Limits of Exact Algorithms For Inference of Minimum Size Finite State Machines

Arlindo L. Oliveira[1] and Stephen Edwards[2]

[1] Cadence European Laboratories/INESC-IST, 1000 Lisboa, Portugal
[2] UC Berkeley, Berkeley CA 94720, USA

Abstract. We address the problem of selecting the minimum sized finite state machine consistent with given input/output samples. The problem can be solved by computing the minimum finite state machine equivalent to a finite state machine without loops obtained from the training set. We compare the performance of four algorithms for this task: two algorithms for incompletely specified finite state machine reduction, an algorithm based on a well known explicit search procedure and an algorithm based on a new implicit search procedure that is introduced in this paper.

1 Introduction and Related Work

We address the problem of inferring the finite state machine (FSM) with minimum number of states that is consistent with a given training set. This problem is important for the machine learning community because of the well known connections between hypothesis compactness and predictive accuracy.

This problem is equivalent to the problem of determining if there exists a k-state DFA consistent with a set of labeled strings. This problem is known to be NP-complete [7]. Finding an approximate solution, within any polynomial factor, is also an NP-hard problem [12]. The problem can be solved in time polynomial on the input size if all strings of length n or less are given [13], but remains NP-complete if a small fixed fraction of these strings are missing [1].

The problem becomes easier if the algorithm is allowed to make queries or experiment with the unknown machine. Angluin [2] and Schapire [14] proposed algorithms that solve the problem in polynomial time by allowing the algorithm to ask membership queries.

Bierman et al. [3, 4] proposed the best algorithm known to the authors for the specific problem addressed here, where the learner has no control over the training set. This algorithm is briefly described in section 3.

Algorithms for the reduction of incompletely specified finite state machines (ISFSMs) can also be used to solve the problem addressed here. The reduction of incompletely specified finite state machines is a more general problem and is also known to be NP-complete [6]. This problem has been the subject of extensive research and several implementations of the best known algorithms are available. The results section compares the performance of two special purpose algorithms with two algorithms for ISFSM reduction: `stamina` [8], the most popular program for the simplification of finite state machines and `ism` [10], an implementation of a similar algorithm that uses implicit enumeration techniques.

2 Definitions

The algorithms described in this chapter can be used with minor modifications to induce either Mealy or Moore machines. Due to space limitations, we will describe only the more general case, the induction of Mealy machines. We define a finite state machine with unspecified transitions in the standard way:

Definition 1. A finite state machine is a tuple $M = \big(\Sigma, \Delta, Q, q_0, \delta, \lambda\big)$ 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, $\delta(q, a) : Q \times \Sigma \rightarrow Q \cup \{\phi\}$ is the transition function, and $\lambda(q, a) : Q \times \Sigma \rightarrow \Delta \cup \{\epsilon\}$ is the output function.

We will use $q \in Q$ to denote a particular state, $a \in \Sigma$ a particular input symbol and $b \in \Delta$ a particular output symbol. Furthermore, ϕ denotes an unspecified transition while ϵ denotes an unspecified output.

The domain of the second variable of functions λ and δ is extended to strings of any length in the usual way. Let $s = (a_1, \ldots, a_k)$ be a string of input symbols and the notation $\lambda(q, s)$ denote the output of finite state machine after sequence s is applied in state q. The output of such a sequence is defined to be $\lambda(q, s) \equiv \lambda\big(\delta(\delta(\cdots\delta(q, a_1)\cdots), a_{k-1}), a_k\big)$. Similarly, $\delta(q, s)$ denotes the final state reached by a finite state machine after a sequence of inputs $(a_1, \ldots, a_k)$, is applied in state q. The final state is defined to be $\delta(q, s) \equiv \delta(\delta(\ldots\delta(\delta(q, a_1), a_2)\ldots), a_k)$. To avoid unnecessary notational complexities, $\lambda(\phi, a)$ is defined to be equal to ϵ and $\delta(\phi, a) = \phi$.

Definition 2. A training set is a set of pairs $\{(s_1, l_1), \ldots, (s_n, l_n)\}$ where each pair $(s, l) \in \Sigma^k \times \Delta$ 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). 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 *Tree Finite State Machine* (TFSM).

Definition 3. A Tree Finite State Machine T is a finite state machine satisfying definition 1 and the following additional requirements:

$$\forall q \in Q \setminus q_0,\ \exists^1 s \in \Sigma^z \text{ s.t. } \delta(q_0, s) = q$$
$$\forall q \in Q,\ \forall a \in \Sigma\ \delta(q, a) \neq q_0$$

These requirements specify that the graph that describes the TFSM is a tree rooted at state q_0. A TFSM T is said to contain a string s if $\lambda(q_0, s) \neq \epsilon$.

The output in a given transition $b_i = \lambda(q_i, a_i)$ is said to be compatible with $b_j = \lambda(q_j, a_j)$ and denoted $b_i \equiv b_j$ iff $b_i = b_j$ or $b_i = \epsilon$ or $b_j = \epsilon$. The objective is to construct a machine M that exhibits a behavior equal to T for all strings contained in T, where T is the TFSM that contains all strings in the training set and only these. Assume that $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$, $Q = \{q_0, \dots q_k\}$ and $T = (\Sigma, \Delta, Q', q'_0, \delta', \lambda')$, $Q' = \{q'_0, \dots q'_{k'}\}$ unless otherwise stated.

A machine M is consistent with a TFSM T if, for any input string $s = (a_1, \dots, a_k)$ 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 4. A function F satisfies the output and transition requirements iff:

$$\forall q = F(q'),\ \lambda'(q', a) \equiv \lambda(q, a) \tag{1}$$

$$\forall q = F(q'),\ F(\delta'(q', a)) = \delta(q, a) \tag{2}$$

With these definitions, we can now state the following result:

Theorem 1 *For any machine $M = (\Sigma, \Delta, Q, q_0, \delta, \lambda)$ consistent with the tree finite state machine $T = (\Sigma, \Delta, Q', q'_0, \delta', \lambda')$ there exists a mapping function $F : Q' \rightarrow Q$, $F(q'_0) = q_0$, that satisfies the output and transition requirements.*

Proof: let $s^i_k = (a^i_1, a^i_2 \dots a^i_k)$ be an arbitrary substring of some string $s^i = (a^i_1, a^i_2 \dots a^i_z)$ contained in T and let the mapping function F be defined by $F(\delta'(q'_0, s^i_k)) = \delta(q_0, s^i_k)$.

Consider now all strings $s^i_{k+1} = (a^i_1, a^i_2 \dots a^i_{k+1})$ contained in T. By the hypothesis, $\lambda(q_0, s^i_{k+1}) \equiv \lambda'(q'_0, s^i_{k+1})$ and therefore the output requirement has to be satisfied (simply make q' in expression (1) equal to $\delta'(q'_0, s^i_k)$).

Furthermore, since the strings s^i_{k+1} are themselves substrings of some string contained in T (or else no s^i_{k+1} contained in T exists, in which case the requirement is automatically satisfied), $F(\delta'(q'_0, s^i_{k+1}))$ equals $\delta(q_0, s^i_{k+1})$ and therefore F also meets the transition requirement. □

The result in this theorem is important because it is not valid, in general, if the incompletely specified machine T is not a TFSM.

Two states q'_i and q'_j in a finite state machine T are incompatible if, for some input string s, $\lambda(q'_i, s) \not\equiv \lambda(q'_j, s)$. The *incompatibility graph* represents this information. 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.

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. Indentifying the largest clique in a graph is in itself an NP-complete problem [6]. A large clique (not necessarily the maximum one) can be identified using a simple branch and bound algorithm with an extra stopping condition. The size of the clique provides a lower bound on the number of states needed in the resulting machine. This lower bound is used as the starting point for both the explicit and the implicit enumeration algorithms described below.

3 The Explicit Search Algorithm

The explicit search algorithm implemented for the purpose of comparison is based on the algorithm proposed by Bierman et al. [4]. It builds a finite state machine and a mapping function F by fitting transitions from the TFSM T into the machine M, one by one, forcing the transition (2) and output requirements (1) to be satisfied for all the transitions considered.

The algorithm is started with a machine containing only the reset state. At any time, the algorithm selects a transition in T and has to verify that transitions in M generate outputs consistent with the transitions in T. Let q'_s be the state where the transition under consideration origins and the transition under input a be the one under consideration. Two main cases should be considered:

A The choice of the mapping of the destination state is forced by an existing transition, labeled with a. If this is the case, two things may happen:
1) The output of the corresponding transition in M is consistent with the output of the transition in T. This means that the machine M is, so far, consistent with T.
2) The output of the corresponding transition is not consistent with the output of the transition in T. In this case, some transition in M (not necessarily this one) is wrong and the algorithm backtracks to the last point where it had a choice.

B There is no existing transition labeled with a, so any of the existing states or a new state is a possibility. The algorithm picks one choice and proceeds.

4 The Implicit Search Algorithm

The implicit approach described in this section avoids the need to explicitly search for the right mapping function. It does so by keeping an implicit description of all the mapping functions that satisfy the output and transition requirements. This approach makes the implicit algorithm very simple to describe, but incurs the overhead imposed by the use of discrete function manipulation routines. This overhead can be recovered if the regularities of the problem make the use of an implicit enumeration technique more efficient than an explicit one.

To simplify the explanation, we assume that the output alphabet Δ is equal to the set $\{0, 1\}$. The approach can be easily applied to the more general case.

4.1 Discrete Functions and Multi-valued Decision Diagrams

The discrete function manipulation needed to keep this implicit list of possible mappings is performed using multi-valued decision diagrams to represent the discrete functions involved. A full description of this technique is outside the scope of this work and only a brief introduction is made here. The reader is referred to the work by Kam and Brayton [9] for a complete treatment of the subject. The approach is based on the fact that any binary valued function of k discrete variables, $x_1, x_2, \ldots, x_k$ $F : P_1 \times P_2 \times \cdots \times P_k \rightarrow \{0, 1\}$ can be represented by a Multi-valued Decision Diagram (MDD). An MDD is a rooted, directed, acyclic graph where each non-terminal node is labeled with the name of one variable. An MDD for F has two terminal nodes n_z and n_o that correspond to the leaves of the graph. Every non-terminal node n_i, labeled with variable x_j, has $|P_j|$ outgoing edges labeled with the possible values of x_j. Each of these edges points to one child node. The value of F for any point in the input space can be computed by starting at the root and following, at each node, the edge labeled with the value assigned to the variable tested at that node. The value of the function is 0 if this path ends in node n_z and 1 if it ends in node n_o.

A decision diagram is called *reduced* if no two nodes exist that branch exactly in the same way and it is never the case that all outgoing edges of a given node terminate in the same node [5]. A decision diagram that is both reduced and ordered is called a reduced ordered decision diagram. For a given variable ordering, reduced, ordered MDDs are canonical representations for functions defined over that domain.

Packages for the manipulation of discrete functions using MDDs [9] allow the user to realize, amongst others, the following operations:
1) Creation of a function from an arithmetic relation. For example, $f := (x_i = x_j)$ returns the function that is 1 for all points of the input space where $x_i = x_j$.
2)Boolean combination of existing functions. For example, $f := g \wedge h$ returns the function that is 1 only when functions g and h are 1.

4.2 Implicit Enumeration of Solutions

An implicit list of the valid mapping functions $F : Q' \rightarrow Q$ can be directly manipulated using simple Boolean operations. This list is kept by considering a function $\mathcal{F} : Q^{|Q'|} \rightarrow \{0, 1\}$ defined as follows:

Definition 5. $\mathcal{F}(x_0, x_1, \ldots, x_{|Q'|-1}) = 1$ for the point $v_0, v_1, \ldots, v_{|Q'|-1}$ if the mapping function F defined by $F(q'_0) = v_0, F(q'_1) = v_1, \ldots, F(q'_{|Q'|-1}) = v_{|Q'|-1}$ induces a machine M with $|Q|$ states that satisfies the output and transition requirements in expressions (1) and (2).

There is a one-to-one correspondence between each variable x_i in the support of $\mathcal{F}$ and each state $q'_i \in Q'$. Therefore, restrictions on valid mapping functions can be written as restrictions on the variables x_i. If two states in T, q'_i and q'_j, have to be mapped to different states in Q, this is equivalent to the statement that $\mathcal{F}$ can only be 1 for points where $x_i \neq x_j$.

The transition and output requirements impose restrictions on the function $\mathcal{F}$. Let q'_i and q'_j be two states in Q'. For any two transitions out of these states that take place on the same input and have different outputs, the output requirement forces the source states of the transition to be mapped to different states. Let $\lambda'(q'_i, a_i) = b_i$ and $\lambda'(q'_j, a_j) = b_j$. Then, for Mealy machines this requirement translates into:

$$(a_i = a_j) \wedge (b_i \not\equiv b_j) \Rightarrow x_i \neq x_j \tag{3}$$

Next-state determinism implies that, for any two transitions in the original machine that take place on the same input, the same assignment for the initial

states implies the same assignment for the final states. Let $q'_k = \delta'(q'_i, a_i)$ and $q'_l = \delta'(q'_j, a_j)$. This requirement translates into the restriction:
$(a_i = a_j \wedge x_i = x_j) \Rightarrow (x_k = x_l)$. This can be rewritten as

$$(a_i = a_j) \Rightarrow (x_i \neq x_j \vee x_k = x_l) \tag{4}$$

Expressions (3) and (4) can be used to form $\mathcal{F}$ using the algorithm in figure 1. This algorithm can be made more efficient by using the incompatibility graph information to assign arbitrary but different values to the states on a large clique of the graph.

```
MainLoop()
    F := 1
    R := ∅                                                Stores the processed states
    foreach q'_i ∈ Q'
        R := R ∪ q'_i                                      Add this state to the list
        foreach q'_j ∈ R
            foreach a ∈ Σ s.t. δ(q'_i, a) ≠ φ ∧ δ(q'_j, a) ≠ φ
                if λ'(q'_i, a) ≢ λ'(q'_j, a)                    Output requirement
                    F := F ∧ (x_i ≠ x_j)
                q'_k := δ(q'_i, a)
                q'_l := δ(q'_j, a)
                if q'_k ≠ φ ∧ q'_l ≠ φ
                    F := F ∧ ((x_i ≠ x_j) ∨ (x_k = x_l))        Transition requirement
    return F
```

Fig. 1. Pseudo-code for the implicit enumeration algorithm

5 Experimental Results

We performed the comparison between the four algorithms, using three simple target machines with no more than 8 states. For each machine, a number of training sets was generated, each training set consisting of a single random string of length between 10 and 65. For each time point, the value of the output was available, and, therefore, each training set was effectively equivalent to a set of labeled strings with a size comprised between these two limits. The two general purpose algorithms considered are `stamina` and `ism`. The two special purpose algorithms are `mmm`, the explicit enumeration algorithm and `iasmin`, the implicit enumeration algorithm described in section 4.

For each length considered, five training sets were generated. The various programs were then used to find the minimum machine consistent with each of the training sets. The leftmost graph in figure 2 describes the growth in computation time observed for the training sets derived from the first machine. The behavior observed in this graph is very similar to the behavior observed for the other two machines. Each point represents the average over the five different training sets generated for each given length. In all cases, state minimization algorithms require a time that increases exponentially in the length of the training set while **iasmin** and **mmm** show a less drastic increase. The different behavior observed illustrates well the distinct exponential dependences off the two approaches: state minimization algorithms require time exponential in the size of the original training set, while the special purpose algorithms require time exponential in the size of the final machine. It is also clear that **mmm** is much more efficient than **iasmin**, but, for small problems, this is to be expected because **iasmin** has some overhead that is only recovered for larger problems.

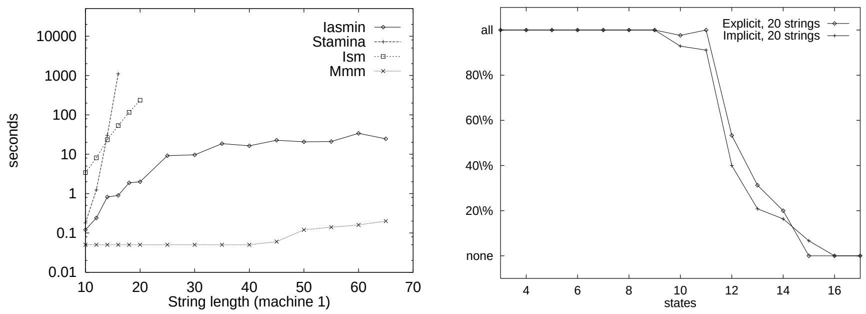

Fig. 2. CPU time for the four algorithms with increasing training set sizes (left) and fraction of problems solved for randomly generated FSMS with a given size (right).

To examine in more detail the relative performance of the algorithms based on explicit or implicit search presented on sections 3 and 4 we performed another experiment with a large set of randomly generated finite state machines. For each randomly generated machine, the minimum equivalent finite state machine was identified using standard logic synthesis techniques [11] and used to label the training sets. In total, 575 training sets were generated from 115 different machines. Each training set contained twenty strings of length 30. The original finite state machines were reduced and unreachable states were removed before the experiments were run. Each program was given one hour and 256 Megabytes of memory to find the minimum consistent machine in a DEC/alpha workstation.

The rightmost graph in figure 2 shows the fraction of the problems each algorithm was able to solve in the allotted time/space plotted as a function of the number of states in the minimum machine.

6 Conclusions and Future Work

The results presented show that general purpose algorithms for ISFSM reduction cannot be used effectively for FSM inference from samples, despite the fact that these algorithms are very effective in logic synthesis applications. The two special purpose algorithms addressed performed much better in the set of problems studied. The performance on the implicit enumeration algorithm was compared with the performance off a well known explicit search algorithm. The results obtained in a set of problems obtained with randomly generated FSMs show very similar performances. This result is surprising because the two algorithms use very different techniques and search the space of solutions using a totally dissimilar approach. It remains an open question whether the behavior observed implies that this type of problems becomes intrinsically very difficult when the machines reach 13 to 15 states, or whether alternative algorithms will be able to be successful in at least a significant fraction of machines of this size.

Both algorithms have some potential for improvement. For the implicit enumeration algorithm, it may be possible to use a different representation as the support for discrete function manipulation. For the explicit search algorithm, more powerful search techniques may lead to interesting gains in performance. Because of the known complexity of the task, exact solutions for this problem are likely to be always limited on the size of the problems they can handle. However, we believe that the use of alternative techniques like the ones described here will eventually push outwards the limits on the size of the problems that can be solved effectively.

References

1. D. Angluin. On the complexity of minimum inference of regular sets. *Inform. Control*, 39(3):337–350, 1978.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, November 1987.
3. A. W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Trans. on Software Engineering*, SE-2:141–153, 1976.
4. A. W. B. R. I. Biermann and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. on Computers*, C-24:122–136, 1975.
5. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
6. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.
7. E. M. Gold. Complexity of automaton identification from given data. *Inform. Control*, 37:302–320, 1978.
8. G. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. In *The Proceedings of the European Design Automation Conference*, 1991.
9. T. Kam and R.K. Brayton. Multi-valued decision diagrams. *Tech. Report No. UCB/ERL M90/125*, December 1990.
10. T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni Vincentelli. A fully implicit algorithm for exact state minimization. Proc. Design Automat. Conf., 1994.
11. Arlindo L. Oliveira and Stephen A. Edwards. Inference of state machines from examples of behavior. Technical report, UCB/ERL Technical Report M95/12, Berkeley, CA, 1995.
12. L. Pitt and M. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, 1993.
13. S. Porat and J. A. Feldman. Learning automata from ordered examples. In *Proc. 1st Annu. Workshop on Comput. Learning Theory*, pages 386–396, San Mateo, CA, 1988. Morgan Kaufmann.
14. R. E. Schapire. *The Design and Analysis of Efficient Learning Algorithms.* MIT Press, Cambridge, MA, 1992.