# Random DFA's can be Approximately Learned from Sparse Uniform Examples*

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

October 1, 1998

**Abstract**

Approximate inference of finite state machines from sparse labeled examples has been proved NP-hard when an adversary chooses the target machine and the training set [Ang78, KV89, PW89]. We have, however, empirically found that DFA's are approximately learnable from sparse data when the target machine and training set are selected at random.

## 1 A Greedy Learning Algorithm

Trakhtenbrot and Barzdin described the following polynomial time algorithm for constructing the smallest DFA consistent with a complete labeled training set [TB73]. The input to the algorithm is the prefix-tree acceptor which directly embodies the training set. This tree is collapsed into a smaller graph by merging all pairs of states that represent compatible mappings from string suffixes to labels. The algorithm contains two nested loops. In the outer loop, each node $i$ is visited in breadth-first order starting at the tree's root. In the inner loop, each node $j$ between the root and node $i-1$ is evaluated for compatibility with node $i$ by comparing the labels in the subtrees rooted at $i$ and $j$. If every corresponding label is the same, the transition from $i$'s parent to $i$ is altered to point at node $j$ instead. The node $i$ and its descendents are then inaccessible

*This downloadable postscript document has been reformatted from the COLT-92 original (which was put together with scissors and paste). It is still owned by the ACM, whose copyright notice follows: Copyright 1992 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

and can be discarded. An upper bound on the running time of this algorithm is $m \cdot n^2$, where $m$ is the size of the initial tree and $n$ is the size of the final graph.

This contraction procedure for complete labeled trees can be generalized as follows to produce a (not necessarily minimum) machine consistent with a sparsely labeled tree. In a sparsely labeled tree, the absence of a labeling conflict between the subtrees rooted at a pair of nodes $i$ and $j$ does not guarantee that the two nodes correspond to the same state in the smallest consistent machine. However, one can be greedy and merge the nodes anyway. To maintain consistency with the training set, state labels from the subtree rooted at $i$ must be copied into the subgraph rooted at $j$ before the subtree at $i$ is discarded.

Implementing this label copying process correctly requires careful attention to details (see the appendix), but the conceptually important thing is that the resulting merger of different parts of the training set increases its effective density and constrains succeeding choices of which states to merge. This can be good or bad depending on whether the algorithm's greedy initial state merging choices are correct. If they are not, the resulting merger of unrelated sets of labels can cause the training set to look random and lead to an explosion in the size of the hypothesis. Conversely, if the initial choices are correct there can be a snowballing of constraints leading to a highly accurate hypothesis. Both of these effects are visible in the experimental results of the next section. Because the algorithm's initial choices are so important they should be based on as much evidence as possible. This is why both loops of our implementation run through the nodes in breadth-first order starting at the root.

## 2 Approximate Learning of Random DFA's over the Uniform Distribution

Here we describe a learning experiment in which randomly generated DFA's were approximately identified from positive and negative examples drawn from a uniform distribution. Each of 1000 learning trials went as follows. We generated a target machine of nominal size 512 by first selecting random destinations for the 1280 transitions of a 640-state DFA and then discarding all states that were not reachable from the starting state. As discussed in the next section, we rejected the machine if its depth wasn't 16. Otherwise, we randomly declared each of its states to be an acceptor with probability 1/2. We then selected a training set size ranging from 200 to 200000 and randomly drew that many strings without replacement from the set of 4194303 binary strings not longer than 21 bits. We labeled these strings according to the target machine and fed the resulting training set into the learning algorithm, yielding a hypothesis whose generalization was measured on the remaining binary strings of length not greater than 21.

Scatter plots of the size and generalization of the hypotheses appear in figure 1. Evidently, the performance of the learning algorthm went through three distinct stages as the density of the training set was increased. When the density

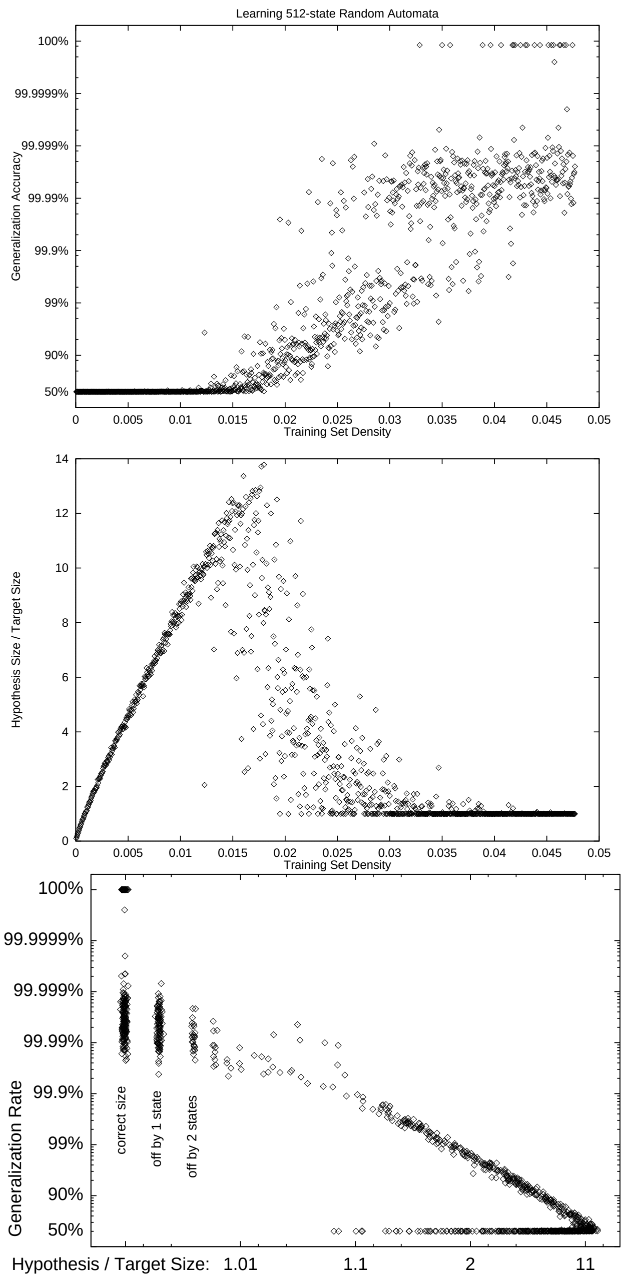

Figure 1: These three scatter plots show the tradeoffs between generalization, hypothesis size, and training set density that were measured during a learning experiment on randomly generated 512-state target machines.

was very low, the algorithm totally failed to detect the structure of the training set; each hypothesis was as large as it would have been if the training strings had been labeled randomly, while the predictions of the hypotheses were no better than random guesses. In this stage, additional training data increased the size of the hypotheses but did not lead to better generalization.

At a training set density of 1.5 percent the algorithm entered a second performance stage in which additional training data led to smaller hypotheses with better generalization. The gradual improvement of this stage ended with hypotheses that were were still 10 percent too large but were able to classify new strings with an accuracy of about 99.9 percent.

The third stage began when the training set density reached about 3 percent. The performance in this stage was an order of magnitude better than in the previous one: the hypotheses were essentially the correct size, and their generalization rates ranged from 99.99 to 99.999 percent. In this stage further increases in the density of the training set had little effect on performance, so the optimal amount of training data was the 3 percent required make the jump from the second to the third stage. In the next section we shall see that this threshold density scales favorably with increasing problem size.

## 2.1 How Performance Scales with Target Size

The tree contraction algorithm of Trakhtenbrot and Barzdin will identify any finite state machine from the labels on the complete set of all strings not longer than $d + 1 + \rho$, where $d$ is the depth of the machine and $\rho$ its degree of distinguishability.[1] In the worst case $d = \rho = n - 1$, but Trakhtenbrot and Barzdin proved that in the average case $\rho = \log_a \log_2 n$ and $d = C \log_a n$, where $C$ is a constant that depends on $a$. Thus an average DFA can be exactly identified from a uniform complete sample of size $a^2 n^C \log_2 n - 1$.

Because the constant $C$ is loose for small $a$, we obtained a tighter bound on $d$ experimentally: roughly three fourths of a sample of several thousand $n$-state degree-2 DFA's had depth not greater than $2log_2n - 2$. We also found that $\rho$ very rarely exceeded 4 for machines containing up to 10000 states. For experimental clarity we restricted our attention to $n$-state target machines of exactly $2log_2n - 2$ depth.[2] We also pretended that $\rho$ was exactly 4 so that the size of a uniform complete sample would always be $2 * 2^{2log_2n-2+1+4} - 1 = 16n^2 - 1$.

For each $n$ in the set $\{32, 64, 128, 256, 512, 1024\}$ we performed 2000 learning trials in which the greedy state-merging algorithm was applied to a different sized subset of a uniform complete sample that had been labeled by a different randomly generated $n$-state target machine.[3] Each hypothesis was evaluated

[1]The degree of distinguishability of a DFA is the smallest integer $i$ such that for every pair of non-equivalent states in the machine there exists a suffix not longer than $i$ that sends exactly one of the two states to an accepting state.

[2]Other depths yield similar but shifted learning curves.

[3]The transition diagram of a target machine of nominal size $n$ was obtained by first constructing a random degree-2 digraph on $\frac{5}{4}n$ nodes and then extracting the subgraph reachable from an arbitrarily selected root node. The graph was discarded if its depth wasn't $2log_2n - 2$. The average size of our target machines were 31.9, 63.9, 127.4, 254.8, 510.7, and 1020.5 states.

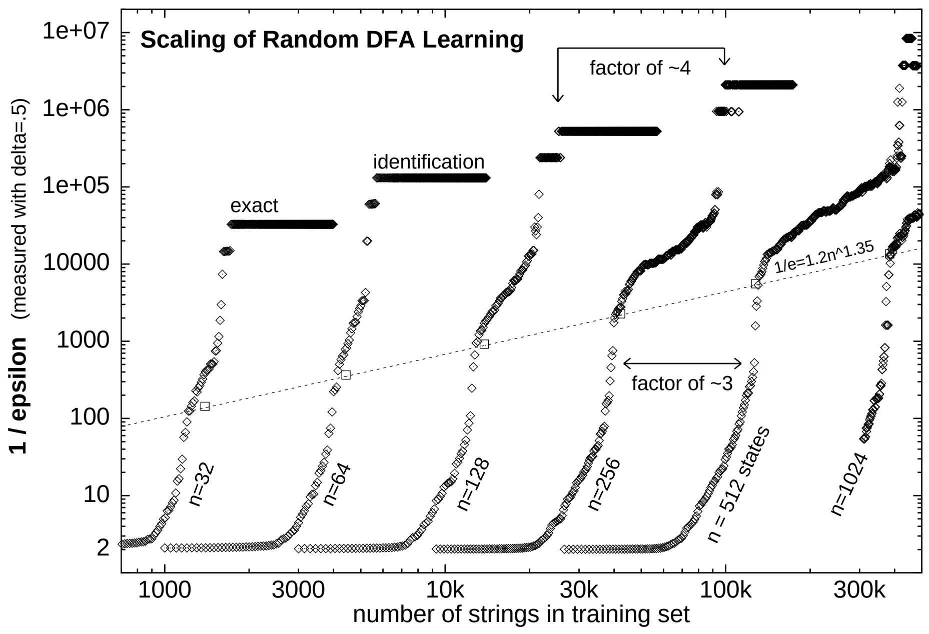

Figure 2: Results of scaling experiment

by measuring its generalization to the unseen portion of the labeled uniform complete sample. Each dot in figure 2 represents the median generalization in a window of width 200 that was passed over the set of 2000 trials for a given value of $n$.[4] Thus, the curves in the figure show the tradeoff between training set size and generalization that is achievable with a confidence of 1/2.

The key feature of each curve is the S-shaped piece in the middle which corresponds to the stage-3 generalization cloud in the scatter plot of figure 1. The left and right boundaries of this piece represent transitions from low to high generalization and from high generalization to exact identification. We are interested in how these two transition points scale with increasing problem size.

Roughly speaking, when the size of the target machine was doubled the amount of training data required for good generalization increased by a factor of 3. Since the size of a uniform complete set of strings increased by a factor of 4, we see that as the problem was scaled up the fraction of a complete training set that was required for good generalization decreased. The calculations of section 4 suggest that if the problem could be made arbitrarily large the required training set density would approach zero. This scaling behavior for approximate learning of random DFA's may be contrasted with [Ang78] where it is proved that in the worst case, exact identification of a DFA requires a training set containing all but a vanishingly small subset of a uniform complete set of strings.

Our experiments suggest that exact identification from sparse data is not

[4]There were only 750 trials for machines of size 1024.

possible even in the average case. Each time we doubled the size of our random target machines the number of training examples required for exact identification with confidence 0.5 increased by at least a factor of 4 (see figure 2), so exact identification of random DFA's using our learning algorithm requires at least a fixed fraction of a complete set of strings.

## 3 Trellis Machines

Let us now consider the class of fixed-width layered feedforward finite state machines, or "trellis" machines for short. We are interested in the task of identifying randomly generated depth-$d$ trellises from labeled subsets of the strings of length $d$. We originally expected the absence of short training strings to cause this inference problem to be harder than the one described in section 2, but it turned out to be much easier after we constrained our greedy learning algorithm to produce layered hypothesis machines. Roughly speaking, when this version of the algorithm combines a collection of raw nodes into a hypothesis state, the merging of the sets of suffixes that had been associated with the nodes causes an increase in the effective density of the training set when the next layer of the tree is processed. Successive layers of the target machine are therefore progressively easier to identify, and the sample complexity of the problem is dominated by the need to have a few common suffixes for each pair of nodes in the earliest layers of the tree. This begins to occur when the size of the training set is on the order of the square root of $a^d$, the number of strings of length $d$. The requirement for training data actually grows faster than this because the suffix labeling functions associated with the early nodes of a trellis machine become more correlated with increasing $d$. The informal calculations described below suggest that the number of training examples needed to exactly identify a randomly generated trellis with a given level of confidence is asymptotically $k \cdot (aw/(w-1))^{d/2}$, where $w$ is the width of the target machine, $a$ is the size of the input alphabet, and $k$ is a constant that depends on the details of the target machine's architecture but not on $d$. While exponentially large in the size of the target machine, this is a vanishingly small subset of the complete set of strings of length $d$.

### 3.1 A Simple Performance Model

We shall now give a sketch of a calculation for estimating the probability of exactly identifying a randomly generated trellis from a uniform sparse training set of size $x$ using a version of the greedy state merging algorithm that only merges states which lie in the same layer. In this abstract we shall describe the calculation without justifying its assumptions; we believe that the calculation is essentially correct because its predictions are consistent with our experimental results. We begin by defining the function

$$\text{Suffixes}(i,x) = \text{merge}\left(\frac{a^i}{a \cdot w(i-1)}\ ,\ \frac{x}{a^i}\ ,\ a^{d-i}\right)$$

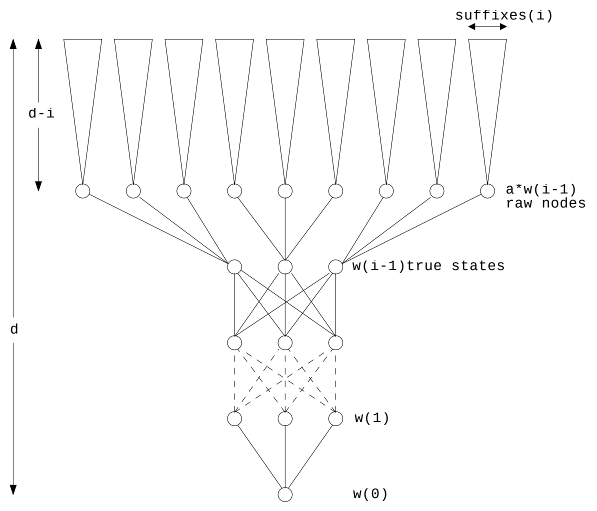

Figure 3: The computational state before processing the i'th layer while learning a trellis grammar.

which gives the expected number of training string suffixes associated with a node in layer $i$ of the hypothesis, assuming that the learning algorithm has already combined the nodes in every previous layer into the target machine's true states (see figure 3). The function $\text{merge}(n, s, t) = t - t\left(1 - \frac{s}{t}\right)^n$ is the expected size of a set formed by merging $n$ independent size-$s$ subsets of a set of size $t$. $w(i)$ denotes the number of states in the $i$-th layer of the target machine.

$$\text{Overlap}(i, x) = \frac{\text{Suffixes}(i, x)^2}{a^{d-i}}$$

is the expected number of common elements in the two sets of suffixes associated with a pair of nodes in layer $i$, assuming that the nodes in all previous layers of the hypothesis have been correctly merged.

$$\text{P}_{\mathit{diff}}(i) = \prod_{j=i+1}^{d} \frac{w(j) - 1}{w(j)}$$

is the probability of a random suffix of length $d - i$ mapping two different states in layer $i$ of the target machine to different output labels. This will happen when the two paths through the machine defined by the states and the suffix do not meet in any layer from $i + 1$ to $d$. Then, $\text{P}_{pairwise}(i, x) =$

$$\max\left(1 - (1 - \text{P}_{\mathit{diff}}(i))^{\text{Overlap}(i,x)}, \frac{\text{Overlap}(i, x)}{a^{d-i}}\right)$$

is the probability of not erroneously merging a pair of layer $i$ nodes that actually correspond to different target machine states, conditioned on the correct processing of all previous layers. The first argument to the max function is a model

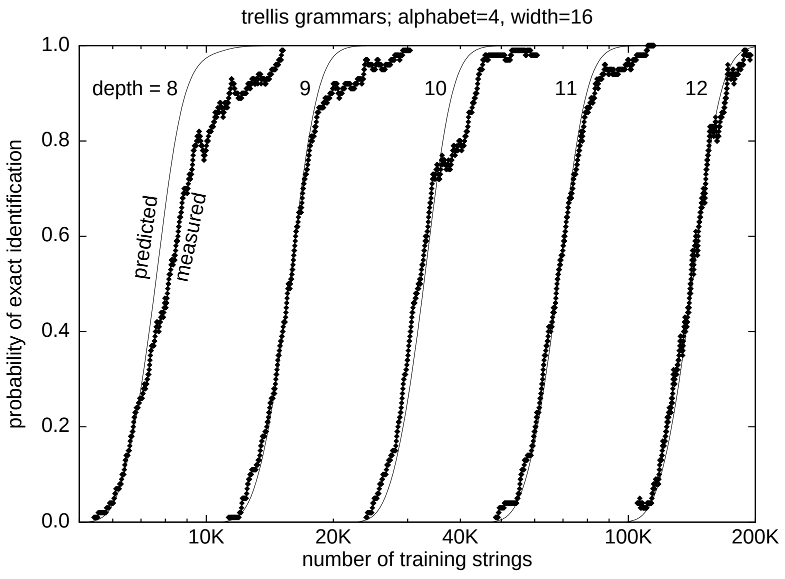

Figure 4: Does the trellis model match reality?

of the discrimination process as a sequence of Overlap$(i, x)$ independent trials, each with a $\mathrm{P}_{diff}(i)$ chance of distinguishing between the two nodes. This model breaks down in the final layers of the machine because the number of possible suffixes becomes small. We note that any pair of non-equivalent nodes must assign different labels to at least one string; the second argument to the max function is the probability of finding this string among the common suffixes.

$$\mathrm{P}_{layer}(i,x) = \mathrm{P}_{pairwise}(i,x)^{\frac{1}{2}(w(i)-1)\cdot aw(i-1)}$$

Assuming that we have already identified layers 1 through $i$ of the target machine, $\mathrm{P}_{layer}(i,x)$ is the probability of making no mistakes while combining the $a \cdot w(i-1)$ raw nodes in layer $i$ of the hypothesis into the set of $w(i)$ true states. The probability of identifying the whole machine is then

$$\mathrm{P}_{ident}(x) = \prod_{i=1}^{d} \mathrm{P}_{layer}(i,x).$$

We have compared the predictions of this model with statistics collected from 5,000 learning trials in which sparse training sets of various sizes were used to identify trellis machines with $a = 4$, $w = 16$, and with $d$ ranging from 8 to 12. The results of this comparison can be seen in figure 4. A similarly good match between theory and experiment was observed for target machines with $a = w = 5$, and $d$ ranging from 5 to 9.

### 3.2 Sample Complexity of Random Trellises

We can now derive an approximate formula for the average sample complexity of trellis machines with respect to our learning algorithm by inverting the function $\mathrm{P}_{ident}(x)$ to obtain the function Examples($\delta$). Recall that $\mathrm{P}_{ident}(x) = \prod_{i=1}^{d} \mathrm{P}_{layer}(i, x)$. Because successive layers in the fixed-width portion of a trellis machine are progressively easier to identify, for a given trellis architecture we can choose a fixed $k$ such that for large $d$ and $x_0 = \text{Examples}(\frac{1}{2})$, for every $i$ such that $k < i \leq d$, the term $\mathrm{P}_{layer}(i, x_0)$ in the above product is as close as we want to 1. Thus, an approximation to $\mathrm{P}_{ident}$ in the vicinity of $x_0$ can be obtained by truncating the product to its first $k$ terms. We claim (but have not proved) that a reasonable approximation to the scaling behavior of Examples($\delta$) can be obtained by inverting this truncated product. This yields the bound

$$\text{Examples}(\delta) \;\leq\; \max_{1 \leq i \leq k} \text{Examples}(i, \delta^{\frac{1}{k}})$$

where Examples($i, \delta$) is the sample complexity of the $i$-th layer of the target machine. This function can be expressed in closed form. First, we rewrite the definition of $\mathrm{P}_{diff}(i)$ as $k_1(i)\left(\frac{w-1}{w}\right)^{d-k_2(i)-i}$ by collecting the $k_2(i)$ terms corresponding to the fixed set of layers above $i$ that are not of width $w$ into the constant $k_1(i)$. Then, assuming that $i$ is small enough relative to $d$ that our first estimate of $\mathrm{P}_{pairwise}$ is the relevant one, we have Examples($i, \delta$) =

$$a^d - a^d \left( 1 - \sqrt{\frac{a^{i-d} \cdot \log_2 \left( 1 - 2^{\left[\frac{2 \log_2 \delta}{a \cdot (w(i)-1) w(i-1)}\right]} \right)}{\log_2 \left( 1 - k_1(i) \left(\frac{w-1}{w}\right)^{d-k_2(i)-i} \right)}} \right)^{\frac{w(i-1)}{a^{i-1}}}$$

For small $x$, $\log(1+x) \approx x$, so for $d \gg w$ this equation reduces to

$$\text{Examples}(i, \delta) \approx k_3(i, \delta) \left( \frac{a \cdot w}{w - 1} \right)^{d/2}$$

where $k_3(i, \delta)$ is an expression that doesn't depend on $d$. We have checked the validity of this rough derivation by inverting the full equations for $\mathrm{P}_{ident}(x)$ numerically. For each of several values of $a$ and $w$ that we have tested, the scaling factor associated with incrementing $d$ appears to asymptotically approach $\sqrt{aw/(w-1)}$.

## 4 Locating the Threshold to Good Generalization for Random Targets

The transition from low to high generalization observed in our learning experiments on random DFA's (see figure 1) is easy to understand when the target machine's nodes have been organized into layers based on their distance from the root. This jump in generalization is associated with a similar transition in the relationship between training set density and the number of correctly identified layers (see figure 5). An elaboration of our trellis performance model shows that the effective density of a training set increases as one successively identifies a sequence of target machine layers that are not growing too rapidly in width and do not contain too many backwards connections. This beneficial effect does not occur in the expanding first half of a random DFA, but if the middle layer of the machine can be successfully identified then things will get so much easier for a while that several more layers are likely to identified as well. The final layers of the machine are then hard again because backwards connections drain away most of the training set. Fortunately, the same thing happens during testing, so these layers are not important for good generalization. Thus the sample complexity of approximately learning a random DFA is dominated by the number of examples required to identify the machine's middle layer. The tree-like structure of machine's first half makes it easy to estimate this data requirement. The first three equations here are essentially the same as in our analysis of trellis machines.

$$P_{idMid} = (P_{pair})^{\text{npairs}}$$

$$P_{pair} = 1 - (1 - P_{\text{diff}})^{\text{overlap}}$$

$$\text{overlap} = \text{suffixes}^2/\text{universe}$$

We approximate the number of pairwise state differentiations that must occur by the following rough argument. The depth of the middle layer is $\log_2 n - 1$, so it contains about $2^{\log_2 n - 1} = n/2$ candidate nodes (actually fewer), each of which must be compared with an average of half the roughly $n/2$ nodes (actually more) that will remain in the hypothesis after we have processed the middle layer. Thus the number of comparisons is about

$$\text{npairs} \approx n^2/8$$

Because the first half of the target machine is nearly a tree, very little state merging has occurred before the middle layer and so the number of suffixes associated with a middle-layer candidate node is just

$$\text{suffixes} \approx \frac{\text{strings}}{2^{\log_2 n - 1}} = \frac{2 \cdot \text{strings}}{n}$$

These suffix samples are subsets of a complete set of size

$$\text{universe} = 2 \cdot 2^{(2\log_2 n + 3 - (\log_2 n - 1))} - 1 \approx 32n$$

Combining these equations gives

$$P_{idMid} \approx \left[1 - (1 - P_{\text{diff}})^{(\frac{1}{8} \cdot \text{strings}^2/n^3)}\right]^{n^2/8}$$

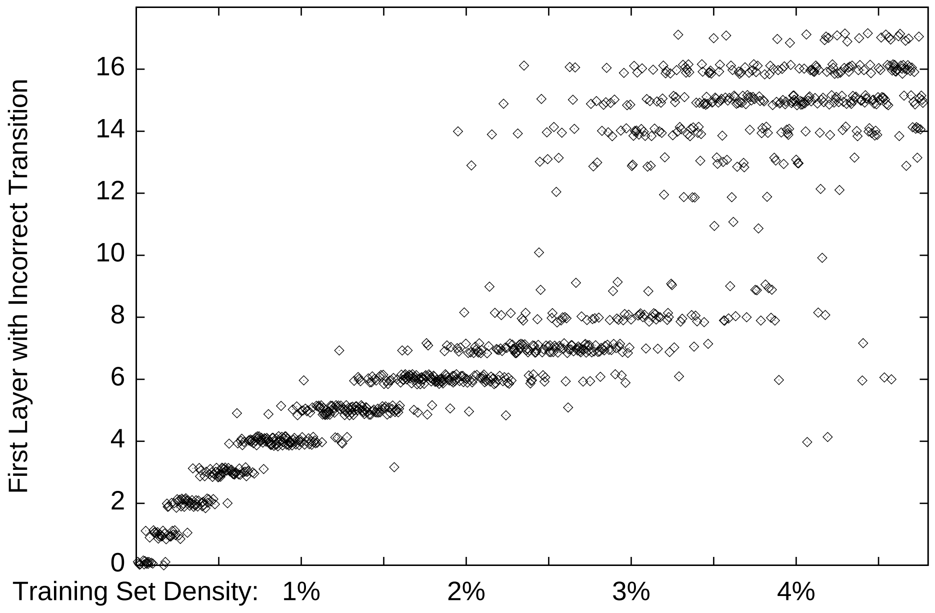

Figure 5: The number of correctly identified layers in 512-state target machines plotted as a function of training set density.

and

$$\text{strings} \approx \sqrt{\frac{8n^3 \log_2(1 - 2^{8\log_2 P_{idMid}/n^2})}{\log_2(1 - P_{\text{diff}})}}$$

Plugging in $P_{\text{diff}} = 0.5$ and $P_{\text{idMid}} = 0.5$ we see that the middle layer can be identified with confidence $1/2$ from a training set of size

$$\text{strings} \approx \sqrt{-8\log_2(1 - 2^{-8/n^2})} \cdot n^{3/2}$$

after which more layers will be probably be identified as well and we will end up with a hypothesis with good generalization. The transition densities predicted by this argument are plotted as small squares in figure 2.

## 5 Learning Randomly Labeled Strings

Pitt and Warmuth proved that in the worst case it is NP-hard to find a DFA that is consistent with a given set of strings and is within a polynomial factor of the size of the smallest such machine. To study the average-case difficulty of this problem we have applied the algorithm of section 1 to randomly labeled sets of strings with several degrees of sparseness. Figure 6 shows the size of the resulting DFA's, divided by a lower bound on the average size of the smallest consistent machine. The bound was obtained by a counting argument equating $2^x$, the number of ways of labeling a set of $x$ strings, with $n2^n n^{an}/n!$, an estimate of the number of different $n$-state DFA's. Evidently it is possible to get within a factor of 2 of this bound.

The dotted line in the figure shows the expected size of the smallest DFA consistent with a randomly labeled *complete* set of strings. Recall that Trakhtenbrot and Barzdin's algorithm merges a node with an earlier node whenever the subtrees rooted at the two nodes are labeled in the same way. Thus by computing the number of different subtree labelings that are expected to occur in each layer of the tree we can determine how many nodes will remain after every pair of compatible nodes has been merged. The calculation goes like this: if $\text{Ways}(i) = 2^{a^{d-i+1}-1}$ is the number of ways of labeling a subtree of depth $d-i$, and $\text{Draw}(t,n) = t - t\,(1-1/t)^n$ is the expected size of a set created by independently drawing $n$ random elements with replacement from a set of size $t$, then $\text{Uniq}(i) = \text{Draw}(a \cdot \text{Nodes}(i-1), \text{Ways}(i))$ is the expected number of different subtree labelings associated with the raw nodes of layer $i$, $\text{Prev}(i) = \text{Draw}(\sum_{j=0}^{i-1} \text{Nodes}(j), \text{Ways}(i))$ is the expected number of different subtree labelings associated with accepted nodes in previous layers of the tree, and $\text{Nodes}(i) = \text{Uniq}(i)\,(1 - \text{Prev}(i)/\text{Ways}(i))$ is the expected number of nodes remaining in layer $i$ after all nodes whose subtree labelings are compatible with those of previously accepted nodes have been eliminated by merging. The size of the contracted tree is then $\sum_{i=0}^{d} \text{Nodes}(i)$.

## 6 Summary

We have described average-case learning experiments in which target machines and training sets were both drawn randomly from uniform distributions. We found that a variant of Trakhtenbrot and Barzdin's tree contraction algorithm [5] can learn random $n$-state DFA's of depth $2\log_2 n - 2$ with confidence $1/2$ and generalization $1 - \frac{1}{1.2n^{1.35}}$ from a set of $\sqrt{-8\log_2(1-2^{-8/n^2})} \cdot n^{3/2}$ training strings drawn randomly from the uniform distribution over the set of $16n^2 - 1$ binary strings not longer than $2\log_2 n + 3$. A slightly different algorithm can exactly identify randomly generated fixed-width layered feedforward machines with high probability from training sets whose size asymptotically scales with target machine depth like $(aw/(w-1))^{d/2}$, where $w$ and $d$ are the width and depth of the target and $a$ is the size of the input alphabet. For both of these target classes, the required fraction of a complete set of training strings approaches zero as the size of the target machine increases.

[5] The algorithm can also be viewed as a variant of Veelenturf's method for constructing Moore machines from input/output string pairs.

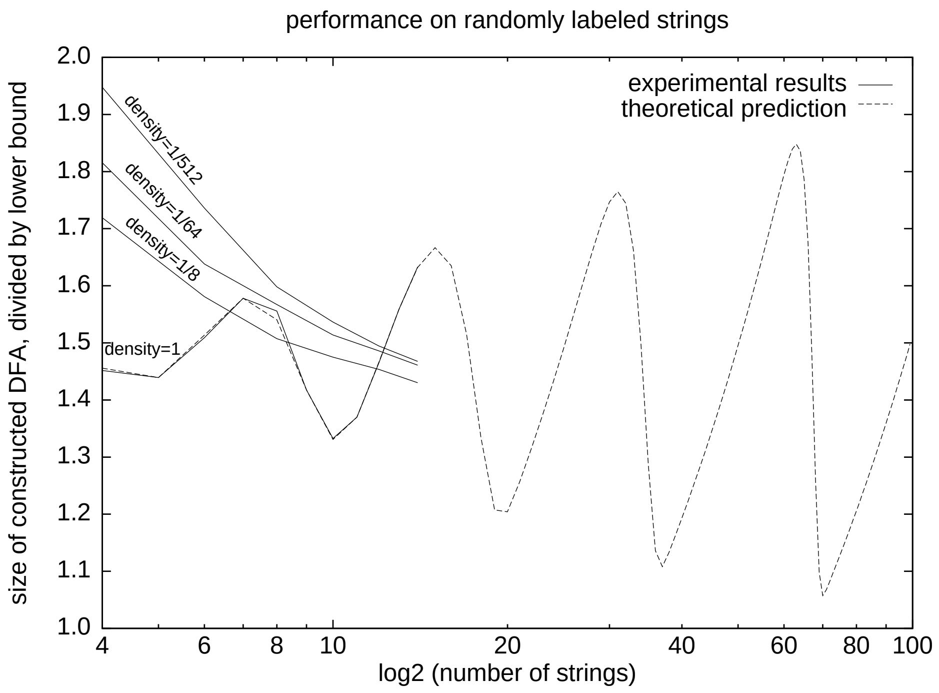

Figure 6: Performance on Randomly Labeled Strings

## Acknowledgements

The author thanks Les Valiant, Eric Baum, Satish Rao, Bruce Maggs, Lee Giles, Mark Goudreau, and Cliff Miller for helpful discussions.

## References

[Ang78] D. Angluin. (1978) *On the Complexity of Minimum Inference of Regular Sets.* Information and Control, Vol. 39, pp. 337-350.

[KV89] M. Kearns and L. Valiant. (1989) *Cryptographic Limitations on Learning Boolean Formulae and Finite Automata.* STOC-89.

[PM88] S. Park and K. Miller. (1988) *Random Number Generators: Good Ones are Hard to Find.* Communications of the ACM, Vol. 31, No. 10, pp. 1192-1201.

[PW89] L. Pitt, M. Warmuth. (1989) *The Minimum DFA Consistency Problem Cannot be Approximated Within any Polynomial.* STOC-89.

[TB73] B. Trakhtenbrot and Ya. Barzdin'. (1973) *Finite Automata: Behavior and Synthesis.* North-Holland Publishing Company, Amsterdam.

[V78] L. Veelenturf. (1978) *Inference of Sequential Machines from Sample Computations.* IEEE Transactions on Computers, Vol. 27, pp. 167-170.

## Appendix

Here is a Scheme implementation of the greedy learning algorithm of section 1. The tricky part is determining whether the subtree rooted at an outer loop node is mergeable with the subgraph rooted at an inner loop node. Because the subgraph can contain cycles and other non-tree features, the conformation of the subtree to the shape of the subgraph can induce labeling conflicts within the subtree. Thus one cannot pre-check the legality of a potential merge by a simple walk starting at the two nodes; our solution is to optimistically perform a destructive merger of the subtree and subgraph while keeping backup copies of modified nodes so that the work can be cheaply undone if a labeling conflict is detected. Cycles can also cause unexplored tree branches to be spliced into regions of the graph that have already been traversed by the program's outer loop. After a successful merge any such branches must be added to the outer loop's breadth-first search queue.

```
(define-structure dfa-state
  parent         ;
  children       ; a list, indexed by input alphabet.
  state-label    ; a symbol, or () if unknown.
  incoming-char  ; how to get from parent to self.
  )
```

```
(define (perform-greedy-tree-contraction root)

  (let ((search-queue (make-queue))
        (unique-nodes '()))

    (add-item-to-queue search-queue root)

    ; Outer loop visits states in breadth-first order.

    (iterate breadth-first-search-step ()
      (let ((curnode (get-next-item-in-queue search-queue)))

        ; Inner loop tries to merge a state with previously accepted ones.

        (iterate find-node-to-merge-with ((merge-candidates unique-nodes))

          (if (null? merge-candidates)

              ; If every attempted merge fails, the state is unique, so add
              (block ;its children to the search queue and exit inner loop.
               (set unique-nodes (append! unique-nodes (list curnode)))
               (dolist (x (children curnode))
                 (when x (add-item-to-queue search-queue x))))

              ; Otherwise, try merging the state with the next candidate.
              (let ((candidate (car merge-candidates)))
                (initialize-backup-table) ; Note: this table and this
                (set list-of-splices '()) ; list are global resources.

                (if (eq? 'succeeded (attempt-to-merge-states curnode
                                                             candidate))
                    ; Visit new children of previously accepted nodes.
                    (dolist (x list-of-splices)
                      (when (memq (car x) unique-nodes)
                        (add-item-to-queue search-queue (cdr x))))

                    (block ; Else undo side effects of failed merge and
                     (revert-backed-up-nodes) ; proceed with the search.
                     (find-node-to-merge-with (cdr merge-candidates)))
                    ))))

        ; End of inner loop.  This point is reached when the current state
        ; has been merged with an earlier state or has been found unique.

        (when (not (queue-empty? search-queue))
          (breadth-first-search-step))))

    ; End of outer loop

    root))
```

```
(define (attempt-to-merge-states curnode candidate)
  ; The transition from the parent of the current state
  ; is altered to point to the merge candidate instead.
  (backup (parent curnode))
  (set (nth (children (parent curnode))(incoming-char curnode))
       candidate)
  (catch fail-tag (fold-tree-into-graph curnode candidate fail-tag)))



(define (fold-tree-into-graph tree-node graph-node abort-address)

  ; This procedure destructively copies labels and transitions from
  ; the tree into the graph.  It makes backup copies of modified nodes.

  (when (not (null? (state-label tree-node)))

    (if (null? (state-label graph-node))

        ; If the current graph node is unlabeled, adopt tree node label.
        (block
         (backup graph-node)
         (set (state-label graph-node)(state-label tree-node)))

        ; Bail out immediately if graph and tree node labels differ.
        (when (not (eq? (state-label graph-node)(state-label tree-node)))
          (throw abort-address 'failed))))

  (dotimes (i alphabet-size)
    (let ((graph-child (nth (children graph-node) i))
          (tree-child  (nth (children tree-node)  i)))

      (when (not (null? tree-child))

        (if (null? graph-child)

            ;If the graph lacks a branch, splice in the
            (block ; corresponding branch from the tree.
             (backup graph-node)(backup tree-child)
             (set (nth (children graph-node) i) tree-child)
             (set (parent tree-child) graph-node)
             (push list-of-splices (cons graph-node tree-child)))

            ; Otherwise, continue walking the tree and the graph.
            (fold-tree-into-graph tree-child graph-child abort-address)))))

  'succeeded)
```