An Optimal Parallel Algorithm for Learning DFA

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

July 6, 1994

Abstract

Sequential algorithms given by Angluin (1987) and Schapire (1992) learn deterministic finite automata (dfa) exactly from Membership and Equivalence queries. These algorithms are feasible, in the sense that they take time polynomial in n and m, where n is the number of states of the automaton and m is the length of the longest counterexample to an Equivalence query. This paper studies whether parallelism can lead to substantially more efficient algorithms for the problem. We show that no CRCW PRAM machine using a number of processors polynomial in n and m can identify dfa in $o(n/\log n)$ time. Furthermore, this lower bound is tight up to constant factors: we develop a CRCW PRAM learning algorithm that uses polynomially many processors and exactly learns dfa in time $O(n/\log n)$.

*Department of Software (L.S.I.) Universitat Politècnica de Catalunya Pau Gargallo 5, 08028 Barcelona, Spain Research was partially supported by the ESPRIT II Program of the EC under contract No. 7141 (project ALCOM II), and by DGICYT, project number PB92–0709. E-mail: {balqui|diaz|gavalda}@lsi.upc.es

†Department of Computer Science, Tokyo Institute of Technology, Meguro-ku, Tokyo 152, Japan. Supported in part by Takayanagi Foundation of Electronics and Science Technology. E-mail: watanabe@cs.titech.ac.jp

1 Introduction

In the last fifteen years there has been an important effort aimed at giving a rigorous and formal theoretical framework to the act of learning concepts using a reasonable amount of resources. Several models have been proposed; among them, the PAC-learning model proposed by Valiant [Val84] and the query-learning model proposed by Angluin [Ang87, Ang88]. In query learning, the learning process must exactly identify the unknown concept by obtaining information through certain types of queries.

For any of the models, it is an interesting question whether parallel machines can speed-up learning significantly. In the PAC model, the issue of parallelism has been addressed by Vitter and Lin [VL88]. In query learning, Bshouty and Cleve [BC92] have studied the complexity of exactly learning boolean formulas in parallel. This paper is concerned with how much help we can expect from parallelism when learning Deterministic Finite Automata, from now on DFA.

The concept class of DFA has been successfully studied in the context of sequential query learning. In the seminal paper of the field [Ang87], Angluin exhibits an algorithm that learns DFA from Membership and Equivalence queries. More efficient algorithms for the same problem were developed later by Rivest and Schapire [RS89, Sch92].

Here, we study first the parallelization of such algorithms, and give a CRCW PRAM algorithm that uses a number of processors polynomial in n and m and runs in time $O\left(\frac{n}{\log n}\right)$, where n is the number of states of the unknown automata and m is the length of the longest counterexample received. This algorithm is in essence a parallelization of Angluin's. However, to obtain the maximum parallelism, a number of important changes have to be introduced in the algorithm as well as in its analysis. In particular, we have to use amortized analysis to prove the running time bound.

Next we show that this algorithm is actually time-optimal, by proving a matching lower bound. More precisely, there is no CRCW PRAM learner running in time $o\left(\frac{n}{\log n}\right)$ with a polynomial number of processors. In other words, even with strong use of parallelism, learning DFA must take close to linear time.

Finally, we show how our time-optimal algorithm can be converted into the PAC setting, obtaining another algorithm that PAC-learns DFA from Membership and Example queries in time $O(n \cdot \log(n+m)/\log n)$.

Both the upper bound and the lower bound are partially based on our recent work [BDGW92], where we give tight bounds on the number of Equivalence queries needed to learn DFA in polynomial time. The same results were obtained independently by Bshouty, Goldman, Hancock, and Matar [BGHM93].

Besides potential implementations in multiprocessor machines, we believe that studying parallelization of learning algorithms is useful as it gives information on the fine structure of learning problems. For example, in the case of dfa, Equivalence queries seem to be the bottleneck for parallelization; our results roughly say that getting counterexamples is an inherently sequential process. It would be interesting to know whether this is a general phenomenon in query learning.

2 Preliminaries

In this paper we follow standard definitions and notations in formal language theory and computational complexity theory; in particular, those for finite automata are used without definition. The reader will find them in standard textbooks such as [HU79].

All our languages are defined over a fixed finite alphabet Σ. For any set A of strings, let $\overline{A}$ denote the complement of A, i.e., $\Sigma^* - A$. The length of a string x is denoted by $|x|$, and ϵ denotes the empty string. Symbols $\Sigma^{\leq m}$ and $A^{\leq m}$ denote the sets $\{x \in \Sigma^* : |x| \leq m\}$ and $\{x \in A : |x| \leq m\}$ respectively. For any sets A and B of strings, $A \triangle B$ denotes the set $(A - B) \cup (B - A)$. The cardinality of a finite set A is written as $\|A\|$.

The type of learning we consider is called *learning via queries* or *query learning*. In it, some *learning algorithm* or *learner* tries to infer a *representation* of some *target concept* by making *queries* to some external *teacher* for the concept. See [Ang87, Ang88, Wat94] for detailed explanations of the model. In our case, representations are encodings of deterministic finite automata over the fixed alphabet Σ. The target concept is some regular language over Σ, fixed during the computation of the learner. The *teacher* is a function mapping query strings to answers.

We consider two types of queries, Membership and Equivalence. The input to a Membership query is a string $x \in \Sigma^*$, and the answer is YES or NO depending on whether $x \in L(M)$ or not, i.e., whether M accepts or rejects x. The input to an Equivalence query is a string $r \in \Sigma^*$ that encodes a dfa in some straightforward way. The answer is either YES if M is equivalent to r, or else the pair (NO, w) where w is any string in $L(M) \triangle L(r)$. String w is called the *counterexample.* Note that each Membership query has a unique answer but that there may be many counterexamples to the same Equivalence query, and hence many different teachers for the same concept. A learner that asks Membership and Equivalence queries is called a *(Mem,Equ)-learner.*

The model of computation used in this paper consists of a number of Random Access

Machines (or processors) that execute synchronously a single instruction flow and communicate through a common memory. Simultaneous reading and writing by several processors on the same memory cell is allowed. Such a model is called the Concurrent–Read, Concurrent–Write Parallel Random Access Machine, or CRCW PRAM. See [JJ92, KR90] for a survey of algorithm design using PRAMs.

Our model of a PRAM is thus a SIMD machine. However, MIMD programs can be run in SIMD machines with only a constant factor slowdown and no additional processors, so our lower bound result is also valid for more general synchronous MIMD models.

Several conventions can be defined to resolve write conflicts. For example, in the COMMON convention, all processors writing into the same cell must write the same value (otherwise the operation is illegal and the program stops). In the ARBITRARY convention, any of the conflicting processors can succeed in writing, and the program must give correct results regardless of the choice. In the PRIORITY convention, the processor that succeeds in writing is the one with highest processor number. It is immediate that COMMON programs work correctly under ARBITRARY protocol, and these also under PRIORITY.

In the usual PRAM model, memory cells contain arbitrary integers and the processors execute a RAM instruction on these integers in unit time. This assumption becomes unrealistic as the integers involved grow. To perform an accurate analysis of resources, we sometimes assume that each memory cell in global memory contains a single bit and that, at each step, a processor reads one bit from global memory and writes another bit depending on the first one and its internal state. (This restriction is not imposed on local memory, and simple instructions for address arithmetic and indirect addressing of global memory are allowed). We call these two models the "integer processor" and "bit processor" model. We assume that a string over Σ is represented as an integer in the first case and as a sequence of consecutive bits in memory in the second case.

Note that we show our upper bound (Theorem 3.1) for the weakest of the CRCW PRAM models mentioned (COMMON, bit processors), and our lower bound (Theorem 4.1) for the strongest (PRIORITY, integer processors).

A processor makes a query to the teacher by presenting it with the memory address of the queried string and the address where the answer should be returned.

We can now define our concept of learnability. A PRAM learner S learns DFA if for every target dfa M and every teacher function T that answers queries according to $L(M)$, S halts and outputs some dfa M' such that $L(M) = L(M')$. Let us stress that learning has to occur even with the least cooperative teacher.

We consider two resources in a PRAM algorithm, the number of processors used and the (parallel) execution time. In a learning algorithm, we measure these resources as a function of two parameters, the number of states of the target dfa and the length of longest counterexample. That is, we allow more resources to learn larger automata, and also if the teacher provides longer counterexamples in the course of the computation. For example, we say that a learner uses a polynomial number of processors meaning polynomial in these two parameters.

Sometimes we apply these definitions to "sequential learners" instead of parallel ones. In this paper, we define a sequential learner as a PRAM learner that uses exactly one processor.

3 The Optimal Algorithm

This section is devoted to the proof of the following theorem.

Theorem 3.1. There is a PRAM (Mem,Equ)-learner for DFA running in time $O\left(\frac{n}{\log n}\right)$ with a polynomial number of processors. This PRAM learner is a uses the CRCW COMMON convention and bit processors.

In the following, let M_* be a given target dfa, and let L_* denote the language accepted by M_*. Without loss of generality, we may assume that M_* is the minimum dfa for L_*. Let also n_* be the number of states in M_*.

We design an algorithm as claimed in the theorem that takes n_* as input, i.e., knows n_* before starting the computation. In Appendix C we explain how to modify the algorithm so that this prior knowledge is not needed.

The algorithm is strongly based on Angluin's sequential algorithm [Ang87]. Before presenting it formally, let us discuss the problems that arise when trying to parallelize it optimally.

Angluin's algorithm keeps a two-dimensional table with all experiments performed so far on the target dfa. It can be rewritten as a loop in which each iteration does one of the following: (1) build a dfa out of the table, ask an Equivalence query, and process the counterexample, or (2) using Membership queries, find that the table is not closed or not consistent, then add a new row or a new column, hence adding a new state to the table. The goal is to restructure the algorithm so that at most $O\left(\frac{n_*}{\log n_*}\right)$ iterations are needed, and each iteration takes $O(1)$ time on a CRCW PRAM. Note that this latter requirement explains why the number of processors depends on m and not only on n_*, unlike the running time.

To reduce the number of iterations of type (1), we start from the algorithm in [BDGW92, BGHM93] making no more than $O\left(\frac{n_*}{\log n_*}\right)$ Equivalence queries. To reduce the number of iterations of type (2), the first idea is to try in parallel all possible outcomes of the next $\log n_*$ iterations of Angluin's algorithm; then, seeing all answers, select the correct outcome and discard all others to avoid exponential blow-up. Ideally, this should be done in constant time and add about $\log n_*$ states to the table.

Two problems arise, however. One is that finding the actual outcome followed by Angluin's algorithm seems to require more than constant time, due to the adaptive nature of the Membership queries. A greedy solution is to blindly keep all new rows and columns, and purge the table of all rows and columns that seem redundant at this moment. This leads to the second problem: because of this strategy, it may be false that as many as $\log n_*$ states are added at every iteration. Still, doing some sort of amortized analysis. we can show that $\Omega(\log n_*)$ states must be added in average.

Now we present the algorithm in detail. The following notions are inherited from [Ang87].

(Observation Table)

An *observation table* is a 3-tuple (S, E, T), where:

- $S \subseteq \Sigma^*$ is a finite, prefix-closed set of strings;
- $E \subseteq \Sigma^*$ is a finite, suffix-closed set of strings;
- T is a two-dimensional table with rows indexed by strings S, columns indexed by strings in E, and entries whose value is 0 or 1.

For any row s and column e, the value of entry (s, e) in T is denoted as $T[s, e]$. In our learning algorithm, $T[s, e]$ is set to 0 if $s \cdot e$ is not in the target set L_*, and is set to 1 if it is in L_*. In general, for any strings w and v, let $L_*[w, v]$ denote the membership of $w \cdot v$ to L_*. For a string $s \in S$, $row(s)$ is the row vector defined by $(T[s, e])_{e \in E}$, and for a string $e \in E$, $col(e)$ is the column vector defined by $(T[s, e])_{s \in S}$. Similarly, for any string w, $row_*(w)$ is the row vector defined by $(L_*[w, e])_{e \in E}$.

To describe the parallel algorithm, we introduce some notations. A string $s \in S$, or $row(s)$, is called *essential* if there is no s' lexicographically smaller than s such that $row(s) = row(s')$. We say that *column e separates rows s_1 and s_2* if $T[s_1, e] \neq T[s_2, e]$, and we say that $e \in E$, or $col(e)$, is *essential* if it is the lexicographically smallest column that separates some two rows in the table. Rows or columns that are not essential are called *nonessential.*

(Consistent DFA)

We say that a dfa M is *consistent* with table (S, E, T) if for every $s \in S$ and every $e \in E$, $T[s, e] = 1$ if and only if M accepts $s \cdot e$.

(Closed and Self-Consistent Table)

We say that table (S, E, T) is *closed* if for every $s \in S$ and every $a \in \Sigma$, there is some $s' \in S$ such that $row_*(s \cdot a) = row(s')$. (S, E, T) is *self-consistent* if for every $s_1, s_2 \in S$, $row(s_1) = row(s_2)$ implies that $row_*(s_1 \cdot a) = row_*(s_2 \cdot a)$ for all $a \in \Sigma$.

For any closed and self-consistent table (S, E, T) we can define an associated dfa $M(S, E, T) = \langle Q, \Sigma, \delta, q_0, F \rangle$ as follows:

- $Q = \{ \ row(s) \ : \ s \in S \ \}$;
- $q_0 = row(\epsilon)$;
- $F = \{ \ row(s) \ : \ s \in S \quad \wedge \quad T[s, \epsilon] = 1 \ \}$;
- $\delta(row(s), a) = row_*(s \cdot a)$, for every $s \in S$ and $a \in \Sigma$.

State q_0 is guaranteed to exist because S is prefix-closed. Entry $T[w, \epsilon]$ exists for each $w \in S$ because E is suffix-closed. Transition function δ is well defined since the table is closed and self-consistent. In particular, because (S, E, T) is closed, every $row_*(s \cdot a)$ corresponds to some $row(s')$, $s' \in S$. Clearly, the number of states of $M(S, E, T)$ is the same as that of essential rows in (S, E, T).

(Algorithm *POLFA*)

Figure 1 presents the learning algorithm *POLFA* ("*P*arallel *O*ptimal *L*earner for *F*inite *A*utomata") that achieves the time-optimal upper bound.

We first detail the meaning and implementation of some of the lines in the algorithm.

Line 1 "build initial table": This means setting $S = \{\epsilon\}$ and $E = \{\epsilon\}$, and then filling in entry $T[\epsilon, \epsilon]$ of T by asking one Membership query "$\epsilon \in L_*$?".

Line 3 "expand table (S, E, T) by $h = \lceil \log n_* \rceil$": This means expanding table (S, E, T) and making an *auxiliary table* (S, E, T_{aux}) by asking Membership queries. Table T_{aux} is a two-dimensional table that is used to check whether (S, E, T) is closed and self-consistent in line 5. In a way similar to T, T_{aux} keeps $L_*[s \cdot a, e]$ for every $s \in S$, $a \in \Sigma$, and $e \in E$.

The detail of line 3 is as follows:

1. build initial table (S, E, T);
2. **while** true **do**
3. expand table (S, E, T) by $h = \lceil \log n_* \rceil$;
4. remove useless rows and columns from (S, E, T);
5. **if** (S, E, T) is closed and self-consistent **then**
6. build $M(S, E, T)$;
7. ask $M(S, E, T)$ as Equivalence query;
8. **if** answer = YES **then**
9. **output** $M(S, E, T)$ and **halt**
10. **else** (i.e., answer = $\langle \mathrm{NO}, w \rangle$ for some w)
11. add w to (S, E, T) **fi fi**
12. **od**.

Figure 1: Learning Algorithm *POLFA*

for all pairs (s, u) with $s \in S$ and $u \in \Sigma^{\leq h} - \{\epsilon\}$ **do in parallel** add $s \cdot u$ to S;
for all pairs (e, v) with $e \in E$ and $v \in \Sigma^{\leq h} - \{\epsilon\}$ **do in parallel** add $v \cdot e$ to E;
for all pairs (s, e) with $s \in S$ and $e \in E$ **do in parallel**
  fill the entry $T[s, e]$ of T by asking Membership query "$s \cdot e \in L_*$?";
for all triples (s, a, e) with $s \in S$, $a \in \Sigma$, and $e \in E$ **do in parallel**
  fill the entry $T_{\mathrm{aux}}[s \cdot a, e]$ of T_{aux} by asking Membership query "$s \cdot a \cdot e \in L_*$?";

Line 4 "remove useless rows and columns": This means removing from S all elements such that: (i) they were not in S before executing line 3, (ii) they are nonessential, and (iii) they are only prefixes of nonessential elements in S. Remove from T all rows indexed by elements removed from S. All in all at most one representative for each row is left, if we consider only the elements of S that are not prefixes of other elements in S.

Useless columns are removed similarly. That is, "remove useless columns" means removing from E all elements such that: (i) they were not in E before executing line 3, (ii) they are nonessential, and (iii) they are only suffixes of nonessential elements in E. Remove from T all columns indexed by elements removed from E.

Note that, by condition (i), an element added to S or E in line 3 is either removed immediately after at line 4, or else it is never removed in later iterations.

$$\begin{array}{l}
(S_{1,0}, E_{1,0}, T_{1,0}) \to \overbrace{(S_{1,1}, E_{1,1}, T_{1,1}) \ \to \ \cdots \ \to \ (S_{1,h}, E_{1,h}, T_{1,h})}^{h \text{ expansions at line 3}}, \\
(S_{2,0}, E_{2,0}, T_{2,0}) \to (S_{2,1}, E_{2,1}, T_{2,1}) \ \to \ \cdots \ \to \ (S_{2,h}, E_{2,h}, T_{2,h}), \\
\qquad\qquad \vdots \\
(S_{I,0}, E_{I,0}, T_{I,0}) \to (S_{I,1}, E_{I,1}, T_{I,1}) \ \to \ \cdots \ \to \ (S_{I,h}, E_{I,h}, T_{I,h})
\end{array}$$

Figure 2: Execution of *POLFA* viewed as sequences of expansions

---

Line 5 "if (S, E, T) *is closed and self-consistent"*: The test can be done just following the definition by using auxiliary table (S, E, T_{aux}).

Line 10 "add w *to* (S, E, T)*":* This means adding u to S as a new row of T, for each $u \notin S$ prefix of w, then filling in the new entries with Membership queries.

(Correctness of *POLFA*)

Consider the execution of the algorithm above when target dfa M_* has n_* states. We prove that the algorithm terminates in $O(n_*/h) = O(n_*/\log n_*)$ iterations, and that after these many iterations it holds $M(S, E, T) = M_*$, i.e., M_* has been identified.

For the following technical discussion, we introduce some more notions. For any table (S, E, T), the *k-expansion* of (S, E, T) is a table (S', E', T') (or building a table (S', E', T')) such that $S' = S \cdot \Sigma^{\leq k}$, $E' = \Sigma^{\leq k} \cdot E$, and T' is consistent with L_*. When $k = 1$, we omit "1-" and simply say *expansion.*

Although line 3 in the algorithm computes the h-expansion of T in parallel, we can regard this as a sequence of h expansions. More precisely, we view any execution of *POLFA* taking I iterations as I sequences of expansions of tables. See Figure 2. In the figure, each table $(S_{i+1,0}, E_{i+1,0}, T_{i+1,0})$ is obtained by removing useless rows and columns in $(S_{i,h}, E_{i,h}, T_{i,h})$, and possibly adding a counterexample.

We prove that the number of iterations I is at most $(2n_* + h)/h$ by showing that the number of expansions in the trace of *POLFA* is at most $2n_* + h$. In the following, let us assume that observation tables are consistent with the target L_*.

Let (S', E', T') be the expansion of (S, E, T). We say that *the expansion of* (S, E, T) *is flat* if no essential row is introduced in (S', E', T'), or more precisely, both (S, E, T) and (S', E', T') have the same number of essential rows.

The total number of expansions in the execution of *POLFA* is given by the following two lemmas, whose proofs are given in Appendix A.

Lemma 3.2. The total number of non-flat expansions in the execution of *POLFA* is at most $n_* - 1$.

Lemma 3.3. The total number of flat expansions in the execution of *POLFA* is at most $n_* + h$.

This completes the proof that *POLFA* correctly learns *DFA* in $O(n_*/\log n_*)$ iterations.

(Conclusion of proof)

To conclude the proof of Theorem 3.1, it remains to show that 1) each iteration takes constant time using a polynomial number of CRCW processors, and 2) it is possible to learn in almost the same time without knowing n_*. Proofs for these two facts are given in Appendix B and Appendix C, respectively.

4 Lower Bound

In this section we prove that the algorithm in the previous section is essentially time-optimal, at least among those using a feasible (i.e., polynomial) number of processors.

Theorem 4.1. There is no CRCW PRAM (Mem,Equ)-learner for DFA running in time $o\left(\frac{n}{\log n}\right)$ with a polynomial number of processors. This holds even for the PRIORITY convention and the integer processor model.

In the proof we use two main ideas. First, a lower bound on the number of Equivalence queries needed for *sequential* learning, which we obtained in a previous work [BDGW92]. Second, a trick introduced by Bshouty and Cleve [BC92] to reduce the number of Equivalence queries when transforming a parallel into a sequential learner.

The following result was proved in [BDGW92] (Corollary 4.7), and independently obtained by Bshouty, Goldman, Hancock, and Matar [BGHM93].

Theorem 4.2. There exists no polynomial-time sequential (Mem,Equ)-learner S for *DFA* making $o\left(\frac{n}{\log n}\right)$ Equivalence queries.

In fact, this is only a special case of more general trade-offs between Membership and Equivalence queries shown in [BDGW92, BGHM93].

We prove Theorem 4.1 by contradiction. Assume that there is a CRCW PRAM that learns dfa in $o\left(\frac{n}{\log n}\right)$ parallel time and using a $p(n)$ processors, with p a polynomial.

We use this PRAM to build a sequential learner for dfa that makes $o\left(\frac{n}{\log n}\right)$ Equivalence queries and runs in polynomial time. This contradicts Theorem 4.2.

So take the hypothetical PRAM, without loss of generality a SIMD. The sequential learner S simulates sequentially steps 1, 2, ... of the PRAM. If, at step i, processors compute, S simply updates the simulated PRAM memory. If, at step i, processors ask Membership queries, S asks all of these queries sequentially and returns to each processor its corresponding answer. If, at step i, processors ask Equivalence queries $r_1, r_2, \ldots, r_{p(n)}$, it uses the technique in [BC92] to answer all of them by making $p(n) - 1$ Membership queries and a single Equivalence query.

To do this, find a string in the symmetric difference of $\Phi(r_1)$ and $\Phi(r_2)$. This can be done in polynomial time because r_1 and r_2 are dfa. Ask this string as a Membership query. Depending on the answer, the string can be used as a counterexample for either r_1 or r_2. Let r_{i_1}, $i \in \{1, 2\}$, be the query that did not get a counterexample. Find another string in $\Phi(r_{i_1}) \triangle \Phi(r_3)$, ask it for membership, obtain a counterexample for one of the dfa, and call the other r_{i_2}, $i_2 \in \{1, 2, 3\}$; and so on. After $p(n)-1$ rounds, we are left with only one query $r_{i_{p(n)-1}}$, $i_{p(n)-1} \in \{1, \ldots, p(n)\}$, whose counterexample is obtained with an Equivalence query.

Simulating a parallel step of the PRAM clearly takes polynomial time, and there are $o(n/\log n)$ parallel steps. Hence, S runs in polynomial time. Also, S makes at most one Equivalence query per simulated step. This gives us the desired contradiction.

Finally, note that the proof of the lower bound applies in fact to PRAMs with unit-cost string operations, i.e., if we assume that a PRAM processor can compare, concatenate, and extract prefixes of arbitrarily long strings in a single step. Also, the proof works for every Write-conflict rule that can be computed in polynomial time.

5 Parallel PAC

To conclude, let us argue that our learner can be turned into a parallel PAC algorithm with Membership queries. It suffices to use the technique developed by Angluin in the original paper on learning DFA from queries [Ang87]. She showed how to replace Equivalence queries with a source of examples, at the expense of transforming an algorithm that learns exactly into another one that learns only approximately with high probability.

The idea is as follows. When the exact learning algorithm asks an Equivalence query, the PAC-learner instead asks for a certain number of labeled examples. Then it tests whether any of the examples is a counterexample for the query. If so, computation proceeds with that counterexample. Otherwise, the learner assumes that the answer to the Equivalence query is YES and stops. It is shown in [Ang87] that a polynomial number of examples is enough to guarantee a small error probability, and in fact achieve learning in the PAC sense.

The observation now is that this procedure can be parallelized very well. Asking for a polynomial number of examples takes constant parallel time. The problem of determining whether a given dfa of n states accepts a given string of length m can be solved in $O(\log(n+m))$ time. Hence, applying this method to our $O(n/\log n)$-time algorithm gives a PAC algorithm for learning DFA using Membership queries, a polynomial number of processors, and $O(n \cdot \log(n+m)/\log n)$ time, where m is the length of the longest labelled example it receives.

References

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

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

[BDGW92] J.L. Balcázar, J. Díaz, R. Gavaldà, and O. Watanabe, "A note on the query complexity of learning DFA", in *Proc. Third Workshop on Algorithmic Learning Theory,* Japanese Society for Artificial Intelligence, 1992, pp. 53–62. Final version to appear in *New Generation Computing.*

[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, pp. 513–522.

[BGHM93] N.H. Bshouty, S.A. Goldman, T.R. Hancock, and S. Matar, "Asking queries to minimize errors", in *Proc. Sixth Annual ACM Conference on Computational Learning Theory,* ACM Press, 1993, 41–50.

[HU79] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[JJ92] J. Ja'Ja', *An Introduction to Parallel Algorithms,* Addison-Wesley, 1992.

[KR90] R.M. Karp and V. Ramachandran, "Parallel algorithms for shared-memory machines", in *Handbook of Theoretical Computer Science,* vol. A, J. van Leeuwen (editor), Elsevier / MIT Press, 1990, pp. 869–942.

[RS89] R.L. Rivest and R.E. Schapire, "Inference of finite automata using homing sequences", in *Proc. 21st Annual ACM Symposium on the Theory of Computing,* ACM Press, 1989, pp. 411–420.

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

[Val84] L.G. Valiant, "A theory of the learnable", *Communications of the ACM,* Vol. 27, 1984, pp. 1134–1142.

[VL88] J.S. Vitter and J.-Y. Lin, "Learning in parallel", in *Proc. 1988 Workshop on Computational Learning Theory,* Morgan Kaufmann Publishers, 1988, pp. 106–124.

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

Appendix A. Proofs of Lemmas 3.2 and 3.3

We use the following three facts concerning observation tables. The first one is Lemma 5 in [Ang87], the second one is Theorem 1 in [Ang87], and the third one easily verified. Therefore, we omit their proofs here.

Fact A.1. Let (S, E, T) be an observation table that has n essential rows. Then any dfa consistent with (S, E, T) must have at least n states.

Fact A.2. Let (S, E, T) be a closed and self-consistent observation table. Then the acceptor $M(S, E, T)$ is the minimum dfa consistent with (S, E, T).

Fact A.3. For any observation table (S, E, T), (S, E, T) is closed and self-consistent if and only if the expansion of (S, E, T) is flat.

Now Lemma 3.2 follows from Fact A.1.

Proof of Lemma 3.2. Suppose that n_* non-flat expansions are made during the execution. Then after the n_*th expansion, observation table (S, E, T) has at least $n_* + 1$ essential rows because the number of essential rows increases at least by one by each non-flat expansion. Then from Fact A.1, any dfa consistent with (S, E, T) has at least $n_* + 1$ states, which contradicts that M_*, which has n_* states, is consistent with (S, E, T). $\square$

Bounding the number of flat expansions is more difficult. We need the following lemma, that generalizes Lemma 4 in [Ang87] and whose proof is similar to that of Lemma 4.2 in [BDGW92].

Lemma A.4. Let (S, E, T) be any closed and self-consistent observation table whose next k expansions are all flat. Then any acceptor that is consistent with the k-expansion of (S, E, T) but inequivalent to $M(S, E, T)$ must have at least k more states than $M(S, E, T)$.

Proof. Define $M = M(S, E, T) = (Q, q_0, F, \delta)$ and $n = \|Q\|$. Let $M' = (Q', q_0', F', \delta')$ be an acceptor consistent with the k-expansion of (S, E, T), with $\|Q'\| < n + k$. We will show that M and M' are isomorphic and thus equivalent. We assume without loss of generality that M' is minimum. That is, every state of M' can be reached from q_0' and no two states in M' are equivalent.

To show the isomorphism, we use the following definitions.

- For a string $s \in S \cdot \Sigma^{\leq h}$, let $row(s)$ denote the finite function mapping each $e \in E$ to $T_k(s,e)$, where T_k is the k-expansion of T.
- For every $q' \in Q'$, define $row'(q')$ to be a finite function from E to $\{0,1\}$ such that $row'(q')(e) = 1$ iff $\delta'(q',e) \in F'$.
- For every $s \in S$, define $f(s) = \delta'(q'_0, s)$. Note that $row'(f(s)) = row(s)$, from the assumption that M' is consistent with T_k. In fact, for every $u \in \Sigma^{\leq k}$, $row'(\delta'(f(s),u)) = row(s \cdot u)$.
- For every $q \in Q$, define $\phi(q) = \{f(s) : row(s) = q\}$.

The following sequence of claims shows that ϕ defines a bijection between Q and the set $\{\{q'\} : q' \in Q'\}$. Clearly, then we can transform ϕ into a bijection from Q to Q', which turns out to be the desired isomorphism.

Claim 1. $\|Range(f)\| \geq n$.

Proof. From the above remark, it is easy to see that $f(s_1) = f(s_2)$ implies $row(s_1) = row(s_2)$. On the other hand, there are n different rows $row(s)$ in T (i.e., the states of M); hence, there must be at least n different $f(s)$. Thus, $\|Range(f)\| \geq n$. $\Box$ Claim 1

Intuitively, the next claim states that for any two states in M' there is already some string in $\Sigma^{\leq k} \cdot E$ that proves them different.

Claim 2. For any two different states q'_1 and q'_2 in Q', $row'(q'_1) \neq row'(q'_2)$.

Proof. By induction on the length of a string x witnessing that q'_1 and q'_2 are not equivalent.

If x is the empty string, functions $row'(q'_1)$ and $row'(q'_2)$ differ when applied to x.

Consider the case where x is not empty. For the first symbol $a \in \Sigma$ of x, define $q'_3 = \delta'(q'_1, a)$ and $q'_4 = \delta'(q'_2, a)$. Then $q'_3 \neq q'_4$ (otherwise x is not a witness), and a string shorter than x witnesses that q'_3 and q'_4 are not equivalent. By induction hypothesis, $row'(q'_3)$ is different from $row'(q'_4)$.

On the other hand, because there are less than $n+k$ states in $Q' - Range(f)$ (since $\|Range(f)\| \geq n$ from Claim 1), q'_1 and q'_2 must be reachable from states in $Range(f)$ with a path of length less than k. More precisely, there exist u, v in $\Sigma^{<k}$ and s_1, s_2 in S such that $q'_1 = \delta'(f(s_1), u)$ and $q'_2 = \delta'(f(s_2), v)$. Then

$$row(s_1 \cdot ua) = row'(q'_3) \neq row'(q'_4) = row(s_2 \cdot va)$$

(the equalities are true because M' is consistent with T_k and ua and va are in $\Sigma^{\leq k}$).

On the other hand, it must happen that $row(s_1 \cdot u) \neq row(s_2 \cdot v)$; otherwise, T_k has one more row than T (namely, the result of separating $row(s_1 \cdot u)$ and $row(s_2 \cdot v)$ by a). But $row(s_1 \cdot u) = row'(q'_1)$ and $row(s_2 \cdot v) = row'(q'_2)$, again because M' is consistent with T_k and u and v are in $\Sigma^{\leq k}$. Hence $row'(q_1) \neq row'(q_2)$. $\square$ Claim 2

Now using Claim 2, we prove that ϕ is a bijection from Q to $\{\{q'\} : q' \in Q'\}$ in the following way.

Claim 3.
(1) For every $q \in Q$, $\|\phi(q)\| \leq 1$,
(2) $Q' \subseteq Range(f)$, and
(3) ϕ is a bijection from Q to $\{\{q'\} : q' \in Q'\}$.

Proof. Part (1): Suppose that some $\phi(q)$ has two different states q'_1 and q'_2 in Q'. By Claim 2, $row'(q'_1) \neq row'(q'_2)$. Since both q'_1 and q'_2 are in $\phi(q)$, there are strings s_1, s_2 in S such that $q'_1 = f(s_1)$, $q'_2 = f(s_2)$, and $row(s_1) = row(s_2) = q$. However, we have $row(s_1)$ $(= row'(f(s_1))) = row'(q'_1) \neq row'(q'_2) = (row'(f(s_2)) =)\ row(s_2)$. A contradiction.

Part (2): Take any q' in Q'. By an argument as in Claim 2, q' must be reachable from some state $f(s_1)$ using a string $u \in \Sigma^{<k}$, that is, $row'(q') = row(s_1 \cdot u)$. Then there is some $s_2 \in S$ such that $row(s_1 \cdot u) = row(s_2)$, or otherwise T_k has one more row than T. Therefore, $row'(q') = row(s_2) = row'(f(s_2))$. By Claim 2 this means $q' = f(s_2)$, and thus $q' \in Range(f)$.

Part (3): From the above part (2) and our definitions, we have

$$Q' \subseteq Range(f) \subseteq \bigcup_{q \in Q} \phi(q) \subseteq Q'.$$

Note also that $\|Range(f)\| \geq n$ (but $\|Q\| = n$) and that $\|\phi(q)\| \leq 1$ for every $q \in Q$. Thus, it must happen that $\|\phi(q)\| = 1$ for every $q \in Q$ and that all $\phi(q)$ are different. Furthermore, every $q' \in Q'$ has some $q \in Q$ such that $q' \in \phi(q)$, which is in fact $\{q'\} = \phi(q)$. $\square$ Claim 3

Finally we must show that ϕ is not only a bijection but also an isomorphism between M and M'. That is, it carries q_0 to q'_0, it preserves δ to δ', and it carries F to F'. But having proved that Q and Q' have the same cardinality, the rest is exactly as in [Ang87].
$\square$

Proof of Lemma 3.3. Again, we view all table expansions as in Figure 2. We group them into sequences of *adjacent* tables, where adjacency is defined as follows:

- Tables $(S_{i,j}, E_{i,j}, T_{i,j})$ and $(S_{i,j+1}, E_{i,j+1}, T_{i,j+1})$ are adjacent if and only if the expansion from one to the other is flat.
- Tables $(S_{i,h}, E_{i,h}, T_{i,h})$ and $(S_{i+1,0}, E_{i+1,0}, T_{i+1,0})$ are adjacent if and only if no counterexample is added between them. Note that there is no expansion here, and that removing rows and columns alone does not prevent adjacency.

Let the first sequence of adjacent tables start from table A_1 and end in table B_1, let the second start from A_2 and end in B_2, ..., and let the last be from A_m to B_m. Note that each B_i is either followed by a non-flat expansion, or else the end of an iteration and a counterexample is inserted immediately after.

Define k_i as the number of expansions from A_i to B_i.

Based on the previous facts and lemmas, we observe the following:

- Each A_i is closed and self-consistent, because it has a flat expansion (Fact A.3). Thus, for every i, $M(A_i)$ is well defined and is the minimum dfa consistent with A_i (Fact A.2).
- Acceptor $M(A_{i+1})$ is consistent with table A_{i+1}, so it is also consistent with tables A_i and B_i. This is because all strings indexing rows and columns of A_i and B_i are still in A_{i+1}.
- Also $M(A_i)$ and $M(A_{i+1})$ are not equivalent. Indeed, if B_i is followed by a non-flat expansion, then A_{i+1} has at least one essential row more than B_i, and by Fact A.1 $M(A_{i+1})$ at least one state more than $M(B_i)$; otherwise, the algorithm adds some counterexample to table B_i, and this counterexample is classified differently by $M(B_i)(= M(A_i))$ and $M(A_{i+1})$.
- Therefore, by Lemma A.4, $M(A_{i+1})$ has at least k_i more states than $M(A_i)$.

Thus, $M(A_m)$ has at least $k_1 + \cdots + k_{m-1}$ states. Acceptor M_* is consistent with table A_m so, by Fact A.1, $k_1 + \cdots + k_{m-1} \leq n_*$.

Finally, observe that k_m is at most h. Otherwise, the end of some iteration must occur between A_m and B_m. At that moment the observation table is closed and self-consistent so *POLFA* asks an Equivalence query and a counterexample is inserted in the table. This contradicts the assumption that all tables between A_m and B_m are adjacent.

Hence $k_1 + \cdots + k_m \leq n_* + h$, proving the bound on the number of flat expansions.

□

Appendix B. Processor Complexity of *POLFA*

Each line of *POLFA*, except for 3, 4, 5, 6, and 11, takes constant time even if executed sequentially (i.e., using a single processor). We claim that even these lines can be implemented in constant time, using Concurrent Reads and Concurrent Writes and a number of processors that is polynomial in $\|S|$ and $\|E\|$.

As an example, we discuss in detail the implementation of the step "remove useless rows" in line 4. Other steps use similar techniques of CRCW PRAM programming. This part is programmed as follows:

> **for all** $u \in S$ **do in parallel**
>   **for all** pairs $v \in S$, $e \in E$ **do in parallel**
> $rowsequal[u, v] := true$;
>   **if** $T[u, e] \neq T[v, e]$ **then** $rowsequal[u, v] := false$;
> $rowessential[u] := true$;
>   **for all** $v \in S$ **do in parallel**
>   **if** $v < u$ **and** u added to S in this iteration **and** $rowsequal[u, v]$ **then**
> $rowessential[u] := false$;
>   **for all** $v \in S$ **do in parallel**
>   **if** u prefix of v **and** $rowessential[v]$ **then** $rowessential[u] := true$;
>   **if not** $rowessential[u]$ **then**
>   remove u from S and mark row u of T as "deleted"

Clearly, this algorithm takes constant time and $O(\|S\|^2 \cdot \|E\|)$ processors. Note that in steps with Concurrent Write, all processors writing into the same cell write the same value, either *true* or *false*. That is, the program obeys the COMMON convention.

To establish the true polynomial bound on the number of processors, it suffices to bound $\|S\|$ and $\|E\|$ by a polynomial in n_* and m, the length of the longest counterexample received so far.

Note that elements can be added to $S \cup E$ in three ways:

- In line 1, ϵ is added.
- In line 3, $S \cdot \Sigma^{\leq h}$ and $\Sigma^{\leq h} \cdot E$ are added.
- In line 11, strings of length at most m are added.

Since line 3 is executed at most $O(n_*/\log n_*)$ times, strings in $S \cup E$ have length at most $m + O(n_*/\log n_*) \cdot h = O(m + n_*)$.

Next, observe that at the beginning of an iteration S contains the following rows:

- By Fact A.1, at most n_* essential rows from the previous iteration.
- All the prefixes of the strings indexing essential rows, i.e., $O(m + n_*)$ per essential row.
- Plus at most the $m + 1$ prefixes of the counterexample inserted at the last iteration.

So at the beginning of each iteration we have $\|S\| \leq n_* \cdot O(m+n_*)+m+1 = O(mn_*+n_*^2)$. Also, at the beginning of the iteration $\|E\|$ contains only essential columns. Recalling that a column is essential if it is the first one separating a pair of rows, the number of essential rows is bounded above by the number of pairs of rows, i.e., $\|S\|^2$.

After the expansion at line 3, T contains $\|S\| \cdot \|\Sigma\|^{\leq h}$ rows and $\|E\| \cdot \|\Sigma\|^{\leq h}$ columns. But Σ is fixed and h is $\lceil \log n_* \rceil$, so we have $\|\Sigma\|^{\leq h} = n_*^{O(1)}$. Hence, both $\|S\|$ and $\|E\|$ are polynomial in n_* and m at any moment of the execution of *POLFA*.

Let us note that in the analysis above we assumed that processors can move a string, compare two strings, or test whether a string is prefix of another in constant time. However, these operations can be implemented in constant time in a COMMON CRCW PRAM using a number of bit processors linear in the length of the operands. So we can reprogram algorithm *POLFA* in the bit processor model multiplying by at most $O(m+n_*)$ the number of processors.

Appendix C. When n_* is not known

We show how to remove the assumption that n_* is given as input. We substitute line 3 of *POLFA* by the following:

> $n_i :=$ max(2,number of essential rows in T);
> expand table (S, E, T) by $h = \lceil \log n_i \rceil$;

That is, instead of using a fixed h to expand the table we let it grow with the size on the table.

Clearly, the correctness of the algorithm is not affected and the number of processors is not larger than in the original *POLFA*. We only have to show that the number of iterations is still $O(n_*/\log n_*)$.

Consider the number of expansions that can occur for each value of h. Each non-flat expansion adds at least one row, so as in Lemma 3.2, there can be at most 2^k non-flat expansions while $h = k$. With an argument as in Lemma 3.3, we can show that at most $2^k + k$ flat expansions occur while $h = k$. Since an iteration with $h = k$ captures exactly k expansions, the total number of iterations is bounded by

$$\sum_{k=1}^{\lceil \log n_* \rceil} \frac{(2^k + k) + k}{k} = \sum_{k=1}^{\lceil \log n_* \rceil} \frac{2^k}{k} + 2 \cdot \lceil \log n_* \rceil.$$

It is easy to prove by induction on m that

$$\sum_{k=1}^{m} \frac{2^k}{k} < 3 \cdot \frac{2^m}{m}$$

so we conclude that the number of iterations is less than

$$3 \cdot \frac{2^{\lceil \log n_* \rceil}}{\lceil \log n_* \rceil} + 2 \cdot \lceil \log n_* \rceil \leq 6 \cdot \frac{n_*}{\log n_*} + 2 \cdot \lceil \log n_* \rceil.$$