A Genetic Algorithm for Regular Inference

Philip Hingston

Edith Cowan University
Perth, Western Australia
p.hingston@ecu.edu.au

Abstract

We show how a genetic algorithm can be used for the inference of a regular language from a set of positive (and optionally also negative) examples. The genetic algorithm attempts to find the simplest description of the example data in terms of a finite state automaton model.

1 INTRODUCTION

The inference of regular languages has important applications in fields such as exploratory sequential analysis, artificial intelligence, pattern recognition and data mining. In this paper, we show how a genetic algorithm (GA) can be used for the inference of a regular language from a set of positive (and optionally also negative) examples. The GA attempts to find the simplest description of the example data in terms of a finite state automaton model.

The structure of the paper is as follows. We first outline the problem of regular inference. We then review the relevant theory of finite state automata and regular languages, and summarize the main approaches to the regular language induction problem in terms of this theory. Next we explain the concept of Minimum Message Length as a model selection tool, and how we compute it for finite state automata. Then we give a brief account of genetic algorithms. We then pull these strands together to describe our algorithm, a genetic algorithm using Minimum Message Length to induce regular languages. Finally, some experimental results on the performance of the algorithm are presented.

2 REGULAR INFERENCE

Consider the following two sets of strings:

```
I+ = {abbaaaa, ab, ba, aaaaabb},
I- = {aabaaa, a, b, baa, aba}.
```

I+ is a randomly generated sample of strings in a certain regular language, and I- is a randomly generated sample of strings that are not in the language. Can the reader guess the rule that determines membership in this language? Of course, there are many possible answers that would be consistent with this small sample. We want to identify the correct answer as often as possible, and in any case, we want our answers to be useful in predicting whether other strings are in the target language or not. This is the problem of regular inference from positive and negative samples.

(By the way, the "correct" answer in this case, is the set of strings over the alphabet {a,b} in which the number of a's and the number of b's are congruent modulo 3.)

There are many applications of the inference of regular languages. For example, in the case of exploratory sequential analysis, we have coded behavior sequences, and we want to find out something about the underlying processes that produce the observed behavior. In speech recognition applications, we have examples of strings of phonemes representing spoken words, and we want to find models that enable us to recognize these words. As a data mining example, say we have a data set of sequences of credit card transactions, and want to derive a model that will detect possible fraudulent activities.

In the grammatical inference tradition, we usually have a set of negative examples too – strings that are not in the language. Sometimes one is provided with an oracle or teacher that will answer whether a particular string is in the language or not, or other questions that can be used to identify the language. These additional data are needed if one wishes to identify the target language exactly. In the applications listed above, we are more likely to have only a set of positive samples. In that case, the target language can only be approximately identified, and heuristic methods come into play.

Many existing regular inference algorithms use finite state automata to describe regular languages. The connection with finite state automata is explained in the next sections.

3 THEORY OF FSA

A basic result of automata theory states that a language is regular if and only if it is accepted by (or, equivalently, generated by) an FSA. In particular, for any finite set of strings, there is an FSA that accepts exactly that set of strings. One such FSA is the prefix tree acceptor (PTA) of the strings. The PTA may be constructed by simply laying out the strings in the language, using a state to represent each unique prefix of one of the strings.

We can illustrate this using the sample data set I+ from above. The PTA of I+ is shown in the state diagram below, Figure 1. In the figure, circles represent the states, and labeled arcs between them represent transitions. We follow the usual convention of marking the start state with a ">" and using a double-circle for the final states.

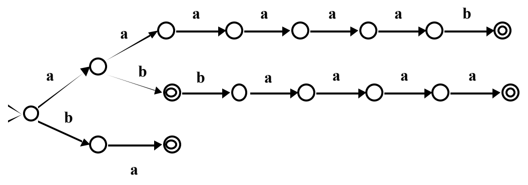

Figure 1: Example Prefix Tree Acceptor

A useful generalization of FSAs is a probabilistic FSA (sometimes called a stochastic FSA). We introduce a special delimiter symbol, which is not in the alphabet of the FSA, say "^". If a state emits the delimiter symbol, this is taken to indicate the end of the string, and no next state is specified (or equivalently, the next state is understood to be the start state). Thus any state that can emit the delimiter symbol is considered a final state. We can now define a probabilistic finite state automaton as an FSA with transition probabilities, giving the probability that a particular symbol will be emitted when we are in a particular state. Figure 2 shows one possible set of transition probabilities for the FSA in Figure 1. Notice the "transitions" from the final states using the delimiter symbol.

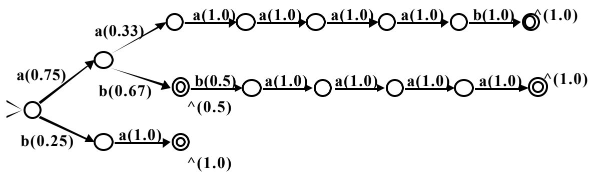

Figure 2: FSA with Transition Probabilities

Note that probabilistic FSAs are essentially a special case of discrete output first-order Hidden Markov Models (HMMs), which have been used extensively in applications such as speech recognition (see, for example, (Stolcke 1994)). The key difference is that HMM states are specified by two probability distributions, which govern the symbol to be output as well as the next state. We do not require the full generality of HMMs for our target applications, and FSAs are much more tractable.

4 THE SEARCH SPACE FOR INDUCTIVE INFERENCE

As stated earlier, we are concerned with the problem of identifying a regular language from a finite sample of strings. We have now seen that this is equivalent to finding a corresponding FSA (the target). We usually assume the sample to be large enough for all the transitions in the target FSA to be represented (the sample is then said to be *structurally complete*). One way to approach the problem is to cast it as a search problem. To do that, we first need to consider what space of solutions we are searching.

It seems reasonable to restrict the search space by requiring that solutions satisfy the following properties:

1. They accept the strings in I+.
2. Every transition is used in accepting some string of I+ (and therefore all states are accessible).

It is easy to see that the PTA satisfies these. It can be shown that every (possibly non-deterministic) FSA obtained by merging states of the PTA does too. It can further be shown that a deterministic FSA that satisfies the two conditions can be obtained by merging states of the PTA (Dupont 1994b). Thus each such FSA can be identified with a partition of the states of the PTA. These FSA's form a lattice, with the PTA at the top of the lattice. Given two FSA's F and G, F≤G in the lattice if F can be obtained by merging states of G (that is, the partition corresponding to G is a refinement of the one for F). At the bottom of the lattice is a single-state FSA, U. This FSA accepts not only I+, but also any string over the same alphabet. In general, as we move down the lattice, the set of languages accepted by the FSA becomes more general. To illustrate this, consider what happens if we merge the two states that follow the start state in Figure 1. The resulting FSA would be non-deterministic, so further states must be merged to obtain a deterministic FSA, giving the FSA shown in Figure 3.

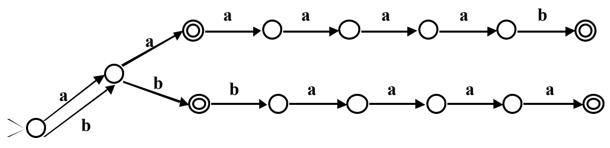

Figure 3: PTA after Merging

This FSA accepts strings that begin with bb, for example, whereas the PTA only accepts strings beginning with aa, ab or ba. Further merges can create loops in the graph. These can be traversed as many times as desired, adding extra symbols to accepted strings.

To recap, the situation is summarized in Figure 4. From I+ we can construct the PTA, which accepts exactly the strings in I+. Every deterministic FSA that satisfies the first two conditions above is a quotient of the PTA by some partition on the states of the PTA. Coarser partitions correspond to FSAs with fewer states, which in turn accept more general languages.

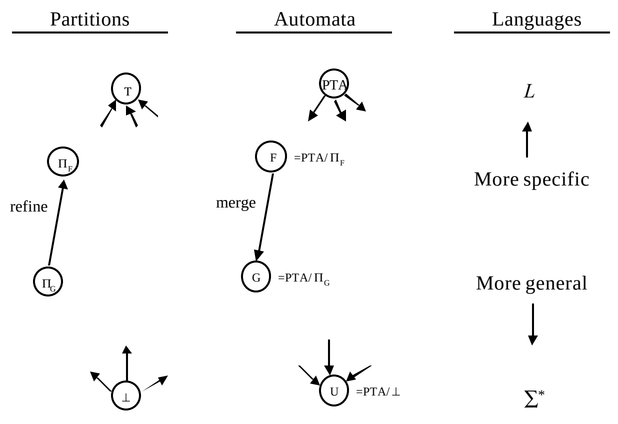

Figure 4: Three Views of the Search Space for Inductive Inference

The inductive inference problem can now been seen as a search problem over this lattice of FSAs, or equivalently over the lattice of partitions. If negative examples are given, these dictate that some potential solutions are infeasible. Assuming that the sample strings are generated from an FSA using some probabilistic process, we can infer not only the structure of the FSA, but also the parameters of the probabilistic process. In particular, we will assume that a probabilistic finite state automaton generates the sample strings.

When we find an FSA from a set of sample strings, we can estimate the transition probabilities by keeping a count of the number of times each arc of the graph is traversed by the strings. When states are merged, the transition counts on any merged arcs are added together. The counts can be converted into probability estimates by dividing each count by the total count of all the arcs from that state. In what follows, when we refer to an FSA, we will often mean an FSA with transition counts, or a probabilistic FSA, depending on the context.

5 SOME EXISTING ALGORITHMS

We know that we can find the target FSA by starting with the PTA, and merging states, so traversing down the lattice. But if we take this to the extreme, we end up with the single-state FSA, U, which accepts everything. The question that remains is – how much is too much merging? There are at least two distinct answers.

If negative samples are supplied, these can be used to stop us merging too far. If we merge too many states, the resulting FSA will accept one of the negative examples. An example of an algorithm of this type is the RPNI (regular positive and negative inference) algorithm (Oncina 1992). RPNI works by starting with the PTA, and merging pairs of states if possible, using a fixed depth-first ordering of state pairs. This algorithm runs in polynomial time and is guaranteed to identify the target FSA given extra completeness conditions on the sample data.

When there are only positive examples, merging must be controlled another way. The Alergia algorithm (Carrasco 1994) is similar to RPNI except that the role of negative samples in RPNI is replaced by a test of similarity of state behaviours. A second possibility when there are only positive samples, is to use Occam’s razor to select the FSA in the lattice that provides the “simplest” explanation of the data. This idea is behind a number of heuristic algorithms. The general pattern is to construct the PTA for the sample strings, and then to perform successive merges of states, seeking to optimize a “figure of merit” or simplicity measure.

In one of the earliest investigations of this kind, (Gaines 1976), Gaines describes ATOM, a system that used a dual-objective minimization criterion (number of states and an entropy-based measure) and looks for discontinuities of the minimal entropy value as the number of states is varied. Patrick and Chong (Patrick 1991) describe a greedy search algorithm that uses Minimum Message Length (MML) (Wallace 1968; Wallace 1983) as the measure of simplicity. This was later improved and adapted to non-deterministic FSAs by Raman et al (Raman 1998). Hingston (Hingston 2001) extended it to make use of negative samples. Stolcke et al (Stolcke 1994) describe a similar algorithm for HMMs. Grunwald (Grunwald 1996) used the Minimum Description Length (MDL) principle as formulated in (Rissanen 1982) to directly induce grammars, rather than FSAs, from positive samples. Both MML and MDL have been used as a model selection principle for a variety of induction problems.

A problem with these heuristic algorithms is that they suffer from local optima (Hingston 2001). Therefore, more powerful search methods are needed to tackle difficult problems. This suggests the use of genetic algorithms (GAs). The use of GAs or other evolution-based methods to evolve FSAs goes back quite a long way. In the context of artificial intelligence, L. Fogel proposed the use of evolutionary programming to evolve finite state transducers (a type of finite state machine with both inputs and outputs) that perform prediction tasks, as early as 1962 (see (Fogel 1995) for a discussion). In the grammatical inference context, Dupont (Dupont 1994a) describes GIG (Grammatical Inference by Genetic search), a GA for induction of FSAs from positive and negative samples, with a fitness function that minimizes the size of the FSA while penalizing FSAs that accept negative examples. Genetic search methods have also been proposed for induction of other classes of automata. For example, Lankhorst (Lankhorst 1995) used a GA for inducing pushdown automata from positive and negative samples.

In this paper, we describe a genetic algorithm for grammatical inference (GARI) based on minimizing MML.

6 MINIMUM MESSAGE LENGTH FOR FSAS

For the case of induction from positive samples alone or with limited numbers of negative examples, we need a suitable simplicity measure to guide the search. We use Minimum Message Length (MML), which is described below.

The motivation for this choice is that the description with the shortest optimal encoding provides the "simplest", and therefore the best, explanation of the observed data. Imagine the situation where we wish to communicate the data to another person, perhaps over a computer network. We send a message to the other person describing the data set. We want this message to be as short as possible.

The description consists of two parts: a description of the model (the FSA) and a description of the data using that model. At first, it may not be clear why the description of the model must be included. This is because it may be possible to achieve a very compact description of the data using a very complex model, which should not be considered to be a simple explanation. In the extreme case the model could just enumerate the data and no separate description of the data is needed at all! Requiring the description to include both the model and the data provides a trade-off between model complexity and accuracy.

So, given a data set, D, we seek an FSA, F, which minimizes the quantity:

$$DescriptionLength(F) + DescriptionLength(D|F)$$

,where each description is optimally encoded. How can we compute these description lengths? There are two possible ways – we could specify a particular coding scheme, or we could compute the probability of each event (since we know that the length of an optimally encoded description is *-log(p)*, the negative log-likelihood of the event occurring). We decided to specify an encoding scheme for F. This allows us to calculate the description length for F. We have seen that once the FSA is known, a probability distribution over the set of strings from which the data set is drawn is determined. This then lets us calculate the description length for D|F.

It is interesting to note that minimizing description length is equivalent to maximizing the a-posteriori probability of the model given the data. To see this consider the formula

$$prob(F|D) = \frac{prob(D|F)\ ^\prime\ prob(F)}{prob(D)}.$$

The denominator on the RHS is fixed by the data, so to maximize the LHS is to maximize the numerator on the RHS. Taking negative logs, we see that this is the same as minimizing the expression:

$$\begin{aligned} & -log(prob(F)\ x\ prob(D|F)) \\ & = -log(prob(F)) + -log(prob(D|F)) \\ & = DescriptionLength(F) + DescriptionLength(D|F) \end{aligned}$$

This is equal to the MML, as claimed.

Due to space limitations, we must omit the derivation of the MML formula. However, we can say that our description of the FSA lists the following

1. the number of states, N, in the FSA,
2. for each state j, t_j, the total of all transition counts leaving the state,
3. for each state j and symbol i, n_{ij}, the transition count for this symbol leaving the state,
4. for each state j and symbol i, if n_{ij} is not 0 (there is a transition for symbol i) and symbol i is not the delimiter, the description must specify the next state.

Taking into account all these items, the Minimum Message Length (MML) for the FSA may be estimated as

$$M' \log_2(N) - \log_2((N-1)!) + \sum_{j=1}^{N}(\log_2((t_j + V - 1)!) - \log_2((V-1)!) - \sum_{i=1}^{V}\log_2(n_{ij}!))$$

,where V is the number of symbols in the alphabet and M' is the number of non-delimiter arcs in the FSA.

7 THE GA

In this section, we describe our GA for regular inference, which we will call GARI (GA for Regular Inference). Genetic algorithms solve optimization problems by mimicking the essential processes of Darwinian natural evolution. A population of potential solutions is repeatedly subjected to (analogs of) natural selection based on fitness, reproduction, recombination and mutation, until a "sufficiently good" solution is found. The steps in a typical GA are:

1. Create an initial population of potential solutions;
2. Evaluate the fitness of each potential solution;
3. Select parents for the next generation of solutions based on their fitness;
4. Apply recombination (crossover) operators to create new solutions;
5. Apply mutation operators to promote variety;
6. Repeat steps 2-5 until done.

In some variations, the best few solutions from each generation are preserved intact in the next generation. This is called an elitist strategy. Sometimes crossover is only applied to a proportion of the selected pairs. In order to apply a GA to our particular problem, we must complete the missing details in the recipe given above.

Representation

The first detail to be completed is to design a "genome" to represent a potential solution. In (Dupont 1994a), the GIG method used partitions on the states of the PTA. Even though Dupont's algorithm searched over non-deterministic automata, and requires negative examples, we could have adapted his approach. However, after some experimentation, we settled on a different representation that can use more generic crossover and mutation operators.

Rather than use partitions, we use sets of pairs of states to be merged. Such a set of pairs of states determines a partition. We can represent the genome as a Boolean mask over the set of all pairs of states of the PTA. In nature, the genotype determines the physical form of the organism, the phenotype. In our case, the corresponding phenotype will be a quotient of the PTA determined by this set of merges, except that more states may be merged to ensure the FSA is deterministic. Thus it is possible for different genotypes to produce the same phenotype. Each member of the initial population is created by merging a randomly selected pair of states of the PTA.

Fitness

In natural evolution, "fitness" refers to the ability of an organism to survive long enough to reproduce. In GAs this is achieved using a calculated fitness value based on how good the solution is, and using this value in the selection process. In our GA, fitness is determined by the MML of the FSA. The fitness of each phenotype is defined by the formula

$$fitness = exp\left(-ln(2)\ x \left(\frac{MML}{PTA.MML} \right)^2 \right)$$

This expression is equal to 1 if the MML of the FSA is 0, 0.5 if the MML of the FSA is equal to that of the PTA, and otherwise, somewhere in between.

Selection mechanism

We use a common selection method, roulette wheel selection, in which parents are selected in proportion to their fitness. We also used elitism, preserving the fittest individual discovered, and reserving 5% of the new population for copies of this individual.

Crossover operator

Crossover is the process that, in sexual reproduction in nature, recombines genetic material from the parents in the children. While other choices could be tried, we used the standard one-point crossover operator. If there are negative examples, crossover of two feasible solutions can create an infeasible solution, and the crossover is not performed. Crossover probability was set at 0.8.

Mutation operators

Mutation in nature serves to introduce new genetic material into the gene pool, and is central to the creation of new species. In GAs, mutation operators make random changes to individual genotypes, and help to ensure that it is possible to explore the whole search space. We define two mutation operators, each of which simply flips one bit in the genome. The operators are:

1. merge-mutation: a zero bit is randomly selected and set to one.
2. split-mutation: a one bit is randomly selected and set to zero.

Note that merge-mutation aims to merge another pair of states, while split-mutation aims to undo a merge. If there are negative examples, merge-mutation can create an infeasible solution, in which case it is not carried out. One effective modification we made was to adjust the probabilities so that state pairs that come first in depth-first order are more likely to be merged than pairs that come later (by a factor of 10).

The two types of flips are separated so that the probabilities of merging versus splitting can be separately controlled. The probability of each type was set at 0.2.

8 RESULTS

In this section we describe some experiments comparing the performance of GARI with the RPNI algorithm and with GIG when positive and negative samples are given, and investigating the success of GARI when only positive samples are given.

Table 1: Tomita Regular Languages

	Description	# states in the target FSA
L1	a*	1
L2	(ab)*	2
L3	Not having odd number of b's then odd number of a's	4
L4	No more than 2 consecutive a's	3
L5	Even number of a's and even number of b's	4
L6	Number of a's and number of b's congruent modulo 3	3
L7	a*b*a*b*	4
L8	a*b	2
L9	(a*+c*)b	4
L10	(aa)*(bbb)*	5
L11	Even number of a's and odd number of b's	4
L12	a(aa)*b	3
L13	Even number of a's	2
L14	(aa)*ba*	3
L15	bc*b+ac*a	4

For these experiments, we used a standard set of 15 regular languages (Dupont 1994a; Tomita 1982). These languages are listed in Table 1.

8.1 POSITIVE AND NEGATIVE SAMPLES

Dupont used two data sets generated randomly from these languages. For the first data set, positive examples were randomly generated until a structurally complete sample was achieved, and the same procedure was used for the negative sample. Ten pairs of positive and negative samples were generated in this way. The second data set was generated the same way except that 3 times as many examples were generated for each sample. We used the same data sets for our first experiments.

Solutions were evaluated by finding the classification rates for all strings up to a certain length (for L9 and L15, this length was 7, for the rest it was 9), excluding the sample strings. Classification rates for strings in the language and for strings not in the language were calculated separately, then averaged.

Dupont used a population size of 100, maximum number of fitness evaluations of 2000, and ran the GA ten times for each pair of samples. For each pair, he then evaluated the performance of the solution having the minimum number of states. In this experiment, we used a population size of 200, maximum number of fitness evaluations of 5000, and ran our GA four times for each pair of samples, evaluating the performance of the solution having the smallest MML.

As well as comparing GARI with GIG and RPNI, we also used another algorithm, a beam search described in (Hingston 2001) and based on (Raman 1998). The beam search maintains a "beam" of solutions (say 3), starting with the PTA, and at each stage trying all possible merges of state pairs for each solution on the beam. Of the resulting new solutions, the 3 with the lowest MML are selected for the next stage. Eventually, no more merges are possible, and the solution with the lowest MML is the final result. Table 2 shows the classification rates achieved by the four algorithms using the second (larger) data set.

Table 2: Classification Rates for RPNI, GIG, BEAM and GARI

	RPNI	GIG	BEAM	GARI
L1	100	100	100	100
L2	96.6	100	100	100
L3	100	94.6	84.9	87
L4	90.4	81.2	72.5	76.6
L5	63.2	80.5	67.1	64.6
L6	89.7	95	74.9	86.7
L7	92.4	99.2	85.5	92
L8	100	100	100	100
L9	98.7	99.2	99.9	99.8
L10	96.4	96.6	79.5	83.7
L11	95.6	70.8	81.4	64.6
L12	100	99.8	100	100
L13	84.8	100	81.6	95
L14	98.7	99.8	95.2	95.2
L15	99.3	99.4	95	95
Mean	93.7	94.4	87.8	89.3

It is clear that the two algorithms that minimize the number of states (RPNI and GIG) generally outperform the two that minimize message length (BEAM and GARI) on this data set. GARI and BEAM are about equal, with GARI generally doing a little better except on L5 and L11.

The performance of GARI depends on two factors:

1. The effectiveness of MML as a fitness criterion, and
2. The effectiveness of the GA as a search method.

On the first point, we speculate that better performance might be obtained by developing a better MML formula using some prior knowledge about the data set (e.g. we know that the transition probabilities for each state are all equal). On the second point, better results can be achieved by running the GA for more generations. We have kept to only 25 generations here so as to be more directly comparable to GIG. We tested GARI on L5 and L11 using 4 runs of 100 generations, and obtained improved classification rates of 67.4% and 80.4%.

Speculating that the number of states may be a better criterion to use for these data sets, we ran additional tests using the same protocol as before, but with number of states as the fitness criterion for both BEAM and GARI. A slight problem in doing this is that there may be several solutions consistent with the positive and negative samples having the same number of states. For GARI, we resolved this by selecting the solution with the smallest MML among those with the least number of states. For BEAM search, merges were tried in the same depth-first order used by RPNI, and the first solutions found using this ordering were kept. Table 3 shows the results.

This time the performance of both algorithms is about equal with that of RPNI and GIG. Once again, GARI had difficulty with L5 and L11, and once again, performance was improved using 100 generations (73% and 75.6%).

Table 3: Performance of BEAM and GARI when Minimizing Number of States

	GIG	BEAM	GARI
L1	100	100	100
L2	100	100	100
L3	94.6	95.3	94.6
L4	81.2	98.9	84.1
L5	80.5	69.9	66.1
L6	95	82.8	89.1
L7	99.2	100	96.2
L8	100	100	100
L9	99.2	98.8	99
L10	96.6	91.9	92.2
L11	70.8	86.2	67.4
L12	99.8	100	100
L13	100	94.9	100
L14	99.8	98.7	98.7
L15	99.4	99.3	99.7
Mean	94.4	94.5	92.5

8.2 POSITIVE SAMPLES ONLY

To test GARI's performance with positive samples only, we once again used data sets randomly generated from L1 to L15. Note that GIG does not work without negative samples, so we compare GARI with BEAM. Generally, larger data sets are required when there are no negative samples. We used a process of trial and error to determine how many positive example strings to generate for each language. In each case we generated sufficient examples so that the MML for the target automaton was less than that of the single state automaton. 10 samples were generated for each language.

Table 4 shows the performance of BEAM and GARI (population = 200, generations = 100) on these data sets. The figures shown are averages for the 10 samples, using only a single run of the GA on each sample. For each algorithm we show the average MML as a percentage of the MML of the target automaton, and the average classification rate. The two algorithms are roughly comparable. Both had difficulty with L5, L6 and L7, and GARI also had trouble with L3 and L11. Notice that in these cases, the %MML is high, showing that the problem lies with the search, rather than with the MML criterion. Often the search converged early on the single state automaton.

Table 4: Performance of BEAM and GARI with Positive Examples Only

	#	BEAM		GARI	
		%MML	%class	%MML	%class
L1	10	100	100	100	100
L2	10	100	100	100	100
L3	160	100.6	84.9	101.5	69.7
L4	150	99.9	89.2	100	79.6
L5	30	102.2	64.9	102.4	62.4
L6	20	102.9	50	102. 8	50
L7	300	100.3	70	100.6	50
L8	10	100	100	100	100
L9	50	100.2	99.7	100	99.8
L10	30	99.9	99.8	100.5	99. 6
L11	30	102.3	71	103.7	63
L12	10	100	100	100	100
L13	20	102.2	70	100.8	90
L14	20	99.9	100	99.9	100
L15	10	100	100	100	100
Mean	-	100.8	86.6	101.1	84. 3

8.3 CONCLUSION

We have introduced a genetic algorithm, GARI, for the induction of regular languages from positive samples, or positive and negative samples. The algorithm seeks to optimize a measure of model simplicity (MML). The search had difficulty finding the optimal solution with some data sets, often converging instead to the degenerate single state automaton. Further work to overcome this early convergence may improve on these results. Another possible improvement is to reformulate the MML formula using additional prior knowledge about the target automata. In spite of these provisos, the performance of the GA is comparable to that of existing methods. The performance of this algorithm suggests that the strategy of using a genetic algorithm to minimize message length may be applicable to other problems of grammatical inference. We intend to investigate this in future work.

References

Carrasco, R. C., & Oncina, J. (1994). "Learning stochastic regular grammar by means of a state merging method." *The Second International Colloquium on Grammatical Inference (ICGI'94)*, Alicante, Spain, 139-152.

Dupont, P. (1994a). "Regular grammatical inference from positive and negative samples by genetic search: the GIG method." *Second International Colloquium on Grammatical Inference (ICGI'94)*, Alicante, Spain, 236-245.

Dupont, P., Miclet, L., & Vidal, E. (1994b). "What is the search space of the regular inference?" *Second International Colloquium on Grammatical Inference (ICGI'94)*, Alicante, Spain, 25-37.

Fogel, D. (1995). "Fogel: Evolutionary Programming." Evolutionary Computation, IEEE, NY, 75-84.

Gaines, B. R. (1976). "Behaviour/structure transformation under uncertainty." *International Journal of Man-Machine Studies*, 8, 337-365.

Grunwald, P. (1996). "A Minimum Description Length Approach to Grammar Inference." Connectionist, statistical and symbolic approaches to learning for natural language processing, G. S. S. R. Wermter, E., ed., Springer-Verlag, Berlin, 203-216.

Hingston, P. (2001). "Inference of Regular Languages using Model Simplicity." *Australian Computer Science Conference*, Gold Coast, 8 pp.

Lankhorst, M. (1995). "A Genetic Algorithm for the Induction of Pushdown Automata." *International Conference on Evolutionary Computation*, Perth, Western Australia.

Oncina, J., & Garcia, P. (1992). "Inferring regular languages in polynomial update time." Pattern Recognition and Image Analysis, N. Perez et al, ed., World Scientific, 49-61.

Patrick, J. D., & Chong, K.E. (1991). "Real-time inductive inference for analysing human behaviour." *Paper presented at the International Joint Conference on AI (IJCAI'91), Workshop number 6 on Integrating AI into Databases*, Sydney.

Raman, A., Andreae, P. & Patrick, J. (1998). "A Beam Search Algorithm for PFSA Inference." *Pattern Analysis and Applications*, 1, 121-129.

Rissanen, J. (1982). "A universal prior for integers and estimation by minimum description length." *Annals of Statistics*, 11, 416-431.

Stolcke, A., and Omohundra, S. (1994). "Best-first Model Merging for Hidden Markov Model Induction." *TR-94-003*, International Computer Science Institute, Berkeley, CA.

Tomita, M. (1982). "Dynamic construction of finite-automata from examples using hill-climbing." *The 4th Annual Cognitive Science Conference*, 105-108.

Wallace, C., & Boulton, D. (1968). "An information measure for classification." *Computing Journal*, 11, 185-195.

Wallace, C. S., & Georgeff, M.P. (1983). "A general objective for inductive inference." *TR 32*, Monash University, Department of Computer Science.