Correctness and Performance of an Incremental Learning Algorithm for Finite Automata

Muddassar A. Sindhu SINDHU@CSC.KTH.SE

Karl Meinke KARLM@NADA.KTH.SE

School of Computer Science and Communication, Royal Institute of Technology, 100-44 Stockholm, Sweden.

Editor: Chun-Nan Hsu and Wee Sun Lee

Abstract

We present a new algorithm *IDS* for incremental learning of deterministic finite automata (DFA). This algorithm is based on the concept of distinguishing sequences introduced in (Angluin, 1981). We give a rigorous proof that two versions of this learning algorithm correctly learn in the limit. Finally we present an empirical performance analysis that compares these two algorithms, focussing on learning times and different types of learning queries. We conclude that *IDS* is an efficient algorithm for software engineering applications of automata learning, such testing and model inference.

1. Introduction

In recent years, automata learning algorithms (aka. regular inference algorithms) have found new applications in software engineering such as *formal verification* (e.g. (Peled et al., 1999), (Clarke et al., 2002), (Luecker, 2006)) *software testing* (e.g. (Raffelt et al., 2008), (Meinke and Sindhu, 2011)) and *model inference* (e.g. (Bohlin and Jonsson, 2008)). These applications mostly centre around learning an abstraction of a complex software system which can then be statically analysed (e.g. by model checking) to determine behavioural correctness. Many of these applications can be improved by the use of learning procedures that are *incremental*.

An automata learning algorithm is incremental if: (i) it constructs a sequence of hypothesis automata $H_0, H_1, ...$ from a sequence of observations $o_0, o_1, ...$ about an unknown target automaton A, and this sequence of hypothesis automata finitely converges to A, and (ii) the construction of hypothesis H_i can reuse aspects of the construction of the previous hypothesis H_{i-1} (such as an equivalence relation on states). The notion of convergence in the limit, as a model of correct incremental learning originates in (Gold, 1967).

Generally speaking, much of the literature on automata learning has focussed on offline learning from a fixed pre-existing data set describing the target automaton. Other approaches, such as (Angluin, 1981) and (Angluin, 1987) have considered online learning, where the data set can be extended by constructing and posing new queries. However, little attention has been paid to incremental learning algorithms, which can be seen as a subclass of online algorithms where serial hypothesis construction using a sequence of increasing data sets is emphasized. The much smaller collection of known incremental algorithms includes the RPNI2 algorithm of (Dupont, 1996), the IID algorithm of (Parekh et al., 1998) and the algorithm of (Porat and Feldman, 1991). However, the motivation for incremental learning from a software engineering perspective is strong, and can be summarised as follows:

1. to analyse a large software system it may not be feasible (or even necessary) to learn the entire automaton model, and

2. the choice of each relevant observation o_i about a large unknown software system often needs to be iteratively guided by analysis of the previous hypothesis model H_{i-1} for efficiency reasons.

Our research into efficient *learning-based testing* (LBT) for software systems (see e.g. (Meinke, 2004), (Meinke and Niu, 2010), (Meinke and Sindhu, 2011)) has led us to investigate the use of distinguishing sequences to design incremental learning algorithms for DFA. Distinguishing sequences offer a rather minimal and flexible way to construct a state space partition, and hence a quotient automaton that represents a hypothesis H about the target DFA to be learned. Distinguishing sequences were first applied to derive the ID online learning algorithm for DFA in (Angluin, 1981).

In this paper, we present a new algorithm *incremental distinguishing sequences* (*IDS*), which uses the distinguishing sequence technique for incremental learning of DFA. In (Meinke and Sindhu, 2011) this algorithm has been successfully applied to learning based testing of reactive systems with demonstrated error discovery rates up to 4000 times faster than using non-incremental learning. Since little seems to have been published about the empirical performance of incremental learning algorithms, we consider this question too.

The structure of the paper is as follows. In Section 2, we review some essential mathematical preliminaries, including a presentation of Angluin's original ID algorithm, which is necessary to understand the correctness proof for *IDS*. In Section 3, we present two different versions of the *IDS* algorithm and prove their correctness. These are called: (1) *prefix free IDS*, and (2) *prefix closed IDS*. In Section 4, we compare the empirical performance of our two *IDS* algorithms with each other. Finally, in Section 5, we present some conclusions and discuss future directions for research.

1.1 Related Work

Distinguishing sequences were first applied to derive the ID online learning algorithm for DFA in (Angluin, 1981). The ID algorithm is not incremental, since only a single hypothesis automaton is ever produced. Later an incremental version IID of this algorithm was presented in (Parekh et al., 1998). Like the IID algorithm, our *IDS* algorithm is incremental. However in contrast with IID, the *IDS* algorithm, and its proof of correctness are much simpler, and some technical errors in (Parekh et al., 1998) are also overcome.

Distinguishing sequences can be contrasted with the complete consistent table approach to partition construction as represented by the well known online learning algorithm L* of (Angluin, 1987). Unlike L*, distinguishing sequences dispose of the need for an equivalence oracle during learning. Instead, we can assume that the observation set P contains a live complete set of input strings (see Section 2.2 below for a technical definition). Furthermore, unlike L* distinguishing sequences do not require a complete table of queries before building the partition relation. In the context of software testing, both of these differences result in a much more efficient learning algorithm. In particular there is greater scope for using online queries that have been generated by other means (such as model checking). Moreover, since LBT is a black-box approach to software testing, then the use of an equivalence oracle contradicts the black-box methodology.

In (Dupont, 1996), an incremental version RPNI2 of the RPNI offline learning algorithm of (Oncina, 1991) and (Lang, 1992) is presented. The RPNI2 algorithm is much more complex than *IDS*. It includes a recursive depth first search of a lexicographically ordered state set with backtracking, and computation of a non-deterministic hypothesis automaton that is subsequently rendered deterministic. These operations have no counterpart in *IDS*. Thus *IDS* is easier to verify and can be quickly and easily implemented in practise. The incremental learning algorithm introduced in (Porat and Feldman, 1991) requires a lexicographic ordering on the presentation of online queries, which is less flexible than *IDS*, and indeed inappropriate for software engineering applications.

2. Preliminaries

2.1 Notations and concepts for DFA

Let Σ be any set of symbols then Σ^* denotes the set of all finite strings over Σ including the empty string λ. The length of a string $\alpha \in \Sigma^*$ is denoted by $|\alpha|$ and $|\lambda| = 0$. For strings $\alpha, \beta \in \Sigma^*$, $\alpha\beta$ denotes their concatenation.

For $\alpha, \beta, \gamma \in \Sigma^*$, if $\alpha = \beta\gamma$ then β is termed a *prefix* of α and γ is termed a *suffix* of α. We let $Pref(\alpha)$ denote the prefix closure of α, i.e. the set of all prefixes of α. We can also apply prefix closure pointwise to any set of strings. The *set difference operation* between two sets U and V denoted by $U - V$ is the set of elements of U which are not members of V. The *symmetric difference* operation defined on pairs of sets is defined by $U \oplus V = (U - V) \cup (V - U)$.

A *deterministic finite automaton* (DFA) is a quintuple $A = \langle \Sigma, Q, F, q_0, \delta \rangle$, where: Σ is the input alphabet, Q is the state set, $F \subseteq Q$ is the set of final states, $q_0 \in Q$ is the initial state and state transition function δ is a mapping $\delta : Q \times \Sigma \to Q$, and $\delta(q_i, b) = q_j$ meaning when in state $q_i \in Q$ given input b the automaton A will move to state $q_j \in Q$ in one step. We extend the function δ to a mapping $\delta^* = Q \times \Sigma^* \to Q$ inductively defined by $\delta = (q, \lambda) = q$ and $\delta^* = (q_i, b_1, ..., b_n) = \delta(\delta^*(q, b_1, ..., b_{n-1}), b_n)$. The *language* $L(A)$ accepted by A is the set of all strings $\alpha \in \Sigma^*$ such that $\delta^*(q_0, \alpha) \in F$. As is well known a language $L \subseteq \Sigma^*$is accepted by DFA if and only if L is *regular*, i.e. L can be defined by a regular grammar. A state $q \in Q$ is said to be *live* if for some string $\alpha \in \Sigma^*$, $\delta^*(q, \alpha) \in F$, otherwise q is said to be *dead.* Given a distinguished *dead state* d_0 we define *string concatenation modulo the dead state* d_0, $f : \Sigma^* \cup \{d_0\} \times \Sigma \to \Sigma^* \cup \{d_0\}$, by $f(d_0, \sigma) = d_0$ and $f(\alpha, \sigma) = \alpha.\sigma$ for $\alpha \in \Sigma^*$. This is function is used for automaton learning in Section 3. Given any DFA A there exists a minimum state DFA A' such that $L(A) = L(A')$ and this automaton is termed the *canonical* DFA for $L(A)$. A canonical DFA has one dead state at the most.

We represent DFA graphically in the usual way using state diagrams. States are represented by small circles labelled by state names and final states among them are marked by concentric double circles. The initial state is represented by attaching a right arrow $\to$ to it. The transitions between the states are represented by directed arrows from state of origin to the destination state. The symbol read from the origin state is attached to the directed arrow as a label. Fig 1 shows state transitions diagrams of two DFAs.

2.2 The *ID* Algorithm

Our *IDS* algorithm is an incremental version of the *ID* learning algorithm introduced in (Angluin, 1981). The ID algorithm is an online learning algorithm for complete learning of a DFA that starts from a given live complete set $P \subseteq \Sigma^*$ of queries about the target automaton, and generates new queries until a state space partition can be constructed. Since the algorithmic ideas and proof of correctness of IDS are based upon those of ID itself, it is useful to review the ID algorithm here. Algorithm 1 presents the ID algorithm. Since this algorithm has been discussed at length in (Angluin, 1981), our own presentation can be brief. A detailed proof of correctness of ID and an analysis of its complexity can be found in (Angluin, 1981).

A finite set $P \subseteq \Sigma^*$ of input strings is said to be *live complete* for a DFA A if for every live state $q \in Q$ there exists a string $\alpha \in P$ such that $\delta^*(q_0, \alpha) = q$. Given a live complete set P for a target automaton A, the essential idea of the ID algorithm is to first construct the set $T' = P \cup \{f(\alpha, b) | (\alpha, b) \in P \times \Sigma\} \cup \{d_0\}$ of all one element extensions of strings in P as a set of state names for the hypothesis automaton. The symbol d_0 is added as a name for the canonical dead state. This set of state names is then iteratively partitioned into sets $E_i(\alpha) \subseteq T'$ for $i = 0, 1, \ldots$ such that elements α, β of T' that denote the same state in A will occur in the same partition set, i.e. $E_i(\alpha) = E_i(\beta)$. This partition refinement can be proven to terminate and the resulting collection of sets forms a congruence on T'. Finally the ID algorithm constructs the hypothesis automaton as the resulting quotient automaton. The method used to refine the partition set is to iteratively construct

a set V of *distinguishing strings*, such that no two distinct states of A have the same behaviour on all of V.

We will present the ID and *IDS* algorithms so that similar variables share the same names. This pedagogic device emphasises similarity in the behaviour of both algorithms. However, there are also important differences in behaviour. Thus, when analysing the behavioural properties of program variables we will carefully distinguish their context as e.g. v_n^{ID}, $E_n^{ID}(\alpha), \ldots$, and v_n^{IDS}, $E_n^{IDS}(\alpha), \ldots$ etc. Our proof of correctness for *IDS* will show how the learning behaviour of *IDS* on a sequence of input strings $s_1, \ldots, s_n \in \Sigma^*$ can be simulated by the behaviour of ID on the corresponding set of inputs $\{s_1, \ldots s_n\}$. Once this is established, one can apply the known correctness of ID to establish the correctness of *IDS*.

2.3 Behavioural differences between IID and IDS

The IID algorithm of (Parekh et al., 1998) also presents a simulation method for ID. However following points of difference in behaviour of IID and *IDS* are worth mentioning:

1. IID starts from a null DFA as hypothesis while *IDS* constructs the initial hypothesis after reading all $\sigma \in \Sigma$ from the initial state.
2. IID discards all negative examples and waits for the first positive example after the construction of the initial (null) hypothesis to do further construction. *IDS* on the other hand does construction with all negative and positive examples after building an initial hypothesis which makes it more useful for practical software engineering applications identified in Section 1.
3. IID in some cases builds hypotheses which have a partially defined transition function δ rather than being *left total*. This will be shown with an example in the next section. *IDS* fixes this problem due to lines 10-13 of Algorithm 4 described in this paper.
4. Unlike *IDS*, there is no prefix free version of IID.
5. In addition to the above it is easily shown that IID does not satisfy our Simulation Theorem 5, and thus the two algorithms are quite different. The behavioural properties of ID that are needed to complete this correctness proof can be stated as follows.

Theorem 1 *(i) Let $P \subseteq \Sigma^*$ be a live complete set for a DFA A containing λ. Then given P and A as input, the ID algorithm terminates and the automaton M returned is the canonical automaton for $L(A)$.*
(ii) Let $l \in \mathbb{N}$ be the maximum value of program variable i^{ID} given P and A. For all $0 \leq n \leq l$ and for all $\alpha \in T$,

$$E_n^{ID}(\alpha) = \{v_j^{ID} \mid 0 \leq j \leq n,\ \alpha v_j^{ID} \in L(A)\}.$$

Proof. *(i) See (Angluin, 1981) Theorem 3. (ii) By induction on n.*

2.4 The *unclosed* table problem of IID

One difference between ID and IID is the frequency of hypothesis automaton construction. With ID this occurs just once, after a single partition refinement is completed. However, with IID this occurs regularly, after each partition refinement. This difference means that the automaton construction algorithm (Algorithm 1, lines 28-37) used for ID can no longer be used for IID, (as asserted in (Parekh et al., 1998)) as we show below. Suppose we want to learn the automaton A shown in Fig 1.(a).

A positive example for this automaton is $(b, +)$. If we use it to start learning A using the IID algorithm of (Parekh et al., 1998), we have $P_0 = Pref(b) = \{b, \lambda\}$ and $P_0' = \{b, \lambda\} \cup \{d_0\} = \{b, \lambda, d_0\}$. We then obtain the sets $T_0 = P_0 \cup \{f(\alpha, b) | (\alpha, b) \in P_0 \times \Sigma\} = \{b, \lambda\} \cup \{b, \lambda\} \times \{a, b\} = \{\lambda, a, b, ba, bb\}$ and $T_0' = T_0 \cup \{d_0\} = \{d_0, \lambda, a, b, ba, bb\}$. The initial column for the table of partition sets is constructed as shown in Table 1 for $i = 0$ and $v_0 = \lambda$. From this column we see that two elements of the set P_0' have the same value. i.e $E(d_0) = E(\lambda)$ but $E(f(d_0, b)) \neq E(f(\lambda, b))$. Therefore we have $\gamma \in E(f(d_0, b)) \oplus E(f(\lambda, b))$ or $\gamma \in \emptyset \oplus \{\lambda\}$. We can choose $\gamma = \lambda$ which gives the distinguishing

Algorithm 1 ID Learning Algorithm

Input: A live complete set $P \subseteq \Sigma^*$ and a teacher DFA A to answer membership queries $\alpha \in L(A)$?
Output: A DFA M equivalent to the target DFA A.

1. **begin**
2. //Perform Initialization
3. $i = 0$, $v_i = \lambda$, $V = \{v_i\}$
4. $P' = P \cup \{d_0\}$, $T = P \cup \{f(\alpha, b) | (\alpha, b) \in P \times \Sigma\}$, $T' = T \cup \{d_0\}$
5. Construct function E_0 for $v_0 = \lambda$,
6. $E_0(d_0) = \emptyset$
7. $\forall \alpha \in T$
8. { pose the membership query "$\alpha \in L(A)$?"
9. **if** the teacher's response is yes
10. **then** $E_0(\alpha) = \{\lambda\}$
11. **else** $E_0(\alpha) = \emptyset$
12. **end if**
13. }
14. //Refine the partition of the set T'
15. **while** ($\exists \alpha, \beta \in P'$ and $b \in \Sigma$ such that $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$)
16. **do**
17. Let $\gamma \in E_i(f(\alpha, b)) \oplus E_i(f(\beta, b))$
18. $v_{i+1} = b\gamma$
19. $V = V \cup \{v_{i+1}\}$, $i = i + 1$
20. $\forall \alpha \in T_k$ pose the membership query "$\alpha v_i \in L(A)$?"
21. {
22. **if** the teacher's response is yes
23. **then** $E_i(\alpha) = E_{i-1}(\alpha) \cup \{v_i\}$
24. **else** $E_i(\alpha) = E_{i-1}(\alpha)$
25. **end if**
26. }
27. **end while**
28. //Construct the representation M of the target DFA A.
29. The states of M are the sets $E_i(\alpha)$, where $\alpha \in T$
30. The initial state q_0 is the set $E_i(\lambda)$
31. The accepting states are the sets $E_i(\alpha)$ where $\alpha \in T$ and $\lambda \in E_i(\alpha)$
32. The transitions of M are defined as follows:
33. $\forall \alpha \in P'$
34. **if** $E_i(\alpha) = \emptyset$
35. **then** add self loops on the state $E_i(\alpha)$ for all $b \in \Sigma$
36. **else** $\forall b \in \Sigma$ set the transition $\delta(E_i(\alpha), b) = E_i(f(\alpha, b))$
37. **end if**
38. **end.**

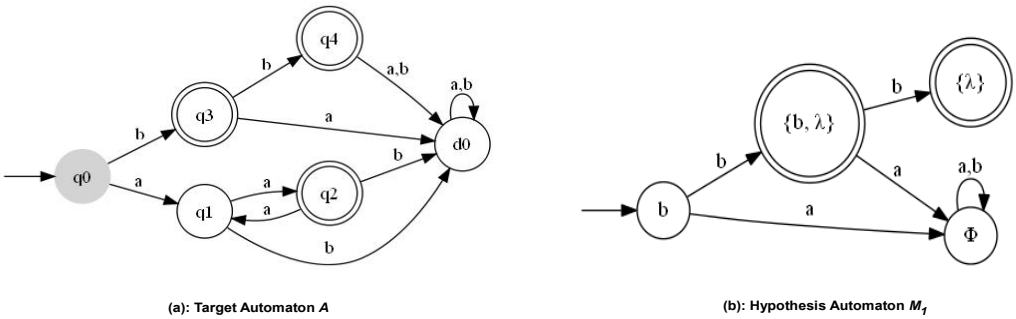

Figure 1: Target and Hypothesis Automata

i	0	1
v_i	λ	b
$E(d_0)$	$\emptyset$	$\emptyset$
$E(\lambda)$	$\emptyset$	$\{b\}$
$E(a)$	$\emptyset$	$\emptyset$
$E(b)$	$\{\lambda\}$	$\{\lambda, b\}$
$E(ba)$	$\emptyset$	$\emptyset$
$E(bb)$	$\{\lambda\}$	$\{\lambda\}$

Table 1: For $(b, +)$

string $v_1 = b\gamma = b\lambda = b$. We then extend Table 1 for $i = 1$. Now we can see that all elements of the set P'_0, which are b, λ and d_0, have distinct values in the last column of the table and so no further refinement of the partition is possible. At this stage IID constructs the next hypothesis automaton M_1 as shown in Figure 1.(b).

Now we observe that the transition function δ for this automaton is only partially defined, since there are no outgoing transitions for the state named $\{\lambda\}$. Therefore the IID algorithm of (Parekh et al., 1998) does not always generate a hypothesis automaton with well defined transition function δ. This created problems when we tried to use this algorithm for practical software engineering applications identified in Section 1, e.g. a model checker when used to verify this kind of a hypothesis can get stuck in such a state with no outgoing transitions. Similarly, an automata equivalence checker which is used to terminate a learning-based testing process goes into an infinite loop because of such a state since it will never find its equivalent state in the target automaton.

The problem seems to stem from an *unclosed* table in some executions of IID. The notion of *closed* and *consistent* observation table is given in (Angluin, 1987) for L* algorithm. The L* algorithm also incrementally builds the observation table but keeps asking queries until it becomes *closed* and *consistent*. In this case the table will be closed when $\forall \beta \in T \setminus P' \; \exists \alpha \in P'$ such that $E_i(\alpha) = E_i(\beta)$. If $E_i(\alpha) \neq E_i(\beta)$ as in the above example where $E_1(bb) = \{\lambda\}$ is not equal to any of $E_1(d_0) = \emptyset, E_1(\lambda) = \{b\}$ *or* $E_1(b) = \{\lambda, b\}$ then the solution in (Angluin, 1987) is to move $\beta \in T \setminus P'$ to set P' and ask more queries to rebuild the congruence that might have been affected by this last addition to set P'. However L* is a complete learning algorithm and it only outputs the description of the hypothesis automaton after learning it completely. Therefore for incremental learning the simplest fix is to set the transitions of such states to the dead state as done in lines 10-13 of Algorithm 4. This doesn't require any new entries or shuffling the previous entries up in the table thus keeping the congruence intact.

3. Correctness of *IDS* Algorithm

In this section we present our *IDS* incremental learning algorithm for DFA. In fact, we consider two versions of this algorithm, with and without prefix closure of the set of input strings. We then give a rigorous proof that both algorithms correctly learn an unknown DFA in the limit in the sense of (Gold, 1967).

In Algorithm 2 we present the main *IDS* algorithm, and in Algorithms 3 and 4 we give its auxiliary algorithms for iterative partition refinement and automaton construction respectively.

The version of the *IDS* algorithm which appears in Algorithm 2 we term the *prefix free IDS* algorithm, due to lines 22 and 25. Notice that lines 23 and 26 of Algorithm 2 have been commented out. When these latter two lines are uncommented and instead lines 22 and 25 are commented out, we obtain a version of the *IDS* algorithm that we term *prefix closed IDS*. We will prove that both prefix closed and prefix free *IDS* learn correctly in the limit. However, in Section 4 we will show that they have quite different performance characteristics in a way that can be expected to influence applications.

We will prove the correctness of the prefix free *IDS* algorithm first, since this proof is somewhat simpler, while the essential proof principles can also be applied to verify the prefix closed *IDS* algorithm. We begin an analysis of the correctness of prefix free *IDS* by confirming that the construction of hypothesis automata carried out by Algorithm 4 is well defined.

Proposition 2 *For each $t \geq 0$ the hypothesis automaton M_t constructed by the automaton construction Algorithm 4 after t input strings have been observed is a well defined DFA.*

Proof The main task is to show δ to be well defined function and uniquely defined for every state $E_i(\alpha)$, where $\alpha \in T_k$.

Proposition 2 establishes that Algorithm 2 will generate a sequence of well defined DFA. However, to show that this algorithm learns correctly, we must prove that this sequence of automata converges to the target automaton A given sufficient information about A. It will suffice to show that the behaviour of prefix free *IDS* can be simulated by the behaviour of ID, since ID is known to learn correctly given a live complete set of input strings (c.f. Theorem 1.(i)). The first step in this proof is to show that the sequences of sets of state names P_k^{IDS} and T_k^{IDS} generated by prefix free IDS converge to the sets P^{ID} and T^{ID} of ID. ■

Proposition 3 *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of input strings $s_i \in \Sigma^*$ for prefix free IDS and let $P^{ID} = \{\lambda, s_1, \ldots, s_l\}$ be the corresponding input set for ID.*

(i) For all $0 \leq k \leq l$, $P_k^{IDS} = \{\lambda, s_1, \ldots, s_k\} \subseteq P^{ID}$.
(ii) For all $0 \leq k \leq l$, $T_k^{IDS} = P_k^{IDS} \cup \{f(\alpha, b) | \alpha \in P_k^{IDS}, b \in \Sigma\} \subseteq T^{ID}$.
(iii) $P_l^{IDS} = P^{ID}$ and $T_l^{IDS} = T^{ID}$.

Proof Clearly (iii) follows from (i) and (ii). We prove (i) and (ii) by induction on k. ■

Next we turn our attention to proving some fundamental loop invariants for Algorithm 2. Since this algorithm in turn calls the partition refinement Algorithm 3 then we have in effect a doubly nested loop structure to analyse. Clearly the two indexing counters k^{IDS} and i^{IDS} (in the outer and inner loops respectively) both increase on each iteration. However, the relationship between these two variables is not easily defined. Nevertheless, since both variables increase from an initial value of zero, we can assume the existence of a monotone re-indexing function that captures their relationship.

Definition 4 *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$. The re-indexing function $\mathbb{K}^S : \mathbb{N} \to \mathbb{N}$* for prefix free IDS on input S is the unique monotonically increasing function such that for each $n \in \mathbb{N}$, $\mathbb{K}^S(n)$ is the least integer m such that program variable k^{IDS} has value

Algorithm 2 IDS Learning Algorithm

Input: A file $S = s_1, \ldots, s_l$ of input strings $s_i \in \Sigma^*$ and a teacher DFA A to answer membership queries $\alpha \in L(A)$?
Output: A sequence of DFA M_t for $t = 0, \ldots, l$ as well as the total number of membership queries and book keeping queries asked by the learner.

1. **begin**
2. //Perform Initialization
3. $i = 0$, $k = 0$, $t = 0$, $v_i = \lambda$, $V = \{v_i\}$
4. //Process the empty string
5. $P_0 = \{\lambda\}$, $P_0' = P_0 \cup \{d_0\}$, $T_0 = P_0 \cup \Sigma$
6. $E_0(d_0) = \emptyset$
7. $\forall \alpha \in T_0$ {
8. pose the membership query "$\alpha \in L(A)$?", $bquery = bquery + 1$
9. **if** the teacher's response is yes
10. **then** $E_0(\alpha) = \{\lambda\}$
11. **else** $E_0(\alpha) = \emptyset$
12. }
13. //Refine the partition of set T_0 as described in Algorithm 3
14. //Construct the current representation M_0 of the target DFA
15. //as described in Algorithm 4.
16.
17. //Process the file of examples.
18. **while** $S \neq empty$ **do**
19. read(S, α)
20. mquery = mquery +1
21. k = k+1, t = t+1
22. $P_k = P_{k-1} \cup \{\alpha\}$
23. // $P_k = P_{k-1} \cup Pref(\alpha)$ //prefix closure
24. $P_k' = P_k \cup \{d_0\}$
25. $T_k = T_{k-1} \cup \{\alpha\} \cup \{f(\alpha, b) | b \in \Sigma\}$
26. // $T_k = P_k \cup \{f(\alpha, b) | \alpha \in P_k - P_{k-1}, b \in \Sigma\}$ //prefix closure
27. $T_k' = T_k \cup \{d_0\}$
28. $\forall \alpha \in T_k - T_{k-1}$
29. {
30. // Fill in the values of $E_i(\alpha)$ using membership queries:
31. $E_i(\alpha) = \{v_j | 0 \leq j \leq i, \alpha v_j \in L(A)\}$
32. $bquery = bquery + i$
33. }
34. // Refine the partition of the set T_k
35. **if** α is consistent with M_{t-1}
36. **then** $M_t = M_{t-1}$
37. **else** construct M_t as described in Algorithm 4.
38. }
39. **end.**

Algorithm 3 Refine Partition

while ($\exists \alpha, \beta \in P'_k$ and $b \in \Sigma$ such that $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$)
do
    Let $\gamma \in E_i(f(\alpha, b)) \oplus E_i(f(\beta, b))$
    $v_{i+1} = b\gamma$
    $V = V \cup \{v_{i+1}\}$, $i = i + 1$
    $\forall \alpha \in T_k$ pose the membership query "$\alpha v_i \in L(A)$?"
        {
        $bquery = bquery + 1$
        if the teacher's response is *yes*
        then $E_i(\alpha) = E_{i-1}(\alpha) \cup \{v_i\}$
        else $E_i(\alpha) = E_{i-1}(\alpha)$
        end if
        }
end while

Algorithm 4 Automata Construction

The states of M_t are the sets $E_i(\alpha)$, where $\alpha \in T_k$
The initial state q_0 is the set $E_i(\lambda)$
The accepting states are the sets $E_i(\alpha)$ where $\alpha \in T_k$ and $\lambda \in E_i(\alpha)$
The transitions of M_t are defined as follows:
  $\forall \alpha \in P'_k$
    if $E_i(\alpha) = \emptyset$
    then add self loops on the state $E_i(\alpha)$ for all $b \in \Sigma$
    else $\forall b \in \Sigma$ set the transition $\delta(E_i(\alpha), b) = E_i(f(\alpha, b))$
    end if
  $\forall \beta \in T_k - P'_k$
    if $\forall \alpha \in P'_k$ $E_i(\beta) \neq E_i(\alpha)$ **and** $E_i(\beta) \neq \emptyset$
    then $\forall b \in \Sigma$ set the transition $\delta(E_i(\beta), b) = \emptyset$
    end if

m while the program variable i^{IDS} has value n. Thus, for example, $\mathbb{K}^S(0) = 0$. When S is clear from the context, we may write $\mathbb{K}$ *for* $\mathbb{K}^S$.

With the help of such re-indexing functions we can express important invariant properties of the key program variables v_j^{IDS} and $E_n^{IDS}(\alpha)$, and via Proposition 3 their relationship to v_j^{ID} and $E_n^{ID}(\alpha)$. Corresponding to the doubly nested loop structure of Algorithm 2, the proof of Theorem 5 below makes use of a doubly nested induction argument.

Theorem 5 (Simulation Theorem) *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$. For any execution of prefix free IDS on S there exists an execution of ID on $\{\lambda, s_1, \ldots, s_l\}$ such that for all $m \geq 0$:*

(i) For all $n \geq 0$ if $\mathbb{K}(n) = m$ then:
(a) for all $0 \leq j \leq n$, $v_j^{IDS} = v_j^{ID}$,
(b) for all $0 \leq j < n$, $v_n^{IDS} \neq v_j^{IDS}$,
(c) for all $\alpha \in T_m^{IDS}$, $E_n^{IDS}(\alpha) = \{v_j^{IDS} | 0 \leq j \leq n, \alpha v_j^{IDS} \in L(A)\}$.
(ii) If $m > 0$ then let $p \in \mathbb{N}$ be the greatest integer such that $\mathbb{K}(p) = m - 1$. Then for all $\alpha \in T_m^{IDS}$, $E_p^{IDS}(\alpha) = \{v_j^{IDS} | 0 \leq j \leq p, \alpha v_j^{IDS} \in L(A)\}$.
(iii) The mth partition refinement of IDS terminates.

Proof By induction on m using Proposition 3(i). ■

Notice that in the statement of Theorem 5 above, since both ID and *IDS* are non-deterministic algorithms (due to the non-deterministic choice on line 17 of Algorithm 1 and line 3 of Algorithm 3), then we can only talk about the existence of some correct simulation. Clearly there are also simulations of *IDS* by ID which are not correct, but this does not affect the basic correctness argument.

Corollary 6 *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$. Any execution of prefix free IDS on S terminates with the program variable k^{IDS} having value l.*

Proof Follows from Simulation Theorem 5.(iii) since clearly the while loop of Algorithm 2 terminates when the input sequence S is empty. ■

Using the detailed analysis of the invariant properties of the program variables P_k^{IDS} and T_k^{IDS} in Proposition 3 and v_j^{IDS} and $E_n^{IDS}(\alpha)$ in Simulation Theorem 5 it is now a simple matter to establish correctness of learning for the prefix free *IDS* Algorithm.

Theorem 7 (Correctness Theorem) *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$ such that $\{\lambda, s_1, \ldots, s_l\}$ is a live complete set for a DFA A. Then prefix free IDS terminates on S and the hypothesis automaton M_l^{IDS} is a canonical representation of A.*

Proof By Corollary 6, prefix free *IDS* terminates on S with the variable k^{IDS} having value l. By Simulation Theorem 5.(i) and Theorem 1.(ii), there exists an execution of ID on $\{\lambda, s_1, \ldots, s_l\}$ such that $E_n^{IDS}(\alpha) = E_n^{ID}(\alpha)$ for all $\alpha \in T_l^{IDS}$ and any n such that $\mathbb{K}(n) = l$. By Proposition 3.(iii), $T_l^{IDS} = T^{ID}$ and $P_l'^{IDS} = P'^{ID}$. So letting M^{ID} be the canonical representation of A constructed by ID using $\{\lambda, s_1, \ldots, s_l\}$ then M^{ID} and M_l^{IDS} have the same state sets, initial states, accepting states and transitions. ■

Our next result confirms that the hypothesis automaton M_t^{IDS} generated after t input strings have been read is consistent with all currently known observations about the target automaton. This is quite straightforward in the light of Simulation Theorem 5.

Theorem 8 (Compatibility Theorem) *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$. For each $0 \leq t \leq l$, M_t^{IDS} is compatible with A on $\{\lambda, s_1, \ldots, s_t\}$.*

Proof By definition, M_t^{IDS} is compatible with A on $\{\lambda, s_1, \ldots, s_t\}$ if, and only if, for each $0 \leq j \leq t$, $s_j \in L(A) \Leftrightarrow \lambda \in E_{i_t}^{IDS}(s_j)$, where i_t is the greatest integer such that $\mathbb{K}(i_t) = t$ and the sets $E_{i_t}^{IDS}(\alpha)$ for $\alpha \in T_t^{IDS}$ are the states of M_t^{IDS}. Now $v_0^{IDS} = \lambda$. So by Simulation Theorem 5.(i).(c), if $s_j \in L(A)$ then $s_j v_0^{IDS} \in L(A)$ so $v_0^{IDS} \in E_{i_t}^{IDS}(s_j)$, i.e. $\lambda \in E_{i_t}^{IDS}(s_j)$, and if $s_j \notin L(A)$ then $s_j v_0^{IDS} \notin L(A)$ so $v_0^{IDS} \notin E_{i_t}^{IDS}(s_j)$, i.e. $\lambda \notin E_{i_t}^{IDS}(s_j)$. ■

Let us briefly consider the correctness of prefix closed *IDS*. We begin by observing that the non-sequential ID Algorithm 1 does not compute any prefix closure of input strings. Therefore, Proposition 3 does not hold for prefix closed *IDS*. In order to obtain a simulation between prefix closed *IDS* and ID we modify Proposition 3 to the following.

Proposition 9 *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of input strings $s_i \in \Sigma^*$ for prefix closed IDS and let $P^{ID} = Pref(\{\lambda, s_1, \ldots, s_l\})$ be the corresponding input set for ID.*

(i) For all $0 \leq k \leq l$, $P_k^{IDS} = Pref(\{\lambda, s_1, \ldots, s_k\}) \subseteq P^{ID}$.
(ii) For all $0 \leq k \leq l$, $T_k^{IDS} = P_k^{IDS} \cup \{f(\alpha, b) | \alpha \in P_k^{IID}, b \in \Sigma\} \subseteq T^{ID}$.
(iii) $P_l^{IDS} = P^{ID}$ and $T_l^{IDS} = T^{ID}$
Proof Similar to the proof of Proposition 3. ■

Theorem 10 (Correctness Theorem) *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$ such that $\{\lambda, s_1, \ldots, s_l\}$ is a live complete set for a DFA A. Then prefix closed IDS terminates on S and the hypothesis automaton M_l^{IDS} is a canonical representation of A.*

Proof Exercise, following the proof of Theorem 7. ■

4. Empirical Performance Analysis

Little seems to have been published about the empirical performance and average time complexity of incremental learning algorithms for DFA in the literature. By the average time complexity of the algorithm we mean the average number of queries needed to completely learn a DFA of a given state space size. This question can be answered experimentally by randomly generating a large number of DFA with a given state space size, and randomly generating a sequence of query strings for each such DFA.

From the point of view of software engineering applications such as testing and model inference, we have found that it is important to distinguish between the two types of queries about the target automaton that are used by *IDS* during the learning procedure. On the one, hand the algorithm uses internally generated queries (we call these *book-keeping queries*) and on the other hand it uses queries that are supplied externally by the input file (we call these *membership queries*). From a software engineering applications viewpoint it seems important that the ratio of book-keeping to membership queries should be low. This allows membership queries to have the maximum influence in steering the learning process externally. The average query complexity of the *IDS* algorithm with respect to the numbers of book-keeping and membership queries needed for complete learning can also be measured by random generation of DFA and query strings. To measure each query type, Algorithm 2 has been instrumented with two integer variables *bquery* and *mquery* intended to track the total number of each type of query used during learning (lines 8, 20 and 32).

Since two variants of the *IDS* algorithm were identified, with and without prefix closure of input strings, it was interesting to compare the performance of each of these two variants according to the above two average complexity measures.

4.1 Experimental Procedure

To empirically measure the average time and query complexity of our two *IDS* algorithms, two experiments were set up. These measured:

(1) the average computation time needed to learn a randomly generated DFA (of a given state space size) using randomly generated membership queries, and

(2) the total number of membership and book-keeping queries needed to learn a randomly generated DFA (of a given state space size) using randomly generated membership queries.

We chose randomly generated DFA with state space sizes varying between 5 and 50 states, and an equiprobable distribution of transitions between states. No filtering was applied to remove dead states, so the average effective state space size was therefore somewhat smaller than the nominal state .

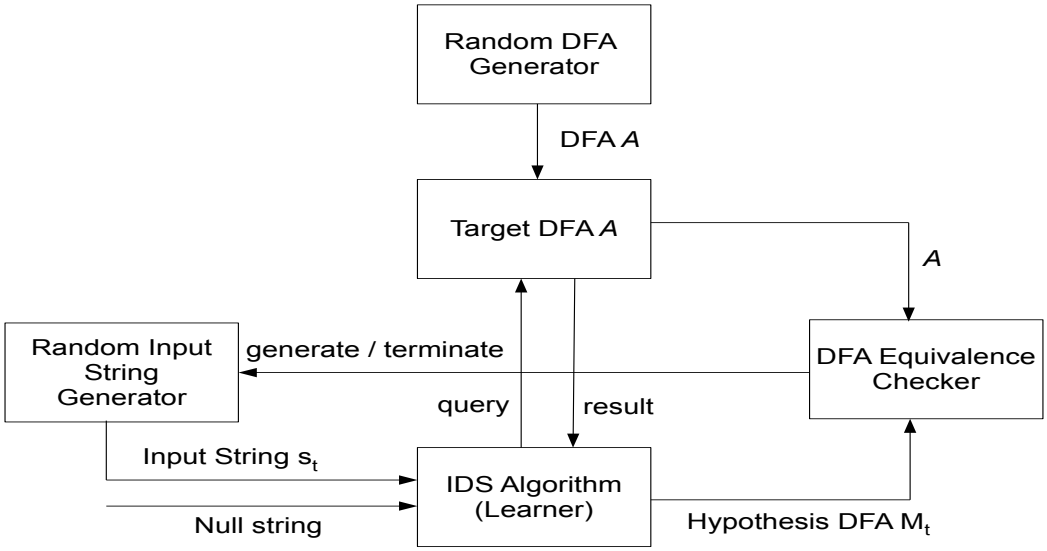

Figure 2: Evaluation Framework

The experimental setup consisted of the following components:
(1) a random input string generator,
(2) a random DFA generator,
(3) an instance of the *IDS* Algorithm (prefix free or prefix closed) ,
(4) an automaton equivalence checker.

The architecture of our evaluation framework and the flow of data between these components are illustrated in Figure 1.

The random input string generator constructed strings over the set of alphabet Σ of the target automaton and the length of the generated strings was always $\leq |Q|$ of the target automaton. Since *IDS* begins learning by reading the null string, therefore, null string was only provided externally and wasn't generated randomly to avoid unnecessary repetition. The random DFA generator started by building a specific sized state set Q. The number of final states $|F| \leq |Q|$ was chosen randomly and then these final states were again marked randomly from the state set Q. The initial state was also chosen randomly from the state set. Similarly the transition function δ was constructed by randomly assigning next states from set Q for each state $q \in Q$ after reading each alphabet $\sigma \in \Sigma$. The *IDS* algorithms and the entire evaluation framework were implemented in Java. The

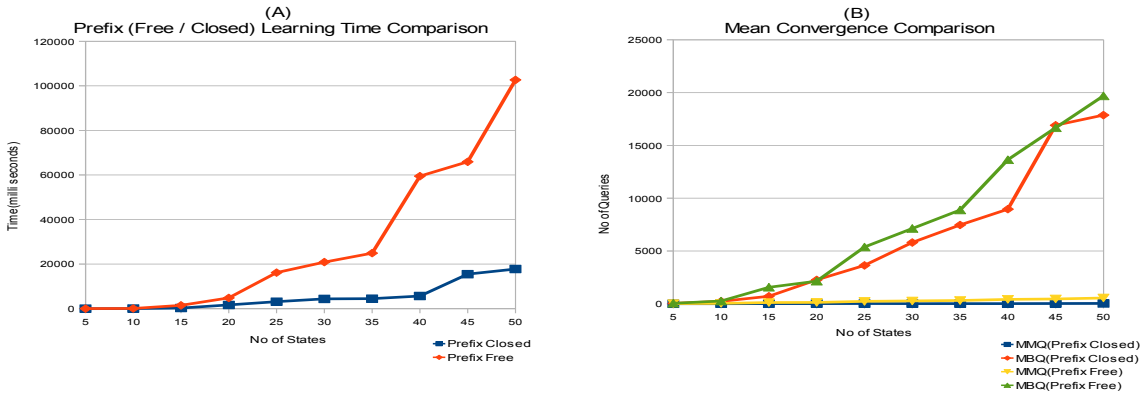

Figure 3: Average Time Complexity

performance of the input string and DFA generators is dependent on Java's Random class which generates pseudorandom numbers that depend upon a specific seed. To minimize the chance of generating the same pseudo random strings/automata again the seed was set to the system clock where ever possible.

The purpose of the equivalence checker was to terminate the learning procedure as soon as the hypothesis automaton sequence had successfully converged to the target automaton. There are several well known equivalence checking algorithms described in literature. These have runtime complexity ranging from quadratic to nearly linear execution times. We chose an algorithm with nearly linear time performance described in (Hopcroft and Karp, 1971). This was to minimise the overhead of equivalence checking in the overall computation time.

Ten different automata were generated randomly for each state size (ranging between 5 and 50). They were learned by both prefix free and prefix closed *IDS* and their learning times (in milliseconds), number of book keeping queries and membership queries asked to reach the target were recorded. The graphs in Figure 3 show a mean of ten experiments for all these values for both variants of *IDS*.

4.2 Results and Interpretation

The graphs in Figures 3.A and 3.B illustrate the outcome of our experiments to measure the average time and average query complexity of both *IDS* algorithms, as described in Section 4.1.

Figure 2.A presents the results of estimating the average learning time for the prefix free and prefix closed *IDS* algorithms as a function of the state space size of the target DFA. For large state space sizes $|Q|$, the data sets of randomly generated target DFA represent only a small fraction of all possible such DFA of size $|Q|$. Therefore the two data curves are not smooth for large state space sizes. Nevertheless, there is sufficient data to identify some clear trends. The average learning time for prefix free *IDS* learning is substantially greater than corresponding time for prefix closed *IDS*, and this discrepancy increases with state space size. The reason would appear to be that prefix free *IDS* throws away data about the target DFA that must be regenerated randomly (since input string queries are generated at random). The average time complexity for prefix free *IDS* learning seems to

grow approximately quadratically, while the average time complexity for prefix closed *IDS* learning appears to grow almost linearly within the given data range. From this viewpoint, prefix-closed *IDS* appears to be the superior algorithm.

Figure 2.B presents the results of estimating the average number of membership queries and book-keeping queries as a function of the state space size of the target DFA. Again, we have compared prefix-closed with prefix free *IDS* learning. Allowance must also be made for the small data set sizes for large state space values. We can see that membership queries grow approximately linearly with the increase in state space size, while book-keeping queries grow approximately quadratically, at least within the data ranges that we considered. There appears to be a small but significant decrease in the number of both book-keeping and membership queries used by the prefix-closed *IDS* algorithm. The reason for this appears to be similar to the issues identified for average time complexity. Prefix closure seems to be an efficient way to gather data about the target DFA. From the viewpoint of software engineering applications discussed in Section 1, now prefix free *IDS* appears to be preferable. This is because the decreasing ratio of book-keeping to membership queries improves the possibility to direct the learning process using externally generated queries (e.g. from a model checker).

5. Conclusions

We have presented two versions of the *IDS* algorithm which is an incremental algorithm for learning DFA in polynomial time. We have given a rigorous proof that both algorithms correctly learn in the limit. Finally we have presented the results of an empirical study of the average time and query complexity *IDS*. These empirical results suggest that *IDS* algorithm is well suited to applications in software engineering, where an incremental approach that allows externally generated online queries is needed. This conclusion is further supported in (Meinke and Sindhu, 2010) where we have evaluated the *IDS* algorithm for learning based testing of reactive systems, and shown that it leads to error discovery up to 4000 times faster than using non-incremental learning.

We gratefully acknowledge financial support for this research from the Higher Education Commission (HEC) of Pakistan, the Swedish Research Council (VR) and the European Union under project HATS FP7-231620.

References

D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, October 1981.

D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(1):87–106, November 1987.

T. Bohlin and B. Jonsson. Regular inference for communication protocol entities. Technical Report 2008-024, Dept. of Information Technology, Uppsala University, 2008.

E. Clarke, A. Gupta, J. Kukula, and O. Strichman. Sat-based abstraction refinement using ilp and machine learning. In *Proc. 21st International Conference On Computer Aided Verification (CAV'02)*, 2002.

P. Dupont. Incremental regular inference. In *Proceedings of the Third ICGI-96*, number 1147 in LNAI, 1996.

E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report TR 71-114, Cornell University, 1971. URL `http://hdl.handle.net/1813/5958`.

K. J. Lang. Random dfa's can be approximately learned from sparse uniform examples. In *Proceedings of the fifth annual workshop on Computational learning theory*, COLT '92, pages 45–52, New York, NY, USA, 1992. ACM. ISBN 0-89791-497-X. doi: http://doi.acm.org/10.1145/130385.130390. URL `http://doi.acm.org/10.1145/130385.130390`.

M. Luecker. Learning meets verification. In F. S. de Boer et al., editor, *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 127–151. Springer, 2006.

K. Meinke. Automated black-box testing of functional correctness using function approximation. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 143–153, New York, NY, USA, 2004. ACM. ISBN 1-58113-820-2. doi: http://doi.acm.org/10.1145/1007512.1007532.

K. Meinke and F. Niu. A learning-based approach to unit testing of numerical software. In *Proc. Twenty Second IFIP Int. Conf. on Testing Software and Systems (ICTSS 2010)*, number 6435 in LNCS, pages 221–235. Springer, 2010.

K. Meinke and M. Sindhu. Correctness and performance of an incremental learning algorithm for kripke structures. Technical report, School of Computer Science and Communication, Royal Institute of Technology, Stockholm, 2010.

K. Meinke and M. A. Sindhu. Incremental learning-based testing for reactive systems. In M. Gogolla and B. Wolff, editors, *Tests and Proofs*, volume 6706 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2011. ISBN 978-3-642-21767-8.

P. Oncina, J.; García. *Inferring regular languages in polynomial update time*, chapter -. World Scientific Publishing, 1991.

R. Parekh, C. Nichitiu, and V. Honavar. A polynomial time incremental algorithm for regular grammar inference. In *Proc. Fourth Int. Colloq. on Grammatical Inference (ICGI 98)*, LNAI. Springer, 1998.

D. Peled, M. Vardi, and M. Yannakakis. Black-box checking. In *Formal Methods for Protocol Engineering and Distributed Systems FORTE/PSTV*, pages 225–240. Kluwer, 1999.

S. Porat and J. A. Feldman. Learning automata from ordered examples. *Mach. Learn.*, 7:109–138, September 1991. ISSN 0885-6125. doi: 10.1007/BF00114841. URL `http://portal.acm.org/citation.cfm?id=125342.125343`.

H. Raffelt, B. Steffen, and T. Margaria. Dynamic testing via automata learning. In *Hardware and Software: Verification and Testing*, number 4899 in LNCS, pages 136–152. Springer, 2008.