Efficient Search Techniques for the Inference of Minimum Sized Finite State Machines

Arlindo L. Oliveira and João P. M. Silva
Cadence European Laboratories
IST-INESC
Lisbon, Portugal
{amljpms}@inesc.pt

Abstract

We propose a new algorithm for the inference of minimum size finite state machines from training set data. Our approach is based on a well known search algorithm proposed by Bierman, but it incorporates a set of techniques known as dependency directed backtracking. Although these techniques have already been used in other applications, we believe we are the first to apply them to this problem. The results show that the application of these techniques yields an algorithm that is, for the problems studied, orders of magnitude faster than existing approaches.

Introduction and Related Work

This work addresses the problem of inferring a finite state machine with minimal complexity that matches a given set of input/output pairs. This problem has been extensively studied in the literature, both from a practical and theoretical point of view. If the output alphabet is the set \{0, 1\}, there exists a trivial transformation between the finite state machine that satisfies a training set specified in this form and the minimal deterministic finite automaton (DFA) that accepts all strings that generate a 1 as output and rejects all strings that generate 0. Therefore, all the previous work done using the DFA formalism is relevant and closely related to the work discussed here.

The problem of selecting the minimum DFA consistent with a set of labeled strings is known to be NP-complete. Specifically, Gold (Go78) proved that given a finite alphabet \(\Sigma\), two finite subsets \(S, T \subseteq \Sigma^*\) and an integer \(k\), determining if there is a \(k\)-state DFA that recognizes \(L\) such that \(S \subseteq L\) and \(T \subseteq \Sigma^* - L\) is NP-complete. Furthermore, it is known that even finding a DFA with a number of states polynomial on the number of states of the minimum solution is NP-complete (PW93).

If all strings of length \(n\) or less are given (a uniform-complete sample), then the problem can be solved in time polynomial on the input size (GH66; PF88; TB73). Note, however, that the size of the input is in itself exponential on the number of states in the resulting DFA. Angluin has shown that even if an arbitrarily small fixed fraction \(\left(\left[\Sigma^{*}\right]\right)^{e}, \epsilon > 0\) is missing, the problem remains NP-complete (Ang78).

The problem becomes easier if the algorithm is allowed to make queries or experiment with the unknown machine. Angluin (Ang87) proposes an algorithm based on the approach described by Gold (Go72) that solves the problem in polynomial time by allowing the algorithm to ask membership queries. Schapire (Sch92) proposes an interesting approach that does not require the availability of a reset signal to take the machine to a known state.

All these algorithms address simpler versions of the problem discussed here. We assume the learner is given a set of labeled strings and is not allowed to make queries or experiment with the machine. The basic search algorithm for this problem was proposed by Bierman (BF72). Later, the same author proposed an improved search strategy that is much more efficient in the majority of the complex problems (BP75). Section describes these algorithms in detail. More recently, the applicability of implicit enumeration techniques to this problem was studied (OE96), but these techniques have not shown any significant advantages when compared with the basic search techniques proposed by Bierman.

The central contribution of this paper is the application of advanced search techniques to the inference problem stated above. More specifically, we show how the application of dependency-directed backtracking techniques improves significantly the search algorithm proposed by Bierman. These techniques, applied to date in other domains like truth maintenance systems and boolean satisfiability solvers (Si88; SS77), allow the search algorithm to prune large sections of the search tree by diagnosing the ultimate causes of conflicts encountered during the search. In many cases, these conflicts are caused by assignments that were made several levels above and significant parts of the search tree can be removed from consideration.

These techniques were implemented in a program, BICA. The approach was evaluated by comparing this program with alternative approaches in sets of prob-
lems with known solutions. The results presented in section show that these techniques are indeed effective in extending the scope of applicability of the search techniques and make the algorithms able to handle many problems of non-trivial size.

Several connectionist approaches have been proposed for the problem of recognizing temporal sequences (DM93; GMC †92; Pol91; WK92) and they are directly applicable to this problem. However, the results presented show that these approaches are not competitive if the system to be induced can be adequately modeled by a finite state machine.

A different approach is to view the problem of selecting the minimum automaton consistent with a set of strings as equivalent to the problem of reducing an incompletely specified finite state machine (ISFSM). This problem is more general than the one addressed here and was also proved to be NP-complete by Pfleeger (Pf73), but previous work (OE96) has shown that these algorithms are extremely inefficient when applied to this problem, and present no advantages to the approach presented here.

**Definitions**

**Finite State Machines**

We use the standard definition of finite state machines:

**Definition 1** A finite state machine is a tuple $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ where $\Sigma \neq \emptyset$ is a finite set of input symbols, $\Delta \neq \emptyset$ is a finite set of output symbols, $Q \neq \emptyset$ is a finite set of states, $q_0 \in Q$ is the initial "reset" state, $\delta(q, a) : Q \times \Sigma \rightarrow Q \cup \emptyset$ is the transition function, and $\lambda(q, a) : Q \times \Sigma \rightarrow \Delta \cup \{\varepsilon\}$ is the output function.

We will assume that $Q = \{q_0, q_1, \ldots, q_n\}$ and use $q \in Q$ to denote a particular state, $a \in \Sigma$ a particular input symbol and $b \in \Delta$ a particular output symbol. A finite state machine is incompletely specified if the destination or the output of some transition is not specified. When referring to incompletely specified finite state machines, we will use $\emptyset$ to denote an unspecified transition and $\varepsilon$ to denote an unspecified output. The function $\delta(q, a)$ defines the structure of the state transition graph of the finite state machine while the function $\lambda(q, a)$ defines the labels present in each of the edges of that graph.

In a Moore Machine $\lambda(q, a_1) = \lambda(q, a_2)$ for all $a_1, a_2 \in \Sigma$ there by implying that the output of a Moore Machine does not depend on the input, only the state. For the purposes of this paper, we will consider that Moore machines are a special case of Mealy machines, and will therefore be handled by the formalism developed for Mealy machines. The domain of the second variable of functions $\lambda$ and $\delta$ is extended to strings of any length in the usual way:

**Definition 2 (Output of a Sequence)** The notation $\lambda(q, s)$, where $s = (a_1, \ldots, a_k)$ denotes the output of a Moore or Mealy machine after a sequence of inputs $(a_1, \ldots, a_k)$, is applied in state $q$. The output of such a sequence is defined to be

$$\lambda(q, s) \equiv \lambda(\delta(\cdots \delta(q, a_1), a_2), \ldots, a_k)$$  

**Definition 3 (Destination State of a Sequence)**

If $s = (a_1, \ldots, a_k)$ the notation $\delta(q, s)$ denotes the final state reached by a Moore or Mealy machine after a sequence of inputs $(a_1, \ldots, a_k)$, is applied in state $q$. This state is defined to be

$$\delta(q, s) \equiv \delta(\delta(\cdots \delta(q, a_1), a_2), \ldots, a_k)$$  

To avoid unnecessary notational complexities, $\lambda(\emptyset, a) = \varepsilon$ and $\delta(\emptyset, a) = \emptyset$, by definition.

**Training Sets and Loop Free Finite State Machines**

The objective is to infer a finite state machine with minimum number of states that is consistent with a given training set. A training set is specified by one or more sequences of input-output pairs:

**Definition 4** A training set is a set of pairs $T = \{(s_1, l_1), \ldots, (s_m, l_m)\}$ where each pair $(s, l) \in \Sigma \times \Delta$ represents one input string and the output observed for that string.

If the output alphabet is the set $\{0, 1\}$ the training set can be viewed as specifying a set of accepted strings (the ones that output 1) and a set of rejected strings (the ones that output 0). An example of a possible training set is given in figure 1.

Accepted:  1  11  1111  11111
Rejected:  0  101  00  011  11110

**Figure 1:** Training set specified as a set of accepted and rejected strings.

Alternatively, the training set can be specified by one or more sequences where, at each time, the value of the input/output pair is known. Both forms of training set descriptions are equivalent and can be viewed as defining a particular type of incompletely specified finite state machine, a Loop Free Finite State Machine (LFFSM). A LFFSM is a finite state machine that has a state transition graph without loops or reconvergent paths. Figure 2 shows the LFFSM that corresponds to the training set in figure 1.

There is a one to one correspondence between loop free finite state machines and training sets. A loop free finite state machine represents a training set if

1. Its output for each input sequence present in the training set agrees with the label in that training set.
2. The output for input sequences not present in the training set is undefined.
Formally,

**Definition 5** A finite state machine is said to be the loop free finite state machine representing a training set \( T \) if it satisfies definition 1 and the following additional requirements:

\[
\forall q \in Q \setminus q_0, 3 \exists (q_i, a) \in Q \times \Sigma \text{ s.t. } \delta(q_i, a) = q
\]

\[
\forall q \in Q, \forall a \in \Sigma \delta(q,a) \not= q_0
\]

\[
\lambda(q_0, s_1) = \lambda(q, s) \text{ if } (s, s_1) \in T, \quad \lambda(q_0, s) = \epsilon \text{ if } (s, 1) \not\in T
\]

Since the training set \( T \) defines uniquely a loop free finite state machine that it corresponds to, we will use \( T \) to denote both the training set itself and the LFFSM that it defines.

The aim is to construct a machine \( M \) that exhibits a behavior equivalent to \( T \), that is, a machine \( M \) that outputs the same output as \( T \) every time this output is defined. A machine \( M \) is consistent with a TFSM \( T \) if, for any input string \( s = (a_1, \ldots, a_k) \) contained in \( T \), \( \lambda(q_0, s) \equiv \lambda(q_0, s) \). Given a specific mapping function \( F : Q' \rightarrow Q \) with \( F(q_0) = q_0 \) from the states in \( T \) to the states in \( M \), it defines a valid solution iff it satisfies the following two requirements:

**Definition 6** A function \( F \) satisfies the output and transition requirements iff:

\[
\forall q = F(q'), \lambda'(q', a) \equiv \lambda(q, a) \quad (3)
\]

\[
\forall q = F(q'), F(\delta'(q', a)) = \delta(q, a) \quad (4)
\]

It is known that the minimum finite state machine that satisfies the training set can be found by selecting an appropriate mapping function. That result is stated in a formal way in the following theorem:

**Theorem 1** Any machine \( M = (\Sigma, \Delta, Q, q_0, \delta, \lambda) \) consistent with the loop free finite state machine \( T = (\Sigma, \Delta, Q', q_0', \delta', \lambda') \) can be obtained by selecting an appropriate mapping function \( F : Q' \rightarrow Q \), \( F(q_0') = q_0' \) that verifies \( \lambda'(q', a) = \lambda(F(q'), a) \) and \( \delta'(q', a) = \delta(q, a) \).

This result has been implicitly used by Bierman (BP75) and a formal proof has been presented in (OE96). It is important to note that, in general, the minimum FSM equivalent to a given incompletely specified FSM can not be obtained by selecting a mapping function in this way. This result is only valid for Loop-Free Finite State Machines.

Since any machine that satisfies these requirements can be found by selecting a mapping function, the objective of selecting the minimum consistent FSM can be attained by selecting a mapping function that exhibits a range of minimum cardinality.

For the sake of simplicity, and following the notation of other authors (BP75) we will define \( S_i \) as the index of state in the target machine that state \( q_i \) in the original LFFSM maps to, i.e., \( g_{S_i} = F(q_i) \).

An equivalent FSM with \( N \) states can therefore be found by selecting an assignment to the variables \( S_0, \ldots, S_{n-1} \) such that each \( S_i \) is assigned a value between 1 and \( N \), and this assignment defines a mapping function that satisfies (3) and (4).

**Compatible and Incompatible States**

Two states \( q'_i \) and \( q'_j \) in a finite state machine \( T \) are incompatible if, for some input string \( s \), \( \lambda(q'_i, s) \not= \lambda(q'_j, s) \). This information can be represented by a graph, the incompatibility graph. The nodes in this graph are the states in \( Q' \), and there is an edge between state \( q'_i \) and \( q'_j \) if these states are incompatible.

The incompatibility graph is represented by a function \( I : Q' \times Q' \rightarrow \{0, 1\} \). \( I(q'_i, q'_j) = 1 \) if and only if states \( q'_i \) and \( q'_j \) are incompatible.

A clique in the incompatibility graph gives a lower bound on the size of the minimum machine. By definition, pairs of incompatible states cannot be mapped to the same state and therefore, a clique in this graph corresponds to a group of states that must map to different states in the resulting machine. Identifying the largest clique in a graph is in itself an NP-complete problem (GJ79). A large clique (not necessarily the maximum one) can be identified using a slightly modified version of an exact algorithm proposed by Carraugh and Pardalos (CP90). The size of the clique is used to find a machine with a number of states equal to the lower bound, we increase \( N \) by one and re-execute the algorithm.

**The Search Algorithm**

As shown in the previous sections, the objective is to select a mapping function \( F \) that has a range of minimum cardinality.

The constraints that need to be obeyed by the mapping function can be restated as follows:
1. If two states \( q_i^j \) and \( q_j^j \) in the original LPFSM are incompatible, then \( S_i \neq S_j \).
2. If two states \( q_i^j \) and \( q_j^j \) have successor states \( q_i^j \) and \( q_j^j \) for some input \( u \), respectively, then \( S_i = S_j \Rightarrow S_k = S_l \).

These two conditions can be rewritten as:

\[
I(q_i^j, q_j^j) = 1 \Rightarrow S_i \neq S_j
\]

and

\[
q_k^j = \delta'(q_i^j, a) \land q_i^j = \delta'(q_j^j, a) \Rightarrow S_i \neq S_j \lor S_k = S_l
\]

Assume, for the moment, that a search is being performed by a machine with \( N \) states. The basic search with backtrack procedure iterates through the following steps:

1. Select the next variable to be assigned, \( S_i \), from among the unassigned variables \( S_i^j \).
2. Extend the current assignment by selecting a value from the range \( 1 \ldots N \) and assigning it to \( S_i \). If no more values exist, undo the assignment made to the last variable chosen.
3. If the current assignment leads to a contradiction, undo it and goto step 2. Else goto step 1.

This search process can be viewed as a search tree (or decision tree), and we define the decision level as the level in this tree where a given assignment was made. The assignment made at the root is at decision level 0, the second assignment at decision level 1, and so on.

To illustrate the fundamental differences between this and succeeding search techniques, consider an hypothetical example where a search is being performed by a machine with 3 states. Under these conditions, each \( S_i \) can assume only the values 1, 2 or 3. Suppose that variables will be assigned in the order \( S_1 \ldots S_9 \) and that the following restrictions exist in this problem:

\[
S_1 \neq S_2 \lor S_8 = S_9
\]

\[
S_8 \neq S_9 \lor S_2 = S_3
\]

The section of the search tree depicted in figure 3 is obtained by the basic search algorithm described above. In every leaf of this tree a conflict was detected and backtracking took place.

Bierzman noted (BP75) that a more effective search strategy can be applied if some bookkeeping information is kept and used to avoid assigning values to variables that will later prove to generate a conflict. This bookkeeping information can also be used to identify variables that have only one possible assignment left, and should therefore be chosen next. This procedure can be viewed as a generalization to the multi-valued domain of the unit clause resolution of the Davis-Putnam procedure (DP60), and can be very effective in the reduction of the search space that needs to be explored.

This can be done in the following way \(^1\):

- For each node in \( Q' \), a table is kept that lists the value that \( S_j \) can take.
- Every time some \( S_i \) is assigned to the value \( z \), the tables for all unassigned nodes are updated according to the following algorithm:
  - If \( I(q_i^j, q_j^j) = 1 \), then \( z \) is removed from the list of values that \( S_j \) can be assigned to. (Equation (5))
  - If the assignment of \( S_i \) forces some specific value \( z \) on some node \( q_i^j \) (because of the equation (6)), then all values except \( z \) are removed from the table of possible values in node \( q_i^j \).
- When selecting the next variable \( S_i \) to assign, priority is given to the variables that correspond to nodes in \( Q' \) that can at that depth take only one possible value.

Clearly, the information on these tables needs to be updated after every assignment is made to some \( S_i \) and also after each backtracking takes place. Bierzman has shown, and our experiments have confirmed, that for hard problems, this improved search strategy leads to considerable improvements in speed, and expands the size of the problems that can be addressed.

For the example considered above, the section of the search tree obtained is shown in figure 4. Note that as soon as a value is assigned to \( S_8 \), the algorithm automatically identifies that no solution exists because

\(^1\)The original formulation is made in different terms. We present here an adapted description of Bierzman's algorithm, suited to follow our different notation. The interested reader is referred to the reference for the original formulation.
no value can possibly be assigned to \( S_9 \). This may lead to very considerable savings, specially if \( S_9 \) is not the next variable to be assigned.

**Figure 4: Reduced search tree for algorithm 2 (Bierman)**

**Improving the Efficiency of the Search Algorithm**

The major contribution of this work is to show that the application of new search techniques can improve considerably the efficiency of the search algorithm and extend the range of problems that can be effectively solved.

The improvement proposed is the application of **conflict diagnosis techniques** to allow for the use of **dependency directed backtracking**. In this paper, we use the term **dependency directed backtracking** to denote two techniques that can, in fact, be used independently. The first technique is based on the realization that, under certain conditions, it is possible to perform jumps in the decision tree that span one or more decision levels. These jumps are called **non-chronological backtracks or backjumps** (RN96). The second technique is due to the fact that, under similar conditions, it is possible to assert a set of conditions that need to be obeyed in the future if a solution is to be found.

Before we describe these techniques, we will first reformulate slightly the problem at hand. For this, we point out that the set of restrictions (5) and (6) can be computed only once at the beginning of the algorithm execution. This means that a set of restrictions is generated, where each restriction is of the form:

\[
(X_1 \text{ op } X_2) \lor (X_3 \text{ op } X_4) \lor \ldots \lor (X_{n-1} \text{ op } X_n) \quad (9)
\]

Each restriction is a disjunction of one or more elements of the form \( X_i \text{ op } X_{i+1} \) where each \( X_i \) is either a variable \( S_j \) or a constant in the range 1 to \( N \). Each operator \( \text{ op } \) is either \( = \) or \( \neq \). Restrictions generated from equation (5) have one element, while restrictions generated from equation (6) have 2 elements. It is clear that the algorithm can be easily extended to satisfy any restrictions with more than 2 elements, but the problem formulation does not create them.

**Conflict diagnosis** (aPMSS96) is a technique first proposed by Stallman and used in the context of truth-maintenance systems (de 86; SS77) and constraint satisfaction solvers. To make clear how conflict diagnosis can be used to prune the search tree, consider again the example described above, and the section of the search tree shown in figure 5. In this problem, for each possi-

**Figure 5: Example of non-chronological backtracking used by algorithm 3 (BICA)**
These facts lead to the following general procedure for handling conflicts and controlling the backtrack search procedure.

1. Every time a conflict is detected, diagnose the conflict and generate a restriction that expresses that conflict. The result of this diagnosis is the consensus of all the assignments that originated the conflict.

2. Identify the variable present in the conflict that is at the highest decision level, and perform a non-chronological backtrack to that level.

3. Store the restriction generated by the conflict diagnosis engine in the restriction database, and use it to restrict choices of variables in the future. This is sometimes described as dependency directed backtracking.

**Conflict diagnosis**

Every time a conflict arises, the assignments that are directly responsible for the conflict are identified. The conditions that need to be obeyed to resolve this conflict can be summarized in a restriction of the form of the general restriction (9). Consider again the example in figure 5, and the conflicts detected in nodes A, B and C. At each of these nodes, it is found that no possible assignments exist to variable S8.

Analyzing in more detail the conflict in node A, we see that S8 cannot take the value 1 because the second restriction in the example, restriction (8) is not satisfied, given the current assignments S2 = 1 and S3 = 2. The other values possible for S2, 0 or 3, are barred because restriction (7) would not be satisfied, since S1 = 1 and S2 = 1. The condition that lead to this conflict can therefore be written as:

\[
S_1 = 1 \land S_2 = 1 \land S_3 = 2 \land S_8 = 1
\]  

(10)

This condition represents simply the consensus of all the restrictions that lead to the conflict in this node. The last element in the restriction, S8 = 1 is also a cause of the conflict, and therefore should be listed in the condition.

For nodes B and C, a similar procedure could be followed and we would arrive at the following two conditions:

\[
S_1 = 1 \land S_2 = 1 \land S_3 = 2 \land S_8 = 2
\]  

(11)

\[
S_1 = 1 \land S_2 = 1 \land S_3 = 2 \land S_8 = 3
\]  

(12)

Note that, in this simplified example, the three conditions are very similar, but that needs not be the case in general. The cause of the conflict detected in node D can now be diagnosed as the consensus of the causes of all the conflicts in the children nodes, A, B and C. The consensus of a set of restrictions is simply the conjunction of all restriction elements that are in agreement in all the restrictions in the set. A variable that is present in all the values of its domain is removed. In this case, all the values for variable S8 were tried, and therefore the conflict cannot be due to any specific choice of S8. This is a general rule, and a non-chronological backtrack can only be made after all the possible choices for a given variable have been tried.

The conflict in node D is therefore diagnosed as being caused by

\[
S_1 = 1 \land S_2 = 1 \land S_3 = 2
\]  

(13)

To solve this conflict, the negation of this condition has to be asserted, and therefore the restriction

\[
S_1 \neq 1 \lor S_2 \neq 1 \lor S_3 \neq 2
\]  

(14)

can be added to the database, since this restriction will have to be satisfied in any assignments that lead to a solution.

Clearly, this restriction can only be satisfied if a non-chronological jump to the level of node \( E \) is performed. Note that the condition in (13) is propagated backwards as the cause of the conflict in node \( D \), and will be used in the computation of the cause of the conflict in node \( E \).

This leads to the following general procedure for the diagnosis of conflicts and control of backtrack:

1. At each leaf in the search tree, compute the set of assignments involved in the conflict.

2. At each non-leaf node where a conflict is detected, compute the consensus of all the conditions involved in the conflicts of children nodes.

3. Complement the resulting condition, and add it to the restriction database. Also, store this condition as the cause of the conflict at this node.

4. Compute the highest decision level involved in this condition, and perform a non-chronological backtrack to that level.

**Experimental Results**

To test the performance of the algorithms, we randomly generated 115 finite state machines with binary inputs and outputs. These finite state machines were reduced and unreachable states were removed before the experiments were run. The size of the machines (after reduction) varied between 3 and 19 states. A total of 575 training sets were generated, with each training set containing twenty strings of length 30. Each program was given 30 minutes of CPU time and 64 Megabytes of memory to find the minimum consistent machine in a 133MHz Pentium running Linux.

Figure 6 shows the number of backtracks required for each problem. The first algorithm (labeled Bierman) is the approach proposed by Bierman (BP75) and described in section . This algorithm uses the improvements described in the final part of that section, but does not use any of the new techniques described in section . The second algorithm, BICA, uses the techniques proposed in this work and described in section . The problems were sorted in order of increasing CPU
time taken by the algorithms. The improved version of the algorithm was able to solve 468 problems, while the original version, as proposed by Bierman, solved only 371 problems. It is clear that the number of backtracks

![Figure 6: Number of backtracks executed by each of the algorithms.](image)

by the improved algorithm is much smaller than the number of backtracks used by the original algorithm proposed by Bierman. However, each backtrack takes longer to execute in the modified algorithm. Figure 7 shows the time each backtrack takes to execute, for all the problems that are solved in the allotted time. This graph shows that, in the interesting range, i.e., problems with index 250 to 360, each backtrack is on the order of 2 times more costly if the improved algorithm is used.\(^2\) However, this extra cost is more than offset by

![Figure 7: Cpu time by executed backtrack in the search tree.](image)

the much smaller number of backtracks needed. Figure 8 shows the total CPU time spent by each of the two algorithms. From this graph, it is clear that the techniques proposed in this work improve the performance of the search algorithm by a large factor. The net result of this improvement is that, given a fixed amount

![Figure 8: Comparison of the CPU time for the original and improved algorithms.](image)

\(^2\)For the easier problems, the ones with an index lower than 250 in these graphs, very little CPU time is spent overall and the statistic shown in figure 7 is not very significant. of time, many problems that are outside the reach of the existing approaches can be solved by BICA. Figure 9 shows the fraction of problems that can be solved,

![Figure 9: Fraction of the problems solved by the original and modified algorithms.](image)

plotted as a function of the number of states in the finite state machine that represents the solution. BICA manages to solve all problems that have a solution with no more than 12 states. It is clear from these results that, for the randomly generated machines created by the procedure outlined above, the use of dependency-directed backtracking provides a very significant advantage. In many of these cases, the application of these techniques reduces the CPU time required by many orders of magnitude.

Other techniques have been proposed for the inference of finite state machines, some of them based on recurrent neural network architectures (DM93; GMC+92; Pol91; WK92). This work is highly relevant because the ability to train neural networks to learn and recognize input sequences is an important objective in itself. Moreover, these methods may also work even if the system under study cannot be exactly modeled by a finite state machine. However, given the results obtainable by the algorithms described in this work and the results published in the literature
neural network based methods are not competitive in inference problems where the exact identification of a small finite state machine leads to an exact solution. Note that for each of the problems that was actually solved, the smallest consistent finite state machine was exactly identified, and the testing error would be, in principle, zero. The largest machines inferred are very complex machines. Figure 10 shows an example of a 15 state machine inferred by algorithm. The authors believe that exact inference of machines of this complexity is currently outside the reach of recurrent neural network based algorithms.

Figure 10: Example of a 15 state machine inferred by the algorithm.

Conclusions and Future Work
We presented a new approach for the problem of inferring the finite state machine with minimum number of states that is consistent with a given training set. This approach is based on well known algorithms but uses a technique that provides a very significant improvement, namely dependency-directed backtracking. Although this technique has previously been applied in other domains, it has not been used on this specific problem.

The results show that, for the set of problems studied, the use of these techniques results in an algorithm that is many orders of magnitude faster than the alternative approaches. For this set of problems, the extra overhead incurred by the bookkeeping necessary to apply these techniques is recovered in the vast majority of the problems, except for the smaller and simpler ones.

For the set of problems studied, the algorithms described in this paper find the exact solution in very little time in all problems that have solutions up to 11-12 states, and become progressively less effective in problems that have solutions 13 and 15 states. Naturally, the dimension of the problems that can be solved depends strongly on the exact training set used, the number of possible inputs and outputs and the structure of the state transition graph. We believe, however, that the techniques described here will be extremely effective in a variety of other situations.

There are several open problems that are of interest for future research. One of these problems is related with the outer loop of the algorithm. The current version of BICA starts by looking for a machine with $N$ states, where $N$ is the size of the largest clique found in the incompatibility graph. If this search fails, it increases the target size by one, and restarts the algorithm, therefore losing all the information stored in the restriction database. We are currently looking at ways to change this procedure so that useful and still valid restrictions are kept in the database. Since some restrictions may be no longer valid when the target size is increased, this is an interesting topic for future research. Another topic for future research is the applicability of additional pruning techniques like recursive learning (KP92) to speed up the search process. It may also be possible to improve the performance of the system by a considerable factor if a more strict control is imposed on restrictions added to the clause database. The current version of the algorithm poses no limits on the size of the restrictions and does not analyze whether redundant restrictions are added to the database, although a simple fingerprinting technique is used to avoid duplication of equivalent restrictions.

A distinct possibility for the solution of this type of problems is the use of heuristic algorithms, like, for instance, the ones described by Lang (Lan92). This particular algorithm collapses the large original LFIFS using local and possibly non-optimal choices. Clearly, these algorithms remain the option of choice if the exact algorithms cannot handle the problems under study. We believe, however, that the search techniques described in this work can be used in an heuristic approach in problems where the search for the optimal solution is too expensive.

Finally, it must be observed that the algorithms described here solve problems specified by a very general set of restrictions. The most interesting direction for future research will probably be the application of these techniques to other problems that can be formulated in a similar fashion.

References


