Efficient Learning of Regular Languages Using Teacher-Supplied Positive Samples and Learner-Generated Queries

Rajesh G. Parekh
Iowa State University

Vasant Honavar
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Artificial Intelligence and Robotics Commons, and the Theory and Algorithms Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/143

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Efficient Learning of Regular Languages Using Teacher-Supplied Positive Samples and Learner-Generated Queries

TR93-25
Rajesh Parekh and Vasant Honavar

October 19, 1993

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011
Efficient Learning of Regular Languages Using Teacher-Supplied Positive Samples and Learner-Generated Queries

Rajesh Parekh & Vasant Honavar
Department of Computer Science
226 Atanasoff Hall,
Iowa State University,
Ames, IA 50011. U.S.A.
e-mail: rgparekh@iastate.edu & honavar@iastate.edu

Abstract

We present a new algorithm for efficient learning of regular languages from examples and queries. A reliable teacher who knows the unknown regular grammar $G$ (or is able to determine if certain strings are accepted by the grammar) will guide the learner in achieving the goal of inferring an equivalent grammar $G^*$. The teacher provides the learner with a structurally complete set of positive examples belonging to the unknown grammar $G$. Using this information the learner constructs a canonical automaton which accepts exactly those examples. The canonical automaton defines a set of grammars which are ordered on a lattice to form the hypothesis space. A bi-directional search algorithm is used to systematically search the lattice for the solution $G^*$. While searching for the solution, the learner interacts with the teacher by posing queries. The teacher’s responses enable the learner to eliminate one or more points on the lattice which do not correspond to the correct solution. After successive eliminations, when only one lattice element remains, the process is terminated and that element is accepted as the solution $G^*$. The inferred grammar is proven to be equivalent to original grammar $G$.

---

1This research was partially supported by the Iowa State University College of Liberal Arts and Science
2A preliminary version of this report was published in [Parekh & Honavar, 1993a]
1 Introduction

1.1 Syntactic Pattern Recognition

The ability to recognize and classify patterns is a key characteristic of intelligent systems. In the conventional decision theoretic approach to pattern classification, a set of characteristic measurements called features is extracted from the patterns. The classification is performed on the basis of similarities of the feature vectors. Often, in pattern recognition applications, the structural information that describes each pattern is vital and the recognition task involves describing the structural characteristics of the patterns in addition to classification. In problems concerning scene analysis, 3-D object recognition, natural language processing, character recognition etc., the patterns are described in terms of structural relationships between pattern primitives. Complex patterns may be defined in terms of compositions of simpler sub-patterns. This sub-division of patterns ultimately yields simple atomic sub-patterns which are regarded as pattern primitives. A language called the pattern description language may be defined to provide the structural description of patterns in terms of pattern primitives and their compositions. The patterns may be distributed into several classes on the basis of structural similarity among patterns of the same class. Each class of patterns is represented by a grammar. The recognition process involves identification of pattern primitives followed by syntax analysis, or parsing, to determine if the pattern is grammatically correct i.e. if it is generated by the grammar corresponding to that class of patterns. Since this approach borrows from formal languages and grammars it is called the syntactic approach to pattern recognition. Learning to classify patterns thus reduces to learning the grammars which characterize pattern classes ([Fu, 1982], [Honavar, 1993]). The method of learning a grammar from a set of examples is called grammar inference. The inferred grammar is the description or classification of the observed symbol strings (patterns).

1.2 Regular Languages and Automata

A phrase structure grammar is a 4-tuple \( G = (V_N, V_T, P, S) \) where \( V_N \) is the set of non-terminal symbols, \( V_T \) is a set of terminal symbols, \( P \) is a finite set of production rules of the form \( \alpha \rightarrow \beta \) where, \( \alpha \) and \( \beta \) are strings over \( V_N \cup V_T \) and \( S \in V_N \) is the start symbol. The set of all strings derivable using \( G \) is referred to as the language \( L(G) \). By placing different restrictions on the form of the production rules \( P \), we get different classes of grammars (e.g. regular, context-free, and context-sensitive). Regular grammars have productions of the form: \( A \rightarrow AB \) or \( A \rightarrow b \) where \( A, B \in V_N \) and \( a, b \in V_T \). The language generated by a regular grammar is called a regular language. Regular grammars are the simplest among phrase structure grammars. Although it is well known that the descriptive power of regular grammars is fairly limited, for several practical applications, other grammars (e.g. context-free grammars) can be closely approximated by regular grammars [Fu, 1982]. Finite State Automata can be used to recognize regular grammars. A deterministic finite state automaton \( A \) is a quintuple \( A = (Q, \delta, \Sigma, q_0, F) \) where, \( Q \) is a finite set of states, \( \Sigma \) is the finite set of input symbols called the alphabet, \( F \subseteq Q \) is the set of accepting states, \( q_0 \in Q \) is the start state and \( \delta \) is the transition function \( Q \times \Sigma \rightarrow Q \) giving the next state of the automaton upon reading a particular symbol.
Finite State Machines are traditionally represented using state transition diagrams. A simple transition $\delta(p, a) \rightarrow q$ is depicted in Fig. 1. A non-deterministic finite state automaton has $\delta$ as a function $Q \times \Sigma \rightarrow 2^Q$. This means that $\delta(p, a)$ is subset (possibly empty) of $Q$ instead of a single state in $Q$. The deterministic and non-deterministic finite state automata are equivalent in expressive power and there are polynomial time algorithms which convert a non-deterministic finite state automaton to a finite state machine [Lewis & Papadimitriou, 1981]. We present an algorithm for inference of regular grammars which makes extensive use of finite state automata.

### 1.3 The Grammar Inference Problem

Grammatical Inference is the process of learning a rule-based grammar from a finite set of labeled examples. The grammar inference problem [Biermann & Feldman, 1972] may be described as follows: A finite set of symbol strings $S^+$ generated by an unknown grammar $G$ and possibly a finite set of strings $S^-$ generated by the complement grammar $\tilde{G}$ are known, and a grammar $G^*$ equivalent to the unknown grammar $G$ is to be discovered. $G$ is equivalent to $G^*$ if the languages accepted by $G$ and $G^*$ are exactly the same. More formally, given $S^+ = \{ \alpha \mid \alpha \in G \}$ and possibly $S^- = \{ \beta \mid \beta \notin G \}$ infer a grammar $G^*$ such that $L(G^*) = L(G)$. For a discussion on grammar inference in machine learning see [Fu, 1982] and [Parekh & Honavar, 1993c].

Several solutions to the grammar inference problem have been proposed. These include the uv$^k$w algorithm and the k-tails method due to Solomonoff and Biermann & Feldman respectively [Fu, 1982]. Owing to space constraints we present only a brief synopsis of these methods. The k-tails method involves defining equivalence classes on the canonical grammar generated by the complete positive sample set $S^+$. By use of a heuristic state merging process, one solution is obtained for each $k$ such that $0 < k < (|Q| - 2)$ where $Q$ is the set of states of the canonical grammar. The inferred language $L(G^*)$ is a superset of $L(G)$ but is not guaranteed to be the same as the target language. The uv$^k$w algorithm is designed to search the sample space for repeated sub-strings and makes the hypothesis that these are generated by recursive rules. The depth of recursion which is an important parameter is chosen by the user. The recursive rules thus obtained characterize the grammar being sought. Both these methods are heuristic in nature and rely only on the information provided by the positive samples belonging to $S^+$. The importance of using information from negative samples in learning is well known [Gold, 1967]. There is a potentially infinite number of grammars which are consistent with a given set of positive samples. In order to uniquely identify the grammar it is essential to use information provided by the negative samples.
Giles et al. have done extensive research in grammatical inference using recurrent neural networks [Giles et al., 1991]. They have proposed an incremental, real-time, recurrent learning method that computes the complete gradient and updates the weights of the network after each example string is presented. A dynamic clustering algorithm is then used to extract the production rules that the grammar has learned. Rivest and Schapire have suggested a powerful new technique for grammar inference based on homing sequences [Schapire, 1991]. The output produced in executing the homing sequence completely determines the state reached by the automaton at the end of the homing sequence. Every finite state machine has a homing sequence and Schapire’s technique infers the homing sequence as part of the overall inference procedure. The algorithm is based on the availability of positive samples and counterexamples.

We present an algorithm for grammar inference in which positive samples are provided by a teacher and information about negative samples is given by the teacher in the form of answers to queries. Using this information the learner can perform the inference by searching through a space of candidate solutions. The use of a teacher-student model for inference of regular grammars was originally proposed by Pao [Pao & Carr, 1978]. Their method for mapping candidate grammars to a regular structure (lattice) is used in our algorithm. However, their formulation contained a few errors [Parekh, 1993]. In addition to correcting these errors, our algorithm employs a bi-directional search to probe the space of candidate grammars. This results in considerable speed improvement over Pao’s method. Section 2 describes the framework of the algorithm. Section 3 presents the actual algorithm formally and Section 4 details the proof of correctness. Finally, Section 5 discusses the advantages of our method and points out directions for future work in this area. All concepts introduced in the following sections are illustrated using an example.

2 Framework of the Algorithm

We present a method to solve the grammar inference problem which uses information from both positive and negative samples. A teacher provides the set of positive samples $S^+$ using which the learner constructs a set of candidate grammars (forming the hypothesis space). To eliminate candidate grammars the learner then generates queries which are classified as either positive or negative samples of the target grammar by the teacher. Based on the teacher’s response the learner discards one or more candidate solutions. This interaction between the teacher and the learner continues till only one candidate solution remains. The four key steps in the algorithm are:

1. Construction of the Canonical Automaton - $M_{S^+}$.
2. Construction of the Lattice - $\omega$.
3. Query generation.
4. Candidate elimination.
$S^+ = \{ \text{aa}, \text{b} \}$

Figure 2: Target Automaton - $M_G$ and Canonical Automaton - $M_{S^+}$

2.1 Constructing the canonical automaton - $M_{S^+}$

A teacher provides a set of symbol strings $S^+$ to the learner. $S^+$ needs to be a structurally complete set for correctly inferring the grammar $G^*$. A structurally complete set of symbol strings covers each production of the grammar $G$ at least once. Equivalently, if $M_G$ is the finite state automaton corresponding to the grammar $G$, then each arc of $M_G$ must be covered at least once by some symbol string $x$ in $S^+$. Using this information the learner constructs a canonical automaton $M_{S^+}$ which accepts only the strings which belong to $S^+$. [Pao & Carr, 1978]. The strings in $S^+$ are ordered by increasing length. Let $s_0$ be the start state of $M_{S^+}$ and $s_F$ be the final state. A path is constructed for the first string $x_0 = a_1a_2 \ldots a_n$ with $a_i$ being an edge from $s_{i-1}$ to $s_i$ and $a_n$ being an edge from $s_{n-1}$ to $s_F$. For all the other strings $x_i$ in the set $S^+$, if there is an existing path from $s_0$ for a prefix of $x_i$, then that path is followed. If there is no edge corresponding to a symbol $a_k$ along the path being followed then a new path is constructed with the remaining symbols of the string leading from one new state to another. The last symbol labels an arc terminating in $s_F$.

Example: Consider that the language of the automaton $M_G$ in Fig. 2 is to be inferred. The teacher provides a structurally complete set $S^+ = \{ \text{aa}, \text{b} \}$ from which the learner can construct $M_{S^+}$ shown in Fig. 2.
2.2 Constructing the Lattice $\omega$

2.2.1 Partitioning the set of states of $M_{S^+}$

The lattice is constructed by systematically merging the states of the canonical automaton to form partitions. Each partition of states is a lattice element. Individual elements of a partition are called cells.

1. To each partition (obtained for an automaton with N-1 states) add the new state (N) as a separate cell.

2. To each existing cell in each partition (of the automaton with N-1 states) add the new state (N).

It is shown in Section 4 that the solution to the grammar inference is one of these partitions.

**Example:** If an automaton has just one state (numbered 0) then the only partition will be $\{0\}$. The partitions of a 2 state machine will now be: $\{0,1\}$ (by step 1) and $\{01\}$ (by step 2). There are 15 partitions of the canonical automaton $M_{S^+}$ from our example. These are depicted in Fig. 3.

2.2.2 The Grammar Cover relation

The lattice (Fig. 3) is organized from bottom to top in the decreasing order of number of cells per partition. i.e. the bottom-most element is the partition with N cells. Partitions
with N-1 cells are placed (in arbitrary order) in the second level and so on. The topmost element has just one cell. The grammar cover relation between elements in successive levels of the lattice is described as follows. If each cell of a partition say \( P_i \) on one level is contained in some cell of a partition say \( P_j \) in the above level then \( P_j \) covers \( P_i \) (\( P_i \subseteq P_j \)). This means that the language accepted the automaton corresponding to partition \( P_j \) is a super set of the language accepted by the automaton corresponding to partition \( P_i \) and is indicated in Fig. 3 by an arrow from \( P_i \) to \( P_j \). The learner exploits this relation between partitions along with the information provided by the teacher in the form of answers to the queries to eliminate one or more lattice elements. A list of partitions \( \omega_L \) which represents the hypothesis space is constructed in order from the bottom of the lattice to the top. For our example \( \omega_L = \{ P_0, P_1, \ldots, P_{14} \} \).

### 2.2.3 Constructing a FSA from a partition

A finite state automaton is constructed corresponding to each partition of states in the lattice by systematically merging the states in each cell of the partition. Each cell of a partition forms a state in the FSA being constructed. The FSA is a five tuple \((Q, \delta, \Sigma, q_0, A)\). Each cell within the partition forms a state. Thus the set \( Q \) is the set of the individual cells. The start state \( q_0 \) is the cell containing the start state \( s_0 \) of the canonical automaton and the accepting state \( A \) is the cell \( s_F \) of the partition. The alphabet \( \Sigma \) is the same as that of the canonical automaton while the transition function \( \delta \) is defined on the basis of the transitions within the canonical automaton. If several states of the canonical automaton are merged together in a cell then the inputs to each of those states will be inputs to the merged state and the outputs from each of them will be the output from the merged state. Transitions from one state to another in the same cell form self loops.

### 2.2.4 Equivalence of Finite State Automata

The test for equivalence of finite state automata [Harrison, 1965], [Rabin & Scott, 1959] is summarized below. Let \( U = (S, M, \Sigma, s_0, F) \) and \( V = (T, N, \Sigma, t_0, G) \) be two finite state automata. The cross-product machine without output is defined as \( U \times V = (W, P, \Sigma, w_0) \) where \( W = S \times T \), \( w_0 = (s_0, t_0) \) and \( P((s, t), \sigma) = (M(s, \sigma), N(t, \sigma)) \) for all \( \sigma \in \Sigma \). The choice of the set of accepting states \( (H) \) defines three different machines:

1. The direct union. \( H_1 = \{(s, t) \mid s \in F \text{ or } t \in G\} \)

2. The direct intersection. \( H_2 = \{(s, t) \mid s \in F \text{ and } t \in G\} \)

3. The difference machine. \( H_3 = \{(s, t) \mid s \in F \text{ and } t \in T - G\} \)

If \( H_1 = H_2 \), we know that \( U \cup V \) is the same as \( U \cap V \). Hence, \( L(U \cup V) = L(U) \cup L(V) = L(U) \cap L(V) \). Therefore, \( L(U) = L(V) \) and \( U \equiv V \). If \( H_1 \neq H_2 \), the two automata are equivalent only if the language \( L(U - V) \cup L(V - U) \) is empty. The language of a finite state machine is not an empty set if the finite state machine accepts a string \( x \) of length less than the number of states of that machine. Since we are dealing with finite state automata, there can only be a finite number of strings of length less than \( n \) where \( n \) is the number of states of \( U - V, V - U \). The moment we find some string \( x \) such that \( x \in U - V \) or \( x \in V - U \) we conclude that \( U \neq V \) and the string \( x \) which is accepted by the difference machine of \( U \) and \( V \) is the query to be posed to the teacher.
2.3 Query Generation

During the search process, fully specified, deterministic finite state automata \((M_i = \{S, \delta_i, \Sigma, s_0, A\})\) and \((M_j = \{T, \delta_j, \Sigma, t_0, B\})\) are constructed for two partitions \(P_i\) and \(P_j\). If \(M_i\) and \(M_j\) are found to be equivalent then one of these two partitions is deleted from \(\omega_L\). However, if they are not equivalent then there exists at least one input string \(y\) such that \(\delta_i(s_0, y) \in A\) but \(\delta_i(t_0, y) \notin B\) or vice-versa (in which case the roles of \(M_i\) and \(M_j\) are simply reversed). This string \(y\) belongs to the difference machine \([\text{Harrison}, 1965]\) is \(M_i - M_j = \{W, \delta_w, \Sigma, w_0, C\}\) where, \(W = S \times T, w_0 = (s_0, t_0)\), \(\delta_w( (s,t), \sigma) = (\delta_i(s, \sigma), \delta_i(t, \sigma))\) for all \(\sigma \in \Sigma\) and \(C = {(s,t) | s \in A\text{ and } t \in T - B}\). The query is of the form: “\(y \in G\?”

2.4 Elimination of Candidate grammars

The learner eliminates one or more candidate grammars in the hypothesis space on the basis of the teacher’s answer to the query. Since \(y \in M_i - M_j\) it is clear that \(y \in M_i\) and \(y \notin M_j\). Now if \(y \in G\), then since \(M_j\) does not accept \(y\) (a positive sample), the partition \(P_j\) is eliminated from the list \(\omega_L\). By the grammar cover relation all partitions \(P_k\) such that \(P_k \subseteq P_j\) can be eliminated because all these lower bounds of \(P_j\) will also not accept the string \(y\) which is a positive example. Similarly, if \(y \notin G\) then clearly the partition \(P_i\) cannot correspond to a solution since \(M_i\) accepts a negative example. All partitions \(P_l\) such that \(P_i \subseteq P_l\) are out of contention because these upper bounds of \(P_i\) will also accept the negative example \(y\). Ultimately, the only partition remaining in the list \(\omega_L\) corresponds to the inferred grammar \(G^*\).

3 The Grammar Inference Algorithm

A bi-directional search \([\text{Pohl}, 1971]\) comprising of two independent searches (which may proceed in parallel) is used to navigate through the lattice (hypothesis space) for the solution. The two searches start from opposite ends of the lattice. One proceeds top-down while the other one is bottom-up. Although in the worst case even the bi-directional search is exponential there is empirical evidence that in the average case the search significantly outperforms a uni-directional search. The efficiency of the bi-directional search can be improved further by conducting the two searches in parallel if the number of available partitions in the list \(\omega_L\) is greater or equal to four. Otherwise, only one of the two searches is conducted. These steps are repeated till only one partition is left behind in the list \(\omega_L\). The algorithm is formally stated below:

\[
\text{WHILE } |\omega_L| > 1 \text{ DO }
\]

- **Bottom-Up Search**:

  1. Select the two partitions \((P_i\text{ and } P_j)\) appearing in the left-most positions of the list of partitions \(\omega_L\).

  2. Construct fully specified, deterministic finite state automata \(U_i\) and \(U_j\) (corresponding to the partitions \(P_i\) and \(P_j\) respectively).
3. (a) If $U_i \equiv U_j$ then delete the partition $P_i$ from the list $\omega_L$.
   (b) Otherwise construct the difference machine $U_i - U_j$. Obtain a string $y \in U_i - U_j$ and pose the query $"y \in G?"$ to the teacher.
   (c) If $y \in G$ delete the partition $P_j$ from the list $\omega_L$ otherwise delete the partition $P_i$ and all partitions $P_l$ such that $P_i \subseteq P_l$.

- **Top-Down Search**:

  1. Select the two partitions ($Q_i$ and $Q_j$) appearing in the right-most positions of the list of partitions $\omega_L$.
  2. Construct fully specified, deterministic finite state automata $V_i$ and $V_j$ (corresponding to the partitions $Q_i$ and $Q_j$ respectively).
  3. (a) If $V_i \equiv V_j$ then delete the partition $Q_i$ from the list $\omega_L$.
     (b) Otherwise construct the difference machine $V_i - V_j$. Obtain a string $y \in V_i - V_j$ and pose the query $"y \in G?"$ to the teacher.
     (c) If $y \notin G$ then delete the partition $Q_i$ from the list $\omega_L$ otherwise delete the partition $Q_j$ and all partitions $Q_k$ such that $Q_k \subseteq Q_j$.

**Example**: Partial execution of the grammar inference algorithm is depicted in figures 4, 5, and 6. By eliminating partitions during the search we finally converge on the partition $P_{11}$. The corresponding automaton $M_{11}$ (Fig. 7) is exactly the same as the target automaton $M_G$. In practice, we may end up with a $M_{G^*} \equiv M_G$.

Fig. 4 depicts a bottom-up search wherein automata $M_0$ and $M_1$ are compared for equivalence. A test for equivalence shows that $M_0 \neq M_1$. The string $y = ab \in M_1 - M_0$. Since $y \in G$, $P_0$ is eliminated. In parallel, a top-down search as shown in Fig. 5 compares $M_{14}$ and $M_{13}$. Again $M_{14} \neq M_{13}$. The string $y = ab \in M_{14} - M_{13}$. Since $y \in G$, $P_{13}$ is eliminated from the list of partitions. In addition to this, the lower bounds of $P_{13}$ i.e. $P_3$ and $P_4$ as the automata corresponding to these partitions also do not accept the positive example $y = ab$. Fig. 6 depicts a test of equivalence of $M_1$ and $M_2$. The result is $M_1 \neq M_2$. The query $y = \lambda$ (null string) is accepted by $M_2$ but does not belong to $G$. Thus, $P_2$ is eliminated for accepting a negative sample. The upper bounds of $P_2$ i.e. $P_7$, $P_9$, and $P_{12}$ which also accept the null string are eliminated from the list $\omega_L$. Thus, we see that the search quickly eliminates candidate grammars which cannot correspond to the solution. Fig. 7 shows the solution automaton $M_{11}$ which is in fact exactly the same as the target automaton in Fig. 2.
Figure 4: Finite State Automata - $M_0$ and $M_1$

Figure 5: Finite State Automata - $M_{14}$ and $M_{13}$
Figure 6: Finite State Automata - $M_1$ and $M_2$

Figure 7: Finite State Automaton $M_{11}$
4 Proof of Correctness

The proof of correctness of our algorithm directly follows from the two theorems stated below:

**Theorem 1:** The solution grammar lies in the hypothesis space defined by the lattice.

**Proof:**
Let $Z_i = \{S, \delta_i, \Sigma, s_0, A\}$ and $Z_j = \{T, \delta_i, \Sigma, t_0, B\}$ be two deterministic, finite state machines. The state $s_i$ of $Z_i$ is said to be similar to the state $t_j$ of $Z_j$ (denoted as $s_i \sim t_j$) if one of the following two conditions is fulfilled:

1. $s_i = s_0$ and $t_j = t_0$ i.e. the start states of the two finite state machines.

2. There exists a state $s_x$ of $Z_i$ and $t_y$ of $Z_j$ such that:
   
   (a) $s_x \sim t_y$, and
   
   (b) $\delta_i(s_x, \gamma) = s_i$ and $\delta_i(t_y, \gamma) = t_j$.

**Lemma 1:** Two deterministic finite state machines are behaviorally equivalent if the following two conditions hold:

1. $s_i \sim t_j$ for $s_i \in S$ and $t_j \in T$ is a one-to-one and onto function, and

2. If $s_i \sim t_j$ then $s_i \in A$ if $t_j \in B$.

**Proof:**
Let $\psi$ be a one-to-one and onto function mapping the set $S$ to the set $T$ such that $\psi(s_i) = t_j$ if $s_i \sim t_j$. If we can prove the following then by condition 2 above we can conclude that the two finite state machines are behaviorally equivalent.

$$x \in \text{path}(s_0, s_i) \iff x \in \text{path}(\psi(s_0, \psi(s_i))) \text{ for all } s_i \text{ of } Z_i$$

(1)

We prove $x \in \text{path}(s_0, s_i) \implies x \in \text{path}(\psi(s_0, \psi(s_i)))$ for all $s_i$ of $Z_i$ by mathematical induction on the length of string $x$ (i.e. $|x|$).

**Base Case:**
If $|x| = 1$ i.e. $x$ is an edge from state $s_0$ to a state $s_i$ of $Z_i$. We know that $s_0 \sim t_0$ and there is a state $t_j$ such that $s_i \sim t_j$. Therefore, $x$ is an edge from $t_0$ to state $t_j$ of $Z_j$. Since $t_0 = \psi(s_0)$ and $t_j = \psi(s_i)$ we have proved the base case.

**Induction Hypothesis:**
Assume that the lemma is true for $|x| \leq n$.

If $|x| = n$, then $x \in \text{path}(s_0, s_{pn}) \implies x \in \text{path}(\psi(s_0, \psi(s_{pn}))$ for a state $s_{pn}$ of $Z_i$.

**Induction Proof:**
Now we show that the lemma is true for $|x| = n + 1$. Let $x = a_1a_2a_3\ldots a_{n+1}$. Since $x$ is a path from $s_0$ to $s_i$ then there exist states $s_{p_1}, s_{p_2}, \ldots, s_{p_n}$ in $S$ such that $a_1$ is an edge from $s_0$ to a state $s_{p_1}$, $a_j$ is an edge from $s_{p_{j-1}}$ to a state $s_{p_j}$ for $2 \leq j \leq n$, and $a_{n+1}$ is an edge from $s_{p_n}$ to a state $s_i$. By the induction hypothesis, we know that $a_1a_2a_3\ldots a_n$ is a path from $s_0$ to $s_{pn}$. Thus, $a_1a_2a_3\ldots a_n$ is a path from $\psi(s_0)$ to $\psi(s_{pn})$. Now $s_0 \sim t_0$ and there is only one state $t_{qn} \sim s_{pn}$. From this we conclude that $a_1a_2a_3\ldots a_n$ is a path from...
t_0 to t_{yn}. Since there is only one state t_j of Z_j which is similar to s_i of Z_i we conclude that a_{n+1} is an edge from t_{yn} to t_j. Thus, x is a path from t_0 to t_j i.e. x is a path from \( \psi(s_0) \) to \( \psi(s_i) \). The converse can be proved by similar argument.

Given \( S^+ \subseteq L(G) \) we constructed the canonical automaton \( M_{S^+} \) as described in Section 2. We establish the following two lemmas.

**Lemma 2:** For each \( s_i \) of \( M_{S^+} \) there is exactly one state \( t_j \) of \( M_G \) such that \( s_i \sim t_j \).

**Proof:**
For any string \( x = a_1a_2a_3 \ldots a_n \) of \( M_{S^+} \) we know that there exist states \( s_{p1}, s_{p2}, \ldots s_{pn} \) in \( S \) such that \( a_1 \) is an edge from \( s_0 \) to a state \( s_{p1}, a_j \) is an edge from \( s_{pj-1} \) to a state \( s_{pj} \) for \( 2 \leq j \leq n \). Since \( x \) is contained in \( L(G) \) there are states \( t_{r1}, t_{r2}, \ldots t_{rn} \) in \( M_G \) such that \( a_1 \) is an edge from \( t_0 \) to a state \( t_{r1}, a_j \) is an edge from \( t_{rj-1} \) to a state \( t_{rj} \) for \( 2 \leq j \leq n \). By definition of similarity of states we know that \( s_0 \sim t_0 \) and from above we conclude that \( s_{pj} \sim t_{rj} \) for \( 1 \leq j \leq n \). Since the languages generated by finite state machines are deterministic (unambiguous), there is one and only one way to generate a string a given string \( x \in L(G) \). Thus, we have proved that for each \( s_i \) of \( M_{S^+} \) there is exactly one \( t_j \) of \( M_G \) such that \( s_i \sim t_j \).

**Lemma 3:** For each \( t_j \) of \( M_G \) there is at least one state \( s_i \) of \( M_{S^+} \) such that \( s_i \sim t_j \).

**Proof:**
Since the construction of \( S^+ \) requires that each edge of the machine \( M_G \) must be covered at least once, we know that for any edge of \( M_G \) say ‘a’ from \( t_{rk} \) to \( t_{rl} \) there exist states \( s_{pk} \) and \( s_{pl} \) of \( M_{S^+} \) with the edge ‘a’ connecting them. Thus, for each \( t_j \) of \( M_G \) there is at least one state \( s_i \) of \( M_{S^+} \) such that \( s_i \sim t_j \).

**Constructing the solution automaton \( M_G^* \)**

Merge each state \( s_{p1}, s_{p2}, \ldots s_{pk} \) of the automaton \( M_{S^+} \) such that each \( s_{pn} \sim t_j \) where \( n = 1, 2, \ldots k \) into a single state \( s_i \). Thus, \( s_i \in M_{S^+} \sim t_j \in M_G^* \). The new machine thus obtained is called \( M_G^* \). From lemma 2 we know that each \( s_{pn} \) \( (n = 1, 2, \ldots) \) is merged into exactly one \( s_i \). From lemma 3 it is clear that for each \( t_j \) of \( M_G^* \) there is exactly one \( s_i \) of \( M_G^* \). Thus, there is a one-to-one and onto correspondence between \( s_i \) and \( t_j \). A state \( s_F \) of \( M_G^* \) is a final/accepting state if a state \( s_{pG} \) of \( M_{S^+} \) is merged into \( s_F \) and \( s_{pG} \) is an accepting state for \( M_{S^+} \). Thus, using Theorem 1, lemma 2, lemma 3, and the above method of constructing the solution automaton we have proved that \( M_G^* \equiv M_G \).

**Theorem 2:** The solution is not deleted during the search process.

**Proof:**
A candidate grammar is eliminated from the lattice during the search process if one of the following conditions are satisfied.

1. It is equivalent to another grammar.
2. It incorrectly accepts a negative example.

Clearly, the candidate grammar cannot be a solution and its corresponding partition is eliminated. If the language of the eliminated partition is \( L(G') \), and the languages
of the partitions which are its upper bounds are \( L(G^k_i) \) (\( k = 1, 2, 3 \ldots \)) then, by the properties of the lattice we know that \( L(G^k_i) \supseteq L(G_i) \) for all \( k \). Thus, each of the upper bounds which would accept the same negative example, cannot correspond to the correct solution.

3. It does not accept a positive example.

Since the candidate grammar fails to accept a positive example it cannot be a solution. The corresponding partition is therefore rightly eliminated. If the language of the eliminated partition is \( L(G^k_i) \), and the languages of the partitions which are its lower bounds are \( L(G^k_i) \) (\( k = 1, 2, 3 \ldots \)) then, by the properties of the lattice we know that \( L(G^k_i) \supseteq L(G_i) \) for all \( k \). All lower bounds of the eliminated candidate grammar will also not accept the positive example and thus are ruled out.

In each of the above three cases we have seen that the correct solution is always preserved in the lattice during the search process. The search process terminates when there is only one partition remaining in the lattice. This corresponds to the correct solution.

5 Summary and Future Work

We have presented a provably correct mechanism for inference of regular grammars. Bi-directional search is used to efficiently search the lattice for the correct solution. On an average the bi-directional search effectively prunes the search space thus resulting in considerable saving of time when compared with a uni-directional search. Thus, we conjecture that our method will result in the examination of considerably fewer finite-state automata than the method proposed by Pao. Quite a few issues need to be explored thoroughly before a definitive statement can be made on the superiority of our algorithm.

The grammar inference algorithm proposed here, can potentially be used in several pattern recognition applications. These include, among others, chromosome identification, speech synthesis, and DNA analysis [Miclet & Quinqueton, 1986], [Gonzales & Thomason, 1978], and [Fu, 1982]. The performance of an implementation of the grammar inference algorithm proposed here is being evaluated in several different application domains. The average case analysis of the bi-directional search, detailed experiments, and comparisons with performances of other grammar inference methods is in progress [Parekh & Honavar, 1993b].

A few promising directions for future research are enlisted hereunder:

1. **Lambda Pruning:**

   A simple modification to the algorithm could potentially yield considerable savings in terms of the number of candidate grammars that are examined. Systematic merging of the states of the canonical automaton \( M_{S^+} \) results in several candidate automata that accept the null string \( \lambda \) and several that don’t. Depending upon whether the target grammar accepts \( \lambda \) or not a significant number of candidate solutions can be eliminated (i.e. the ones which differ from the target grammar on accepting the null string.)
2. **Compact lattice representation:**
   Given the properties of the lattice, it might be possible to construct the lattice incrementally as the search progresses. This would result in a considerable saving of memory.

3. **Efficient search techniques:**
   If some additional knowledge about the type of the grammar is available it may be possible to use certain heuristics which may improve the search efficiency and help in quickly converging to the solution.

4. **Intelligent query generation:**
   Currently the queries are generated by the learner with reference to only two automata corresponding to two partitions of the lattice. With more knowledge about the type of the grammar one might be able to generate more intelligent queries which can eliminate a considerably larger number of candidate solutions.

5. **Extensions:**
   *Regular tree grammars* and *attributed grammars* have similar characteristics as those of regular finite state grammars so it will be useful to modify this algorithm to facilitate inference of these grammars.

6. **Incremental Learning:**
   The current algorithm is based on prior availability of the set of positive samples $S^+$. In practical applications, it is often the case that all of these positive samples are not available at the start. An incremental algorithm, which is able to refine the inferred solution based on the availability of new positive samples will be immensely useful in several applications.

7. **Issues related to structural completeness of $S^+$:**
   The current algorithm is based on the assumption of structural completeness of the set $S^+$. In practice it may not be possible to guarantee that the set of positive samples is structurally complete. We would like to see to what extent the performance of our grammar inference algorithm is affected (i.e. how significantly the inferred grammar $G^+$ differs from the target grammar $G$) when the set $S^+$ is structurally incomplete.

### References


