Machine Learning, 7, 109-138 (1991)

Learning Automata from Ordered Examples

SARA PORAT*
Department of Computer Science, University of Rochester

JEROME A. FELDMAN†
Department of Computer Science, University of Rochester

Abstract. Connectionist learning models have had considerable empirical success, but it is hard to characterize exactly what they learn. The learning of finite-state languages (FSL) from example strings is a domain which has been extensively studied and might provide an opportunity to help understand connectionist learning. A major problem is that traditional FSL learning assumes the storage of all examples and thus violates connectionist principles. This paper presents a provably correct algorithm for inferring any minimum-state deterministic finite-state automata (FSA) from a complete ordered sample using limited total storage and without storing example strings. The algorithm is an iterative strategy that uses at each stage a current encoding of the data considered so far, and one single sample string. One of the crucial advantages of our algorithm is that the total amount of space used in the course of learning for encoding any finite prefix of the sample is polynomial in the size of the inferred minimum state deterministic FSA. The algorithm is also relatively efficient in time and has been implemented. More importantly, there is a connectionist version of the algorithm that preserves these properties. The connectionist version requires much more structure than the usual models and has been implemented using the Rochester Connectionist Simulator. We also show that no machine with finite working storage can iteratively identify the FSL from arbitrary presentations.

Keywords. Learning, finite automata, connectionist

1. Introduction

The ability to adapt and learn has always been considered a hallmark of intelligence, but machine learning has proved to be very difficult to study. There is currently a renewed interest in learning in the theoretical computer science community (Valiant, 1984; Valiant, 1985; Kearns, et al., 1987; Natarajan, 1987; Rivest & Schapire, 1987; Rivest & Schapire, 1987) and a, largely separate, explosive growth in the study of learning in connectionist networks (Hinton, 1987). One purpose of this paper is to establish some connections (sic) between these two research programs.

The setting for this paper is the abstract problem of inferring Finite State Automata (FSA) from sample input strings, labelled as + or − depending on whether they are to be accepted or rejected by the resulting FSA. This problem has a long history in theoretical learning studies (Angluin, 1976; Angluin, 1981; Angluin, 1987) and can be easily mapped to common connectionist situations. There are arguments (Brooks, 1987) that interacting FSA constitute a natural substrate for intelligent systems, but that issue is beyond the scope of this paper.

*Current Address: Science & Technology, IBM Israel Ltd., Technion City, Haifa, Israel.
†Current Address: International Computer Science Institute, Berkeley, CA.

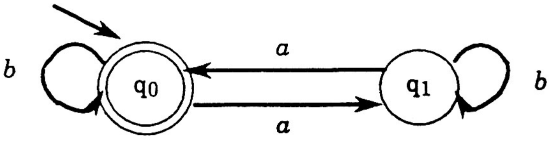


Figure 1. A parity FSA.

We will start with a very simple sample problem. Suppose we would like a learning machine to compute an FSA that will accept those strings over the alphabet $\{a, b\}$ that contain an even number of a's. One minimal answer would be the following two-state FSA shown in Figure 1.

We adopt the convention that states drawn with one circle are rejecting states and those drawn with a double circle are accepting. The FSA always starts in state q_0, which is accepting iff the empty string λ is to be accepted. We will present in Section 3 an algorithm that will always learn the minimum state deterministic FSA for any finite state language which is presented to the learning algorithm in strict lexicographic order. There are a number of issues concerning this algorithm, its proof and its complexity analysis that are independent of any relation to parallel and connectionist computation.

It turns out that the "even a's" language is the same as the well-studied "parity problem" in connectionist learning (Hinton, 1987). The goal there is to train a network of simple units to accept exactly binary strings with an even number of 1's. In the usual connectionist situation, the entire string (of fixed length) is presented to a bottom layer of units and the answer read from a pair of decision units that comprise the top layer. There are also intermediate (hidden) units and it is the weights on connections among all the units which the connectionist network modifies in learning.

The parity problem is very difficult for existing connectionist learning networks and it is instructive to see why this is so. The basic reason is that the parity of a string is a strictly global property and that standard connectionist learning techniques use only local weight-change rules. Even when a network can be made to do a fairly good job on a fixed-length parity problem, it totally fails to generalize to shorter strings. Of course, people are also unable to compute the parity of a long binary string in parallel. What we do in this situation is much more like the FSA of Figure 1. So one question concerns the feasibility of connectionist FSA systems.

There are many ways to make a connectionist version of an FSA like that of Figure 1. One of the simplest assigns a connectionist unit to each state and to the answer units + and −. It is convenient to add an explicit termination symbol $\vdash$ and to use conjunctive connections (Feldman & Ballard, 1982) to capture transitions. The "current input letter" is captured as the activity of exactly one of the top three units. Figure 2 is the equivalent of Figure 1 under this transformation.

Thus unit 0 corresponds to the accepting state q_0 in Figure 1 because when it is active and the input symbol is $\vdash$, the answer + is activated. Similarly, activity in unit 1 and in the unit for a leads to activity in unit 0 for the next time step. Note that activity is allowed in only one of the units 0, 1, +, − for each step of the (synchronous) simulation. In Section 5, we will show how the construction of Section 3 can be transformed into one which

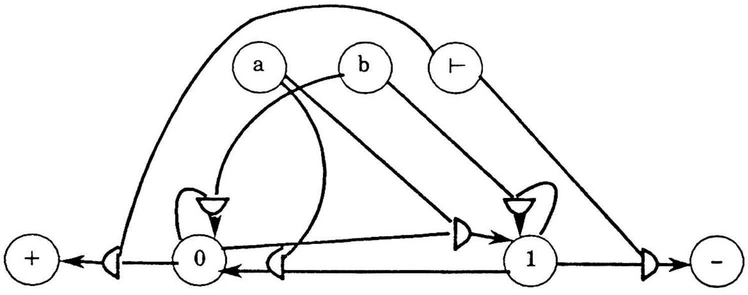


Figure 2. A connectionist parity network.

has a connectionist system learn to produce subnets like that of Figure 2. There have been some attempts (Williams, 1987) to extend conventional connectionist learning techniques to sequences, but our approach is quite different. It would be interesting to compare the various techniques.

More generally, we are interested in the range of applicability of various learning techniques and on how theoretical results can contribute to the development of learning machines. The starting point for the current investigation was the application of the theory of learning FSA to connectionist systems. As always, the assumptions in the two cases were quite different and had to be reconciled. There is, as yet, no precise specification on what constitutes a "connectionist" system, but there are a number of generally accepted criteria. The truism that any machine can be built from linear threshold elements is massively irrelevant. Connectionist architectures are characterized by highly parallel configurations of simple processors exchanging very simple messages. Any system having a small number of control streams, an interpreter or large amounts of passive storage is strongly anticonnectionist in spirit. It is this last characteristic that eliminated almost all the existing formal learning models as the basis for our study. Most work has assumed that the learning device can store all of the samples it has seen, and base its next guess on all this data. There have been a few studies on "iterative" learning where the guessing device can store only its last guess and the current sample (Wiehagen, 1976; Jantke & Beick, 1981; Osherson, et al., 1986). Some of the techniques from Jantke and Beick (1981) have been adapted to prove a negative result in Section 4. We show that a learning device using any finite amount of auxiliary memory cannot learn the Finite State Languages (FSL) from unordered presentations.

Another important requirement for a model to be connectionist is that it adapts. That is, a connectionist system should reflect its learning directly in the structure of the network. This is usually achieved by changing the weights on connections between processing elements. One also usually requires that the learning rule be local; a homunculus with a wire-wrap gun is decidedly unconnectionist. All of these criteria are based on abstractions of biological information processing and all were important in the development of this paper. The algorithm and proof of Section 3 do not mention them explicitly, but the results arose from these considerations. After a pedagogical transition in Section 5.1, Section 5.2 presents the outline of an FSL learner that is close to the connectionist spirit. Error tolerance, another connectionist canon, is only touched upon briefly but appears to present no fundamental difficulties.

In a general way, the current guess of any learning algorithm is an approximate encapsulation of the data presented to it. Most connectionist paradigms and some others (Valiant, 1984; Horning, 1969) assume that the learner gets to see the same data repeatedly and to refine its guesses. It is not surprising that this can often be shown to substitute, in the long run, for storing the data. As mentioned above, we show in Section 4 that in general an algorithm with limited storage will not be able to learn (even) FSA on a single pass through the data. But there is a special case in which one pass does suffice and that is the one we consider in Section 3.

The restriction that makes possible FSA learning in a single pass is that the learning algorithm be presented with the data in strict lexicographic order, that is, $\pm\lambda$, $\pm a$, $\pm b$, $\pm aa$, ... In this case the learner can construct an FSA, referred to also as the current guess, that exactly captures the sample seen so far. The FSA is nondeterministic, but consistent—every path through the FSA gives the same result for every sampled string considered so far. It turns out that this is a minimal state FSA consistent with the data and can thus be viewed as best guess to date. The idea of looking at strict lexicographic orders came to us in considering the algorithm of Rivest and Schapire (1987). Their procedure is equivalent to receiving ± samples in strict order. One would rarely expect such benign training in practical situations, but the general idea of starting with simpler examples is common.

Since the sample is presented in lexicographic order, our learning algorithm will be able to build up its guesses in a cumulative way. If the empty string is (is not) in the inferred language L, then the first guess is a machine with one accepting (rejecting) state. Each subsequent example is either consistent with the current guess, or leads to a new guess. The details of this comprise the learning algorithm of Section 3. When a new state is added to the current guess, a set of incoming and outgoing links to and from this new state are added. Consider the "even *a*'s" language. With the sample $+\lambda$, the initial accepting state q_0 has links to itself under every letter. These links are all mutable and may later be deleted. When $-a$ is presented, the self-looping link under a is deleted and replaced by a permanent link to a new rejecting state q_1. We further add a mutable link from q_0 to q_1 under b, and the whole set of links from q_1. Figure 3 shows the guess for the "even *a*'s" language after the initial sample $+\lambda$ and $-a$. The link from q_0 to q_1 under b is pruned when $+b$ is presented. $+aa$ will imply the deletion of the current self-link of q_1 under a, and $-ab$ will finally change the guess to that of Figure 1.

The remainder of the paper is divided into three major sections. Section 3 considers the general problem of learning FSA from lexicographically ordered strings. An algorithm

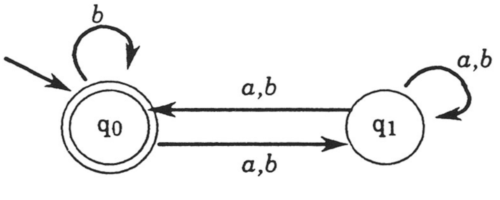


Figure 3. An intermediate guess for the parity FSA.

is presented and its space and time complexity are analyzed. The proof of correctness for this algorithm in Section 3.3 uses techniques from verification theory that have apparently not been used in the learning literature. In Section 4 we show that the strong assumption of lexicographic order is necessary—no machine with finite storage can learn the FSA from arbitrary samples. Section 5 undertakes the translation to the connectionist framework. This is done in two steps. First a distributed and modular, but still conventional version, is described. Then a transformation of this system to a connectionist network is outlined. Some general conclusions complete the paper.

2. Relation to previous work

The survey by Angluin and Smith (1983) is the best overall introduction to formal learning theory; we just note some of the most relevant work. Our learning algorithm (to be presented in the next section) identifies the minimum state deterministic FSA (DFSA) for any FSL *in the limit*: Eventually the guess will be the minimum state DFSA, but the learner has no way of knowing when this guess is found. The learner begins with no *a priori* knowledge. We can regard the sample data as coming from an unknown machine that identifies the inferred FSL. As stated above, our algorithm is an iterative strategy that uses at each stage a current encoding of the data considered so far, and the *single* current sample string. One of the crucial advantages of our algorithm is that the total amount of space used in the course of learning, for encoding any finite prefix of the sample, is polynomial in the size of the inferred minimumm-state DFSA. Any encoding of a target grammar requires $0(n^2)$ space, and our algorithm is $0(n^2)$.

Iterative learning strategies have been studied in Wiehagen (1976); Jantke and Beick (1981); and Osherson, et al. (1986). Jantke and Beick (1981) prove that there is a set of functions that can be identified in the limit by an iterative strategy, using the strict lexicographic presentations of the functions, but this set cannot be identified in the limit by an iterative strategy using arbitrary presentations. The proof can be slightly modified in order to prove that there is no iterative algorithm that can identify the FSL in the limit, using arbitrary representations for the languages. We generalize the definition of an iterative device to capture the ability to use any finite auxiliary memory in the course of learning. Hence, our result is stronger than that in Jantke and Beick (1981).

Gold (1967) gives algorithms for identifying FSA in the limit both for *resettable* and *nonresettable* machines where resettable machines are defined by the ability or need to "reset" the automation to some start state. These algorithms identify by means of enumeration. Each experiment is performed in succession, and in each stage all the experiments performed so far are used in order to construct the next guess. Consequently, the storage needed until the correct guess is reached is exponential in the size of the minimum state DFSA. The enumeration algorithm for resettable machines has the advantage (over our algorithm) that it does not specify the experiments to be performed; it can use any data that identifies the inferred FSL. This property is not preserved when the machines are nonresettable.

Gold (1972) introduces another learning technique for identifying a minimum state DFSA in the limit by experimenting with a resettable machine. This variation is called the *state* *characterization* method which is much simpler computationally. This technique specifies the experiments to be performed, and again has the disadvantage of having to monitor an infinitely increasing storage area.

Angluin (1987) bases her result upon the method of state characterization, and shows how to infer the minimum state DFSA by experimenting with the unknown automata (asking membership queries), and using an oracle that provides counterexamples to incorrect guesses. Using this additional information Angluin provides an algorithm that learns in time polynomial in the maximum length of any counterexample provided by the oracle, and the number of states in the minimum-state DFSA. Angluin's algorithm differs from our approach mainly in the use of an oracle that answers equivalence queries in addition to accepting or rejecting certain sample strings. The algorithm is comparable to ours in the sense that it uses experiments that are chosen at will.

Recently Rivest and Schapire (1987) presented a new approach to the problem of learning in the limit by experimenting with a nonresettable FSA. They introduce the notion of *diversity* which is the number of equivalence classes of *tests* (basically, an experiment from any possible state of the inferred machine). The learning algorithm uses a powerful oracle for determining the equivalence between tests, and finds the correct DFSA in time polynomial in the diversity. Since the lower bound on the diversity is log the number of states, and it is the best possible, this algorithm is practically interesting. Again, the experiments in this algorithm are chosen at will, and in fact they are a finite prefix of a lexicographically ordered sample of the inferred language.

Another variation of automation identification is that from a *given* finite subset of the input-output behavior. Bierman and Feldman (1972) discuss this approach. The learning strategy there includes an adjusted parameter for inferring DFSAs with varying degrees of accuracy, accomplished by algorithms with varying complexities. In general, Gold (1978) and Angluin (1978) prove that finding a DFSA of n states or less that is compatible with a given data is NP-complete. On the other hand, Trakhtenbrot and Barzdin (1973) and Angluin (1976) show that if the sample is *uniform-complete*, i.e., consists of all strings not exceeding a given length and no others, then there is a polynomial time algorithm (on the size of the whole sample) that finds the minimum state DFSA that is compatible with it. Note that the sample size is exponential in the size of the longest string in the sample. We can regard our algorithm as an alternative method for identifying the minmum state DFSA from a given uniform-complete sample. As stated above, our algorithm is much more efficient in space, since it does not access the whole sample, but rather refers to it in succession, and needs just a polynomial space in the number of states in the minimum state DFSA. The time needed for our algorithm is still polynomial in the size of the whole sample, though logarithmic in an amortized sense, as we show in Section 3.4.

Recently, Miri Sharon in her Master's thesis (Sharon, 1990) presented an algorithm for inferring FSA in the FS-IT model. The algorithm is based on Ibarra and Jiang, Lemma 1 (Ibarra & Jiang, 1988), and on the fact that the union, intersection, complement and reduction operations on finite automata, all require time and space polynomial in the size of the machine. Sharon's algorithm improves ours in that the time needed for each sample, as well as the storage, is polynomial in n, the size of the minimum state DFSA. The size of the inferred FSAs is not monotonically increasing; yet it is bounded by 6*n*n−3*n. The design of the algorithm is inherently sequential and centralized.

3. Sequential version for learning FSA

3.1. Notation and definitions

We use the following notation and definitions:

- A *finite-state automata* (FSA) M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a finite nonempty set of *states.*
  - Σ is a finite nonempty set of *letters.*
  - δ is a *transition function* that maps each pair (q, σ) to a set of states, where $q \in Q$ and $\sigma \in \Sigma$. This function can be represented by the set of *links* E so that $(p\ \sigma\ q) \in E$ iff $q \in \delta(p, \sigma)$. Each link is either *mutable* or *permanent.*

  δ can be naturally extended to any string $x \in \Sigma^*$ in the following way: $\delta(q, \lambda) = \{q\}$, and for every string $x \in \Sigma^*$ and for every letter $\sigma \in \Sigma$, $\delta(q, x\,\sigma) = \{p | (\exists\, r \in Q)(r \in \delta(q, x)$ and $p \in \delta(r, \sigma))\}$.
  - q_0 is the *initial* state, $q_0 \in Q$.
  - F is the set of *accepting* states, $F \subseteq Q$. ($Q - F$ is called the set of rejecting states).
  - The *parity* f(q) of a state $q \in Q$ is + if $q \in F$ and is − if $q \in Q - F$. By extension, assuming for some $q \in Q$ and $\sigma \in \Sigma$ that $\delta(q, \sigma) = \emptyset$, we define the parity of this state-symbol pair $f(q, \sigma)$ to be + if all successors of q under σ are + and − if they are all −. If all $r \in \delta(q, \sigma)$ do not have the same parity, then $f(q, \sigma)$ is undefined.
- A *deterministic* FSA (DFSA) is an FSA where the transition function δ is from $Q \times \Sigma$ into Q.
- The *language* L(M) accepted by a DFSA, M, is the set $\{x \in \Sigma^* \mid \delta(q_0, x) \in F\}$.
- Given a regular language L, we denote by M_L the (up to isomorphism) minimum state DFSA s.t. $L(M_L) = L$. Q_L is the state set of M_L.

The late lower case letters v, w, x, y, z will range over strings. Given a current FSA, the new string to be considered is denoted by w (the wrecker that may break the machine). Lower case letters p, q, r, s, t range over names of states. Whenever the current w wrecks the current guess, a new state, denoted by s (supplemental state) is added. σ, ϕ, ψ will range over letters, and i, j, k, m, n over the natural numbers.

- $M^x = (Q^x, \Sigma, \delta^x, q_0, F^x)$ is the FSA, referred to also as the *guess*, after the finite prefix $\pm\lambda, \ldots, \pm x$ of the complete lexicographically ordered sequence. E^x is the corresponding set of links.
- For $x \in \Sigma^*$, succ(x) stands for the string following x in the lexicographic order.
- The incremental construction of M^x admits for every state q, a unique string *minword*(q) that leads from q_0 to q using permanent links only. The path for *minword*(q) is referred to as the *basic* path to state q. These basic paths, which cover all the permanent links, form a spanning tree on the set Q^x.
- The guessing procedure also establishes for any M^x and any string y, unless $y =$ *minword*(q) for some q, the existence of a unique state p, a letter ϕ and a string $z \in \Sigma^*$, such that $y =$ *minword*(p)ϕz, and all the links from p under ϕ are mutable links. We

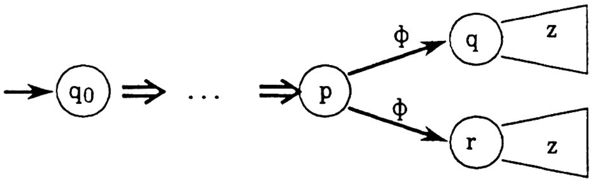


Figure 4. Tested state, tested letter and tested tail.

refer to these state, letter and string as the *tested state*, *tested letter* and *tested tail* (respectively) for y in M^x. Figure 4 shows the tree of all the paths for some string in some FSA, indicating the tested state p, tested letter ϕ and tested tail z.

We use the convention of representing a permanent link by ⇒ and a mutable link by →.

- For a given M^x and a word y, a path for y in M^x is *right* if it ends with an accepting state and $y \in L$, or it ends with a rejecting state and $y \notin L$. Otherwise, this path is called *wrong*.
- For two strings $x, y \in \Sigma^*$, and a language L, $x =_L y$ if both strings are in L, or both are not in L.

3.2. *The learning algorithm*

Let L be the regular language to be incrementally learned. Initially M^λ is constructed according to the first example $\pm\lambda$. $Q^\lambda = \{q_0\}$; $E^\lambda = \{q_0\ \sigma\ q_0 \mid \sigma \in \Sigma\}$ and each link is mutable. If $\lambda \in L$, then q_0 is an accepting state, otherwise it is a rejecting one. *minword*(q_0) is set to λ.

Given M^x, the value *minword*(q) for every $q \in Q^x$, and a new string $\pm w$, $w = succ(x)$, the learning algorithm for constructing the new M^w is given in Figure 5. The algorithm is annotated with some important assertions (invariants in some control points) written between set brackets $\{\ldots\}$.

The subroutine *delete-bad-paths* (M, y, accept) is a procedure that constructs a new FSA out of the given M, in which every path for y leads to an accepting state iff accept = true. In the case y = w, *delete-bad-paths* breaks all wrong paths (if any) for w in M. In the case $y < w$, the paths in M are checked against the behavior of the old machine old-M. In any case, each bad path for y in M is broken by deleting its *first mutable link*. Note that all the first mutable links along bad paths are from the same tested state p for y in M. Furthermore, if all the paths for y in M are bad (and we will show that this can happen only if y = w), then after the execution of *delete-bad-paths* there will be no link from p under the tested letter for y in M. Such an execution will be followed by an execution of *insert-state*.

The procedure *insert-state* constructs a new FSA by extending the given spanning tree defined by the permanent links in old-M. A new state s is added. Let p and ϕ be the tested state and tested letter for w in old-M. Note again that all the mutable links from p under ϕ

```
begin
    old-M ← $M^x$;
    if w ∈ L then accept-w ← true else accept-w ← false;
    new-M ← delete-bad-paths (old-M, w, accept-w); /* Drop mutable links */
        {all the paths for w in new-M are right}
    if there is no path for w in new-M
    then {all the paths for w in $M^x$ are wrong}
        {old-M is consistent with all strings up through x}
        repeat
            new-M ← insert-state; /* Insert new state, a new permanent link, and */
                                  /* new mutable links */
            {new-M may be inconsistent with previous strings}
            y ← λ;
            while succ(y) < w /* check against all previous strings */
                begin
                    y ← succ(y);
                    if all the paths for y in old-M lead to accepting states
                    then accept ← true
                    else accept ← false;
                    {accept is true if and only if y ∈ L}
                    {there exists a right path for y in new-M}
                    new-M ← delete-bad-paths (new-M, y, accept) /* Drop mutable links */
                    {new-M is now correct with respect to the strings λ, . . ., y}
                end;
            old-M ← new-M;
            {old-M is consistent with all strings up through x}
            new-M ← delete-bad-paths (new-M, w, accept-w) /* Drop mutable links */
        until there exists a path for w in new-M;
    output new-M {$M^w$ will be the new FSA new-M}
end
```

Figure 5. The learning algorithm for constructing M^w.

had been deleted in the last execution of *delete-bad-paths*. A new permanent link (p ϕ s) is added. *minword*(s) is set to *minword*(p)ϕ. The parity of s is set under the following rule: If *minword*(s) = w, then s is an accepting state iff accept = true. In other words, in that case the parity of s is opposite to those states at the ends of the paths for w in old-M. If *minword*(s) < w then s is an accepting state iff all the paths for *minword*(s) in old-M end with accepting states. Next, mutable links to and from the new state s are added according to the following rule: For any existing state q, and for any letter σ, if *minword*(s)σ > *minword*(q), then add the mutable link (s σ q). Also, in the other direction, if the current links from q under σ are all mutable, and *minword*(q)σ > *minword*(s), add the mutable link (q σ s). Note that this rule adds (for q = s) all possible self links for the new state s. In other words, for every letter σ, the mutable link (s σ s) exists after *insert-state*.

Given M^x and w = succ(x), if all the paths for w in M^x are wrong, then the repeat loop takes place. This loop defines the extension process, which is a repetition of one or more applications of the *insert-state* procedure. It is easy to see that there will be at most $|w| - |minword(p)|$ insertions (application of *insert-state*), where p is the tested state for w in M^x. Suppose there are i insertions between M^x and M^w, each adding a new state. We can refer to a sequence of length i of machines: M_0^w, M_1^w, . . . , M_{i-1}^w, each of which is the old-M at the beginning of the repeat loop. M_0^w is the old-M as set to M^x at the beginning, the others are iteratively set in the body of the repeat loop. For every j, $0 \leq j \leq i - 1$, the execution of *insert-state* defines a new machine out of M_j^w, referred to as $M_j^w(\lambda)$. Thereafter, for every j, $0 \leq j \leq i - 1$, and for every y, succ(λ) $\leq y < w$, the execution of *delete-bad-paths* within the while loop defines a new machine (possibly the same as the preceding one), referred to as $M_j^w(y)$, indicating that this machine is ensured to be consistent with those strings up through y.

The algorithm was successfully implemented as a student course project by Lori Cohn, using C.

Before going on to the correctness proof ot his algorithm, we will discuss an example over the alphabet $\Sigma = \{a, b\}$. Suppose that the unknown language L is "number of *a*'s is at most 1, and the number of *b*'s is at least 1" or $bb^*(\lambda + ab^*) + abb^*$ as given by a regular expression. Figure 6 below shows some of the guesses.

Initially, since the first input example is $-\lambda$, q_0 is a rejecting state, having both (a, b) self-loop mutable links. For the next example $-a$, *delete-bad-paths* does not change the machine, hence M^a is the same as M^λ. When $+b$ is encountered, the mutable link $(q_0\ b\ q_0)$ is deleted, and the repeat loop takes place ($M_0{}^b = M^a$). A new state $s = q_1$ is added, and a new permanent link $(q_0\ b\ q_1)$ is added. *minword*(q_1) is set to *b*. Since *minword*(q_1) is the current example, q_1 gets the opposite parity from that of q_0. Hence, q_1 is an accepting state. The new mutable links are $(q_1\ a\ q_1)$, $(q_1\ b\ q_1)$, $(q_1\ a\ q_0)$ and $(q_1\ b\ q_0)$. Note that $(q_0\ a\ q_1)$ is not added, since $a =$ *minword*$(q_0)a$ is less than $b =$ *minword*(q_1). The new machine is $M_0^b(\lambda)$. Since all the paths (there is only one) for *a* in $M_0^b(\lambda)$ are right with respect to the old machine M_0^b, we get that $M_0^b(\lambda) = M_0^b(a)$. The only path for *b* is right, hence M^b is $M_0^b(a)$. The examples $-aa$ and $+ab$ do not change the current guess. When $+ba$ is encountered, there exists a right path $\langle q_0\ b\ q_1,\ q_1\ a\ q_1\rangle$ and there exists a wrong path $\langle q_0\ b\ q_1,\ q_1\ a\ q_0\rangle$. The first (and only) mutable link $(q_1\ a\ q_0)$ along the wrong path is deleted. A similar treatment is involved for the example $+bb$. Note that at this stage M^{bb} is a DFSA, but obviously $L(M^{bb}) \neq L$.

The next string $-aaa$ does not change the current guess, but $-aab$ causes a new application of *insert-state*. A new state q_2 is added, with *minword*(q_2) being *a*. The string *aa* is the first string that changes the machine while testing $M_0^{aab}(b)$ against the old machine M_0^{aab}. The new mutable link $(q_2\ a\ q_1)$ is deleted. Other new mutable links are deleted while retesting *ab*, *ba* and *bb*. The execution of *delete-bad-paths* on $M_0^{aab}(aaa)$ (= $M_0^{aab}(bb)$) deletes the two mutable links $(q_2\ a\ q_2)$ and $(q_2\ a\ q_0)$, hence causing the nonexistence of a path for *aab* in the new machine. Thus, a new insertion is applied, replacing the two mutable links $(q_2\ a\ q_2)$ and $(q_2\ a\ q_0)$ by a new permanent link from q_2 to a new state q_3 under *a*. The retesting of the strings *a*, *b*, and *aa* against M_1^{aab}(= $M_0^{aab}(aaa)$) cause no change (no deletion) on $M_1^{aab}(\lambda)$. Some of the new mutable links are deleted while retesting *ab*, *ba*, *bb* and *aaa*. In $M_1^{aab}(aaa)$ there exist three right paths for *aab*, and one wrong path, causing the deletion of $(q_3\ b\ q_1)$, yielding M^{aab}. Given this guess, M^{aab}, the only path for *aba* is wrong. Note that this path has two mutable links and the first one is now replaced by a permanent link to the new accepting state q_4. The retesting deletes some of the new mutable links just recently added within the insertion of q_4.

When $-baa$ is checked, given M^{abb}, there are three right paths and two wrong paths. The tested state is q_1, the tested letter is a, and the tested tail is a. The first mutable link along the wrong paths is (q_1 a q_1). Hence, this link is deleted leaving (q_1 a q_4) as the only mutable link from q_1 under a. This link is the first mutable link along the three right paths. When $-aaab$ is checked, given M^{aaaa}, there are again three right paths and two wrong paths. This time, two mutable links (q_3 a q_0) and (q_3 a q_2) are deleted, each breaking a different wrong path. Note that this deletion leaves only one right path for *aaab*, as the two deleted links served as second mutable links along two of the original right paths for *aaab* in M^{aaaa}. The correctness proff will show that after the deletion process, there exists at least one right path for the current sample. Finally, M^{abba} accepts the langauge L.

3.3. *The correctness proof*

Next we prove that the guessing procedure is correct. We first show that each M^x satisfies a set of invariants. Consider the following constraints for a FSA $M = (Q, \Sigma, \delta, q_0, F)$ and a given string $x \in \Sigma^*$:

(1) *Consistency*:
$\forall y \leq x$, all the paths for y are right.

(2) *Completeness*:
$(\forall q \in Q)(\forall \sigma \in \Sigma)$
$((\exists r \in Q)(\delta(q, \sigma) = \{r\}$ and $(q\ \sigma\ r)$ is a permanent link)
or
$(\exists Q' \subseteq Q)(\delta(q, \sigma) = Q'$ and $Q' \neq \emptyset$ and $(\forall r \in Q')((q\ \sigma\ r)$ is a mutable link)))

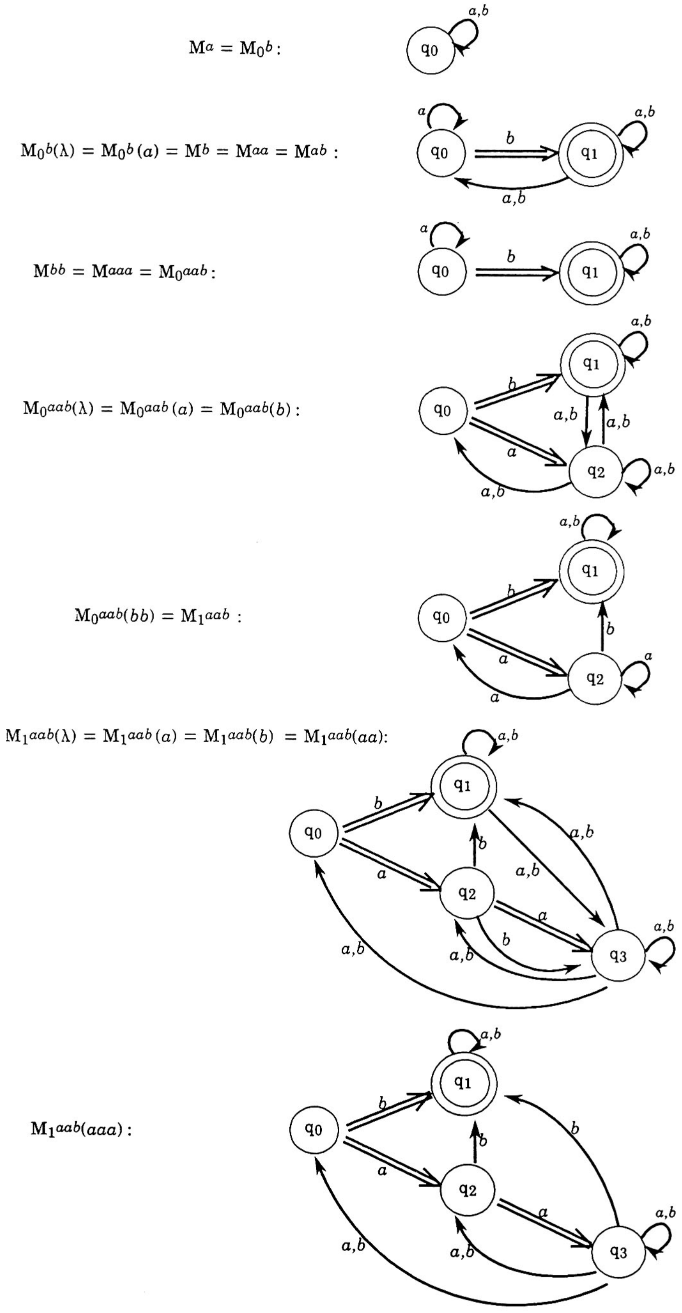
(3) *Separability*:
$(\forall q, r \in Q \mid q \neq r)(\exists y \in \Sigma^*)(minword(q)y \neq_L minword(r)y$ and $minword(q)y \leq x$ and $minword(r)y \leq x)$

(4) *Minimality*:
$((\forall q \in Q)(\forall y \in \Sigma^*) \mid q \in \delta\ (q_0, y))(y \geq minword(q))$

(5) *Minword-property*:
$(\forall q \in Q)(x \geq minword(q)$ and there is a unique string, namely $minword(q)$, that has a path from q_0 to q using permanent links only)

Note that some properties refer to a designated sample string x. We say that M y-satisfies a set of properties if whenever some property in this set relates to x, and y is substituted, then M satisfies the corresponding conjunct of properties.

The consistency constraint is the obvious invariant we would expect the learning algorithm to maintain. The completeness constraint means that for every state and for every letter there is either a permanent link that exits this state under this letter, or else there is a non-empty set of links leaving the state under this letter, all of which are mutable. The separability constraint together with Myhill-Nerode theorem (Hopcroft & Ullman, 1979) will lead to the claim that for each sample x, the number of states in M^x is at most the number of states in the minimum-state DFSA for the inferred language L. This can be established



$M^a = M_0^b$:

$M_0^b(\lambda) = M_0^b(a) = M^b = M^{aa} = M^{ab}$:

$M^{bb} = M^{aaa} = M_0^{aab}$:

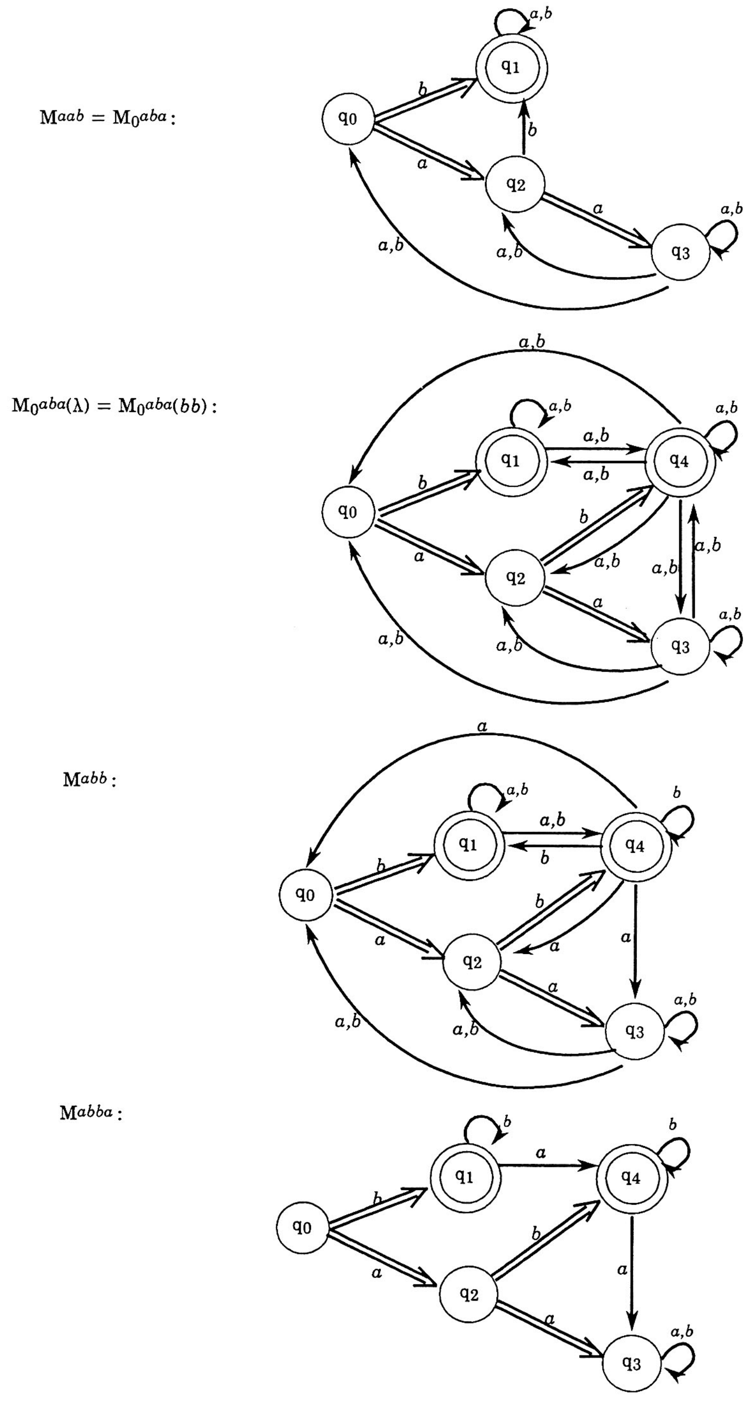
$M_0^{aab}(\lambda) = M_0^{aab}(a) = M_0^{aab}(b)$:

$M_0^{aab}(bb) = M_1^{aab}$:

$M_1^{aab}(\lambda) = M_1^{aab}(a) = M_1^{aab}(b) = M_1^{aab}(aa)$:

$M_1^{aab}(aaa)$:

Figure 6. Learning bb* (λ + ab*) + abb*.



$M^{aab} = M_0^{aba}$:

$M_0^{aba}(\lambda) = M_0^{aba}(bb)$:

M^{abb}:

M^{abba}:

Figure 6. continued

by continually preserving the minimality constraint. The minword-property together with the completeness constraint implies the existence of the spanning tree formed by the permanent links.

Following are simple facts that are implied by the above properties. We will refer to these facts frequently in the sequel.

Fact 1. $\forall q \in Q$, $\forall y \in \Sigma^*$, there is a path in M for y from the state q. (Implied by the completeness constraint.)

Fact 2. $\forall y \in \Sigma^*$, if $\forall q \in Q$, $y \neq$ *minword*(q), then there exists a unique tested state, tested letter and tested tail for y in M. (Implied by the completeness constraint.)

Fact 3. $\forall y \in \Sigma^*$, if $y \leq x$, then all the paths for y from q_0 lead either to accepting states, or they all lead to rejecting states. (Implied by the consistency constraint.)

Fact 4. $\forall q \in Q$, *minword*(q) has a unique right path from q_0 to q through permanent links only. (Implied by the consistency, completeness and minword-property constraints.)

Fact 5. $\forall q \in Q$, $\forall y \in \Sigma^*$, $\forall z \in \Sigma^*$, if there exists a path for y from q_0 to q that uses at least one mutable link, then yz > *minword*(q)z. (Implied by *Fact 4* and the minimality constraint.)

The correctness of the algorithm of Figure 5 will be established by showing that after each completion of the algorithm, yielding a new guess M^w by using the current sample ±w, the five constraints are w-satisfied.

Clearly, M^λ λ-satisfies the constraints. Suppose (inductively) that M^x x-satisfies these constraints, and let w = succ(x).

If all the paths for w in M^x are right, then $M^w = M^x$, and if M^x x-satisfies the invariants, then M^w w-satisfies them.

By the minword-property constraint, w > *minword*(q) for each q. By *Fact 2*, let p, ϕ and z be the tested state, tested letter and tested tail (respectively) for w in M^x. Consider any of the states r, such that $(p\ \phi\ r) \in E^x$. By the definition of the tested elements, $(p\ \phi\ r)$ is a mutable link. By *Fact 5*, *minword*(r)z ≤ w. Therefore, *minword*(r)z has already been checked. Hence, by *Fact 3*, all the paths for z from r behave the same, i.e., $\delta^x(r, z) \subseteq F^x$ or $\delta^x(r, z) \subseteq Q^x - F^x$. Thus, either all the paths for w that use the mutable link $(p\ \phi\ r)$ are wrong paths, or all of them are right paths.

If there exist a wrong path and a right path for w in M^x, then by breaking each possible wrong path for w by *delete-bad-paths*, the consistency constraint is w-satisfied in M^w. To establish the completeness constraint in this case, note that all the deleted mutable links are of the form $(p\ \phi\ r)$, where p is the tested state, ϕ is the tested letter, and r is some state in M^x. Because there exists a right path for w in M^x, there must be a mutable link $(p\ \phi\ r')$ in M^x that is not deleted, and so M^w satisfies the completeness constraint. The other three constraints, separability, minimality and minword-property are obviously w-satisfied.

If all the paths for w in M^x are wrong, the expansion process takes place. Suppose there are i insertions in between M^x and M^w. We will show that the intermediate FSAs M_0^w, M_1^w, ..., M_{i-1}^w all x-satisfy the consistency, the completeness, the minimality and the minword-property constraints. Moreover, all the paths for w in $M_j^w (0 \leq j \leq i - 1)$ are wrong, causing the re-application of *insert-state*.

M_0^w, being the same as M^x, obviously x-satisfies the consistency, the completeness, the minimality and the minword-property constraints, and all the paths for w in M_0^w are wrong.

Suppose for the moment that M_j^w, $0 \leq j \leq i - 1$, x-satisfies these four constraints. By the minword-property constraint, $w > \mathit{minword}(q)$ for each q. By *Fact 2*, let p, ϕ and z be the tested state, tested letter and tested tail (respectively) for w in M_j^w. Let s be the new state in $M_j^w(\lambda)$. In constructing $M_j^w(\lambda)$, the whole set of mutable links from p under ϕ in M_j^w are deleted but they are replaced by the new permanent link, $(p\ \phi\ s)$. This, plus the fact that we add all possible self-looping links for the new state s, establishes the part of the completeness constraint that ensures a nonempty set of links for each state and each letter. The other part—indicating that this set is either a singleton of a permanent link, or a set of mutable links—is easily implied by the construction. By the definition of *insert-state*, and the fact that permanent links are never deleted in *delete-bad-paths*, $M_j^w(\lambda)$ obviously w-satisfies the minword-property. As for the minimality constraint, suppose by way of contradiction that there exists a state q, such that the minimal string y that leads from q_0 to q is smaller than $\mathit{minword}(q)$. By the minword-property, this path uses at least one mutable link. By the minimality constraint of M_j^w, at least one of those mutable links is a new one just added while constructing $M_j^w(\lambda)$. Consider one of them, $(r\ \sigma\ t)$. (Note that it is either the case that $r = s$ or $t = s$.) From the way new links are added we immediately get a contradiction to the minimality assumption of y. Hence we conclude that $M_j^w(\lambda)$ w-satisfies the completeness, the minword-property and the minimality constraints.

The retesting process (within the while loop) checks the current machine $M_j^w(y)$ against the old machine M_j^w, that is assumed to be consistent up through x. The whole retesting process involves defining a sequence of FSAs: $M_j^w(\lambda)$, $M_j^w(\mathrm{succ}(\lambda))$, $M_j^w(\mathrm{succ}(\mathrm{succ}(\lambda)))$, $\ldots$, $M_j^w(x)$. We will show that each $M_j^w(y)$ is consistent up through y, and that it w-satisfies the completeness, the minword-property and the minimality constraints. For this we need to refer to another inductive hypothesis, that will indicate, for each $y > \lambda$, the similarity between $M_j^w(y)$ and $M_j^w(\lambda)$. Let E' be the set of mutable links that were added while constructing $M_j^w(\lambda)$ from M_j^w. We will claim that each execution of *delete-bad-paths* along the construction of $M_j^w(y)$ out of $M_j^w(\lambda)$ deletes (if at all) only links from E'. Moreover, in the next paragraph we define a subset of E' that will definitely remain in $M_j^w(x)$. A link in this subset will be called a *necessary* link. Intuitively, these links will establish the existence of right paths on $M_j^w(\lambda)$ that reconstruct paths in M_j^w that use one of the mutable links in M_j^w from p under ϕ.

A link $(s\ \sigma\ t)$ in $M_j^w(\lambda)$ is *necessary* iff $t \neq s$, $w > \mathit{minword}(s)\sigma$, and there is a path for $\mathit{minword}(s)\sigma$ from q_0 to t in M_j^w. Figure 7 below shows how M_j^w and $M_j^w(\lambda)$ relate to each other with respect to p, s and t. The mutable link $(s\ \sigma\ t)$ will definitely be added while constructing $M_j^w(\lambda)$, because by *Fact 5* applied on M_j^w, $\mathit{minword}(s)\sigma = \mathit{minword}(p)\phi\sigma > \mathit{minword}(t)$.

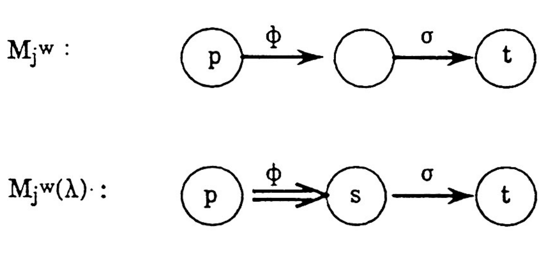


Figure 7. A necessary link $(s\ \sigma\ t)$.

In order to prove that M_{j+1}^w x-satisfies the consistency, the completeness, the minword-property and the minimality constraints (given that M_j^w satisfies these conditions), we refer to another property, namely the *similarity constraint*: For each j, $0 \leq j \leq i - 1$, and each y, $\lambda \leq y \leq x$, $M_j^w(y)$ is the same as $M_j^w(\lambda)$ except for the removal of some of the new mutable links that were added while constructing $M_j^w(\lambda)$ out of M_j^w. Moreover, all the necessary links still exist in $M_j^w(y)$.

For every j, $0 \leq j \leq i - 1$, and for every y, $0 \leq y \leq x$, we will prove the following *intermediate invariant*: $M_j^w(y)$ satisfies the completeness, the minimality and the similarity constraints, it y-satisfies the consistency constraint, and w-satisfies the minword-property constraint.

We have already shown that $M_j^w(\lambda)$ preserves the completeness and the minimality constraints, and that it w-satisfies the minword-property constraint. Obviously, it satisfies the consistency up through λ, and the similarity (to itself). Thus, assuming inductively that $M_j^w(y)$, $\lambda \leq y < x$, satisfies the intermediate invariant, we need to show that so does $M_j^w(\text{succ}(y))$.

Recall that in the current execution of *delete-bad-paths*, the paths for succ(y) are checked against the behavior of succ(y) in M_j^w. If the current execution of *delete-bad-paths* causes no deletions (all the paths for succ(y) are right) then $M_j^w(\text{succ}(y))$ trivially maintains the intermediate invariant.

Otherwise, we show that it cannot be the case that all the paths for succ(y) in $M_j^w(y)$ are wrong. Moreover, if there exists a wrong path for succ(y) it will be broken by deleting one of the non-necessary new mutable links.

Assuming succ(y) has a wrong path in $M_j^w(y)$, we get by *Fact 4* that succ(y) $\neq$ *minword*(q) for each state q in $M_j^w(y)$. By *Fact 2*, let r, ψ and v be the tested state, the tested letter and the tested tail for succ(y) on $M_j^w(y)$. (As before, r is the last state reached by permanent links.) We distinguish between two possible cases:

1) r = s, i.e., succ(y) = *minword*(s)ψv.

Therefore, in M_j^w, the tested state and the tested letter for succ(y) were p and ϕ. Figure 8 shows the relation between M_j^w and $M_j^w(y)$ with respect to p and s. Let q, t be states such that (p ϕ q) and (q ψ t) are links in M_j^w. By the similarity constraint of $M_j^w(y)$, there must be some paths for succ(y) on $M_j^w(y)$ that use the existing necessary

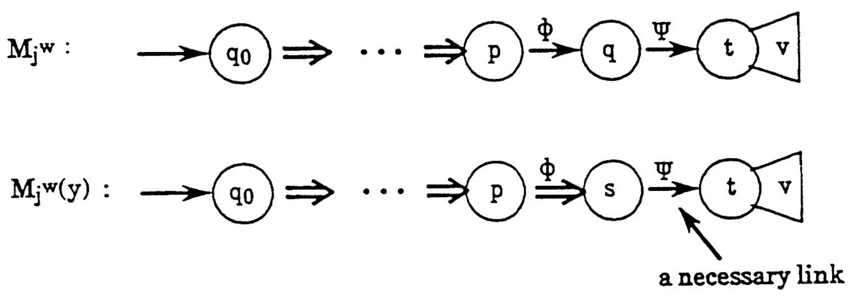


Figure 8. Proving the intermediate invariant: Case 1.

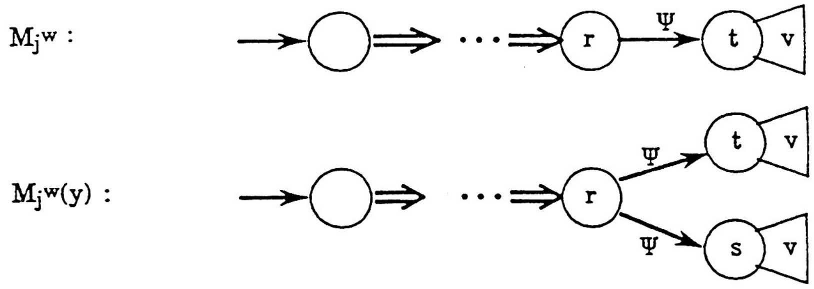


Figure 9. Proving the intermediate invariant: Case 2.

link (s ψ t). By *Fact 5* applied to $M_j^w(y)$, *minword*(t)v < *minword*(s)ψv = succ(y). Finally, by the consistency constraint of $M_j^w(y)$, all the paths for *minword*(t)v in $M_j^w(y)$ are right. Clearly, by *Fact 3* applied to M_j^w, *minword*(t)v $=_L$ succ(y), which implies in turn that all these paths for succ(y) in $M_j^w(y)$ that use the existing necessary link (s ψ t) must be right. Hence, this necessary link (s ψ t) will not be deleted. Other non-necessary mutable links from s that establish wrong paths for succ(y) (and there exists such a wrong one) will be deleted by the current execution of *delete-bad-paths*.

2) r ≠ s. Hence, r, ψ and v serve as the tested state, tested letter and tested tail for succ(y) on M_j^w also, and succ(y) = *minword*(r)ψv. Figure 9 below indicates the relation between M_j^w and $M_j^w(y)$ in this case. Let t be a state in M_j^w such that some paths for succ(y) in M_j^w use the mutable link (r ψ t) (right after the permanent prefix). By the similarity constraint of $M_j^w(y)$, some paths for succ(y) on $M_j^w(y)$ use this existing mutable link. Again, by the consistency constraint, *Fact 3* and *Fact 5* applied to $M_j^w(y)$, all the paths in $M_j^w(y)$ for *minword*(t)v behave the same and are right. By *Fact 3* applied to M_j^w, *minword*(t)v $=_L$ succ(y). Hence, all the paths for succ(y) in $M_j^w(y)$ that use (r ψ t) are right, implying in turn that this (old) mutable link will not be deleted. By the assumption, there exists a wrong path for succ(y) in $M_j^w(y)$. Hence, there exists a new mutable link in $M_j^w(y)$, (r ψ s), that will be now deleted in order to break a bad path. This new mutable link is clearly a non-necessary one.

This terminates the discussion on the relation between $M_j^w(y)$ and $M_j^w(\text{succ}(y))$, and based on this we can easily conclude that the intermediate invariant is satisfied by $M_j^w(\text{succ}(y))$. Consequently, $M_j^w(x)$ satisfies this intermediate invariant, and in particular it is consistent up through x. For $0 \leq j < i - 1$, $M_{j+1}^w = M_j^w(x)$. We get the desired hypothesis that this M_{j+1}^w x-satisfies the consistency, the completeness and the minimality constraints, and that all the paths for w in this new machine are wrong. Since $M_j^w(x)$ w-satisfies the minword-property constraint, and there exists a wrong path for w in $M_j^w(x)$, we get by *Fact 4* applied to $M_j^w(x)$ ($=M_{j+1}^w$) that M_{j+1}^w x-satisfies the minword-property constraint. For $j = i - 1$, we break all the wrong paths for w on $M_{i-1}^w(x)$ by deleting the first mutable links along them. By similar arguments as above, we get that the new M^w w-satisfies the consistency, the completeness, the minword-property and the minimality constraints.

The last thing to be shown is that M^w (obtained after the extension process) w-satisfies the separability constraint.

Suppose that in executing the repeat loop we have inserted i new states. For $0 \leq j \leq i - 1$, let s_{j+1} be the new state added while constructing $M_j^w(\lambda)$ from M_j^w. Obviously $Q^w = Q^x \cup \{s_1, \ldots s_i\}$.

We say that two states q, r are w-separable if $\exists y \in \Sigma^*$ such that *minword*(q)y $\neq_L$ *minword*(r)y, where *minword*(q)y $\leq$ w and *minword*(r)y $\leq$ w.

We prove by induction on j, $0 \leq j \leq i$, that each pair of states in $Q^x \cup \{s_k \mid k \leq j\}$ is w-separable. The basic assumption, for $j = 0$, is directly implied from the fact that M^x w-satisfies the separability constraint.

Assuming that each pair of states in $Q^x \cup \{s_k \mid k \leq j \text{ and } j < i\}$ is w-separable, we have to show that each state in this set is w-separable from s_{j+1}. Formally, let Q be the state set of M_j^w, we need to show that $(\forall q \in Q)$ $(\exists y \in \Sigma^*)$ (*minword*(q)y $\neq_L$ *minword*(s_{j+1})y and *minword*(q)y $\leq$ w and *minword*(s_{j+1})y $\leq$ w).

Let p, ϕ and z be the tested state, tested letter and tested tail for w in M_j^w, and w = *minword*(s_{j+1})z. Let q be an arbitrary state of M_j^w. We distinguish between the case where q was connected to p in M_j^w through the mutable link (p ϕ q), versus the case where they were not connected like this.

1) M_j^w has the mutable link (p ϕ q). By the corresponding execution of *delete-bad-paths*, this link will be deleted and replaced in $M_j^w(\lambda)$ by the new permanent link (p ϕ s_{j+1}). (p ϕ q) was deleted since all the paths for w = *minword*(s_{j+1})z on M_j^wwere wrong. By *Fact 5* applied on M_j^w, *minword*(q)z < *minword*(p)ϕz = w. By the consistency constraint of M_j^w, all the paths for *minword*(q)z in M_j^w are right. Hence, *minword*(s_{j+1})z $\neq_L$ *minword*(q)z, *minword*(s_{j+1})z = w and *minword*(q)z < w.
2) The link (p ϕ q) does not exist in M_j^w. There can be two different sub-cases under this condition. The first one is that (p ϕ q) had never been added while constructing one of the previous FSAs. The second sub-case is that (p ϕ q) has once been deleted. Note that in each of the previous FSAs, the links that leave p under ϕ are always mutable links, and $M_j^w(\lambda)$ will be the first FSA having a permanent link from p under ϕ. Hence, if (p ϕ q) has been once added, and thereafter deleted, the deletion was due to an execution of *delete-bad-paths*.
   2.1) (p ϕ q) had never been added. Since it is not the case that there exists a permanent link from p under ϕ (establishing a possible reason for not adding (p ϕ q)), it must be that *minword*(p)ϕ < *minword*(q).

   Now clearly there exists a mutable link in M_j^w from p under ϕ. Let t be a state such that (p ϕ t) exists in M_j^w. By the induction hypothesis, q and t (being two distinct states in M_j^w) are w-separable, hence $\exists y \in \Sigma^*$, such that *minword*(q)y $\leq$ w, *minword*(t)y $\leq$ w and *minword*(q)y $\neq_L$ *minword*(t)y. Figure 10 shows the relation between M_j^w and $M_j^w(\lambda)$ with respect to p, q and t.

   By the initial assumption for this subcase, *minword*(p)ϕ < *minword*(q). Thus, since *minword*(q)y $\leq$ w and *minword*(t)y $\leq$ w, we get that both *minword*(t)y and *minword*(p)ϕy are less than w. By the consistency constraint of M_j^w, all the paths for *minword*(p)ϕy, and all the paths for *minword*(t)y are right, and clearly *minword*(p)ϕy $=_L$ *minword*(t)y. Since y is separating between t and q, and

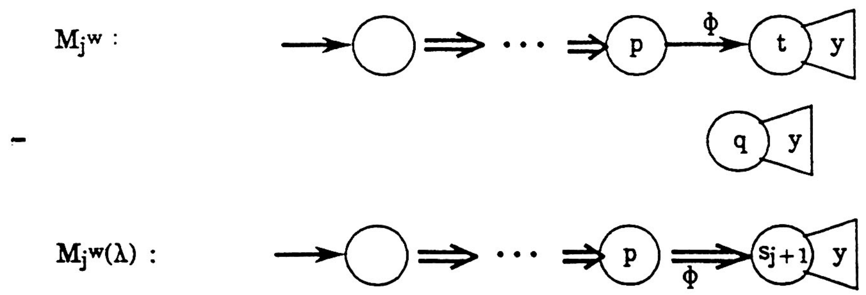


Figure 10. The case in which (p ϕ q) had never been added.

minword(p)ϕ = *minword*(s_{j+1}), we can conclude that y is a sufficiently small tail separating between s_{j+1} and q.

2.2) The next subcase deals with the mutable link (p ϕ q) being deleted due to some string v, $v \leq x$. (p ϕ q) has been the first mutable link along a wrong path for v on some previous FSA. Thus, there exists $y \in \Sigma^*$, such that v = *minword*(p)ϕy. The automata in which the decision to delete (p ϕ q) has been taken was obviously consistent with respect to *minword*(q)y, as *minword*(q)y < v by the minimality constraint which is continually satisfied. This establishes the claim that *minword*(p)ϕy $\neq_L$ *minword*(q)y. As *minword*(q)y < *minword*(p)ϕ y = *minword*(s_{j+1})y $\leq$ x, we get a perfect separating string for q and s_{j+1}.

This finishes the proof of the claim on the separability constraint.

Let R_L be the equivalence relation (Myhill-Nerode relation) associated with L, such that for x, $y \in \Sigma^*$, x R_L y iff $(\forall z \in \Sigma^*)(xz =_L yz)$. By Myhill-Nerode theorem, (Hopcroft & Ullman, 1979), $|Q_L|$ = number of equivalence classes of R_L.

By the separability constraint, $\forall x \in \Sigma^*$, $|Q^x| \leq |Q_L|$. Thus, there exists $x^* \in \Sigma^*$, such that $\forall y \geq x^*$ $|Q^y| = |Q^{x^*}|$ (after reaching M^{x^*} the extension process would never be applied again). Consider such an FSA, M^y where $y \geq x^*$. Suppose there is a state $q \in Q^y$ for which there are at least two distinct mutable links (q σ r) and (q σ t). By the separability constraint, $\exists z \in \Sigma^*$ such that *minword*(r)z $\neq_L$ *minword*(t)z. If both links still exist while considering the string *minword*(q)σz (by the consistency constraint, *minword*(q)σz > y) then, at this stage, one of them will definitely be deleted. Hence, eventually we will get a DFSA. Moreover, if M^x is a DFSA where $|Q^x| < |Q^L|$, then $\exists y > x$ such that the path for y in M^x is wrong. Therefore, we finally conclude that eventually we will get a minimum-state DFSA isomorphic to M_L. This completes the correctness proof.

Having proven the correctness of the learning algorithm, we now consider some properties of this process, mainly with respect to time and space complexity.

3.4. Complexity analysis

Given a current guess M^x, the value *minword*(q) for every $q \in Q^x$ and the successor sample string ±w, we analyze first the time complexity. Let $|Q^x| = n$, $|w| = m$, and $|\Sigma| = a$.

Obviously, the size of the whole sample considered so far λ, succ(λ), succ(succ(λ)), . . . , w is exponential in m (greater than a^{m-1}).

First we note that executing *delete-bad-paths* on M^x and w requires only polynomial time with respect to the size of M^x and w. Notice that the algorithm does not need to check *every* path of w in M^x, but rather to consider one path for each first mutable link along paths for w in M^x. This is due to the observation we have already made that if (p ϕ q) is some first mutable link for w in M^x, then all the continuing paths from q behave the same. Each path for w can be checked in polynomial time, and there can be at most n different paths to be checked. Moreover, note that within this process, that must be applied for each string, the tested state and tested letter can be recorded, and this might later be used.

If *insert-state* is activated, then it can be done in polynomial time, gaining some efficiency by using the recorded tested state and tested letter. The dominating step is the addition of all possible new mutable links. This obviously involves considering such existing state q (and its *minword*), and each letter σ. Since, the length of w is m, *insert-state* can be repeatedly activated at most m times.

The retesting while loop is repeated for each string y, succ(λ) $\leq$ y $\leq$ x. For each such y, the condition that determines the value of the boolean variable accept can again be checked in polynomial time. Note that, due to the consistency constraint, only *one* path for y in the current old-M has to be checked to determine the behavior of the old machine with respect to y. The total retesting process (reaching the case for which succ(y) = w) can take exponential time in the size of the current example w, since all previous strings are checked, but it is still polynomial in the size of the whole sample considered so far. Such retesting happens very infrequently; in fact it is invoked once per state insertion. Therefore, taking into account the whole sample considered so far, the *amortized* cost of the retesting process is polynomial in the size of the current input. Finally, we conclude that the whole amount of time needed for the learning algorithm is polynomial in n in an amortized sense.

Furthermore, the extension process can be somewhat improved. First, in *insert-state*, we change the rule that adds a new mutable link from an old state q to the new state s under some letter σ. The new rule states that we add (q σ s) only when two conditions are met. The first condition is the original one, namely that the current links from q under σ are all mutable, and *minword*(q)σ > *minword*(s). The second condition is that either *minword*(q)σ $\geq$ w or *minword*(q)σ < w (so that the current parity of the pair (q, σ) is defined) and s is of the correct parity, i.e., f(s) = f(q, σ) in the current M_j^w. This rule is obviously correct, since if all the current links from q under σ are mutable, and *minword*(s) < *minword*(q)σ (so that (q σ s) would have been added under the previous rule), and moreover *minword*(q)σ < w and f(s) $\neq$ f(q, σ) (so that (q σ s) would not have been added under the new rule), then the retesting process will definitely prune this new mutable link while considering y = *minword*(q)σ. Hence, omitting those links immediately in *insert-state* might achieve some efficiency in the inspection of all possible paths for some y within the while loop.

A more significant improvement is due to the fact that within the retest process (the while loop) only new mutable links (to and from the new state) might be deleted as being first muable links along wrong paths for some y. Consequently, we need only check the following subset of the sample. For each state q and symbol σ, such that a new mutable link from q under σ has just been added within the last execution of *insert-state*, test the strings *minword*(q)σz that are smaller than w. Note that the new rule for adding new mutable links has now a more considerable impact on the performance, by omitting a whole set of strings from being checked. There are still n * $a^{|z|}$ such strings, but $|z|$ will usually be small and there even might be states for which no string will need to be tested.

As stated in the introduction, one of the major goals that motivated this work was to design an algorithm for learning an FSA that will require only a modest amount of memory, much less than the sample size which is exponential in m. Clearly, the storage needed for the current guess is proportional to $n^2 * a$, and the storage needed for all the values *minword*(q), for each state q, is proportional to n. The easiest way to envision the learning algorithm is to imagine that it uses two separate representations—one for new-M, and the other for old-M. Taken literally, this would double the size of the storage needed for the current guess. A more efficient solution is to indicate some links on the current guess as old ones, and thus analyze both machines on "one" representation. Another improvement might be gained (with respect to the amount of storage needed for the algorithm) by modifying the insertion process so as to avoid the need for storing *minword*(q) for each q. These values might as well be computed for every new state by linearly traversing the prefix-coded tree of the permanent links.

Let $|Q_L| = n_L$. As shown above, the current size n of the machine is at most n_L. Any two distinct states in M_L are obviously separable. Moreover, it can be shown that the shortest string that distinguishes between such pairs of states is at most of length n_L. Since each *minword*(q) is at most of length n_L, we can conclude that the maximum string after which the current guess is isomorphic to M_L is of length linear in n_L. In order to prove this, consider a guess M^x for which $|Q^x| = n_L$ and $|x| > 2 * n_L + 1$, and suppose (by way of contradiction) that M^x is not isomorphic to M_L. In other words M^x is non-deterministic, hence there exists a state p, such that there are at least two distinct mutable links from p under some letter ϕ. Let t and r be those states having incoming mutable links from p under ϕ. As indicated above, $|minword(p)| \leq n_L$, and there is a string of length at most n_L that separates between t and r. Hence, M^x admits two different paths for a word of length at most $2 * n_L + 1$, one that leads to an accepting state, and the other to a rejecting one. This obviously contradicts the consistency assumption of M^x. In summary, our algorithm could get by with space proportional to $a * n_L^2$ to store its guess plus m for the current string. This corresponds to the abstract notion of iterative learning of Wiehagen (1976).

4. Iterative learning using finite working storage

In this section we will formally characterize some "practical" properties of the learning algorithm introduced in Section 3. Taking into account the limitation of space in all realistic computations, the most important property of our algorithm is that for every sample, given in a lexicographic order, the algorithm uses a finite amount of space. The restriction on the strict order of the sample may seem to be too severe. We will show in this section that this restriction is necessary for learning with finite memory.

Our first definition follows the approach of Wiehagen (1976); Jantke & Beick (1981); and Osherson (1986):

Definition: An algorithm IT (iteratively) identifies a set of languages S iff for every $L \in S$, given the complete lexicographic sample $\langle \pm\lambda, \pm\text{succ}(\lambda), \pm\text{succ}(\text{succ}(\lambda)) \ldots \rangle$, the algorithm defines a sequence of finite machines $\langle M_0, M_1, M_2, \ldots \rangle$, such that $\forall i > 1$ M_i is obtained from the pair $(M_{i-1}, \pm x_i)$, where x_i is the i-th string in the sample, and $\exists j$ such that $\forall k \geq j$ $M_k = M_j$ and $L(M_j)$ (the language accepted by the machine M_j) is L.

It is easy to see that the algorithm of Section 3 meets the requirements of the IT definition. In other words, we exhibit an algorithm that IT identifies the FSL. Moreover, it finds a minimum-state DFSA for the inferred language L.

We now define a weaker characterization of a learning algorithm. We allow finite working storage in addition to that required for defining the current guess and sample string. The ultimate goal will be to show that the restriction on the order of the sample is necessary even for this kind of algorithm.

Definition: An algorithm FS-IT (iterative algorithm that uses finite storage) identifies a set of languages S iff for every $L \in S$, given the complete lexicographic sample $\langle \pm\lambda, \pm\text{succ}(\lambda), \pm\text{succ}(\text{succ}(\lambda)), \ldots \rangle$, there exists a *finite* set of states Q, such that the algorithm defines a sequence of configurations $\langle (M_0, q_0), (M_1, q_1), (M_2, q_2), \ldots \rangle$ that satisfies the following: $\forall i$, $q_i \in Q$, $\forall i > 1$, (M_i, q_i) is obtained from the triple $(M_{i-1}, q_{i-1}, \pm x_i)$, where x_i is the i-th string in the sample, and $\exists j$ such that $\forall k \geq j$ $M_k = M_j$ and $L(M_j) = L$. Such a j (in the course of learning) is referred to as a "semi-stabilization" point. Note that the states within the configurations after the semi-stabilization point can still change.

Obviously, if an algorithm IT identifies S, then it also FS-IT identifies this set.

A (non-repetitive) complete sample for a language L is a sequence of its strings $\langle \pm x_1, \pm x_2, \pm x_3, \ldots \rangle$ such that $\forall i$, $x_i \in L$, $\forall i \neq j$, $x_i = x_j$, and $\forall x \in L$ $\exists i$ such that $x = x_i$. The ability to learn languages by presenting an *arbitrary* complete sample, rather than the strict lexicographic one, obviously strengthens the characterization of the learner. We denote the above situations by IT^{arb} and $FS\text{-}IT^{arb}$ if we do not require the sample to be in lexicographic order.

For a complete sample $\langle \pm x_1, \pm x_2, \pm x_3, \ldots \rangle$, an algorithm that $FS\text{-}IT^{arb}$ identifies a set of language S uses a finite set of states Q, and defines a sequence $\langle (M_0, q_0), (M_1, q_1), \ldots \rangle$. M_i and $q_i \in Q$ are referred to as the *current guess* and the *current state* after the finite prefix $\langle \pm x_1, \ldots, \pm x_i \rangle$, $\forall i \geq 0$.

Theorem. There is no algorithm that $FS\text{-}IT^{arb}$ identifies the finite state languages.

Proof. Suppose, to the contrary, that an algorithm A $FS\text{-}IT^{arb}$ identifies the FSLs. We will look for some FSLs that will lead to a contradiction.

Let L_0 be Σ^*, for some alphabet Σ.

For the lexicographically ordered sample for L_0, A defines the sequence $\langle (M_0^0, q_0^0), (M_1^0, q_1^0), \ldots \rangle$. By the definition of $FS\text{-}IT^{arb}$ and in particular due to the finiteness of the working storage, there exists some semi-stabilization point, i, that corresponds to some word x, such that the following two conditions are satisfied:

1) The current guess M^0_m, $\forall m > i$, is the same as M^0_i and characterizes L_0. Call this guess M_{L_0}.
2) There are infinitely many m's, $m \geq i$, such that $q^0_m = q^0_i$ (i.e., the state q^0_i occurs infinitely often).

Let $L_1 = \{w \in \Sigma^* \mid w \leq x\}$.

For the lexicographically ordered sample for L_1 $\langle +\lambda, \ldots, +x, -\mathrm{succ}(x), -\ldots \rangle$, A defines the sequence $\langle (M^1_0, q^1_0), (M^1_1, q^1_1), \ldots, (M^1_i, q^1_i), (M^1_{i+1}, q^1_{i+1}), \ldots \rangle$. Obviously, $\forall m, 0 \leq m \leq i, M^0_m = M^1_m$ and $q^0_m = q^1_m$. In particular $M^1_i = M_{L_0}$ and $q^1_i = q^0_i$. By the infinitely repeating property of q^0_i, and by the finiteness condition on the set $\{q^1_m \mid m \geq 0\}$, q^0_i must coincide with some q^1_m infinitely often. In other words, there exists a j that corresponds to some word z, $j > i$, such that the following three conditions are satisfied:

1) $q^0_j = q^0_i$.
2) $\forall m > j$ $M^1_m = M^1_j$ and characterizes L_1. Call this guess M_{L_1}.
3) There are infinitely many m's, $m > j$, such that $q^0_m = q^0_i$ and $q^1_m = q^1_j$ (the pair of states q^0_i, q^1_j appears infinitely many times at the same points for the strict ordered samples for L_0 and L_1).

Pick a place k, $k > j$, such that $q^0_k = q^0_i$ and $q^1_k = q^1_j$. The existence of such a place is established by the properties of the chosen j. Let y be the string at place k in the lexicographically ordered sample of any language over Σ. Note that $y > z$.

Let $L_2 = L_1 \cup \{w \in \Sigma^* \mid z < w \leq y\}$.

For the following ordered sample $\langle +\lambda, \ldots, +x, +\mathrm{succ}(z), \ldots, +y, -\mathrm{succ}(x), \ldots, -z, -\mathrm{succ}(y), -\mathrm{succ}(\mathrm{succ}(y)), \ldots \rangle$, A defines the sequence $\langle (M^2_0, q^2_0), (M^2_1, q^2_1), \ldots \rangle$. From the above we get (see Figure 11):

1) After the finite prefix $\langle +\lambda, \ldots, +x \rangle$ the current guess is M_{L_0} and the state is q^0_i.
2) By the definition of z and y, after the finite prefix $\langle +\lambda, \ldots, +x, +\mathrm{succ}(z), \ldots +y \rangle$ the current guess is still M_{L_0}, and the state is again q^0_i.
3) By the definition of z with respect to its occurrence in the strict ordered sample for L_1, after the finite prefix $\langle +\lambda, \ldots, +x, +\mathrm{succ}(z), \ldots, +y, -\mathrm{succ}(x), \ldots, -z \rangle$ the current guess M^2_k is M_{L_1}, and $q^2_k = q^1_j$.
4) By the property of q^1_j, $\forall m > k$, $M^2_m = M_{L_1}$, and $q^2_m = q^1_m$.

Hence A cannot FS-IT$^{\mathrm{arb}}$ identify L_2.

Thus we have shown formally that the FSL can be IT-identified (from lexicographically ordered samples) but cannot be IT$^{\mathrm{arb}}$-identified. The theorems provide end-case results, but there is a wide range of possible presentation disciplines between IT and IT$^{\mathrm{arb}}$. Obviously enough, our algorithm will still identify the FSL from presntations in which redundant strings happen to be missing. That is, any w such that $w = \mathrm{succ}(x)$ and $M^w = M^x$ could be missing from the sample without effect. As stated above, for any FSL for which $|Q_L| = n$, namely an n-FSL, all the strings longer than $2 * n + 1$ will be redundant. Moreover, most strings of length at most $2 * n + 1$ could be missing from the sample.

$$L_0: \quad +\lambda, \ldots, \underset{M_{L_0}, q_i^0}{+x}, \quad \ldots, \quad \underset{M_{L_0}, q_i^0}{+z}, \quad +\mathrm{succ}(z), \ldots, \underset{M_{L_0}, q_i^0}{+y}, \quad \ldots$$

$$L_1: \quad +\lambda, \ldots, \underset{M_{L_0}, q_i^0}{+x}, \quad -\mathrm{succ}(x), \ldots, \underset{M_{L_1}, q_j^1}{-z}, \quad -\mathrm{succ}(z), \ldots, \quad \underset{M_{L_1}, q_j^1}{-y},$$
$$\underset{M_{L_1}, q_{k+1}^1}{-\mathrm{succ}(y)}, \quad \underset{M_{L_1}, q_{k+2}^1}{-\mathrm{succ}(\mathrm{succ}(y))}, \quad \ldots$$

$$L_2: \quad +\lambda, \ldots, \underset{M_{L_0}, q_i^0}{+x}, \quad +\mathrm{succ}(z), \ldots \underset{M_{L_0}, q_i^0}{+y}, \quad -\mathrm{succ}(x), \ldots, \underset{M_{L_1}, q_j^1}{-z},$$
$$\underset{M_{L_1}, q_{k+1}^1}{-\mathrm{succ}(y)}, \quad \underset{M_{L_1}, q_{k+2}^1}{-\mathrm{succ}(\mathrm{succ}(y))}, \quad \ldots$$

Figure 11. Proving that A cannot FS-IT^{arb} identify L_2.

This follows because there are only $a * n^2$ links to be added or deleted and more than a^{2*n+1} strings of length $\leq 2 * n + 1$. Therefore a teacher could, in principle, get by with a greatly reduced presentation if she knew what to present. The same reduction could also be used in the retest phase of our algorithm.

We conjectured that perhaps for an ordered presentation of some particular subset, followed by an arbitrary sequence of the remaining strings, our algorithm would be still applicable.

Let S_L be the set $\{minword(q) \mid q \in Q_L\}$, and let y be the maximal string in this set. We easily found a counterexample that shows that the subset $\{x \in \Sigma^* \mid x \leq y\}$ does not suffice, and it is not even the case that Q^y—the set of states in the last guess M^y—is the same as Q_L. We then examined a larger set. Let $S_{L'}$ be the set $\{x \mid (\exists p, q \in Q_L)\ (x = minword(p)z$ and $x \not\equiv_L minword(q)z$ and $(\forall z' \in \Sigma^*)\ (minword(p)z' \not\equiv_L minword(q)z' \Rightarrow x \leq minword(p)z'))\}$. The intuition behind this set is to include all the least strings that can distinguish between two distinct states in M_L. First we observe that $S_L \subseteq S_{L'}$. Consider some state p in Q_L. There must be some q in Q_L the parity of which differs from that of p, hence $minword(p) \not\equiv_L minword(q)$. Obviously, for every $z' \in \Sigma^*$, $minword(p) \leq minword(p)z'$. Hence, $minword(p) \in S_{L'}$. Now, let y′ be the maximal word in $S_{L'}$. The following counterexample shows that even the subset $\{x \in \Sigma^* \mid x \leq y'\}$ does not suffice. Let L be the language (given as an example in Section 3): "the number of *a*'s is at most 1, and number of *b*'s is at least 1." $S_{L'} = \{\lambda, a, b, aa, ab, ba, aab, aba\}$. After the finite prefix $\langle \lambda, -a, +b, \ldots, -aba \rangle$, M^{aba} is as shown in Figure 12.

Obviously, $M^{aba} \neq M_L$. Moreover, let $-bbaba$ be the next sample string. The rule for breaking all wrong paths by deleting the *first* mutable links along them cannot be applied in this case. The first mutable link along the bad path $\langle q_0\ b\ q_1, q_1\ b\ q_1, q_1\ a\ q_1, q_1\ b\ q_1, q_1\ a\ q_1 \rangle$ is the link $\langle q_1\ b\ q_1 \rangle$ that should remain in M_L. In fact, every other possible rule

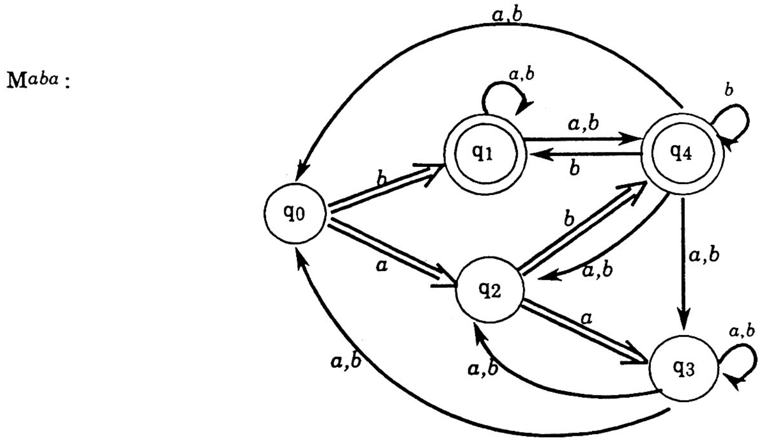


Figure 12. Applying the algorithm on an ordered subset.

that determines the mutable link to be deleted, according to its place in the bad path, would not work here. For every i = 1, 2, 3, 4, there is a bad path whose i-th mutable link exists in M_L.

It is an open question whether any characterization of the minimum training set exists.

It is also easy to see intuitively why finite storage learners will fail on arbitrary presentations. An arbitrary presentation can, for example, have only very long strings for a very long time and the learner has no idea what to make of them. This is the basic cause of the NP-completeness results of Gold (1978) and Angluin (1978) for minimal DFSA learning. On the other hand, if the learning device knew in advance the size, n, of the n-FSL, it might be able to collapse the long sample strings into equivalence classes. This is another open question. The realistic version of this is for the learner, which has finite storage, bounded by some polynomial in n, to limit its guesses to M with no more than n states.

There do seem to be some general consequences of the outcome of these open questions. If, as we surmise, knowing a bound on n for the target n-FSL does not permit FS-IT^{arb} with the finite storage being bounded by a polynomial in n, then learning simple examples first has inherent major advantages. If there are optimal training presentations, it will be interesting to understand their nature. As we will show in the next section, the algorithm of Section 3.1 works in a way that is compatible with connectionist and thus (at least for some people) with neural computation.

5. Distributed and connectionist versions

5.1. Distributed realization

As we discussed in the introduction, there is no generally accepted formalization of what precisely constitutes a connectionist model. In this section we show how the algorithm of Section 3.2 can be translated into a network of simple computing units that falls within the range of connectionist models. In particular, the network involves only simple units that broadcast very simple outputs on all their outgoing links. Learning is realized by local weight change that restructures the network. There is, of course, no interpreter, but there is some central control. There are several places in the construction where system-wide parallelism holds, but there are also sequential aspects that seem to be inherent. For any finite system to recognize unbounded inputs, it will have to look at pieces of the input sequentially. Also in the retest phase of our algorithm, it is necessary to test individual samples sequentially.

The conceptual distance from the algorithms and analysis above to the sketched connectionist version is considerable and we will traverse it in two steps. We will start with a realization in terms of module-message model, like PLITS or CSP. Each state of the target FSA will be represented by a module and there will be a control module and several other fixed modules.

Each state-module, q, will have data structures for its activation state, its parity, its minimal string *minword*(q), and its outgoing links to other states and other modules. We suppose that the system is synchronous and that the control module broadcasts each letter of the input string, w, at the start of a major cycle. The first benefit of the parallel implementation is that all paths for the target string can be checked in parallel. Initially, q_0 is active. Each active module looks at the next symbol, σ, and sends an "activation" message along each of its outgoing links that correspond to σ. When these signals have been sent, any state that has not received new activation inactivates itself, and the states that have received such signals are active for the next cycle. There are three kinds of activation messages that are sent by states along paths of w. Recall the notation p and ϕ for the tested state and tested letter of w on the current machine. The first kind of an activation message is sent by those states along the basic path for *minword*(p). This message indicates that no mutable link has yet occurred. Let p be the first state to use mutable outgoing link, p sends a message that encodes its identity, p, and that of the tested letter, ϕ. Each active module q that receives a message of the second kind, encoding the pair (p, ϕ), looks at the next symbol σ, and sends an activation message of the third kind along each of its outgoing links that correspond to σ. This message encodes the triple (p ϕ q), indicating a specific first mutable link along a path for w. All successor states that receive this kind of message send that same message. Note that each path is represented by its first mutable link, encoded within the message that passes through the corresponding suffix.

At the end of the string, marked by a terminator $\vdash$ or by a control signal, the states that are active report their corresponding first mutable links plus their parity $\pm$. This could be reported directly to the controller or (more connectionist) by sending activation to global variables (or modules) that represent good and bad strings. The control now compares the provided answer and if all reports are right, it goes on to the next string, as before.

If some parses are right and some are wrong, a deletion process corresponding to the subroutine *delete-bad-paths* must be executed. As mentioned and proved in Section 3, each of the first mutable links along paths for w corresponds either to a set of right paths, or to a set of wrong paths. The controller can obviously identify all paths by their first mutable links, and knows which ones were right or wrong from the reported parity. It then composes a message to all the "bad" mutable links, and sends this information. Each module that is the origin of a bad mutable link will then delete the corresponding outgoing link, thus breaking a bad path for w.

Finally, we need to model the state-addition and retest procedures. Clearly the control can easily discover if all paths are wrong. A new state is "recruited" by generating a new module, s, and initializing it with its parity, its path, *minword*(s) and with its permanent link. All this information can be easily generated using the data structure within the module that corresponds to the tested state p, namely the one that identifies itself as the source of all the bad mutable links to be deleted at that stage. With this information the new state/module can establish links with old states following the strictures of *insert-state*. Again, this can be done in parallel except for the serialization within module s itself. The retest procedure is sequential, the controller cycles through the required strings and tests them against the old machine. The difference between the links of the old and new machines are also part of the data structures of the appropriate modules. Of course, within each string test, the parallel checking and deletion above still apply. Much of this will carry over to the connectionist version, but there are also several differences.

5.2. Connectionist realization

Connectionist models in the literature vary somewhat (Rumelhart & McClelland, 1986; Waltz & Feldman, 1986) but all are restricted to simple units that pass only numerical messages and always send the same number on each outgoing link. The links may have weights that modify the value being received and many models also allow conjunctive connections like we used in Figure 2. Rochester practice allows for a unit to have a small amount of internal data and to be in one of a small number of different "states" which we will denote here as "modes" to reduce confusion. The limited repertoire of connectionist systems forces the use of more elaborate structures than the previous version. We will present an outline of one such model.

The connectionist version of the FSA learner will have a control subnetwork that will sequence and modulate the basic learning net. There will be "registers," banks of units whose activation pattern represents a letter or a string of letters, like the top row of Figure 2. The basic process of testing a string against the current guess works as outlined in Figure 2. Each letter of the input string serves in turn as the gate on the conjunctive connections from state-unit to state-unit. The conjunction of activation of the prior state and the appropriate letter-unit leads to activation of the next state-units along appropriate links. One way to have the state-units turn off when they should is to have a just-sent mode. A unit in just-sent mode will inactivate itself (set its activation flag to zero) if it receives only a control signal for the next cycle.

A somewhat similar mechanism can be used to mark the first mutable link along each path. Suppose that permanent links have weight 1 and mutable links weight ½. Let the activation rule for a unit be as follows: The initial state q_0 always sends activation value 10 to start each string. If a subsequent state-unit sees an input value of 5, it knows that it is at the far end of the first mutable link in a path, and will record which input link was active. It will also send out a lower value, say 4. Units that receive either 2 or 4 will also send out 4 as a value. This effectively marks the receiving end of the first mutable link in every path, with a tagged input in the unit for that state marking the path.

Upon termination of the testing for some string, w, the global + and − units are compared by control with the answer provided (as activation of another Winner-Take-All pair). If all paths are right, then the next string is tried. If there are both right and wrong paths, the deletion process must occur. There is no obvious way to do this in parallel, but the following sequential scheme works. Assume that the mechanism includes a "buffer" that can record the input string w and another buffer that can be made to cycle through strings. Control recycles the input string (in delete mode) until the unit having the first mutable links is encountered. That is, a state-unit that is activated in delete mode and has its tag set sends a different signal which is detected by the control net. Then each such state is tested sequentially and the ones leading to wrong answers delete their corresponding incoming mutable link. Deletion can be just setting the weight to zero. There needs to be some mechanism for sequencing these states, e.g., enabling each state in sequential order.

The insertion process for the connectionist version involves even more technical details. It is reasonable to assume that the learning net has unused state-units that are connected to all the ones used thus far, one of these is "recruited" to be the new state s. It is not hard to determine which link to s should be permanent, it is the one from the state p with the first mutable links in the (wrong) parses of w = *minword*(p)ϕz. The string w could be reparsed and the output of 5 from the states that receive conjoined signals from p and ϕ could inform the new state, s, that it should set the weight of its active input link to be 1 (permanent). It is also reasonable to assume that the state-units can mark the current links from p under ϕ as "old," and thus to be used only in retest mode.

What we cannot assume is that state-units, q, can store the minimal string *minword*(q) and compare it with *minword*(s) to determine which (q σ s) and (s σ q) links should be added. Again, the apparent answer is to go sequential. We can assume that the control net has buffers for *minword*(s) and *minword*(q), *minword*(s) is fixed for the addition process, but *minword*(q) cycles through all the other existing states. The control net finds *minword*(q) by testing strings, and with this in the buffer, the tests *minword*(q)σ > *minword*(s) and *minword*(s)σ > *minword*(q) can be carried out by the control net. The signal to break the appropriate links can be transmitted to the state-units involved. This leaves just the retesting process. The obvious way to handle this is to have the control net cycle through every string y < w and test and correct the current guess. The basic process of testing a string and the deletion process work as before. Each unit needs to have an "old machine" and "new machine" mode and to know which links go with each. Each string less than w is tested in old machine mode and the answer is stored. Then the same string is tested in new machine mode and the deletion process is invoked for all wrong paths.

In this design, each unit would need internal data for recording its number, whether it is active, has an active first mutable link and is performing as the old or new machine. It would need "modes" for just-sent, for normal testing, for deletion, for recruiting and for pruning links. If we restrict ourselves to just state-less linear threshold elements, the complexity expands by an order of magnitude.

Yifat Weissberg's thesis project (Weissberg, 1990) describes an implementation of the learning algorithm in a connectionist network, using the Rochester Connectionist Simulator. Learning is achieved by weight changes. The rules by which the links change their weight are simple and local. Nevertheless, the whole implementation is quite complex, basically due to the net representation. The network consists of two parts, one represents an automaton, the other represents an example string. A special control unit connects these two parts. Every unit has a considerable number of sites (at which the incoming links arrive), so as to distinguish between different stages of the learning process. The basic algorithm has been slightly modified, to fit the connectionist style. The major changes are in the process of adding a new state to the current automaton. The thesis includes proofs for the correctness of the modified algorithm.

Since the details of each parse play an important part in the learning procedure, there are at least indirect connections with explanation-based learning. But the case of learning from a perfect, lexicographically ordered sample is a very special one. It is well worth exploring how the algorithms of this paper could be modified to deal with less controlled examples. An obvious approach is to change the delete process (of mutable links) to one that reduces weights to some non-zero value, perhaps halving them each time. The question of how to revise the rest of the network's operation to properly treat conflicting evidence is another topic worthy of further effort.

Acknowledgments

We thank John Case for providing useful comments on the manuscript.

This work was supported in part by ONR/DARPA Research contract No. N00014-82-K-0193 and in part by ONR Research contract no. N00014-84-K-0655.

References

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

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

Angluin, D. (1978). On the complexity of minimum inference of regular sets. *Information and Control*, *39*, 337–350.

Angluin, D. (1976). *An application of the theory of computational complexity to the study of inductive inference.* Ph.D. dissertation, Department of Electrical Engineering & Computer Science, Univ. California, Berkeley.

Angluin, D., & Smith, C.H. (1983). Inductive inference: Theory and methods. *Computing Surveys*, *15*, 237–269.

Biermann, A.W., & Feldman, J.A. (1978). On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. on Computers*, *C-21*, 592–597.

Brooks, R.A. (1987). Intelligence without representation. *Proceedings of the Conf. on Foundations of AI*. Cambridge, MA: MIT.

Feldman, J.A., & Ballard, D.H. (1982). Connectionist models and their properties. *Cognitive Science*, *6*, 205–254.

Gold, E.M. (1978). Complexity of automaton identification from given data. *Information and Control*, *37*, 302–320.

Gold, E.M. (1972). System identification via state characterization. *Automatica*, *8*, 621–636.

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

Hinton, G.E. (1987). *Connectionist learning procedures* (TR CMU-CS-87-115). Pittsburgh, PA: Carnegie Mellon University, Computer Science Department.

Hopcroft, J.E., & Ullman, J.D. (1979). *Introduction to automata and formal languages*. Reading, MA: Addison-Wesley.

Horning, J.K. (1969). *A study of grammatical inference*. Ph.D. thesis, Stanford University.

Ibarra, O.H., & Jiang, T. (1988). Learning regular languages from counterexamples. *Proceedings of the 1988 Workshop on Computational Learning Theory* (pp. 371–385). Boston, MA.

Jantke, K.P., & Beick, H-R. (1981). Combining postulates of naturalness in inductive inference. *Journal of Information Processing and Cybernetics*, *17*, 465–484.

Kearns, M., Li, J., Pitt, L., & Valiant, L. (1987). On the learnability of boolean formulae. *Proceedings of the 9th Annual ACM Symp. on Theory of Computing* (pp. 284–295). New York, NY.

Natarajan, B.K. (1987). On learning boolean functions. *Proceedings of the 9th Annual ACM Symp. on Theory of Computing* (pp. 296–394). New York, NY.

Osherson, D.N., Stob, M., & Weinstein, S. (1986). *Systems that learn: An introduction to learning theory for conginitive and computer scientists.* Cambridge, MA: MIT Press.

Rivest, R.L., & Schapire, R.E. (1987). A new approach to unsupervised learning in deterministic environments. *Proceedings of the 4th International Workshop on Machine Learning.* Irvine, CA.

Rivest, R.L., & Schapire, R.E. (1987). Diversity-based inference of finite automata. *Proceedings of the 28th Annual Symp. on Foundations of Computer Science.* Los Angeles, CA.

Rumelhart, D.E., & McClelland, J.L. (Eds.). (1986). *Parallel distributed processing, explorations in the microstructure of cognition.* Cambridge, MA: Bradford Books/MIT Press.

Sharon, M. (1990). *Learning automata.* M.Sc. Thesis in Computer Science, Technion, Haifa, Israel, (in Hebrew).

Trakhtenbrot, B.A., & Barzdin, Ya.M. (1973). *Finite automata.* Amsterdam: North-Holland.

Valiant, L.G. (1985). Learning disjunctions of conjunctions. *Proceedings of the 9th IJCAI* (pp. 560–566). Los Angeles, CA.

Valiant, L.G. (1984). A theory of the learnable. *CACM, 27,* 1134–1142.

Waltz, D., & Feldman, J.A. (Eds.). (1987). *Connectionist models and their implications.* Ablex Publishing Corp.

Weisberg, Y. (1990). *Iterative learning finite automata—Application by neural net.* M.Sc. thesis in Electrical Engineering, Technion, Haifa, Israel, (in Hebrew).

Wiehagen, R. (1976). Limeserkennung rekursiver funktionen durch spezielle strategien. *Elektronische Informationsverarbeitung und Kybernetik, 12,* 93–99.

Williams, R.J. (1987). *Reinforcement-learning connectionist systems* (TR NU-CCS-87-3).