, the set of maximal specializations of G reached by substitutions may be computed [Gior:94]. The set of maximal specializations reached by deletion are obtained by deleting only one rule, unless its left-hand side may be changed. When one of the preceding transformations leads to a CFG G' with useless symbols, then G' must be reduced (thus decreasing the number of nonterminals), and therefore, some grammars with the preset number of nonterminals and equivalent to the produced grammar must be introduced in the set $\mathcal{G}$.

Although the version space approach by Giordano seems to involve much less computational burden than the one by Vanlehn and Ball, the author acknowledged that its efficiency is not clear on significant samples (presumably exponential due to the breadth-first search). Moreover, some heuristics have to be used to limit the growth of the search space, thus impeding the identification in the limit. In particular, the requirement of fixing the number of nonterminals seems too restrictive. In addition, the computation and handling of the representatives of the classes of structurally equivalent grammars is somewhat obscure.

To end this subsection, I will also mention a recently reported work that investigated the application of the *genetic algorithm* (GA) to the inference of CFGs from a sample of both positive and negative strings [Wyard:94]. The paper by Wyard focused mainly on the representational issues of the problem, which include three interrelated factors:

- the selected type of CFGs (among normal forms) and their encoding in the chromosomes corresponding to the individuals of the evolving population;
- the choice of genetic operators and parameters; and
- the evaluation function chosen to determine the fitness of the individuals.

Wyard presented some encouraging results on the inference of some small CFGs from positive and negative examples using a GA and taking some decisions about the above factors, at the expense of a very long run-time ("the evaluation of a generation may take a number of hours on a SPARC station"). However, the challenge remains to devise suitable representations to enable the GA to operate effectively on any CFL, and to learn non-small CFGs in reasonable time [Wyard:94].

3.1.3 Inference of stochastic CFGs

A *stochastic CFG* is a grammar $G = (V_N, V_T, P_s, S)$ such that $|V_N| = |P_s|$ and for every nonterminal $A_j \in V_N$ there is a production of the form $A_j \longrightarrow \alpha_{j1}|...|\alpha_{jl_j}$ $(p_{j1}, ..., p_{jl_j})$, where $\alpha_{ji} \in (V_N \cup V_T)^*$, $\sum_{i \in [1,l_j]} p_{ji} = 1$, and the basic rewriting rule $A_j \longrightarrow \alpha_{ji}$ (p_{ji}) means that A_j can be rewritten in α_{ji} with probability p_{ji}. For every string $x \in V_T^*$ that is generated by a stochastic CFG G, there is an associated probability $P(x|G)$, computed as the sum over all possible derivations leading to x of the products of the probabilities of the rules used in each derivation. A *stochastic positive sample* S_s of a stochastic CFG G is a set of strings (possibly including some repetitions) such that each string $s \in S_s$ is stochastically generated according to the rule probabilities of G.

Two classical methods for the problem of inferring a stochastic CFG from a stochastic positive sample will be recalled here:

1) The *enumerative Bayesian algorithm* by Horning [Horn:69].
2) The *hill climbing* method by Cook *et al.* [CoRA:76].

A Bayesian approach to an inductive inference problem requires certain probability measures and it is based on looking for a hypothesis that maximizes the conditional probability of a hypothesis given that a particular sample is observed. Let $P(h)$ be a probability measure defined on the hypothesis space, let $P(S)$ be a probability measure defined on the sample space, and let $P(S|h)$ be the probability with which a given sample S can be generated by a given hypothesis h. By using Bayes' Theorem, the a-posteriori conditional probability $P(h|S)$ of hypothesis h when the sample S is observed can be computed as

$$P(h|S) = \frac{P(h)P(S|h)}{P(S)}$$

Therefore, in order to maximize $P(h|S)$, it suffices to maximize $P(h)P(S|h)$, and it is not required to know the sample a-priori probability $P(S)$. To use this approach, a computable way to assign probabilities to the elements of the hypothesis space is needed. Typically, $P(h)$ will be a measure of the simplicity of the hypothesis h, with higher probability assigned to simpler hypotheses, and $P(S|h)$ will be a measure of the goodness of fit of h to the sample S, with higher probability assigned to a better fit. Consequently, a Bayesian approach can combine the measures of simplicity of the hypothesis and goodness of fit to the data to select the "best guess" for a given sample.

Horning used the above Bayesian approach for the inference of stochastic CFGs [Horn:69]. In this case, the probability of a sample S given a stochastic grammar G is defined as $P(S|G) = \sum_{x \in S} P(x|G)$ if all the strings in S can be generated by G, and $P(S|G) = 0$ otherwise, and the probability measure $P(G)$ is defined by a stochastic CFG generator. Horning described an enumerative algorithm for finding G to maximize $P(G)P(S|G)$, given a sample of strings S as input, and he proved that, if the input is a stochastic positive sample S_s, then the enumerative Bayesian algorithm leads to correct identification in the limit (for larger and larger samples) of the target stochastic language with probability 1. Note that even though CFLs are not identifiable in the limit from positive presentations, stochastic CFLs are identifiable in the limit with probability 1 from stochastic positive presentations. The enumerative Bayesian algorithm determines at most one of a set of equivalent stochastic CFGs maximizing $P(G|S_s)$, but, unless the class of considered stochastic CFGs be dramatically restricted, the method is absolutely unpractical due to the huge computational cost of enumerating and testing all grammars in the class.

Cook *et al.* proposed a hill climbing method for inferring a stochastic CFG G from a stochastic positive sample S_s [CoRA:76]. A detailed description of their method can also be found in the survey by Miclet [Micl:90]. Cook's method is based on two measures: a measure of grammar *complexity* $C(G)$, which is derived from information theory concepts, and that only involves the information contained in the production rules of G; and a measure of the *discrepancy* $D(L(G), S_s)$ between the language generated by G and the given sample S_s, whose computation requires the knowledge of all the strings in $L(G)$ (together with their associated probabilities) up to the length of the longest string in S_s. To search for the optimal grammar, a positive linear combination of these two measures (e.g. $M(G, S_s) = C(G) + D(L(G), S_s)$) is taken as the global measure $M(G, S_s)$ to be minimized.

Cook *et al.* also defined several grammar transformations (substitution, disjunction, reduction) designed to decrease the global measure $M(G, S_s)$ without changing very much the generated language, in order to constrain the search space to the set of possible solutions (stochastic CFGs generating all the strings in S_s). Their algorithm starts with a canonical grammar, that generates precisely the observed sample, with high complexity and low discrepancy. Then, it searches the neighborhood that consists of all the grammars obtained by applying a single transformation to the current grammar. If any of these transformed grammars is better than the current one according to the measure $M(G, S_s)$, the procedure moves to the best grammar in the neighborhood and iterates the search. Otherwise, the algorithm stops and outputs the locally optimal grammar that has been found. Again, the computational cost is very great, and therefore, the Cook's method is not generally applicable.

In addition to the preceding methods that try to infer directly a stochastic CFG from a stochastic positive sample, a two-step general approach can also be followed to infer stochastic CFGs, as it is commented next.

A *stochastic CFG* $G = (V_N, V_T, P_s, S)$ can be seen as a pair of two components $G = (G_c, q)$, where G_c is the *characteristic grammar* corresponding to G, this is the CFG $G_c = (V_N, V_T, P, S)$ obtained by removing all the probabilities of the productions, and q is a function $q : P \to (0, 1]$ (where P contains basic rewriting rules of the form $A \to \alpha$, $A \in V_N$, $\alpha \in V^*$) such that $\forall A \in V_N : \sum_{A \to \alpha \in P} q(A \to \alpha) = 1$.

Therefore, both components of a stochastic CFG can be learned from a sample $S = (S^+, S^-)$ (with maybe $S^- = \emptyset$) by first using a (non-stochastic) CFGI algorithm to infer the characteristic grammar G_c from S and then a *probability estimation* technique to infer the corresponding rule probabilities from S^+ and G_c.

The most popular technique for probabilistic estimation of stochastic CFGs is the Inside-Outside algorithm (see e.g. [LaYo:90,LaYo:91]). However, for the application of this algorithm, the characteristic CFG must be in Chomsky normal form. Casacuberta [Casa:94] has extended the applicability of this technique to the probability estimation of stochastic CFGs with *simple* characteristic CFGs (the definition of a *simple* CFG has been given in section 3.1.2). The extension is based on using the well-known transformation from a simple CFG to a CFG in Chomsky normal form [AhUl:72], prior to the Inside-Outside algorithm, and a posterior (easy) assignment of probabilities to the rules of the original grammar. Casacuberta showed that the extended algorithm does not bear a significant increase in the time complexity with respect to the Inside-Outside algorithm. Kupiec [Kupi:92] has proposed a different method to estimate the probabilities of an unrestricted stochastic CFG, which is based on an extension of the concepts used in the estimation of parameters of Hidden Markov Models to a type of Transition Networks (see section 3.3 for the definition of Transition Networks and their relationship with CFGs).

3.1.4 Inference of CFGs from queries

The following methods have been proposed to learn a CFG using an oracle to which certain types of queries can be addressed:

1) The *recursive structure discovery* method by Solomonoff [Solo:64].
2) The L^{cf} *algorithm* by Angluin [Angl:87].
3) The method based on *structural membership and equivalence queries* by Sakakibara [Saka:88].

Solomonoff [Solo:64] described a heuristic method by which to discover the recursive structure of a CFL L from a positive sample S^+, that uses an informant who answers membership queries. The strategy consists of two main steps that must be iterated:

i) Select a string $s \in S^+$, delete some substring(s) of s, and ask the oracle if the remaining string belongs to the target language;
ii) If the answer is "yes", reinsert the deleted substring(s) with several repetitions and ask the informant whether each of the resulting strings is also in the target language; if all the strings built in this manner are accepted by the oracle, then a recursive construction is formed consisting of some rules of a CFG (e.g. $A \rightarrow aAa$ and $A \rightarrow b$).

The size of the given sample S^+ is a critical factor in discovering the recursive constructions of the target CFL. If S^+ is too small, some of the recursive constructions might not be discovered, whereas if S^+ is too large, the number of substrings to be considered becomes astronomically large.

The L^{cf} algorithm by Angluin [Angl:87] is aimed at identifying a CFG $G = (V_N, V_T, P, S)$ in Chomsky normal form, such that V_N, V_T and S are known a priori, and it uses to this end a *minimally adequate teacher*, which can answer two types of questions. The first type is a membership query, MEMBER(x, A), where $x \in V_T^*$, $A \in V_N$, in which the teacher determines whether the string x can be derived from A using the rules of G and he answers "yes" or "not". The other type of question is an equivalence query, EQUIV(H), in which a CFG H is conjectured as solution and the teacher determines whether H is equivalent to G (i.e. they generate the same set of terminal strings from the start symbol S), answering "yes" if they are equivalent and providing a counterexample t if not. A counterexample will be a string generated by G but not H or viceversa.

The L^{cf} algorithm starts by enumerating explicitly all the possible productions of G and placing them in the hypothesized set of productions P. This can be effectively done in time bounded by a polynomial in $|V_N|$ and $|V_T|$, because of the Chomsky normal form restriction and the a priori knowledge of V_N and V_T. Then, L^{cf} asks an EQUIV(H) query for the current grammar $H = (V_N, V_T, P, S)$. If H is equivalent to G, then L^{cf} halts with output H. Otherwise, it diagnoses the counterexample t returned, which results in one production being removed from P, and it asks again for the resulting grammar. These steps are repeated until a CFG H (in Chomsky normal form) equivalent to G is obtained. The current set of productions P will always contain the productions of G as a subset, so the current grammar H will always generate a superset of $L(G)$. Hence, the only type of counterexample is one that is generated by H but not by G. To diagnose the counterexample t, L^{cf} finds a derivation tree of t from S using the productions in P, and it asks some membership queries concerning the parse of t until a single production is found which necessarily cannot be in G [Angl:87].

It is easy to show that the running time of L^{cf} is bounded by a polynomial in V_N, V_T, and the maximum length of any counterexample provided during the run. However, since the question EQUIV(H) is not in general decidable for CFGs, there is no computable implementation of the minimally adequate teacher; moreover, the shortest counterexample will not in general be bounded by a polynomial in $|V_N|$ and $|V_T|$. Although the former problem may be overcome by replacing the equivalence queries by a stochastic presentation of examples by the teacher (similar to that explained for the algorithm L^*, see Chapter 2), the very strong requirements of the L^{cf} algorithm make it quite unrealistic to cope with practical situations.

Finally, I will mention a work by Sakakibara [Saka:88], where a polynomial-time algorithm is proposed to learn the full class of CFGs from positive structural examples whenever a teacher is available to answer both structural membership and structural equivalence queries. The former type of question demands the teacher to determine whether a certain skeleton s belongs to $K(D(G))$, the set of unlabeled derivation trees of the target grammar G. The second type of question demands the teacher to determine whether a given grammar G' is structurally equivalent to G (i.e. if $K(D(G')) = K(D(G))$) and to return a skeleton counterexample if not. The algorithm always end by obtaining a CFG G' that is structurally equivalent to the target G.

3.2 Inference of controlled grammars

Recently, Takada demonstrated that there exists a hierarchy of language families, properly contained in the class of context-sensitive languages (CSLs), in which the learning problem for each family is reduced to the learning problem for regular languages [Taka:94]. This paper extended his previous work on the learning of even-linear languages (ELLs) through the inference of regular control sets [Taka:88], which has been commented in Section 3.1.2.

Let $G = (V_N, \Sigma, P, S)$ be a grammar over Σ. Let us assume that each production in P is labeled by its own label symbol and therefore uniquely referred with its label. We write $x \overset{p}{\Rightarrow}_G y$ to mean that y is derived from x using the production $p \in P$, where $x, y \in (V_N \cup \Sigma)^*$, and $x \overset{\alpha}{\Rightarrow}_G y$ to denote that y is derived from x by applying a sequence of productions $\alpha = p_1 ... p_j \in P^*$. In this case, $x \overset{\alpha}{\Rightarrow}_G y$ is called a *derivation from x to y with the associate word α* in G. In addition to the *language generated by G*, defined as $L(G) = \{\omega \in \Sigma^* \mid \exists \alpha \in P^* : \; S \overset{\alpha}{\Rightarrow}_G \omega \}$, we can define the *associate language of G* as the set $A(G) = \{\alpha \in P^* \mid \exists \omega \in \Sigma^* : \; S \overset{\alpha}{\Rightarrow}_G \omega \}$. Finally, a *control set* on G is simply a subset C of P^*, and the *language generated by G with the control set C* is defined as $L(G, C) = \{\omega \in \Sigma^* \mid \exists \alpha \in C : \; S \overset{\alpha}{\Rightarrow}_G \omega \}$.

The definition of a *universal* even-linear grammar (ELG) $U = (\{S\}, \Sigma, \Psi, S)$ over an alphabet Σ has been given in Section 3.1.1. Recall that, for any alphabet Σ, $L(U) = \Sigma^*$, U is unambiguous and unique up to renaming of the start symbol S, and U has exactly $|\Sigma|^2 + |\Sigma| + 1$ productions. Let $\mathbf{U}$ denote the collection of all universal ELGs and $\mathcal{R}$ denote the class of regular languages. Takada defined inductively the following language families [Taka:94]:

$$\begin{aligned} \mathcal{L}_0 &= \mathcal{R} \\ \mathcal{L}_i &= \{L(U, C) \mid U \in \mathbf{U} \wedge C \in \mathcal{L}_{i-1}\} \quad \text{for any integer } i \geq 1. \end{aligned}$$

Given an alphabet Σ and an integer $i \geq 1$, a unique sequence of universal ELGs $U_1, U_2, ..., U_i$ is obtained such that $U_j = (\{S\}, \Psi_{j-1}, \Psi_j, S)$ for $1 \leq j \leq i$, where $\Psi_0 = \Sigma$. Given a regular language R over Ψ_i, the universal ELGs $U_1, U_2, ..., U_i$ specify exactly one language in $\mathcal{L}_i$, namely $L(U_1, L(U_2, ..., L(U_i, R)...))$. Takada proved the following theorem that establishes a hierarchy among the preceding language families [Taka:94]:

Theorem 3.1. $\mathcal{R} = \mathcal{L}_0 \subset \mathcal{L}_1 \subset \mathcal{L}_2 \subset \cdots \subset \mathcal{CS}$, where $\mathcal{L}_1$ is the class of ELLs and $\mathcal{CS}$ denotes the class of CSLs.

In order to demonstrate that for each integer $i \geq 1$, $\mathcal{L}_i$ is contained in the class of CSLs, Takada presented a constructive algorithm [Taka:94] which, given a universal ELG $U = (\{S\}, \Sigma, \Psi, S)$ and a CSG $G = (V_N, \Psi, P, S_G)$ that generates a language $L(G) = C \in \mathcal{L}_i$, it builds another CSG $G' = (V_N \cup \Psi \cup \{S, S_G, S_{G'}\}, \Sigma, P', S_{G'})$ such that $L(G') = L(U, C) \in \mathcal{L}_{i+1}$; since $\mathcal{L}_1$ is the class of ELLs, and hence a CSG (an ELG indeed) can be defined for each language in $\mathcal{L}_1$, the base of the induction is established. In order to demonstrate that the class of CSLs properly contains all the language families in the hierarchy, a counterexample, the CSL $L_p = \{a^p \mid p \text{ is a prime}\}$, was given that did not meet the "pumping lemma" for every family $\mathcal{L}_i$ [Taka:94], thus showing $L_p \notin \mathcal{L}_i \ \ \forall i \geq 0$.

For any integer $i \geq 1$ and any alphabet Σ, let L be a language over Σ in the family $\mathcal{L}_i$. Then, a language C is called a *canonical control set* for L if and only if

i) $L = L(U_1, L(U_2, ..., L(U_i, C)...))$ for universal ELGs $U_1, U_2, ..., U_i$, and
ii) for any $\alpha \in C$, $\exists \omega \in \Sigma^*$, $\{\omega\} = L(U_1, L(U_2, ..., L(U_i, \{\alpha\})...))$.

Takada demonstrated that for each integer $i \geq 1$ and any language $L \in \mathcal{L}_i$, there exists a unique canonical control set C for L, and C is a regular language. This permits to reduce the problem of learning each family $\mathcal{L}_i$ $(i \geq 1)$ to the problem of learning $\mathcal{R}$, since to identify an unknown language $L \in \mathcal{L}_i$, a GI algorithm for $\mathcal{L}_i$ has only to identify the corresponding regular canonical control set for L.

For each family $\mathcal{L}_i$, Takada proposed a learning algorithm that is based on using a front-end processing algorithm, which is specific for each $\mathcal{L}_i$, and a RGI algorithm to learn the regular canonical control set. The front-end processing algorithm has to convert the input strings over Σ into associate words, which are supplied to the RGI algorithm, by iteratively parsing in the universal ELGs $U_1, U_2, ..., U_i$. If the RGI algorithm outputs associate words (e.g. when equivalence queries are answered by means of counterexamples), then the front-end processing algorithm must also convert the associate words to strings over Σ by iteratively generating in the universal ELGs $U_i, U_{i-1}, ..., U_1$. The representation of the inferred language $L \in \mathcal{L}_i$ consists of the

sequence of universal ELGs $U_1, U_2, ..., U_i$ plus the regular language C inferred by the RGI algorithm.

If the RGI algorithm that is used identifies in the limit the class of regular languages $\mathcal{R}$ (e.g. [Angl:87,OnGa:92b]), then Takada's learning algorithm for $\mathcal{L}_i$ will also identify in the limit the family $\mathcal{L}_i$. It must be recalled that, to this end, both positive and negative strings [OnGa:92b] or an informant (answering membership and equivalence queries) [Angl:87] must be available. Given a polynomial-time RGI algorithm such as Angluin's [Angl:87] or Oncina-Garcia's [OnGa:92b], the whole learning procedure for $\mathcal{L}_i$ can be run in polynomial time, since both parsing and generating in universal ELGs have a polynomial time complexity.

An open problem is learning with a variable i, i.e. to find a suitable i such that the target language L is in the class $\mathcal{L}_i$. A possible strategy is to start with some arbitrary i and to increase or decrease i only after determining that the target L is not in $\mathcal{L}_i$. However, it may be quite difficult to decide whether the unknown L is in $\mathcal{L}_i$ or not. On the other hand, the gap in expressive power between each of the language families $\mathcal{L}_i$ and the class of CSLs is not very clear, although it seems to be rather large, according to the "pumping lemma" given by Takada for each family $\mathcal{L}_i$ [Taka:94]. Furthermore, the author does not present any real-world or, at least, interesting pattern which can be described by a language in a class $\mathcal{L}_i$ with $i \geq 2$. This would have been helpful to assess the applicability of his GI approach to learn language models of patterns for syntactic pattern recognition tasks.

3.3 Inference of transition networks

Transition networks were introduced by Woods as models for natural language analysis [Woods:70]. The *basic transition network* (BTN), which was called *recursive transition network* in Woods' paper, is a generalized pushdown automaton and it is equivalent to a CFG. The *augmented transition network* (ATN) is an extension of the BTN in which each arc carries also a test condition that must be satisfied before it can be selected and a sequence of actions that are executed if the arc is selected. ATNs have been shown to be as powerful as Turing machines, and all the acceptors associated with the classes of languages in the Chomsky's hierarchy (including CSLs) can be derived from special cases or restrictions of ATNs [ChFu:75].

Chou and Fu discussed the matter of inferring transition networks from positive samples of strings [ChFu:76]. They proposed an inference method for BTNs which is an extension of the k-tails RGI technique. Since BTNs are equivalent to CFGs, their BTN inference method may have been included also in the group of CFGI methods. They presented afterwards in the same paper a sketch of a semi-automated inference procedure for ATNs, that involves a teacher and a trial-and-error process. Chou and Fu's approaches for the inference of BTNs and ATNs are summarized in the two following subsections, respectively.

3.3.1 Inference of BTNs

A *basic transition network* (BTN) is a directed graph with labeled states and arcs, a distinguished state called the start state, and a distinguished set of states called final states. It differs from an FSA transition diagram in that the label on an arc may be not only a terminal symbol of the input alphabet but also the name of a state. The interpretation of an arc with a state name as its label is that the state at the end of the arc will be saved on a stack and the control will jump (without advancing the input pointer) to the state that is the arc label. When a final state is encountered, the control is transferred to the state which is named on the top of the stack and the stack is "popped". An attempt to pop an empty stack when the last input symbol has just been processed is the criterion for acceptance of an input string. The state names that can appear on arcs represent certain constructions that may be found as substrings (or "phrases") of the input string (or sentence). The effect of a state-labeled arc is that the transition that it represents may take place if a construction of the indicated type is found as a substring at the appropriate point in the input string.

The BTN model is fundamentally a non-deterministic mechanism, and any parsing

algorithm for BTN grammars must be capable of following all the analysis paths for any given string. A BTN can be viewed as a finite set of FSAs controlled by a stack, and this is essentially a pushdown automaton [HoUl:79] whose stack vocabulary is a subset of its state set. On the other hand, it is quite direct to build a BTN equivalent to a given CFG, and this CFG may be selected as equivalent to a given pushdown automaton. Hence, the recognition power of BTNs and pushdown automata is the same, namely, that of accepting the class of CFLs.

Formally, a BTN N can be defined as a six-tuple $N = (\Sigma, Q, Q_0, Q_F, q_0, A)$, where Σ is a finite set of input symbols, Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states of the FSAs contained in the network, $Q_F \subseteq Q$ is the set of final states of the FSAs, $q_0 \in Q_0$ is the initial state of the BTN, and A is a finite set of arcs. For each state, there can be several outgoing arcs, each of which must belong to one of the following cathegories:

1. CAT arc: (CAT c). A transition is made from the present state to the state at the end of the arc consuming an input symbol $c \in \Sigma$.
2. PUSH arc: (PUSH q). The destination state of the arc is saved on the stack and the next state is $q \in Q_0$, that is the label of the arc.
3. POP arc: (POP). The next state is the one shown on the top of the stack, and the stack is popped one element up. Note that POP arcs can only depart from final states.

The language accepted by a BTN N is denoted $L(N)$.

The method proposed by Chou and Fu for the inference of BTNs [ChFu:76] concerns the revealing of the self-embedding structures from a positive sample set using the idea of formal derivatives. As in the Solomonoff's CFGI method [Solo:64], the approach is based on the *pumping lemma* for context-free languages:

Theorem 3.2. For any CFL L, there exists integers p and q such that if a string $s \in L$ with $|s| > p$ then s can be decomposed to the form $s = uvwxy$, where $vx \neq \lambda$, $|vwx| \leq q$, and $\forall i \geq 0 : \; uv^iwx^iy \in L$.

Let S be a set of strings over Σ and $v, x \in \Sigma^*$,

$D_v(S,k) = \{w \in \Sigma^* \mid vw \in S \;\wedge\; |w| \leq k\}$ is the set of right k-derivatives of S by v,
$E_x(S,k) = \{w \in \Sigma^* \mid wx \in S \;\wedge\; |w| \leq k\}$ is the set of left k-derivatives of S by x, and
${}_vg_x(S,k) = \{w \in \Sigma^* \mid vwx \in S \;\wedge\; |w| \leq k\} = D_v(E_x(S,\infty),k)$ is the set of interior k-derivatives of S by the string pair v, x.

By Theorem 3.2, if a structurally complete (large enough) positive sample S^+ of a CFL L is given, then for some k, $k = |w|$, ${}_{uv^i}g_{x^iy}(S^+, k) \supseteq \{w\}$ for some values of $i \geq 0$, where $uv^iwx^iy \in L$ for all $i \geq 0$, and $vx \neq \lambda$. It can be seen that the substrings in $\{v^iwx^i \mid i \geq 0\}$ are accepted by the recursive subnetwork (BTN) shown in Fig.3.1 (a); this subnetwork can be constructed as the *canonical derivative acceptor* (see Chap.2, Sect.2.2), $CDA(S_A^+)$, of a sample set $S_A^+ = \{w, vAx\}$, where A is the name of the subnetwork. If such a recursive structure is found after the analysis of the interior k-derivatives of S^+, then the sample set S^+ is rewritten by replacing the substrings in $\{v^iwx^i \mid i \geq 0\}$ by the name of the subnetwork A, and the search of another self-embedding is performed on the new sample set, treating the nonterminal A as a terminal symbol. The search procedure is repeated until no more recursive subnetwork can be found. Finally, the main subnetwork, named S, where S is the start state of the inferred BTN, is built as the canonical derivative acceptor of the last rewritten sample set. The inferred BTN is the set of all the subnetworks constructed during the process.

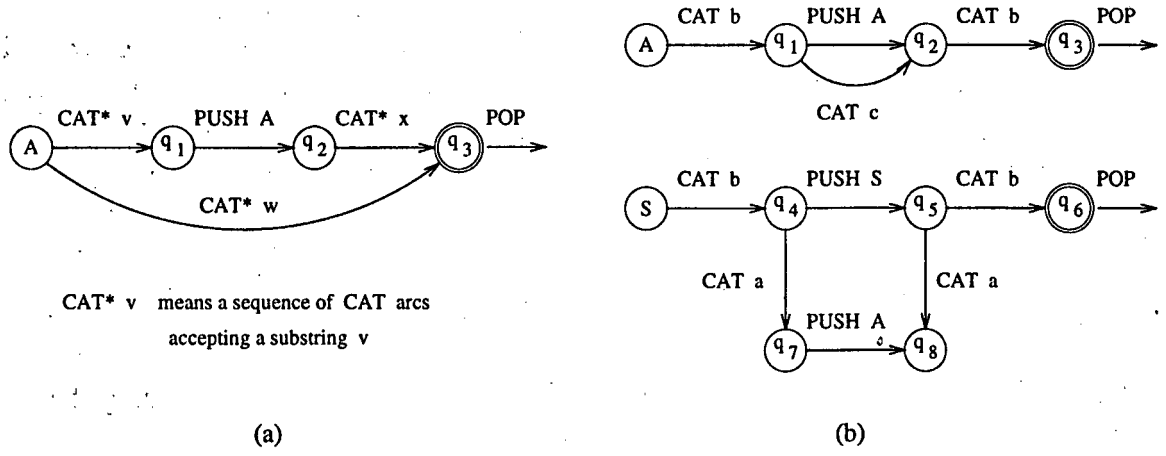

Fig. 3.1 (a) *The BTN accepting* $\{v^iwx^i \mid i \geq 0,\ v, w, x \in \Sigma^*\}$.
(b) *The BTN accepting* $L = \{b^lab^kcb^kab^l \mid k, l \geq 1\}$.

In order to search for a self-embedding, all the derivative sets ${}_{v_i}g_{x_j}(S^+, n)$ are computed for every prefix v_i and suffix x_j of S^+, where n is an integer such that $\forall s \in S^+ : |s| \leq n$. Then, for $k = 1, 2, ..., (k < n)$, the list of equivalence classes $U_{k,l}$ $(l = 1, ..., L_k)$ of the derivatives ${}_vg_x(S^+, k)$ is obtained. A recursive structure is detected when a class $U_{k,l}$ includes a nonempty subset of the set $\{{}_{uv^i}g_{x^iy} \mid i \geq 0\}$ for some $u, v, x, y \in \Sigma^*$, $vx \neq \lambda$. In such a case, let i_1 be the smallest integer $i \geq 0$ such that ${}_{uv^i}g_{x^iy}$ is in $U_{k,l}$, and let the value of the class $U_{k,l}$, denoted $\overline{U}_{k,l}$, be a set of strings $\overline{U}_{k,l} = {}_{uv^{i_1}}g_{x^{i_1}y}(S^+, k) = \{w_m \mid m \geq 1,\ |w_m| \leq k\}$. Then, a sample $S_A^+ = \{v^{i_1}\,\overline{U}_{k,l}\,x^{i_1}\} \cup \{vAx\}$ can be written that allows the construction of the recursive subnetwork A for the self-embedding just detected.

Chou and Fu [ChFu:76] presented an example of application of their method using a sample of the CFL $L = \{b^l ab^k cb^k ab^l \mid k, l \geq 1\}$ that led to the inference of the BTN shown in Fig.3.1 (b), which accepts the target language L. Although their BTN inference heuristic procedure is not too difficult to implement, it should be noted that a sample set S^+ large enough to comprise the self-embedding structures of the target CFL is required, whereas for a large sample set, the computational cost of the method may be considerable both in space and time resources.

3.3.2 Inference of ATNs

An *augmented transition network* (ATN) differs from a BTN in that each arc of the network may include an arbitrary condition, which must be satisfied in order for the arc to be followed, and a set of structure building actions to be executed if the arc is followed [Woods:70]. The ATN builds up a partial structural description of the string as it proceeds from state to state through the network. The pieces of this partial description are held in *registers* which can contain any rooted tree or list of rooted trees, and which are automatically pushed down when a recursive application of the transition network is called for and restored when the lower level recursive computation is completed. The structure-building actions on the arcs specify changes in the contents of these registers in terms of their previous contents, the contents of other registers, the current input symbol, and/or the result of lower level computations. In addition, the registers may also be used to hold flags or other indicators to be interrogated by conditions on the arcs. Each final state of the ATN has associated with it a test condition, which must be satisfied in order for that state to cause a "pop", and a function which computes the value to be returned by the (sub)network storing it in a special register.

Formally, an ATN N can be defined as a six-tuple $N = (\Sigma, Q, Q_0, Q_F, q_0, A)$, where Σ, Q, Q_0, Q_F, q_0 are as in a BTN, and A is a finite set of arcs, each departing from a specified state and belonging to one of the following four cathegories [Woods:70]:

1. CAT arc: (CAT *c test action* term_act*). A CAT arc is followed if the current input symbol is $c \in \Sigma$ and the test is satisfied.
2. PUSH arc: (PUSH *q test action* term_act*). If the test is satisfied, a PUSH arc saves the destination state q' on the stack and transfers control to state $q \in Q_0$.
3. TST arc: (TST *test action* term_act*). A TST arc permits an arbitrary test to determine whether the arc is followed.
4. POP arc: (POP *test form*). A POP arc is a dummy arc which indicates under what conditions the state is to be considered a final state, and the form to be returned as the value of the computation if the POP alternative is chosen. In

such a case, it transfers control to the state on the top of the stack, and the stack is popped.

In the three former types of arcs, the *actions* on the arc are structure-building actions, which consist of setting a specified register to the value of an indicated form, whereas the *terminal action* specifies the destination state q' of the arc. The two possible terminal actions, TO q' and JUMP q', indicate whether the current input symbol is to be consumed or not, respectively. The *forms* as well as the *tests* of the ATN may be arbitrary functions of the register contents, represented in some functional specification language such as LISP. This great computational capability makes ATNs as powerful as Turing machines, but it difficults enormously the eventual development of learning techniques for the inference of ATNs.

Chou and Fu presented an outline of a semi-automated approach to the inference of ATNs accepting context-sensitive languages (CSLs) [ChFu:76]. The type of ATN they dealt with was rather more restricted than the general ATN model defined by Woods which has been recalled in the preceding paragraphs. More precisely, their model of ATN allowed the following augmentations with respect to the features of BTNs:

- An optional action HOLD may be placed in a CAT arc in order to save the consumed input symbol in a hold list (a register). This serves for future tests of context relationship.
- A particular type of TST arc (VIR c), called VIR arc, may be included that makes a transition to its destination state when the symbol $c \in \Sigma$ displayed in the arc is in the hold list.
- A JUMP arc is also allowed, that transfers control to its destination state without advancing the input pointer if a certain condition is satisfied. Thus, a JUMP arc corresponds to a TST arc of the form (TST *test* JUMP q') with no associated action.

It is well-known that a CSG can be seen as a CFG (base) plus a set of transformational rules [Chom:65, ChFu:75]. This is, a CSL is obtained by applying a sequence of transformations to the strings in the CFL generated by a CFG. In terms of transition networks, the CFL accepted by a given BTN can be transformed in a CSL by adding to the network some proper "augmented arcs" and/or by including certain test and actions on the existing arcs. On the other hand, if the reverse transformations are known, then by applying them to a given positive sample of a CSL, a CFL sample set can be obtained. The ATN inference system proposed by Chou and Fu [ChFu:76] (see Fig.3.2) follows this idea by combining their BTN inference method with a teacher who defines a set of hypothesized transformation rules, which determine both the prior sample rewriting (reverse transformations) and the augmented arcs that must be added to the inferred BTN (forward transformations). Chou and Fu claimed that this

inference system may be operated by the teacher in the fashion of trial and error. They also presented a simple inference example, in which, assuming a reverse transformation "$bbc \rightarrow bcb$", a positive sample of the CSL $L = \{a^n b^n c^n \mid n \geq 1\}$ led to an ATN accepting L, that is shown in Fig.3.3.

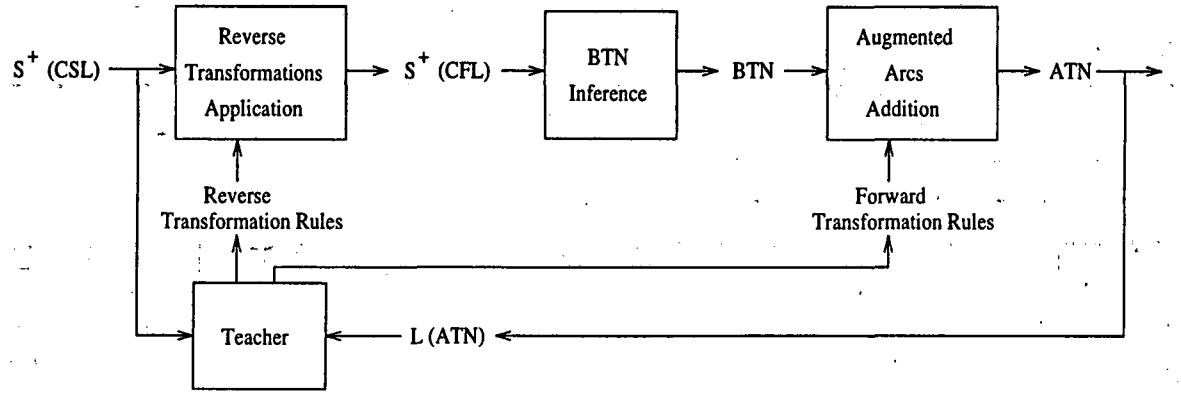

Fig. 3.2 *Diagram of Chou and Fu's approach to the inference of ATNs.*

It is apparent, however, that many important aspects of the above ATN inference method are extremely unclear, such as the availability of the reverse transformations, the process followed to add the proper augmented arcs to the inferred BTN, the criteria to restart or to stop the inference, and the modifications carried out by the teacher on the transformation rules when several inference cycles are run. In general, the problem of inferring ATNs seems to be very difficult, since it is unclear how to infer automatically the test conditions and register-setting actions that permit to represent the context-sensitive structures. Unfortunately, but not surprisingly, the line of research started by Chou and Fu in the seventies on the subject of inference of transition networks has not been continued so far, as no other related work has been reported since then.

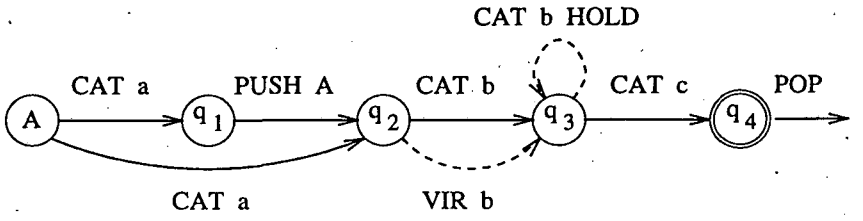

Fig. 3.3 *An ATN accepting $L = \{a^n b^n c^n \mid n \geq 1\}$ (the dashed arcs denote the "augmented" arcs not included in the inferred BTN).*

3.4 Inference of pattern languages

The class of *pattern languages* was first introduced by Angluin [Angl:80b]. Roughly speaking, a *pattern* p is a finite string of constant and variable symbols, and the *pattern language* $L(p)$ represented by p is the set of strings of constant symbols that can be obtained from p by substituting a specific constant string for each variable in p. For example, $0x01x$ and $10xy1x0y$ are patterns over constants 0 and 1 with one and two variables respectively, and $L(0x01x)$ = {00010, 01011, 0000100, 0010101, 0100110, 0110111, 000001000, ... }. A more formal definition follows.

Given an alphabet Σ of symbols called *constants* and a countable set X of symbols called *variables* (disjoint from Σ), a *pattern* over (Σ, X) is a finite string in $(\Sigma \cup X)^+$. Let p be a pattern over (Σ, X), the *number of variables* in p is the number k of distinct elements in X, i.e. $X = \{x_1, ..., x_k\}$. Let PN denote the set of all patterns and, for $k \geq 0$, let PN_k denote the set of all patterns of k variables. A *substitution* over (Σ, X) is a mapping $f : X \rightarrow \Sigma^+$. A substitution f can be extended to strings in $(\Sigma \cup X)^+$ in a natural way: i) $f(a) = a$, if $a \in \Sigma$; ii) $f(uv) = f(u)f(v)$, for all $u, v \in (\Sigma \cup X)^+$. Given a pattern $p \in PN_k$, $p[t_1/x_1, ..., t_k/x_k]$, or $p[t_i/x_i]_{i=1}^k$, denotes the string in Σ^+ obtained from p by the substitution f that maps each x_i to t_i, $i = 1, ..., k$. For example, if $p = x_1x_2x_2$, then $p[00/x_1, 10/x_2] = 001010$.

The *pattern language* defined by a pattern $p \in PN_k$ is the set $\{ p[t_i/x_i]_{i=1}^k \mid t_i \in \Sigma^+, i = 1, ..., k\}$. It is assumed that equivalent patterns (those that can be obtained from each other by renaming the variables) have only one canonical representation. Namely, the variables in a k-variable pattern p are always $x_1, ..., x_k$ and if $1 \leq m < n \leq k$ then $j_m < j_n$, where j_m, j_n refer to the positions of the first occurrence of x_m, x_n in p, respectively. Under this canonical representation, it can be said that each pattern p with k variables represents a different pattern language $L(p)$, i.e. let p and q be patterns in PN_k for some $k \geq 0$, then $p = q$ iff $L(p) = L(q)$.

Although pattern languages are not comparable to the Chomsky's hierarchy of languages, it is clear that patterns provide a limited mechanism to describe some context influences, namely, the repetition of variable substrings along the strings of a language. Consider, for example, the simple pattern $p = xx$ over $(\{0,1\}, \{x\}), p \in PN_1$. The language $L(p) = \{xx \mid x \in (0+1)^+\}$ is a CSL that can be generated by a CSG G_p containing the 28 productions shown in Fig.3.4. However, the expressive power of patterns is insufficient to describe even simple CSLs such as $\{a^nb^nc^n \mid n \geq 1\}$. Furthermore, most of the CFLs and regular languages cannot be represented by patterns either, e.g. the CFL $\{0^m1^{m+n}0^n \mid m, n \geq 1\}$ and the even-parity regular language $(0^*10^*1)^*0^*$.

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

Fig. 3.4 *A CSG G_p that generates the pattern language $L(p) = \{xx \mid x \in (0+1)^+\}$.*

The inductive inference of pattern languages has been studied by several researchers. The reported methods can be classified in two groups, depending on whether the input data is restricted to just positive strings or some type of queries are used.

3.4.1 Inference of pattern languages from positive examples

If the number k of variables in a target pattern p is known, then p can be identified in the limit from a positive presentation of constant strings by an algorithm that, for each sample S^+, outputs a k-variable pattern p' such that $L(p')$ is a smallest pattern language containing S^+ that is defined by a k-variable pattern [Ang:80b].

Let $k > 0$. For a given set S^+ of strings in Σ^*, a *minimal k-variable pattern* for S^+ is a pattern $p \in PN_k$ such that $S^+ \subseteq L(p)$ and for any $q \in PN_k$ such that $S^+ \subseteq L(q)$, $L(q)$ is not a proper subset of $L(p)$. The *k-variable pattern finding problem* is to find, for a given set S^+ of strings, a minimal k-variable pattern for S^+. It can also be regarded as the search of one of the *longest* k-variable patterns for S^+. Some works have dealt with the k-variable pattern finding problem, and some algorithms that solve it for particular cases of k have been reported:

1) The *one-variable pattern finding* algorithm by Angluin [Angl:80b].
2) The *two-variable pattern finding* algorithm by Ko and Hua [KoHu:87].
3) The *k-variable pattern finding* algorithm by Jantke [Jant:84].

Angluin presented an algorithm that solves the one-variable pattern finding problem in polynomial time, using *pattern automata* to represent concisely the set of all the patterns that could have generated a given string or set of strings [Angl:80b]. Next, pattern automata are defined formally, and later, Angluin's algorithm is reviewed.

Let s be a string and w a nonempty substring of s. Let $patt1(s;w)$ denote the set of one-variable patterns that generate s by substituting w for each occurrence of the variable, i.e. $patt1(s;w) = \{p \in PN_1 \mid s = p[w/x]\}$. In order to recognize the set $patt1(s;w)$, a corresponding pattern automaton $A(s;w) = (Q, \Sigma \cup X, \delta, q_0, F)$ is built as follows: The states in Q are the ordered pairs $(i,j) \subset \mathcal{N} \times \mathcal{N}$ such that $i+j|w| \leq |s|$; the initial state is $q_0 = (0,0)$; the final states in F are all states (i,j) such that $j \geq 1$ and $i + j|w| = |s|$; and the transition function δ is defined by

$$b \in \Sigma \Rightarrow \delta((i,j),b) = \begin{cases} (i+1,j) & \text{if } s[1+i+j|w|] = b, \\ \text{undefined} & \text{otherwise;} \end{cases}$$

$$\delta((i,j),x) = \begin{cases} (i,j+1) & \text{if } w \text{ occurs in } s \text{ beginning at position } 1+i+j|w|, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Hence, the states of the automaton $A(s;w)$ are numbered in such a way that the state (i,j) will be reached after i constants and j x's are processed. For example, for $s = 01110$ and $w = 11$, the automaton $A(s;w)$ is shown in Fig.3.5 and it recognizes the set $patt1(s;w) = \{0x10,\ 01x0\}$.

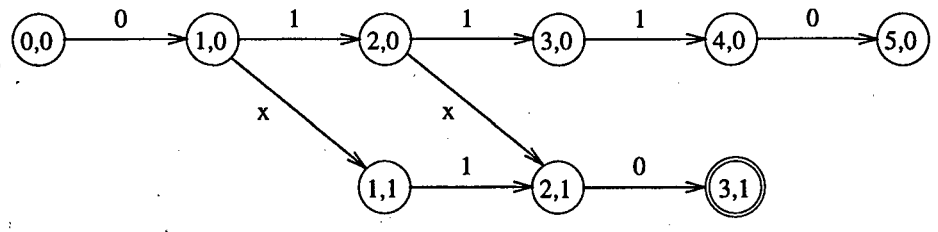

Fig. 3.5 *An example of one-variable pattern automaton.*

Let $A_1 = (Q_1, \Sigma \cup X, \delta_1, q_0, F_1)$ and $A_2 = (Q_2, \Sigma \cup X, \delta_2, q_0, F_2)$ be two finite automata with state sets $Q_1, Q_2 \subset \mathcal{N} \times \mathcal{N}$ and $q_0 = (0,0)$. Then, a relation $A_1 \subseteq A_2$ is defined, that is true iff $Q_1 \subseteq Q_2$, $F_1 \subseteq F_2$ and whenever δ_1 is defined, δ_2 is also defined and agrees with δ_1. A finite automaton A is called a *one-variable pattern automaton* iff $A \subseteq A(s;w)$ for some string s and substring w in Σ^+. The set of patterns accepted by A is denoted $L(A)$.

Let $A_1 = (Q_1, \Sigma \cup X, \delta_1, q_0, F_1)$ and $A_2 = (Q_2, \Sigma \cup X, \delta_2, q_0, F_2)$ be two one-variable pattern automata. Then the *intersection of automata* A_1 and A_2 is the finite automaton $A_1 \cap A_2 = (Q_1 \cap Q_2, \Sigma \cup X, \delta, q_0, F_1 \cap F_2)$, where

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{if } \delta_1(q,a) \text{ and } \delta_2(q,a) \text{ are both defined and equal,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Angluin proved that $A_1 \cap A_2$ is also a one-variable pattern automaton, and $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$ [Angl:80b]. The intersection $\bigcap_i A_i$ of t pattern automata A_i, $1 \leq i \leq t$ can be computed in time $O(t \cdot n)$, where n is the number of states of the largest A_i.

Now, for each one-variable pattern p, the three following attributes are defined:

(i) $\text{count}(\Sigma, p)$ = the number of constant symbols in p,
(ii) $\text{count}(x, p)$ = the number of occurrences of x in p,
(iii) $\text{first}(x, p)$ = the position of the first occurrence of x in p.

Let $PN(i,j,k)$ be the set of all patterns $p \in PN_1$ such that $\text{count}(\Sigma, p) = i$, $\text{count}(x, p) = j$, and $\text{first}(x, p) = k$. It is obvious that PN_1 is partitioned into pairwise disjoint sets $PN(i,j,k)$.

A triple (i,j,k) is feasible for a string s if there is potentially a pattern $p \in PN(i,j,k)$ such that $s \in L(p)$, more precisely, if $0 \leq i \leq |s|$, $1 \leq j \leq |s|$, $1 \leq k \leq (i+1)$, and $(|s| - i) \bmod j = 0$. If (i,j,k) is feasible for s, let w be the substring of s beginning at position k with length $(|s| - i)/j$; then, it can be shown that for any pattern $p \in PN(i,j,k)$ such that $s \in L(p)$, $s = p[w/x]$. Thus, for each string s and each triple (i,j,k) that is feasible for s, a pattern automaton $A(s; w)$ can be constructed, where w is uniquely determined.

Let $S^+ = \{s_1, ..., s_t\}$ be a set of strings in Σ^+ given to the inductive algorithm. A triple (i,j,k) is *feasible for a set* S^+ if it is feasible for all $s_r \in S^+$, $1 \leq r \leq t$. Let $F(S^+)$ be the set of all feasible triples for S^+. Then, each triple (i,j,k) in $F(S^+)$ defines t automata $A_r(i,j,k)$, $1 \leq r \leq t$, where each $A_r(i,j,k)$ is the pattern automaton $A(s_r; w_r)$ defined by s_r and (i,j,k), with some useless states and transitions removed. $A_r(i,j,k)$ recognizes all patterns $p \in PN(i,j,k)$ such that $s_r \in L(p)$. The one-variable pattern finding method by Angluin [Angl:80b], defined in Algorithm 3.1, is based on the following property:

$$\{p \in PN_1 \mid S^+ \subseteq L(p)\} = \bigcup_{(i,j,k) \in F(S^+)} \bigcap_{r=1}^{t} L(A_r(i,j,k)) = \bigcup_{(i,j,k) \in F(S^+)} L\left(\bigcap_{r=1}^{t} A_r(i,j,k)\right).$$

ALGORITHM 3.1: *Angluin's one-variable pattern finding algorithm*
Input: $S^+ = \{s_1, ..., s_t\}$.
Output: a minimal one-variable pattern p for S^+.
begin
for each (i,j,k) **in** $F(S^+)$ **do**
    for $r := 1$ **to** t **do**
        construct automaton $A_r(i,j,k)$
    end_for
    $A(i,j,k) := \bigcap_{r=1}^{t} A_r(i,j,k)$
end_for
sort $F(S^+)$ in descending order according to the value of $i+j$
for each (i,j,k) **in** sorted $F(S^+)$ **do**
    if $L(A(i,j,k)) \neq \emptyset$ **then**
        return any $p \in L(A(i,j,k))$ and **exit**
    end_if
end_for
end_algorithm

The time complexity of the above algorithm is determined by two factors: the number of feasible triples for S^+, i.e. $|F(S^+)|$, and the amount of time to construct $A(i,j,k)$. Let n be the length of the longest string in S^+. It can be shown that the size of $F(S^+)$ is $O(n^2 \log_2 n)$, and that the automaton $A(i,j,k)$ can be constructed in time $O(tn^2)$, since each of the automata $A_r(i,j,k)$ contains a number of states of $O(n^2)$. Therefore, Angluin's algorithm runs in time $O(tn^4 \log_2 n)$. Angluin pointed out that although her algorithm may be generalized to the k-variable cases for $k > 1$ in a straightforward manner, the generalized algorithm does not seem to run in polynomial time in the cases $k > 1$ [Angl:80b].

Ko and Hua further investigated the generalization of Angluin's algorithm to the *two-variable* pattern finding problem [KoHu:87]. To this end, they defined the *two-variable pattern automata* in a similar way and showed that, for two-variable pattern automata A_1, A_2, their intersection $A_1 \cap A_2$ is also a two-variable pattern automaton such that $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$. They also defined the *feasible 6-tuples* (i,j,k,l,m,n) as an extension of the feasible triples in the one-variable case, so that the set of all two-variable patterns PN_2 may be partitioned in sets $PN(i,j,k,l,m,n)$ and the set $F_2(S^+)$ of feasible 6-tuples for a given set of strings S^+ may be found. The number of feasible 6-tuples in $F_2(S^+)$ is $O(n^6)$, each representing a group of patterns.

Unfortunately, for some 6-tuples (i,j,k,l,m,n) in $F_2(S^+)$, the set of patterns in $PN(i,j,k,l,m,n)$ that generate a string $s \in S^+$ may require more than one pattern

automaton to represent them. Thus, the problem of finding whether there exists a pattern p in $PN(i,j,k,l,m,n)$ generating the sample S^+ becomes the Intersection of Pattern Automata (IPA) problem: given a set of two-variable pattern automata $\{A_{i,j} \mid 1 \leq i \leq t,\ 1 \leq j \leq u_i\}$, find an automaton A that accepts the patterns in $\bigcap_{i=1}^{t} \{p \mid p \in L(A_{i,j}),\ \text{for some } j,\ 1 \leq j \leq u_i\}$. Ko and Hua showed that the IPA problem is NP-hard, and hence provided an explanation why the straightforward generalization of Angluin's algorithm to the k-variable case for $k \geq 2$ does not run in polynomial time [KoHu:87].

On the other hand, Jantke proposed, for each $k > 1$, a generalized algorithm for the k-variable pattern finding problem that was slightly different from the generalization of Ko and Hua, and he claimed that his algorithm runs in polynomial time [Jant:84]. However, the proof of the polynomial time bound of a critical step of his algorithm was omitted. Although the proof of the NP-hardness of the IPA problem cannot be directly transformed into a proof of the NP-hardness of the k-variable pattern finding problem for $k \geq 2$, it is generally suspected that in fact the latter may be NP-hard.

3.4.2 Inference of pattern languages using queries

The following methods have been reported for the identification of pattern languages using some type of queries:

1) The *identification methods from examples and membership queries* by Marron and Ko [MaKo:87].
2) The *superset queries method* by Angluin [Angl:88].
3) The *prefix queries method* by Siromoney *et al.* [SiSM:92].

Marron and Ko addressed the problem of identifying pattern languages from both positive data and membership queries [MaKo:87]. In this setting it is assumed that there exists a predetermined unique *target pattern* p and the goal is to identify p precisely. The problem can be viewed as a two-person game, in which the first player A (the learner) attempts to identify a k-variable pattern p that was chosen by (and is known only to) the second player B (the oracle). The number k of variables in the target pattern is assumed to be known. Initially, B provides A with a finite set S^+ of strings in $L(p)$; then B answers questions of whether $s \in L(p)$ for each string s queried by A.

Marron and Ko showed that it is easy to identify p by asking $s \in L(p)$ for all constant strings s of length $|s| \leq |p|$ (indeed without the need to know $|p|$) [MaKo:87],

and thus A can always identify the target pattern p by making $(|\Sigma|^{|p|+1}-1)/(|\Sigma|-1)$ queries, i.e. $O(|\Sigma|^{|p|})$. Hence, their interest was in developing more efficient algorithms leading to the identification of p through a polynomial number of queries, but subject to certain assumptions about the initial sample. Thus, a target pattern p is said to be *polynomially inferable* from the initial sample S^+ if the learner A can identify p by making only $\phi(|p|)$ many queries to the oracle B for some fixed polynomial ϕ.

Marron and Ko gave simple sufficient conditions on the initial sample S^+ for the polynomial inferability of patterns together with the corresponding inference algorithms for the cases $k=1$ and $k \geq 2$. From the point of view of a two-person game, a condition on the initial sample is *(polynomially) sufficient* if A has a strategy that can always identify a pattern p with $\phi(|p|)$ queries for some polynomial ϕ, whenever the initial sample S^+ provided by B satisfies this condition. The sufficient condition on S^+ typically involves not only properties of the strings in S^+, but also the inter-relations between the example strings and the target pattern.

For the case of patterns with one variable ($k=1$), the sufficient conditions on S^+ are the following:

(i) The total length of the strings in S^+ is bounded by a polynomial in $|p|$.
(ii) S^+ contains two strings w_0 and w_1 which are obtained from p by substituting the variable x by two *incompatible* strings u_0 and u_1 respectively, i.e. $w_0 = p[u_0/x]$ and $w_1 = p[u_1/x]$, where two strings u_0 and u_1 are *incompatible* if neither is both a prefix and a suffix of the other.

For example, a simple sufficient sample S^+ is created by substituting two strings u and v for x, where $|u| = |v|$ and $u \neq v$. The first condition above is required so that the two examples w_0 and w_1 can always be found in polynomial time. Marron and Ko also demonstrated that the above sufficient conditions are also necessary to identify p by a polynomial number of queries. In particular, there is no polynomial time algorithm that finds patterns using membership queries alone (without any given example string).

Now, two sequences of strings $\langle u_1, ..., u_k \rangle$ and $\langle v_1, ..., v_k \rangle$ are *uniformly incompatible* if either, for all i, $1 \leq i \leq k$, u_i and v_i are not prefixes of each other, or, for all i, $1 \leq i \leq k$, u_i and v_i are not suffixes of each other. Then, for the case of k-variable patterns with $k \geq 2$, the sufficient condition on S^+ turns out to be the following:

(i) S^+ contains a subset of $k+1$ strings $S' = \{p[u_i/x_i]_{i=1}^k\} \cup \{p[v_i/x_i, u_j/x_j]_{j=1, j\neq i}^k \mid 1 \leq i \leq k\}$ where $\langle u_i \rangle_{i=1}^k$ and $\langle v_i \rangle_{i=1}^k$ are two sequences of strings that are *uniformly incompatible*, $\forall i \leq k: \ |u_i|, |v_i| \leq \phi(|p|)$, and $\begin{pmatrix} |S^+| \\ |S'| \end{pmatrix} \leq \phi(|p|)$, for some polynomial $\phi(|p|)$.

Basically, the subset S' contains a basis string $p[u_i/x_i]_{i=1}^{k}$ and k additional strings which differ from the basis string only by a single substitution. Again, the necessity of the incompatibility of the substitution strings was shown.

It must be remarked that the identification algorithms given by Marron and Ko work only if the assumption about the number k of variables in the target pattern is correct (i.e. if k is known in advance). In other words, the discovery of k cannot be included in the general algorithm if the polynomial inferability of p from S^+ is desired. The main reason is the fact that (polynomially) sufficient samples for patterns in PN_k can be viewed also as insufficient samples for an unknown pattern of k' variables with $k' > k$. Hence, there may be cases where the pattern that is output as solution by the identification algorithm will not correspond to the target pattern.

In another paper, Angluin reported a polynomial-time learning algorithm for identifying pattern languages that uses *restricted superset queries* [Angl:88]. Let p be the unknown target pattern. In a *restricted superset query*, the learner asks "$L(p') \supseteq L(p)$?", for some pattern p', and the oracle answers "yes" or "no", without giving any counterexample. The algorithm proposed by Angluin to identify p using this type of queries is as follows.

Firstly, the length of the pattern p is determined. Note that if p is a pattern of length n, then $L(p)$ contains only strings of length n or greater and at least one string of length n. Also, $L(x_1x_2...x_n)$ is precisely the set of all the strings over Σ of length n or greater. Thus, the length n of p can be determined by asking superset queries on the sequence of patterns x_1, x_1x_2, $x_1x_2x_3$, and so on, until $L(x_1x_2...x_{n+1})$ is found to be not a superset of $L(p)$.

Secondly, the positions and values of the constant symbols in p are determined. For each $a \in \Sigma$ and $i = 1, 2, ..., n$, it is asked whether $L(x_1...x_{i-1}ax_{i+1}...x_n) \supseteq L(p)$. If the answer is positive, then the i-th symbol of p is the constant symbol a. Otherwise, if the answer is negative for all $a \in \Sigma$, then the i-th symbol of p is a variable symbol.

Thirdly, for each pair of positions containing variables, it is determined whether the variables are the same or not. For each pair (i, j), $i < j$, of positions of variable symbols in p, it is queried whether $L(p_{i,j}) \supseteq L(p)$, where the pattern $p_{i,j}$ is obtained from $x_1x_2...x_n$ by replacing both x_i and x_j by a new variable x. If the answer is "yes", then positions i and j of p contain the same variable; otherwise, they contain different variables.

Once all the above tests have been completed, the canonical form of p can be directly constructed. The total number of queries used by this method is bounded by $(n+1)+n|\Sigma|+n(n-1)/2$. Hence, the computation time for Angluin's superset queries

method for identifying pattern languages is of $O(|p|^2)$. Angluin also demonstrated that any algorithm that exactly identifies all the patterns of length n using membership, equivalence, and subset queries[2] must make at least $2^n - 1$ queries in the worst case [Angl:88].

The previous works on pattern inference that have been commented [KoHu:87, MaKo:87, Angl:88] point out that, if no restriction is placed on the pattern (such as the number of variables k) or the strings substituted for variables in the examples, learning pattern languages using positive examples and/or queries such as membership and equivalence queries takes exponential time. We have seen that the Angluin's algorithm using superset queries is a method for learning pattern languages without any restriction in time polynomial in the length of the pattern. More recently, Siromoney *et al.* introduced the concept of *prefix query*, and they formulated another simple identification algorithm for unrestricted patterns using prefix queries that also runs in $O(|p|^2)$ [SiSM:92].

Let $p = p_1 p_2 ... p_n$ be the target pattern in canonical form (i.e. the leftmost occurrence of variable x_i in p always appear before the leftmost occurrence of x_{i+1}, for $1 \leq i < k$), where both the length n and the number of variables k are unknown. In a *prefix query*, a pattern p' is proposed by the learner, and the oracle answers "yes" if p' is a prefix of p, and "no" otherwise. The following algorithm leads to the identification of p using prefix queries [SiSM:92].

The algorithm starts by trying to determine whether the first symbol p_1 of p is a constant. Hence, for each $a \in \Sigma$, it is asked whether the string a is a prefix of p. If the answer is negative in all cases, then p_1 is a variable, and since p is in canonical form, $p_1 = x_1$. Suppose that, at some stage i, the algorithm has found that $p_1 ... p_i$ is a prefix of p and j is the largest index of a variable $x_j \in \{p_1, ..., p_i\}$. Then, it is checked whether p_{i+1} is a constant by asking whether $p_1 ... p_i a$ is a prefix of p for each $a \in \Sigma$. If all the answers are negative, then p_{i+1} should be a variable, and therefore, it is queried whether $p_1 ... p_i x_m$ is a prefix of p for $1 \leq m \leq j + 1$. If a negative answer is received for each of these queries, then it is clear that the pattern p has been learnt, so the algorithm halts.

The total number of prefix queries required is bounded by $n(\Sigma + k)$, and thus, the time complexity of the algorithm is $O(n^2)$ (since $k \leq n$).

[2] In a *subset query*, the learner asks "$L(p') \subseteq L(p)$?", for some pattern p', the oracle answers "yes" or "no" and, if the answer is negative, the oracle also supplies a string $s \in L(p') - L(p)$ as a counterexample.

Although Siromoney's algorithm is very simple and efficient and it does not impose any assumption on the target pattern p, the availability of a teacher answering prefix queries about p, besides being too ideal, converts the learning task into a rather trivial one. A similar comment may be applied to the superset queries based method by Angluin, which has been described previously. Hence, the interest of these pattern identification methods is mainly theoretical.

On the other hand, Siromoney *et al.* showed that certain simple picture and contour languages can be considered as interpretations of pattern languages, and thus, pattern learning algorithms can be applied for their inference [SiSM:92]. In particular, they proposed three variations on the definition of pattern languages that result in three types of languages with applications in the representation of pictorial data.

The first variation involves requiring the patterns to contain only variables and restricting the strings u_i used to replace the variables x_i's to be of equal length. Let $\mathbf{L}_1$ be the class of languages described by patterns of this kind. A language $L_1(p)$ can be considered as a family of arrays (pictures) by writing the strings u_i replacing each variable x_i in the pattern as a separate column. The class of languages $\mathbf{L}_1$ can be identified in the limit from positive examples in polynomial time using an algorithm proposed by Lange and Weihagen [LaWe:90] (which does not identify the ordinary pattern languages).

A second variation of the concept of pattern language involves associating with each of the variables $x_1, ..., x_k$, a corresponding variable $\overline{x}_1, ..., \overline{x}_k$, respectively, and partitioning the alphabet Σ into two disjoint subsets Σ' and $\overline{\Sigma'}$, where $\overline{\Sigma'} = \{\overline{a} \mid a \in \Sigma'\}$. Whenever a string $u_i \in \Sigma^*$ is substituted for x_i in the pattern, the corresponding string $\overline{u}_i$ is substituted for $\overline{x}_i$. A canonical form is assumed, such that, for each i, the leftmost occurrence of x_i occurs before the leftmost occurrences of both $\overline{x}_i$ and x_{i+1}. The language described by a pattern p of this kind is denoted $L_2(p)$. The pattern identification algorithm based on prefix queries can be modified easily (by adding queries of whether $p_1...p_i\overline{x}_m$ is a prefix of p) to learn pattern languages in $\mathbf{L}_2$.

As an example of application, let $\Sigma = \{r, l, u, d\}$ denoting right, left, up, and down segments, respectively, with $\Sigma' = \{r, u\}$ and $\overline{\Sigma'} = \{l, d\}$, where $l = \overline{r}$ and $d = \overline{u}$. Pattern languages over this Σ represent line drawings in the form of strings using chain codes with 4 primitives . Note that pattern languages of the type $\mathbf{L}_2$ are able to represent families of closed contours with certain symmetries, since the contours can be constrained to have the same number of l's and r's and the same number of u's and d's.

Finally, a third type of extension, which gives rise to the class of pattern languages $\mathbf{L_3}$, involves fixing sets $A_1, ..., A_k \subseteq \Sigma^+$ such that the strings u_i to be substituted for the variable x_i are taken from the set A_i. For example, rectangles can be described by a pattern $p = x_1x_2\overline{x}_1\overline{x}_2$ over $\Sigma = \{r, l, u, d\}$ with x_1 replaced by a word in r^+ and x_2 replaced by a word in u^*. Again, the prefix query algorithm can be applied, but the learning of a language $L_3(p)$ is not complete unless the sets of strings A_i, $i = 1, ..., k$ are identified. Siromoney *et al.* claimed that the problem of identifying a pattern language in $\mathbf{L_3}$ can be solved if the sets A_i's are chosen from some class of learnable sets such as regular languages [SiSM:92], but they did not present any global learning procedure to solve it.