Results of the Abbadingo One DFA Learning Competition and a New Evidence Driven State Merging Algorithm

Kevin J. Lang
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
kevin@research.nj.nec.com

Barak A. Pearlmutter
Comp Sci Dept, FEC 313
Univ of New Mexico
Albuquerque, NM 87131
bap@cs.unm.edu

Rodney A. Price[*]
Emtex
Milton Keynes, England
rod@emtex.com

May 8, 1998

Abstract

This paper first describes the structure and results of the Abbadingo One DFA Learning Competition. The competition was designed to encourage work on algorithms that scale well—both to larger DFAs and to sparser training data. We then describe and discuss the winning algorithm of Rodney Price, which orders state merges according to the amount of evidence in their favor. A second winning algorithm, of Hugues Juillé, will be described in a separate paper.

Part I
Abbadingo

1 Introduction

The Abbadingo One DFA Learning Competition was organized by two of the authors (Lang and Pearlmutter) and consisted of a set of challenge problems posted to the internet and token cash prizes of $1024. The organizers had the following goals:

- Promote the development of new and better algorithms.
- Encourage learning theorists to implement some of their ideas and gather empirical data concerning their performance on concrete problems which lie beyond proven bounds, particulary in the direction of sparser training data.
- Encourage empiricists to test their favorite methods on target concepts with high Kolmogorov complexity, under strict experimental conditions that permit comparison of results between different groups by eliminating the possibility of hill climbing on test set performance.

1.1 The learning task

The task of the Abbadingo One competition was DFA learning from given training data consisting of both positive and negative examples. The learner was provided with a set of training strings that had been labeled

[*] Lang and Pearlmutter ran the Abbadingo competition. Price was one of the winners. Prior to Price's participation in the competition, there was no connection between Price and the Abbadingo administrators.

		dense **training set density**			sparse	lower
		U.B.=IV	III	II	I	bound
small	64	4456	3478	2499	1521	542
target	128	13894	10723	7553	4382	1211
size	256	36992	28413	19834	11255	2676
large	512	115000	87500	60000	32500	5862

Table 1: Training set sizes for the Abbadingo One competition problems.

by an unseen deterministic finite automaton (the target concept), and was required to predict the labels that the target would assign to a set of testing strings. All three of these—the DFA, the training strings, and the testing strings—were drawn from uniform random distributions.

1.2 Some history

DFA learning can be very hard in the worst case. Pitt and Warmuth (1989) proved that it is NP-hard to find a DFA that is consistent with a given set of training strings and whose size is within a polynomial factor of the size of the smallest such DFA. Kearns and Valiant (1989) proved that predicting the output of a DFA can be as hard as breaking cryptosystems that are widely believed to be secure.

However, DFA learning does not seem to be so hard in the average case. Trakhtenbrot and Barzdin (1973) proved that a simple state merging algorithm is guaranteed to find the smallest DFA consistent with a complete training set consisting of *all* strings out to a given length. Lang (1992) showed empirically that this same algorithm can often construct an approximately correct hypothesis from a sparse subset of a complete training set, when both the target concept and training sets are randomly chosen from uniform distributions. Freund *et al.* (1993) proved the approximate learnability of DFA's with worst-case graph structure and randomly labeled states, from randomly chosen training strings.[1]

We note that many papers have been published on the application of generic methods such as neural networks and genetic search to the problem of DFA learning. Unfortunately, this literature has largely focused on tiny benchmarks (the largest target machine in the widely used "Tomita" suite contains five states), so the scalability of the proposed methods is hard to assess.

2 Experimental setup

Abbadingo One used random target DFA's because they have some relevance to the average case, they have high Kolmogorov complexity, and they are easy to generate in any desired size. The procedure for constructing a target concept of nominal size n was: construct a random degree-2 digraph on $\frac{5}{4}n$ nodes, extract the subgraph reachable from the randomly chosen root node, and label the graph's states by flipping a fair coin.

This procedure yields graphs with a distribution of sizes centered near n, and a distribution of depths centered near $2\log_2 n - 2$. The size variation is of no great consequence, but the depth variation would complicate our training set construction, so it was eliminated by selecting only those graphs with a depth[2] of exactly $2\log_2 n - 2$.

A training set for a target of nominal size n consisted of a random sample drawn without replacement from a uniform distribution over the collection of $16n^2 - 1$ binary strings whose length lies between 0 and $2\log_2 n + 3$ inclusively. A testing set was drawn from the remaining strings in this same collection. Training strings were labeled, while testing strings were not.

[1]The Freund *et al.* (1993) theorem concerns a slightly different protocol, in which the learner sees the label of every state that is encountered rather than just the label of the final state.

[2]By analogy to trees, the depth of a DFA is maximum over all nodes x of the length of the shortest path from the root to x.

3 Competition design:

3.1 Target and training set sizes

As shown in table 1, the sixteen Abbadingo One problems represented the cross product of 4 values of a target size parameter and 4 values of a training set density parameter. Both of these parameters influenced the difficulty of the problems. Our intention was to make the target concepts large enough to challenge the empirical learning community, and the training data sparse enough to challenge the theoretical learning community.

The size parameter was simply the nominal size of the target concept. Its four values were 64, 128, 256, and 512 states.

The density parameter took on values from one to four, shown as roman numerals in the tables. A density parameter value p was turned into an actual training set size s by linearly interpolating between rough upper and lower bounds on sample complexity: $s = L + (p/4)(U - L)$. The lower bound L came from the simple counting argument which equates $2^n n^{2n}/(n-1)!$, an estimate of the number of different n-state (binary alphabet) target DFA's, with 2^s, the number of ways of labeling a training set of s strings. The upper bound U was determined by visually inspecting the learning curves for the Trakhtenbrot-Barzdin algorithm which appeared in Lang (1992). In addition, some rounding was performed on the training set sizes for targets of size 512.

Because the problems in column IV were already solvable by the Trakhtenbrot-Barzdin state merging algorithm, an implementation of which we distributed before the competition, these problems were considered practice problems, not official challenge problems.

3.2 Testing protocol

Test set tuning is an insidious problem that afflicts even the well intentioned. The Abbadingo One testing protocol was designed to eliminate this phenomenon. The test set for each problem consisted of 1800 unlabeled strings, none of which appeared in the training set. Proposed labelings were submitted to a testing oracle provided by the Abbadingo web server at http://abbadingo.cs.unm.edu. Instead of providing a score that could be used for hill climbing, the oracle provided only 1 bit of feedback, which told whether or not the accuracy of the labeling was at least 99%. Since this was the threshold at which a problem was considered solved, the feedback bit would always be zero while a participant was working on a problem, so it carried essentially no information.

Thanks to a new cryptographic technique of Joe Kilian's, the testing oracle was implemented without storing the answers anywhere online (Kilian and Lang, 1997). This reduced the temptation to break into the Abbadingo web server.

3.3 Additional rules

Two rules governed the selection of competition winners. The priority rule stated that the first person to solve a problem (to 99% accuracy) would get the credit for solving it. The dominance rule stated that problem A dominates problem B when TrainingSetDensity(A) $\leq$ TrainingSetDensity(B) and NodeCount(A) $\geq$ NodeCount(B).

The winners of the competition would be participants who, at the termination of the competition, had credit for solving problems that were not dominated by other solved problems.

4 Competition results

According to our logs (which do not include accesses to the European mirror site), training data was downloaded from the primary Abbadingo web site by about 460 IP addresses, including many major proxy servers. Proposed test set labelings were submitted from 45 IP addresses, which we estimate corresponds to about 25

		dense **training set size** sparse		
		III	II	I
small	64	Juillé-PBS	Juillé-PBS	Juillé-EDSM+search
target	128	Juillé-PBS	Juillé-PBS	unsolved
size	256	Price-EDSM	Juillé-EDSM	unsolved
large	512	Price-EDSM	Price-EDSM	unsolved

Table 2: The person and algorithm that first solved each of the twelve challenge problems. The data remains available at http://abbadingo.cs.unm.edu

different participants.[3] Nine of the twelve challenge problems were ultimately solved. The person and algorithm that first solved each problem is shown in table 2. The order of events was as follows.

First, Hugues Juillé solved the four problems in the upper left of the table using a parallel beam search technique. Because this method was computationally expensive and didn't scale well to the larger problems, there was a lull in the competition until Rodney Price discovered an evidence driven state merging algorithm (EDSM) that handles sparse data better than previous state merging algorithms, and that has much better time complexity than the beam search method which Juillé had been using. This algorithm,[4] which is discussed in detail below, quickly polished off the problems in columns II and III. Note that according to the competition rules, these results dominated the earlier results of Juillé. However, Price's algorithm could not handle training data as sparse as that in column I. There was another lull in the competition until Juillé solved the smallest problem in column I, using EDSM augmented with some search over its initial decisions.

According to the competition's priority and dominance rules, the two winners were Rodney Price, by virtue of solving problem 512-II, and Hugues Juillé, by virtue of solving problem 64-I.

The three largest problems in column I remain unsolved.

5 Post-competition work

We have done some additional work since the competition. First, we ran EDSM on new random problems lying on the Abbadingo problem grid to discover the algorithm's typical behavior. The results are summarized by table 3. EDSM works well on columns IV, III, and II, whereas Trakhtenbrot-Barzdin can only handle column IV. Both algorithms die on column I.

Second, because differing choices about small details can turn Price's basic idea into many different programs of varying performance, we decided to provide some guidance by defining an official reference version of the EDSM algorithm. This algorithm will be described in part 2 of this paper. We will also describe a couple of optimizations which make the reference algorithm practical without hurting its performance too much, plus Juillé's fast and simple implementation of EDSM using the "blue-fringe" control strategy.

6 Conclusion of part I

The Abbadingo One competition had three goals. The goal of promoting the development of new and better algorithms was clearly satisfied. Both Rodney Price and Hugues Juillé made useful contributions to the state of the art in DFA induction.

Although we have heard a few amusing anecdotes, we have no solid evidence that theorists have empirically explored the limits of their algorithms in the sparse data regime, or that empiricists have carefully measured the scaling properties of their algorithms. We therefore conclude this report by repeating our call for theorists to implement their best ideas, and for experimentalists to try their ideas on problems that are hard enough to really test them.

[3]Participants did not necessarily submit labelings to the Oracle. They could tune their algorithms using cross-validation or their own DFAs drawn from the same distribution. We made no attempt to count such silent participants.

[4]Including a similar program which Juillé coded up after a conversation with Price.

		dense **training set density** sparse				
		IV	III	II	I	**algorithm**
nominal target size	64	2000.0	15.0	2.4	2.1	TB-92
	128	1600.0	21.0	2.6	2.1	TB-92
	256	850.0	8.1	2.1	2.0	TB-92
	512	130.0	13.0	2.2	2.0	TB-92
	64	2700.0	900.0	250.0	2.1	EDSM-97
	128	4500.0	2400.0	720.0	2.1	EDSM-97
	256	6600.0	2500.0	700.0	2.0	EDSM-97
	512	11000.0	6800.0	2300.0	2.0	EDSM-97

Table 3: Median reciprocal error rates of two algorithms on 100 new random instances of each of the 16 Abbadingo One problems. Higher scores are better; the values 2.0 and 100.0 correspond to generalization rates of 50 percent and 99 percent respectively. The latter value is the Abbadingo threshold for considering a problem solved. TB-92 is an implementation of the Trakhtenbrot-Barzdin state merging algorithm. EDSM-97 is an earlier and worse version of the reference algorithm of section 9. It is interesting to note that EDSM is getting better as one moves to lower matrix rows, while TB is getting worse.

Meanwhile, we are preparing a more flexible DFA learning challenge problem generation scheme, and considering other grammar learning tasks that might be appropriate for Abbadingo Two.

Part II
Evidence driven state merging

7 Background

A simple and effective method for DFA induction from positive and negative examples is the state merging method (Trakhtenbrot and Barzdin, 1973; Oncina and Garcia, 1992; Lang, 1992). This method starts with the prefix tree acceptor for the training set and folds it up into a compact hypothesis by merging compatible pairs of states.[5] Two states are compatible when no suffix leads from them to differing labels.

When a state merging algorithm is applied to sparse training data, it can almost never be sure that an apparently compatible merge is truly valid. Thus, most of the algorithm's actions are hopeful guesses, which unfortunately have serious consequences later: each merge introduces new constraints on future merges, and these new constraints will be wrong when an incorrect merge is made.

Because there is a snowballing of right or wrong decisions, it is critically important for the algorithm's early decisions to be correct, and hence a good strategy is to first perform those merges that are supported by the most evidence. Lang (1992) claimed that this consideration supported the choice of breadth-first order for candidate merges, because then the earliest merges must survive the comparison of the largest trees of suffixes.

Higuera, Oncina, and Vidal (1996) suggested that a better strategy is to look at the training data and perform merges exactly in order of the amount of evidence, rather than in a predetermined order that hopefully correlates with that quantity. While this is a very good point, the actual algorithm described in Higuera *et al.* (1996) does not work well on the Abbadingo challenge problems due to a couple of flaws. One was a mistake in the algorithm's control strategy that will be described in section 10. A more serious mistake was the measure of evidence that they proposed, essentially the number of labels on strings that pass through the two candidate nodes. This quantity is only a weak upper bound on evidence, since labeled nodes on one side which line up with unlabeled nodes on the other side have absolutely no value in testing whether the two

[5]Note that state merging frequently introduces non-determinism into the hypothesis, which can then be removed by a determinization procedure that recursively merges the children of the original nodes, as shown in figure 5. In this paper, we always do merging with determinization.

candidate nodes actually represent the same mapping from suffixes to labels.

Rodney Price was able to win the Abbadingo One competition because he realized that a more accurate evidence measure is the number of labels that are tested during a merge.

8 Price's motivation for EDSM

Suppose that a state merging program does m merges, and that each merge is verified by t independent tests, each of which has a probability p of revealing that an incorrect merge is wrong. Let c be the probability that any given one of these m merges is valid, and d be the probability that all of them are valid. Then, $d = c^m$, $1 - c = (1-p)^t$, and finally $t = \log(1 - d^{\frac{1}{m}}) / \log(1-b)$ shows how many tests will suffice to ensure that the whole computation is correct with confidence d.

Blue-fringe state merging algorithms do at most $n(a-1)+1$ merges when constructing an n-state hypothesis over an alphabet of size a (see section 10.2 for a proof). Combining this fact with the calculation of the previous paragraph and the assumption that the label comparisons which occur during a merge are independent tests having a 50 percent chance of revealing an invalid merge, one can see that problems in the top row of the Abbadingo matrix can be solved with confidence .93 by restricting the program to merges that are supported by 10 or more label comparisons. Since the highest scoring initial merges for the top-row problems in columns III, II, and I have scores of 19, 13, and 5 respectively, one would expect this method to work for the first two problems, but not the last, which is exactly what happens.

Note that while one could write a program that is willing to do any merge whose score exceeds the threshold computed above, better performance can be obtained by ignoring the threshold and simply doing the highest scoring merge in all cases.

9 Reference algorithm

Here we describe a post-competition version of EDSM. Compared to the programs that were used during the competition, this algorithm produces a slightly better distribution of generalization rates on random problems (see section 11).

9.1 Definition of a merge's score

We award one point for each state label which, as a result of a merge, undergoes an identity check and turns out to be okay. Any mismatch results in a negative overall score. Details appear in section 9.4.2.

9.2 Initial hypothesis

The initial hypothesis is the prefix tree acceptor which directly embodies the training set.

9.3 Outer loop

The key insight of EDSM is that bad merges (which can't be directly detected when the training data is very sparse) can often be avoided if we instead do high scoring merges that have passed many tests and hence are likely to be correct. To have the best chance of finding a high-scoring merge to perform at any given moment, we need the largest possible pool of candidate merges. Thus, we would like to consider the possibility of merging every pair of hypothesis nodes, as in the following outer loop:

1. For every pair of nodes in the hypothesis, compute the score for merging that pair.
2. If any merge is valid, perform the highest scoring one, otherwise halt.
3. Go to step 1.

Note that this outer loop requires us to be able to merge nodes that are the roots of arbitrary subgraphs of the hypothesis, not just nodes that are the roots of trees. In the next section we show how to do this.

```
(define (compute-classes hypo       ; current hypothesis DFA (not modified)
                         ufer       ; union-find data structure (modified)
                         input-set) ; list of nodes asserted to be equivalent
  (when (> (length input-set) 1)
    (let ((learned-something-new? #f)
          (guy1     (car input-set)))
      (dolist (guy2 (cdr input-set))
        (when (not (uf-same-class? ufer guy1 guy2))
          (uf-unify-classes ufer guy1 guy2)
          (set! learned-something-new? #t)))
      (when learned-something-new?
        (dotimes (i alphabet-size)
          (compute-classes hypo ufer
           (delete-duplicates-and-undefineds
            (map (lambda (node) (get-child hypo node i))
                 (uf-get-members-of-guys-class ufer guy1)))))))))
```

Figure 1: Scheme code for working out which states are combined by a given merge.

9.4 Merging and scoring

To merge a pair of nodes, we must work out the partition of hypothesis nodes into equivalence classes which is implied by the assertion that the two candidate nodes are equivalent, plus the determinization rule (figure 5) which states that the children of equivalent nodes must be equivalent. Note that we can perform this computation regardless of whether the merge is valid, since validity depends on state labeling, whereas the equivalence classes only depend on the transition function.

Once we have determined the set of equivalence classes, it is trivial to consult the labels and compute a merge score, and, if the merge is in fact valid, to construct a new hypothesis reflecting the merge.

9.4.1 Computing equivalence classes

Figure 1 shows the Scheme language subroutine `compute-classes`, which works out the equivalence classes implied by a merge and the determinization rule. It employs a union-find data structure to keep track of sets of states that are known to be equivalent.

To assert that a particular set of states is equivalent, we call `compute-classes` on that set. The procedure checks the union-find data structure to determine whether the assertion is new information. If not, the routine returns immediately. Otherwise, it unifies the equivalence classes associated with all the members of the input set, and then calls itself recursively on each of the sets of i^{th} children of members of the newly unified equivalence class.

When considering a merge, we initiate the computation by calling `compute-classes` on the set consisting of the two nodes that we are thinking of merging. We also pass in a fresh union-find data structure that has been initialized with a singleton set for each state in the pre-merge hypothesis.

Note that the computation terminates because recursive calls only occur when separate classes have actually been unified, and the number of states in the hypothesis is an upper bound on the number of times this can happen.

Figure 4 shows an execution trace of `compute-classes` on a toy example.

9.4.2 Scoring

A merge's score is the sum over equivalence classes of the following quantity: if there are conflicting labels in the class, minus infinity; if there are no labels in the class, zero; otherwise, the number of labels minus one. We subtract one because the first label in the class establishes the correct label for the class, but is not checked.

9.4.3 Constructing a merged hypothesis

Once a candidate merge has been shown valid by a non-negative score, and we have decided to actually perform the merge, we can construct an updated hypothesis from the equivalence classes as follows. The new hypothesis has one state per equivalence class.

Let C_1 be an equivalence class, and i be an input symbol. Let s_1 be any state in C_1 that has a defined transition for i. Let s_2 be the target of that transition, and let C_2 be the class of s_2. Then i takes us from C_1 to C_2. If no state in C_1 has a defined transition for i, then C_1's transition for i is undefined.

Let s_3 be any state in C_1 that has a defined label. The label of s_3 becomes the label for C_1. If no state in C_1 has a defined label, then C_1's label is undefined.

10 Blue-fringe algorithm

Because the algorithm of section 9 performs merges in arbitary order, both nodes in a merge can be the roots of arbitrary subgraphs of the hypothesis. It turns out that by placing a restriction on merge order (described below), one can guarantee that one of the two candidate nodes is always the root of a tree, resulting in a particularly fast and simple program.

Much previous work has employed a restriction of this type, including the papers of Lang (1992); Oncina and Garcia (1992); Higuera *et al.* (1996); and the Abbadingo competition programs of Price and Juillé. Note that the restriction shrinks the pool of merge candidates, so it increases the failure rate of the algorithm as compared to the unrestricted algorithm of section 9. However, the idea is well worth describing.

As usual, we start with the prefix tree acceptor. The root is colored red. Its children are blue, and all other nodes are white. We maintain the following invariants:

- There is an arbitrary connected graph of mutually unmergeable red nodes.
- All non-red children of red nodes are blue.
- Blue nodes are the roots of isolated trees.

We restrict ourselves to the following actions:

- Compute the score for merging a red/blue pair.
- Promote a blue node to red if it is unmergeable with any red node.
- Merge a blue node with a red node.[6]

This basic framework of invariants and actions can be turned into different algorithms of widely varying performance, depending on the details of the policy for choosing which action to perform when. A particularly good policy is described in Juillé and Pollack (1998):

1. Evaluate all red/blue merges.
2. If there exists a blue node that cannot be merged with any red node, promote the shallowest such blue node to red, then goto step 1.
3. Otherwise (if no blue node is promoteable), perform the highest scoring red/blue merge that we know about, then goto step 1.
4. Halt.

Note that the algorithm of Higuera *et al.* (1996) has the priority of steps 2 and 3 reversed, which drastically reduces its effectiveness.[7] It is important to not start merging until many merge candidates have accumulated, so that one with a high score is likely to be available.

[6] Note that the last two actions might also require some white nodes to be recolored blue.

[7] On a set of 100 problems like the ones in section 11 but with 2500 training strings, the median generalization error rate for Juillé's policy is .004. Reversing the priority of steps 2 and 3 increases this to .39, which is nearly as bad as the value of .44 for the plain Trakhtenbrot-Barzdin algorithm.

```
(define (merge-and-compute-score red-cand blue-cand)
  (make-blue-guys-father-point-to-red-guy red-cand blue-cand)
  (set! score 0) ; using global variable for simplicity here
  (merging-walk-it red-cand blue-cand)
  score)

(define (merging-walk-it r b)
  (let ((r-label (get-label r))(b-label (get-label b)))
    (when (defined? b-label)
      (if (defined? r-label)
          (if (= r-label b-label)             ; compare labels
              (set! score (+ score 1))
              (set! score -infinity))
          (set-label! r b-label))))           ; copy in missing label
  (dotimes (i alphabet-size)
    (let ((r-child (get-child r i))(b-child (get-child b i)))
      (when (defined? b-child)
        (if (defined? r-child)
            (merging-walk-it r-child b-child)
            (set-child! r i b-child))))))     ; splice in missing branch
```

Figure 2: Code for performing and scoring a merge in the blue-fringe framework.

10.1 Merging in the blue-fringe framework

Thanks to the guarantee that every blue node is the root of an isolated tree, merging (and hence score keeping) is very simple in the blue-fringe framework. Scheme code for doing this is shown in figure 2. Since this procedure modifies the hypothesis, when using it to merely compute a score one would provide a mechanism for reverting the hypothesis to its unmodified state after each call.

10.2 The number of merges performed by blue-fringe algorithms

All blue-fringe state merging algorithms do at most $n(a-1)+1$ merges when constructing an n-state hypothesis over an alphabet of size a, with equality occuring when the final hypothesis contains no undefined edges. To see this, let r and b be the number of red and blue nodes at any point in the computation, let m be the number of merges so far, and let x be the number of undefined children of red nodes. Then at all times, the algorithm maintains the invariant $r(a-1) = m + x + b - 1$. We begin with $r = 0, m = 0, b = 1, x = 0$. Every promotion of a blue node with j defined children causes $r \leftarrow r+1, x \leftarrow x+a-j, b \leftarrow b-1+j$. Every merge that causes k new children to be spliced into red nodes causes $m \leftarrow m+1, x \leftarrow x-k, b \leftarrow b-1+k$. The program terminates when $b = 0$, so we have $r(a-1) - x + 1 = m$, which gives the result.

10.3 Two incorrect methods for computing scores

Here we warn against two plausible sounding but incorrect methods for evaluating merges in the blue-fringe framework. The mistake in both cases is a failure to account for the graph structure of the red part of the hypothesis.

The first incorrect method compares auxiliary trees constructed from the set of training set strings which pass through the two candidate nodes. This method ignores labels that are reachable from the red candidate node but which come from strings that go around that node rather than through it.

The second incorrect method compares labels during a non side-effecting simultaneous walk of the hypothesis starting at the red and blue merge candidates. This method can see all of the labels on red nodes, but it doesn't account for the fact that pairs of labels in the blue-rooted tree can end up conflicting with each other because the merge forces the tree to conform to the shape of the red graph.

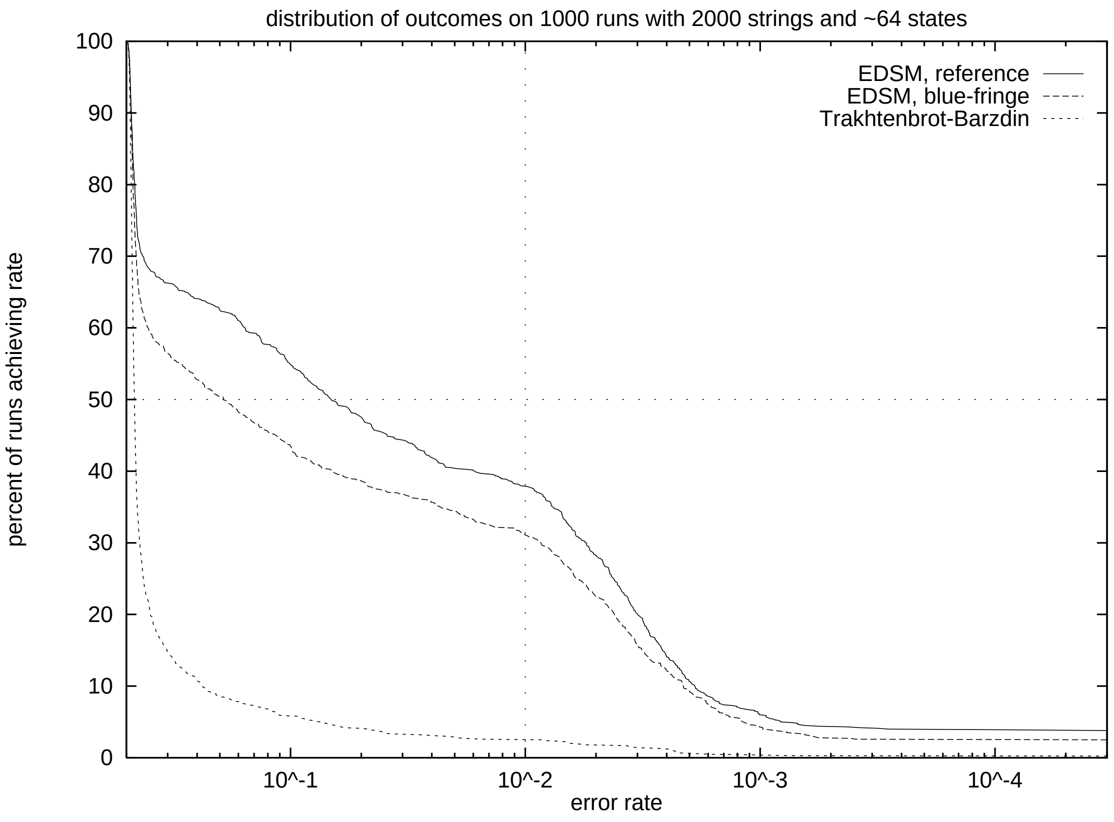

Figure 3: The complete distributions of results of three algorithms on a set of 1000 random problems. Each problem had 2000 training strings of length 0-15, and a depth-10 target DFA with about 64 states.

11 A comparison of two EDSM implementations

We have described two implementations of EDSM. A table (like table 3) could be made for either one showing that it scales well[8], and that it can usually solve problems at density level II but not density level I. In this section we put aside the question of scaling and focus on the question of how well the two versions can generalize on problems that lie halfway between columns II and I, that is, near the edge of typical solvability for the EDSM method.

Figure 3 and table 4 show the results of a comparison on a set of 1000 such problems. Clearly, both implementations of EDSM are much more powerful than the plain Trakhtenbrot-Barzdin program. The reference program is slightly more effective than the blue-fringe program. We attribute this to its larger pool of candidate merges.

We also mention that there is a strong stochastic component to the behavior of both EDSM programs[9], and that there are many problem instances where the reference program fails and the blue-fringe program succeeds. Given the somewhat uncorrelated failures of the two programs, it is natural to combine them by running both and then choosing the smaller of the two resulting DFA's. Table 4 shows that this combined approach works better than either program alone. In fact, the combined performance level is well into the range reported by Juillé and Pollack (1998) for the search-intensive SAGE system.

[8]The reference algorithm needs some speedups to be practical. See the appendix.

[9]This is due to randomness in the training data and the fact that even high scoring merges can be wrong.

algorithm	median generalization rate	number of solutions (out of 1000)
Trakhtenbrot-Barzdin	.537	26
blue-fringe EDSM (Juillé)	.809	311
reference EDSM	.934	379
combination of the previous two	.955	423

Table 4: A comparison of two implementations of EDSM on 1000 random problems. Each problem had 2000 training strings of length 0-15, and a depth-10 target DFA with about 64 states. Solutions are hypotheses with a generalization rate of .99 or better.

12 Notes on run time

The run time of Trakhtenbrot-Barzdin is upper bounded by PH^2, where P is the size of the inital PTA, and H is the number of nodes in the final hypothesis. The bound for the blue-fringe algorithm is PH^3. We don't have a tight upper bound on the run time of the reference algorithm, but we conjecture that it would be closer to P^3H than to P^4H.

13 Conclusion of Part II

We have described two versions of a polynomial time DFA learning algorithm that works very well on randomly generated problems. While the algorithm can be defeated by a malicious adversary, we believe that it will degrade gracefully as one moves gradually away from the average case. We recommend that anyone faced with a DFA learning task give this algorithm a try.

Acknowledgements

We thank Hugues Juillé for sending us code and an early draft of Juillé and Pollack (1998), which is the source of the blue-fringe control policy described in section 10.

Appendix: speedups for the reference algorithm

For the experiment of section 11, we sped up the reference algorithm by only considering merges between nodes that lie within a distance `w` of the root on a list of nodes created by a breadth-first traversal of the hypothesis. This change hurts performance by causing the algorithm to miss the (relatively rare) high scoring merges involving deep nodes. Note that while the existence of the new `w` parameter appears to make the algorithm less general by requiring prior knowledge of the size of the target DFA, one can use the standard doubling trick to eliminate this requirement. However, in our section 11 experiment on size-64 DFA's, we simply used a `w` value of 256.

We also employed the following optimizations, which don't change the behavior of algorithm except to make it faster. Whenever the deeper of a pair of candidate nodes is the root of an isolated tree, the blue fringe scoring routine of figure 2 is used to cheaply compute the same score that would be returned by the expensive general-purpose code of section 9.4. Also, before finally resorting to the general-purpose code, we first do a quick walk looking for labeling conflicts; if one is found, we can immediately return a score of minus infinity.

References

B. Trakhtenbrot and Ya. Barzdin'. (1973) *Finite Automata: Behavior and Synthesis.* North-Holland Publishing Company, Amsterdam.

The two loops: $0 \to 1 \to 2 \to 3 \to 0$
$4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 4$

classes before call	argument to compute-classes
{0} {1} {2} {3} {4} {5} {6} {7} {8} {9}	(0 4)
{04} {1} {2} {3} {5} {6} {7} {8} {9}	(1 5)
{04} {15} {2} {3} {6} {7} {8} {9}	(2 6)
{04} {15} {26} {3} {7} {8} {9}	(3 7)
{04} {15} {26} {37} {8} {9}	(0 8 4)
{048} {15} {26} {37} {9}	(1 9 5)
{048} {159} {26} {37}	(2 4 6 8)
{02468} {159} {37}	(3 5 7 9)
{02468} {13579}	(0 6 8 4 2)

Figure 4: Execution trace of `compute-classes` while merging two cycles of length 4 and 6 to create a cycle of length 2.

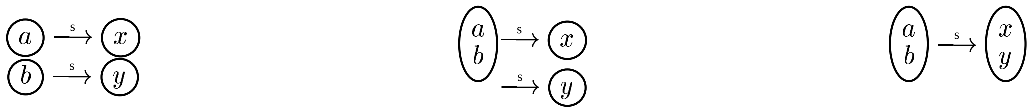

Figure 5: Explanation of determinization. **Left:** prior to the merge, the symbol s leads from state a to state x, and also from state b to state y. **Center:** after states a and b are merged, the automaton is non-deterministic. **Right:** to restore determinism, states x and y must be merged, which can lead in turn to the merging of other states.

D. Angluin. (1978) *On the Complexity of Minimum Inference of Regular Sets*. Information and Control, Vol. 39, pp. 337-350.

L. Veelenturf. (1978) *Inference of Sequential Machines from Sample Computations*. IEEE Transactions on Computers, Vol. 27, pp. 167-170.

M. Kearns and L. Valiant. (1989) *Cryptographic Limitations on Learning Boolean Formulae and Finite Automata*. STOC-89.

L. Pitt and M. Warmuth. (1989) *The Minimum DFA Consistency Problem Cannot be Approximated Within any Polynomial*. STOC-89.

Kevin J. Lang. Random DFA's can be Approximately Learned from Sparse Uniform Examples. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pp 45-52, July 1992.

J. Oncina and P. Garcia. Inferring Regular Languages in Polynomial Updated Time. In *Pattern Recognition and Image Analysis*. pp. 49-61, World Scientific, 1992.

Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert Schapire, and Linda Sellie. *Efficient Learning of Typical Finite Automata from Random Walks*, STOC-93, pp. 315-324.

P. Dupont, L. Miclet, and E. Vidal. What is the search space of the regular inference? In *Proceedings of the International Colloquium on Grammatical Inference ICGA-94*, Lecture Notes in Artificial Intelligence 862, pp. 25-37, Springer-Verlag, 1994.

C. de la Higuera, J. Oncina, and E. Vidal. Identification of DFA: Data-Dependent Versus Data-Independent Algorithms. In *Proceedings of the International Colloquium on Grammatical Inference ICGA-96* Lecture Notes in Artificial Intelligence 1147, pp. 313-325, Springer-Verlag, 1996.

Joe Kilian and Kevin J. Lang. (1997) A Scheme for Secure Pass-Fail Tests. NECI Technical Note 97-016N.

Hugues Juillé and Jordan B. Pollack. (1998) SAGE: a Sampling-based Heuristic for Tree Search. Submitted to *Machine Learning*.