Evolution of Learning Rules for Supervised Tasks II: Hard Learning Problems

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

November 10, 1995

Abstract

Recent experiments with a genetic based encoding schema are presented as a potentially powerful tool to discover learning rules by means of evolution. The representation used is similar to the one used in Genetic Programming (GP) but it employs only a fixed set of functions to solve a variety of problems. In this paper three Monks' and parity problems are tested. The results indicate the usefulness of the encoding schema in discovering learning rules for hard learning problems. The problems and future research directions are discussed within the context of GP practices.

Keywords: Supervised Learning, Genetic Programming, Monk's Problems

1 Introduction

The main characteristic of a supervised learning is that the problem is always defined in terms of an input/output mapping. Actually, the target mapping contains a rule of some sort and the task of learning is to discover this rule and represent it in such a way that unseen inputs can be correctly mapped to the outputs. In the simplest form, the rule will be as such: "if the input variable(s) has/have such value(s) then the output variable(s) has/have that value(s) else

the input variable(s) .....". This means that there is a direct correlation between particular input values and particular output values.

However, sometimes the rule may not refer to particular values of variables. Rather it may refer to possible 'relationships' among input values. It has been shown by Clark and Thornton [2] that learning behaviors based on some training sets which take into account the relationship among values of the input variables can be extremely difficult (named as type-2 learning problems). In one of the studies [10] well-known learning algorithms such as ID3, back-propagation and classifier systems are tested on a type-2 problem and all showed poor results.

In a previous paper [6] an encoding schema has been presented and tested on several simple supervised tasks. Combined with genetic algorithms it 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. Although the representation schema is very similar to the one used by of Koza [5] in Genetic Programming (GP) paradigm, introducing prior knowledge into the representation of initial solutions using problem specific functions is minimal, if any at all. The main motivation to exclude problem specific functions is to see whether evolution can produce (i.e. discover) a learning rule which can, in some ways, represent those functions. In this strategy potential learning rules are encoded as random mathematical expressions at variable lengths and only four functions is allowed: plus, minus, multiplication and protected division. The terminal units can be 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. This value is then mapped to a value in the range of value of target outputs through a squashing function and is used to determine the success of a potential rule in correctly learning the supervised task.

In this paper several experiments where the model is applied to hard learning problems such as three Monk's problems and parity problems will be presented. In the sections that follow, I will first describe the Three Monk's problems. Next, I will introduce Genetic Programming (GP) in relation to the Monks problems. The following section contains the representation strategy and the process of applying genetic algorithms. Then the experiments and the results will be presented. Finally, I will conclude with a discussion and future research possibilities using the genetic based encoding schema.

2 Three MONK's Problems

The three MONK's problems are used to compare the performance of different symbolic and non-symbolic learning techniques [11] 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.

MONKS'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.

2.1 Genetic Programming

In the genetic Programming Paradigm of Koza [5] problems of Artificial Intelligence (AI) are viewed as the discovery of computer programs which produce desired outputs for particular inputs. The computer programs can be an expression, formula, plan, control strategy, decision tree or a model depending on the sort of AI problem.

He claims that solving AI problems requires searching the space of all possible computer programs for the fittest individual computer program. Genetic Programming (GP) is the method of searching for this fittest individual computer program based on Darwinian natural selection and genetic operations.

Genetic programming steps, as in the application of conventional Genetic Algorithms (GA), involve initialisation of random population of computer programs and for a number of generation, evaluating the fitness of the individual programs and applying genetic operators.

One of the important feature of the GP is that it uses variable length of genome (i.e. computer programs) which reflects hierarchical and dynamical aspects of the potential solutions to a particular problem. Since the shape and the size of the solution to a problem may not be known in advance, specification or restriction of the potential solutions to certain format may limit the search space so that it may be impossible to reach a solution. By moving from fixed length genotype to the adaptation of variable length genotype, GP improves the capabilities of conventional GA.

In GP the genotype (i.e. computer programs) is composed of a set of functions and terminal units appropriate to the problem domain. The set of terminals are either some variable atoms or some constants. The set of functions would include arithmetic operations, mathematical functions, programming operations, boolean operations or any other domain specific functions.

Consider the even-2-parity problem (not XOR). This problem requires that the output is true if an even number of inputs are true otherwise it is false. The possible set of functions for this problem would include AND, OR and NOT and the terminals would include D1 and D2 representing the input variables. The potential solutions would be represented as a composition of these functions and with the help of evolution probably, the following would be found as the solution to even-2-parity problem.

```
(OR (AND (NOT D0) (NOT D1)) (AND D0 D1))
```

GP has been successfully applied to quite a number of problems from several domains of AI. However, for each problem a different set of functions and terminals which are *appropriate* to the problem are used. In most of Koza's experiments these primitives are chosen carefully in order to (1) avoid failing to find a solution and (2) improve the performance in finding a solution.

The selection of the functions and terminals are guided by the *sufficiency* property which states that "the set of terminals and the set of primitive functions be capable of expressing a solution to the problem" [5] p.86. Since there is not an universal set of functions which is capable of solving every problem, the need for reducing the set of primitives to a minimally sufficient set seems justified. However, how to choose a minimally sufficient set remains an open question. In answering the question Koza focused on determining a set of functions which would yield a solution which is simple and elegant.

In this paper, I will focus on a different aspect of selection of primitive functions. Suppose that a particular function from a minimally sufficient set of functions for a particular problem can be defined by some relatively more general functions. For example, XOR can be defined by AND, OR and NOT. For a particular problem requiring XOR function in the composition of its solution,

typical GP practice would favor using XOR function since it would drastically facilitate finding a solution and the solution would be simple and elegant. Experiments in this paper aim to *discover* such specialised functions by starting the search with more general functions which can define the specialised functions.

In GP practice a typical function set for each of the Monks' problems would probably, at least, be as shown below in F function sets for each of the problem:

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

F = { EQUAL, OR, (possibly) TEST-ATTRIBUTE-FOR-A-VALUE }

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

F = { EQUAL, TEST-NUMBER-OF-ATTRIBUTE-FOR-A-VALUE, NOT, OR, AND}

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

F = { EQUAL, NOT, OR, AND}
```

For all of the three problems, I will use only protected division, multiplication, plus, minus and a squashing function which maps the value of an expression after it is EVALuted to a value in the range of 1 to 0 (see later discussion).

```
F* = { %,  *, +, -, SQUASHING }
```

In a previous paper, [6] I have shown that arithmetical functions together with SQUASHING function can define OR, AND and XOR and compared to these functions they are relatively more general. In this paper I will show that the encoding system using only the above five functions in F* is capable of coding learning rules for the monks problems which would typically require explicit introduction of the problem-specific-functions in the GP practice. Moreover, I'll attempt to prove that the strategy employed can be useful in finding solutions to hard learning problems such as Monk 2.

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 is represented as a tree. This structure is used as a basis of applying genetic operators: crossover and mutation. A random point in selected expression (tree) is chosen as crossover or mutation point. More details will be given about this later. The typical structure of an expression would look like as in Figure 1.

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 Genetic Algorithms

The Schema Theorem developed by Holland[4] based on genetic search has been proven to be useful in many applications involving large, complex and deceptive search spaces [3]. So genetic search is most likely to allow fast, robust evolution of genotypes encoding for potential learning rules as mathematical expressions. Using Genetic Algorithms (GA) the model is implemented in LISP. The top level structure of the system exhibits the following:

1. Initialize the population of expressions
2. Evaluate each expression and determine its fitness
3. Select expressions to reproduce more
4. Apply genetic operators to create new population

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

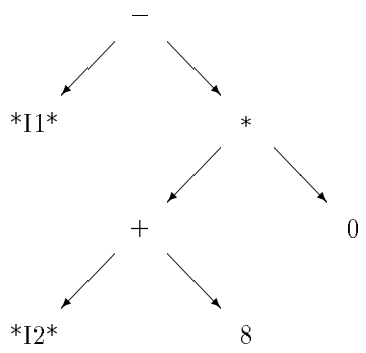

Figure 1: Tree representation of an expression.

5. If the solution found or sufficient number of generations are created then stop; if not go to 2.

The initialization technique is randomly generating mathematical expressions. This introduces the least amount of domain specific knowledge into the initial population through the variables used in the expressions. Unlike Koza's genetic programming applied to particular problems there are no domain specific functions. Only four mathematical functions are allowed; addition, subtraction and multiplication and protected division.

3.2.1 Evaluation

In order to provide a basis to determine the fitness of the expressions, each generation the expressions are evaluated using Lisp's "EVAL" statement by instantiating input values for each of the patterns from the training set.

The fitness of an expression is based on its success in learning a specific task. Since the target outputs are in the range of 0 to 1, the values, once obtained after the evaluation of the expressions, are mapped to values between 0 and 1 by using a squashing function. Several functions have been tested in this mapping including logistic activation function used by [9]. One of the functions which

showed the most success, especially in mapping to binary target outputs, was the following:

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

The fitness (success) of the individual expression is computed by testing them on all training patterns, and dividing the total error by the number of patterns, subtracting from 1 and multiplying by 100 yielding a fitness percentage between 0 and 100.

The expressions are ranked after each generation according to their success. Those who are higher in the rank (higher scoring ones) are said to be most fitting expressions.

3.2.2 Selection

The purpose of selection in GA is to give better opportunity of reproducing to those members of the population which shows better fitness. For the model this means to select those expressions with higher scores (beginning part of the rank) and give them more chance to reproduce.

In the model, parent selection technique for reproduction is normalizing by using an exponential function taken from Whitley's [12] rank-based selection technique. The function generates integer numbers from 1 to population size. The generation of numbers exhibits characteristics of a non-linear function where there is more tendency to produce smaller numbers (since higher scoring expressions are on top of the rank).

The function is $Z = X - \sqrt{\frac{X*X-4*(X-1)*Y}{2*(X-1)}}$ The selection algorithm is based on the X, Y, Z values in the above formula where X is a bias computed as $1+Y$ where Y is a random number between 0 and 1. The value of Z lies between 0 and 1 and in the rank-ordered population N the expression at position $N*Z$ is chosen.

The number produced by this function is used as an index to the ranked population of expressions from highest scoring ones to the lowest scoring. Then, after producing two indices by using the selection function the corresponding expressions are selected to undergo the genetic operators.

3.2.3 Genetic Operators

Applying genetic operators introduces variation to the population of expressions and allows the components (genes) of better performing expressions to live longer. This creates the necessary environment to cause evolution. In order to accomplish this it uses two different genetic operators; crossover and mutation. However, implementation of the both of the operators used by the system

are different than conventional implementations on bit strings since the length of expressions is variable.

In order to apply genetic operators two parent expressions are selected by using the selection function. The crossover algorithm (one point) requires that a point should be selected on each parent at random dividing parents into two. Then the corresponding parts of the parents are exchanged producing two different children. The internal representation of expressions in the system is binary tree representations. In order to choose a point in this tree two different probabilities are used. One probability determines whether we want to go to the left or right branches of the tree and the other determines whether to go down more or to stay at that level. Figure 2 shows systematically how these are implemented in the system.

```
if mutate
   then create new expression
else
   if at end node of either tree
      or probability-down > cutoff
      then swap parts of trees
   else
      if probability-left > cutoff
         then go down on the left branch
              and recurse
      else
         go down on the right branch
         and recurse
```

Figure 2: Crossover and mutation algorithm.

When the point is chosen, the next thing to decide is whether there will be a mutation. If there will be a mutation on both of the trees at that point a new expression is added. Otherwise the parts of the trees side apart from that point are swapped.

4 Experiments and Results

The model is applied to Three Monks' Problems and Parity problems. The performance of the model on the MONKS'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 5 and the binary coding of these original sets (except that

MONK1 is not tested for binary coding). The parameters of GA include a population size of 300, 250 generations, 90 percent probability of crossover and 20 percent probability of mutation. The length of the individuals are determined probabilistically by a scale down factor which was set to 0.75 in most of the runs. This would reduce the probability of growth of the expressions gradually during their creations as well as during the application of the Genetic Operators.

In this particular application of GA on the variable length genotypes, it is not very easy to decide what set of GA parameters to use. Koza uses 90 and 0 percent probability of crossover and mutation respectively. His population size is at least 500 depending on the problem and the number of generations is 51. There are two different issues of concern in deciding what probability of crossover and mutation to use. These issues are also the problems of GP like practices and described in detail below. For the moment it is sufficient to say that since the number of possible solutions is very large for any given problem, when initialising the populations one wants to include as much useful expressions as possible (a good diversity) and maintain the diversity during the process of evolution by means of a higher probability of crossover and mutation. However, this might increase the probability of loosing useful building blocks since it is not clear in this sort of representation whether the new individual created after the genetic operators will be at least as much useful.

The best results of more than 30 runs are given below. One of the findings of the experiments is that a satisfactory solution has not been found in every run. This is not unusual in either GA or GP practice. They both involve a probabilistic process in creating an initial population, in selection of individuals for genetic operations and in selecting a point on the individual for crossover. For this reason, they can not guarantee that any given run would produce a successful solution. It is the usual convention to take independent multiple runs for the same problem to find a satisfactory solution.

BEST RESULTS

	Original Coding		Binary Coding	
	Training	Testing	Training	Testing
MONK 1	91	88	-	-
MONK 2	74	68	79	69
MONK 3	93	98	93.5	97

The results obtained are better than some of the learning algorithms used in the comparison experiment of Thrun [11]. The performance on MONK 1 and MONK 3 is at the level of competing with most of the algorithms. Although the performance on MONK 2 is very low, this is not surprising and similar to the results obtained by Thrun. Moreover, in a recent extension of the experiments [7] where the representation is improved and the performance in learning, especially on the MONK 2 problem, is increased.

The results emphasise how the encoding can enable us to evolve learning rules for these problems with fixed, general and non-problem-specific set of functions. However there are several problems observed during the course of the experiments. They include the difficulty and the computational cost of reaching a solution for complex and larger problems when usin a GP like model. In the discussion section these problems and their possible solutions will be explained within the context of Genetic Programming.

The following are some of the evolved learning rules for the problems. The first of the two numbers at the top of each learning rule shows the success level on the testing and the second shows the success level on the training set respectively.

The evolved learning rules are a more complex re-representation of the original learning rules. Although they don't always produce 100 percent performance in most cases they provide satisfactory success.

Evolved Learning Rules for MONK 1:

In the run producing the first learning rule for MONK 1, random numbers between 0 and 1 are used. This facilitates finding a solution but in later experiments they have been removed. Removing random numbers implies that the solutions merely correspond to the some sort of relationship among the input variables described in terms of fixed set of functions. The performance of second learning is very close to the first one but it does not make use of random numbers

```
Solution 1) 0.885742 0.916289
(- (- (|%| (- *I1* 0.901633) (- (+ (* *I6* 0.55636)
(- (* (- (|%| (- (+ (|%| *I5* *I3*) (* *I4* *I1*)))
(- *I1* *I2*))) (* *I5* 0.720715))))))
(- (|%| (- *I1* 0.050599) (- (+ (* *I6* 0.55636)
(- (* (- (|%| (- (+ (|%| *I5* *I3*) (* *I4* *I1*)))
(- *I1* *I2*))) (* *I5* 0.720715)))))))))
(|%| *I1* *I2*))))

Solution 2) 0.870968  0.913179
(+ (- *I4* (|%| *I4* (- (* (- (* (- (|%| *I3* *I5*)) *I6*))
(- (* *I5* *I6*)))))) (- (* *I4* (- (+ *I4*
(* (- *I1* *I2*) *I5*))))))
```

Evolved Learning Rules for MONK 2 (Binary Coding):

The following are the rules for the Monk 2 problem. Note that the model is relatively successful in coding for the rules of the training set but shows poor performance generalising over the testing set. This is typical for the hard learning problems such as parity problems or Monk 2 where the learning rule

contains a relationship among the input variables. Although it is very difficult to get a successful performance on such problems, in general, the poor performance observed here is not solely due to the strategy employed in the experiments. This is proved to be the case as a result of our control experiment on the parity problems discussed later in the section. As it will be explained later in the discussion section, it has a close link with the way GP like practices.

```
Solution 1) 0.687865  0.745562
(- (- (* *I17* (- *I12* *I6*))) (* (- *I9* *I8*) *I14*))

Solution 2) 0.668543  0.739645
(- (|%| (- (* (+ *I3* *I5*) *I7*))
   (- (+ (- (+ *I6* *I11*)) *I16*))))
```

Evolved Learning Rules for MONK 2 (Original Coding)

```
0.661737  0.794811
(|%| (- (+ (- 0 *I1*) (- *I2* *I2*)))
     (- (- (- (* 2 *I3*) (+ *I5* *I1*))) (+ *I1* *I5*)))
```

Evolved Learning Rules for MONK 3 (Binary Coding):

```
0.973482  0.935242
(- (- (* *I12* *I13*) (+ *I6* *I14*)))))
```

Evolved Learning Rules for MONK 3 (Original Coding)

The learning rule evolved for MONK 3 (original) is the simple but perfect solution discovered in terms of the functions and random numbers used. Here the range of random numbers was 10. The rule is easily understandable and corresponds exactly to the second part of 'OR' in the original learning rule of MONK 3: attribute five is not equal to four and attribute 2 is not equal to 3. Note that the evolved expression implicitly code for relative more specialised functions EQUAL, NOT and AND than arithmetical operators. This clearly demonstrates the power of the model as a potentially useful tool in discovering learning rules for learning problems. The resulting rules can sometimes be a complex and totally new representation or simple re-representations.

```
0.983607 0.935651
(- (* (- *I5* 4) (- *I2* 3)))))
```

4.1 Parity Problems: a control experiment

In order to test whether the poor result obtained on MONK2 is directly attributable to the coding strategy I have carried out a control experiment. It involved testing the model on the similar hard learning problems such as parity.

The MONK2 and Parity problems are similar in that the learning rule describing either refers to some kind of *relationship* among the input variables. The general rule for parity problems states that the output is true if there are even number of true values among the input values. As it can be observed from the following results the model can code for the solutions to the supervised tasks where the learning rule describes a relationship among the input variables. However, when the problem gets larger and more complex (5 bit-parity or higher) it becomes more difficult for the model to code for the solution. In this case, a larger population size and an increase number of generations as well as longer and more complex representations of the solutions may be required. For example Koza in his experiments with even-5-parity problems increased the number of population from 4000 to 8000 to find a solution [5]p.533. This is a huge number compared to our 300 population size and 250 generations.

followings are the results of evolving learning rules for the parity problems. Note that for each of the problem our fixed set of functions are capable of coding at least for OR, AND and NOT.

Evolved Learning Rules for 2-Bit-Parity Problem

```
1.00
(|%| (- (- *I1* *I2*)) (- (- (+ *I1* (- (* *I2* *I2*))) *I2*)))

1.00
(+ (+ *I2* *I1*) (- (* *I2* (+ (- (+ *I1* *I1*)
(- (* *I2* (+ (- *I1* *I2*) *I2*)))) *I1*))))
```

Evolved Learning Rules for 3-Bit-Parity Problem

```
1.0
(+ (- (- (- (|%| *I1* (- (* (- (* *I2* *I1*)
(+ *I2* (* *I3* *I1*))) (+
(* (- (+ *I2* *I1*)) *I3*) *I1*))))) (+ (+ (- (* *I1* *I1*)) (*
(- (+ *I2* *I3*)) (- (+ *I1* (+ (- (- *I2* (- (* *I2* *I2*))))
*I1*))))) *I2*)) (- (- *I1* *I3*) *I1*)) (- (* (- (* (- (+ *I3*
*I3*) *I3*) (+ (- (|%| *I3* *I2*)) *I1*))) (- (- (- (- *I2* (-
(|%| (+ *I2* *I2*) *I2*)))) *I2*)))))

1.0
(- (+ (- (- (|%| (- (* *I3* *I2*)) (* *I1* (- (|%| (- (+ *I2*
(+ *I1* (* *I1* (|%| *I3* *I3*))))) *I3*)))))
(+ *I2* (- (+ *I1* *I3*))))
(- (* (|%| *I1* (- (+ *I3* *I2*))) (- (- *I2* *I1*) *I1*)))) (|%|
(- (- (- (+ (- (- *I2* *I3*)) (- (- (- *I3* *I3*)) *I2*)) (|%|
(+ (- (- *I1* *I1*)) (- *I2* *I2*)) *I1*)) *I2*)) (- (- (+ *I1*
(* (+ *I1* *I2*) *I2*))) (- (* *I3* *I3*)))))
```

Evolved Learning Rules for 4-Bit-Parity

```
0.9375
(+ (+ (- (- (+ (+ *I2* (- *I4* (+ (- (* *I3* *I1*)) *I1*)))
(- (- (- (+ (|%| (* *I4* *I1*) *I2*) (- (|%| *I3*
(+ *I2* *I1*))))) (- (* (- (|%| *I4* *I3*)) (- *I2* *I3*))))))
(|%| (- (|%| *I4* (* *I1* (|%| *I1* (+ *I3* *I4*)))))
(* *I3* (+ *I1* *I2*)))) (- (|%| (- (- (* *I3* *I2*) *I3*))
(- (+ (- (|%| *I1* *I3*) *I4*) *I4*) (- (- *I4*
(- (+ (- (- (- (- *I4* *I3*)) *I2*) *I4*) *I2*)))))))
(* (- (|%| (- (- *I1* (- (* *I1* *I4*)))) *I3*)) (- (|%| (- *I2*
(- (|%| *I4* *I2*))) *I4*)))) (- (* (- (+ (+ *I4* *I4*) *I3*))
(- (|%| (|%| (- (* (|%| *I4* *I2*) *I4*)) *I4*) (- (- (- *I4*
*I3*)) *I1*))))))

0.9375
(+ (- (+ (- (- *I4* (- (- *I1* (- (- *I3* *I2*)))))
(* *I1* (+ *I4* *I4*)))
(* (- (+ (- (* (- (|%| (- (- (+ *I2* *I3*)) (- (+ (|%| *I3* *I3*)
*I4*))) (+ *I2* (- (* *I3* *I3*)))) *I3*)) (* (+ *I4* *I2*) *I4*)))
(* (- (+ (- (- (* (- (- (* *I3* *I1*) (- (* *I3* (- (+ *I2* *I2*)))))
) (+ *I2* *I2*)) *I3*)) *I4*)) (* (|%| *I2* (+ *I3* *I4*)) *I1*))))
(|%| (- *I3* (- (* *I3* (+ *I2* *I1*)))) (|%| (- (+ (- (- *I1*
(- *I2* *I4*))) (- (* *I4* *I1*)))) *I1*))) (* (- (* (|%| *I3*
(* (- *I4* *I2*) (- (* *I4* *I4*)))) (- *I4* *I2*))) (- (|%| (-
(- (- (- *I4* (- (+ *I2* *I3*)))) (+ *I4* *I3*))) (- *I3* *I1*)))))
```

5 Discussion and Future Research Directions

In this paper, the main concern was to see whether the genetic based model will be able to evolve learning rules for Three Monk's problems starting from non-problem-specific functions. Achieving this aim would prove that (1) 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 and that (2) the model is helpful in solving hard learning problems such as Monk 2 and parity problems. The results of the experiments in [6] and being able to discover or re-represent solutions to Monk 1 and Monk 3 problems in this paper provided evidence in support of the first hypothesis. Failing to find a successful solution for Monk 2 seems a poor support for the second hypothesis. However, the model is able to find solutions to the parity problems which are similar to Monk 2 problem where the learning

rule is described in terms of relationships among input values. This provides an additional support in favor of the second hypothesis. Moreover, when the problem gets larger and more complex (i.e. moving from 3 to 4 bit and higher parity problems), evolution of successful learning rules becomes more difficult. As the complexity and size of the problem increase, the current strategy of encoding should search for the larger space to find the solution. One of the problem comes from the non-convergent characteristic of GP like methods. When a solution or a more fit individual is found during the course of the evolution, it can easily turn to be an unfit individual after the application of crossover. Although, as a whole the average fitness of the population increases over generations, it is not very clear whether evolution works as effectively as it is in conventional GA. It seems that tree representation of the individuals and the richness of the alphabet make it more difficult to move to a decreased dimensionality in the search space.

The second issue is that more complex and larger problems might require solutions which are represented hierarchically. In fact, this is exactly what I have found through my recent experiments [7]. The new representation provides a direct hierarchical coding for the possible solutions to Monks and parity problems and improves the possibility of finding solution as well as the speed of it. Although Koza claims that GP is most proper for those problems which require hierarchical representation, there is a strong evidence in my experiments and in [8] that this aspect of the GP might be quite limited. In the new experiment, the representation is allowed to be random expressions organised in layers so that it can code for larger and more complex solutions [7]. In the new experiments, by incrementally building up expressions on the way towards the solution it has been found that (1) in every run a solution can be reached, (2) the solution is reached faster and (3) the power of representation is improved to code for more complex and larger problems.

For all of the experiments reported in this paper the population size and number of generations used are 300 and 250 respectively. This is quite a small number compared to the population size of 4000 and 250 number of generations used by Koza in his experiments with parity problems. Although Koza used problem specific boolean functions such as OR, AND, NAND, and NOT, the experiments reported in this paper only used a fixed set of mathematical functions. Thus, the model seems to be useful and efficient in discovering rules for hard learning problems. However, the solutions produced are complex and difficult to interpret. The complexity (i.e. the longness) of the solution is a general problem in GP practices but the solutions are relatively more easily interpretable due to the problem specific functions used. One of the next step in the future is to find ways of simplifying these solutions, probably, by editing as used by Koza in Genetic Programming. However, I am hoping that a thorough analysis of the evolved learning rules will help to find out how the model works in the future.

Finally, an important finding of the experiments has been raising a ques-

tion about the ability of GP like practices to generalize over the test cases. Although, recent experiments shows that the model can generalise over simple, linearly separable problems, there is no clear evidence whether it can succesfully generalise over hard learning probelms. This issue should be one of the major concern for the next experiments.

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] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Massacheusettes, 1989.

[4] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, USA, 1975.

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

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

[7] Ibrahim Kuscu. Incrementally learning the rules for supervised tasks: Monk's problems. Technical Report CSRP-396, Uni. of Sussex, COGS, 1995.

[8] Una-May O'Reilly and Franz Oppacher. An experimental perspective on genetic programming. In R. Manner and B. Manderick, editors, *Proc of 2nd Intl Conf on Parallel Problem Solving from Nature*, pages 331–340, 1992.

[9] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. In D. Rumelhart, J. McClelland, and the PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Micro-structures of Cognition. Vols I and II*. MIT Press, Cambridge, Mass., 1986a.

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

[11] 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.

[12] D. Whitley. The genitor algorithm and why rank based-based allocation of reproductive trials is best. In J.D. Schaffer, editor, *Proceedings of Third iInternational Conference on Genetic Algorithms*, pages 116–123, 1989.