Generalisation and Domain Specific Functions in Genetic Programming

Ibrahim Kuscu
Department of Computing
University of Surrey
Guildford, GU2 7XH, U.K.
Email: i.kuscu@surrey.ac.uk

Abstract- This research presents an evaluation of user defined domain specific functions of genetic programming using relational learning problems, generalisation for this class of learning problems and learning bias. After providing a brief theoretical background, two sets of experiments are detailed: experiments and results concerning the Monk-2 problem and experiments attempting to evolve generalising solutions to parity problems with incomplete data sets. The results suggest that using non-problem specific functions may result in greater generalisation for relational problems.

1 Introduction

There are many ways in which learning systems may be considered as very useful; for example, the fact that learning systems use input-output data makes them suitable for the situations where some tasks cannot be defined well except by examples. In this case it is possible to specify some input-output pairs as instances of the problem but it may not be possible to state a concise relationship between inputs and desired outputs. We would like learning systems to adjust their internal structures in order to produce desired outputs for a very large set of inputs. A learning system is expected to approximate a target relationship[1] implicit in the examples presented to it during the training process. It may be possible that the complete set of examples related to a problem may not be readily available or may be too cumbersome to present to the learner. The learning system, in such cases, is not only expected to capture the implicit rule in the training data but also to provide correct outputs to inputs which are presented after the training phase. The current state of learning research shows that for those problems where the implicit rule in the training data is not so difficult, learning systems perform well. However, if the implicit rule to be extracted from the instances of a given problem is more difficult and depends on complex inter-relationships among the inputs, the task of the learner becomes much more difficult. This is a real challenge for learning systems especially in being able to produce a generalisation performance for such relational problems.

The solution for the problem of generalisation is directly related to learning bias [7]. Without a proper bias there is a great risk that, for a given problem, generalisation may not be possible. The issue of bias will be raised later in the paper within the context of genetic programming. In fact, the method proposed in this paper is effective in finding general solutions to relational problems due to their capabilities of improving on the learning bias. More specifically, user defined problem (domain) specific functions of genetic programming [4] are evaluated as a factor influencing representational bias. User defined problem-specific functions can introduce a high level of prior knowledge when encoding the potential solutions and may impose constraints on finding generalising solutions to relational problems which require discovery of higher order relationships that may exist in the input-output mapping. Finding generalising solutions to relational learning problems may be realised through the use of some more general, non-problem-specific functions.

This paper presents two sets of experiments where domain specific functions are compared in terms of their generalisation ability with functions which are more general and not so specific to the problem in hand. It will be shown that, contrary to common understanding [4] using non-problem specific functions can result in better generalisation performance for relational learning problems. In the rest of the paper, I will first introduce the relational learning problems, domain specific functions and their relation to learning bias. Then, I will present experimental evidence that using non-problem-specific functions can improve generalisation performance for some relational problems such as Monk-2 and parity problems.

2 Three Monks Problems

Originally, the three monks problems were used to compare the performance of different symbolic and non-symbolic learning techniques [9] including AQ17-DCI, AQ17-FCLS, AQ14-NT, AQ15-GA, Assistant Professional, mFOIL, ID5R-hat, TDIDT, ID3, AQR, CN2, CLASSWEB, ECOBVEB, PRISM, Back-propagation and Cascade Correlation. They involve classification of robots which are described by six different attributes.

There were a total of 432 instances and 124 of them were drawn at random to compose the training set for Monk-1 and Monk-3. For Monk-2 there were 169 randomly chosen cases in the training set. Testing sets for all three problems included the 432 cases. So, training cases were replaced in the testing sets.

In Thrun's experiments two different codings of data sets

[1] There may be more than one relationships existing in the *training* data. The aim of the learning is normally to discover a desired relationship which is applicable to the unseen data.

are used. The first set adapted an original coding for the problems where each of the attributes may have 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}
```

The training and testing sets used for the experiment in this paper are the same as the original coding used by Thrun in the performance comparison experiments. The rules describing the true cases can be formulated as:

MONK-1:
$(attribute1 = attribute2) or (attribute5 = 1)$

MONK-2:
$(attributeN = 1)$ *for EXACTLY TWO choices of n (n 1,2,...,6)*

MONK-3:
(attribute5 = 3 and attribute4 = 1) or
(attribute5 != 4 and attribute2 != 3)

The second set of training and testing cases for the problems were the conversion of the original coding into the binary coding. 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 2, 3 or 4 digit binary numbers where each digit represents the presence of a specific value of the attributes.

There are $27 * 16 = 432$ possible instances and 2^{432} possible monks problems. From these, three were chosen for experimentation. Each of the three problems requires learning of a binary classification task. The most difficult one amongst these problems is the second 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 five percent misclassification.

The results of the comparison (please refer to Table 1) have shown that only Back-propagation, cascade correlation and AQ17-DCI had 100 percent performance on the Monk-2 problem. However, the success of Back-propagation is probably due to the conversion of the original training set values into binary values which will affect the learning rule representing the true cases. The success of AQ17-DCI is attributable to the fact that it has a function which tests the number of attributes for a specific value. Monk-1 and Monk-3 were relatively easy to learn by most of the algorithms.

	M1	M2	M3
AQ17-DCI,	100	100	94.2
AQ17-FCLS,		92.6	97.2
AQ14-NT,			100
AQ15-GA,	100	86.8	100
Assist. Prof.	100	81.3	100
mFOIL,	100	69.2	100
ID5R-hat,	90.3	65.7	
TDIDT,	75.7	66.7	
ID3,	98.6	67.9	94.4
AQR,	95.9	79.7	87
CN2,	100	69	89.1
CLASSWEB,0.10	71.8	64.8	80.8
ECOBVEB,	82.7	71.3	68
Back-prop	100	100	93.1
Casc. Corr.	100	100	97.2

Table 1: Selected Results (in percentages) of Comparison Experiment (Thrun et al '91)

3 Learning Bias and Genetic Programming

In a typical learning scenario two kinds of bias can be identified: representational and procedural bias [7]. Representational bias is defined as the set of possible states in the search space that can be described by the representational language which is used to encode potential solutions. Procedural bias refers to the way in which the states of the space (defined by the representational language) will be searched. Selection of the correct biases can have great influence on whether a problem can be solved and whether generalisation can be achieved. One of the ways to obtain better generalisation of the solutions produced by GP is to improve on learning bias. For GP the following types of learning bias can be identified.

- representational bias: This is introduced when the encoding strategy to represent the possible solutions is chosen. The encoding strategy should define a representation language and some rules for using the language. The language is mainly determined by the user-defined problem-specific functions and terminals and the rules are defined by the decisions on how they will be combined (i.e. structure and maximum depth of the tree). For example, the functions and terminal units should be sufficient to encode a solution.

- evolutionary bias: This refers to the mechanisms of evolution applied to the individuals and can be viewed on two different levels: (1) bias in the selection process and (2) bias in applying the genetic operators.

  The bias in the selection process is mainly determined by the preferences stated in the fitness function to select a potential solution over another to produce more Offspring. For example, the idea that simpler programs are relatively more general can be introduced into the fit-

ness function giving more preference to simpler functions to undergo genetic operators.

The bias in genetic operations refers to the ways genetic operators are used. This can be seen as the bias existing in the main mechanism of searching the space of possible solutions. For example the rate and type of the crossover or mutation operators may affect whether a desired solution can be reached or not.

Review of the research on generalisation in genetic programming [5] showed that almost all of these researchers concentrated on improving on the evolutionary bias in order to promote generalisation, generally by introducing changes to the fitness function.

4 The Role of Domain Specific Functions

In general, a reasonable amount of prior knowledge allowed in a learning model restricts the learner's uncertainty and/or biases and expectations about the domain and can be useful in shaping and reaching a good solution. In GP, however, user defined functions often introduce excessively rich domain knowledge (human intervention) into the model so that the degree of learning to be achieved is excessively poor.

Although the step of selecting user defined functions has been claimed to be common to several other learning paradigms ([4] p.88), the amount of domain knowledge introduced in this step by GP and other paradigms is not compared. For example, the degree of prior knowledge involved in definitions of the character of inputs and outputs to an artificial neural network seems negligible compared to GP's provision of almost a halfway solution in the form of user defined functions and terminal units. Moreover, in the light of substantial research on automating the topological structure and parameter selection of artificial neural networks [6], any claim that neural networks and GP require a similar level of human intervention would be even less convincing. Examining a large number of problems solved by conventional GP suggests that user defined functions and terminal units bias the model strongly towards the user's conception of the possible solutions so that what is left as "learning" becomes merely finding an optimal or near optimal composition of those functions and terminal units. A good modelling of learning using GP should probably be using less problem-specific functions.

This is particularly important in solving the relational learning problems since their solutions, rather than reflecting a direct relationship between inputs and outputs, depend on discovery of a higher order relationship (i.e., a reformulation from the original input/output mapping). A learning model will only be considered successful if it is able to discover (re-represent or reformulate) these implicit relationships and use them in generating a solution to relational learning problems. If user defined functions refer to these "to be reformulated" relationships, a process of discovery will be prevented. If such a re-formulation is not discovered the solution found after evolving on the training set may be one which can only memorise at least some part of the training instances by compression. This should be avoided if general solutions to relational learning problems are to be found.

For these reasons, 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, the 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 favour 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. There are two main reasons to use non-problem-specific functions:

- To reduce the introduction of excessive prior knowledge to the representation of possible solutions so that the solutions are the result of learning process with little or no human intervention.
- To allow the learning process to *discover* (as a result of a re-reformulation process) the functions required to solve relational learning problems. One would expect that after such a reformulation is achieved, generalisation to the test cases will be more likely.

To give a concrete example for the issues above, let's consider the Monks problems. In GP practice a typical function set for each of the monks problems would be as shown below in F function sets for each of the problems. The actual rules describing the true cases are given in Italics.

MONK-1:
$(attribute1 = attribute2)$ or $(attribute5 = 1)$
F={EQUAL, OR, TEST-VALUE-OF-AN-ATTRIBUTE}

MONK-2:
$(attribute N = 1)$ *for EXACTLY TWO choices of N (N 1,2,...,6)*
F={EQUAL, TEST-VALUE-OF-A-NUMBER-OF-ATTRIBUTES, NOT, OR, AND }

MONK-3:
(attribute5 = 3 and attribute4 = 1) or (attribute5 != 4 and attribute2 != 3)
F={EQUAL, NOT, OR, AND }

Note that the above functions make the learning process required for the solutions of relational learning problems a simple one and prevent the assessment of our learning model's performance in solving these problems. If we were to use those functions, the learner would only be finding an optimal composition of the problem specific functions. A more appropriate task is, however, to discover these functions and represent them in a solution in such a way that it can successfully generalise after training. For all of the three problems, I

will use only protected division (where division by zero result in zero rather than an error), multiplication, plus and minus.

5 Experimental Set-up: Monk-2 Problem

The experiment involving the Monk-2 problem using non-problem-specific functions is carried out using lil-gp package.[2] The parameters of the problem are shown in Figure 1. The population size is set to 5000. Initially, the number of generations was 250 but no significant increase in the performance has been observed after the generation 100. The results reported here use 101 generations. The parameters of creating an initial population and breeding were adapted from early implementations of Koza with standard GP ([4] Chp. 7).

```
 max_generations = 100
 pop_size = 5000
#random_seed = 1
 output.basename =  Monk2
#original coding of
 training-testing
 (169 and 263) cases
 data.file = m2o.dat
#how to generate the
 initial population
 init.method = half_and_half
 init.depth = 2-6
#limits on tree size.
 max_nodes = 1000
 max_depth = 17
##breeding parameters
  (emulates Koza Ch. 7)
  breed_phases = 2
  breed[1].operator = crossover,
  select=fitness_overselect
  breed[1].rate = 0.9
  breed[2].operator = reproduction,
  select=fitness_overselect
  breed[2].rate = 0.1
```

Figure 1: Lil-gp parameters used for the Monk-2 problem.

The problem summary is provided in Figure 2. The function set encourages evolution to produce learning rules based only on the four arithmetical functions and any successful performance can only be credited to the discovery of a solution as a reformulation of these functions through an evolutionary process.

Original coding of training and testing data is used rather than the binary conversion (see above). In this way, the relational nature of the problem is preserved. As in the Thrun's experiments, the same 169 training cases and 263 test cases are used. However, unlike Thrun's experiments, no overlapping is allowed between training and testing cases (i.e. the cases which were used for training are not placed in the test cases and likelihood of artificial increases in the test performance due to the success on the previously seen training cases is eliminated).

[2]Lil-gp is a C-based programming environment for development of Genetic programming (GP) applications. It is developed by Douglas Zongker at Michigan State University and publicly available at URL: http://isl.cps.msu.edu/GA/software/lil-gp/index.html

Problem	Monk2
Terminals	attributes
Functions	+, -, %, *,
Random Number	None
Fitness Cases	Same as Thrun's experiments (169 training cases)
Fitness	Number of correct outputs during training
Wrapper	None
Parameters	PopSize=5000, generations=101

Figure 2: Problem summary for Monk-2 problem

5.1 Results on Monk-2 Problem

The training and testing performances of 25 independent runs are reported as a graph in Figure 3. The test scores reflect the performances over *completely* unseen cases. The best training scores observed are more than 90 percent success indicating that it is possible to discover an individual which can learn successfully those instances of the problem which are presented to it. However, the success in predicting the unseen cases does not seem to be as successful. First, the maximum generalisation performance on the test cases is just under 80 percent (3rd and 25th runs). If training cases were replaced back among the test cases (as in Thrun's experiments) the comparative scores would be 0.83 for the third run and 0.81 for the 25th run (these figures are computed based on the overall success on 432 cases; correct hits during training plus correct hits during testing divided by the total number of training and testing cases). These figures are slightly higher than the figures obtained when the training cases are not replaced back into the testing cases.

When these figures are compared to Thrun's experiment (reported in Table 1), GP with non-problem-specific functions outperforms most of the well-known learning methods except AQ family, cascade correlation and back-propagation algorithms. As noted previously, AQ methods used *equality*, a domain specific function, when solving Monk problems. The aim of the experiments presented, however, was to discover such functions. As shown, such an objective can be realised at least with some success. Cascade correlation and back-propagation solved the Monk-2 problem using binary conversion of the data set. In the experiments reported here, the original coding is used since it preserves the true relational nature of the problem. Clearly, it is easier to solve the

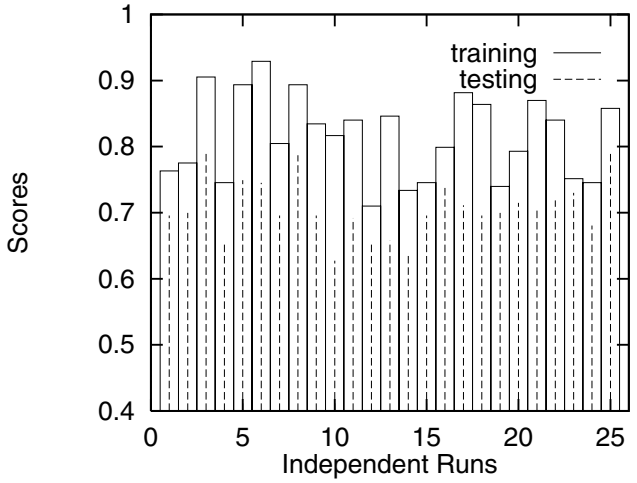

Figure 3: Training and testing performances on Monk-2 problem.

problem if the binary conversion of the data is used.

For almost all of the runs, test performances are drastically lower than the training performance. A general observation is that, for some of the runs, a successful performance in training does not lead to a successful performance in testing. As the evolutionary process progresses, gradual increases in the training performances is observed (though not for all of the runs). However, as expected, these increases are not correlated and reflected in the testing performances. It seems that given the experimental set up, the evolutionary process of GP often results in a learning performance which is good only for learning a set of training cases presented to the system during the course of the evolution (i.e. compression based learning) but not sufficient for predictive generalisation.

However, it is not clear whether the problem with generalisation is attributable to the nature of the relational learning problems or the encoding strategy. In order to discover this, in the next section a set of experiments attempting to find solutions to parity problems using non-problem-specific functions with incomplete data sets will be carried out.

6 Parity problems

Parity problems involve a test that determines whether the number of ones or zeros in an array of binary digits is odd or even [8]. For example, in parity problems, the rule for the true cases is follows:

> The output is true if an odd number of ones are encountered among the n number of inputs.

It has been shown in [3] that parity mappings do not show any regularity which can be described in the form of a direct correlation between value(s) of particular inputs and value(s) of output. So, parity problems are well-known examples of relational problems.

A general attitude in solving parity problems is to provide the learner with the complete set of cases defining the problem [4]. For example, XOR is solved if a successful network topology and set of weights are found based on a training over all (four) instances of the problem. A similar approach is taken for higher bit problems. In fact, all of the parity problems reported in [4] are solved in the same way. This approach adapts compression based learning and does not promote generalisation for such relational problems.

In this section, a set of experiments to solve parity-5 and parity-6 problems with a generalisation oriented approach will be presented. The instances defining these problems will be split into two: training and testing instances. Using genetic programming, the generalisation performance with non-problem-specific functions (i.e. four arithmetical functions) and with problem-specific functions (i.e. logical functions) will be compared. This will help us to understand if the use of non-problem-specific functions can in any way lead to generalisation performance for relational learning problems (i.e., to see if it is possible to discover some higher re-formulations required for the solutions of these problems).

6.1 Experimental Set-up

For this set of experiments (even) parity-5 and parity-6 problems are chosen since they have reasonable number of instances defining the problem (i.e. not too few nor too large). These instances are split into two to determine the training and testing cases. For parity five there are 32 cases defining the problem. This set is split into 22 training and 10 testing cases. For parity 6, a total of 64 cases is split into 45 training and 19 test cases (for each problem around 70 percent (arbitrary) of the instances are selected as training instances representing a reasonably good size for each problem).

The problem summary is presented in Figure 4. Terminal sets constitute the input variables; 5 for parity-5 and 6 for parity-6. Two sets of functions are used for comparison purposes. One set included problem-specific functions. Since parity problems are Boolean problems the basic problem-specific primitives constitute AND, OR and NOT functions. The other set included four mathematical non-problem specific functions. Since lil-gp package is used, the basic GP parameters were the same as those used for the Monk-2 problem reported above except for the number of generations. For parity-5 30, and for parity-6, 70 generations were found to be satisfactory.

6.2 Results on Parity problems

Using the above parameters, for each of the parity problems two sets of 30 runs were carried out ($2 * 2 * 30 = 120$ independent runs). The differences between the sets were the set of functions used. In the first set, problem specific functions (i.e. three Boolean functions AND, OR, and NOT) were used. In the second set, non-problem specific functions (four arithmetical functions plus, minus, multiplication and protected

Problems	Parity-5 (p5), Parity-6 (p6)
Terminal Sets	Input variables
Functions	either {+, -, %, *,} or {and, or, not}
Random Number	None
Fitness Cases	22 for p5, 45 for p6
Fitness	Number of correct outputs during training
Wrapper	None
Population Size	5000
Generations	30 for p5, 70 for p6

Figure 4: Problem summary for Parity problems

division) were used. Figure 5 and Figure 6 depict the results on parity-5 using Boolean functions and parity-5 using mathematical functions respectively. Figure 7 and Figure 8 show the same information for parity-6 problem. Each of the bars represents the training score and the dotted line presents the testing score for every run. The result of the experiments presented some surprising new findings related to generalisation performance on parity problems; a kind of relational learning problem. Looking at these figures some of the significant observations can be summarised as follows:

1. Regardless of the function sets used training scores tend to be high. Surprisingly, (in general) problem specific functions result in slightly lower performances than non-problem specific functions for the same amount of evolutionary time. Often, training performances using non-problem specific functions can reach up to 100 percent. However, in the case of problem specific functions this can only be observed for a few cases. This is a significant new evidence contrary to a major belief [4] that problem specific functions make it easier to find solutions. At least for these problems, *non-problem specific functions may perform better than problem specific functions*.

2. *Testing performances are consistently low when problem specific functions are used whereas, they are frequently much higher when non-problem specific functions are used*. Best test performances with problem specific functions hardly reach 40 percent in the case of parity-5 and are even lower in the case of parity-6. Best training performances with non-problem specific functions, though not so frequently, are greater than 90 percent for each of the parity problems.

3. *The fact that non-problem specific functions can generalise to parity problems with incomplete data, by itself, is a very significant result*. To my knowledge this is the only time parity problems are being shown to be generalisable with incomplete data. Moreover, the results are reached without introducing excessive prior knowledge.

These issues clearly indicate that use of non-problem specific functions can promote generalisation better than problem specific functions. The problem with generalisation observed in the previous experiments seems to be resulting from the complexity of Monk-2 problem rather than the encoding strategy used in representing the potential solutions. This provides evidence that non-problem specific functions might be better for finding generalising solutions for relational problems.

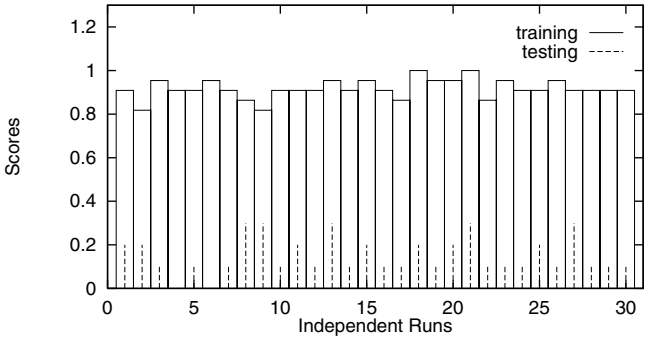

Figure 5: Training and testing performances on Parity-5 problem using problem specific (boolean) functions.

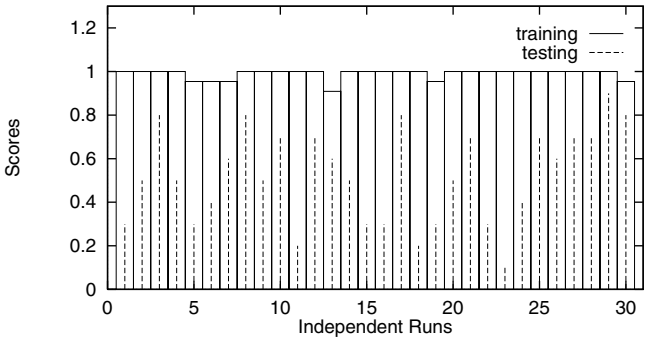

Figure 6: Training and testing performances on Parity-5 problem using non-problem-specific functions.

Looking at generational training and testing scores of 30 runs per problem type (i.e. parity-5 and parity-6) with problem-specific-functions and without problem-specific-functions and observing training and testing scores over evolutionary time for a particular run reveals that, for both of the problems, use of Boolean functions (problem-specific functions) shows a *negative* effect on testing as evolutionary time progresses. In most cases, when Boolean functions are used generalisation performance improves earlier in the evolutionary time and it *reduces* eventually. However, when non-problem-specific functions are used, testing performance will tend to increase while evolutionary time proceeds further. Although this is not observed in all of the runs, the overall performance of non-problem specific functions is greater than

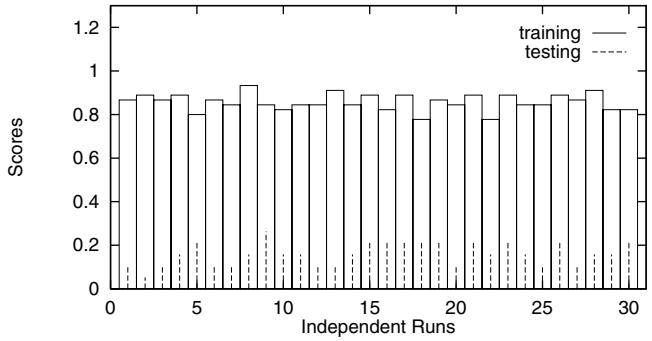

Figure 7: Training and testing performances on Parity-6 problem using problem specific (Boolean) functions.

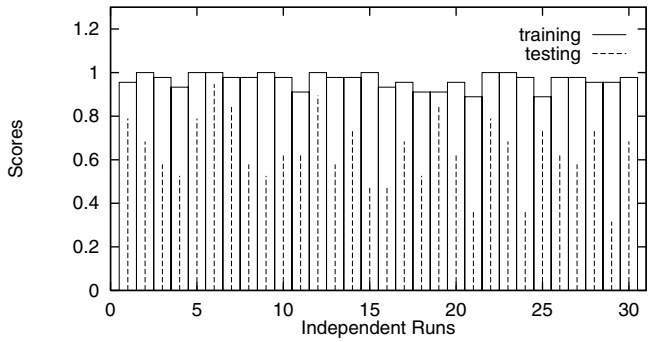

Figure 8: Training and testing performances on Parity-6 problem using non-problem-specific functions.

that of problem-specific functions.

One of the reasons why non-problem-specific functions produce better results than Boolean functions for these problems may be related to the nature of the search space that evolution is operating on. The logical functions create discrete Boolean functions whereas the arithmetical functions have a continuity. This may result in different search spaces. It can be argued that it must be easier for evolution to find its way through the space of possible solutions which are created by arithmetical functions with continuity. The fact that arithmetical functions are more general than the Boolean function may result in a situation where there are various ways of representing a particular Boolean function. This, in turn, may help evolution to find a better generalising solution more easily among many alternatives when arithmetical functions are used. However, when Boolean functions are used evolution may be forced to find a rare and particular representation of the solution. In this case the distinction between using problem-specific and non-problem specific functions reveals itself in the contrast between using Boolean functions and a set of more general functions which may constitute a less rugged search space of the possible solutions.

7 Computational Cost

Using evolutionary methods for supervised problems is normally costly due to the fact that, in every generation, all of the individuals in a population should be evaluated through all the training cases. This is necessary for fitness computations. Relational problems such as Monk-2 pose extra costs due to complexity and size of the potential solutions. They not only add to the cost of evaluation and memory consumption but also make it harder to find a satisfactory (i.e. a success score higher than 90 percent) solution in every run.

Several methods are proposed in [1, 2] to reduce the computational cost in solving supervised learning problems using GP. This research mainly focuses on reducing evaluation cost by choosing subsets of the fitness cases.

8 Conclusions

In this paper, experiments on relational learning problems were presented using lil-gp genetic programming development package. Several runs using non-problem specific functions to solve the Monk-2 problem and parity problems were done to understand the dynamics of non-problem specific functions in obtaining generalising solutions for relational learning problems.

The experiments show that it is possible to solve the Monk-2 problem using non-problem specific functions with a success comparable to most of the well-known learning methods reported in Thrun's experiments. One of the interesting findings of the experiments concerning Monk-2 problem was that even though the system was able to learn the training cases fairly well, it was not as successful in generalising to the test cases. It seems that there is an evolutionary tendency to find a representation which would compress the training cases. However, this compression based learning did not produce a representation which could predictively generalise what is learned during the training to the testing cases. Moreover, in almost all of the runs there was a consistent tendency to over-learn (over-fit) the training cases and do badly on the testing cases.

Better results, both in terms of training and testing scores were found in the parity experiments. The parity experiments supported the idea that non-problem-specific functions may perform better than problem-specific functions in finding generalising solutions to relational problems, though generalisation performances are not ideal.

Finally, the successful performances on the training phase of relational problems indicate why relational problems such as parity are attempted with a complete set of cases. It is easier to find a solution by means of compression to relational learning problems but it is more difficult to generalise for them.

Two main problems still require further investigation. One is how to speed up the search for a solution. The computational cost of evaluating a large population of individuals for every fitness case in the training set is very high. The sec-

ond problem is how to improve on the likelihood of finding a satisfactory solution in every run. These are the issues of concern for forthcoming stages of this research.

Acknowledgements: I'd like to thank to Dr. Inman Harvey for his invaluable support during the process of development of this research and S. West for stimulating discussions. Middle East Technical University, Turkey, has provided the funding for this research.

Bibliography

[1] Chris Gathercole and Peter Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, pages 312–321, Jerusalem, 9-14 October 1994. Springer-Verlag.

[2] Chris Gathercole and Peter Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[3] G. Hinton and T. Sejnowski. Learning and re-learning in boltzmann machines. In D. Rumelhart, J. McClelland, and the PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition. Vols I and II*. MIT Press, Cambridge, Mass., 1986.

[4] John Koza. *Genetic Programming:On the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, 1992.

[5] I. Kuscu. *Evolutionary Generalisation and Genetic Programming*. PhD thesis, The University of Sussex, 1999.

[6] I. Kuscu and C. Thornton. Design of artificial neural networks using genetic algorithms:review and prospect. In C. Bozsahin, editor, *Proceedings of Third Turkish Syposium on Artificial Intelligence and Neural Networks*, pages 411–420, 1994.

[7] Pat Langley. *Elements of Machine Learning*. Morgan Kauffmann, San Fransisco, 1996.

[8] 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., 1986.

[9] S. B. Thrun, J. Bala, E. Bloendorn, I. Bratko, and et al. 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.