Incrementally Learning the Rules for Supervised Tasks: the Monk's Problems

Ibrahim KUSCU
Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QH
Email: ibrahim@cogs.susx.ac.uk

December 7, 1995

Abstract

In previous experiments [4] [5] evolution of variable length mathematical expressions containing input variables was found to be useful in finding learning rules for simple and hard supervised tasks. However, hard learning problems required special attention in terms of their need for larger size codings of the potential solutions and their ability of generalisation over the testing set. This paper describes new experiments aiming to find better solutions to these issues. Rather than evolution a hill climbing strategy with an incremental coding of potential solutions is used in discovering learning rules for the three Monks' problems. It is found that with this strategy larger solutions can easily be coded for. Although a better performance is achieved in training for the hard learning problems, the ability of the generalisation over the testing cases is observed to be poor.

Keywords: Supervised learning, hard learning, Monk's problems, hill climbing, genetic programming.

1 Introduction

In a previous paper [4] a genetic based encoding schema has been presented as a potentially powerful tool to discover learning rules for several simple supervised tasks. In another paper [5] the model is applied to more difficult supervised learning problems such as three Monks' problems and parity problems.

Combined with genetic algorithms the model can successfully produce evolution of learning rules. Rather than searching for a general learning algorithm (as in the work of Chalmers [1]), the aim is to see whether evolution would produce a specific learning rule for the problem in hand. The representation schema is very similar to the one used by Koza [3]. However, introducing prior knowledge into the representation of initial solutions using problem specific functions is minimal, if any at all. In this strategy potential learning rules are encoded as random mathematical expressions at variable lengths. The expressions are made up of random numbers and random variables. The variables are to be instantiated to input values of training set in a typical supervised learning. By using LISP's "EVAL" statement, the expressions are evaluated to certain numbers and by the help of a squashing-function this value is mapped to a value in the range of output values of the supervised task. The success of an expression in learning the task is determined by the number of correct mappings from the training set and by the degree of generalisation over the testing set.

The experiments showed that the encoding strategy and evolution are useful to discover or re-represent the problem-specific-functions describing the learning rules by using a relatively more general, fixed set of non-problem-specific functions. The model is also helpful in solving hard learning problems (i.e. in the context of this research hard learning problems are considered to be those supervised learning problems where the learning rule refers to the relationship *among* the input values rather than representing a direct correlation between value(s) of input(s) and output variable(s) [2] [7]) such as Monk 2 and parity problems. However, when the problems was larger and more complex (such as Monk2 and 5-bit and higher parity problems), the ability of the model in coding for good solutions and effectively generalising over the testing set was not found to be satisfactory.

In this paper these issues are investigated further. The experiments involve an incremental encoding of an expression as a potential solution. Although the basic encoding strategy (forming potential solutions as random mathematical expressions) is the same as the one used in previous experiments, the searching strategy is different. Rather than using evolution with a population of expressions, a hill climbing strategy is employed by incrementally forming a single expression. In this way it is hoped that problems requiring large solutions might be effectively coded.

In the sections that follow, I will first describe Three Monk's problems. The next section contains the representation strategy. Then the experiments and the results will be presented. Finally, I will conclude with a discussion and future research directions.

2 Three Monk's Problems

The three Monk's problems are used to compare the performance of different symbolic and non-symbolic learning techniques [8] including AQ17-DCI, AQ17-FCLS, AQ14-NT, AQ15-GA, Assistant Professional, mFOIL, ID5R-hat, TDIDT, ID3, AQR, CN2, CLASSWEB, ECOBVEB, PRISM, Backpropagation and Cascade Correlation.

Monk's problems involve classification of robots which are described by six different attributes. The attributes and their possible values are as follows:

ATTRIBUTES	VALUES
head_shape	round, square, octagon
body_shape	round, square, octagon
is_smiling	yes, no
holding	sword, balloon, flag
jacket_color	red, yellow, green. blue
has_tie	yes, no

Each of the three problem requires learning of a binary classification task. Whether the robot belongs to a particular class or not is decided based on the following rules:

```
Problem M1: (headshape=bodyshape) or (jacketcolor= red)


Problem M2: Exactly two of the six attributes have their  first
            value.
```

```
Problem M3: (jacketcolor = green and holding = sword) or
            (jacketcolor = (not blue) and bodyshape = (not octagon))
```

The most difficult one among these problems is the second problem since it refers to a complex combination of different attribute values and is very similar to parity problems. Problem one can be described by standard disjunctive normal form (DNF) and may easily be learned by all symbolic learning algorithms such as AQ and Decision Trees. Finally, problem three is in DNF form but aims to evaluate the algorithms under the presence of noise. The training set for this problem contains 5 percent misclassification.

The results of the comparison have shown that only Backpropagation, Backpropagation with decay, cascade correlation and AQ17-DCI had 100 percent performance on Monk 2 problem. However, the success of Backpropagation is probably is due to the conversion of original training set values into binary values which obviously this will directly effect the learning rule representing the true cases. The success of AQ17-DCI is clearly attributable to one of its function which tests the number of attributes for a specific value. Monk 1 and Monk 2 were relatively easy to learn by most of the algorithms.

2.0.1 Training and Testing Sets

The training and testing sets used for the experiment in this paper are the same as those used by Thrun in the performance comparison experiments. In these experiments two different sets are used. The first set adapted an original coding for the problems where each of the attributes would have one of the following values:

```
attribute#1 :   {1, 2, 3}
attribute#2 :   {1, 2, 3}
attribute#3 :   {1, 2}
attribute#4 :   {1, 2, 3}
attribute#5 :   {1, 2, 3, 4}
attribute#6 :   {1, 2}
```

Thus the rules describing the true cases can be reformulated as below:

```
MONK-1:
(attribute_1=attribute_2) or (attribute_5=1)

MONK-2:
(attribute_n = 1)
for EXACTLY TWO choices of n (n {1,2,...,6})

MONK-3:
(attribute_5  = 3 and attribute_4  = 1) or
(attribute_5 != 4 and attribute_2 != 3)
```

The second set of training and testing cases for the problems are the conversion of the original coding into the binary coding. Obviously, this has a direct effect on the rules describing the true cases and the formulation of the problems. The number of input variables increases from 6 to 17 since each possible value of the attributes is represented as 3 digit binary numbers where each digit represents the presence of a specific value of the attributes.

3 The Model

3.1 The Encoding Schema

The potential learning rules are encoded as simple mathematical expressions rather than bit representation. They are at variable lengths. The expressions are produced randomly involving random numbers (in some experiments real numbers and in others integers or the combination of the two has been tried) and a number of variables to be instantiated to the values of inputs from each pattern in the training set. The mathematical operators include plus, minus and multiplication (In addition to these MOD and division operators are also tried. Although their absence for the experiments to be described here did not show any noticeable difference, it reduced significantly the computational cost of processing individuals). A typical expression for a problem with two input values would look like this:

```
(((1 + *I1*) + (*I2* * *I1*)) - (( 0  - *I2*) - (*I1* * *I2*)))
```

This expression is randomly produced for a problem with two input values. *I1* and *I2* are the variables to be instantiated to the input values from the patterns at each time of evaluation.

When generating the expressions a variable parameter called *percentage* is used to impose how complex we want the expressions (i.e. longness or shortness of the expressions). It can have values from 0 to 100. The higher the percentage value the more complex the expression tends to be. In the experiments variable *percentage* values are used depending on the complexity of the problem (in the range of 75 to 85).

Internally each of the expressions are represented as trees. The typical structure of an expression would look like as in Figure 1.

```
The expression: (*I1* - ((*I2* + 1) * 0))
```

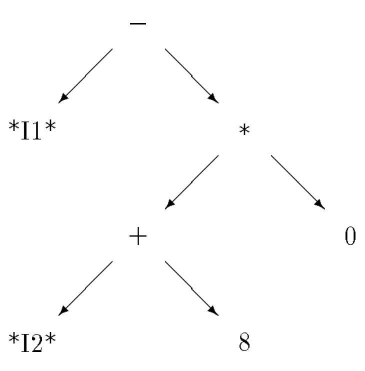

Figure 1: Tree representation of an expression.

In order to balance the behavior of the expressions (i.e. the bias toward positive expressions) half of the expressions are given a minus sign in front of them. This is to achieve, potentially an equal chance of producing negative and positive values when generating the expressions in the initial population.

3.2 Forming Incremental Representations

An expression as a potential solution to the problems is built incrementally as follows:

- Produce an expression randomly and test it over the training cases. Repeat this for a number of times and retain the one which produces the highest success(call it E1).
- Next, create another random expression (call it E2). Combine E2 with E1 by either (+) or (−) function and test the combined expression over the training cases. Repeat this for a number of times and retain the expression which produces higher success when it is combined with E1 than the success produced by E1 alone. (if a recently added expression does not contribute to the level of success, repeat creating and combining until an expression which contributes to the success is found or termination condition is reached).
- Then create a third random expression (call it E3) and combine it with the previous two by using either of (+/−) functions again and test on the training cases. Repeat this for a number of times and retain the expression which produces higher success when E3 is combined with E1 and E2 than the success produced by E1 and E2 combined only.
- Iterate above incremental process until a satisfactory level of success is reached by any combined expression and retain that expression as a potential solution.

Thus, in the experiments described here a modified representation of the expressions is used. This form of encoding introduces an explicit hierarchy to the representation of possible solutions. The general structure of any expression encoding for a possible solution would look like as follows:

```
(Squashing-Function ((E1) +/- (E2) +/- ...+/-(EN)))
```

A combined expression is evaluated and the value obtained is mapped to a value in the range of target outputs (between 0 and 1 in the case of MONKS problems) by using the following squashing function.

```
if value > 1 return 1
if value < 0 return 0
otherwise return the value
```

Each sub expression (E1, E2 and so forth) are random expressions which involve a composition of mathematical functions and the terminal units (each of these have exactly same structure as the ones used in the previous experiments). When the combined expression is evaluated, the sub-expressions are handled in two different ways. In the first one the value of the each sub-expression is mapped to a value in a specific range (between −2 and +2) by using another squashing function called LF2. LF2 behaves like logistic activation function used in [6] but the boundary values are −2 and +2. In the second way, each sub-expression is multiplied by a random number usually in the range of 0 an 1. In either way, the values of sub-expressions are kept in a certain smaller range and they can be used as an offset to each other to produce a variation in the overall value of the combined expression. In the letter case their values could contribute to the overall value of the whole expression in an excitatory or inhibitory manner.

4 Results

The performance of the model on the Monk's problems is tested using both the original coding of training and testing sets where attributes might have a value in the range of 1 to 4 and the binary coding of these original sets. In general, the model found satisfactory solutions by processing fewer individuals compared to the evolutionary method. The improved success can be attributed to the forced hierarchy in the representation as well as preserving characteristics of incremental encoding. It was observed that increase in the in the number of trials in generating expressions at the same level would not necessarily result in increased performance. This is especially true when a highly successful expression is generated earlier in the trial session. However, combining a new expression (i.e. forcing a hierarchy) would ensure an increased performance. Thus, combining expressions frequently would increase the speed of the search but result in longer expressions. In these experiments, usually 10 trials of subexpressions are employed before they are combined. Here are the best performances in percentage of correctly classifying the output.

BEST RESULTS
INCREMENTAL SOLUTION

	Original Coding		Binary Coding	
	Training	Testing	Training	Testing
MONK 1	92.9	94.5	88.8	92.2
MONK 2	85.8	79.0	96.6	83.9
MONK 3	98.1	95.1	97.2	95.7

As far as Monk 1 and Monk 2 are concerned, there are direct correlation between the input values and the required output. The results on these problems are satisfactory and very similar to the one found by the evolutionary approach.

However, performance on a hard learning problem such as the Monk 2 increased, especially, during the training process but a poor generalisation over the testing set is observed.

The model is also tested on parity problems. In this case, rather than using LF2 function, each sub-expression is multiplied by a random number. Up to 5 bit parity problems can be solved satisfactorily (over 90 percent) and by processing fewer individuals than it was necessary when evolution was used. In order to test the generalisation ability of the coding four and higher parity problems are trained over an incomplete parity mappings and then tested on the complete set. Incomplete set is constructed by randomly taking out some of the cases from the data set. The results have shown that in every run more than 90 percent success has been reached during the training but the performance on the testing was not satisfactory with a few exceptions. The exceptions constituted the cases where the missing tuples from the training set had a similar sequence of input-output values.

In general, the expressions are longer and more complex than those found by evolution but the performance on the hard learning problems are relatively higher both in terms of slightly higher learning and ensured coding ability. The followings are sample solutions obtained by the incremental approach. The row at the top of each solution indicates the name of the problem (i.e. M1O stands for Monk 1 original coding of input-output mappings and M2B stands for Monk 2 binary coding and so) the success level on the testing and the success level on the training set respectively.

4.1 MONK-1:

```
(M10 0.929011  0.944294

(- (- (- (+ (- (+
    (LF2 (+ (|%| (- *I4* *I6*) (- *I3* *I4*))
            (- (+ (* *I4* 3.49823)
                  (|%| (- *I6* *I3*)
                       (+ *I5* 2.82685))))))
(LF2 (- (* (- *I1* 1.27274) (* *I5* 2.4371)))
     ))
(LF2 (|%| (- *I2* *I1*) (|%| *I2* 3.55577))))
(LF2 (* *I3* *I6*)))
(LF2 (- *I3* *I3*)))
(- (LF2 (+ *I3* *I4*)) (LF2 (|%| *I5* *I1*))))
(- (LF2 (|%| (* *I6* *I2*)
   (|%| (- *I2* 4.9967) (- (- *I3* 2.54955)
   (* (* *I2* 3.11273) (|%| *I5* *I1*))))))
(LF2 (|%| *I5* 2.09487))))))
M10 6273)


(M10 0.914352 0.959677
(+ (- (+ (- (- (+
(* 0.509814 (|%| *I1* *I2*))
(* 0.193956
   (|%| (|%| (+ (- (|%| (|%| *I6* *I5*)
   (- (|%| *I3* *I4*)
   (|%| *I1* *I5*)))) (+ *I1* *I6*))
   (|%| *I4* *I4*)) (+ *I5* *I5*))))
(* 0.847709
   (- (* (- (+ *I3* *I2*) (+ *I1* *I3*))
   (- *I4* *I2*)))))
(* 0.793833
   (* (|%| (|%| *I4* *I5*)
   (* (- (|%| (+ *I5* *I5*) (* *I5* *I3*)))
   (* *I5* *I5*))) (|%| *I3* *I5*))))
(* 0.007444
   (* (- (+ (- *I1* *I1*) (- *I1* *I2*)))
   (- (|%| (- (* (- (* (- (+ (- (+ (- *I4* *I5*)
   (+ *I2* *I4*)))(* *I6* *I1*))) (* *I6* *I1*)))
   (* *I1* *I4*))) (|%| *I4* *I1*))))))
(* 0.481286
   (- (* (- *I5* *I2*) (- (* (- *I2* *I1*)
      (- *I2* *I1*)))))))
(* 0.283561
   (|%| (+ *I4* *I4*)
   (* (- (|%| (|%| (* *I5* *I3*) (* *I2* *I6*))
   (- *I1* *I2*)))
   (- (+ (- *I5* *I2*) (|%| *I5* *I1*)))))))
M10 35667))
```

```
 (M1B 0.888889 0.927419

(+ (+ (+ (- (- (- (+ (- (+
(- (LF2 (+ *I1* *I4*))
   (LF2 (* *I8* *I3*)))
(- (LF2 (|%| *I9* *I14*))))
   (LF2 (|%| *I15* *I14*)))
(LF2 (|%| *I6* *I11*)))
(LF2 (- *I14* *I4*)))
(LF2 (|%| *I10* *I14*)))
(LF2 (|%| *I6* *I3*)))
(LF2 (* *I2* *I6*)))
(LF2 (|%| *I13* *I3*)))
(LF2 (+ (|%| *I5* *I3*)
     (- (|%| (* *I4* *I2*) (|%| *I9* *I1*))))))))
```

4.2 MONK-2

```
(M20 0.790773 0.858852

(+ (+ (+ (+ (+
(- (LF2 (|%| *I3* 2.64081))
   (LF2 (|%| *I3* 4.44516)))
(- (LF2 (* *I3* *I4*)) (LF2 (|%| *I3* 1.30051))))
(- (LF2 (+ *I5* 0.621448)) (LF2 (* *I3* 1.20426))))
(- (LF2 (+ *I3* 3.8609)) (LF2 (* *I4* 1.08588))))
(+ (LF2 (- *I3* 3.18636)) (LF2 (* *I1* *I2*))))
(+ (LF2 (- (|%| *I6* 3.92829) (+ *I2* *I1*)))
(LF2 (+ *I6* *I4*))))))
 M20 1440))


(M20 0.773434 0.83432

(- (* (F (- (+ (* (F (+ (-
(* -0.396026
   (* (- (* *I5* *I3*) (* (|%| *I3* *I5*)
   (|%| *I5* *I3*))) (- *I3* *I6*)))
(* 0.623572
   (|%| (|%| *I3* *I4*) (- *I5* *I3*))))
(* 0.11661
   (|%| (|%| *I1* *I3*) (- *I5* *I6*))))))
(* -0.621351 (|%| *I3* *I4*)))
(* 0.674136
   (|%| (|%| *I4* *I6*) (* *I1* *I2*))))))
(* -0.150945
   (|%| (+ (|%| (+ *I4* *I5*) (- *I1* *I3*))
   (|%| *I2* *I5*)) (* (* *I5* *I4*)
   (* (* (- *I3* *I6*) (* *I3* *I1*))
   (|%| (- *I5* *I6*) (* (+ *I3* *I5*)
   (- *I3* *I4*))) )))))
 M20 17861))
```

```
(M2B 0.763958 0.834312

(- (- (- (+ (- (+ (- (-
(LF2 (- *I11* 0.410278))
(LF2 (|%| *I11* *I7*)))
(LF2 (* *I8* *I11*)))
(LF2 (* (|%| *I1* *I7*) (+ *I11* *I14*))))
(LF2 (* *I6* 0.018643)))
(LF2 (- (|%| (- *I10* 0.445533)
     (|%| (- (+ *I8* 0.177315) (- (- (|%| *I4* *I7*)
     (* *I2* 0.929654)))) (|%| *I13* *I7*))))))
(LF2 (* *I2* 0.015149)))
(LF2 (- (* (|%| *I7* *I15*) (|%| *I10* 0.33154)))))
(LF2 (- (- (|%| (- *I15* *I1*) (* *I5* *I14*))
     (- (|%| (|%| *I10* *I8*)
     (* (- *I3* 0.277117) (+ *I4* *I3*))))))))
  M2B 6066)


(M2B 0.839254 0.961007
(- (+ (+ (- (- (- (- (- (- (- (-
(LF2 (- *I11* 0.417749))
(LF2 (|%| *I7* *I11*)))
(LF2 (* (|%| *I7* *I13*)
     (|%| *I6* 0.025976))))
(LF2 (- (|%| (|%| *I11* *I1*)
     (- (|%| (+ *I11* *I15*)
     (+ *I15* *I12*)))))
     ))
(LF2 (* *I8* *I11*)))
(LF2 (- (* (* *I4* *I2*) (* *I9* *I11*)))))
(LF2 (- (|%| (* (- *I3* *I7*) (|%| *I12* *I9*))
     (|%| *I5* *I1*)))))
(LF2 (* *I8* 0.015916)))
(LF2 (* *I15* 0.032574)))
(LF2 (* (* *I3* 0.464306)
     (- (+ (- (+ (|%| *I5* *I4*)
     (|%| (- *I10* 0.882678)
     (* *I2* 0.098271))))
     (- (- (|%| *I13* *I8*)
     (- (* (* *I6* *I3*)
     (- (* (|%| *I10* 0.45645)
     (|%| *I12* *I5*)))))))))))
     )
(LF2 (- (- (* (* *I6* *I13*) (* *I6* *I10*)))
     (* *I13* *I11*))))
(LF2 (- (* (+ *I10* *I8*)
     (|%| (* (+ (* *I15* *I3*) (|%| *I8* 0.856525))
     (|%| *I7* *I10*))
     (+ *I1* 0.916587))))))
M2B 15699));;;
```

4.3 MONK-3

```
(M3O 0.981558 0.951792

(+
(- (LF2 (* *I2* 0.195083)) (LF2 (* *I1* 1.29916)))
(+ (LF2 (- *I5* 4.39331)) (LF2 (* *I1* *I4*))))
M3O 1049))


(M3B 0.972093 0.957318

(+ (+ (+ (+ (+ (+
(LF2 (- *I6* 0.181332))
(LF2 (|%| *I14* *I14*)))
(LF2 (* *I15* 0.405052)))
(LF2 (+ *I6* 0.18335)))
(LF2 (- (* *I2* 0.048406) (* *I6* 0.873582))))
(LF2 (* *I13* 0.054364)))
     (+ (LF2 (* (* *I2* *I13*)
        (- (|%| *I4* *I3*) (|%| *I10* *I5*)))))))
```

4.4 Parity Problems

4.4.1 2-BIT-PARITY

```
((1.0 (+ (-
(* 0.852533 (- (* *I1* *I2*))) (* -0.497317 *I1*))
(* 0.262082 *I2*))
P2 62))

(1.0 (- (+
(* -0.730121
   (- (* (* *I1* *I1*) (+ (- (|%| (- (+ *I1* *I2*))
   (- (- (* *I1* *I2*)) *I2*))) *I2*))))
(* -0.011222 (- *I2* (- (- *I2* *I1*)))))
(* -0.841233 (- (* (- (|%| *I1* *I2*) *I2*) *I2*))))
      P2 56))

(1.0 (-
(* -0.682833
    (- (- (* (- (* *I2* *I2*)) *I2*)
    (+ *I1* (- (* (+ *I2* *I2*) *I2*))))))
(* -0.838546 (- (* *I1* (- *I2* *I1*)))))
P2 3))

((1.0 (+ (-
(* -0.622615 *I2*) (* -0.454473 *I1*))
(* 0.896444 (- (|%| *I2* (- (- (- (* *I1* *I2*))
   (- *I1* *I2*)))))))
P2 26))
```

4.4.2 3-BIT-PARITY

```
((1.0 (+ (+
(* 0.782908
   (- (+ (+ (- (* *I2* *I3*) (+ *I1* *I3*))
   (* (* *I3* *I3*)
   (- (|%| (* *I2* *I3*) (|%| *I1* *I3*)))))
   (- (- (|%| (* *I3* *I2*)
   (- (|%| (- *I2* *I1*) (+ *I1* *I3*)))))
   (+ *I1* *I2*)))))
(* 0.944947
   (- (* (+ (- (- (* *I2* *I2*) (|%| *I2* *I2*)))
   (+ *I2* *I1*)) (+ *I3* *I1*)))))
(* 0.906881 (|%| (+ *I3* *I1*) (|%| *I3* *I2*))))
P3 632))

(1.0 (+ (- (- (+
(* 0.554274 *I1*)
(* -0.571618 (* *I1* (- *I2* *I3*))))
(* -0.63217 (|%| *I3* (- (- (* *I2* *I3*) *I3*)))))
(* 0.14732 (+ (- (- *I2* (* *I3* *I3*))) (+ *I1* *I1*))))
(* -0.649549 (* *I1* (|%| (* (- *I1* (|%| *I2* *I3*))
      (- (- (- (+ (* *I1* *I3*) *I3*)) *I3*))) *I3*))))
P3 727))
```

4.4.3 4-BIT-PARITY

```
((0.950411 (+ (+ (+ (+ (+
(* 3.07007 (LF2 (- *I4* *I3*)))
(* 2.86816 (LF2 (- *I2* *I1*))))
(* 9.55089
   (LF2 (- (* (- *I4* *I3*)
   (- (- (|%| (|%| *I3* *I1*) (+ *I3* *I1*))
   (|%| *I1* *I3*))))))))
(* 0.171425 (LF2 (- *I3* *I4*))))
(* 0.182571 (LF2 (* *I3* *I1*))))
(* 6.52985
   (LF2 (- (+ (- (* (|%| *I4* *I1*) (|%| *I3* *I4*)))
        (|%| *I2* *I4*))))))
   P4 9926))
```

4.4.4 5-BIT-PARITY

```
(0.9375 (- (- (+ (+ (- (- (+ (+ (+
(* 0.922868
   (|%| (- (- (|%| (- (+ (- (|%| (-
        (|%| (|%| *I4* *I1*) (- *I3* *I2*)))
   (* *I5* *I3*))) (- *I2* *I5*)))
   (|%| *I1* *I3*))) (+ *I1* *I3*))
   (- (- (+ *I2* *I3*)
      (- (+ (+ *I5* *I2*) (- *I1* *I4*)))))))
(* 0.081095
   (|%| (- (* (|%| *I3* *I2*) (|%| *I3* *I2*)))
   (- (+ (- *I4* *I5*) (+ *I2* *I5*))))))
(* 0.112516
(* 0.389813
   (- (|%| (- *I5* *I4*)
   (+ (* *I2* *I1*) (|%| (- (* (+ *I2* *I4*)
   (+ *I4* *I5*))) (- *I1* *I1*))))))))
(* 0.16357
   (- (* (- (- (+ *I5* *I1*) (- *I4* *I3*)))
   (- *I2* *I4*)))))
(* 0.216359 (- (* *I3* *I3*) (* *I3* *I2*))))
(* 0.340928
   (- (* (- (- (- (- (+ *I1* *I1*)
   (* (+ *I2* *I4*) (|%| *I1* *I1*))) )
   (- (|%| (|%| (- *I1* *I4*)
   (- (- (* *I3* *I4*) (|%| *I3* *I2*))
   (* *I4* *I4*))) (- *I1* *I2*)))))(* (- *I3* *I4*)
   (- (|%| (+ *I4* *I4*) (* *I5* *I5*))))))))
(* 0.910895
   (|%| (|%| *I1* *I5*)
   (* (- (|%| (* *I1* *I1*) (- *I2* *I4*)))
   (|%| *I3* *I2*)))))
(* 0.073151
   (- (- (- (* (|%| *I2* *I5*)
   (- (* (* *I1* *I4*) (* *I2* *I2*)))))
   (|%| (- (- (- (* (- *I3* *I1*) (- *I1* *I2*)))
   (|%| *I4* *I4*))) (|%| *I2* *I2*))))))
(* 0.470867
   (|%| (- (|%| (- (+ (- *I1* *I2*) (- *I4* *I5*)))
   (+ *I5* *I3*))) (- (+ *I3* *I1*)
   (- (- (- (- (* (- (* (|%| *I2* *I4*)
   (* *I5* *I3*))) (- (|%| (|%| *I3* *I2*)
   (|%| *I1* *I4*))))) (* *I2* *I1*)))
   (+ (- *I2* *I2*) (- (+ (- *I4* *I3*)
   (- *I1* *I1*)) (* *I3* *I4*))))))))
(* 0.425371
   (|%| (+ (|%| *I1* *I3*)
   (- (- (* (- (+ (|%| (- *I2* *I5*)
   (|%| *I5* *I4*)) (* *I4* *I2*)))
   (- *I2* *I2*))) (* *I4* *I4*))) (- *I4* *I2*))))
      P5 45451)
```

```
(0.90625 (- (+ (- (- (+ (+ (+ (+
(* 0.942357
   (|%| (* (* (+ (|%| *I5* *I1*)
   (- (- (+ *I2* *I2*) (|%| *I5* *I5*))))
   (- (* *I2* *I3*) (|%| *I5* *I1*)))
   (* *I3* *I3*)) (- (+ (* *I4* *I1*)
   (- (+ (- (- (+ (+ *I1* *I5*)
   (|%| (+ *I5* *I4*) (- *I4* *I5*))))
   (+ *I5* *I4*)) (* (- (* (- *I5* *I2*)
   (- *I5* *I5*))) (- *I1* *I4*))))))))
(* 0.532938
   (- (+ (- *I1* *I5*)
   (- (- (- *I4* *I3*) (|%| *I2* *I4*)) )))))
(* 0.72181
   (* (- (+ (- (|%| (+ *I1* *I2*)
   (- (+ (* (+ *I4* *I3*) (- *I4* *I2*))
   (- (- (|%| (|%| *I5* *I4*) (+ *I3* *I2*))
   (|%| *I5* *I1*)) )))))
   (* (|%| *I5* *I3*) (|%| (* *I4* *I2*)
   (- (* (* *I5* *I1*) (+ *I5* *I4*)))))))
   (|%| (+ *I1* *I1*)
   (- (+ (- *I1* *I1*) (* *I2* *I5*))))) ))
(* 0.04847
   (- (- (- (+ (* *I1* *I3*) (* *I2* *I2*)))
   (|%| *I5* *I3*)))))
(* 0.090138
   (- (* (- (+ (- (|%| (|%| *I5* *I1*)
   (- (- (- (* (|%| *I2* *I3*)
   (* *I5* *I4*))) (|%| *I4* *I3*)))))
   (- *I2* *I3*))) (- *I4* *I5*)))))
(* 0.045263 (- (|%| (+ *I1* *I4*) (- *I4* *I1*)))))
(* 0.995207
   (- (* (- *I1* *I5*) (* (* (- *I2* *I5*)
   (- (* (- (- (* *I2* *I1*)
   (- (|%| (* *I5* *I1*) (+ *I2* *I4*)))))
   (+ *I5* *I2*))))
   (* (+ *I5* *I4*) (|%| *I1* *I1*)))))))
(* 0.751767
   (|%| (- (* (- (- (|%| *I1* *I5*) (|%| *I4* *I1*)))
   (+ *I4* *I4*))) (- (* (- *I3* *I2*) (+ *I5* *I3*))))))
(* 0.612434
   (- (* (- (+ (- (+ (+ *I1* *I3*)
   (- (* (* *I2* *I5*) (* *I4* *I4*)))))
   (+ *I1* *I4*)))
   (- (|%| (+ (- (|%| (+ *I1* *I4*) (- *I5* *I1*)))
   (* *I2* *I5*))
   (- (+ (+ (- (* *I5* *I1*) (|%| *I5* *I4*))
   (* (- *I3* *I3*) (+ *I1* *I4*)))
   (|%| (* *I4* *I5*) (+ *I4* *I5*))))))))))
P5 34267)
```

5 Conclusions

The research described in this paper is aimed to see whether a hill climbing strategy with an incremental encoding of random mathematical expressions can improve the ability to code for the solutions and generalise over the testing set for simple and hard supervised learning tasks. Among the Three Monk's problems, solutions to Monk 1 and Monk 2 were found easily and satisfactorily. Although, the performance in learning the rule for Monk 2 increased compared to performance of the solutions found in previous experiments and was better than the performance of the most learning algorithms reported in [8], it was not as good as the performance of back-propagation. In all cases, during the training a solution with a performance higher than 90 percent found for Monk 2 but they all showed a poor generalisation over testing set. When tested with the parity problems, similar results are observed. The model showed increased performance in coding for up to 5 bit parity problems, but when tested on incomplete parity mappings it showed a poor generalisation ability. Thus, the encoding strategy together with hill climbing is useful in finding a solution for hard learning problems. However, it is not clear whether the problem of generalisation is attributable to the nature of the hard learning problems or the encoding strategy. In order to discover this, an analysis of the solutions to parity problems with incomplete data sets is being carried out. The experiments also showed that the number of individuals processed in finding a solutions are less than what is required by the evolutionary approach but solutions found are longer and more complex.

The experiments have shown clearly the advantage of incremental encoding. However, in this strategy sub expressions are produced randomly and far from being optimal in terms size and ability to generalise. In future experiments in order to enhance the incremental strategy evolution can be used to find better sub-expressions.

References

[1] D.J. Chalmers. Evolution of learning: an experiment in genetic connectionism. In Touretzky et al, editor, *Connectionist Models*. Morgan Kaufmann, 1990.

[2] A. Clark and C. Thornton. Trading spaces: Computation, representation and the limits of learning. Technical Report 291, COGS, University of Sussex, 1993.

[3] J. Koza. *Genetic Programming:On the programming of computers by means of natural selection.* MIT press, 1992.

[4] Ibrahim Kuscu. Evolution of learning rules for supervised tasks i: Simple learning problems. Technical Report CSRP-394, Uni. of Sussex, COGS, 1995.

[5] Ibrahim Kuscu. Evolution of learning rules for supervised tasks ii: Hard learning problems. Technical Report CSRP-395, Uni. of Sussex, COGS, 1995.

[6] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by back-propagating errors. *Nature'*, 323:533–536, 1986b.

[7] C. Thornton. Supervised learning of conditional approach: a case study. Technical Report 291, COGS, University of Sussex, 1993.

[8] S. Bala et al Thrun. The monk's problems - a performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, School of Computer Science, Carnegie-Mellon University., USA, 1991.