Inferring Automata and its Application to Compositional Verification
WPEII : Critical Review Report

Arvind Easwaran
arvinde@cis.upenn.edu
University of Pennsylvania, Philadelphia

December 15, 2005

Abstract

Given an oracle which can answer membership and equivalence queries for an unknown regular language, we present algorithms for learning the regular set perfectly. For the case where the oracle cannot reset the black box automaton to a fixed initial state, we present efficient learning algorithms using homing sequences. Algorithms have been described for two different representations of the finite automaton. In the special case where the automaton is a permutation automaton, we present algorithms which can deduce the unknown regular set without the use of equivalence queries. We also describe a modified learning algorithm which can be used to learn context-free grammars from their structural descriptions. Application of these learning techniques for compositional verification of symbolic modules has also been discussed. We also describe improvements to the learning algorithm for context-free languages under certain restrictions on the result of equivalence queries.

1 Introduction

Consider a simple robot placed in an unfamiliar environment. The robot would be equipped with some sensors which provides the robot with information about its current state in the environment. At each state the robot will have the option of executing some simple actions. It may have no knowledge about the structure of the environment nor of the meaning of actions it is executing. The goal of the robot is to use its experience and observations to infer a model of the unknown environment. Once such a model has been inferred, the robot can perform more effectively in the environment.

In this report, we summarize general algorithmic solutions developed for the specific case when the unknown environment is a deterministic finite state automaton or a context-free grammar. The goal is to infer the unknown structure by observing the input-output behavior of the system. It has been shown in [2, 3, 4] that a combination of active experimentation (membership queries asking whether a particular string belongs to the unknown language) and passive observation (equivalence queries which conjectures an automaton and asks whether the conjecture is correct) is both necessary and sufficient to learn the unknown automaton. Angluin [2] and Gold [8, 7] show that it is NP-complete to find the smallest automaton consistent with a given sample of data (passive experimentation). Angluin [2] also

shows that learning by just actively experimenting is hard for a family of automaton. The difficulty here is in reaching certain hard to reach states.

In spite of these hardness results, Angluin in [4] described an efficient inference procedure for regular languages that is able to actively experiment with the unknown automaton and in addition passively receives a counterexample for each incorrect conjecture. This algorithm, exactly identifies the unknown structure in time polynomial in the size of the automaton and the length of the longest counterexample received. But a serious limitation of this procedure has been the need to reset the automaton prior to execution of any membership or equivalence query. Rivest and Schapire in [13] describe efficient algorithms which circumvent this problem and no longer require the teacher to reset the automaton for each query. These algorithms make use of homing sequences which every finite state automaton must possess. A homing sequence is a sequence of input symbols whose execution from a particular state of the automaton results in an output sequence which uniquely identifies the state reached. The authors have given learning algorithms for two different types of finite state automata representations. The paper [13] also improves the running time of Angluin's algorithm [4] by reducing the number of membership queries that must be performed (modifies the handling of counterexamples). Rivest and Schapire [13] have also described algorithms for inference of unknown permutation automata in which the algorithms never seek response to any equivalence queries.

Sakakibara in [14, 15] has described an adaptation of the Angluin's algorithm [4] for inference of context-free grammars and reversible context-free grammars. Since for a given context-free language there are infinitely many context-free grammars generating that language, it is difficult to infer the grammar without any structural information. Hence Sakakibara in [14] has described an inference procedure for computing an unknown context-free grammar from structural descriptions of the grammar. A set of structural descriptions for a grammar consists of all the derivation trees for that grammar with the non-terminal labels on the interior nodes removed. Alur et. al. in [1] have described a direct application of the learning algorithm for the symbolic compositional verification problem. In compositional verification, a global property is verified locally at each component by making assumptions about the other components in the system using assume-guarantee rules. If the actual components are a refinement of the assumptions made regarding the environment, then the verification of the assumption also implies verification of the actual components. In [1], the authors have used the learning algorithm proposed by Angluin [4] and modified by Rivest [13] to generate a correct assumption for the environment. Counterexamples generated using symbolic reachability techniques are then used to refine the unknown assumption automaton being inferred.

This report is organized as follows: Section 2 gives basic definitions for automata representation. Section 3 describes Angluin's $L*$ algorithm along with improvements made to it by Rivest et. al. Section 4 discusses the learning algorithms based on homing sequences and Section 5 details the learning algorithms for permutation automata. Section 6 adapts the $L*$ algorithm for learning context-free grammars and Section 7 describes application of the algorithms to compositional verification. Section 8 improves the learning algorithm for context-free grammars under certain restrictions on the result of equivalence queries.

2 Preliminary Definitions

We introduce two different representations of finite automata; Global State-Space based representation and Diversity based representation. In the global state-space representation the observer of the automaton moves from one state of the automaton to another observing the outputs generated at different states. On the other hand, in the diversity based representation, the observer is stationary only observing the output generated at the start state of the automaton. The outputs generated at the other states of the automaton are observed only when they are transferred by the environment to the start state.

2.1 Global State-Space Representation

Global State-Space Representation A finite automaton M is a tuple $M = \langle Q, \Sigma, \delta, q_0, \gamma \rangle$ where

- Q is a finite and non-empty set of states of the automaton
- Σ is the finite alphabet set also called the input symbols
- δ is the transition function of the automaton mapping $Q \times \Sigma \; to \; Q$
- $q_0 \in Q$ is the initial state or start state of the automaton
- γ is the output function of the automaton mapping Q to $\{0, 1\}$.

In this definition the output alphabet set is binary, but all results summarized in this report are applicable for any finite output alphabet set. A bijection can be easily established between a finite automaton defined using acceptance/rejection and one defined based on a binary output alphabet set as above (states with output symbol 1 can be regarded as accepting states and those with output 0 can be regarded as rejecting states). Let Σ^* refer to the set of all finitely long sequence of input symbols generated from the alphabet set Σ. Then we can extend the definition of the transition function δ in the usual way over the domain Σ^*: $\delta(q, \epsilon) = q$ where ϵ refers to the empty input sequence and $\delta(q, ua) = \delta(\delta(q, u), a)$ where $u \in \Sigma^*, q \in Q$ and $a \in \Sigma$. $\delta(q, ua)$ refers to the state reached in the automaton M by executing the input sequence ua starting from state q. For short, we will denote $\delta(q, u)$ by qu. Similarly, we refer to the sequence of outputs produced by an input sequence $u = a_1, \cdots, a_n \in \Sigma^*$ starting from a state $q \in Q$ as the output of u at q and is denoted by $q < u >$.

$$q < u > = < \gamma(q), \gamma(qa_1), \gamma(qa_1a_2), \cdots, \gamma(qu) >.$$

It is important to note that $\gamma(qu)$ refers to the output generated at the last state of M reached by executing the sequence u from state q whereas $q < u >$ is a $(|u| + 1)$ tuple referring to the sequence of outputs generated during execution of the input string u. Also, for $u \in \Sigma^*$ we define $Q < u >$ to be the set,

$$Q < u > = \{q < u > | q \in Q\}.$$

$Q < u >$ is the set of different output sequences that can be generated by the input string u when executed from all the states of the automaton. Now, we give some definitions which will be used in the later sections of the report.

Permutation Automata A finite automaton M is a permutation automaton if and only if $\forall a \in \Sigma, \delta(., a)$ is a permutation of Q. In other words, in a permutation automaton, for each state $q \in Q$ and each input symbol $a \in \Sigma$, there is exactly one transition labeled with a incident on q.

Distinguishing Sequence An input sequence $u \in \Sigma^*$ is said to distinguish two states q_1 and q_2 if and only if $q_1 < u > \neq q_2 < u >$. A distinguishing sequence of a finite automaton M is an input sequence $u \in \Sigma^*$ with the property that $q_1 = q_2$ whenever $q_1 < u >= q_2 < u >$.

Homing Sequence An input sequence h is a homing sequence if and only if $(\forall q_1, q_2 \in Q), q_1 < h >= q_2 < h > \Rightarrow q_1 h = q_2 h$. A homing sequence for a finite automaton is an input sequence h such that the state reached by executing h is uniquely determined by the output produced as a result of that execution. An important property of homing sequences is that every finite automaton has at least one homing sequence.

For any finite automaton, it is easy to see that every distinguishing sequence is also a homing sequence. In the special case of a permutation automaton, every homing sequence of the automaton is also a distinguishing sequence. If h is a homing sequence, then for two states q_1 and q_2 if $q_1 < h >= q_2 < h >$ then $q_1 h = q_2 h$ by definition. But since the automaton is a permutation automaton, we have that $q_1 = q_2$ which then implies that h is a distinguishing sequence. If $q_1 \neq q_2$ then the path from q_1 to $q_1 h$ using the sequence h and the path from q_2 to $q_2 h = q_1 h$ using the sequence h must meet at some state on or before $q_1 h$. But this implies that there exists a state in the automaton having two incident transitions labeled with the same input symbol. This is clearly a contradiction for permutation automaton and hence the states q_1 and q_2 must be the same.

The homing sequence defined above is a preset homing sequence. There also exists an adaptive homing sequence for finite automata which is like the preset homing sequence in that the output produced by executing the adaptive homing sequence can be used to determine the state reached. However, at each step, the input action to be executed in an adaptive sequence depends on the output observed upto that point.

Adaptive Homing Sequence An adaptive homing sequence u is a binary decision tree with the following properties:

- The root node is labeled ϵ.
- Every other node is labeled with one of the basic input symbols from Σ.
- Every transition in the tree is labeled with one of the output symbols ($'0'$ or $'1'$).
- Every node in the tree has at most one outgoing transition labeled with either output symbol.
- A sequence u is an adaptive homing sequence if and only if $q_1 < u >= q_2 < u >$ implies $q_1 u = q_2 u$ for all $q_1, q_2 \in Q$.

Depending on the output of the current state, appropriate transition in the adaptive sequence is traversed and the input action corresponding to the destination node is executed.

2.2 Diversity Based Representation

In many practical applications, the diversity based representation of finite automata is more compact and natural. It is based on the notion of input sequences as tests and equivalence of these tests based on the outputs generated by them. Two tests $t_1, t_2 \in \Sigma^*$ (input sequences) are equivalent (written as $t_1 \equiv t_2$) if and only if for every state $q \in Q$, $\gamma(qt_1) = \gamma(qt_2)$. $' \equiv'$ defines an equivalence relation on the set of all tests for the automaton. We denote the equivalence class for a test t as $[t]$ which consists of all the tests equivalent to t. The value of $[t]$ at a state q is $\gamma(qt)$. The diversity of the automaton ($D(M) = |\{[t] : t \in \Sigma^*\}|$) is the number of such equivalence classes. We now show that the bounds on the diversity of any finite automaton M with state space Q is given by $\log(|Q|) \leq D(M) \leq 2^{|Q|}$. Each test class can be represented by a combination of $'0'/'1'$ output values that the class generates for all the states. The upper bound is now trivial because $2^{|Q|}$ is the maximum number of possible combination of $'0'/'1'$ outputs for all states that can be generated by a finite automaton with state space Q. Since any test must generate one such combination the diversity is clearly less than or equal to $2^{|Q|}$. The lower bound is also trivial because the diversity of the representation must distinguish all the states of the finite automaton. Two states of M are equal if and only if every test has the same value in both the states. Since any state of M can be determined from the set of outputs generated by all the test classes, at least $\log(|Q|)$ classes are required to distinguish the states.

The diversity based representation for a finite automaton can be defined using an update graph where each node of the graph represents a test equivalence class. Thus the total number of nodes of the update graph is equal to the diversity $D(M)$. An edge labeled $a \in \Sigma$ is directed from a vertex $[t_1]$ to a vertex $[t_2]$ if and only if $t_1 \equiv at_2$. These edges are well defined because $t_1 \equiv t_2$ implies $at_1 \equiv at_2$ and hence by transitivity each vertex of the update graph has exactly one edge labeled with a particular input symbol incident on it. The observer in an update graph can only observe the output values generated at the state $[\epsilon]$. If the current state of the state-space based representation is q, then the output value at a state $[t]$ of the update graph will be equal to $\gamma(qt)$. Hence the value at $[\epsilon]$ observed by the observer will be equal to $\gamma(q\epsilon) = \gamma(q)$. When the automaton M makes a move on input $a \in \Sigma$ and reaches state $q_1 = \delta(q, a)$, the values of the test classes get updated accordingly. The new value at a state $[t]$ will be given by the old value of the output at state $[t']$ such that $t' \equiv at$. This is trivially obtained from the fact that $\gamma((qa)t) = \gamma(qat)$. Hence the update graph can be used to efficiently simulate the finite automaton it represents.

A homing sequence for the diversity based representation of finite automaton is any input sequence h which has the property that the observed output sequence $q < h >$ is sufficient to determine the values of all the test classes at the state qh reached on execution of h. In particular, a sequence h is a homing sequence, if for every test $t \in \Sigma^*$, there exists some prefix p of h such that $p \equiv ht$ i.e., $\forall q \in Q, \gamma(qp) = \gamma(qht)$.

Any update graph can be viewed more formally as a special kind of automaton called the simple assignment automaton.

Simple Assignment Automaton A simple assignment automaton SAA is a tuple of the form $\langle V, \Sigma, \Gamma, v_0, \omega \rangle$ where:

- V is a set of variables
- Σ is the set of input symbols

- Γ is the update function that maps $V \times \Sigma$ to V updating the variable values.
- $v_0 \in V$ is the output variable and
- ω is the initial value function mapping V to $\{0, 1\}$.

The state of the automaton is given by a valuation of all the variables in V. The start state of the automaton is given by the initial value function ω and the output of the machine at any state is denoted by the variable v_0. When a particular action $a \in \Sigma$ is executed, all the variables in the automaton are updated using the function Γ to generate the next state of the automaton. The new value of any variable $v \in V$ is given by the value of the variable $v' = \Gamma(v, a)$. Function Γ can be extended to Σ^* as follows: $\Gamma(v, \epsilon) = v$ and $\Gamma(v, au) = \Gamma((\Gamma(v, u)), a)$. It is worth noting that SAA processes the input symbols in reverse order thereby ensuring that the output generated on execution of an input sequence $u \in \Sigma^*$ from the initial state is given by $\omega(\Gamma(v_0, u))$.

Any update graph of a finite automaton can be represented as a simple assignment automaton as follows. The set of variables V can be mapped to the set of test equivalence classes $\{[t]|(t \in \Sigma^*)\}$. The update function can be defined using the Γ function as $\Gamma([t], a) = [at]$. The output variable v_0 is the test class $[\epsilon]$ because that is the value observed by any external observer. The initial value function of a test class $[t]$ is just the value of t in the initial state ($\gamma(q_0, t)$). Thus it can be seen that $\Gamma(v_0, t) = [t]$ and hence $\omega(\Gamma(v_0, t)) = \gamma(\delta(q_0, t))$ as required.

3 Learning Regular Sets using Minimally Adequate Teachers

In this section, we describe the first algorithm ($L*$) developed for learning a regular language from samples of data. The intuition for this algorithm was developed by Gold [7] and the algorithm has been formally described and analyzed by Angluin [4]. We describe the data structures and assumptions of the algorithm, describe the algorithm and analyze its time and space complexity. Let the unknown regular language being learned be U with the corresponding automaton M_U, the input alphabet set over which U is defined be Σ and the output symbol set be $\{0, 1\}$. U consists of all strings in Σ^* which outputs a $'1'$ on execution in M_U.

3.1 Minimally Adequate Teachers

A teacher in a learning algorithm is an oracle or black box which can respond to queries posed by the learner trying to learn the unknown regular language. For the algorithm described in this section, we assume that the teacher can answer two types of queries. In a membership query, the learner wants to determine the output (0 or 1) generated by the black box finite automaton in response to a particular input sequence u. These type of queries are referred to as "active experiments" because the learner is controlling the experiments completely. The second question that the learner can ask the teacher is validity of a conjecture of the unknown finite automaton also known as an equivalence query. If the conjecture is correct, then the teacher responds so and the algorithm can terminate outputting the correct machine. On the other hand, if the conjecture is incorrect, then the teacher will respond with a counterexample string u such that u is in the symmetric difference of the languages generated by the conjectured automaton and the actual automaton. Such equivalence queries are

referred to as "passive experiments" because the learner has no control over the counterexamples that are generated by the teacher. Teachers which can answer both these queries correctly are referred to as minimally adequate teachers. Since membership queries have only one correct answer, all minimally adequate teachers must give the same answer. But for equivalence queries, the teacher can respond with any one of the many counterexamples for a given conjecture. As will be seen later, since the running time of the algorithm depends polynomially on the size of the counterexample, a minimally adequate teacher generating the smallest possible counterexample for each conjecture will result in an efficient algorithm.

We assume that the minimally adequate teacher has information about the unknown regular language in the form of a deterministic finite state machine. We also assume that the teacher can always reset the current state of the automaton to the start state. This is a strong assumption which will be relaxed when we describe learning algorithms based on homing sequences. If the automaton representation has n states, then the teacher can answer any membership query in time polynomial in n and the length of the input sequence whose membership is being sought. Equivalence queries can similarly be answered by using a polynomial time algorithm for testing equivalence of two automata which also returns a counterexample if the automata are not equivalent.

3.2 Data Structures for the L* Algorithm

The $L*$ algorithm progressively classifies various input sequences based on the outputs they produce until using the current membership information it can correctly conjecture the black box automaton. To do this, the algorithm maintains an observation table whose rows are labeled with a prefix-closed set of input strings over $(S \bigcup S\Sigma)$ and columns are labeled with suffix-closed set of input strings E. Each entry in the table is given by a function T which maps $(S \bigcup S\Sigma) \times E$ to the output alphabet set $\{0, 1\}$ i.e., $T(s_i e_j) = 0/1$ iff $\gamma(q_0 s_i e_j) = 0/1$ in the unknown automaton, respectively. This observation table is denoted by (S, E, T). The algorithm starts with an observation table having one row and one column labeled with the empty string ϵ and augments it with strings as and when required. To make a conjecture, the algorithm uses the strings S of the table to represent the states of the conjectured finite automaton. The strings $(S\Sigma)$ are used to compute the transition function and the strings E represent the distinguishing experiments for the conjectured machine. In order for the algorithm to make a conjecture, the observation table must satisfy certain conditions which we define below.

closed, consistent observation table Define $row(s)$ for $s \in (S \bigcup S\Sigma)$ to be the row vector representing the values of the function T for the string s over all strings in E. An observation table (S, E, T) is closed if for every $s \in S\Sigma$ there exists a $s' \in S$ such that $row(s) = row(s')$. The observation table is said to be consistent if for every $s, s' \in S$ such that $row(s) = row(s')$, $\forall a \in \Sigma, row(sa) = row(s'a)$.

If the observation table (S, E, T) is closed and consistent then we can define a finite automaton $M(S, E, T) = \langle Q, \Sigma, \delta, q_0, \gamma \rangle$ which will be consistent with the observation table (S, E, T) as follows:

- $Q = \{row(s) | s \in S\}$
- $q_0 = row(\epsilon)$

- $\delta(row(s), a) = row(sa), \forall s \in S, \forall a \in \Sigma$ and
- $\gamma(row(s)) = T(s), \forall s \in S$

Now, δ is a well defined function (observation table (S, E, T) is closed), $row(\epsilon)$ exists in the table (S is prefix-closed) and output function γ is well defined (E is suffix-closed and $T(s)$ is defined for each $s \in S$). Hence the mapping of the observation table (S, E, T) to the finite automaton M described above is well defined.

Theorem 3.1 (Minimality of $M(S, E, T)$) *If (S, E, T) is a closed and consistent observation table, then $M(S, E, T)$ as defined above is consistent with (S, E, T) and any other finite automaton M' consistent with (S, E, T) has at least as many states as M.*

We prove this theorem using the following lemmas.

Lemma 3.2 *If (S, E, T) is closed and consistent then $\delta(q_0, s) = row(s)$ in $M(S, E, T)$ for all $s \in (S \bigcup S\Sigma)$.*

Proof We can prove this lemma by induction on the length of the string s. If $s = \epsilon$ then by definition of $M(S, E, T)$ we have $\delta(q_0, \epsilon) = q_0 = row(\epsilon)$. Let the lemma be true for all strings of length less than or equal to k. Let $s = s'a$ be a string in $(S \bigcup S\Sigma)$ of length $k + 1$ where $a \in \Sigma$. Since S is prefix-closed, s' must belong to S and hence by the induction hypothesis we have $\delta(q_0, s') = row(s')$. Then,

$\delta(q_0, s) = \delta(\delta(q_0, s'), a) = \delta(row(s'), a) = row(s'.a) = row(s)$ which completes the proof. □

Lemma 3.3 (Consistency of $M(S, E, T)$) *If (S, E, T) is closed and consistent, then $M(S, E, T)$ is consistent with (S, E, T) i.e., for all $s \in (S \bigcup S\Sigma)$ and $e \in E, \gamma(q_0 se) = T(se)$.*

Proof We prove this lemma by induction on the length of the string $e \in E$. If $e = \epsilon$ and $s \in S$ then $\gamma(q_0 s) = T(s)$ by definition. If $s \in S\Sigma$ then since (S, E, T) is closed, there exists $s' \in S$ such that $row(s') = row(s)$. Now, $\gamma(q_0 s') = T(s')$ by definition and since $row(s) = row(s')$ we have $\gamma(q_0 s') = T(s)$. But from the definition of the transition function δ for $M(S, E, T)$ and using Lemma 3.2 we have,

$q_0 s' = \delta(q_0, s') = row(s') = row(s) = \delta(q_0, s) = q_0 s$

This proves the lemma for $e = \epsilon$. Let the lemma be true for all strings in E of length less than or equal to k and let $e = ae'$ be a string of length $k + 1$ in E. Since E is suffix-closed $e' \in E$ and let $s \in (S \bigcup S\Sigma)$. Since (S, E, T) is closed and consistent, there exists $s' \in S$ such that $row(s) = row(s')$. Now,

$\delta(q_0, s) = row(s) = row(s') = \delta(q_0, s')$

Hence, $\delta(q_0, se) = \delta(q_0, sae') = \delta(\delta(q_0, s), ae')$

$= \delta(\delta(q_0, s'), ae') = \delta(q_0, s'ae')$

Thus, $\gamma(q_0se) = \gamma(q_0s'ae') = T(s'ae')$ by the induction hypothesis on e'. But since $row(s) = row(s')$, $T(s'ae') = T(sae') = T(se)$ which proves the lemma. □

Lemma 3.4 *If (S, E, T) is closed and consistent and if $M(S, E, T)$ has n states, then any other finite automaton $M' = \langle Q', \Sigma, \delta', q_0', \gamma' \rangle$ consistent with T and having n or fewer states is isomorphic to $M(S, E, T)$.*

Proof Refer to [4] for the proof of this lemma. □

Combining the results from Lemmas 3.2, 3.3 and 3.4, will then imply validity of Theorem 3.1.

3.3 Hardness of Learning Regular Sets

In this section, we show that learning a regular language from a given sample of experiments and their results ("passive experiments") is an NP-complete problem [8, 6]. Let (S, E, T) be some observation table as defined in Section 3.2 and D be a sample of data such that $D = \{\langle s_1, o_1 \rangle, \cdots, \langle s_n, o_n \rangle\}$ where for $1 \leq i \leq n, s_i$ is an experiment string and o_i is the $'0' -' 1'$ output generated by the unknown automaton as a result of execution of string s_i. Holes in the observation table are entries whose T values are unknown (not given by the sample data D). Two holes in an observation table are said to be tied iff their representative strings are equivalent ($s_ie_j = s_ke_l \Rightarrow T(s_ie_j) = T(s_ke_l)$). The hole filling problem $P_{hole}((S, E, T), D)$ is then defined as the problem of finding a suitable $'0' -' 1'$ assignment to the holes in the observation table (S, E, T) such that all the tied holes are assigned equal values and the resulting table (S, E, T') is closed. We assume that all the states represented by the set S are distinct and hence the table is trivially consistent. If the problem $P_{hole}(.,.)$ has a solution then using Theorem 3.1 given in Section 3.2 a finite automaton consistent with T can be constructed. Since the automaton is consistent with T, it will be consistent with the given sample of data D. We show that the $P_{hole}(.,.)$ problem restricted to certain types of observation tables is NP-complete which then implies that the general problem is also hard. We will be reducing the satisfiability problem of conjunctive normal form (CNF) boolean formulas to the hole filling problem. A boolean formula F' is in conjunctive normal form iff,

$F' = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ where each C_i is a clause of the form, $C_i = c_{i1} \vee c_{i2} \cdots \vee c_{in_i}$

where the c_{ij} are literals z_k or $\neg z_k$ where z_k are boolean variables. We first reduce this general satisfiability problem for F' to the satisfiability problem of an equivalent boolean formula F where F is in CNF and in each clause either all the literals are complemented or none are. This reduction can be performed by trivially replacing each clause,

$C_i = z_{k_1} \vee \cdots \vee z_{k_a} \vee \neg z_{k_{a+1}} \cdots \vee \neg z_{k_b}$

in F' with two clauses,

$C_{2i-1} = z_i' \vee z_{k_1} \cdots \vee z_{k_a}$ and $C_{2i} = \neg z_i' \vee \neg z_{k_{a+1}} \cdots \vee \neg z_{k_b}$

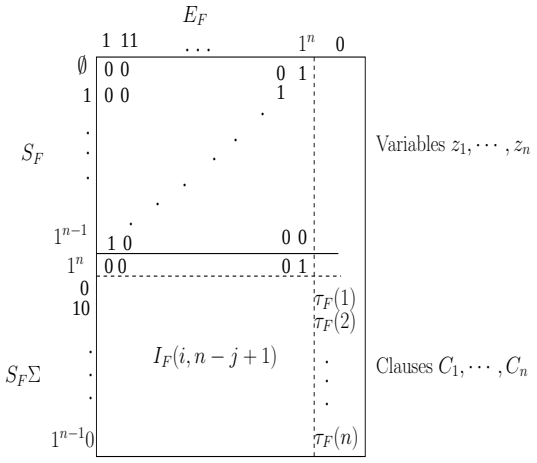

Figure 1: Reducing Satisfiability to the Hole Filling Problem

in F where $z'_1, \cdots z'_n$ are all new boolean variables. It is easy to show that F' is satisfiable iff F is satisfiable. We also assume that F has the same number n of variables and clauses. This can be easily done by adding an arbitrary number of new clauses having all new boolean variables. For any such F we define two characteristic functions,

$I_F(i, j) =$ "$hole$" if $z_j \in C_i$ or $\neg z_j \in C_i$ and 0 otherwise.

$\tau_F(i) = 1$ if C_i contains complemented variables and 0 otherwise.

We now prove that the hole filling problem $P_{hole}(.,.)$ is NP-complete by reducing the satisfiability problem for F to a corresponding hole filling problem.

Theorem 3.5 *For the observation table (S, E, T), let $S = \{\emptyset, 1, \cdots, 1^{n-1}\}$ for some n, $|\Sigma| = 2$ and S be pairwise distinct with respect to the sample data D. Then the hole filling problem $P_{hole}((S, E, T), D)$ is NP-complete.*

Proof For any boolean formula F of the above form, the sample data D_F is given by the observation table $M_F = (S_F, E_F, T_F)$ described in Fig. 1. We now show that F is satisfiable iff the hole filling problem $P_{hole}((S_F, E_F, T_F), D_F)$ has a solution. Firstly, none of the holes in M_F are tied. The first row in $S_F\Sigma$ can certainly be made equivalent to the first row in S_F as shown. In order to make the C_i-th row in $S_F\Sigma$ equivalent to some z_j-th row in S_F, one must assign $\tau_F(i)$ to the rightmost position of the z_j-th row. This implies assigning the value $\tau_F(i)$ to the variable z_j in the formula F. However not every z_j can be assigned $\tau_F(i)$ to satisfy C_i. This is controlled by the function $I_F(i, n-j+1)$ in the $j-$th entry of row C_i. If this entry is 0 then row C_i can never be made equal to row z_j. Hence row C_i can be made equal to row z_j iff $I_F(i, n-j+1)$ has a hole in that position iff in F clause C_i can be satisfied by setting $z_j = \tau_F(i)$. □

3.4 The $L*$ Algorithm

The $L*$ learning algorithm which in every iteration computes a closed and consistent observation table and then makes a conjecture for the finite automaton is given in Algorithm 1.

Algorithm 1 $L*$ Algorithm

```
1: Initialize S and E to the empty string ϵ
2: Construct initial observation table using membership queries for each a ∈ Σ
3: while Teacher does not respond yes to an equivalence query do
4:    while (S, E, T) is not closed or consistent do
5:       if (S, E, T) is not closed then
6:          Determine a string s ∈ SΣ such that ∀s' ∈ S, row(s) ≠ row(s').
7:          Add s to the set S and compute row(sa) for each a ∈ Σ.
8:       end if
9:       if (S, E, T) is not consistent then
10:         Find s1, s2 ∈ S and a ∈ Σ such that row(s1) = row(s2) but row(s1a) ≠ row(s2a).
11:         Determine string e ∈ E such that T(s1ae) ≠ T(s2ae).
12:         Add the string ae to E and update the observation table.
13:      end if
14:   end while
15:   Since (S, E, T) is closed and consistent, generate the conjecture automaton M(S, E, T).
16:   if Teacher responds with a counterexample t then
17:      Add t and all prefixes of t to the set S.
18:      Compute the observation table by extending T to (S ⋃ SΣ)E using membership
         queries.
19:   end if
20: end while
```

The algorithm first computes the initial observation table for $S = E = \epsilon$. It then iteratively appends strings to E and S until the table becomes closed and consistent. If the table is not closed, then there exists some string $s \in S\Sigma$ such that $row(s) \neq row(s')$ for any string $s' \in S$. The algorithm then adds string s to the set S thereby generating a new state for the conjecture automaton. If the table is not consistent then for some $s_1, s_2 \in S, a \in \Sigma$ and $e \in E$ such that $row(s_1) = row(s_2)$, we have $T(s_1ae) \neq T(s_2ae)$ and hence $row(s_1a) \neq row(s_2a)$. This would result in the transition function of the conjecture being inconsistent. Hence the algorithm appends the string ae to the set E which will then distinguish the two states s_1 and s_2 which were indistinguishable earlier. Finally, when the table is closed and consistent, the algorithm generates the conjecture automaton $M(S, E, T)$ as described in Section 3.2. If the teacher returns a counterexample t, the algorithm adds t and all its prefixes to the set S in order that the next conjecture contains information about string t.

Correctness of the $L*$ algorithm is trivial because the algorithm terminates if and only if the teacher returns yes which happens only when the algorithm generates a correct conjecture. Now we show that the algorithm eventually terminates. It is trivially true that any finite automaton consistent with an observation table having n distinct rows, must have at least n states. Let n be the number of states in the minimal automaton recognizing the unknown language U. We now show that the number of distinct rows in the observation table (S, E, T) constructed by the algorithm grows monotonically towards n. When the table is not

closed, a string $sa \in S\Sigma$ is added to the set S and since $\forall s' \in S, row(sa) \neq row(s')$ the number of distinct rows in the observation table increases by at least one. Similarly, if the table is not consistent, a string ae is added to the set E and since this string distinguishes two strings $s_1, s_2 \in S$ which were earlier indistinguishable, the number of distinct rows in S again increases by at least one. Hence the total number of times either operation is repeated over the entire run of the algorithm cannot exceed $n-1$. Thus the algorithm will eventually find the table to be closed and consistent and make a conjecture. Now we show that the number of distinct conjectures that the algorithm makes cannot exceed $n-1$ which will then prove that the algorithm terminates. If the algorithm makes an incorrect conjecture $M(S,E,T)$, the teacher returns a counterexample t which is a string in the symmetric difference of U and the conjectured language. Since $M(S,E,T)$ is consistent with T and not equivalent to M_U, M_U must have at least one more state than $M(S,E,T)$. The algorithm now adds t and all its prefixes to the set S and computes a new consistent, closed table (S',E',T') and then generates a new conjecture $M'(S',E',T')$. $M'(S',E',T')$ is consistent with T (T' extends T) and classifies t similar to M_U which was not true for $M(S,E,T)$. Hence $M'(S',E',T')$ has at least one more state than the $M(S,E,T)$ which implies that the algorithm will make at most $n-1$ incorrect conjectures before it guesses the right automaton M_U.

Now we analyze the running time of the algorithm and show that it is polynomial in the number of states n of M_U and the size m of the longest counterexample returned by the teacher. Let k be the size of the input alphabet Σ. Since a string is added to E every time the table is found to be not consistent which occurs at most $n-1$ times, the total number of strings in E cannot exceed n (it initially contains ϵ). Every time the table is not closed a string is added to S and since this can occur at most $n-1$ times, the number of such strings cannot exceed n. For each counterexample returned, the set S gets appended with at most m strings and since this can occur at most $n-1$ times, the total number of such strings added to the set S cannot exceed $m(n-1)$. Also, the maximum length of any string in E cannot exceed $n-1$ because it increases by one every time the table is not consistent. Similarly, the maximum length of any string in S cannot exceed $m+n-1$ because any counterexample adds a string of length at most m to S and whenever the table is not closed, it increases the length of the strings in S by at most one which occur at most $n-1$ times. Thus the maximum size of (S,E,T) cannot exceed $(k+1)(n+m(n-1))n = O(mn^2)$ and the maximum length of any string in $(S \bigcup S\Sigma)E$ cannot exceed $(m+n-1+n) = O(m+n)$. Checking whether the table is closed or consistent can be done in time polynomial in the size of the table which is a polynomial in n and m and this is done at most $n-1$ times. The conjecture automaton can be constructed in time polynomial in the size of table and each membership query can be computed in time $O(m+n)$ and there are at most $O(mn)$ such queries. Since the number of states of the correct automaton M_U is n, for any incorrect conjecture there always exists a counterexample of length less than or equal to n. Thus the entire algorithm can be executed in time and space polynomial in n if the minimally adequate teacher can always return a counterexample of length at most n.

3.5 Improved $L*$ Algorithm

We now describe an improvement given by Rivest et. al. [13] to the $'L*'$ algorithm that reduces the worst case number of membership queries generated. Entries of the observation table (S,E,T) are filled using membership queries posed to the teacher. The number of membership queries generated is equal to the cardinality $|(S \bigcup S\Sigma)E|$ of the table (S,E,T).

For the $L*$ algorithm $|S|$ was bounded by $O(mn)$ and E was bounded by $O(n)$. We now give an algorithm which bounds the size of the set S to $O(n)$ thereby reducing the number of membership queries. In order to achieve this bound, the algorithm will make additional $n \log m$ membership queries thereby giving a bound of $O(n^2 + n \log m)$ on the total number of membership queries.

The modified algorithm maintains the invariant that $s_1, s_2 \in S$ such that $s_1 \neq s_2 \Rightarrow q_0 s_1 \neq q_0 s_2$ and hence $|S| \leq n$ all the time. Since $\forall s_i, s_j \in S, row(s_i) \neq row(s_j)$, the table can never be inconsistent. Thus the algorithm starts with the sets $S = E = \epsilon$ and builds a closed observation table. Adding a row $s \in S\Sigma$ to S in order make the table closed, ensures that the rows in S are still distinct because by definition $row(s) \neq row(s'), \forall s' \in S$. The algorithm now generates a conjecture automaton using the closed table and if the conjecture is correct then it outputs the machine and halts. If the conjecture is incorrect, the algorithm receives a counterexample from the teacher which it uses to update the observation table.

Let the conjecture made by the learner be incorrect resulting in the teacher generating a counterexample $z \in \Sigma^*$. We now show that with an additional $\log |z|$ membership queries, only one string e will be added to the set E (instead of adding all prefixes of z to S as in the original $L*$ algorithm) such that the added string will distinguish two states $s_1 \in S$ and $s_2 \in S\Sigma$ which were as yet indistinguishable. This will result in a new string s_2 being added to the set S in order to make the table closed thereby increasing the size of S. Hence even in the modified algorithm, the number of equivalence queries is bounded by $n - 1$.

We now demonstrate a procedure to compute the appropriate string e from the counterexample z for the conjectured machine $M' = \langle Q', \Sigma, \delta', q_0, \gamma' \rangle$. For $0 \leq i \leq |z|$, let $z = p_i r_i$ such that $|p_i| = i$. Since z is a counterexample, there exists a longest suffix of z which when executed from two equivalent states $s_1 \in S$ and $s_2 \in S\Sigma$ in the conjectured machine give different outputs. This procedure aims to compute this suffix by asking membership queries on every suffix of z to the teacher. We execute string p_i on the conjectured machine and then seek a membership query for $s_i r_i$ where $s_i = \delta'(q_0, p_i)$. If $\gamma(q_0 z) = 0$ then since z is a counterexample, $\gamma'(q_0 z) = 1$. Now, let $\alpha_i = \gamma(q_0 s_i r_i)$ which is the output generated by processing the first i symbols in z on the conjecture and the remaining symbols on the correct automaton. Then trivially, $\alpha_0 = 0$ and $\alpha_{|z|} = 1$. This then implies that there exists some i such that $\alpha_i \neq \alpha_{i+1}$ which can be determined by a binary search in the range $1 \leq i \leq |z|$. Thus using $\log |z|$ membership queries we can compute the index i such that $\alpha_i \neq \alpha_{i+1}$. From the definition of α_i it is then clear that r_{i+1} is the experiment that must be added to E. Let $a \in \Sigma$ be the first symbol in r_i. Then $\gamma(q_0 s_i r_i) = \gamma(q_0 s_i a r_{i+1}) \neq \gamma(q_0 s_{i+1} r_{i+1})$. By definition of the observation table, $s_{i+1} = \delta'(q_0, p_i a) = \delta'(s_i, a)$ and hence $row(s_{i+1}) = row(s_i a)$. But the string r_{i+1} clearly distinguishes these two states and hence addition of r_{i+1} to the set E will result a new state $s_i a$ being added to the set S. For each of the $n - 1$ equivalence queries $\log m$ membership queries are required to compute the longest suffix to be added to E. In addition each of the entries in the table will require $O(n^2)$ membership queries. Thus the total number of membership queries required is $O(n^2 + n \log m)$. The length of the string for each membership query is at most $O(m + n)$. This modification to the original $L*$ algorithm leads to a set E which is no longer suffix-closed. We now modify the proof of Lemma 3.3 and show that the lemma holds even if E is not suffix closed. This will then prove the correctness of the modified algorithm using Theorem 3.1 given in Section 3.

Lemma 3.6 *If (S, E, T) is closed and consistent, then $M(S, E, T)$ is consistent with (S, E, T) i.e., for all $s \in (S \bigcup S\Sigma)$ and $e \in E, \gamma(q_0 se) = T(se)$.*

Proof Let $e = a_1, \cdots, a_m$ where $a_i \in \Sigma$. Now using definition of δ and extending it to Σ^* we get,

$\delta(q_0, se) = \delta(q_0, sa_1 \cdots a_m)$

$= \delta(\delta(q_0, s), a_1 \cdots a_m) = \delta(row(s), a_1 \cdots a_m)$ from Lemma 3.2

$= \delta(\delta(row(s), a_1), a_2, \cdots, a_m) = \delta(row(sa_1), a_2 \cdots a_m) = \delta(row(s'), a_2 \cdots a_m)$ since (S, E, T) is closed

$= \delta(row(s''), a_m)$ by repeating the same argument as above for some $s'' \in S$

$= row(s''a_m) = row(s_f)$ where $s_f \in S$

$= \delta(q_0, s_f)$ from Lemma 3.2 in Section 3.

Now, $[\delta(q_0, se) = \delta(q_0, s_f)] \Rightarrow [q_0 se = q_0 s_f] \Rightarrow [\gamma(q_0 se) = \gamma(q_0 s_f)] \Rightarrow [T(s_f) = T(se)]$

Since $s_f \in S$ and by definition $row(s_f) \in F$ iff $T(s_f) = 1$ we get that

$\delta(q_0, se) = row(s_f) \in F$ iff $T(s_f) = 1$ iff $T(se) = 1$. □

4 Learning Regular Sets using Homing Sequences

Angluin's $L*$ algorithm necessitates the minimally adequate teacher to reset itself to the start state before answering every query. In this section we describe a modified algorithm given by Rivest and Schapire [13, 12, 16, 9] which uses homing sequences to learn an automaton even in the absence of a means to reset the automaton. Specifically, we describe algorithms which with probability at least $1 - \delta$ output the correct conjecture in time polynomial in the size of the automaton, the size of the longest counterexample returned by the teacher and $\log(1/\delta)$ using a minimally adequate teacher without reset. We describe efficient algorithms for the global state-space representation as well as the diversity based representation. In the case of permutation automaton, we give a probabilistic algorithm which does not require the teacher to provide any counterexamples. In this section we assume that the unknown automaton being learnt is a strongly connected automaton (every state is reachable by some string from every other state). This assumption is essential in the absence of a reset, because the experiment can get stuck in one of the strongly connected components very easily and never escape.

4.1 Algorithms for Global State-Space based Representation

We now describe the algorithm given in [13] that infers a regular language represented by a global state-space representation using a minimally adequate teacher without reset. This algorithm makes use of homing sequences defined in Section 3 for the global state-space representation. We first describe an algorithm which assumes the knowledge of this homing sequence and learns the automaton using the sequence. Later we describe a modification

to the algorithm which will not only infer the unknown regular set but also simultaneously learn the homing sequence.

Given a finite state automaton, a simple algorithm for computing a correct homing sequence is given in algorithm 2. At each step, the algorithm determines a string x which distinguishes two states that were not distinguishable by the homing sequence constructed so far. It then appends this string x to the current homing sequence which will then distinguish the two as yet indistinguishable states. If the automaton has n states, then in each iteration of the loop the resulting sequence distinguishes at least one more state. Hence the main loop is executed at most $n-1$ times. Further, in each execution of the loop since the length of the string x appended to the sequence is at most $n-1$ (for an automaton with state space size n, there exists a string of length less than or equal to $n-1$ that distinguishes any given two states). Hence the total length of the homing sequence constructed by this algorithm is $O((n-1)^2)$.

Algorithm 2 Homing Sequence Generation for State-Space Representation

1: $h = \epsilon$.
2: **while** $\exists q_1, q_2 \in Q, q_1 < h >= q_2 < h >$ and $q_1 h \neq q_2 h$ **do**
3: Find a $x \in \Sigma^*$ that distinguishes $q_1 h$ and $q_2 h$.
4: Set $h = hx$
5: **end while**

4.1.1 Global State-Space based Learning Algorithm (known homing sequence)

The main idea of this algorithm is to replace the implicit reset in the $L*$ algorithm with the homing sequence. The homing sequence works like a reset in the sense that on execution of the homing sequence, the output generated by the sequence uniquely identifies the destination state reached. The modified algorithm using homing sequence is described in Algorithm 3. Given a black box automaton M and a perfect homing sequence h for M the algorithm outputs a perfect conjecture for M.

Algorithm 3 Learning Regular Sets using State-Space based Homing Sequences

1: **while** true **do**
2: Execute h producing output σ.
3: If L^*_σ does not already exist then create it as a new copy of $L*$ (represents a new virtual teacher whose initial state is represented by the output σ).
4: Simulate the next query using L^*_σ.
5: **if** L^*_σ queries membership of input sequence $u \in \Sigma^*$ **then**
6: Execute u and provide L^*_σ with the output of the final state reached.
7: **end if**
8: **if** L^*_σ makes an equivalence query **then**
9: If the conjectured model is correct, stop and return the model.
10: Else supply the returned counterexample to L^*_σ.
11: **end if**
12: **end while**

When we execute h from some state q in the unknown automaton and get an output σ,

then we know that the destination state reached qh can be represented by the string σ. But since re-execution of the homing sequence does not guarantee that the output generated will be σ, there is no way to reset the automaton to this state repeatedly. Instead, the algorithm described in Algorithm 3 maintains separate observation tables for each output observed. Hence it simulates different copies of the observation table (one for each output of the homing sequence). Since in each iteration, at least one of the copies must make progress, after $n(N_M + N_E)$ iterations the conjectured automaton must be correct. Here N_M is the number of membership queries and N_E is the number of incorrect conjectures made by a single copy of the observation table before it can conjecture the correct automaton. From Section 3 $N_M = O(kn^2 + n \log m)$ and $N_E = n - 1$. Since there are at most n copies of the observation table (n states in the unknown automaton will result in n distinct outputs for the homing sequence), after $n(N_M + N_E)$ iterations at least one of the observation tables must represent a correct conjecture of the unknown automaton.

4.1.2 Global State-Space based Learning Algorithm (unknown homing sequence)

In this section we describe the algorithm given in [13] that combines inference of a correct homing sequence along with the inference of the unknown automaton. At every stage, we assume that the known sequence h is a correct homing sequence. We use the algorithm described in Section 4.1.1 to construct multiple copies of the observation table. If evidence arises that h is not a correct homing sequence, then we will append an appropriate string to h and re-execute the entire algorithm assuming that the new string is the homing sequence. If h is not a correct homing sequence then there exists two states $q_1, q_2 \in Q$ in the unknown automaton such that $q_1 < h >= q_2 < h >= \sigma$ but $q_1 h \neq q_2 h$. Now as part of the simulation of L^*_σ, if we execute some string x once from $q_1 h$ and then from $q_2 h$, output generated on the two occasions will be different. Thus x then represents a string that distinguishes the two states $q_1 h$ and $q_2 h$. Changing h to hx will then help distinguish the states $q_1 h$ and $q_2 h$. At this point the algorithm discards all existing copies of the observation table and starts afresh except with the modified homing sequence hx. Since h can be extended in this fashion only $n - 1$ times, the entire process is slower than the earlier algorithm by a factor of $O(n)$. The algorithm initializes h to ϵ and is given in Algorithm 4. Given a black box automaton M and the number of states n in M, the algorithm outputs a perfect conjecture for M.

Let $(S_\sigma, E_\sigma, T_\sigma)$ denote the observation table for L^*_σ. If $L^*{}_\sigma$ makes more than $N_M + N_E$ queries, then the number of distinct rows in the observation table will go beyond n. This happens only if the sequence h is not a correct homing sequence. In order to find a suitable string x to be appended to h, we have to find an inconsistency in the observation table. This can be done using a probabilistic technique shown in the algorithm. Let on execution of the sequence h from a state q we generate the output σ. Since there are $n + 1$ distinct rows in the table by the pigeon-hole principle there is at least one pair of rows s_i and s_j such that $qhs_i = qhs_j$. But since $row(s_i) \neq row(s_j)$, there is some $e \in E_\sigma$ for which $T_\sigma(s_i e) \neq T_\sigma(s_j e)$ but $\gamma(qhs_i e) = \gamma(qhs_j e)$. Hence an inconsistency in the table can be detected by executing the strings $s_i e$ and $s_j e$. The probability that a randomly chosen pair of states will possess the desired inconsistency is $1/\binom{n+1}{2}$. The chance of choosing the correct experiment to run among $s_i e$ and $s_j e$ is $1/2$. Hence the probability of finding an inconsistency in one iteration is at least $1/n(n+1)$. Repeating this process $n(n+1)ln(1/\delta)$ times gives a probability of at least $(1 - \delta)$ of finding an inconsistency. h is extended at most $n - 1$ times and each time at most n copies of $L*$ are ever created. Hence the total number of counterexamples required

Algorithm 4 Learning Algorithm with Unknown State-Space based Homing Sequence

1: $h = \epsilon$.
2: **while** a correct conjecture is not made **do**
3: Execute h producing output σ.
4: If L^*_σ does not already exist then create it as a new copy of $L*$ (represents a new virtual teacher whose initial state is represented by the output σ).
5: **if** $|\{row(s) : s \in S_\sigma\}| \leq n$ **then**
6: Simulate the next query on L^*_σ as in Algorithm 3.
7: Check for inconsistency (A string x that distinguishes two as yet indistinguishable states).
8: **else**
9: Let $\{s_1, \cdots s_{n+1} \subset S_\sigma\}$ be such that $\forall i, j, row(s_i) \neq row(s_j)$.
10: Choose a pair s_i, s_j randomly such that $\exists e \in E_\sigma, T(s_i e) \neq T(s_j e)$.
11: Execute one of $s_i e$ or $s_j e$ with equal probability and check for inconsistency.
12: **end if**
13: **if** inconsistency found executing some string x **then**
14: Discard all existing copies of L^* and update h to hx.
15: **end if**
16: **end while**

is at most $n(n-1)N_E$. Further, since each extension of h has length at most $m+n$, the total length of h cannot exceed $O(n^2 + nm)$. Replacing δ by δ/n^2 above will result in an overall probability success of $1-\delta$ for all the $n(n-1)$ copies. This gives us the following theorem,

Theorem 4.1 *Given $\delta > 0$, the algorithm given in Algorithm 4 halts and outputs a perfect model with probability atleast $1-\delta$ in time polynomial in n, m, k and $log(1/\delta)$ and executes*

$O(n^3(n+m)(n^2 log(n/\delta) + N_M + N_E))$ *actions.*

The total number of counterexamples required is at most $O(n^2 N_E)$

If the length of the longest counterexample returned is $m = O(n)$, then the number of actions executed will be $O(n^6 log(n/\delta))$. If the number of states n of the unknown automaton is not known a priori, then we can use a sequence of estimates $E_i, i \geq 0$ where $E_0 = 1$ and $E_i = 2^i$ for the number of states. After executing the loop in lines 8-11 a fixed number of times, if no inconsistency is detected then the estimate for n can be revised (increased by a factor of 2). For each $i \geq 0$, let $C_i = 2^i(2^i+1)ln(1/\delta)/(1-\epsilon_i)$ be the maximum number of times lines 8-11 will be executed for a particular estimate of E_i. ϵ_i is the probability with which no pair of states in the observation table are inconsistent when the number of states in the table has exceeded the current estimate E_i by 1. We now prove a lemma which will show that the modified algorithm halts and outputs a correct automaton with probability at least $1-\delta$.

Lemma 4.2 *For any $\delta > 0$, the modified algorithm for unknown state space size, halts and outputs the correct automaton with probability at least $1-\delta$.*

Proof Let the current estimate for the number of states be $E_i = 2^i$ and let the current number of distinct states in the observation table (S_i, E_i, T_i) be $2^i + 1$. Since ϵ_i is the probability that

none of the pairs of states are inconsistent, the probability that some pair of states $s_i, s_j \in S_i$ are inconsistent is $1 - \epsilon_i$. Hence,

$Pr(Some\ randomly\ selected\ pair\ of\ states\ s_i,\ s_j\ are\ inconsistent) = (1 - \epsilon_i)/\binom{2^i+1}{2}$.

Then, $Pr(Inconsistency\ is\ detected\ between\ states\ s_i\ and\ s_j) = (1 - \epsilon_i)/2\binom{2^i+1}{2}$

$= (1 - \epsilon_i)/[(2^i + 1)2^i]$.

This is the probability that some inconsistency is detected during an execution of the loop when the current estimate is E_i. The probability that no inconsistency is detected during all C_i executions of the loop is given by,

$Pr(No\ inconsistency\ detected\ at\ some\ stage\ i) = [1 - (1 - \epsilon_i)/(2^i)(2^i + 1)]^{C_i} = \delta$

Now the probability that some inconsistency is detected during the entire execution over all estimates $E_i, i \geq 0$ is then given by,

$Pr(Some\ inconsistency\ detected\ during\ entire\ execution) = 1 - \Pi_{i \geq 0}\delta$

Since $\Pi_{i \geq 0}\delta$ is at most δ for any upper bound on i (δ is less than 1) we get that the probability that an inconsistency is detected is at least $1 - \delta$ as desired. □

The algorithm given in Algorithm 4 can be modified very easily to handle adaptive homing sequences. The resulting procedure can generate the correct conjecture with high probability using only $O(n^5 log(n/\delta))$ actions. The entire algorithm remains the same except the part where a distinguishing string x is appended to the existing wrong homing sequence h. Since h is an adaptive homing sequence, it is represented in the form of a tree as described in Section 3. Let $x = a_1, \cdots a_r$ (for each i, $a_i \in \Sigma$) be a string that distinguishes $q_1 h$ and $q_2 h$ and v_0 be the last node in h visited while executing h from q_1 and q_2. v_0 is the same node in both cases since $q_1 < h >= q_2 < h >$. Since the execution terminated at v_0, the tree h does not have a $\gamma(q_1 h)$-child at v_0. In the appended sequence h' we add a $\gamma(q_1 h)$-child which is the root of a linear subtree corresponding to the execution of x. More precisely, in h' each node v_{i-1} has a $\gamma(q_1 h b_1 \cdots b_{i-1})$-child v_i labeled $b_i, \forall 1 \leq i \leq r$. It is easy to see that the modified adaptive homing sequence h' distinguishes the two states q_1 and q_2. This leads to an improved algorithm because not all copies of $L*$ need to be discarded. It is only sufficient to discard L^*_σ, the particular copy on which an inconsistency was discovered. Thus at most $n - 1$ copies of $L*$ are ever discarded and hence there are at most $2n - 1$ copies of $L*$ ever generated. Hence the bound for the adaptive homing sequence based algorithm decreases by a factor of $O(n)$.

4.2 Learning Algorithms for Diversity based Representations

In the presence of a reset in the diversity based representation of an automaton, we can show that Angluin's $L*$ algorithm can be modified to learn the unknown regular set in time polynomial in D. The update graph described in Section 3 captures intuitively the reverse behavior of the actual automaton being learned. By reversing the edges of the update graph,

we will get a finite state machine that accepts strings $w \in \Sigma^*$ such that $w^R \in U$. This isomorphic structure can be learned using the $L*$ algorithm by reversing all the membership queries and reversing all the edges in the equivalence queries.

In the absence of a reset, we describe an algorithm given in [13, 16, 12] that constructs a simple-assignment automaton equivalent to the update graph being learnt. The algorithm builds a set of tests T which will eventually consist of one representative for each test equivalence class. The tests are variables of the simple assignment automaton, where tests $t \in T$ correspond to nodes $[t]$ of the update graph. Intuitively, since in the update graph each test class $[t]$ has a single incoming edge labeled with $a \in \Sigma$ from $[at]$, the algorithm computes for each test t and $a \in \Sigma$ a test $t' \in T$ equivalent to at. Now setting $\Gamma(t, a) = t'$, the simple assignment automaton generated will be isomorphic to the update graph. The algorithm constructs a set T of tests representing all the equivalence classes and determines for each test $x \in \Sigma T$ a test in T equivalent to x.

Initially the set T is a singleton $\{\epsilon\}$. A test t is added to T iff t is inequivalent to every other test in T and hence $|T| \leq D$. For each test $x \in \Sigma T$ a set $r(x) \subseteq T$ is maintained that lists all the tests in T currently equivalent to the test x. The algorithm starts with $r(x) = T$ and refines the set repeatedly when some test $t \in r(x)$ is found to be not equivalent to x. If $|T| = D$ and $r(x) = \{t_x\}$ for all $x \in \Sigma T$, then x must be equivalent to t_x and a simple assignment automaton isomorphic to the update graph can be easily constructed. Its variable set is T, output variable is ϵ and update function Γ is defined as $\Gamma(t, a) = t_{at}$. For constructing a conjecture using existing information where $r(x)$ need not consist of only a singleton, we can choose $V = T$, $v_0 = [\epsilon]$ and $\Gamma(t, a)$ to be an arbitrary element of $r(at)$. The counterexample returned can be used to determine a $t \in T$ and $a \in \Sigma$ such that $\Gamma(t, a) \not\equiv at$. The set $r(at)$ can then be updated by removing $\Gamma(t, a)$ from the set. If at any point in execution some set $r(x)$ is reduced to the empty set, then x is inequivalent to every test in T and hence x gets added to the set T. Since $|T| \leq D$ and $r(x) \subset T$ for each x, it is easy to see that this algorithm converges after at most $(k + 1)D^2$ iterations.

4.2.1 Algorithm for Diversity based Representation (known homing sequence)

Let h denote a known homing sequence for the diversity based representation of the unknown automaton. By definition for every test ht there exists some prefix of h equivalent to ht. For each test t the algorithm will aim to find that prefix of h equivalent to ht and hence maintains a candidate set $C(t) \subset \{0, \cdots |h|\}$ representing prefixes of h equivalent to ht. If a prefix h_i is found to be inequivalent to ht for some test t, then index i is removed from $C(t)$. Suppose executing h from some state q produces the output $\sigma =< \sigma_0, \cdots \sigma_h >$. Now a set $X \subset \{0, \cdots |h|\}$ is coherent with respect to σ iff $\forall i, j \in X, \sigma_i = \sigma_j$. If X is coherent then the common value of σ_i for all i in X is the value selected by X for σ and is denoted by $\sigma[X]$. If $C(t)$ is coherent then the value of t in the current state qh is known. On the other hand, if $C(t)$ is incoherent, then executing t will result in at least one element of $C(t)$ being removed. Further, if i is removed from $C(t)$, then every other prefix h_j of h such that $h_j \equiv h_i$ is also removed from $C(t)$. Hence $C(t)$ reduces in this fashion at most $D - 1$ times. Further, if $C(t_1)$ and $C(t_2)$ are found to disjoint at some stage, then the tests ht_1 and ht_2 can never be equivalent. Also, if for some $a \in \Sigma$ we find that $C(at_1) \not\equiv C(at_2)$ then $at_1 \not\equiv at_2 \Rightarrow t_1 \not\equiv t_2$. If for each test t, the set $C(t)$ is coherent, then the output of each test t at the current state qh is known. This can be used to compute the initial value function ω for the simple assignment automaton. $\omega(t)$ can be taken to be the selected value of $C(t)$.

We now describe how a counterexample z can be used to refine the sets $C(t)$. Let $z = p_i s_i$ where $|p_i| = i$ for all $1 \leq i \leq |z|$. Let $t_i = \Gamma(\epsilon, s_i)$ and $u_i = p_i t_i$. The algorithm now maintains a candidate set for each such u_i. If at a certain stage of the algorithm, there exists some i such that $C(u_i) \bigcap C(u_{i+1}) = \emptyset$ then $u_i \not\equiv u_{i+1}$. Since $u_i = p_i t_i$ and $u_{i+1} = p_i a t_{i+1}$ for some $a \in \Sigma$, it implies that $t_i \not\equiv a t_{i+1}$. But $t_i = \Gamma(t_{i+1}, a) \in r(a t_{i+1})$ it follows that t_i can be deleted from $r(a t_{i+1})$. It is easy to see that $C(u_0)$ and $C(u_{|z|})$ are disjoint. Since z is a counterexample, let the predicted value of z at the current state qh be $\omega(\Gamma(\epsilon, z)) = \omega(t_0)$ be 0 where $\gamma(qhz) = 1$. By definition, $C(u_0) = C(t_0) \subset \sigma^{-1}(0)$ and $C(u_{|z|}) \subset \sigma^{-1}(1)$ and hence $C(u_0) \bigcap C(u_{|z|}) = \emptyset$. Although the conjecture is inconsistent with z at the state qh, z could be consistent with the actual automaton at other states. Executing h over and over again to reduce the sets $C(u_i)$ may lead to a state where the candidate sets $C(u_i)$ are all coherent without any consecutive pair being disjoint. The algorithm then makes a new conjecture with the same V, v_0 and Γ but a new ω chosen according to the current state. This then generates a new set of u_i's for which candidate sets must be maintained. We later show that no more than $D - 1$ such sequences need ever be created before one of the sets $r(x)$ is reduced.

The complete algorithm is given in Algorithm 5. On receiving a counterexample, a sequence of tests $u_{l0}, \cdots u_{lm_l}$ is created where l counts the number of such counterexamples generated for a given Γ. The set $K(i, j)$ is a candidate set for u_{ij}.

4.2.2 Correctness of the Learning Algorithm

We first prove a lemma bounding the number of counterexamples generated for a given Γ.

Lemma 4.3 $l \leq (D - 1)$

Proof On each iteration l is reset to 0 iff $K(i, j) \bigcap K(i, j + 1) = \emptyset, 1 \leq i \leq l, 0 \leq j < m_i$. Hence in each iteration for l to be non-zero $K(i, j) \bigcap K(i, j+1) \neq \emptyset$. Further, on each iteration $K(i', j')$ respects $K(i, j)$ for $1 \leq i' \leq i \leq l, 0 \leq j \leq m_i$ and $1 \leq j' \leq m_{j'}$. A set S respects another set S' iff either $S \subseteq S'$ or $S \cap S' = \emptyset$. If $K(i, j)$ is ever found to be incoherent, then $K(i', j')$ will be coherent on that iteration (i is the smallest index such that $K(i, j)$ is incoherent for some j). If some element of $K(i', j')$ in $K(i, j)$ is removed on updation, then all elements of $K(i', j')$ will be removed. Hence $K(i', j')$ respects $K(i, j)$ on each iteration for $i' < i$.

To prove $l \leq D - 1$ we define a sequence of undirected graphs $G_0, \cdots G_l$. The vertex set of each graph is $\{0, \cdots |h|\}$. In G_i an edge connects r and s iff $h_r \equiv h_s$ or $\{r, s\} \subset K(i', j)$ for some $1 \leq i' \leq i, 0 \leq j \leq m_{i'}$. We now show that the number of connected components in this sequence of graphs reduces by at least 1 from one graph to the other. Since any graph has at least one connected component and the number of components of G_0 is at most D, we would then have $l \leq D - 1$.

The edge set of G_{i-1} is a subset of the edge set of G_i. So we find a single pair of vertices connected in G_i but not in G_{i-1}. Since $K(i, 0) = C(u_0)$ and $K(i, m_i) = C(u_{|z|})$ it is evident that $K(i, 0)$ and $K(i, m_i)$ are disjoint sets. Let $r \in K(i, 0)$ and $s \in K(i, m_i)$. Clearly, r and s are connected in G_i because $K(i, j) \bigcap K(i, j + 1) \neq \emptyset$ on each iteration. But r and s are not connected in G_{i-1}. If they were, then there are adjacent vertices r' and s' on the path from r to s such that r' but not s' is in $K(i, 0)$. Since r' and s' are adjacent, either $h_{r'} \equiv h_{s'}$ or $\{r', s'\} \subset K(i', j)$ for some $i' < i$. But this implies $\{r', s'\}$ respects $K(i, 0)$ which is a contradiction.

This completes the proof proving that $l \leq D - 1$. □

Algorithm 5 Learning Algorithm for Diversity based Representation

Initialize T to ϵ, $C(\epsilon) = \{0, \cdots |h|\}$, $\forall a \in \Sigma, r(a) = T$ and $\Gamma(\epsilon, a) = \epsilon$
Let $l = 0$
while true **do**
  Execute h producing output σ
  if $C(t)$ is incoherent for some $t \in T$ **then**
    Execute t and update $C(t)$
  else
    if $K(i,j)$ is incoherent for some $1 \leq i \leq l, 0 \leq j \leq m_i$ **then**
      Choose the smallest i for which some $K(i,j)$ is incoherent
      Execute u_{ij} and update $K(i,j)$
    else
      $\omega(t) = \sigma[C(t)], t \in T$
      Conjecture $S = (T, \Sigma, \Gamma, \epsilon, \omega)$
      if S is a perfect model **then**
        Stop and output S
      else
        Obtain counterexample z
        Let $l = l + 1, m_l = |z|$
        for $0 \leq j \leq m_l$ **do**
          $u_{lj} = p_j \Gamma(\epsilon, s_j)$ where $z = p_j s_j$ and $|p_j| = j$
          $K(l,j) = \{0, \cdots |h|\}$
        end for
        $K(l,0) = \sigma^{-1}(\omega(u_{l0}))$
        Execute $z = u_{lm_l}$ and update $K(l, m_l)$
      end if
    end if
  end if
  if $K(i,j) \bigcap K(i,j+1) = \emptyset$ for some $1 \leq i \leq l, 0 \leq j < m_i$ **then**
    $x = a_0 t_0$ where $u_{i,j+1} = p a_0 t_0$, $|p| = j$ and $a_0 \in \Sigma$ ($u_{ij} = p\Gamma(t_0, a_0)$)
    $r(x) = r(x) \setminus \{\Gamma(t_0, a_0)\}$
    if $r(x) = \emptyset$ **then**
      $r(t) = r(t) \cup \{x\}$ for all $t \in (\Sigma T \setminus T)$
      $T = T \cup \{x\}$ and $C(x) = \{0, \cdots |h|\}$
      $r(ax) = T$ for all $a \in \Sigma$
    end if
    $\Gamma(t, a) =$ any member of $r(at)$ for $a \in \Sigma, t \in T$
    $l = 0$
  end if
end while

Theorem 4.4 *The algorithm described in Algorithm 5 halts in polynomial time after executing*

$O(kmD^4(|h| + D + m))$ *actions*

Proof On each iteration, if $i \notin C(t)$ then $h_i \not\equiv ht$ for any t by definition. Further, $C(t)$ is non-empty always because ht is equivalent to some h_i since h is a diversity based homing sequence. These statements are also true for all the sets $K(i,j)$. This implies that $K(i,j) \bigcap K(i,j+1) = \emptyset$ implies $u_{ij} \not\equiv u_{i,j+1}$ which then means $x = a_0 t_0 \not\equiv \Gamma(t_0, a_0)$. Hence in each iteration $t \notin r(x)$ only if $x \not\equiv t$ for $t \in T$ and $x \in \Sigma T$. Further, $r(x)$ is non-empty on each iteration. Hence if the last element of $r(x)$ is removed, then $x \not\equiv t$ for all $t \in T$. Hence it follows that $|T| \leq D$ on each iteration implying that the inner loop is executed at most $(k+1)D^2$ times i.e., l is reset to 0 at most this many times.

The set $\{0 \leq i \leq |h| : h_i \equiv x\}$ respects $C(t)$ for all tests t and x. Hence $C(t)$ can be reduced at most $D-1$ times which then implies that $C(t)$'s are incoherent at most $D(D-1)$ times. Further $K(i,j)$'s are incoherent at most $(k+1)mD^2(D-1)^2$ times where $\forall 1 \leq i \leq l, m_i \leq m$. Further all the sets will be coherent at most $(k+1)D^2(D-1)$ times giving an overall bound on the total number of iterations. Since at most $|h|+m+(D-1)$ actions are executed in each iteration, the result follows. □

Learning algorithm when the diversity based homing sequence is unknown is a slight modification to Algorithm 5 and given in [13]. In order that the algorithm executes in time proportional to the bound on the number of actions, checking the coherence of candidate sets must be done efficiently. In a naive implementation, checking coherence of a set $S \subset \{0, \cdots |h|\}$ will take time proportional to $O(|h|)$ which will then give a bound of $O(D|h|)$ for checking coherence of candidate sets of all tests in T. This exceeds the action bound by a factor of D. A different representation will maintain a partition π of the set $\{0, \cdots |h|\}$ with the interpretation that i, j are in different partitions iff they have been found to be not equivalent. Hence $|\pi| \leq D$. Since the partition respects all the candidate sets, each candidate set can be maintained as a set of pointers to this partition π. Since each set has at most D pointers, the coherence of the set can be determined in time $O(D)$.

Maintaining the partition π is a simple task. Each time h is executed coherence of all the blocks in the partition can be tested in time $O(|h|)$. If any block s is found to be incoherent, then it is split into two blocks $s \cap \sigma^{-1}(0)$ and $s \cap \sigma^{-1}(1)$. Since $|\pi| \leq D$ this can happen at most $D-1$ times. Each time this happens, all the pointers from the candidate sets can be updated in time $O(mD^2)$.

5 Learning Algorithms for Permutation Automaton

A permutation automaton $A = \langle Q, \Sigma, \delta, q_0, \gamma \rangle$ is a special type of automaton such that the transition function $\delta(., a)$ generates a permutation of the state space Q for every input symbol $a \in \Sigma$. In this section we describe algorithms from [13] for inferring an unknown permutation automaton without the need for equivalence queries.

5.1 State-Space based Learning Algorithm

For permutation automata we can show that the homing sequence h is also a distinguishing sequence i.e., $q_1 < h >= q_2 < h >$ iff $q_1 = q_2$. Hence the identity of a state q is readily

given by the output of the execution of h from q. The problem of learning a permutation automaton given the homing sequence can then be reduced to the problem of learning a visible automaton. A visible automaton is an automaton which outputs the name of the state reached at each step. Inference of a visible automaton is trivial; for each known state q we determine all the successor states by querying $\delta(q, a)$ for all $a \in \Sigma$. This can be done by simply executing a from q and observing the output. The entire automaton can be learned by executing $O(kn^2)$ actions.

To infer the permutation automaton, we label each state with $q < h >$ (to identify the current state just execute h and observe the output). But executing h will leave us in a state whose identity is not known immediately. Since the inference procedure for the visible automaton requires us to know the identity of this state, the algorithm must be able to identify a state without the need for executing h from that state. This can be done by maintaining a table u whose entries are such that for every $\sigma = q < h >$, $u(\sigma) = qh < h >$. Hence we can reach a state qh whose identity can be obtained from the table u, execute an experiment $a \in \Sigma$ as suggested by the visible automaton procedure and then execute h again to determine the identity of the state qha. This simple procedure is described in the algorithm 6.

Algorithm 6 Learning Algorithm for Permutation Automaton

Initialize u, d to be undefined everywhere
Execute h producing output σ
while true **do**
  if $u(\sigma)$ is not defined **then**
    Execute h and store the output τ in $u(\sigma)$
    $\sigma = \tau$
  else
    if $\exists a \in \Sigma, w \in \Sigma^*, d(u(\sigma), w)$ is defined but $d(u(\sigma), wa)$ is undefined **then**
      Choose the shortest such wa.
      Let $\alpha = d(u(\sigma), w)$
      Execute wa and then h producing output τ
      $d(\alpha, a) = \tau$ and $\sigma = \tau$
    else
      exit loop
    end if
  end if
end while
On input $w \in \Sigma^*$, let $\alpha = d(u(\sigma), w)$. Output $\gamma(qw) = \alpha_0$ where α_0 is the first symbol in α and $q = u(\sigma)$.

Let q_σ denote the state in Q for which $\sigma = q_\sigma < h >$. The table entry $u(\sigma)$ is then $q_\sigma h < h >$. The transition function $d : Q < h > \times \Sigma \longrightarrow Q < h >$ will be used to store and compute the output of h in future states. Given $\sigma \in Q < h >$ and $a \in \Sigma$, $d(\sigma, a) = q_\sigma a < h >$. The following theorem proves correctness of the algorithm and bounds its running time.

Theorem 5.1 *The algorithm given in Algorithm 6 halts after executing at most $O(kn(|h| + n))$ actions and in time $O(kn(|h| + kn))$*

Proof Since the number of states is n after at most $n + kn$ iterations the procedure must halt since every entry of u and d will be defined. We can view the function d as defining a directed

graph with vertices $Q < h >$ and an edge from σ to $d(\sigma, a)$ with label a. The process of finding the smallest sequence wa can then be achieved by finding a path in this graph from $u(\sigma)$ to another vertex α. This can be done using BFS in time $O(kn)$ and the experiment wa has length at most n. Hence the number of actions executed is upper bounded by $O(kn(|h|+n))$. The rest of the algorithm can be executed in time $O(|h|)$ using a binary tree representation for $Q < h >$.

It is trivial to see that the following invariants hold during the execution of the algorithm,

If $\sigma \in Q < h >$ and $u(\sigma)$ is defined then $u(\sigma) = q_\sigma h < h >$

If $\sigma \in Q < h >, a \in \Sigma$ and $d(\sigma, a)$ is defined then $d(\sigma, a) = q_\sigma a < h >$

Hence the output generated by the algorithm is correct with respect to the unknown permutation automaton being learnt. □

We now show that any sufficiently long random sequence will be a homing sequence for the permutation automaton with high probability. We state the following lemma without proof.

Lemma 5.2 *Let q_1 and q_2 be two distinct states in the automaton and x be a random sequence of length $2kn^4ln(n)$. To construct x, at each step, with equal probability, we either do nothing or execute a uniformly and randomly chosen basic action from Σ. Then the probability that $\gamma(q_1x) \neq \gamma(q_2x)$ is at least $1/(2n)$.*

We will also use the following form of Chernoff bound.

Lemma 5.3 *If $X_1, \cdots, X_m$ are m independent Bernoulli trials, each succeeding with probability p so that $E[X_i] = p$, then for $S = X_1 + X_2 \cdots + X_m$ and $0 \leq \gamma \leq 1$,*

$Pr(S > (1+\gamma)pm) \leq e^{-\gamma^2 pm/3}$

$Pr(S < (1-\gamma)pm) \leq e^{-\gamma^2 pm/2}$

Theorem 5.4 *Let $\delta > 0$ and h be a random sequence of length $8kn^5.ln(n).(n + ln(1/\delta))$. Then h is a homing sequence with probability at least $1 - \delta$.*

Proof Let $x_1, \cdots, x_r$ be a sequence of random strings constructed using the procedure described in Lemma 5.2. Let $y_i = x_1x_2 \cdots x_i$. We can claim that y_r is a homing sequence with high probability. Consider a sequence of trials where success in trial i means that either y_{i-1} is a homing sequence or $|Q < y_i > | > |Q < y_{i-1} > |$. Clearly, if n trials succeed then y_r is a homing sequence. For any y_{i-1} the probability of success on the ith trial is $1/(2n)$ from Lemma 5.2. Applying the chernoff bound on these trials, we get that the probability of fewer than n successes in r trials is at most δ if $r \geq 4n(n + ln(1/\delta))$. □

Diversity based learning algorithm for learning permutation automaton are similar in structure to the algorithm described in this section and can be referred to in [13].

6 Learning Context-Free Languages

Structural descriptions of a context-free grammar are unlabeled derivation trees of the grammar. In this section we present a learning algorithm given in [14, 15] which efficiently learns structural descriptions of an unknown context-free language using membership and equivalence queries similar to the $L*$ algorithm described in Section 3 for regular languages. Levy and Joshi in [10] and Fass in [5] have shown that efficient grammatical inferences in terms of structural descriptions is possible. It has been shown that the set of derivation trees representing a particular context-free grammar is a rational set of trees. Since a rational set of trees can be recognized by a tree automaton, the problem of learning a context-free language reduces to the problem of learning a tree automaton. Here the problem has been slightly modified from the standard grammatical inference problem of learning the context-free grammar. For a given context-free language it is known that there are infinitely many grammars that generate the language such that the structural descriptions of those grammars are different. We first give the basic definitions and then describe the algorithm along with its analysis. This algorithm not only finds a context-free grammar for the unknown language but also guarantees that the grammar is structurally equivalent to the unknown language (equivalence is based on the structural descriptions returned by the teacher).

6.1 Basic Definitions

Let N be the set of natural numbers and $N*$ denote the free monoid generated by N with identity element ϵ and the concatenation operator '.'. For any $x, y \in N*$ we define the ordering relation $y \leq x$ iff $x = y.z$ for some $z \in N*$. Further, $y < x$ iff $y \leq x$ and $y \neq x$. Here we will denote a concatenated string $x.y$ as xy. Let V be a finite set of symbols associated with a rank relation $r_v \subseteq V \times N$. V_n denotes the subset $\{f \in V | (f, n) \in r_v\}$ of V. V_n identifies the set of symbols having arity n. In terms of context-free grammars these refer to non-terminals that can derive n elements in some production. Hence V_0 refers to the terminal symbols or the input alphabet of the grammar. A tree or structural derivation over V is a mapping t from Dom_t into V where Dom_t is a finite subset of $N*$ such that,

- If $x \in Dom_t$ and $y < x$ then $y \in Dom_t$
- If $yi \in Dom_t$ then $\forall 1 \leq j \leq i, yj \in Dom_t$
- $t(x) \in V_n$ iff $\forall 1 \leq i \leq n, xi \in Dom_t$

A element of Dom_t is the node of the tree and the mapping $t(x)$ is called the label assigned to node x. Let V^T denote the set of all trees over the set V. For convenience, $f(t_1, \cdots, t_n)$ denotes a tree with root node labeled $f \in V_n$ and having n subtrees t_1 thru t_n. A node y is called terminal if $\forall x \in Dom_t, y \not< x$. The set of all terminal nodes of a tree is called the $frontier$ of the tree. All non-terminal nodes are called interior nodes. The depth of $x \in Dom_t$ denoted $depth(x)$ is the length of x and depth of a tree t is defined as $depth(t) = \max\{depth(x) | x \in Dom_t\}$.

Let \$ be a new symbol of arity 0 not in V. Let $V_\$^T$ denote the set of all trees over $V \bigcup \$$ such that there is exactly one terminal node in the tree labeled \$. For $s \in V_\$^T$ and $t \in (V^T \bigcup V_\$^T)$ we define tree append operation $\#$ as follows,

$s\#t(x) = s(x)$ if $x \in Dom_s$ and $s(x) \neq \$$ and

$s\#t(x) = t(y)$ if $x = zy, s(z) = \$$ and $y \in Dom_t$

For sets $S \subseteq V_{\$}^T$ and $T \subseteq (V^T \bigcup V_{\$}^T)$, let $S\#T$ define the set $\{s\#t | s \in S, t \in T\}$.

A skeletal structural description consists of only the special symbol $V = \sigma$ with the rank relation $r_v \subseteq \{\sigma\} \times \{1, 2, \cdots, m\}$ where m is the maximum rank. A tree defined over $\sigma \cup V_0$ is called a skeleton. Given any tree in $t \in V_{\$}^T$ its skeleton can be obtained by replacing the labels of all the interior nodes with σ and the operation is denoted by $K(t)$.

6.2 Tree Automaton and Context Free Grammars

We now describe a tree automaton which is a generator for a rational set of skeletal trees. Since any context-free grammar can also be represented by a rational set of structural descriptions, the problem of learning a context-free grammar then reduces to the problem of learning a tree automaton that generates an identical set of skeletal structural descriptions.

A tree automaton over V is a tuple $A = \langle Q, V, \delta, F \rangle$ such that Q is a finite set of states, $F \subseteq Q$ is the set of accepting states and $\delta = (\delta_0, \cdots \delta_m)$ consists of the following maps,

$\delta_0(a) = a, \forall a \in V_0$

$\delta_k : V_k \times (Q \bigcup V_0)^k \longrightarrow Q, \forall 1 \leq k \leq m$

The transition function δ can be extended to trees in the usual way,

$\delta(f(t_1, \cdots t_k)) = \delta_k(f, \delta(t_1), \cdots, \delta(t_k))$ if $k \geq 1$

$\delta(f) = \delta_0(f)$ where $f \in V_0$

A tree t is accepted by a tree automaton iff $\delta(t) \in F$. Let $T(A)$ denote the set of trees accepted by the automaton A. We now state the replacement lemma for tree automata which has been proved in [14].

Lemma 6.1 (Replacement Lemma for Tree Automata) *Let A be a tree automaton as defined above. For $s, s' \in V^T$ and $t \in V_{\$}^T$, if $\delta(s) = \delta(s')$ then $\delta(t\#s) = \delta(t\#s')$*

A context-free grammar is denoted by the tuple $G = \langle N, \Sigma, P, S \rangle$ where N is the set of non-terminals, Σ is the set of terminals, $S \in N$ is the start symbol of the grammar and P is a production of the form $A \longrightarrow \alpha$ where $A \in N$ is a non-terminal and α is a string in $(N \bigcup \Sigma)^*$. For any strings β and γ a derivation step of the grammar is denoted as $\beta A \gamma \Rightarrow \beta \alpha \gamma$ and $\Rightarrow^*$ is the transitive closure of $\Rightarrow$. The language generated by a grammar is defined as $L(G) = \{\omega | \omega \in \Sigma^*, S \Rightarrow^* \omega\}$. The set $D_A(G)$ is defined as a set of trees over $N \bigcup \Sigma$ as,

$D_A(G) = \{a\}$ if $A = a \in \Sigma$

$D_A(G) = \{A(t_1, \cdots, t_k) | A \longrightarrow B_1, \cdots B_k \in P, \forall 1 \leq i \leq k, t_i \in D_{B_i}(G)\}$ if $A \in N$

Thus $D_A(G)$ is the set of derivation trees of G with root A and hence $D_S(G) = D(G)$ represents the set of all the derivation trees of G whose frontier gives the set of all the strings that can be generated by G. The skeletal structural description of G is then denoted by $K(D(G))$. Two context-free grammars are structurally equivalent iff $K(D(G_1)) = K(D(G_2))$. We now give a mapping from a grammar G to a tree automaton $A(G) = \langle Q, \sigma \cup \Sigma, \delta, F \rangle$ as follows:

- $Q = N$ and $F = \{S\}$
- $\delta_0(a) = a, \forall a \in \Sigma$
- $A \in \delta_k(\sigma, B_1, \cdots, B_k), \forall A \longrightarrow B_1, \cdots B_k \in P$

Proposition 6.2 (Equivalence of skeletal structural descriptions) $T(A(G)) = K(D(G))$ *i.e., the structural descriptions generated by the tree automaton $A(G)$ and those representing the grammar G are identical.*

Proof Proof of this proposition is given in [14].

Now, we give a mapping of a tree automaton A to a context-free grammar $N(A) = (N, \Sigma, P, S)$ as follows,

- $N = Q \bigcup \{S\}$
- $P = \{\delta_k(\sigma, x_1, \cdots x_k) \longrightarrow x_1 \cdots x_k | x_1, \cdots, x_k \in (Q \bigcup \Sigma)\}$
  $\bigcup \{S \longrightarrow x_1 \cdots x_k | \delta_k(\sigma, x_1, \cdots, x_k) \in F\}$

Proposition 6.3 *Given a tree automaton A the grammar $N(A)$ generated as described above has the property that $K(D(N(A))) = T(A)$.*

Proof Proof of this proposition is given in [14].

6.3 Learning Algorithm for Tree Automaton

In this section we extend the classical Angluin's algorithm described in Section 3 for learning tree automata. We first describe the modifications required to the observation table and then describe the algorithm along with its time and space complexity.

6.3.1 Observation Table (S, E, T)

Let S be a finite set of subtree-closed skeletons over $(\sigma \cup \Sigma)^T$ (if $s \in S$ then all subtrees of s of length at least 1 is also in S). Let E be a set of trees over $(\sigma \cup \Sigma)^T_{\$}$. E is said to be prefix-closed with respect to S if $e \in E - \{\$\}$ then there exists $e' \in E$ such that $e = e'\#\sigma(s_1, \cdots, s_{i-1}, \$, s_i \cdots, s_{k-1})$ for some $s_1, \cdots, s_{k-1} \in S$ and $i \in N$. The observation table (S, E, T) consists of a non-empty finite subtree closed set S, a nonempty finite set E prefix-closed with respect to S, $X(S) = \{\sigma(u_1, \cdots u_k) | u_1, \cdots, u_k \in (S \bigcup \Sigma), \sigma(u_1, \cdots u_k) \notin S \; for \; k \geq 1\}$ and function T mapping $(E\#(S \bigcup X(S)))$ to $\{0, 1\}$. $T(s) = 1$ iff s is a structural description of the language generated by the tree automaton being conjectured. A closed and consistent observation table can be defined similar to the description in Section 3. If (S, E, T) is a closed and consistent observation table then a tree automaton $A(S, E, T)$ can be conjectured from the table as follows:

- $Q = \{row(s) | s \in S\}$
- $F = \{row(s) | s \in S \; and \; T(s) = 1\}$
- $\delta_k(\sigma, row(s_1), \cdots, row(s_k)) = row(\sigma(s_1, \cdots s_k)), \forall s_1, \cdots s_k \in (S \bigcup \Sigma)$
- $\delta_0(a) = a, \forall a \in \Sigma$

where for all $a \in \Sigma$ the function row is augmented to be $row(a) = a$. The deterministic tree automaton so generated is well-defined. The following lemma can now be shown to be true using arguments similar to Lemma 3.4 in Section 3.

Lemma 6.4 *If (S, E, T) is closed and consistent then $A(S, E, T)$ is consistent with T i.e., for every $s \in (S \bigcup X(S))$ and $e \in E$, $\delta(e\#s)$ is in F iff $T(e\#s) = 1$.*

Suppose G_U is the unknown language to be learned (up to structural equivalence). We assume that terminal alphabet Σ is known. A structural membership query proposes a skeleton s and asks whether it is in $K(D(G_U))$. A structural equivalence query proposes a grammar G' and asks whether $K(D(G_U)) = K(D(G'))$. Since equivalence of two structural descriptions can be done in time polynomial in the size of the two descriptions an efficient teacher can be synthesized for answering these queries. The algorithm and its analysis is similar to the $L*$ algorithm given in Section 3 and hence not described here.

7 Compositional Verification by Learning Assumptions

The verification problem for a system consisting of multiple components can be decomposed into subproblems of verifying individual components using assume-guarantee reasoning [1, 11]. A component is verified by making reasonable assumptions regarding the other components in the system. Verification succeeds if the properties of the component are satisfied under the assumptions and the other components are a refinement of the assumptions (generate a strict subset of the behaviors allowed by the assumption). In this section we describe a direct application of Angluin's $L*$ algorithm to the compositional verification problem. We first give basic definitions and then describe the application.

7.1 Basic Definitions

Let X be a set of boolean variables and the corresponding set of primed variables be denoted by $X' = \{x' | x \in X\}$. ϕ is a boolean formula over X and a valuation s of X satisfying ϕ is denoted by $\phi(s)$. A symbolic module or component is given by $M = (X, X^I, X^O, Init, T)$ where,

- X is a finite set of boolean variables controlled by the component,
- X^I is a finite set of boolean input variables to the component disjoint from X,
- $X^O \subseteq X$ is the set of output variables generated by the component visible to the environment,
- $Init(X)$ is the initial state predicate over X,

- $T(X, X^I, X')$ is the transition predicate where X' encodes the successor state of the component.

Let $X^{IO} = X^I \bigcup X^O$ denote the set of communication variables in the system. Let S be the set of states of M (all possible valuations of X) and S^I, S^O and S^{IO} denote all possible valuations of X^I, X^O and X^{IO} respectively. Let $s[Y]$ for $Y \subseteq X$ and $s \in S$ denote the valuation of Y at state s. A run of M is denoted by a sequence of states $s_0, s_1, \cdots$, where $s_0 \in Init(X)$, each $s_i \in (S \bigcup S^I)$ and $T(s_i[X], s_i[X^I], s'_{i+1}[X'])$ holds such that $s'_{i+1}[X'] = s_{i+1}[X]$. $M \models \phi$ holds iff property ϕ over X^{IO} holds for module M (for every sequence of states representing a run of the module, the property holds at every state). A trace of M is a run of M confined to the communication variables. The set of all traces generated by a module is called the language of the module and denoted as $L(M)$. If M_1 and M_2 are two modules then M_1 is said to be a refinement of M_2 iff $L(M_1) \subseteq L(M_2)$ and is denoted as $M_1 \sqsubset M_2$.

Given two modules $M_1 = (X_1, X_1^I, X_1^O, Init_1, T_1)$ and $M_2 = (X_2, X_2^I, X_2^O, Init_2, T_2)$ such that $X_1 \cap X_2 = \Phi$, composition of M_1 and M_2 is given as $M_1||M_2 = (X, X^I, X^O, Init, T)$ where,

- $X = X_1 \cup X_2$, $X^I = X_1^I \cup X_2^I$ and $X^O = X_1^O \cup X_2^O$
- $Init(X) = Init_1(X_1) \wedge Init_2(X_2)$
- $T(X, X^I, X') = T_1(X_1, X_1^I, X_1') \wedge T_2(X_2, X_2^I, X_2')$

The compositional verification problem can now be defined as,

Given two modules M_1 and M_2 as above with the additional constraint that $X_1^I = X_2^O$ and $X_1^O = X_2^I$ and a safety property $\phi(X^{IO} = X_1^{IO} = X_2^{IO})$, does $(M_1 \parallel M_2) \models \phi$.

The assume guarantee rule used to prove this property in a compositional manner is that,

$((M_1 \parallel A) \models \phi)$ and $(M_2 \sqsubset A) \Rightarrow (M_1 \parallel M_2 \models \phi)$

To verify the property ϕ on $M_1 \parallel M_2$ we verify it on $M_1 \parallel A$ where A is some assumption such that $M_2 \sqsubset A$.

7.2 Compositional Verification using $L*$ Algorithm

Let M_1 and M_2 be two modules as defined in Section 7.1. Let L_1 be the set of traces $\sigma = s_0, s_1 \cdots$ where each $s_i \in S^{IO}$ such that $\forall \sigma \in L_1, \sigma \notin L(M_1)$ or $\phi(s_i)$ holds for all $s_i \in \sigma$. The assumption A of the assume-guarantee rule must satisfy the constraint $A \subseteq L_1$. Let $L_2 = L(M_2)$ be the set of traces generated by M_2. Now A must also satisfy the constraint $L_2 \subseteq A$ which then implies that a valid A is any language that is a superset of L_2 and a subset of L_1. The $L*$ algorithm described in Section 3 is used to compute a correct assumption A. Equivalence queries $L(C) =^? U$ are answered based on the result of a subset query ($L(C) \subseteq L_1$) and a superset query ($L_2 \subseteq L(C)$).

The paper answers membership queries with respect to the language L_1. This intuitively tries to build the largest assumption A possible for a given M_1 and M_2 given $M_1 \parallel M_2 \models \phi$. On the other hand, if the oracle answers queries with respect to the language $L_2 = L(M_2)$,

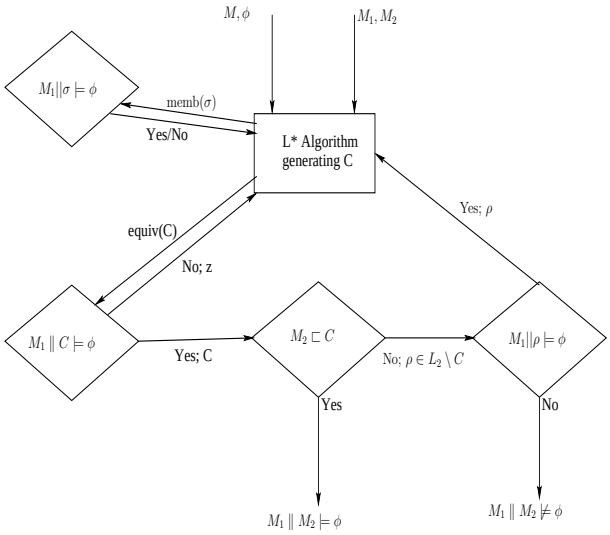

Figure 2: Symbolic Compositional Verification using Learning

then the algorithm is trying to construct the smallest valid assumption A for the components M_1 and M_2 when $L_2 \subseteq L_1$.

As shown in Fig. 2, membership queries in the algorithm will be answered by checking for safety with respect to module M_1. To answer equivalence query, we first perform a subset check by checking the conjecture with respect to M_1; if the query fails then the returned counterexample trace will be used by the learner to compute the next conjecture. If the query succeeds, then we check for refinement of M_2 with respect to the conjecture. If this superset query fails, then the returned counterexample is checked for safety with respect to M_1. Since this counterexample is present in M_2, if the query fails then $(M_1 \parallel M_2) \not\models \phi$ and the algorithm halts. If the query succeeds then the counterexample is returned to the learner.

We now compute bounds on the size of the automaton being constructed by the algorithm. All membership queries and counterexamples are consistent with the language L_1. If $M_1||M_2$ does indeed satisfy ϕ then L_2 will be a subset of L_1 and hence B_1 (automaton recognizing L_1) is an adequate assumption to witness satisfaction of the property. If $M_1||M_2$ does not satisfy ϕ then L_1 is a subset of L_2 and again B_1 is an adequate assumption to witness failure of the property. Thus, this procedure always halts and reports correctly whether $M_1||M_2$ satisfies ϕ and never generates any conjecture with states more than the states in a minimal DFA accepting L_1. Using the $L*$ algorithm directly will result in an inefficient implementation because the size of the alphabet S^{IO} is exponential in the size of the communication variables X^{IO} which then implies that the size of the observation table constructed by the algorithm will be exponential in the size of X^{IO}. Alur et. al in [1] have presented a symbolic implementation of the algorithm which will represent sets as BDDs and hence reduces the size of the observation table by storing the transition function implicitly.

8 Comparison and Improvements

Angluin's $L*$ algorithm described in Section 3 along with the improvements suggested by Rivest et. al. described in Section 3.5 is the best known algorithm for computing an unknown regular language using active and passive experiments (minimally adequate teachers) assuming that the teacher has the ability to reset the automaton prior to answering any query. In the absence of a reset facility, Rivest and Schapire as described in Sections 4, 4.2 have given efficient algorithms for inferring the unknown regular set using homing sequences. They have described algorithms for both global state-space based representation and diversity based representation of the finite state automata. Homing sequence based algorithms assume that the unknown automaton being inferred consists of a single strongly connected component. If there are multiple strongly connected components in the automaton, then the experiments can get stuck in one such component without any means of coming out of it due to the absence of a reset. But the homing sequence based algorithms halt and output the correct automaton with a certain error probability if the homing sequence of the automaton is not known. Learning context-free grammars from their structural descriptions is similar to learning regular sets and hence Angluin's $L*$ algorithm has been appropriately adapted by Sakakibara in [14] for learning tree automaton. In this section, we discuss problems associated with applying the improvements to the $L*$ algorithm suggested by Rivest et. al. to the learning algorithm for grammars. We also show that the number of membership queries executed by the algorithm can be significantly reduced if certain restrictions are enforced on the counterexamples returned by the teacher.

8.1 Improved Algorithm for Context-Free Languages

Rivest and Schapire in [13] described improvements to the original $L*$ algorithm which ensured that the number of elements in S was always less than or equal to n instead of mn which was the case earlier. This reduced the number of membership queries from $O(kmn^2)$ to $O(kn^2 + nlog(m))$. In order to ensure that the number of entries in S was always less than or equal to n, the algorithm had to make an additional $log(m)$ membership queries for each counterexample. The result was addition of a single string to the set E for each counterexample returned, instead of m strings being added to the set S as in the original algorithm. If $z(|z| \leq m)$ was the counterexample returned, the $log(m)$ membership queries were used to determine the state $sa \in S\Sigma$ of the conjecture such that $row(sa) = row(s')$ for some $s' \in S$ but $\delta(q_0, sa) \neq s'$ where δ is the transition function of the correct automaton. The string e added to the set E distinguishes the states sa and s' thereby increasing the size of set S by at least 1 (sa gets added to set S in the next iteration to make the table closed).

The counterexample t returned by the teacher for context-free languages is a tree. Let us assume that the skeletal structural description t is accepted by the conjecture tree automaton $A(S, E, T) = (S, V, \delta, F)$ ($t \in T(A(S, E, T))$) but not by the unknown context-free grammar G ($t \notin K(D(G))$). We assume that the grammar G is represented by a tree automaton $A(G) = (S', V', \delta', F')$. Further, let t have m internal nodes with state labels $s_1, \cdots, s_m$ such that $row(s_i) = \delta(s_i)$ where $s_i \in (S \bigcup S\Sigma)$ for each i. Since t is not derived by the grammar G, there exists at least one node s in t such that the derivation from s is not in G i.e., if $row(s) = \delta_k(\sigma, row(s_1), \cdots, row(s_k))) = row(\sigma(s_1, \cdots, s_k))$ for some $s, s_1 \cdots, s_k \in S$ and $\sigma(s_1, \cdots, s_k) \in S\Sigma$ then in $A(G)$, $\delta'_k(\sigma, q_1, \cdots, q_k) = row(\sigma(q_1, \cdots, q_k)) \neq q_s$ where $q_i = \delta'(s_i)$ for each i and $q_s = \delta'(s)$ such that $q_s, q_1, \cdots, q_k \in S'$. There could be other such derivations

in t which are also not in $A(G)$. We now restrict the type of counterexamples returned by the teacher for which improvements to the algorithm described in Section 6 can be made.

Suitable Counterexample A counterexample t returned by a minimally adequate teacher for context free grammars is said to be a suitable counterexample iff

- $t \in A(S,E,T) \setminus A(G)$ and there exists exactly one set of nodes $s, s_1, \cdots s_k \in internal(t)$ such that $row(s) = \delta(\sigma(s_1, \cdots, s_k))$ but $\delta(s) = q_s \neq \delta'(\sigma(s_1, \cdots s_k))$.
- $t \in A(G) \setminus A(S,E,T)$ and there exists exactly one set of nodes $s, s_1, \cdots s_k \in internal(t)$ such that $\delta(s) = q_s = \delta'(\sigma(s_1, \cdots, s_k))$ but $row(s) \neq \delta(\sigma(s_1, \cdots s_k))$ and

Assuming that the minimally adequate teacher returns only "suitable counterexamples" (counterexamples with exactly one incorrect derivation) for every incorrect conjecture, we show that modifications can be done to the learning algorithm given in Section 6 which reduces the number of membership queries executed. Given a suitable counterexample t, the algorithm will construct m new trees of the form $u_i = e_i \# \sigma(s_{i1}, \cdots s_{ik_i})$ for $1 \leq i \leq m$ where $row(s_i) = \delta(\sigma, row(s_{i1}), \cdots, row(s_{ik_i}))), t = (e_0 \# s_0), e_i \in V_{\$}^T, s_i \in (S \bigcup S\Sigma)$ and $s_{i1}, \cdots, s_{ik_i} \in S$. Each e_i replaces the subtree rooted at s_i in t with the terminal \$. We assume that t is a valid derivation tree for the automaton $A(S,E,T)$ but is not derivable in $A(G)$. This implies there exists some derivation step in t of the form $row(s_i) = \delta(\sigma, row(s_{i1}), \cdots, row(s_{ik_i}))$ but $\delta'(\sigma, q_{i1}, \cdots, q_{ik_i}) \neq q_{s_i}$. Starting from the root, the algorithm will seek answers to membership queries u_i in order of increasing depth of node \$ in e_i. Now, $u_0 = t \notin T(A(G))$. Since t is a counterexample, there exists some nodes s_i and $s_{ij} = s_j$ such that $row(s_i) = \delta(\sigma, row(s_{i1}), \cdots, row(s_{il}), row(s_j), row(s_{i,l+1}) \cdots, row(s_{i,k-1}))))$, $u_i = e_i \# s_i \notin T(A(G))$ and $u_j = e_j \# s_j \in T(A(G))$. This condition is shown in Figure 3 in which case we add the tree e_i to E. Since t was a suitable counterexample and s_j was the first node (from root) with this property, the only incorrect derivation in t must have occurred at node s_i. Now for the tree e_i, $T(e_i \# s_i)$ will be 0 ($e_i \# s_i \notin T(A(G))$) but $T(u_i = e_i \# \sigma(s_{i1}, \cdots s_{il}, s_j, s_{i,l+1}, \cdots s_{i,k-1}))$ will be 1 (t is a suitable counterexample). Thus tree e_i will distinguish as yet indistinguishable states s_i and $\sigma(s_{i1}, \cdots, s_{il}, s_j, s_{i,l+1}, \cdots s_{i,k-1})$ increasing the number of elements in S by at least 1.

The number of membership queries can be reduced from $O((n+mn)+l(n+mn+k)^d)n = O(m^d n^{d+1})$ to $O(n + l(n+k)^d)n + mn = O(n^{d+1} + mn)$ where $k = |\Sigma|$, l is the number of distinct ranks of elements in V and d is the largest rank of any element in V. $(n + mn)$ represents the number of elements in S (n elements to ensure table is closed and mn elements from counterexamples). n is the number of elements in E and $l(n + mn + k)^d$ represents the number of elements in $S\Sigma$. In the modified algorithm, for each suitable counterexample generated, we add one element to the set E and at most n counterexamples will be generated. Each time, the table is found to be not closed, a new element is added to S. Hence the table will be found to be not closed at most n times. Further the size of set S will always be less than or equal to n. This bound on the set S is achieved by an additional m membership queries for each counterexample. Since $|S|$ and $|E|$ are bounded by n, the total number of membership queries is $O(n + l(n+k)^d)n + mn = O(n^{d+1} + mn)$. Also, the size of any tree added to E has at most m nodes (size of the counterexample). Proof of lemma 6.4 given in Section 6 must then be modified as was done for the $L*$ algorithm since E is no longer \$-prefix closed with respect to S.

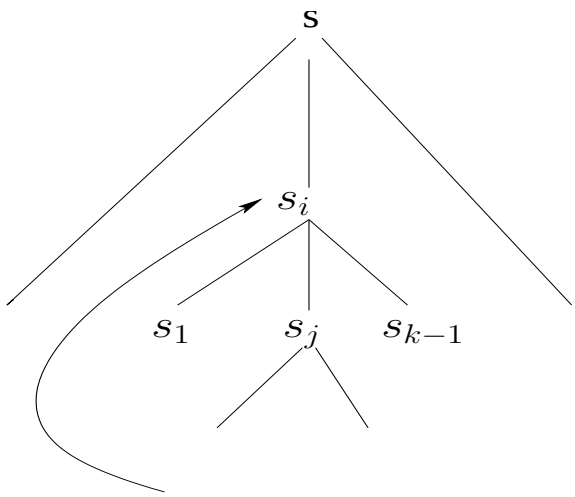

Figure 3: Suitable counterexample for modified learning algorithm

9 Conclusion

In this report, we have summarized results on learning algorithms for regular languages and context-free grammars. We showed that learning a regular language from given data without access to any active experimentation is a NP-complete problem. We then described algorithms for learning regular languages using active and passive experimentation with or without reset. Algorithms for learning permutation automata were then detailed where the oracle is no longer required to answer equivalence queries. Modifications to the learning algorithm for learning context-free languages from skeletal structural descriptions have also been explained. We then show a direct application of regular set inference to symbolic compositional verification of components using assume-guarantee reasoning. We then suggest a new efficient algorithm for learning context-free languages where the teacher is required to provide counterexamples satisfying certain properties.

References

[1] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In *Proceedings of Seventeenth International Conference on Computer aided Verification*, 2005.

[2] Dana Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39:337–350, 1978.

[3] Dana Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51:76–87, 1981.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[5] L. F. Fass. Learning context-free languages from their structured sentences. In *SIGACT News, 15*, pages 24–35, 1983.

[6] Valiant L. G. A theory of the learnable. In *Communications ACM, 27*, pages 1134–1142, 1984.

[7] E. Mark Gold. System identification via state characterization. *Automatica*, 8:621–636, 1972.

[8] E. Mark Gold. Complexity of automata identification from given data. *Information and Control*, 37:302–320, 1978.

[9] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, second edition, 1978.

[10] L. S. Levy and A. K. Joshi. Skeletal structural descriptions. *Information and Control*, 39:192–211, 1978.

[11] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[12] Ronald. L. Rivest and Robert E. Schapire. Diversity-based inference of finite automata. In *Proceedings of Twenty Eighth Annual Symposium on Foundations of Computer Science*, pages 78–87, 1987.

[13] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[14] Yasubumi Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76:223–242, 1990.

[15] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992.

[16] Robert E. Schapire. Diversity-based inference of finite automata. In *Master's Thesis, MIT Lab, Technical Report : MIT/LCS/TR-413*, 1988.