2001

Inference of regular languages using model simplicity

Philip Hingston
Edith Cowan University

10.1109/ACSC.2001.906625

Inference of Regular Languages using Model Simplicity

Philip Hingston
Edith Cowan University
p.hingston@ecu.edu.au

Abstract

We describe an approach that is related to a number of existing algorithms for the inference of a regular language from a set of positive (and optionally also negative) examples. Variations on this approach provide a family of algorithms that attempt to minimise the complexity of a description of the example data in terms of a finite state automaton model.

Experiments using a standard set of small problems show that this approach produces satisfactory results when positive examples only are given, and can be helpful when only a limited number of negative examples is available. The results also suggest that improved algorithms will be needed in order to tackle more challenging problems, such as data mining and exploratory sequential analysis applications.

Keywords: grammatical inference, Minimum Message Length principle

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 describe an algorithm framework encompassing a number of existing approaches to the problem, and show, through examples with a standard set of benchmark problems, how these algorithms perform.

There are two main classes of algorithms for grammatical inference, depending on whether there are both positive and negative examples from the target language, or positive examples only. In the former case, "exact" methods are available, with guaranteed convergence, provided sufficient examples of the right kind are given. In the latter case, "heuristic" methods must be employed. While some of the more abundant applications are of the second kind, heuristic algorithms have not received as much attention as exact ones to date. One of the aims of the work presented here is to start to understand the strengths and weaknesses of the heuristic search approach to inference.

The structure of the paper is as follows. We first review the relevant theory of finite state automata and regular languages, and then describe the induction problem and summarise the main approaches to its solution 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 describe the basic algorithm and its extension to include negative examples, and note some possible improvements. Finally, some experimental results on the performance of the algorithm are presented.

2. Theory of FSA

In this section we review the basic theory of finite state automata and their relationship with regular languages. A *finite state automaton* (FSA) is a quintuple $A = \langle Q, S, q_0, F, next \rangle$, where Q is a finite set of states, S a finite set of symbols, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of final states, and $next: Q \times S \rightarrow Q$ is a partial mapping called the *state transition function*. The elements of *next* are called transitions. If the mapping is deterministic (i.e. if there is at most one transition for a given Q,S combination), then A is also said to be deterministic. In this paper we consider only deterministic FSA's.

An FSA, A, models a process that begins in the start state, and at each step chooses a transition $(q, s) \rightarrow q'$ where q is the current state, outputs the symbol s, and moves to state q'. The process can stop at any final state. In the course of this computation, a string of symbols is output. The set of such strings is the language *generated by A*. One can also view the symbols as inputs: A begins in the start state, and at each step, the next input symbol determines the next state via the state transition function. If there is no transition from the current state with the given input symbol, the computation fails. If the computation succeeds, and finishes with A in a final state, then the string is *accepted by A*. The set of such strings is the *language accepted by A* (and is the same as the language generated by A).

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 L, using a state to represent each unique prefix of one of the strings.

We can illustrate this with an example from Gaines, (cited in [1] as originally due to Feldman et al.). The set of strings is $L = \{CAAAB, BBAAB, CAAB, BBAB, CAB, BBB, CB\}$. The PTA of L 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.

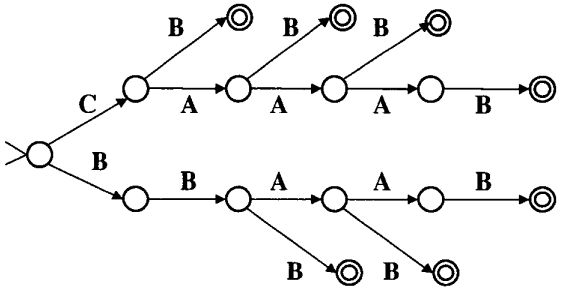

Figure 1. Example Prefix Tree Acceptor

This FSA represents the language L exactly, but does so somewhat wastefully – there are smaller FSA's that accept L. Given any FSA, Nerode [2] showed how to construct a minimal deterministic FSA that accepts the same language as the given FSA (minimal in the sense of having the least number of states). This is sometimes called the *canonical* FSA of that language. In this example, the canonical FSA is shown in Figure 2.

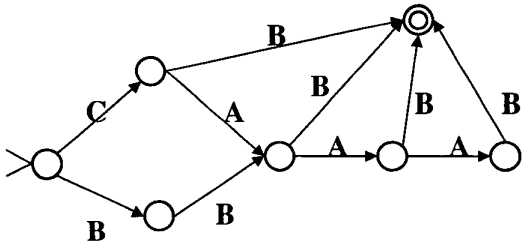

Figure 2. FSA with least number of states (canonical FSA)

Note that the FSA in Figure 2 is obtained from the one in Figure 1 by "merging" states together – all the final states have been merged into one, and three other pairs of states, corresponding to the prefixes $\{CA,BB\}$, $\{CAA,BBA\}$, and $\{CAAA,BBAA\}$ have also been merged. These two FSA's share three properties:

1. They accept the strings in L.
2. Every transition is used in accepting some string of L (and therefore all states are accessible).
3. They do not accept any strings that are not in L.

More generally, every (possibly non-deterministic) FSA that satisfies the first two conditions can be obtained by merging states of the PTA, and conversely [3]. Thus each such FSA can be identified with a partition of the states of the PTA. Successive merges correspond to composition of partitions. The FSA's thus form a lattice, and the deterministic ones form a sub-lattice. The PTA is 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.

At the bottom of the lattice is a single-state FSA. This FSA accepts not only L, but also any string over the same alphabet. In fact, as we move down the lattice, the set of languages accepted by the FSA becomes less restrictive (i.e. more general). To illustrate this, consider what happens if we merge the two states that follow the start state in Figure 2. 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.

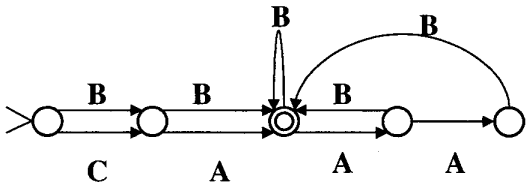

Figure 3. Canonical FSA after further merging

This FSA accepts strings that begin with BA, whereas the canonical FSA only accepts strings beginning with BB, CA or CB. Also, there are now several "loops" in the graph. These can be traversed as many times as desired to add extra symbols to accepted strings.

Finite languages are of theoretical interest, but the practical applications listed earlier concern infinite languages. In the general case, we have a finite sample of strings from some infinite language and are interested in identifying the underlying language, i.e. finding an FSA that accepts (or generates) the language.

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 (also sometimes called a stochastic finite state automaton) generates the sample strings.

Next we will define what we mean by a probabilistic FSA, but we first need to introduce a technical device that makes the description easier. Rather than specifying a set of final states, F, as before, 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. Now we are ready for the definition of a probabilistic FSA.

A *probabilistic finite state automaton* is formally a quintuple $<Q, S, q_0, next, p>$, where the first four elements are as before, and $p: Q \times S \rightarrow (0,1]$ is a partial mapping which is defined whenever *next* is defined, and where the sum of the values of p for any given state is 1. We interpret p as giving the probability that a particular symbol will be emitted when we are in a particular state (we call these *transition probabilities*). Once the output symbol is chosen, *next* determines the next state from that symbol. Figure 4 shows one possible set of transition probabilities for the FSA in Figure 2. Notice the "transition" from the final state using the delimiter symbol. This is the reason for introducing the device – the probability of ending the string can now be treated in the same way as the probabilities of transitions between states. A probabilistic FSA determines a probability distribution over the strings in the language – the probability for a particular string is the product of the probabilities of the transitions that generate it.

Note that probabilistic FSA's are essentially a special case of discrete output first-order Hidden Markov Models (HMM's), which have been used extensively in applications such as speech recognition (see, for example, [4]). We do not require the full generality of HMM's for our target applications, and FSA's are much more tractable.

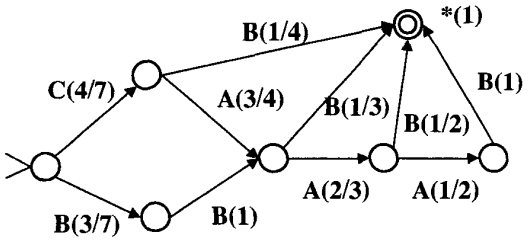

Figure 4. FSA with transition probabilities

When we construct 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 has been traversed. When states are merged, the transition counts on the 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. Doing this with the FSA in Figure 2 yields the probabilistic FSA in Figure 4. 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.

3. The induction problem

As was stated earlier, we are concerned with the problem of identifying a regular language from a finite sample of strings in the language, i.e. with finding an FSA that accepts the language. For example, in the case of exploratory sequential analysis, we are given coded behaviour sequences, and we are interested in finding out something about the underlying processes that are producing the behaviours. In speech recognition applications, we are given examples of strings of phonemes representing spoken words to be recognised, and we want to find models that enable us to recognise these words. In data mining, we might, for example, have a data set of sequences of credit card transactions, and might want to derive models that will help to detect possible fraudulent activities.

In the grammatical inference tradition, we usually also have a set of negative examples – 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 FSA. These additional data are needed if one wishes to identify the target FSA *exactly*. In the applications listed above, it is more likely that we will have a given set of positive examples only. In this case, the target FSA can only be approximately identified, and heuristic methods come into play.

In each case, if the underlying language is infinite, then an FSA that accepts only the finite sample (e.g. the canonical FSA) cannot be the right model! Supposing that we already have an FSA that accepts the sample strings (say the PTA or the canonical FSA), there are two ways to make it accept more strings. We could add transitions that are not used in the sample strings, or we could merge states, creating alternate paths and loops in the state graph. Since we have no rational basis for hypothesising extra transitions, we usually assume that the sample is large enough for all the necessary transitions to be represented (the sample is then said to be *structurally complete*).

Looking at the second possibility, we know that we can expand the language accepted by an FSA by merging states, so traversing down the lattice. If we keep merging, we will eventually reach the single-state FSA, so we want to stop before then. The question is - when should we stop merging? There are at least two distinct answers.

If negative examples are supplied, these can be used to stop merging too far. If we merge too many states, the resulting FSA will accept one of the negative examples. An algorithm of this type is the RPNI (*regular positive and negative inference*) algorithm [5]. RPNI starts with the PTA, and merges pairs of states if possible, using a fixed depth-first ordering of state pairs. It runs in

polynomial time and is guaranteed to identify the correct FSA given completeness conditions on the sample data.

The second possibility, when only positive examples are supplied, is to use a formulation of 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 algorithms. The general pattern is to construct the PTA for the sample strings, and then to perform successive merges of states, seeking to optimise a "figure of merit" or simplicity measure.

In one of the earliest investigations of this kind, [1], 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. He studied data sets from a range of applications, showing the ability of ATOM to identify various kinds of structure inherent in the data.

Patrick and Chong [6] describe an algorithm that uses Minimum Message Length [7, 8] as the measure of simplicity. The algorithm was implemented as part of a system for recording, coding and analysing behaviours from videos. It was later generalised and improved by Raman et al [9]. In that version, FSA's are allowed to be non-deterministic. Hingston and Lees [10] implemented an improved version of Patrick and Chong's algorithm and Lees has used it to analyse data from a vocabulary learning experiment [11]. Stolcke et al [4] describe a similar algorithm for HMM's. The Alergia algorithm [12] is similar to both RPNI and these simplicity-based algorithms, where the role of negative examples in RPNI (in preventing merges) is replaced by a test of similarity of behaviours of states. Grunwald [13] used the Minimum Description Length principle as formulated in [14] to induce grammars rather than FSA's, from positive samples. Both MML and MDL have been used as model selection principles for other induction tasks.

4. Minimum Message Length for FSA's

For the case of induction from positive examples, since a simplicity measure will be used to guide the search down the lattice, a suitable measure must be chosen. We have chosen to use the Minimum Message Length (MML), which is described below.

The motivation behind 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 minimises the quantity:

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

where the each description is optimally encoded. How can we compute these description lengths? There are two possible ways – specify a particular coding scheme, or compute the probability of each event (then the length of an optimally encoded description is $-log(p)$, the negative log-likelihood of the event occurring). We designed an encoding scheme for the FSA's, so as to calculate the description length for F. Once the FSA is specified, a probability distribution over the set of strings from which the sample is drawn is determined, (as in Figure 4), and we can calculate the second description length.

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

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

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

$$-log(prob(F) \times prob(D|F))$$
$$= -log(prob(F)) + -log(prob(D|F)$$
$$= DescriptionLength(F) + DescriptionLength(D|F).$$

4.1 Coding Scheme

Here we will describe our coding scheme. A 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.

Finally we want to encode the data. This can be done by beginning at the start state, and traversing the FSA, encoding the symbol to be selected for each transition at each step. As each symbol is emitted, the transition counts can be decremented. Assuming that an optimal code is selected based on the probabilities when each symbol is emitted, the total message length required for symbols emitted from state j can be calculated.

The detailed derivation of the final formula is omitted here, but the result is:

$$\sum_{j=1}^{N}(t_j + log((t_j + V - 1)!) - log((V - 1)!) - \sum_{i=1}^{V} log(n_{ij}!)) + N - log(N!) + Mlog(N),$$

where V is the number of symbols in the alphabet, and M is the number of non-delimiter arcs in the FSA. Notice that the formula is written so that the additive contribution of each state can be separated from that of other states.

5. The induction algorithm

We first introduce the basic induction algorithm for the case of positive examples only. We then show how it is modified to handle positive and negative examples.

The basic algorithm is a greedy search over the lattice of FSA's by performing successive deterministic merges of pairs of states, starting with the PTA of the (positive) example set. Being greedy, the search is incomplete, and may not find the optimal solution. In pseudo-code, the basic algorithm can be written as in Figure 5.

```
function Greedy(example set): FSA;
begin
    b := PTA(example set);  t := BestMerge(b);
    while FOM(t) < FOM(b) do begin
        b := t; t := BestMerge(b);
    end;
    Greedy := b;
end;
```

```
function BestMerge(FSA: f): FSA;
begin
    b := f;
    for each s1,s2 in States(f) do begin
        t := DeterministicMerge(f, s1, s2);
        if FOM(t) < FOM(b) then b := t;
    end;
    BestMerge := b;
end;
```

```
function DeterministicMerge(FSA: f; State: s1,s2): FSA;
begin
    d := Merge(f, s1, s2);
    while d is not deterministic do begin
        find a state with two transitions for the same
        output, but different next states q1 and q2;
        d := Merge(d, q1, q2);
    end;
end;
```

Figure 5. Greedy search with positive examples only

DeterministicMerge can be implemented using a stack of pairs of states that need to be merged, and using a version of the Union-Find algorithm, (see, for example, [15], pp 236-245), to represent and update the partition that is being created. The pair of states to be merged is pushed onto the stack initially. When a pair of states is popped, they are merged (if not already merged), and more state pairs may be pushed onto the stack if needed to keep the merged state deterministic.

In the case where only positive examples are available, if the figure of merit is Minimum Message Length, then the result of greedy search is an FSA that is at a local optimum of the MML, which we hope is also a global optimum or close to it.

5.1 Including negative examples

When negative examples are provided, merging a pair of states can result in an FSA that accepts one of the negative examples. In this case, we say that the FSA and the negative examples are *inconsistent*. We modify *BestMerge* by adding a consistency check after the merge. This prevents the FSA from collapsing too far, and the result will be an FSA that is at a local minimum of message length, number of states or transitions, depending

on the figure of merit chosen, among FSA's that are consistent with the negative examples.

6. Improving greedy search

In this section, we discuss some enhancements to the basic algorithm. One dimension of improvement is completeness of the search. Raman et al, in [9], modified the algorithm by replacing greedy search with a beam search, and extending the search space to non-deterministic automata. Beam search keeps several candidate FSA's at each step in the search, rather than only the best (a beam rather than a point). This makes the search more complete, but much slower! We have implemented a version of beam search that works with deterministic FSA's.

A second dimension of improvement is execution speed. We have implemented several methods that improve the speed of the algorithm. These can be used with search methods that merge states, such as greedy and beam search and RPNI.

6.1 Incremental calculations

Lazy and incremental calculations can be used to good effect to improve the efficiency of the basic algorithm. For example, the formula for the MML of an FSA is expressed as the sum of a global part and a contribution from each state. These state contributions only need to be recalculated when each particular state is modified, thus making the calculation of the overall MML much faster. In fact, when a candidate pair is being tried, it is not necessary to actually carry out the merge. It is possible to calculate the change in the figure of merit that would result from the merge by manipulating only the states involved in the merge (or copies of them). Only when it is known that the change in figure of merit is good enough do we actually carry out the merge. This is a large saving when the FSA has many states.

6.2 Quick consistency checking

When negative examples are provided, some merges will be rejected because the merged FSA would accept one of the negative examples. To avoid doing the merge in these cases saves a lot of needless computation. Therefore, a quick way to check for this beforehand is worth doing.

One way to do a quick check relies on the following observation: If at any stage the merge of a pair of states is rejected, then the merge will also be rejected at any time from then on (even if more transitions are added to the FSA, or more states are merged). This applies not only to the original pair of states, but also to any states that they are merged into. So it is worthwhile remembering when a merge is rejected. We will say that such a pair of states is *inconsistent* with each other with respect to the sets of positive and negative examples. We implemented this by keeping, for each state, a list of states known to be inconsistent with it. When a pair of states is merged, the merged state inherits the union of its parents' known inconsistent states. In particular, a state cannot be allowed to be inconsistent with itself, which provides another quick check – if the known inconsistent states for two states intersect, then those two states are themselves inconsistent (and can be added to each other's list of known inconsistent states).

7. Results

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

For these experiments, we used a standard set of 15 regular languages [16]. These languages are listed in Table 1.

Table 1 - Tomita regular languages

	Description	# states in the canonical FSA
L1	a*	1
L2	(ab)*	2
L3	Not ending in odd number of b's then odd number of a's	4
L4	Not having 3 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

7.1 Positive and negative examples

For each language, we used canonical FSA's to generate a random sample of 100 positive examples and 100 negative examples. When generating positive examples, we assigned equal probability to each possible

symbol that a given state may emit. A different choice of probabilities would lead to different results for the greedy search algorithm, which models these probabilities, whereas RPNI does not. We then used these samples to induce FSA models, using RPNI and greedy search with MML as the figure of merit. If the target FSA was identified, we counted this as a success. If not, we generated another random sample and used it to test the FSA, recording the percentage of positive examples accepted by the FSA (TP), and the percentage of negative examples rejected by the FSA (TN). We repeated this procedure 100 times for each language. Table 2 shows the number of times the target was correctly identified by each algorithm, and the mean values for TP and TN for those trials that *did not* correctly identify the target.

Generally, both RPNI and greedy search correctly identified the target most of the time, and both algorithms achieve good classification performance even when the target was not identified. Each algorithm clearly outperformed the other on particular target languages (these are highlighted in Table 2). It is instructive to consider why this occurred in each case. (Incidentally, in the experiment described above, RPNI was generally about 3-4 times faster than greedy search.)

Table 2 - Accuracy of RPNI and greedy search with positive and negative examples

	RPNI			Greedy MML		
	%	TP	TN	%	TP	TN
L1	100	-	-	99	100	100
L2	100	-	-	100	-	-
L3	92	93.1	99.4	52	96.8	98.4
L4	95	99.2	94.6	67	98.1	99.8
L5	100	-	-	100	-	-
L6	100	-	-	100	-	-
L7	97	93.3	98.6	79	99.3	98.9
L8	100	-	-	100	-	-
L9	0	99.9	98.8	81	99.8	98.7
L10	14	99.9	99.0	100	-	-
L11	100	-	-	100	-	-
L12	92	99.1	95.6	100	-	-
L13	100	-	-	100	-	-
L14	64	100	99	100	-	-
L15	49	99.9	99.1	100	-	-

Since RPNI is guaranteed to identify the canonical FSA given a sufficient sample (technically a *characteristic* sample), it is clear that the random generation procedure did not always provide such a characteristic sample. But in real applications, if the number of negative examples available is limited, it is not possible to know whether or not the sample is characteristic (without prior knowledge of what the target FSA is). It is interesting to note that greedy search sometimes correctly identified the target FSA in these cases, where RPNI did not. Using L9 as an example, we saw by inspection that FSA's found by RPNI accept either *CAB* or *ACB* or both, neither of which is in L9. So we know that the samples generated lacked negative examples of one or both of these types. Yet greedy search found the correct target. This was because the positive examples also lack strings of this type, reflected in a lower message length to the target FSA.

What about the languages with which greedy search had trouble? There are two ways in which greedy search can "fail" to find the target language – since the search is incomplete, it may fail to find the FSA with the lowest message length, or, it may be that the FSA with the lowest message length is not the target FSA. How can this happen, since we used the target FSA to generate the sample data? The explanation is that message length is related to the *probability* that the particular model (FSA) is the target one. Thus it is possible, though less likely, that a model other than the one with smallest message length is the target. Taking L3 as an example, we saw that in many cases the target FSA and the one found by greedy search had almost the same MML. Thus these two FSA's were almost equally likely given the sample data, and their behaviours were almost identical.

These results suggest that greedy search, or another method based on MML, can be a useful alternative to "exact" methods like RPNI when a limited number of negative examples is provided.

7.2 Positive examples only

As discussed earlier, when only positive examples are available, exact methods do not apply and a heuristic method like greedy search with MML is needed. To investigate how effectively a model simplicity criterion like MML replaces negative examples, we applied greedy search to the same example sets as in the first experiment, this time ignoring the negative examples.

Table 3 – Mean MML's found by greedy search with and without negative examples

	target	Greedy +/-	Greedy +
L3	877.16	880.71	869.81
L4	746.54	746.2	741.83
L5	2072.55	2072.55	2203.89
L6	1626.54	1626.54	1738.39
L7	762.3	762.88	751.38
L11	2615.20	2615.20	2790.5
L13	1182.58	1182.58	1253.15

For the languages L1, L2, L8, L9, L10, L12, L14 and L15, greedy search with positive examples only performed just as well as when negative examples were

given. However for the remaining languages, L3, L4, L5, L6, L7, L11 and L13, it failed to find the target on any trial, instead collapsing to a single-state FSA. Table 3 shows the average MML's for the FSA's found for these languages. For languages L3, L4 and L7, the single-state FSA had a lower message length than the target FSA. Thus for these languages, the single-state FSA is the more likely model given the data (even though it is the "wrong" one). We ran a smaller number of trials (just 5) for each of these languages, with a sample size of 500 rather than 100. With this sample size, the target FSA becomes more likely than the single-state FSA, but greedy search still converged on the single-state FSA. For L5, L6, L11 and L13, the target FSA has the lower message length, even with a sample size of 100. We tried beam search on these cases (again just 5 trials on each, with a beam width of 10), and found that beam search was able to identify the target, although it took several hundred times longer to run than greedy search. In this situation, an advantage of beam search is that it can return the best few FSA's that it finds, rather than just one. If several FSA's have similar MML's, then this is valuable additional information.

In summary, given enough samples, merging states to optimize the MML was capable of correctly identifying the target language. When more data was needed, a simple greedy search was not powerful enough to find the FSA with the optimal MML, and beam search was slow. This suggests that better, faster search algorithms will be needed to tackle larger problems.

8. Conclusion

In this paper we have described a family of search algorithms that can be used to induce finite state automata either from positive examples alone, or from a combination of positive and negative examples. The algorithms perform well even when only a limited number of negative examples is available. There are many potential applications in which only positive examples are available. Data mining applications such as analysis of sequences from user logs of web sites are a topical example. These applications will require algorithms that can handle large volumes of data, and large models. Using Minimum Message Length as a model-merging criterion is very effective with small problems, but search completeness and execution speed are issues that need further research before more challenging problems can be handled.

9. References

[1] B. R. Gaines, "Behaviour/structure transformation under uncertainty," *International Journal of Man-Machine Studies*, vol. 8, pp. 337-365, 1976.

[2] A. Nerode, "Linear automaton transformation," *Proceeding of the American Mathematical Society*, vol. 9, pp. 541-544, 1958.

[3] P. Dupont, Miclet, L., & Vidal, E., "What is the search space of the regular inference?," presented at Second International Colloquium on Grammatical Inference (ICGI'94), Alicante, Spain, 1994.

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

[5] J. Oncina, & Garcia, P., "Inferring regular languages in polynomial update time," in *Pattern Recognition and Image Analysis*, N. Perez et al, Ed.: World Scientific, 1992, pp. 49-61.

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

[7] C. Wallace, & Boulton, D., "An information measure for classification," *Computing Journal*, vol. 11, pp. 185-195, 1968.

[8] C. S. Wallace, & Georgeff, M.P., "A general objective for inductive inference," Monash University, Department of Computer Science, Technical Report 32, 1983.

[9] A. Raman, Andreae, P. & Patrick, J., "A Beam Search Algorithm for PFSA Inference," *Pattern Analysis and Applications*, vol. 1, pp. 121-129, 1998.

[10] P. F. Hingston, & Lees, C.D., "Sequential analysis with probabilistic finite state automata," University of Western Australia, Department of Computer Science, Perth, Western Australia, Technical Report 94/12, 1994.

[11] C. D. Lees, "Analyzing event sequences using Probabilistic Finite State Automata," *In Preparation*, 2000.

[12] R. C. Carrasco, & Oncina, J., "Learning stochastic regular grammar by means of a state merging method," presented at The Second International Colloquium on Grammatical Inference (ICGI'94), Alicante, Spain, 1994.

[13] P. Grunwald, "A Minimum Description Length Approach to Grammar Inference," in *Connectionist, statistical and symbolic approaches to learning for natural language processing*, G. S. S. R. Wermter, E., Ed. Berlin: Springer-Verlag, 1996, pp. 203-216.

[14] J. Rissanen, "A universal prior for integers and estimation by minimum description length," *Annals of Statistics*, vol. 11, pp. 416-431, 1982.

[15] E. Horowitz, & Sahni, S., *Fundamentals of Data Structures in Pascal*, Second Edition ed. Rockville, Maryland, USA: Computer Science Press, 1987.

[16] M. Tomita, "Dynamic construction of finite-automata from examples using hill-climbing," presented at The 4th Annual Cognitive Science Conference, 1982.