Algorithms for Learning Finite Automata from Queries: A Unified View

José L. Balcázar *
Josep Díaz *
Ricard Gavaldà *
Osamu Watanabe †

September 13th, 1996

Abstract

In this survey we compare several known variants of the algorithm for learning deterministic finite automata via membership and equivalence queries. We believe that our presentation makes it easier to understand what is going on and what the differences between the various algorithms mean. We also include the comparative analysis of the algorithms, review some known lower bounds, prove a new one, and discuss the question of parallelizing this sort of algorithms.

1 Introduction

One of the main positive results in the area of computational learning is the fact that deterministic finite automata (dfa) can be learned from membership

*Department of Software (LSI), Universitat Politècnica de Catalunya. Pau Gargallo 5, E-08028 Barcelona, Spain. E-mail: {balqui,diaz,gavalda}@lsi.upc.es. Partially supported by the E.U. through the ESPRIT Long Term Research Project ALCOM IT (nr. 20244), by the ESPRIT Working Group NeuroCOLT (nr. 8556), and by DGICYT, project number PB92–0709.

†Department of Computer Science, Tokyo Institute of Technology, Meguro-ku Ookayama, Tokyo 152, Japan. E-mail: watanabe@cs.titech.ac.jp.

and equivalence queries. There are by now several versions of the algorithm available [Ang87, RS93, KV94].

The most recent version [KV94] provides ample additional intuitions that are rather hard to grasp in the previous works. In [BDGW92, BDGW94], the authors have contributed with versions that reduce the number of equivalence queries or that parallelize in an optimal way. Our main aim in this text is to cast our previous work in terms similar to those of [KV94], or at least providing as much, or more, intuition besides the formal correctness and efficiency proofs.

In order to do this, we develop a somewhat different combinatorial concept, the observation pack, that abstracts from the various data structures used for bookkeeping in the previous versions. We prove all our lemmas and main results in terms of observation packs. We believe that this approach is clearer, and allows one to see rather clearly "what makes the proof work"[1] (which was not clear to some of the authors even in our own proofs of our own variants of the algorithm).

Together with the discussion of known upper and lower bounds, we prove a new lower bound relating the number of equivalence and membership queries needed for learning dfas. In particular, this lower bound proves that, at least for acyclic dfa, the main algorithm presented here is optimal with respect to the number of queries.

Remark that we survey only research on learning *deterministic* automata, and only within the framework of query learning. There is a lot of work done on learning more general types of automata or learning them in other learning formalisms. For example, Tzeng [Tze92] investigates learning probabilistic automata via queries. The survey by Pitt [Pit89] and the Introduction in Ron's Ph. D. thesis [Ron95] discuss automata learning in other frameworks.

2 Preliminaries

We assume that the reader is familiar with the basic notions on formal languages and automata. Most of our notation is standard. For a string x, $|x|$ denotes its length; for a set A, $|A|$ is its cardinality; for a finite automata $\mathcal{M}$, $|\mathcal{M}|$ is the number of states it has, and $L(\mathcal{M})$, the regular language it accepts.

[1] The question that Ron Book so many times has asked the authors in so many different opportunities.

Learning from queries was introduced by Angluin [Ang87], who in the same paper gave an algorithm for learning dfa in that model. We present the model as it applies to this specific problem. See the continuation paper by Angluin [Ang88] and the later framework paper by Watanabe [Wat94] for a full discussion.

A query learning algorithm for dfa interacts with a *teacher* who has in mind a fixed regular language T over a known alphabet, the *target language.* The goal of the algorithm is to come up with a deterministic finite automata accepting T.

Interaction is initiated by the algorithm, who can ask one of two types of queries. In a *membership* query, the algorithm presents a string x and asks whether x is in T; the teacher answers YES or NO. In an *equivalence* query, the algorithm produces a dfa M, and asks whether $L(M) = T$; the teacher either answers YES if this is the case, or else returns a *counterexample,* any string witnessing $L(M) \neq T$. The algorithm must eventually output a dfa for T no matter how the teacher chooses the counterexamples.

We demand that the algorithm is efficient. Formally, the time spent at any moment must be bounded by a fixed polynomial of two parameters: the number of states of the minimum dfa accepting T (which is taken as a measure of complexity of the target), and the length of the longest counterexample received so far (which somehow measures the inefficiency of the teacher). These two parameters are customarily denoted by n and m. For algorithms that work on any alphabet, we also take into account the cardinality of the alphabet, usually denoted by k.

Besides the *time complexity* of an algorithm, it is also interesting to measure its *query complexity,* this is, the number of queries it asks. For an algorithm A, $\#equiv(A, n, m, k)$ denotes the number of equivalence queries made by A in the worst case over any teacher when the relevant parameters are n, m, and k. The definition of $\#memb(A, n, m, k)$ is analogous. For simplicity, in these notations we sometimes omit the parameters that are clear from the context or irrelevant to the discussion.

3 Observation Packs

Our main concept to talk about learnability of finite automata is that of observation pack, which is simply a set of examples, appropriately organized to go on with the identification process. Here an *example* is a pair of the form $(x, +)$, which means that the word x is in some formal language T, or $(x, -)$, which means that the word x is not in T. This interpretation makes sense only for *consistent* sets of examples, also called *samples*, those in which the same word does not appear with different $+/-$ labels.

Specifically, the examples in an observation pack are organized in a number of finite sets, not necessarily disjoint, that we will call *components*. We use, for now, the notation $\mathcal{A} = (A_0, \ldots, A_{n-1})$ to denote a pack with the n components A_i. The following conditions are to be obeyed by the components of an observation pack $\mathcal{A}$:

- Let $s_k \in A_k$ be the shortest word in A_k; then s_k is a prefix of all other words in A_k.

- For each two components A_k and A_ℓ with $\ell \neq k$, there exists $w_{k\ell}$ such that both $s_k w_{k\ell} \in A_k$ and $s_l w_{k\ell} \in A_\ell$ but they have different labels.

Thus s_k is just, in essence, a word used to identify A_k, while the various suffixes appended to it to obtain the actual words in the pack allow us to distinguish between the various s_k. We define $E_k = \{w | s_k w \in A_k\}$. Thus the word $w_{k\ell}$ mentioned above is actually in $E_k \cap E_\ell$. Also, $\lambda \in E_k$ since $s_k \in A_k$.

The notation just introduced, while reasonable as introduction, rapidly ends in an acute case of indexitis. We will simplify it somewhat by collecting in a set S all the strings s_k, which (for reasons to be apparent soon) we will call *access strings,* and by using the access strings to index both components A_k and sets of suffixes E_k. The pack is now given, therefore, by the finite set S of access strings and a mapping from S associating to each s the finite set E_s; the number of components is $|S|$. The set A_s is obtained by prepending s to each word in E_s. So, λ must belong to all the E_s, and for $s \neq s'$ from S, some $w \in E_s \cap E_{s'}$ distinguishes them as described above: $sw \in T \iff s'w \notin T$.

This notation turns out to be more comfortable, while the one introduced before describes in a clearer way, intuitively, the use to which we will put the concept. The expression "words in the pack" will refer still to those of the sets A_s.

As already said, the labels of the words in the pack mean something: they represent membership in a set. This is formalized as follows:

Definition 3.1 *A language T is* consistent *with $\mathcal{A}$ if all $+$ labels mark words in T, while all $-$ labels mark words not in T.*

The reader familiar with finite automata will see that the suffixes in sets E_s must include proofs that the access words $s \in S$ belong each to a different class in the right congruence associated with any regular set T consistent with the pack. Recall that these classes correspond to the states of the minimum deterministic finite automaton (dfa) for the set: access strings are used to reach them, whence the name. Clearly, there can be no more access strings than states in such an automaton. Formally:

Lemma 3.2 *Let $\mathcal{A}$ be a pack, S its set of access strings, T a regular language which is consistent with $\mathcal{A}$, and $\mathcal{M}_T$ the minimal dfa that recognizes T. Then $|\mathcal{M}_T| \geq |S|$.*

Proof. Let δ^T and q_0^T be the transition function and the initial state of $\mathcal{M}_T$, respectively. Map S into the states of $\mathcal{M}_T$ in the natural way: from s to $\delta^T(q_0^T, s)$. It is easy to prove that this mapping is injective, since the experiments in the pack must distinguish the corresponding states. □

A set that is consistent with the pack also provides an interesting partial function, which will be the basis of many of our proofs and explanations throughout.

Definition 3.3 *Let $\mathcal{A}$ be a pack, with access strings S, and T a set consistent with $\mathcal{A}$. We say that a word z is like $s \in S$ for T if and only if $\forall w \in E_s$ $(sw \in T \iff zw \in T)$.*

Note for later use that we can check this condition easily, if T is available as an oracle. We know whether $sw \in T$ because the word is consistently labelled in the pack. We can know whether $zw \in T$ just asking a membership query.

The following easy fact holds:

Lemma 3.4 *For every word z there is at most one word $s \in S$ such that z is like s for T.*

Proof. Let $s \neq s'$, both in S, and assume z like s: $\forall w \in E_s(sw \in T \iff zw \in T)$. The pack provides a word $w \in E_s \cap E_{s'}$ such that $sw \in T \iff s'w \notin T$. Thus, $zw \in T \iff s'w \notin T$, so that z is not like s' for T. □

Clearly, each access string is like itself. The partial function alluded to above, $\gamma^{\mathcal{A},T} : \Sigma^\star \to S$, maps each z to the only access string it is like to for T, if there is one; remains undefined if z is not like any access word for T. From now on, we will use this function in a context in which both $\mathcal{A}$ and T are fixed, so we will omit the superscripts and use only γ; but it is sometimes useful to remember that it depends both on the pack $\mathcal{A}$ and on the set T. Note that, whereas in Lemma 3.2 we have assumed that T is regular, this assumption is nowhere else necessary, and it will not be again until the moment of using that lemma.

3.1 Expanding a Pack

The fact that γ, as a function, can be partial was not particularly relevant in the previous section; we move now to explain its significance. For a given pack (that we assume known from the context), let us say that a word w is *escaping* when $\gamma(w)$ is undefined.

The crucial process will be that of adjusting an observation pack to better match our target. There is a simple way to modify it, provided that some escaping string w is known. Indeed, in this case we can collect the appropriate experiments and *expand* the pack, adding a new component to it.

Namely: the fact that w escapes implies that, for each access string s in the pack, there is a word $sv \in A_s$ providing the experiment v that distinguishes w from s, in the sense that $sv \in T \iff wv \notin T$. Thus, a set formed by all such words wv, additionally including w itself, and each labeled by the corresponding $+/-$ label, forms a component that can be added to the pack preserving all necessary properties, as can be routinely checked. It follows from previous facts that each expansion by one component brings the pack one state closer to the true minimum automaton for T.

This idea can be used to *close* a pack: as long as some word sa escapes, where s is an access string and a is a letter, expand the pack. By the time this process gets stuck, it trivially holds that no such word sa escapes. Later we will come back to the point of alternative dens to hunt for escaping strings. For now, let us define closed packs: those where this process ends. Formally:

Definition 3.5 *A pack $\mathcal{A}$ consistent with T is* closed for T *if:*

- $\gamma(\lambda)$ *is defined;*
- $\forall s \in S, \forall a \in \Sigma, \gamma(sa)$ *is defined.*

Remark that this definition *does* depend on T since $\gamma = \gamma^{\mathcal{A},T}$ does. For a pack closed for T, define $\delta(s,a) = \gamma(sa)$, and, overloading the operator in a rather usual form, $\delta(s,\lambda) = s$, and $\delta(s,wa) = \delta(\delta(s,w),a)$. Note again that δ depends both on the pack and on T, through γ.

The first definition actually gives rise to a deterministic finite automaton, whose states are the access strings of the pack (or, equivalently, its components), with δ as transition function, as soon as we decide that $\gamma(\lambda)$ is the initial state, and that accepting states are those access strings with label $+$ (i.e., those that belong to T by the consistency assumption). The symbol δ has been extended to operate also on words, in the usual sense, and it corresponds to iterating γ symbolwise. (We have not stated, though, that $\delta(s,w) = \gamma(sw)$. This is not true in general, as we shall see, for $|w| > 1$, and the inequality will be actually a useful source of escaping strings.)

These automata that we get from closed packs are important: one of them is the one we are looking for. Formally:

Theorem 3.6 *If $\mathcal{A}$ is a pack, T is regular and consistent with $\mathcal{A}$, and $\mathcal{A}$ has as many components as $\mathcal{M}_T$ (the minimal dfa that recognizes T), then $\mathcal{A}$ is closed for T so that an automaton can be obtained from $\mathcal{A}$ in the manner above, and furthermore this automaton is isomorphic to $\mathcal{M}_T$.*

Proof. By Lemma 3.2, no pack consistent with T can have more than $|\mathcal{M}_T|$ components. Therefore, it is not possible to expand $\mathcal{A}$ preserving consistency with T, and thus it must be closed. Besides, by the cardinality condition, the function defined and used in Lemma 3.2 becomes bijective. It is routine to show that this function is an isomorphism, that is: 1/ it maps $A_{\gamma(\lambda)}$ to the initial state of $\mathcal{M}_T$; 2/ it maps exactly those A_s with $s \in T$ to the final states; and 3/ it commutes with $\mathcal{M}_T$'s transition function. $\square$

Yet an additional comment about escaping strings is in order. An escaping string must provide us with proofs that it indeed escapes, and those must be experiments already in the pack. We can, as well, add one or more words, with appropriate prefixes and labels, to the pack in order to force strings to escape. In particular, in many cases we will start from a word w for which $\gamma(w)$ is defined, say s; by the uniqueness property, there are experiments in the pack distinguishing w from all other access strings. Then, it suffices to find one experiment v that distinguishes w from s, and to add sv to the component A_s, to make w escaping. We will use this idea in several opportunities.

3.2 Using a Counterexample

How do we go on from a closed pack? We have seen how to construct an automaton from it, and it is natural to use it to ask an equivalence query. A positive answer means success of the learning process. Otherwise, a counterexample is received: of course, it should now be possible to get some information out of it; ideally, we would like an escaping string. We use the idea sketched in the previous section: we shall get from the counterexample a word to be added to one of the components and, as a consequence, a word that escapes the larger pack.

Let us describe how. Let w be the counterexample, of length m, composed by the letters a_i as $w = a_0 \cdots a_{m-1}$. Let u_i be the prefix of length i of w, and v_i such that $w = u_i v_i$; here $0 \leq i \leq m$ so that $u_0 = v_m = \lambda$ and $v_0 = u_m = w$. Also, $w = u_i a_i v_{i+1}$ for $i < m$.

Additionally, let $s_i = \delta(\gamma(\lambda), u_i)$ be the state of the automaton based on the pack that is reached by computing on u_i. Note that, by definition of δ, $s_0 = \gamma(\lambda)$ and $s_{i+1} = \delta(s_i, a_i)$. The classification of w by this automaton is given by whether the final state s_m is accepting, which corresponds to whether $s_m \in T$. The fact that w is a counterexample means that $w \in T \iff s_m \notin T$.

Consider each suffix v_i as an experiment on the corresponding state s_i; this amounts to finding out, via membership queries, whether $s_i v_i \in T$. Intuitively, this corresponds to running i steps of the queried automaton and checking out what the remaining suffix tells us about the state we are in.

Observe the following: $s_0 v_0 \in T \iff v_0 \in T \iff w \in T \iff s_m \notin T \iff s_m v_m \notin T$ [2]. Consequently, there must exist one or

[2] Actually, this assumes that $\lambda v_0 = \gamma(\lambda) v_0 (= s_0 v_0)$. Otherwise, experiment v_0 distin-

more "breakpoint" positions i such that $s_i v_i \in T \iff s_{i+1} v_{i+1} \notin T$. But $s_i v_i = s_i a_i v_{i+1}$, thus v_{i+1} is an experiment that distinguishes $s_i a_i$ from $s_{i+1} = \delta(s_i, a_i) = \gamma(s_i a_i)$. It suffices now to add $s_{i+1} v_{i+1}$, with the appropriate label, to the component $A_{s_{i+1}}$. Now $\gamma(s_i a_i)$ is not anymore s_{i+1}, $s_i a_i$ escapes, and we can go on with the pack expansion process. One or more such i can be found easily through membership queries.

As a whole, we either have a pack that is not closed, and then get escaping words of the form sa for some access string s and letter a, or a closed pack; then, an equivalence query either tells us that the task is complete, or provides us with escaping words through a counterexample. By Lemma 3.2, we never have too many components, and by Theorem 3.6, when we have gathered sufficiently many components an equivalence query is made and answered YES. It is immediate how to build a learning algorithm from these considerations.

4 Implementations of Observation Packs

Although observation packs, as such, have not been introduced in the previous literature, they are very close relatives of the discrimination trees of Kearns and Vazirani [KV94], and actually all the intuitions that allowed us to cast the learning algorithm in terms of observation packs are already in that book. The learning algorithms known for dfa, by Angluin, Rivest and Schapire, and Kearns and Vazirani, are essentially the same; we describe in this section how to compare them by showing that all three of them can be considered as implementations of the one outlined above, by choosing different data structures to implement the observation packs. This allows one to see better in what sense they are the same algorithm, and what exactly the differences mean, and why they are relevant in deeper studies of the problem.

4.1 Discrimination Trees

A *discrimination tree* is a rooted binary tree labeled by a word in each node. The root must be labeled λ. It implements an observation pack as follows: the access strings are at the leaves of the tree, and the set of experiments

guishes λ from $\gamma(\lambda)$ and we have λ escaping. To avoid complicating the algorithm with this case, let it start from a pack that contains λ as an access string.

associated to access string s is formed by all the words labeling internal nodes along the path from the root. Having λ at the root ensures $\lambda \in E_s$ for every E_s.

The $+/-$ labeling is implicit: word sv gets label $+$ if and only if s is in the right subtree of the internal node labeled v, and $-$ otherwise. Accordingly, access strings that are in the target language (and correspond, therefore, to accepting states in the automaton constructed from the pack) are those at the right subtree of the whole tree. The distinguishing experiment necessary to separate two access strings lies, therefore, at their lowest common ancestor.

This structure allows for a rather fast and easy to understand computation of the function $\gamma(w)$: just sift w down the tree, querying wv for membership at each internal node labeled v, and going right or left according to the answer. The value of $\gamma(w)$ is the leaf this process ends in: a word for which all the experiments along its path fail to distinguish it from w. Moreover, γ is total, since this process can be done for any input string w. Therefore, a pack implemented by a discrimination tree is always closed: escaping strings are obtained only through counterexamples. In the algorithm in [KV94], at each counterexample, the escaping string is taken via the shortest prefix u_i for which $s_i a_i v_{i+1} \in T \iff s_{i+1} v_{i+1} \notin T$ holds.

In that case, since $s_{i+1} = \gamma(s_i a_i)$, we know that the discrimination tree already distinguishes $s_i a_i$ from all other access strings. The leaf s_{i+1} is then replaced by an internal node labeled v_{i+1}, separating the old leaf from the new one $s_i a_i$. Then the tree is ready for another equivalence query.

The tree is initialized to contain a single interior node labelled λ that distinguishes between two leaves, one labelled λ and another labelled with some string whose membership in the target is opposite to that of λ. This tree is easy to form with one membership query and one equivalence query. As leaves are never removed in the process described above, this ensures that λ remains an access string and therefore $\gamma(\lambda) = \lambda$.

4.2 Observation Tables

Angluin's version of the algorithm [Ang87] was the first to prove learnability of dfa's via queries. It implements the observation pack in the form of a two-dimensional table, called *observation table,* with both rows and columns indexed by strings. The entry (s, v) of the table is the label $+/-$, according to whether $sv \in T$. The table is split into two sets of rows. Access strings index rows of the first part of the table, but also many other rows with other indexing strings may appear in this part; in fact, the words indexing rows form a prefix-closed set (and thus include λ), whereas the words indexing columns are the experiments and form a suffix-closed subset (and thus include λ as well).

The second part of the table contains all the words of the form sa, where a is a symbol and s appears in the first part, unless they already appear also in the first part. The set of experiments is the same for all the access strings, and is applied to all the words indexing rows in both parts of the table to fill it in completely.

Observation tables are observation packs with the following interpretation: whenever two rows differ at the column indexed by string v, then v is an experiment to distinguish the two row indexes. Thus, a state is now a maximal set of equal rows, and we can choose as an access string any of them for each state; only those words that have as prefix a chosen access string belong to the observation pack implemented by the table.

The second part of the table is used to detect whether the table is closed and, whenever this is the case, to define the transition function of the associated automaton. If the table is not closed, the second half of the table contains a witness for this fact: a row indexed by a escaping string sa that is different from all rows in the first half. Moving this different row to the first half is equivalent to expanding the pack with string sa.

The fact that an observation table contains "redundant" rows provides a new way of catching escaping strings, via the concept of "self-consistency" to be used later. This concept was called simply "consistency" in [Ang87], but we have preferred to use this term to mean agreement between a pack and the target language.

A table is called *self-consistent* if for any two strings s_1 and s_2 indexing equal rows in the first half, and for every symbol a, the corresponding rows indexed s_1a and s_2a in the first or second half are also equal.

We have seen that having a closed table is enough to define a dfa. If moreover the table is self-consistent, the dfa is in fact independent of the string chosen as access string for each different row. If the table is not self-consistent, then there are strings s_1 and s_2, a symbol a, and an experiment v such that the rows s_1 and s_2 are equal but entries (s_1a, v) and (s_2a, v) of the table are different. This means that at least one of s_1a or s_2a is escaping, namely, one that disagrees on v with their common access string. Equivalently, experiment av distinguishes strings s_1 and s_2. Therefore, adding a new column indexed by string av, and filling in the new entries with membership queries, makes rows s_1 and s_2 different.

Once the table is closed and self-consistent, no more escaping strings can be found in these ways so an equivalence query is made. Suppose that a counterexample w is received. As we explained before, this w can be used to find an experiment distinguishing two previously equal rows of the table. and thereby increasing the number of components. Angluin's algorithm, however, does not directly follow this scheme. Instead, it adds a new row for each prefix of the counterexample that was not there, and fills these new rows with membership queries (no new column, or experiment, added so far). We will argue in a moment that this causes the table to become not self-consistent (and, possibly, not closed too). This inconsistency will be removed, as described above, sometime before the next equivalence query. It is at that time when the number of components of the underlying pack increases.

Here is why adding all prefixes of w as rows breaks self-consistency. The fact that w is a counterexample means that $w = u_m\lambda \in T \iff s_m = s_m\lambda \notin T$, i.e., u_m is not like s_m using as experiments the strings that index columns. On the other hand, $u_0 = \lambda$ is like $s_0 = q_0 = \gamma(\lambda)$ by definition of γ. Therefore, there is some minimum i, $0 < i \leq m$ such that u_i is not like s_i. Now, u_{i-1} is like s_{i-1} because i is minimum, $s_{i-1}a_{i-1}$ is like s_i by definition of the queried dfa, yet $u_{i-1}a_{i-1} = u_i$ is not like s_i. Therefore, rows indexed u_{i-1} and s_{i-1}, together with symbol a_{i-1}, witness that the table is not self-consistent.

Note that, to remove such an inconsistency, the algorithm eventually adds a column (experiment) of the form $a_{i-1}v$, for some existing column v. This new experiment distinguishes rows u_{i-1} and s_{i-1}, hence u_{i-1} becomes a new "access string". This process exactly mimics the processing of a counterexample that we explained in the pack-based algorithm.

4.3 Reduced Observation Tables

Rivest and Schapire's algorithm is also based on the concept of observation table. Unlike in Angluin's, only access strings are kept on the first half of the table. Therefore, all rows in this first half are different and self-consistency trivially holds at every moment. Furthermore, the table structure implies that all access strings (row labels) have the same set of experiments.

Having to fill a smaller observation table, this algorithm is more efficient than Angluin's in the use of membership queries. A second source of efficiency comes from a faster way of processing a counterexample. Recall that, given a counterexample w of length m, processing w means finding a breakpoint i such that $s_i v_i \in T \iff s_{i+1} v_{i+1} \notin T$, where the s_i are the states visited by w along the automaton and the v_i are the corresponding suffixes of w. Some such breakpoint must exist since $s_0 v_0 \in T \iff s_m v_m \notin T$, so that an obvious sequential search will find, say, the first one with m membership queries. Rivest and Schapire show how a binary search finds a breakpoint (not necessarily the first one) with $\log m$ queries.

A point worth noting is the following. The reduced table is kept small by not adding all prefixes of the counterexample as rows. This means that the new automaton may still classify incorrectly the counterexample, so that the same counterexample can potentially be used to answer several equivalence queries. This happens as well in the algorithm using discrimination trees.

5 Query Complexity of the Algorithms

We discuss first the complexity of the algorithm based on observation packs, then of each of the three implementations in the previous section. We estimate only their query complexity, i.e., the number of queries; their time complexity can be estimated with similar arguments.

In what follows, let n, m and k be $|\mathcal{M}_T|$, the length of the longest counterexample given by the teacher and $|\Sigma|$ respectively.

It is easy to see that the number of equivalence queries is at most n: each counterexample processed immediately adds at least one new component to the pack; hence each new equivalence query has at least one more state than the previous one. This is also true for the three implementations of the algorithm discussed, so we do not consider equivalence queries any more.

Membership queries are performed for two different purposes: to check for closure and to process a counterexample.

Consider the first type of queries. Recall that the pack is closed when, for every access string s_i and letter a, $s_i a$ is like some other access string. This is easily determined with membership queries. If the check fails, it provides for a witness of non-closure. If it succeeds, a dfa can be built from the answers to the queries.

Note that each component contains an access string plus at most $n - 1$ strings used to separate it from the other at most $n - 1$ components. Therefore, in the worst case, checking for closure means asking n queries for each of n strings s_j and n queries for each of the kn strings $s_i a$, so a total of $(k+1)n^2$ queries At this point we can implement the algorithm in two different ways:

- Check for closure (and rebuild the automaton) from scratch every time. This means $n \cdot (k+1)n^2 = O(kn^3)$ queries.

- Use the fact that access strings are never removed from the pack. This means that the set of queries asked in one closure check is a subset of that in the next check, so the total number of *different* queries over all checks is at most $(k+1)n^2$. We can avoid repeating queries by recording all answers to membership queries, at the expense of some more memory.

Consider now the queries used to process the counterexample. If we do not insist on obtaining the shortest distinguishing experiment, we can use Rivest and Schapire's binary search, discussed above. This means using $O(\log m)$ queries for each counterexample, hence $O(n \log m)$ for the at most n counterexamples.

In total, the algorithm that records all answers uses at most $O(kn^2 + n \log m)$ membership queries and n equivalence queries. We will show in a later section that, at least for acyclic automata, this is optimal up to constant factors.

This is the cost achieved by Rivest and Schapire's algorithm: their reduced observation table is precisely the data structure used to record all answers to membership queries.

The Kearns-Vazirani algorithm, as described in [KV94], rebuilds the automaton from scratch every time and processes the counterexample sequentially, so it uses $O(kn^3 + nm)$ queries. It is not difficult, however, to make it record previous queries and use binary search to process the counterexample. This modified version will also have $O(kn^2 + n \log m)$ cost.

In Angluin's algorithm, the number of columns in the table (or experiments to distinguish states) is still n, but the number of rows can be as large as $O(knm)$ because all prefixes of counterexamples are added as rows; consequently, the number of queries can be up to $O(kn^2m)$.

6 Milking counterexamples via lookahead

This section goes back to the promise of considering other ways to obtain escaping strings. We have seen already the concepts of closure and self-consistency. Now we combine both, by using a classical idea in computer science: looking ahead, beyond the current point of interest, by some number of symbols h.

The concept of closure immediately suggests a definition: $\gamma(su)$ must be defined for all access strings $s \in S$, and for suffixes u of length up to h. This idea is useful but does not seem to suffice to gain anything. We combine it with a self-consistency condition. The proof of the following is an easy induction:

Lemma 6.1 *In an observation pack $\mathcal{A}$, for each access string $s \in S$, the following are equivalent:*

a/ $\gamma(su)$ is defined and equal to $\delta(s, u)$ for all words u of length less than or equal to h;

b/ $\gamma(sua) = \gamma(\gamma(su)a)$ for all symbols a and words u of length less than h.

In condition b/, we understand that all three applications of γ must be defined. If these conditions hold, we say that the pack is *h-closed* for T (note that both γ and δ depend on T).

Note that condition a/, so also b/, extend the notions of closure and self-consistency. In a/, closure is the fact that $\gamma(su)$ is defined and self-consistency the fact that it equals $\delta(s, u)$ for the special case $|u| = 1$. In b/, the same is obtained by taking $|u| = 0$. In other words, our previous notion of closure is now 1-closure.

This provides a way of getting escaping words using the lookahead h. Consider condition b/: if one of the three applications of γ is undefined, we get an escaping string; and, if they are defined but inequality holds, some experiment w distinguishes $\gamma(sua)$ from $\gamma(\gamma(su)a)$, and it is easy to see that aw distinguishes su from $\gamma(su)$. Thus, adding aw to $E_{\gamma(su)}$ makes su escaping.

Now we propose the following parameterization of the algorithm: instead of making the equivalence query with a closed pack, we insist in having an h-closed pack. We keep expanding it via escaping strings, found as just indicated, as long as the pack is not h-closed. In this way we ensure that all equivalence queries made are all h-closed. Comparing h-closed packs gives interesting information. We start with a technical fact: informally, in an h-closed pack, γ and δ commute up to length h.

Lemma 6.2 *If $\mathcal{A}$ is an h-closed pack then for every access string s in $\mathcal{A}$ and every two strings u, v with $|u| + |v| \leq h$, $\gamma(\gamma(su)v) = \gamma(suv)$, and $\delta(\delta(q_0, su), v) = \delta(q_0, suv)$.*

Here $q_0 = \gamma(\lambda)$, the initial state of the automaton built from the closed pack. From this lemma it is routine to show the following:

Lemma 6.3 *Let $\mathcal{A}$ be h-closed and $\mathcal{A}'$ be an extension of $\mathcal{A}$ that is 1-closed and has strictly more components. Then $\mathcal{A}'$ has at least h more components than $\mathcal{A}$.*

Therefore, given an h-closed pack $\mathcal{A}$ and a counterexample, we can find at least h new states with membership queries. It is enough to process the counterexample as in previous algorithms, therefore adding one component. Remark now that new experiments are formed by prefixing existing experiments with words of length at most h, instead of single letters as in the standard algorithm.

Then keep expanding the pack using membership queries until and a new h-closed pack $\mathcal{A}'$ is obtained. We are guaranteed to terminate because the number of components in $\mathcal{A}'$ cannot exceed n, and it is easily shown that at that point it must be h-closed. By the lemma, the automaton we obtain from $\mathcal{A}'$ has at least h states more than the last equivalence query.

Therefore, at most n/h equivalence queries are enough to gather the n states of the target automaton. Note that keeping an h-closed pack is expensive in membership queries: roughly 2^h queries are needed for each of the n components. Setting $h = \theta(\log n)$, $n \cdot 2^h$ is still polynomial, so we obtain a polynomial time algorithm that makes a sublinear number of equivalence queries.

Corollary 6.4 *There is a polynomial-time algorithm that learns dfa making $O(\frac{n}{\log n})$ equivalence queries.*

This algorithm was presented independently in [BDGW92, BGHM93]. In both references, the implementation described uses Angluin's observation tables.

6.1 Adaptiveness and Parallelization

An idea coming from the study of parallel models of computation, which turned out to be rather suggestive in some other areas of complexity theory, was the concept of *phase*. In its initial setting, which was the comparison of sequential with parallel models of computation, a phase corresponds to a fragment of the computation of a sequential model in which no information obtained during the same fragment is actually used. All decisions and computations only use information computed along previous phases. In a sense, this corresponds to a lack of adaptiveness during the fragment: everything that is done in that fragment is independent of anything else done in the same fragment. The interest of the idea is that it means that everything in each phase can be parallelized, due to the lack of adaptiveness, so that the number of phases in the sequential model corresponds (roughly) to the parallel time in the parallel model.

An important step was Ron Book's intuition that, in an oracle model, the queries can be seen as dividing the computation in a sort of phases. Whereas this intuition does not seem at first very clear, since the intuition of adaptiveness is only focused on the oracle queries, it turned out that it suggested much better and clearer proofs on previously known facts regarding positive relativizations [BBS85]: the key idea was to consider models possibly equipped with several oracles, and to use one of them as the phase separator.

We should also mention that this intuition was eventually fully formalized in the setting of space-bounded computation, where an appropriate model of functional oracle machine gave an exact characterization in which the "oracle queries" resource matched rather well the "parallel time" resource [ABJ95].

The same intuition shows up now in our context: in the implementation by observation tables, all experiments being the same for all strings, the membership queries needed to check closure and to identify escaping strings are nonadaptive. But equivalence queries with counterexample are not: everything after the query depends heavily on the counterexample obtained. Each equivalence query must, therefore, correspond to a different phase. The analogy with the two oracles in [BBS85] lies now in the two sorts of queries.[3]

This leads to the following idea: can we parallelize all the activities between equivalence queries, even with logarithmic lookahead? If we can do it in constant time, on a parallel model, we would obtain a parallel algorithm running in $O(n/\log n)$ time.

On the other hand, a parallel algorithm running in time t can be sequentialized in such a way that equivalence queries correspond to phases, to get a simulation within t equivalence queries, and thus completing the picture of an optimal parallel algorithm. This part was based in a simulation technique of [BC92].

We could not reach down to constant time for each iteration, but a sort of amortized analysis still gave us this optimal parallel algorithm. We describe it briefly; the proof of optimality is deferred to the section on lower bounds.

A phase of the algorithm is as follows: check whether the current pack (or table) table is h-closed. If it is, use it to make an equivalence query. Otherwise, expand the pack by adding new components having as access strings all those of the form sv, with s an existing access string and v having length h or less. Then delete components until we have again a legal observation pack; namely, until there are no two components such that their access strings are like each other. This resulting pack is called the h-expansion of the original pack.

Some programming shows that one such phase is completely parallel: it can be done in constant time in a Concurrent-Read, Concurrent-Write Parallel RAM, using a number of processors that is polynomial in the size of the pack and 2^h.

We argue that the number of phases is $O(n/h)$. Consider first the phases that end in an h-closed packs. By Lemma 6.3, each one has at least h more components than the previous one. Therefore, there are at most n/h such phases (hence, at most n/h equivalence queries). Now take a phase where the pack is not h-closed, say it has i components. Suppose its h-expansion

[3] Restricted equivalence queries are nonadaptive, though; just assume a NO answer and go on until exhausting the time bound, then collect and query all equivalences made. This point is important in some PAC simulations of query learning (see [DW90]).

has $i+j$ states, with $j>0$. In case $j \geq h$, at least h components have been added, so again this case can occur at most n/h times. In case $j<h$, it is easy to argue that the new pack must be $(h-j)$-closed, otherwise the expansion process can continue. Applying again Lemma 6.3, the next 1-closed pack must have at least $(i+j)+(h-j)=i+h$ components. Therefore, two consecutive phases of this kind are guaranteed to add h components. This shows that the total number of phases is $O(n/h)$.

7 Lower Bounds and Open Problems

Finally, let us consider how close to optimal are the algorithms we have been presenting. In most cases, we consider "optimal" an algorithm that makes as few queries as possible. The exception is the parallel algorithm discussed in Section 6.1, where only parallel time is considered.

All lower bounds apply to alphabets Σ with at least 2 symbols – learning dfa over one-letter alphabets is extremely easy.

We start with an easy observation: for any learning algorithm, the sum of membership and equivalence queries must be at least n. Otherwise, there is a state in the minimum dfa for the target that the algorithm never sees, that is, no string ending in it is either asked as a membership query or returned as a counterexample. Therefore, changing that state from rejecting to accepting or vice-versa changes the target language but not the output of the learning algorithm.

With a far more complex argument based on Vapnik-Chervonenkis dimension, Ishigami and Tani [IT93] showed that actually $\Omega(kn \log n)$ queries are needed in total. They conclude, in particular, that algorithms making only $O(n)$ equivalence queries must make $\Omega(kn \log n)$ membership queries. This particular case is interesting because the available algorithms make at most n equivalence queries (independently of k).

Another known lower bound limited the possibility of reducing the number of equivalence below n, as we tried in Section 6. With some care, it can be shown that the algorithm there makes $O(n^2 \cdot k^h)$ membership queries for n/h equivalence queries. The lower bound in [BDGW92] showed that this is at most $O(n^2)$ away from optimal.

Theorem 7.1 *[BDGW92] For every learning algorithm, if* $\#equiv(A,n,k) \leq n/h$ *then* $\#memb(A,n,k) \geq k^h$.

In particular, this shows that the number of equivalence queries must be at least $\Omega(n/\log n)$ for any algorithm making polynomially many membership queries. It also allowed us to prove, later, the following lower bound on parallel time.

Corollary 7.2 *[BDGW94] Every parallel learning algorithm for dfa using polynomially many processors runs in time $\Omega(n/\log n)$.*

Therefore, the algorithm sketched in Section 6.1 is time-optimal up to multiplicative constants in time, and polynomials in the number of processors.

Observe, however, that the lower bound in Theorem 7.1 is meaningless when the number of equivalence queries is very close to n. We present now a new lower bound that partially remedies this gap.

Theorem 7.3 *There is a constant $c_0 > 0$ such that for every learning algorithm and every $k \geq 2$,*

$$\#memb(A, n, k) \geq c_0^2 \cdot (k-1) \cdot n^2 - c_0 \cdot n \cdot \#equiv(A, n, k).$$

Proof. For a given n, define $m = (n+1)/4$. We assume for simplicity of presentation that m is a power of 2; it is straightforward to extend the proof to other values of n.

Fix a letter $a \in \Sigma$ and define $L_1 = \{\ waw \mid |w| = \log m\ \}$. Informally, a minimum dfa for L_1 consists of a selector tree with a branch for every w, transitions with letter a, then a "checking" tree converging to a single accepting state. Let δ be its transition function and q_0 its initial state. For each w of length $\log m$, define $q_w = \delta(q_0, w)$ and $p_w = \delta(q_0, wa)$ so that $p_w = \delta(q_w, a)$. Since all states q_w and p_w must be distinct, this automaton has at least with $(2m-1) + (2m-1) + 1 = 4m - 1 = n$ states (including the sink).

Now for each q_w and each letter $b \in \Sigma - \{a\}$ we can define the transition $\delta(q_w, b)$ to be any of the states p_v, or leave it undefined. There are $(k-1) \cdot m^2$ such potential transitions, of which up to $(k-1)m$ can actually exist in a dfa. Once a set of such transitions has been added, the resulting minimum dfa still has n states and accepts a set of the form $L_1 \cup S$, where S is a subset of $L_2 = \{\ wbv \mid |w|, |v| = \log m \wedge b \in \Sigma - \{a\}\ \}$ containing at most one string wbv for each w and b. We confront a given learning algorithm A against the class of such target sets. Informally, A must determine, for every q_w and b, the p_v such that $\delta(q_w, b) = p_v$, if any.

Consider an adversary teacher that answers A's queries as follows. It maintains a set S of strings that have been returned as positive counterexamples. Initially $S = \emptyset$ and, at every moment, the answers given by the adversary are consistent with $L_1 \cup S$ being the target set.

1. Any membership query x: answer YES if $x \in L_1 \cup S$, NO otherwise.

2. An equivalence query rejecting some $wcv \in L_1 \cup S$: return (wcv,YES).

3. An equivalence query accepting some $wbv \in L_2 - S$: return (wbv,NO).

4. An equivalence query accepting exactly $L_1 \cup S$; only here the adversary is forced to reveal a new transition: select some $wbv \in L_2$ such that no string wbv' is in S and wbv never appeared before in the dialog; add wbv to S and return (wbv,YES) as a positive counterexample. (In case no such wbv exists, the adversary admits that A has succeeded and returns YES).

Note that case 2 provides no new information about the target, that cases 1 and 3 discard only one of the candidate transitions, namely, $\delta(q_w, c) = p_v$, and that case 4 reveals the presence or absence of only m of them, namely, $\delta(q_w, b) = p_v$ so $\delta(q_w, b) \neq p'_v$ for all $v' \neq v$. Therefore, if A makes k equivalence queries, it can know at most $k \cdot m$ transitions. The process terminates when $|S| = km$, or in other words, when all of the presence or absence of all $(k-1)m^2$ potential transitions is known. Therefore, the number of membership queries must be at least $(k-1)m^2 - km$. As $n = 4m - 1$, the statement of the theorem follows with c_0 about $1/4$. $\square$

Corollary 7.4 *If* $\#equiv(A, n, k) \leq o(kn)$ *then* $\#memb(A, n, k) \geq \Omega(kn^2)$.

Recall that the algorithm we presented makes $O(kn^2 + n \log m)$ membership queries. For acyclic dfas, only counterexamples having length at most n have to be processed, so this is reduced to $O(kn^2)$. As the number of equivalence queries is n, independent of k, this shows that this algorithm is optimal up to $O(.)$, at least for relatively large alphabets, $k > 1/c_0$.

On the other hand, we can improve Theorem 7.3 to show that, when equivalence queries are substantially reduced, membership queries must grow correspondingly above n^2.

Theorem 7.5 *There is a constant c_1 such that, for any algorithm A and for any function $h = h(n)$,*

$$\#equiv(A, n, k) \leq \frac{n}{h} \quad \textit{implies} \quad \#memb(A, n, k) \geq c_1 \frac{n^2}{k^4 h^3} \cdot k^h.$$

Proof. (*Sketch.*) We proceed as in the proof of Theorem 7.3 except that now we choose a new parameter f and define $L_2 = \{ \, wxv \mid |w|, |v| = \log m \wedge |x| = f \, \}$. At most one string wxv will be added to the target for every w of length $\log m$. This gives a dfa with $n \leq 4m + m(f-1) = m(f+3)$ states. Informally, the learning algorithm must identify not only which is the p_v to which q_w is connected, if any, but which of the k^f strings x has been used for the connection.

It can be argued that an equivalence query will reveal at most one such connection; for any state q_w whose transition is not revealed by an equivalence query, an adversary teacher can force $m \cdot k^f$ membership queries. Therefore, if the number of equivalence queries n/h is less than m, the algorithm must ask in the worst case

$$(m - n/h) \cdot m \cdot k^f = \theta((n/f - n/h) \cdot (n/f) \cdot k^f)$$

membership queries. Taking $f = h - 4$, we obtain the statement of the theorem. □

Choosing the appropriate $h(n)$ we obtain the following tradeoffs.

Corollary 7.6 *For any algorithm A,*

- *If $\#equiv(A, n, k) \leq \dfrac{n}{c \log n}$ then*

$$\#memb(A, n, k) \geq \Omega\left(\frac{n^{2+c\log k}}{(c \log n)^3}\right).$$

- *If $\#equiv(A, n, k) \leq \dfrac{n}{c \log n - 3 \log_k \log n}$ then*

$$\#memb(A, n, k) \geq \Omega(n^{2+c\log k}).$$

Hence, it comes close to the optimal by a constant factor in the number of membership queries, and to a factor of only $1 + O(\log\log n/\log n)$ for equivalence queries.

To finish, let us note a few gaps left by these theorems on query complexity. One is fine-tuning Theorem 7.3. As it is, it does not apply to either very small alphabets or a number of equivalence queries very close to n.

Also, recall that current algorithms use $O(n \log m)$ membership queries to process counterexamples. Another open problem is to determine whether this term is necessary, at least for algorithms making less than n equivalence queries. More in general, we do not know whether or not the number of queries for learning must depend at all on m.

Acknowledgements

To Juan Luis Esteban, for acting as a scribe when the first author presented this formulation in a doctorate course, and for letting us copy his scribe notes extensively.

To Ker-I Ko and Ding-Zhu Du for the task of preparing this festschrift volume, and for inviting us to participate in it.

And mostly to Ron, for everything he did.

References

[ABJ95] C. Àlvarez, J.L. Balcázar, and B. Jenner, "Adaptive logspace reducibility and parallel time", *Mathematical Systems Theory*, Vol. 28, 1995, 117–140.

[Ang87] D. Angluin, "Learning regular sets from queries and counterexamples", *Information and Computation*, Vol. 75, 1987, 87–106.

[Ang88] D. Angluin, "Queries and concept learning", *Machine Learning*, Vol. 2, 1988, 319–342.

[Ang90] D. Angluin, "Negative results for equivalence queries", *Machine Learning*, Vol. 5, 1990, 121–150.

[BBS85] J.L. Balcázar, R.V. Book, and U. Schöning, "On bounded query machines", *Theoretical Computer Science*, Vol. 40, 1985, 237–243.

[BDGW92] J.L. Balcázar, J. Díaz, R. Gavaldà, and O. Watanabe, "The query complexity of learning DFA", *New Generation Computing*, Vol. 12, 1994, 337–358. Preliminary version in *Proc. Third Workshop on Algorithmic Learning Theory,* Japanese Society for Artificial Intelligence, 1992, 53–62.

[BDGW94] J.L. Balcázar, J. Díaz, R. Gavaldà, and O. Watanabe: "An optimal parallel algorithm for learning DFA". in *Proc. 7th Annual ACM Conference on Computational Learning Theory,* ACM Press, 1994, 208–217.

[BC92] N.H. Bshouty and R. Cleve, "On the exact learning of formulas in parallel", in *Proc. 33rd Annual Symposium on Foundations of Computer Science,* IEEE Computer Society Press, 1992, 513–522.

[BGHM93] N.H. Bshouty, S.A. Goldman, T.R. Hancock, and S. Matar, "Asking queries to minimize errors", in *Proc. 6th Annual ACM Conference on Computational Learning Theory,* ACM Press, 1993, 41–50. Final version in *Journal of Computer and System Sciences*, Vol. 52, 1996, 268–286.

[DW90] J. Díaz and O. Watanabe, "Simulation of complicated queries by membership queries". 10th ICOT meeting, Tokyo, nov. 1990. Available as Technical Report LSI–90–32, Universitat Politècnica de Catalunya, 1990.

[IT93] Y. Ishigami and S. Tani, "The VC-dimensions of finite automata with n states", *Proc. 4th Intl. Workshop on Algorithmic Learning Theory,* Springer-Verlag Lecture Notes on Artificial Intelligence 744, 328–341.

[KV94] M.J. Kearns and U.V. Vazirani, *An Introduction to Computational Learning Theory*, MIT Press, 1994.

[Pit89] L. Pitt, "Inductive inference, DFAs, and computational complexity". *Proc. International Workshop on Analogical and Inductive In-*

ference AII-89, 18–42. Springer-Verlag Lecture Notes on Artificial Intelligence 397, 1989.

[RS93] R.L. Rivest and R.E. Schapire, "Inference of finite automata using homing sequences", *Information and Computation*, Vol. 103, 1993. 299–347.

[Ron95] D. Ron, *Automata Learning and its Applications,* Ph. D. Thesis, Hebrew University, 1995.

[Sch92] R.E. Schapire, *The Design and Analysis of Efficient Learning Algorithms.* MIT Press, 1992.

[Tze92] W. Tzeng, "Learning probabilistic automata and Markov chains via queries", *Machine Learning,* Vol. 8, 1992, 151–166.

[Wat94] O. Watanabe, "A framework for polynomial time query learnability", *Mathematical Systems Theory,* Vol. 27, 1994, 211–229.