Learning k-bounded context-free grammars

Dana Angluin *
Yale University

August 1987

Abstract

A context-free grammar is *k-bounded* if no production has more than k occurrences of non-terminals in its right-hand side. We demonstrate an algorithm to learn k-bounded context-free grammars using equivalence queries and non-terminal membership queries that runs in time polynomial in the size of the unknown grammar and the length of the longest counter-example.

1 Introduction

We consider the problem of learning a context-free grammar using queries to a teacher. The paper [1] shows that the regular sets can be learned by an algorithm that uses equivalence and membership queries and runs in time polynomial in the size of the smallest deterministic finite acceptor for the unknown set and the length of the longest counter-example returned by an equivalence query.

Berman and Roos [3] show that this result can be extended to the class of languages recognized by deterministic one-counter automata. The deterministic one-counter languages are properly contained in the context-free languages and properly contain the regular languages. A non-regular example from this class is the set of all strings of a's and b's with twice as many a's as b's.

The question of whether there is an analogous result for the full class of context-free languages is open.

We introduce two additional assumptions: that the unknown grammar is k-bounded for some fixed k, and that non-terminal membership queries are available. The assumption of non-terminal membership queries is a strong one: the queries refer not just to the unknown language, but also to the structure of the unknown grammar.

Two implemented systems are particularly relevant to this investigation. Knobe and Knobe [5] describe a system for learning a context-free grammar using equivalence and membership queries. Shapiro [7] gives an algorithm for learning a restricted class of k-bounded context-free grammars using (in effect) full equivalence and non-terminal membership queries. Both systems successfully learned some interesting grammars. This paper may be viewed as an analysis of a variant of Shapiro's algorithm.

*Supported by NSF grant IRI-8404226

The algorithm described in this paper achieves the polynomial time bound claimed for it, but it is not particularly practical. Some partial suggestions for making it more reasonable appear at the end of the paper.

2 Preliminaries

2.1 Context-free grammars and languages

Our notions of context-free grammars and languages are mostly standard. See reference [4] for further information.

An *alphabet* is a finite non-empty set of distinct symbols. If X is an alphabet, the set of finite strings of symbols from X is denoted X^*. The empty string is denoted λ. The length of the string w is denoted $|w|$. If F is a finite set, $|F|$ denotes the cardinality of F.

Given an ordered pair of alphabets N and T, the *context-free productions on N and T* is the set $N \times (N \cup T)^*$. Since we consider only context-free productions, they will be called simply *productions*. The production (A, α) is denoted $A \rightarrow \alpha$. The left-hand side is A and the right-hand side is α. Note that α may be λ.

A *context-free grammar* is a 4-tuple (T, N, P, S), where T is the alphabet of *terminal symbols*, N is the alphabet of *non-terminal symbols*, $S \in N$ is the *start symbol*, and P is a finite subset of the set of productions on N and T. P is called the set of *productions of G*. The *size* of the grammar G, denoted *size*(G), is the sum of $|T|$, $|N|$, $|P|$, and the sum of the lengths of the right-hand sides of all the productions in P.

Let $G = (T, N, P, S)$ be a context-free grammar and let β and γ be elements of $(T \cup N)^*$. γ is *derived from β in one step* if and only if there is a production $A \rightarrow \alpha$ of G such that for some strings β_1 and β_2,

$$\beta = \beta_1 A \beta_2,$$

and

$$\gamma = \beta_1 \alpha \beta_2.$$

That is, γ is derived from β by replacing one occurrence of A by α. This is denoted $\beta \Rightarrow \gamma$. The reflexive, transitive closure of $\Rightarrow$ is denoted $\Rightarrow^*$. The *language* of a non-terminal symbol A, denoted $L(A)$, is the set of all $w \in T^*$ such that $A \Rightarrow^* w$. (To emphasize the grammar being used, we use the subscript G, *e.g.*, $S \Rightarrow_G^* w$ or $L_G(A)$.)

The *language* of the grammar G, denoted $L(G)$, is just $L(S)$, where S is the start symbol. Two grammars G and G' are *equivalent* if and only if $L(G) = L(G')$. They are *fully equivalent* if and only if they contain the same non-terminal alphabet N, and for every $A \in N$, $L_G(A) = L_{G'}(A)$.

G is *k-bounded* if and only if the right-hand side of every production in P contains at most k occurrences of non-terminal symbols. For example, linear grammars are 1-bounded and Chomsky normal form grammars are 2-bounded. A context-free grammar is *simple* if and only if there is only one non-terminal symbol, namely, the start symbol S. An example of a 1-bounded simple grammar is $G_1 = (\{a, b\}, \{S\}, P_1, S)$, where P_1 is the set of productions

$$P_1 = \{S \rightarrow a, S \rightarrow b, S \rightarrow aSa, S \rightarrow bSb\}.$$

With the usual abbreviating conventions, G_1 is simply written as

$$S \to a \mid b \mid aSa \mid bSb.$$

This grammar generates the language of all non-empty palindromes over the alphabet $\{a, b\}$. This language cannot be recognized by a deterministic one-counter automaton.

2.2 Parse-DAGs

In order to be able to handle productions with λ on the right-hand side smoothly, we introduce a slight generalization of a parse-tree, called a *parse-DAG*. Let the grammar $G = (T, N, P, S)$ be fixed.

A parse-DAG for G is a finite directed acyclic graph with at least two nodes that has a number of special properties. At each node there is a fixed linear ordering, called *left-to-right*, of the edges directed out of that node. There is exactly one node with in-degree zero, called the *root*. Each node with out-degree zero is called a *leaf*; the other nodes are called *internal nodes*. Every internal node has a label consisting of an element of N, and every leaf has a label consisting of an element of T^*. For each node n, the nodes n' such that there is an edge from n to n' are called the *children* of n. Finally, for each internal node n, if its label is A and the concatenation of the labels of its children in left-to-right order is α, then $A \to \alpha$ is an element of P.

The *size* of the parse-DAG D, denoted $size(D)$, is defined to be the sum of the number of nodes in D, the number of edges in D, and the sum of the lengths of the labels on all the nodes of D. The *depth* of a parse-DAG D is the maximum number of nodes in any directed path in D.

If n is any internal node of a parse-DAG D, the *sub-DAG rooted at n* is the induced subgraph of D on all the nodes reachable from n by a directed path in D. Note that this is also a parse-DAG for the grammar G.

We define the *yield* of any node of a parse-DAG inductively as follows. The yield of any leaf is just its label, an element of T^*. Once the yields have been determined for the children of a node n, the yield of n is the concatenation of the yields of its children in left-to-right order. The *yield* of a parse-DAG is the yield of its root node. Note that the yield of every node of a parse-DAG can be computed in time polynomial in its size.

An example showing the advantages of a parse-DAG over a parse tree is the grammar G_2 with productions $S \to A_1A_1$, $A_i \to A_{i+1}A_{i+1}$ for each $i = 1, \ldots, n-1$, and $A_n \to \lambda$. To witness that $S \Rightarrow^* \lambda$ requires a parse tree with 2^n leaves. However, a parse-DAG with $n+2$ nodes labelled S, $A_1, \ldots, A_n$, and λ, and two edges from each of these nodes to its successor suffices to to witness $A \Rightarrow^* \lambda$.

2.3 Replacements, incorrectness, correctness

Assume that a non-terminal alphabet N and a terminal alphabet T are fixed. A *replacement* is a finite tuple

$$\langle (y_1, A_1), \ldots, (y_m, A_m) \rangle$$

such that each y_i is a finite string of terminal symbols, and each A_i is a non-terminal symbol.

Suppose ρ is the replacement $\langle (y_1, A_1), \ldots, (y_m, A_m) \rangle$, and β is a finite string of symbols from $T \cup N$. Then ρ is *compatible* with β if and only if there are finite strings of terminal symbols $x_0, \ldots, x_m$ such that

$$\beta = x_0 A_1 x_1 A_2 \cdots A_m x_m.$$

(That is, the successive occurrences of non-terminals in β are precisely $A_1, \ldots, A_m$.) If ρ is compatible with β, then the application of ρ to β, denoted $\rho[\beta]$, is the terminal string obtained by substituting the terminal string y_i for the i^{th} occurrence of a non-terminal symbol in β, that is,

$$\rho[\beta] = x_0 y_1 x_1 y_2 \cdots y_m x_m.$$

(A replacement may be empty, in which case it is only compatible with terminal strings, and the result of applying it to a terminal string w is w.)

For example, the replacement

$$\rho = \langle (bug, A)(bit, B)(man, A) \rangle$$

is compatible with $\beta = aABtheA$, and

$$\rho[\beta] = abugbittheman.$$

We define a production $A \rightarrow \alpha$ to be *incorrect for* a grammar $G = (T, N, P, S)$ if and only if there is a replacement

$$\rho = \langle (y_1, A_1), \ldots, (y_m, A_m) \rangle$$

that is compatible with α such that for each i, $y_i \in L_G(A_i)$, but the string $y = \rho[\alpha]$ is not in $L_G(A)$. A production is *correct for* G if and only if it is not incorrect for G. Clearly, every production in P is correct for G.

2.4 Types of queries

Suppose G is the unknown grammar. This is the grammar that we assume is known to the teacher, and that is to be learned (up to equivalence) by the learning algorithm.

A *membership query* proposes a string w and asks whether it is in $L(G)$. The reply is either *yes* or *no*.

A *non-terminal membership query* proposes a string w and a non-terminal symbol A of G and asks whether w is in $L(A)$. The reply is either *yes* or *no*. If the start symbol S is known, then a membership query with w can be accomplished by a non-terminal membership query with w and S.

An *equivalence query* proposes a grammar G' and asks whether $L(G) = L(G')$. The reply is *yes* or *no*. If it is *no*, then a *counter-example* is also provided, that is, a string w such that w is in $L(G)$ and not $L(G')$, or vice versa. If $w \in L(G) - L(G')$, w is called a *positive* counter-example, and if $w \in L(G') - L(G)$, it is called a *negative* counter-example.

The choice of a counter-example is assumed to be arbitrary. Note that equivalence of context-free grammars is an algorithmically unsolvable problem. The paper [2] suggests how to replace equivalence queries by stochastic equivalence testing in Valiant's model [8].

Littlestone [6] shows that equivalence queries are essentially equivalent to mistakes in a prediction scheme.

A *full equivalence query* proposes a grammar G' with the same non-terminal set N as G and asks whether G' is fully equivalent to G. The reply is either *yes* or *no*. In the latter case, a *counter-example* is returned, that is, a pair (w, A) such that w is in $L_G(A)$ but not in $L_{G'}(A)$ or vice versa.

3 The learning algorithm

Let k be any non-negative integer. We assume that there is a k-bounded grammar $G = (T, N, P, S)$ to be learned, and that k, T, N, and S are known to the learning algorithm, but P, the set of productions, is unknown. The assumption that the non-terminal alphabet is known to the learning algorithm is similar to Shapiro's assumption that the names of all the relevant sub-procedures are known to his learning algorithm [7].

The learning algorithm has access to further information about G only via equivalence and non-terminal membership queries. The goal of the algorithm is a set P' of productions such that $G' = (T, N, P', S)$ is equivalent to G.

Theorem 1 *There is an algorithm that learns a grammar equivalent to any k-bounded context-free grammar G using equivalence and non-terminal membership queries that runs in time polynomial in the size of G and the length of the longest counter-example.*

Note that this generalizes the result in [1] for Chomsky normal-form grammars. Since a simple grammar contains only one non-terminal, in this case non-terminal membership queries reduce to membership queries, so we have the following.

Corollary 2 *There is an algorithm that learns a grammar equivalent to any k-bounded simple context-free grammar G using equivalence and membership queries that runs in time polynomial in the size of G and the length of the longest counter-example.*

Thus, in the terminology of [1], languages generated by k-bounded simple context-free grammars can be learned feasibly from a *minimally adequate teacher*. The proof of Theorem 1 is a description and analysis of the learning algorithm.

The Learning Algorithm

1. The set P' of productions is initialized to the empty set. Then the algorithm iterates the following loop.

2. An equivalence query is made, proposing $G' = (T, N, P', S)$. If the reply is *yes*, the algorithm outputs G' and halts. Otherwise, a counter-example w is returned, and there are two cases.

   (a) If w is in $L(G')$, then a parse-DAG is found (with respect to G') with root label S and yield w. The parse-DAG is then diagnosed to find a production, $A \rightarrow \alpha$, that is incorrect for G. This production is removed from P'.

(b) If w is not in $L(G')$, then the set $C(w)$ of all candidate productions is computed from w. Among these must be at least one production in P but not in P'; all of them are added to P'.

The algorithm relies on three sub-procedures: parsing, diagnosis, and the computation of candidate productions. These are described in the following three subsections; the fourth puts the pieces together.

3.1 Parsing

The parsing sub-procedure takes a terminal string w and the grammar G' and determines whether w is an element of $L(G')$. If so, a parse-DAG for G' with yield w and root node labelled S is also returned. It is assumed that G' is k-bounded.

A typical account [4] of parsing an arbitrary context-free grammar calls for the grammar to be converted to an equivalent grammar in Chomsky normal form by elimination of null and chain rules and introduction of new non-terminals. In order to get a parse-DAG with respect to the original grammar, we would have to describe how to reverse the effects of these transformations. Instead we choose to describe a simple but relatively inefficient strategy for parsing in the original grammar G', which is nonetheless sufficient for the proof of the main theorem.

Let

$$L = \{y : y \text{ is a substring of } w\}.$$

There are at most $(|w|+1)^2$ elements of L, and L is easily computed in time polynomial in $|w|$. Let

$$I = \{(y, A) : y \in L, A \in N, \text{ and } y \in L_{G'}(A)\}.$$

There are at most $|N|(|w|+1)^2$ elements of I. Clearly, $w \in L(G')$ if and only if (w, S) is in I. We compute I as follows.

Initially, for each production $A \to y$ in P' such that y is a substring of w, the algorithm puts the pair (y, A) in I. Then the following process is iterated until the first iteration in which no new elements are added to I.

For each production $A \to \alpha$ in P' that contains $m \geq 1$ occurrences of non-terminals, and for every m-tuple

$$\rho = \langle (y_1, A_1), \ldots, (y_m, A_m) \rangle$$

of elements of I such that ρ is compatible with α, if $y = \rho[\alpha]$ is a substring of w, then put the pair (y, A) in Q if it is not already there.

We omit the simple inductive proof that this procedure computes I.

A modification of this procedure records information for a parse-DAG. The nodes of the parse-DAG will be elements of L and I. For each element (y, A) added to I during the initialization, add an edge from $(y, A) \in I$ to $y \in L$. Suppose (y, A) is added to I because the algorithm finds the production

$$A \to x_0 A_1 x_1 \cdots A_m x_m$$

and the m-tuple of elements

$$\langle (y_1, A_1), \ldots, (y_m, A_m) \rangle$$

of I such that

$$y = x_0 y_1 x_1 \cdots y_m x_m$$

is a substring of w. Then for the element (y, A) the algorithm adds an ordered list of $2m+1$ edges from (y, A) to the following elements of L and I:

$$x_0, (y_1, A_1), x_1, \ldots, (y_m, A_m), x_m.$$

(The edges to those $x_i = \lambda$ may be omitted.)

At the end of the procedure, if (w, S) is in I, a parse-DAG is constructed as follows. Discard from $L \cup I$ all elements not reachable from (w, S) by a directed path of edges. Then the label of each node y in L is just y, and the label of each node (y, A) in I is A. The root node is (w, S). The yield of the internal node (y, A) in the resulting parse-DAG is y.

Lemma 3 *There is a non-decreasing polynomial $p_1(x, y)$ such that the time used by the parsing algorithm on inputs G' and w is bounded by $p_1(size(G'), |w|)$.*

Every iteration of the loop except the last adds at least one element to I, so there are at most $|N|(|w|+1)^2$ iterations of the loop. Each iteration of the loop considers at most $|P'|$ productions, and for each production with m occurrences of non-terminals in its right-hand side, considers at most all m-tuples of elements of I. Since G' is k-bounded, $m \leq k$. Hence the basic operation of applying a replacement to the right-hand side of a production and testing whether it is a substring of w is done no more than

$$|P'|(|N|(|w|+1)^2)^{k+1}$$

times in all. The time for parsing is bounded by a polynomial in the length of w and the size of G' (but exponential in k). This proves Lemma 3.

Lemma 4 *There is a non-decreasing polynomial $p_2(x, y)$ such that the parse-DAG returned by the parsing algorithm on inputs G' and w is of size bounded by $p_2(|N|, |w|)$.*

Clearly the parse-DAG has at most $|L|+|I|$ nodes, which is bounded by $(|N|+1)(|w|+1)^2$. Each node has a label of length bounded by $|w|+1$, and each node has at most $2k+1$ edges directed out of it. Thus the total size of the resulting parse-DAG is bounded by

$$(|N|+1)(|w|+1)^2(|w|+2k+3).$$

This proves Lemma 4.

3.2 Diagnosis

The diagnosis routine is essentially a special case of Shapiro's algorithm for diagnosing an incorrect output [7]. The input to the diagnosis routine is a correct parse-DAG D for G' such that y is the yield of D, A is the label of the root, and y is not in $L_G(A)$. That is, y is derivable from A in the grammar G', but not in the correct gramar G. The output of the diagnosis routine is a production in P' that is incorrect for G.

The diagnosis routine considers in turn each child of the root of D. If the child is labelled with non-terminal B and has yield x, then the diagnosis routine makes a non-terminal membership query with x and B. If the reply is *no*, then it calls itself recursively with the sub-DAG rooted at the child, and returns the resulting production. If the reply is *yes*, then it goes on to the next child of the root of D.

If all the queries are answered *yes*, then the diagnosis routine returns the production $A \rightarrow \alpha$, where α is the concatenation of the labels of the children of the root of D in left-to-right order.

Lemma 5 *When the diagnosis routine is given as input a parse-DAG for G' that has yield y and root label A such that $y \notin L_G(A)$, it returns a production in P' that is incorrect for G.*

Assuming the input conditions are met on the initial call, each recursive call preserves the input conditions. Also, each recursive call is with a proper sub-DAG of its input DAG, so the procedure must eventually terminate and return a production.

If the production $A \rightarrow \alpha$ is returned, then it is clearly in P', since the original DAG is a correct parse-DAG for G' (as is every sub-DAG rooted at an internal node). When the production is returned, the queries have witnessed that there is a replacement

$$\rho = \langle (y_1, A_1), \ldots, (y_m, A_m) \rangle$$

compatible with α such that $y = \rho[\alpha]$ is not in $L_G(A)$, that is, the production $A \rightarrow \alpha$ is incorrect for G. This proves Lemma 5.

Lemma 6 *There is a non-decreasing polynomial $p_3(x)$ such that the time required by the diagnosis routine on input parse-DAG D is bounded by $p_3(size(D))$.*

The number of queries made by the diagnosis routine may be k times the depth of the parse-DAG. It is clear that a straightforward implementation of the diagnosis routine runs in time polynomial in the size of the input parse-DAG. This proves Lemma 6.

3.3 Candidate productions

The input to the candidate productions routine is a string w such that w is in $L(G)$ but not in $L(G')$, and the output is a set $C(w)$ of productions such that at least one element of $C(w)$ is in P but not in P'.

The algorithm considers in turn every substring y of w. For every $m \leq k$, every factorization of y into $2m+1$ substrings,

$$y = x_0 y_1 x_1 y_2 x_2 \cdots y_m x_m,$$

and every $m+1$-tuple of non-terminals $A, A_1, \ldots, A_m$ from N, the production

$$A \rightarrow x_0 A_1 x_1 A_2 x_2 \cdots A_m x_m$$

is added to $C(w)$.

Lemma 7 *Let the candidate productions routine have input $w \in L(G) - L(G')$. Then $C(w)$ contains some production in P but not in P'.*

Since w is in $L(G)$, there is a parse-DAG D (with respect to G) with root node labelled S and yield w. We show that every production used in D is in $C(w)$.

Consider any sub-DAG D' of D rooted at an internal node with label A and yield y. Suppose the production used at the root of D' is

$$A \to x_0 A_1 x_1 A_2 x_2 \cdots A_m x_m.$$

There is a replacement

$$\rho = \langle (y_1, A_1), \ldots, (y_m, A_m) \rangle$$

such that

$$y = x_0 y_1 x_1 y_2 x_2 \cdots y_m x_m.$$

Since y is a substring of w and $m \leq k$, the production

$$A \to x_0 A_1 x_1 A_2 x_2 \cdots A_m x_m$$

will be generated from y using the above factorization and choices of non-terminals.

Thus every production used in D is in $C(w)$. If every production in $C(w) \cap P$ were in P', D would be a parse-DAG for the grammar G' witnessing $w \in L(G')$, a contradiction. Thus, $C(w)$ contains some production in $P - P'$, which completes the proof of Lemma 7.

Lemma 8 *There are non-decreasing polynomials $p_4(x, y)$ and $p_5(x, y)$ such that on input w the candidate productions routine runs in time bounded by $p_4(|N|, |w|)$ and produces an output set $C(w)$ with at most $p_5(|N|, |w|)$ elements. Moreover, every production in $C(w)$ has a right-hand side of length at most $|w| + k$ with at most k occurrences of non-terminals.*

For each $m \leq k$, there are no more than $(|w| + 1)^{2m}$ factorizations of the string w into $2m + 1$ substrings, and no more than $|N|^{m+1}$ choices of $m + 1$ non-terminals. Thus, the total number of productions placed in $C(w)$ is at most

$$1 + k(|w| + 1)^{2k} |N|^{k+1}.$$

Computing these productions takes time bounded by a polynomial in $|N|$ and $|w|$. The right-hand side of each production consists of a subsequence of the terminals in w and at most k non-terminals, for a total length bounded by $|w| + k$. This proves Lemma 8.

3.4 Correctness and analysis of the learning algorithm

Since P' is initially empty and is only augmented by productions output by the candidate productions routine, G' is k-bounded at all times, by Lemma 8. Clearly if the learning algorithm ever terminates, its output is a grammar G' equivalent to G, which shows the partial correctness of the learning algorithm.

We bound the number of iterations of the main loop (step 2) as follows.

Lemma 9 *There are at most $|P|$ iterations of the main loop with positive counter-examples. If M_p is the maximum length of any positive counter-example, then there are at most $|P|p_5(|N|, M_p)$ iterations of the main loop with negative counter-examples.*

By Lemma 7, each iteration of the loop with a positive counter-example must add to P' at least one production in $P - P'$. This production is correct for G and therefore cannot be removed from P', because the only elements removed from P' are incorrect for G, by Lemma 5. Hence there are at most $|P|$ iterations with a positive counter-example.

Thus there are at most $|P|$ positive counter-examples, say

$$w_1, w_2, \ldots, w_r,$$

where $r \leq |P|$. Let M_p be the maximum value of $|w_i|$ for $i = 1, 2, \ldots, r$.

The total number of productions added to P' is bounded by

$$|C(w_1)| + |C(w_2)| + \ldots + |C(w_r)|,$$

which by Lemma 8 is bounded by

$$p_5(|N|, |w_1|) + p_5(|N|, |w_2|) + \ldots + p_5(|N|, |w_r|).$$

Since $p_5(x, y)$ is non-decreasing we have a bound of

$$|P|p_5(|N|, M_p)$$

on the total number of productions ever added to P'. (This is also a bound on the cardinality of P'.)

Each iteration of the loop with a negative counter-example removes one production from P'. Hence this can happen at most as many times as there are productions ever added to P', which is bounded by

$$|P|p_5(|N|, M_p).$$

This proves Lemma 9.

Thus, in particular, the learning algorithm must terminate after at most

$$|P| + |P|p_5(|N|, M_p)$$

iterations of the main loop.

Let M_n be the maximum length of any negative counter-example, and let M be the maximum of M_p and M_n. M is the maximum length of any counter-example encountered before termination.

We bound the time used by the learning algorithm as follows.

Lemma 10 *The time required for the parsing routine at each iteration of the main loop of the learning algorithm is bounded by a polynomial in the size of G and the length of the longest counter-example.*

By Lemma 3 the time required for parsing each counter-example w is bounded by $p_1(size(G'), |w|)$. Clearly $|w| \leq M$, but we need to establish a bound on $size(G')$.

We have

$$size(G') = |T| + |N| + |P'| + L',$$

where L' is the sum of the lengths of the right-hand sides of the productions in P'. As shown above,

$$|P'| \leq |P|(1 + p_5(|N|, M_p)).$$

Each production in P' has a right-hand side of at most $M_p + k$ in length, by Lemma 8.

Hence,

$$size(G') \leq |T| + |N| + |P|(1 + |P|p_5(|N|, M_p))(1 + M_p + k).$$

So the size of G' can be bounded by a non-decreasing polynomial in the size of G and the length of the longest positive counter-example,

$$size(G') \leq p_6(size(G), M_p).$$

Thus at each iteration the time for parsing a counter-example is bounded by

$$p_1(p_6(size(G), M_p), M),$$

which proves Lemma 10.

Lemma 11 *The time required by the diagnosis routine in each iteration in which it is called is bounded by a polynomial in the size of G and the length of the longest counter-example.*

By Lemma 6, the time required by the diagnosis routine on input D is bounded by $p_3(size(D))$. By Lemma 4, the size of the parse-DAG D for a negative counter-example w is bounded by $p_2(|N|, |w|)$. Hence the time required by the diagnosis routine at each iteration in which it is called is bounded by $p_3(p_2(size(G), M))$, proving Lemma 11.

Lemma 12 *The time required by the candidate productions routine in each iteration in which it is called is bounded by a polynomial in the size of G and the length of the longest counter-example.*

By Lemma 8, the time required by the candidate productions routine with input w (a positive counter-example) is bounded by $p_4(|N|, |w|) \leq p_4(size(G), M)$, proving Lemma 12.

Putting the bounds from these three lemmas together with the bound of

$$|P| + |P|p_5(|N|, M_p)$$

on the number of iterations of the main loop of the learning algorithm, we conclude that the total time used by the learning algorithm is bounded by a polynomial in the size of G and the length of the longest counter-example, proving Theorem 1.

If full equivalence queries are substituted for equivalence queries in the learning algorithm, it will output a grammar fully equivalent to G, and the same running time bound applies.

4 Comparison with Shapiro's algorithm

Our algorithm is an (impractical) existence proof; Shapiro's [7] is intended to be a practical system. To what extent can we combine the strengths of the two algorithms?

Shapiro's algorithm learns k-bounded context-free grammars with the following property: for each non-terminal A, all the right-hand sides of productions with left-hand side A begin with distinct terminal symbols. This restriction guarantees that a standard transformation of a context-free grammar into a Prolog program will produce a program that parses deterministically in polynomial time. The languages generated by such grammars are deterministic, and therefore a proper subclass of the λ-free context-free languages [4].

Shapiro proves a polynomial bound on his algorithm, but the setting is somewhat different. His algorithm reads in "facts" from an external source, specifying whether a string is generated by a non-terminal or not. These facts are in some arbitrary order. His algorithm reads in a fact, updates the current program until it is consistent with all the known facts, and then reads in another fact.

Shapiro's result is that the time before reading the next fact is bounded by a polynomial in the sum of the lengths of the known facts. For comparison with the setting in this paper, we may assume that the next fact is generated by an equivalence (or full equivalence) query with the current hypothesis. However, even in this case, Shapiro's bound does not translate into a polynomial bound on the total running time of the algorithm until convergence, since there is no obvious bound on the number of counter-examples that will be required.

Shapiro's algorithm incorporates a number of features that would improve our learning algorithm in practice. One is that the answers to queries are saved, and not re-asked. Another is that productions found to be incorrect with respect to G are saved, and not allowed back into P'. Before performing an equivalence query, it is probably useful to check the current hypothesis against all the previous answers to queries, using a discrepancy to generate a counter-example. Shapiro has a method of selecting queries for diagnosis that bounds the number of queries used by the diagnosis routine by a logarithm of the number of nodes in the input parse-DAG.

These improvements would all be easy to incorporate into our learning algorithm. However, a major difference between the two algorithms is that ours adds ALL of the productions in $C(w)$ to P' in response to a positive counter-example, while Shapiro's algorithm just adds one carefully selected production from $C(w)$.

The reason for adding all of $C(w)$ is to guarantee at least one element of $P - P'$ is added to P', which in turn is the basis of the polynomial bound on the number of iterations of the main loop of our algorithm. The problem with adding all of $C(w)$ to P' is that many useless and redundant productions will be added, as well as many incorrect ones that must be removed using the diagnosis routine.

One possible middle ground is to follow Shapiro's approach to locate an "uncovered" pair (y, A), and then to maintain a set of "upper bounds" for the candidate productions for this pair in P'. (This is a subset of the candidate productions which when added to P' allows all the candidate productions to be derived.) This would preserve the bounding argument, but would be more selective in the choice of productions added to P'. This follows Knobe and Knobe's emphasis on trying productions in decreasing order of generality [5].

5 Remarks

A straightforward dovetailing argument shows that the union of two classes learnable in polynomial time from a minimally adequate teacher is learnable in polynomial time from a minimally adequate teacher. The k-bounded simple grammars (Corollary 2) and the determinisitic one-counter automata (Berman and Roos [3]) define incomparable subclasses of the context-free languages. The union of the two classes can be learned in polynomial time from a minimally adequate teacher (who has to answer equivalence queries posed in either representation.)

To improve parsing it is possible to treat the empty string specially. In the initialization we make non-terminal membership queries, and place a production $A \rightarrow \lambda$ in P' for every A such that $\lambda \in L_G(A)$. A small change to the candidate productions routine excludes productions that would cause λ to be derivable from any other non-terminal. Then we could get by with parse trees instead of parse-DAGs, and use a more standard approach to improve parsing time.

As indicated in the preceding section, the primary open problem is to find a more practical algorithm that still has a provable polynomial bound on its total running time.

References

[1] D. Angluin. *Learning regular sets from queries and counter-examples.* Technical Report, Yale University Computer Science Dept., TR-464, 1986. To appear in *Information and Computation.*

[2] D. Angluin. *Types of queries for concept learning.* Technical Report, Yale University Computer Science Dept., TR-479, 1986.

[3] P. Berman and R. Roos. Learning one-counter languages in polynomial time (extended abstract). 1987. Preprint, The Pennsylvania State University Dept. of Computer Science. To appear in IEEE FOCS 87.

[4] M. Harrison. *Introduction to Formal Language Theory.* Addison Wesley, 1978.

[5] B. Knobe and K. Knobe. A method for inferring context-free grammars. *Inform. Contr.*, 31:129–146, 1976.

[6] N. Littlestone. Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm (preliminary draft). University of California at Santa Cruz, May 1987. To appear in IEEE FOCS 87.

[7] E. Shapiro. *Algorithmic program debugging.* PhD thesis, Yale University Computer Science Dept., 1982. Published by MIT Press, 1983.

[8] L. G. Valiant. A theory of the learnable. *C. ACM*, 27:1134–1142, 1984.