Late Breaking Papers at EuroGP'98:
the First European Workshop on
Genetic Programming (Paris, 14-15 April 1998)

edited by Riccardo Poli, W B Langdon, Marc Schoenauer,
Terry Fogarty and Wolfgang Banzhaf

CSRP-98-10
April 1998

School of Computer Science
Research Reports
Late Breaking Papers at EuroGP'98:  
the First European Workshop on  
Genetic Programming (Paris, 14-15 April 1998)  

edited by Riccardo Poli, W B Langdon, Marc Schoenauer,  
Terry Fogarty and Wolfgang Banzhaf  

CSRP-98-10  
April 1998  

School of Computer Science  
and  
Cognitive Science Research Centre  
University of Birmingham  
Birmingham B15 2TT  
United Kingdom
PREFACE

This booklet contains the late-breaking papers of the First European Workshop on Genetic Programming (EuroGP'98) held in Paris on April 14–15 1998. The purpose of the late-breaking papers was to provide attendees with information about research that was initiated, enhanced, improved, or completed after the original paper submission deadline in December 1997.

To ensure coverage of the most up-to-date research, the deadline for submission was set only a month before the workshop. Late-breaking papers were examined for relevance and quality by the organisers of the EuroGP'98 and one of the other members of the programme committee (Bill Langdon), but no formal review process took place.

The 7 late-breaking papers in this booklet (which was distributed at the workshop) were presented during a poster session held on the evening of Wednesday 15 April 1998 during EuroGP'98.

Authors individually retain copyright (and all other rights) to their late-breaking papers.

This booklet is available as technical report CSRP-98-10 from the School of Computer Science, The University of Birmingham, Edgbaston, Birmingham, B15 2TT, UK.

April 1998

Riccardo Poli
W. B. Langdon
Marc Schoenauer
Terry Fogarty
Wolfgang Banzhaf
# TABLE OF CONTENTS

Constructive Learning with Genetic Programming .............. 1  
*K. Govinda Char*

On the Evolution of Interest Operators using Genetic Programming ........................................... 6  
*Marc Ebner*

Better trained ants ........................................ 11  
*W. B. Langdon*

Automatic Evolution of Java Bytecode: First experience with the Java virtual machine ........................................ 14  
*Eduard Lukschandl, Mangus Holmlund and Eirk Moden*

Automatic Discovery of Loop Transformations for Autoparallelisation ........................................ 17  
*Conor Ryan and Laur Ivan*

Genetic Algorithm Decoding for the Interpretation of Infra-red Spectra in Analytical Biotechnology .................. 21  
*Janet Taylor, Jem J Rowland, Richard J Gilbert, Alun Jones, Michael K Winson and Douglas B Kell*

Recursion, Lambda-Abstractions and Genetic Programming ... 26  
*Tina Yu and Chris Clark*
Constructive Learning with Genetic Programming

K. Govinda Char
Department of Electronics and Electrical Engineering
E-mail: kchar@elec.gla.ac.uk

Abstract

This paper investigates the role of genetic programming (GP) as a meta-learning paradigm for connectionist networks. Through a novel approach it is shown how the flexibility in network architecture and learning can be achieved by providing a very general definition for learning and by imposing a single potential constraint within the representation that GP employs.

Keywords: GP, connectionist networks, micro-macro dynamics, learning, emergence.

Introduction

Connectionist networks, despite being powerful paradigms for learning, suffer from a number of limitations that stem from rigidity in terms of fixing the type of network architecture, the type of node activation function and the type of learning for a given problem, a priori, by the designer. The way in which a learning algorithm is implemented for an assumed architecture adds to the above limitations. In recent years, evolutionary algorithms have been successfully employed in evolving flexible architectures and learning mechanisms for known network types (Chalmers, 1990; Dasdan and Oflazer, 1994; Radi and Poly, 1998). If the aim is to achieve real flexibility the architecture and the learning should be allowed to evolve during the process of problem solving, that is while interacting with the task environment. The proposed approach, by naturally combining genetic programming (Koza 1994) and connectionist networks, demonstrates how potential learning rules can be evolved dynamically by providing a general definition for learning and by imposing a potential constraint in the representational structure, that is the genotype that GP employs. It seems that connectionism itself has to be approached from a different perspective if one has to realize its true potentialities.

The Framework

The simulations employ a feature map (Kohonen 1989) as a framework to illustrate the evolution of learning. The map essentially consists of a number of cells (in a competitive layer) competing for a particular signal component from a given input signal space. The winner cell in the network is determined according to the minimum value of the Euclidean distance \( || x - w_n || \), where \( x \) and \( w_n \) are the input and the reference vectors respectively. The learning rule, typically employs an external supervisor to find the winner and adapts its weights for maximal response. The result of the training is described as a process of self-organization that is capable of enforcing a topological ordering. The network learning rule is expressed as:

\[
w_n(t+1) = w_n(t) + \epsilon(t) g(n-n_0(t)) \, (x-w_n(t)) \, \forall \, n
\]

(1)
where \( n \) and \( n_0 \) represent a cell (in general) and the winner cell respectively in the network. The parameter \( \varepsilon(t) \) is the learning rate. The term \( g(n-n_0) \) usually represents a Gaussian function and is expressed as:

\[
g(n-n_0) = \exp(-p) \text{ where } p = \|n-n_0\|^2 / 2\Delta^2
\]  

This function \( g \) has the effect of inducing a lateral-inhibition among the cells and is essential for the success of the algorithm. This function has its maximum size (normalized to unity) when \( n \) coincides with \( n_0 \) and it decays to zero at larger distances. The steepness of the decay is characterized by the width parameter \( \Delta(t) \). The variance \( \Delta^2 / 2 \) controls the radius of the group of cells that are adapted. Thus the winner cell in the network is maximally adapted and the surrounding cells are adapted to a lesser extent depending on the distance \( \|n-n_0\| \). The learning rate \( \varepsilon(t) \) and \( \Delta(t) \) are initially large but reduce monotonically as the learning progresses. Kohonen made certain approximations in generalizing the above rule. Firstly, the total synaptic strength per cell is constant and is the same for every cell. Secondly, all the input signal vectors have the same intensity. Thirdly, the sigmoid function is approximates to a step function. With these approximations every cell can have only one of two states, a zero or a one.

**The Problem**

The sample vectors are drawn from a two-dimensional signal space with real-valued components, taking on a value in a subspace \( V \in \mathbb{R}^n \) with an unknown probability distribution. For the simulations, the sensory input stimuli are provided by a vector \((x,y)\) with components distributed in a chosen subset of a square \([-1,1]^2\). The learning rule mainly consists of the following steps: apply exemplars from the given input signal space for a number of epochs; find the winning cell and adapt the network weights according to the equations (1) and (2).

The learning rule, in essence, moves the two-dimensional reference vector associating each of the cells towards the two-dimensional input signal so as to minimize the quantization error. The quantization error for a given input signal is the distance between the signal and the reference vector of the winning cell over a number of epochs and is defined in terms of error as:

\[
\text{Error} = \sum \text{ABS}(x - w_{x,\text{win}}) + \text{ABS}(y - w_{y,\text{win}})
\]  

where \( x, y \) and \( w_{x,\text{win}}, w_{y,\text{win}} \) are the components of the signal and the reference vectors respectively. The quantization error is finally defined as:

\[
\text{The quantization error} = \text{Error}/(\text{number of cells})
\]  

The training enforces a topological ordering where adjacent vectors in \( \mathbb{R}^n \) are mapped on adjacent (or identical) cells in the competitive layer.

Whether GP is able to evolve the variety of concepts and sequence them appropriately to yield a Kohonen type of learning is to be investigated.

**The GP approach**

The key aspects of the simulation include
• providing a general definition for a connectionist learning rule as a sequence of interacting concepts.

• imposing a single potential constraint that the network weight adaptation should be an integral part of the representational structure, that is the genotype that the GP employs. In this context, the weight adaptation is seen as a symbolic concept, the adaptation process itself being subsymbolic.

• employing a potential strategy such as micro-macro dynamics that enables GP to realize the notion of emergence through its primitives. GP’s primitives are the micro concepts that should enable it to form macro concepts.

Under these assumptions, GP is required to evolve the concept of a winning cell, induce the appropriate direction of weight adaptation for the given signal components and evolve a neighbourhood strategy such as a Gaussian to adapt this cell maximally compared with the rest of the cells in the network. Further, the concepts need to be appropriately sequenced. The simulations initially use a network that has a fixed number of cells to investigate whether GP is capable of evolving any valid learning mechanism. Two possible approaches, the general and the modular approach (Govinda Char, 1998) have been attempted. The general approach is not efficient as it leads to an extremely large search space and also yields learning rules that are difficult to interpret. The modular approach is quite effective and will be considered.

Advantages of modularity
As a meta-learning system, GP seems to be more powerful with the modular approach using ADFs due to the following reasons.
1. Given the general definition for a learning rule as a sequence of interacting concepts, it is possible to modularise each of the macro concepts and make them interact through ADFs. For instance, one of the ADFs can be employed to evolve the concept of the winner while another can adapt the network weights. GP’s primitives as micro-concepts form macro-concepts that, in turn, can be represented in terms of ADFs.
2. The approach enables the tractability and interpretability of the rules that evolve. The formation of concepts and their sequencing can be interpreted easily with the GP hierarchy.
3. As an expression with ADFs can have a number of value-returning branches, each of the ADFs can be assigned an explicit fitness function if needed.
4. The weight adaptation is an integral part of the representational structure and can be achieved through ADFs.
5. The search space becomes more focused (towards the regions of potential concepts) through the use of automatically defined functions enabling the evolution of valid learning mechanisms. This is vital if one expects to achieve a good performance in a reasonable amount of time with the evolutionary paradigm.
6. Co-evolution of neural network structures along with the learning is essential for a variety of problem environments and is possible with a grammar such as the cellular encoding (CE) (Gruau, 1994) that is compatible with genetic programming. The primitives for the grammar can be implemented through ADFs. The advantage of such an approach is that GP can induce the type of network for particular type of inputs/signals including the temporal signals. Also, applications that incorporate different types of architectures and learning at various hierarchical levels are feasible with the proposed method.
The fitness criteria

The definition for the fitness measure need to include the distance information. GP, on its own is unable to induce this information. However, this can be obtained through a clever strategy. For a topological ordering, the weights associated with each of the cells need to move towards the weights of the winner. This difference can be checked and included in the fitness measure.

The sample programs

The initialization file for the GP run is shown.

Population Size: 1000; Number of Generations: 300; Number of ADFs: 5; Creation Type: Ramped Half and Half; Maximum Depth at creation: 6; Maximum Depth at Crossover: 10; Maximum Fitness: 65535; Number to Mutate: 0

The depth parameter should be kept to an optimum value as there is a good number of ADFs co-evolving at a given instant of time. Initially the information on the location of the winner and its distance from other cell(s) are provided in order to see whether GP is able to evolve the right adaptation strategy. As there is no standard measure for the process of self-organization, a maximum fitness of 65535 is assumed. This is based on the fact that the minimum quantization error that can be achieved with a given number of cells reaches a saturation point beyond which it cannot be reduced further. Only further addition of cells can reduce the error. Two possible fitness measures (given by equations 5 and 6) will be discussed. The fitness measure can be defined as a quality function $G(x,y)$ given by:

$$\text{The Fitness} = G(x,y) = \frac{1}{(\text{The quantization error})^2}$$

(5)

The smaller the error the higher is the fitness of the genetic program. This measure can yield an error of almost zero if GP faithfully satisfies equation (3) but in no way guarantees a topological ordering as it does not have the distance information. Another possibility for the fitness measure is:

$$\text{The Fitness} = G(x,y) = \frac{1}{((\text{The quantization error})^2 - (\text{diff})^2)}$$

(6)

where the expression for the quantization error is the same as in the earlier case. The term ‘diff’ represents the average difference from expression (3) with similar expressions for other cells. Reducing this difference enable the weight vectors to move towards the weights associated with the winning cell. GP should have the right information in terms of its fitness measure in order to induce and construct the valid components for the learning mechanism that it evolves. Further, the simulations with a fitness measure as expressed by the equation (5) show that the GP can totally avoid the weight adaptation phase which is crucial. One way of overcoming this problem is by forcing GP to enter this phase. This can be done by having a large quantization error (initially) that will reduce only if the weight adaptation phase is entered. A more natural way is to define the fitness itself in such a way that GP should naturally opt for the weight adaptation. This is possible through a fitness expression as defined by the equation (6) which is found to be quite effective in enforcing a topological ordering and can be observed graphically. A sample program is shown.
Generation : 61
Best of Generation was:
Main: (( ADF2 ( ADF2 ( ADF1 ( ADF5 ( ADF4 ) ( ADF1 ( ADF5 ( ADF5 ) ) ( ADF1 ( ADF5 ( ADF3 ) ) ) ADF1
ADF5 ( ADF3 )))
ADF1: (* (eps (* (glbADF4 (* (wix (glbADF3 )))))
ADF2: (+ (glbADF3 (* (+ (glbADF3 (wiy) (eps))))
ADF3: (Exp (dist)) // the Gaussian
ADF4: ((ABS (- (ABS (wix) (ABS (x)))))
ADF5: ((ABS (ABS (- (- (y (wiy)) (wiy)))))
Fitness : 411; Structural complexity : 40

ADFl and ADF2 adapt the network weights. A number of effective modules (typically as seen in the Kohonen’s rule) are evolved. The weight adaptation phase is naturally entered. Over the course of evolution the modules are appropriately evolved and sequenced. It is to be noted that for the above simulations the information on the winner cell and the distance parameter were provided. In actual practice, these should be allowed to co-evolve with appropriate primitives.

Conclusion

The simulations suggest that GP can be employed as a powerful meta-learning tool for connectionist networks. With a general definition for learning and a single (or a few) potential constraint(s) within the representation, GP can evolve potential learning rules for virtually any type of architecture. In addition, the network architecture can be induced simultaneously by GP with an appropriate grammar.

References

On the Evolution of Interest Operators using Genetic Programming

Marc Ebner
Eberhard-Karls-Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Arbeitsbereich Rechnerarchitektur
Köstlinstraße 6, 72074 Tübingen, Germany
ebner@informatik.uni-tuebingen.de

Abstract

Interest operators play an important role in computer vision. Depending on the type of the environment some features may prove to be more advantageous than others. Thus detection of interesting features has to be made adaptive such that the best features according to some measure are extracted. We are trying to evolve such feature detectors using genetic programming. In this paper we describe our results where the desired operator, which is a Moravec interest operator, is directly specified. We show that the problem is a rather difficult one. Only an approximation to the Moravec operator could be evolved using several sets of elementary functions.

1 Motivation

Interest operators play an important role in computer vision [8]. They highlight points which can be found easily using simple correlation methods. They can be used to calculate accurate distance information and for map building [23]. However no interest operator is suitable for all types of environments. A mobile robot which may be operating in different types of environments should be able to adapt its vision system such that the robot can extract relevant information from its surroundings that can be used best according to some measure.

We are currently trying to equip a mobile robot, a RWI B21 with this type of capability. In this paper we are trying to find a simple interest operator, the Moravec interest operator [23, 22], using genetic programming [14, 15]. The Moravec operator detects points where the minimum of the sum of squared differences between adjacent pixels in four directions, horizontal, vertical and both diagonals is a local maximum. The following section gives a short summary of related work in the area of adaptive feature detection.

2 Background


Other researchers used evolutionary algorithms to extract image features. Lohmann [19, 20] evolved an image filter which determined the Euler number of an image using an evolution strategy [25]. Rizki et al. [26] evolved feature detectors which operate on a stack of images to which morphological operations with structuring elements at different resolutions were applied. Roth and Levine [27] extracted geometric primitives using genetic algorithms [7, 6]. Katz and Thrift [12] generated image filters for target recognition using a genetic algorithm. Bhatacharjya and Roysam [3] used evolutionary optimization for model based object recognition at low signal to noise ratios.

Tackett [29, 28] has applied genetic programming to the task of feature classification. He experimented with moment- and intensity features which are extracted from an already segmented region as well as primitive features such as the mean intensity or standard deviation. Tackett used these features in the terminal set of the algorithm, they are not subjected to an adaptive process. Koza [16] evolved detectors for letter recognition which were able to discriminate the letters “T” and “L”. The detectors moved themselves over the binary pattern and could analyze the pixels in a local 3 × 3 neighborhood. Andre [1] used genetic programming to evolve 2-dimensional feature detectors using 3 × 3 hit-miss-matrices. The task was to discriminate between one designated digit and the rest of the digits. The individuals moved themselves over the image and were able to compare their surroundings with the hit-miss-matrices.

Johnson et al. [9] used genetic programming to evolve Ulman’s Visual Routines [30] for the task of determining the location of hands in the bitmap silhouette of a person. Although Johnson et al. are working on real camera data, they are using preprocessed data for the evolution, namely the bitmap silhouettes which are binarized images obtained by using a blue screen to segment the person from the background.

We previously used evolution, developed by Lohmann [20, 21] a variant of an evolution strategy [25] to evolve hierarchical feature detectors which we applied to the task of character recognition [4]. Using one simple structure changing operator we showed that an increasingly complex detector evolved from simple filter operations. In [5] we evolved edge detectors using genetic programming by approximating the Canny edge detector. In this paper we are focusing on the task of evolving an interest operator. In contrast to the work of Johnson et al. [9] we are working with raw image data that is not preprocessed except for scaling of the pixel intensities.
3 Evolution of interest operators using genetic programming

To evolve interest operators which are being optimized according to some measure, we are using genetic programming. Thus we need to specify the set of terminals, the set of elementary functions, the fitness measure, the parameters for the run and a criterion to terminate the run [14].

3.1 Set of terminals

We selected a gray scale representation of the input image as our sole terminal. The image intensities are scaled to the range [0,1]. Thus the terminal set \( T \) becomes \( T = \{ \text{Image} \} \). Other terminal sets can also be envisaged. For instance one could use the three colors bands red, green and blue or hue, saturation and intensity or some combination of them and let evolution select the terminals which are suited best for the task at hand.

3.2 Set of primitive functions

The set of primitive functions has to be powerful enough such that the problem at hand may actually be solved. The Moravec interest operator is usually written as

\[
I_R(x,y) = \min\left\{ \sum_{-2 \leq x' < x+2} \sum_{-2 \leq y' < y+2} (I(x', y') - I(x', y')^2), \right. \\
\sum_{-2 \leq x' < x+2} \sum_{-2 \leq y' < y+2} (I(x', y') - I(x', y')^2), \\
\sum_{-2 \leq x' < x+2} \sum_{-2 \leq y' < y+2} (I(x', y') - I(x', y')^2), \\
\left. \sum_{-2 \leq x' < x+2} \sum_{-2 \leq y' < y+2} (I(x', y') - I(x', y')^2) \right\}
\]

This expression operating on pixel values can be rewritten into an expression consisting entirely of elementary functions operating on whole images. The structure of the Moravec operator using the such elementary functions is shown in figure 1. The resulting image is filtered by suppressing non-local maxima and applying a thresholding operation to extract interesting points from the images.

In the following text the images used as operands are denoted by \( I \) or \( I_i \) where \( i \in \{1, \ldots, 4\} \) and the resulting image is denoted by \( I_R \). The following unary functions were used:

Negation (Neg): \( I_R(x,y) = -I(x,y) \)
Absolute value (Abs): \( I_R(x,y) = |I(x,y)| \)
Square values (Square):

![Figure 1: Structure of Moravec operator.](image)

Table 1: Different sets of elementary functions used for the experiments.

| 1 | BASE | \{ Avg4x4 \} |
| 2 | BASE \cup \{ Avg4x4 \} | \{ Sum4x4 \} |
| 3 | BASE \cup \{ Sum4x4 \} | \{ Pi3, Add3, Max3, Min3 \} |
| 4 | BASE \cup \{ Sum4x4, Pi3, Add3, Max3, Min3 \} | \{ Pi4, Add4, Max4, Min4 \} |

As raw fitness measure to be minimized we selected the squared pixel differences between the actual and the desired output of the operator. For our problem raw fitness equals standardized fitness.

\[
\text{fitness}_{\text{raw}}(\text{Ind}) = \sum_{i=1}^{5} (U(\text{Ind}(I_i))) + \frac{1}{n} \sum_{p \in I_i} ((\text{Ind}(I_i))(p) - (\text{Moravec}(I_i))(p))^2 
\]

where the five images for the different fitness cases are given as \( \{I_1, \ldots, I_5\} \), \( p \) is a point from the image and \( n \) is the number of points in the image. The evolved operator is denoted by Ind and the desired operator is denoted by Moravec. The term \( U(\text{Ind}(I_i)) \) evaluates to a large value for a uniform image and to zero otherwise.

4 Experiments

We performed five experiments with a population size of 4000 individuals to evolve feature detectors which approximate the response of the Moravec operator. Crossover probability has been set to 85%, reproduction rate has been set to 10% and the mutation rate has been set to 5%. We used ramped half and half initialization and fitness proportionate selection with over-selection. Five fitness cases are evaluated. The five pictures used during the evolution are shown in figure 4. Each run was aborted after 50 generations. For each experiment we performed three different runs. For the experiments we used different sets of elementary functions. The following base set
of elementary functions was used for all experiments:

\[
\text{BASE} = \{\text{Neg}, \text{Abs}, \text{Square}, \text{ShiftL}, \text{ShiftR}, \text{ShiftU}, \text{ShiftD}, -, /, *, +, \text{Max}, \text{Min}\}
\]

The different sets of elementary functions used for the experiments are shown in table 1.

The base set of elementary functions is sufficient to evolve a Moravec interest operator. There seems to be one elementary function missing, namely Sum4x4. However, this elementary function can be constructed using the shift operations and the add operation. The absence of the Sum4x4 elementary function complicates the search for a correct interest operator considerably, because the Sum4x4 elementary function is used 4 times in the correct individual. In each of these places a subroutine performing the desired summation would have to be evolved. This would be a task where automatically defined functions [15] might simplify the problem. We wanted to see what solutions are found if no automatically defined functions are used.

Next, we augmented the base set with the operator Avg4x4. Now genetic programming has the possibility of evolving the constant 16 to produce the required elementary function Sum4x4 (e.g., Sum4x4 = 16 \cdot \text{Avg4x4} \text{ where } 16 = \text{Square}(\text{Square}(\text{Sum}(\text{Min}(\text{Abs}(\text{Min}(\text{Image})))))))). Then we added the required function to the base set. Finally, we added the min, max, +, * operators with arity 3 and arities 3 and 4 to the base set.

The best individual of all runs was found in generation 50 using set 5: \((\text{Min3} (\text{Sum4x4} (\text{Square} (- (\text{Min} \text{Image} \text{Image}))) (\text{Sum4x4} (\text{Square} (- (\text{Neg} (- (\text{Min} \text{Abs} (\text{Min} \text{Image} \text{Image}))) \text{Image})) (\text{ShiftL} \text{Image})))) / (\text{ShiftD} (\text{ShiftR} (- (\text{Max} \text{Image} \text{Image} \text{Image}))) (\text{Square} (\text{Max} \text{Image} \text{Image} \text{Image})))) (\text{Sum4x4} (\text{Square} (- (\text{Neg} (- (\text{Min} \text{Abs} (\text{Min} \text{Image} \text{Image}))) \text{Image})) (\text{ShiftL} \text{Image})))) / (\text{ShiftD} (\text{ShiftR} (- (\text{Max} \text{Image} \text{Image} \text{Image}))) (\text{ShiftL} \text{Image})))) (\text{Sum4x4} (\text{ShiftL} \text{Image}))))).

Set 3 had the highest average adjusted fitness at generation 50. The structure of the best individuals found with the sets 5 and 3 are shown in figure 2. To make sure that random search by itself did not already produce an interest operator we examined generation 0 of the runs which produced these individuals. The structure found by random search in generation 0 both only emphasize vertical edges. The best individual found with set 5 approximates the Moravec operator quite closely. However, it is not a 100% correct individual. The fitness is especially good for 4 pictures where the sum of the squared pixel differences is less than 0.0005. The result that an interest operator with a seemingly simple structure is rather difficult to evolve is quite surprising. Figure 3 shows the average adjusted fitness of best-of-generation individual for all of the base sets. The features detected by the best evolved interest operator can be seen in figure 4. The first two rows in figure 4 show the result of the Moravec operator. The next two rows show the response of the Moravec operator before a suppression of non-local maxima has been applied. Next, the best individual of generation 0 (found by random search) in the run that produced the closest approximation to the Moravec interest operator is shown. The following two rows show the best approximation to the Moravec interest operator found during the experiments. The final two rows show the features extracted by the best evolved individual superimposed on the original images after the non-local maxima suppression and thresholding operator has been applied.

Note that although the response of the evolved operator approximates the response of the Moravec operator very closely,
the actually detected features may still differ. This is due to the fact that a non-local maxima suppression and thresholding operation has been applied that was not included in the fitness function. The task was to approximate the operator response and not to extract the same features. The best evolved interest operator has also been applied to a set of five previously unseen images. The results are shown in Figure 5. The features in the top two rows were extracted with a Moravec interest operator. The next two rows show the response of the Moravec interest operator. The following two rows show the response of the best evolved individual. The final two rows show the features detected by the evolved detector after a non-local maxima suppression and a thresholding operation has been applied.

5 Conclusion and ongoing research

We have shown that genetic programming evolved feature detectors which approximate the Moravec interest operator. However, a 100% correct individual has not been found using a population size of 4000 and terminating the evolution after 50 generations. This could be due to the particular structure of the operator at the top of the tree which could be difficult to find.

We are currently experimenting with fitness functions that are not based on any existing operator. Such a fitness measure only describes the desired characteristics of the interest operator. In addition we are experimenting with high level operators such as edge detection, Gaussian smoothing and Gabor filters which augment the set of elementary functions.

6 Acknowledgements

The author is currently supported by a scholarship according to the Landesgraduiertenförderungsgesetz.

For our experiments we used the lil-gp Programming System, version 1.01 [31]. For image processing we used the Vista software environment [24].

References


Figure 4: The top two rows show five different images were interesting features have been located which a Moravec operator. The final two rows show the features of the best evolved individual superimposed on the original images after the non-local maxima suppression and thresholding operator has been applied. See text for an explanation of the other images.


Figure 5: Five images were used to test the evolved interest operators. See text for further explanation.
Better Trained Ants

W. B. Langdon
School of Computer Science, The University of Birmingham, Birmingham B15 2TT, UK
W.B.Langdon@cs.bham.ac.uk http://www.cs.bham.ac.uk/~wbl
Tel: +44 (0) 121 414 4791, Fax: +44 (0) 121 414 4281

Abstract
The problem of programming an artificial ant to follow the Santa Fe trail has been repeatedly used as a benchmark problem. Recently we have shown performance of several techniques is not much better than the best performance obtainable using uniform random search. We suggested that this could be because the program fitness landscape is difficult for hill climbers and the problem is also difficult for Genetic Algorithms as it contains multiple levels of deception.

Here we redefine the problem so the ant is obliged to traverse the trail in approximately the correct order. A simple genetic programming system, with no size or depth restriction, is show to perform approximately three times better with the improved training function.

1 Introduction
The problem of programming an artificial ant to follow the Santa Fe trail has been repeatedly used as a benchmark problem. Recently we have shown performance of several techniques is not much better than the best performance obtainable using uniform random search [Langdon and Poli, 1998]. We suggested that this was may be because the program fitness landscape is difficult for hill climbers and the problem contains multiple levels of deception which also makes it difficult for Genetic Algorithms.

Analysis of high scoring non-optimal programs suggests many reach high rewards even though they exhibit poor trail following behaviour. Typically they achieve high scores by following the trail for a while and then losing it at a corner or gap. They then execute a random search until they stumble into the trail at some later point and recommence following it. The random search may give them a better score than a competing program which successfully navigated the same bend or gap but lost the trail later.

Here we redefine the problem. The same trail is used but we only place food onto the grid as the ant following along the trail nears it. This makes it difficult for an ant which moves away from the trail to find another part of it by random search. This changes the fitness landscape. We anticipate that almost all optimal points within it will retain their scores and many previously high scoring non-optimal points will be given reduced fitness by the new training regime. This is expected to make the landscape less deceptive and so easier for genetic algorithms. Removal of false peaks may also benefit hill climbing techniques.

The Ant problem and the GP we use to evolve solutions to it are briefly described in Section 2, our results are given and discussed in Section 3, and in Section 4 we give our conclusions.

2 The Artificial Ant Problem
The genetic programming system and the artificial ant problem [Koza, 1992, pages 147–155] were set up identically to [Langdon and Poli, 1998] except the new training technique was used (see Table 1).

The program primitives are the same as before except now food items on the trail are numbered in the order we expect the ant to eat them. Only the first $x$ are placed on the grid initially. As the ant moves and eats them new food pellets are added to the grid. When it eats food pellet $n$ then we ensure all the uneaten food pellets up to pellet $n+x$ are on the grid. 50 (or 100 in the case of $x = 5$) independent runs were carried out with each value of $x$.

3 Results
As expected the improved training technique made the task of evolving suitable programs easier. This can be seen in Figure 1 which plots the estimated minimum number of individuals that the GP with a population size of 500
Table 1: Ant Problem

<table>
<thead>
<tr>
<th>Objective:</th>
<th>Find an ant that follows the “Santa Fe trail”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal set:</td>
<td>Left, Right, Move</td>
</tr>
<tr>
<td>Functions set:</td>
<td>IfFoodAhead, Prog2, Prog3</td>
</tr>
<tr>
<td>Fitness cases:</td>
<td>The Santa Fe trail</td>
</tr>
<tr>
<td>Fitness:</td>
<td>Food eaten</td>
</tr>
<tr>
<td>Selection:</td>
<td>Tournament group size of 7, non-elitist, generational</td>
</tr>
<tr>
<td>Wrapper:</td>
<td>Program repeatedly executed for 600 time steps.</td>
</tr>
<tr>
<td>Population Size:</td>
<td>500</td>
</tr>
<tr>
<td>Initial population:</td>
<td>Created using “ramped half-and-half” with a max depth of 6</td>
</tr>
<tr>
<td>Parameters:</td>
<td>90% crossover, 10% reproduction, no size limit</td>
</tr>
<tr>
<td>Termination:</td>
<td>Maximum number of generations G = 50</td>
</tr>
</tbody>
</table>

needs to create in order to have a probability of at least 99% of finding at least one solution. (This is known as “Effort” required, cf. [Koza, 1992, page 194]).

Figure 2 plots effort calculated from 50 runs (or 100 for \( x = 89 \), size limit=200) for the original Santa Fe trail and the same trail but with \( x \) set to five. No run found any of the very smallest solutions (of length 11). Considering first the original problem, the minimum estimate for the Effort is 189,000 and occurs with a size limit of 100 however values for 25–100 seem similar at about two thirds of the Effort required when no size limit is imposed. Also Effort values for size limits of 200–500 appear to be much the same as when there is no size limit. Now considering the case where food is only placed on the grid as the ant approaches it in along the trail (i.e. \( x = 5 \)), the minimum estimate for the Effort is 104,000 and occurs with a size limit of 50 however values for 25–100 seem similar at about 80% of the Effort required when no size limit is imposed. Again Effort values for size limits of 200–500 appear to be much the same as with no size limit.

Changing amount of food on the trail has no obvious effect on the size of solutions.

4 Conclusions

By making a modest change to the Santa Fe trail problem we have made it better at training GP. In terms of the Effort required it is approximately three times easier and GP performs about as well as the best results previously obtained using variable length hill climbers [Langdon, 1998] and size restricted EP search [Chellapilla, 1997]. By enforcing an optimal size limit on the programs being evolved we are able to do marginally better. Surprisingly GP performance is only marginally affected by a size limit and is roughly constant for wide ranges in maximum size.

This indicates GP is still not greatly out performing random search.

References


Figure 1: Effort v. no. food pellets ahead the ant can see and eat. Error bars indicate estimates given by one standard deviation above and below minimum measured figure. Original Santa Fe trail always allows all 89 food pellets to be seen.

Figure 2: Effort v. maximum program size (11, 18, 25, 50, 64, 100, 200, 300, 400 and 500) for original Santa Fe trail and ant restricted to look ahead of 5. (For comparison, cf. Figure 1, values for runs without a size restriction are plotted at 600). Error bars indicate estimates given by one standard deviation above and below minimum measured figure.
The Automatic Evolution of Java Bytecode

First Experiences with the Java Virtual Machine

Eduard Lukschandl
Ericsson Hewlett-Packard Telecom
Box 333
SE-43124 Moelndal
Sweden
ehsedli@acom.ericsson.se

Magnus Holmlund
Chalmers University of Technology
Gothenburg, Sweden
magnus@hot-shot.com
d3moden@dtek.chalmers.se

Erik Moden

Introduction
The advantages of using Java as a platform for Genetic Programming are well known:
- Ease of programming compared to other Object-Oriented languages
- Portability
- Ease of writing distributed applications
This paper describes our study of another property of the Java system – the fact that the Java compiler
translates Java source code into Java bytecode.
This bytecode is in turn interpreted at runtime by the Java Virtual Machine running on the target
system.
Just-in-time compilers (JITs) can be used, which generate native code on-the-fly, i.e. parallel with the
interpretation of the bytecode. If later already translated bytecode would be interpreted the translated
native code is used instead, giving a performance boost.

The JVM’s architecture and instruction set
The virtual processor is stack based and uses numbered “local variables” instead of registers.
The instructions themselves are coded using only one byte (bytecode) though some of them take
arguments.
There are instructions for
- pushing constants onto the stack
- stack manipulation
- doing arithmetic
- pushing local, instance and static variables onto the stack
- popping stack values into local, instance and static variables
- flow control
- creating objects

The structure of a Java program
Basically a Java program is a class with methods and fields (variables).
The methods define the behaviour of the program, the fields provide state.
There are static fields, i.e. memory shared by all instances of a class, and non-static fields, i.e. memory
that is instance-specific.

The Purpose
This study will have two goals:
- In general, broadly compare the performance of GP using evolved Java bytecode with other
implementations of some standard problems from Koza’s books.
- Compare the influence of using different kinds of memory on different problem domains.
The Method
The genotype of the individuals of the population will be represented as a bytecode array. The individuals will be evaluated by letting the JVM directly execute the bytecode. By simply choosing different subsets of the set of Java instructions as the GP function set, different memory schemes can be tested.

Different possibilities arise when defining the mapping between the GP related terms and the Java system, but as a first try, we decided the following:

- A generation of individuals is implemented as a bytecode array representing a Java class.
- Each individual is implemented as a (static) method of this class.
- The program structure consists of a main loop (over the number of generations) containing an inner loop invoking all the methods.
- During a run there will always be two bytecode arrays in memory – one being executed (seen as code) and one being built up representing the growing number of individuals of the new generation (seen as data).

**Benefits and first results**
- order of a magnitude faster than individuals represented as source code data structures
- compact memory usage in spite of that (not the usual trade-off between speed and memory)
- easy mapping between GP concepts and implementation primitives
- easy distribution of individuals over a number of Java Virtual Machines
- state-of-the-art interpreter techniques provided by external experts

The last point might raise an interesting question for further research. Because of the interpreter (and JIT) trying to optimize the execution of the bytecode (and generated native code), it is actually not the possibly by introns bloated genotype that is evaluated for fitness but an automatically generated phenotype that is beyond the control of the casual programmer. In the case of the optimized interpretation the result is not even stored!

**Related work**
- Astro Teller: Indexed memory
- Peter Nordin: Evolving machine code
- Lee Spector: Cultural Transmission of Information
An example: Symbolic regression

Objective:
Using only integer arithmetic find a function of one independent variable and one dependent variable, in form of a method in a Java classfile (bytecode) that fits a given sample of 20 data points, where the target function is the quartic polynomial

\[ x^4 + x^3 + x^2 + x \]

Terminal set:
iload_0 (parameters are passed this way)

Function set:
iall, imul, isub, idiv, irem, ireturn

Parameters:
M = 500, G = 51.

Result after 12 generations:

```
iload_0
iload_0
iload_0
iload_0
iload_0
imul
iload_0
iall
imul
iall
iload_0
imul
iall
ireturn
```

---

corresponding tree for comparison (evaluation order right-to-left)

The following is worth noting:

The function irem is the remainder function.
The function idiv is the unmodified integer division.
Division by zero causes a Java exception which we catch. The offending individual is punished by adjusting its fitness negatively.
Automatic Discovery of Loop Transformations for Autoparallelisation.

Conor Ryan  
Dept. of Computer Science  
and Information Systems  
University of Limerick  
Ireland  
conor.ryan@ul.ie  
Tel: +353 61 202755  
Fax: +353 61 330876

Laur Ivan  
Dept. of Computer Science  
and Information Systems  
University of Limerick  
Ireland  
laur.ivan@ul.ie

Abstract

We describe an extension to an existing autoparallelisation tool, Paragen, which uses Genetic Programming to transform serial code into functionally equivalent parallel code. The extension takes the form of a library of transformations specific to loops. The new genetic structure is described, as well as a different approach to evaluating the fitness function which drastically reduce the computational load for the system.

Keywords: Genetic Programming, Autoparallelisation

1 Introduction

Converting existing serial code into functionally equivalent code is a difficult and, as yet, unsolved problem [2]. Conversion generally involves examining the original code for possible transformations which, while increasing the parallelism of the code, do not affect the functionality. However, even when all possible transformations have been identified, there is no algorithmic way to determine the best order in which to apply them [2]. Indeed, $n$ transformations can yield up to $n!$ separate programs, all with varying degrees of parallelism.

The Paragen System[4][3] employs Genetic Programming to evolve sequences of transformations for program conversion. GP is used to determine not only which transformations may be used, but also the order in which they should be applied. Furthermore, using a system termed Directed Data Dependency Analysis, these transformations can subsequently be used to prove that the resulting parallel program is functionally equivalent to the original, serial program.

We now report on an extension to Paragen which permits it to perform transformations on and within loops. Loops are critical to parallelism as they tend to be the most time consuming part of a program.

2 The Paragen System

Rather than evolving programs, Paragen generates evolution lists of transformations which are applied to the original program. We introduce the notion of "program segments", where a segment is the part of a program currently undergoing transformation.

There are two main classes of transformation employed by Paragen, each adopted from the laws of Occam. The first is referred to as the $F/L$ class, where each transformation is of the form $Fxxx$ or $Lxxx$, and where $xxx$ is $SEQ/PAR$. These transformations schedule a single instruction from a segment, and pass the remaining group of instructions onto subsequent transformations in the list.
We refer to the other class of transformation as the P/S class. This class divides a segment into two new segments with varying numbers of instructions, each of which can subsequently have further transformations applied to them. In figure 1 the input sequence for the first chromosome is [ABCDEFGH]J. The first chromosome’s P20 transformation (execute first 20% of the segment in parallel with the remainder of the segment) generates two new segments [ABCDEFGH] and [IJ] which are passed to the NULL and S80 subtrees respectively. Increasingly smaller segments are generated by each transformation as the tree is traversed, until eventually a NULL transformation is encountered, which signifies the end of the transformations. Any instructions remaining in a segment at this stage are executed in their original order. If a program segment is reduced to zero instructions before all the transformations are exhausted, the extraneous transformations are ignored.

There is an additional class inserted because of the some situations which may appear. this class is only one instruction called SHIFT. This operator just delays the execution of its input parameter sequence with one time period. The order of the executions inside the sequence is not affected, and all the instructions are passed to the next operator.

All the transformations are based on the well defined laws of occam, i.e.

$$SEQ(A, B) = PAR(A, B)$$

if there are no dependencies between A and B. Other fundamental laws are the transitivity and associativity of instructions:

$$SEQ(A, B, C) = SEQ(SEQ(A, B), C) = SEQ(A, SEQ(B, C))$$

![Figure 1](image)

Figure 1: An example of crossover in Paragen

<table>
<thead>
<tr>
<th>Time Step</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
</tr>
</tbody>
</table>

2.1 Modes of Operation

Paragen operates in two distinct modes, namely Atom mode and Loop mode. The system is in Atom mode when applying transformations to atomic instruction, while Loop mode occurs when the code being operated on is a loop. The system remains in atom mode until it discovers a meta-loop, a group of one or more adjacent loops, which causes Paragen to enter loop mode.
The transformations available in loop mode are standard loop transformations[2], which are quite different to the standard atomic instruction transformations. To allow an individual to manipulate both types of transformation, a second chromosome is added to each individual. This chromosome is linear, and is simply a list of which loop transformations should be applied to the program. Each time a meta-loop is encountered by the tree chromosome, the linear chromosome is examined for an appropriate transformation. Once the meta-loop has been dealt with, execution returns to the tree chromosome.

Having the linear chromosome in addition to the tree chromosome also has the advantage that it can also be used to store genes that control certain control parameters of the run (using a meta-GA). Features of an individual such as breeding preferences or breed size can all be controlled in this manner.

Due space constraints, we are not able to present all the loop transformations, and will therefore describe two of the most significant: Loop Coalescing and Loop Swap.

Loop coalescing can be of use when nested loops are encountered in the body of a program. Nested loops are of particular importance due to the difficulty associated with their scheduling, due in most part to the fact that instructions contain within interrogate the value of the indices of both loops. Thus, if these loops were to be spread across a number of processors, the result would be a large degree of communication. In these cases, it is often best to fuse the two loops, as shown below:

\[
\begin{align*}
&\text{for (i=1; i<=N; i++)} \\
&\quad \text{for (j=1; j<=N; j++)} \\
&\quad a[i][j] = \ldots \\
&\text{end} \\
&\text{end} \\
&\implies \\
&\text{for (k=1; k<=N*N; k++)} \\
&\quad a[k/n][k%n] = \ldots \\
&\text{end}
\end{align*}
\]

Another, more simple operation, which commonly occurs is the swapping of loops. Although swapping the loops in itself does not reduce the execution time, a successful application of this transformation indicates that the loops involved can be executed in parallel [Burns 88] [2]. Below is an example:

\[
\begin{align*}
&\text{for (i=1; i<=N; i++) expression1;} \\
&\text{for (j=1; j<=N; j++) expression2;} \\
&\implies \\
&\text{for (i=1; i<=N; i++) expression2;} \\
&\text{for (j=1; j<=N; j++) expression1;}
\end{align*}
\]

3 The Fitness Function

There are two fitness measurements in Paragen, one each for the speed and the correctness of the resulting program.

Paragen employs the transformation rules which make up an individual to highlight the areas of the program which require analysis, and any individual that executes two (or more) dependent instructions in parallel can be punished accordingly. While in atom mode, only those functions which transform the program, i.e. FPAR, LPAR etc. need be examined, as all the sequential functions preserve the original order.

In Loop mode, however, there is far more scope for data dependency clashes, as there are a number of clashes that can occur. Cross iteration dependencies, dependency clashes within the body of the loop and clashes between instructions within the loop and other, outer instructions.

All dependency clashes are totalled and used as a penalty measure for the individuals. Thus, an individual with a score of zero has no dependency clashes.

3.1 Reducing Evaluations

While Paragen is a fully automatic method for generating transformations, there is a significant amount of time consuming repetition. As each individual must be tested, it is clear that the same instructions will be tested for dependency several times. We adopt a lazy approach to testing, and stipulate that each pair of atoms should be tested for dependency at most once. This laziness is achieved by simply noting which instructions have been tested, so the second and subsequent tests of a pair of instructions need only perform a look up to perform the test.

The simplest approach is to have a lookup dependency matrix with an entry for each pair of variables. If a pair are looked up for the first time in a run, they are analysed for dependencies and an appropriate entry
is made in the matrix. Thus, for second and subsequent look ups, the result will be immediately available. However, due to the manner in which loops are treated, i.e. the same as atoms until some loop specific transformations are applied, more information is required. To this end, we modify the matrix as in figure 2 to include a *loop layer* for each loop. Each loop layer is essentially a miniature dependency matrix which the system uses when transforming the code within a loop.

![Diagram of data dependency matrix](image)

**Figure 2:** The data dependency matrix - Atom mode and Loop mode layouts

The matrix permits Paragen to interrogate it about the presence or otherwise of dependencies between any combination of instructions, either atom-atom, loop-atom or atom-loop. These drastically reduces the amount of dependency checking that must be carried out, as there will only ever be one check made for each possible combination.

## 4 Discussion

We have described an approach to the automatic parallelisation of serial code. Our system, the Paragen system uses Genetic Programming to generate lists of language independent transformations which, when applied to existing serial code, generate a parallel program. Furthermore, the list of transformations can subsequently be used to prove that the new, parallel version is functionally identical to the original program.

A number of useful transformations have been identified, and the system is capable of learning the near optimal sequence in which to apply them. Data dependency analysis is kept to an absolute minimum through use of a dependency matrix which operates in a similar manner to a compilers look up table.

This draft paper by necessity contains only a summary of the Paragen system, and the full paper will contain much more detail on the transformations employed and examples of how the individuals are used to map a serial program onto a parallel machine.

## References


Genetic Algorithm Decoding for the Interpretation of Infra-red Spectra in Analytical Biotechnology.

Janet Taylor, Jem J Rowland, Richard J Gilbert, Alun Jones, Michael K Winson and Douglas B Kell
Department of Computer Science and Institute of Biological Sciences, UW Aberystwyth, Ceredigion, UK SY23 3DA
Tel: +44 (0)1970 622353
Email: jjt95/jjrrog/ajw/ mkw/dbk@aber.ac.uk

Abstract
This paper presents an initial study into the use of a modified Genetic Algorithm (GA) in the analysis of multivariate spectroscopic data that relates to rapid screening for metabolite overproduction in the context of process improvement in the pharmaceutical industry. The development of a GA based method for both wavelength selection and relationship modelling is compared with that of a Partial Least Squares Regression (PLS) in the formation of a predictive model of metabolite concentration. Initial results indicate that the combination of a GA structure and alternative mutation strategies is capable of producing a model that is accurate in terms of predictive ability and concise in the output expression produced.

Introduction
Statistical methods for deducing relationships in high dimensional data are often based on assumptions such as normality of distribution, independence of variables, and linearity(1). In many complex multivariate domains these assumptions do not hold and methods are prone to failure (2). Artificial Neural Networks can be effective in forming nonlinear predictive models (3, 4, 5) but allow little direct analysis of the relationship modelled.

Selection of variables can provide significant improvement in the predictive ability of a model. Thorough searches of the space of possible combinations are, however, infeasible for all but small datasets (6). Traditional Genetic Algorithms (GA’s) (7, 8) have been applied to many optimization problems, including wavelength selection (6) but have been used mainly to select optimal subsets of wavelengths (or variables) prior to modelling by classical methods such as Multiple Linear Regression (MLR) (9) or Partial Least Squares (PLS) (10, 11).

 Genetic Programming (GP) (12, 13) has been applied with success to multivariate analysis of biological data (14, 15, 16) and has the ability to select variables of interest and form a predictive model of the relationship. Although, in this case, the model is explicit, the rules produced can be difficult to interpret.

Both techniques use two-parent crossover as the principal reproductive process, typically around 80% of the time. In some multivariate datasets, such as those considered here, if a variable is selected as important, a near neighbour may be more important in the combination of variables currently selected. This is not fully exploited by the use of crossover, as neighbouring variables to the ones selected are not necessarily considered, neither does mutation by random, single point replacement (17) address this issue.

We are developing a modified Genetic Algorithm approach, to permit interpretation of continuous spectra, that aims to achieve both variable selection and the evolution of a simple expression in a one stage process. The genes of the chromosomes encode continuous regions of a spectrum, along with a weighting, an exponent value and an operator. Emphasis is placed on the use of mutation which can replace a variable with an immediate neighbour or change the width of a spectral region with the aim of reducing the size of each gene to one spectral variable. Crossover is used to aid coverage of different spectral regions.

Data Acquisition
The bacterial strain used was Escherichia coli HB10. Samples were prepared containing different concentrations of Ampicillin. Each of the 40 samples was transferred into a 5mm diameter well in a sandblasted aluminium plate (measuring overall 10cm by 10cm) and dried at 50°C for 30 min. The plate was mounted onto a motorised stage and the samples analysed using a diffuse reflectance TLC accessory (2, 18, 19) connected to a Bruker IFS28 FT-IR spectrometer equipped with an MCT (mercury-cadmium-telluride) detector cooled with liquid N₂.

The IBM-compatible PC used to control the IFS28 was programmed (using OPUS version 2.1 software provided by Bruker) to collect spectra over the wavenumber range 4000 cm⁻¹ to 600 cm⁻¹. Spectra were acquired at a rate of 20 s⁻¹, and spectral resolution was 4 cm⁻¹. To improve signal-to-noise ratio, 256 spectra were co-added and averaged. Each sample was thus represented by a spectrum containing 882 points and spectra were displayed in terms of absorbance as calculated using the Opus software. Each of the 40 samples of different ampicillin concentrations was analysed via four replicates and the spectra for all replicates of an individual

---

1 Bruker Spectrospin Ltd., Banner Lane, Coventry, UK
sample were averaged resulting in a 40 by 882 data matrix. This study used only variables 550 to 750 of the
original dataset, a region known to be rich in ampicillin-related information. The subset was split into training
and test sets, each containing 20 data objects, using in-house software based on the Duplex algorithm (20).

**Modified Genetic Algorithm.**

A simple Genetic Algorithm framework written in C formed a basis for the program. Integers were used
for encoding of each chromosome for ease of interpretation. Each chromosome consisted of 10 genes; each gene
represented one variable region. This number of genes was chosen as a good compromise between predictive
ability and complexity of output expression. Each gene consisted of 5 elements, as in Table 1.

<table>
<thead>
<tr>
<th>Position in Gene</th>
<th>Element</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Weight</td>
<td>0 to 9999</td>
</tr>
<tr>
<td>1</td>
<td>Mid point of spectral subregion</td>
<td>0 to maximum no. variables</td>
</tr>
<tr>
<td>2</td>
<td>Length of subregion</td>
<td>1 to 10</td>
</tr>
<tr>
<td>3</td>
<td>Exponent value</td>
<td>0 to 3000</td>
</tr>
<tr>
<td>4</td>
<td>Operator: +,-,&quot;/&quot;</td>
<td>0,1,2,3</td>
</tr>
</tbody>
</table>

Table 1: Gene structure and encoding.

The initial values for each individual were randomly generated, and a single population was evolved
using the training set of the data. In decoding, weight and exponent values were converted to 3 dp float values
(for example 3456 to 3.456). The midpoint variable and the length were used to average the values of the
variables in the spectral subregions selected. Between-gene crossover was implemented for 30% of the
reproductive processes, with mutation at 70% (chosen from initial experiments - data not shown). There was no
direct reproduction of an individual. Several mutation strategies are implemented: a mutation point was
randomly selected along the chromosome, weight, exponent and operator values were all subject to random
mutation, the midpoint variable was directed to one of its immediate neighbours to exploit the quasi continuous
nature of the data, and the length of a spectral subregion was either increased or decreased.

The objective function decodes the chromosome into an expression consisting of ten terms of the form
weight*(average of variables in subregion)^exponent linked by arithmetic operators. It then evaluates the
expression for each row in the training set to obtain fitness scores for each individual in the form of the RMSEP
(root mean squared error of prediction) value, where:

\[ RMSEP = \sqrt{\frac{\sum(y - \hat{y})^2}{n}} \]

in which \( y \) = measured output, \( \hat{y} \) = predicted output, \( n \) = number of examples.

The modified GA program was run on a 500MHz DEC Alpha Workstation under Digital Unix for 500
generations using a constant population size of 750 individuals.

**PLS**

Partial Least Squares regression (21,1) uses the x (observed) variables to produce a reduced number of
latent variables (factors) which are regressed against the y (predictor) variables to form a model. PLS is a
supervised learning method; the model is normally formed using a training dataset, with a test set used to select
the optimum number of factors, and an unseen set then used for validation. In this case, however, only training
and test sets were used; this is likely to give an optimistic value for the RMSEP of the PLS model in comparison
with that of the GA. The PLS algorithm was implemented using in house software (5); the prediction models
were formed and tested on the same sets that were described previously so as to provide a comparison with the
Genetic Algorithm.

**Experiments, results and discussion**

The results from the modified GA were compared to those from PLS for the same dataset. The GA was
run 10 times to produce an averaged test RMSEP error of 1.22, whilst the PLS produced an RMSEP of 1.04
using 4 factors. A second GA experiment was then carried out to simplify the output expression further; the
exponent value in the chromosome was set to a constant value of 1.000, with mutation of the exponent disabled.
Again the GA was run 10 times to obtain an average training and test error.

Examination of the expressions produced by the most fit individuals from both GAs shows clearly the
variables selected, with predictive ability comparable with that of the PLS model. The mid points of the
subregions selected are shown in the Table 2 as the second value in each gene and their size as the third.
Although there is a slight increase in the averaged RMSEP values, the best model from each GA (shown below) improves on the test RMSEP of the PLS model. Although it is possible to analyses a PLS model to determine the most important variables used, the explicit nature of the GA model gives clear benefits.

<table>
<thead>
<tr>
<th>mGA with Exponent Value between 0 and 3</th>
<th>mGA with Exponent Value constrained to 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5790 111 4 1835 2</td>
<td>9585 20 2 1000 0</td>
</tr>
<tr>
<td>6764 101 5 2925 0</td>
<td>9340 131 2 1000 1</td>
</tr>
<tr>
<td>2153 75 2 2687 0</td>
<td>3447 153 2 1000 3</td>
</tr>
<tr>
<td>2253 80 8 1452 1</td>
<td>7241 61 4 1000 2</td>
</tr>
<tr>
<td>2374 62 5 1914 3</td>
<td>6446 154 9 1000 0</td>
</tr>
<tr>
<td>3463 76 4 2576 0</td>
<td>2654 103 2 1000 3</td>
</tr>
<tr>
<td>8411 179 6 1538 3</td>
<td>1064 54 2 1000 0</td>
</tr>
<tr>
<td>2525 174 7 2085 2</td>
<td>4900 134 6 1000 0</td>
</tr>
<tr>
<td>2363 124 10 893 0</td>
<td>9470 33 1 1000 1</td>
</tr>
<tr>
<td>8356 150 2 2132 0</td>
<td>9982 166 2 1000 2</td>
</tr>
<tr>
<td>Training RMSEP = 0.245</td>
<td>Training RMSEP = 0.460</td>
</tr>
<tr>
<td>Test RMSEP = 0.848</td>
<td>Test RMSEP = 0.976</td>
</tr>
</tbody>
</table>

Table 2: Fittest chromosomes and corresponding error values for the two versions of the mGA.

Figure 1:
Training Set Spectra for Ampicillin in E.coli, showing subregions selected by the mGA without exponent values in the output expression.

A small function external to the mGA decodes selected individual into simple expressions as shown below; for these initial experiments, the final operator is simply ignored in the evaluation function. The subregion average is shown as \( R_x \) where \( x \) is the gene number. The operator precedence in these expressions is strictly left to right. The averaged regions used in equation b) are shown in Figure 1. Most of the regions have evolved to 2 variables with a single variable (region 9) that is in the spectral peak that is characteristic of ampicillin.

a) With exponent:
\[
5.79(R1)^{1.835} \times 6.764(R2)^{2.925} + 2.153(R3)^{2.687} + 2.253(R4)^{1.452} - 2.374(R5)^{1.914} / 3.463(R6)^{2.576} + 8.411(R7)^{1.538} / 2.555(R8)^{2.085} = 2.363(R9)^{0.893} + 8.356(R10)^{2.132}
\]

b) Without exponent:
\[
9.585(R1) + 9.34(R2) - 3.447(R3) / 7.241(R4) \times 6.446(R5) + 2.654(R6) / 1.064(R7) + 4.9(R8) + 9.47(R9) - 9.982(R10)
\]
**Conclusion.**

These initial experiments with a modified Genetic Algorithm using integer representation and directed mutation can give output expressions that are easily interpreted and give good predictions. From the evolved expression, important spectral regions and their contribution to the model are easily identified. Further refinement of the algorithm, representation and encoding may be built upon this framework for simple and rapid analysis of full spectral datasets from a variety of biotechnological samples.

**Acknowledgements.**

We are grateful to the BBSRC, EPSRC, HEFCW, GlaxoWellcome, and Dr. Andrew Edmonds of Science in Finance Ltd., for their support.
References.

Recursion, Lambda Abstractions and Genetic Programming

Tina Yu and Chris Clack
Department of Computer Science
University College London
Gower Street, London WC1E 6BT United Kingdom
T.Yu@cs.ucl.ac.uk C.Clack@cs.ucl.ac.uk

ABSTRACT

Module creation and reuse are essential for Genetic Programming (GP) to be effective with larger and more complex programs. We design a particular kind of program structure to support module creation and reuse: modules are represented as \( \lambda \) abstractions and their reuse is achieved through an implicit recursion. A type system is used to preserve this structure. The structure of \( \lambda \) abstractions and implicit recursion also provides structure abstraction in the program. Since the GP paradigm evolves program structure and contents simultaneously, structure abstraction can reduce the search effort for good program structure. Most evolutionary effort is then focused on the search for correct program contents rather than the structure. Experiments on the Even-N-Parity problem show that, with the structure of \( \lambda \) abstractions and implicit recursion, GP is able to find a general solution which works for any value of \( N \) very efficiently.

1 Introduction

Genetic Programming (GP) [Koza, 1992] is increasing in popularity as the basis for a wide range of learning algorithms. The success of GP is generally attributed to its use of an evolutionary-based search strategy combined with a dynamic, tree-structure representation of the programs. Recently, there have been many attempts to enhance GP performance [Koza et al., 1997]. Among them, supporting modules in program representation has been shown to be beneficial [Koza, 1994; Rosca and Ballard, 1996]. Module creation and reuse can enhance GP in its ability to scale to larger and more complex problems.

In this research, a structure which supports module creation and reuse is incorporated in the program representation. In this structure, modules are represented as \( \lambda \) abstractions (see Section 2.2) and module reuse is achieved through an implicit recursion (see Section 3.1). A type system (see Section 3.3) is used to preserve the structure during program evolution.

This style of module creation and reuse provides the following advantages:

- Module creation is neither a random process nor a prefixed condition. A randomly generated module may or may not be beneficial to the problem to be solved. On the other hand, a hard-wired module template precludes the generation of more advantageous program structures. Our approach is to generate modules dynamically based on the recursion structures specified in advance by the users. This allows the exploration of beneficial program structures under the constraints of the users’ specified conditions (see Section 3.2).

- Implicit recursion provides reuse without the possible side effect of infinite loops because there is no semantics of recursion present in the program. This is an inherent feature of implicit recursion. Such a condition not only relieves GP from handling infinite loops in a program but also from measuring the semantic elements of recursive programs which would be used in directing genetic operation.

- The structure of \( \lambda \) abstraction and implicit recursion provides structure abstraction (see Section 6) in the program. Since the GP paradigm evolves program structure and contents simultaneously, structure abstraction can reduce the search effort for good program structure. Most evolutionary effort is then focused on the search for correct program contents rather than the structure.

We have evolved solutions to the Even-N-Parity problem [Koza, 1992] using GP with the implicit recursion structure included in the program representation. The results indicate that with the structure of \( \lambda \) abstractions and implicit recursion, GP is able to find a general solution which works for any value of \( N \) very efficiently.

The paper is structured as follows: Section 2 provides background and related work; Section 3 presents our new strategy; Section 4 describes the experiments; Section 5 summarizes the results; Section 6 analyzes the results and Section 7 concludes.

2 Background

2.1 Recursion and Genetic Programming

Recursion is a general mechanism for program code reuse. When the name of a program appears in its program body, it is like making a new copy of the program code within the program. Recursion leads to more compact programs and can facilitate generalization.

Although a powerful reuse mechanism, recursion must be used carefully to be effective. There are two important criteria for a recursive program to be effective:

- there is a terminating condition (base case);
- the recursive calls are successively applied to arguments that converge towards the terminating condition.

A recursive program which fails to meet the two requirements may or may not produce a result depending on the program evaluation style. With lazy evaluation, where arguments of a function are evaluated only if their values are needed, it is possible for programs containing infinite loops to halt. On the other hand, strict evaluation requires a function’s arguments to be evaluated before the function body and can make such a program loop forever. How program evaluation style affects GP performance has to be further studied.

To evolve recursive programs, GP faces two challenges:

1. Handling infinite loops in a recursive program: An evolved program may produce infinite loops since the two criteria are not always met. [Brave, 1996] adopted a finite limit on recursive calls in his tree search program. Usually such a limit affects the evolution process since a good program may never be discovered if its evaluation requires more than the permitted recursive calls. This shortcoming, however, does not apply to a tree search program since the maximum number of iterations required to search a tree is its tree-depth. Thus stopping any recursion after tree-depth number of iterations does not affect the behavior of the program. Brave’s approach is not a general
solution for handling infinite loops. [Wong and Leung, 1996] employed a logic grammar to enforce the base-case structure in their Even-N-Parity programs. However, the convergence of recursive calls is not guaranteed. When infinite loops occur, they use an execution time limit to halt the program. [Clack and Yu, 1997] also imposed a recursion limit to evolve the “map” program. A “map” program applies the first argument (a function) to each element of the second argument (a list). Using the length of the input list as the recursion limit seemed to be a sensible decision.

2. Measuring the semantic elements of a recursive program to direct genetic operation: The GP paradigm uses a syntactic approach to build programs; no semantic analysis is supported. A recursive program which contains a perfect base-case statement may not be selected for reproduction since semantics is not considered in the GP fitness function. [Whigham and McKay, 1995] has identified this problem and suggested the application of genetic operators to be performed in an environment where semantic analysis is supported.

2.2 \( \lambda \) Abstractions and Genetic Programming

In our work, we allow \( \lambda \) abstractions in our programs. \( \lambda \) abstractions are local function definitions, similar to function definitions in a conventional language such as C. However, \( \lambda \) abstractions are anonymous and can not be invoked by name. The reuse of \( \lambda \) abstractions is through passing them as arguments to other functions.

\( \lambda \) abstractions are modules that are created and reused in our programs. Similar to the structure of an Automatic Defined Function (ADF) [Koza, 1994], a \( \lambda \) abstraction has formal parameters and a function body. However, the determination of its structure and the overall program structure is different from ADF. Koza adopted two approaches to defining the structure of a program with ADFs. The first one is to statically define it before the GP run. Once the structure is specified, every program in the population has the same structure. Genetic operators are custom-tailored to preserve the program structure. The second approach is to create various kinds of program structure randomly during the first generation. Program structure is then open to evolutionary determination [Koza, 1994 Ch. 21]. With predefined program structure, GP does not have the opportunity to explore more advantageous structures. When an unsuitable program structure is given, GP is doomed. On the other hand, leaving GP to determine the program structure among a wide range of possibilities (Koza limited to 3,906 [Koza, 1994 pp. 529]) can be computationally expensive. This research provides a middle ground: modules are determined dynamically by GP but the possible selections are reduced to those allowed by the higher-order functions present in the function sets. Higher-order functions are functions which take other functions as arguments. Each time one of these functions is used to create a program, the argument with a function type is created as a \( \lambda \) abstraction (more details are given in section 3). A priori knowledge about module creation and reuse is incorporated in functions and terminals to facilitate GP in determining the most effective program structure.

2.3 Even-N-Parity Problem

The Even-N-Parity has been used by Koza as a difficult problem for GP to solve [Koza, 1992]. This program takes a list of \( N \) boolean inputs; it returns True if an even number of inputs are True and False otherwise. The test cases consists of all the \( 2^N \) possible combinations of \( N \) inputs. As \( N \) is increased, the problem becomes exponential harder. Koza was able to use GP to solve the problem up to \( N = 5 \). When \( N = 6 \), none of his 19 runs found a 100%-correct solution.

Koza introduced ADFs as a mechanism for GP to construct modules for reuse. With ADFs, GP is able to solve this parity problem up to \( N = 11 \) [Koza, 1994 Ch. 6].

[Wong and Leung, 1996] proposed a general solution which can handle any \( N \) using recursion. Their approach is to construct a logic grammar to enforce the base-case program structure. Type knowledge can also be incorporated in the logic grammar. When both sets of information are present in the grammar to guide the evolution, GP is able to find the solution more efficiently than using ADFs. Within 60 runs which use 8 fitness cases (Even-3-Parity), 16 runs found a solution that can work on any value of \( N \). The generated programs, however, do not contain any subroutines, although an "or" function can be extracted from the programs. The construction of subroutines within recursive programs is not yet implemented in their work.

3 A New Strategy

A structure of \( \lambda \) abstractions and implicit recursion is used to evolve program solutions for the Even-N-Parity problem. Implicit recursion facilitates GP to generate general solutions which work for any value of \( N \). \( \lambda \) abstractions provide the module mechanism for GP to exploit the structure inherent in the Even-N-Parity problem. By combining implicit recursion with module mechanism, we anticipate performance advantages over previous work with the Even-N-Parity program.

3.1 FOLDR: Implicit Recursion

The issues that explicit recursion has highlighted in GP foster the idea of implicit recursion. There are three popular higher-order functions which can provide recursion without explicit recursive calls [Clack, Myers and Poon, 1995]:

- MAP: applies the first argument, a monadic operator (a function which takes one argument), to each element of the second argument, a list, to produce a list of the results.

- FOLD: places the first argument, a dyadic operator (a function which takes two arguments), between each of the items in the list. With FOLDL, the empty list is substituted with the given terminating value and the resulting expression is associated to the right. With FOLDL, the given terminating value is prefixed to the expression and the resulting expression is associated to the left.

- FILTER: applies the first argument, a predicate operator (a function which returns True or False), to each element in the second argument, a list, to produce a list containing items which satisfy the predicate operator.

An important characteristic of using implicit recursion is that the programs do not produce infinite loops. The terminating condition is an empty list, which is incorporated into the higher-order functions. Moreover, there are no recursion semantics in the programs as the recursion is performed in the higher-order functions. Implicit recursion is therefore an ideal mechanism to support recursion in GP.

For the Even-N-Parity problem, we include FOLDL in our language to provide implicit recursion because FOLDL produces a single output value and so does the Even-N-Parity program. For different problem domains, other higher-order functions might be more suitable.

3.2 \( \lambda \) Abstractions: Module Mechanism

In our work, the modules discovered by GP are represented as \( \lambda \) abstractions. We allow functions to take other functions as arguments. When an argument with function type is present, a \( \lambda \) abstraction is generated and is used as the actual value for that argument. By specifying primitive functions that require functional arguments, users can direct GP to build modules. For
example, suppose the user includes FOLDR in the function set. Since the first argument of FOLDR is a function type, each time FOLDR function is selected to construct the program, a λ abstraction will be generated as the function argument. FOLDR reuses the created λ abstraction through implicit recursion.

λ abstractions are constructed using the same function set as that used to create the main program. The terminal set, however, consists only of the arguments of the λ abstraction to be created: no global variables are allowed. Argument naming in λ abstractions follows a simple rule: each argument is named with a hash symbol followed by an unique integer, for example #1, #2. This is an easy way to create unique arguments within a λ abstraction. Furthermore, this consistent naming style allows crossover to be easily performed between λ abstractions with the same number of arguments.

3.3 Type System: Structure Preserving Engine

A type system is used to preserve the program structure of λ abstractions and implicit recursion in the programs. Initially, each primitive function and terminal is specified with type information. Meanwhile, the input and output types of the program to be evolved are specified. This information is used by the type system to select type-matched functions and terminals to construct type-correct programs.

The type information is specified using a type language. The abstract type syntax can be found in [Yu and Clack, 1997]. A function argument is denoted by the bracketed function type. For example, FOLDR is specified with the following type information:

**FOLDR** : (a->b->b) -> b -> [a] -> b

The type information indicates that FOLDR can take 3 arguments: the first one is a function, the second one is a value and the third one is a list. It returns a single value. Additionally, the first argument is a function which can take two arguments and returns one value. This function argument will be created as a λ abstraction of two arguments. Notice that FOLDR is a polymorphic function whose type signature contains type variables. The type system instantiates these type variables when the function is selected to construct the program. It is very important to assure that type variables are instantiated consistently so that the constructed program is type-correct.

By allowing type variables in our type language, we have enhanced the generality of functions in the function set. For example, FOLDR can provide implicit recursion for many different types of arguments. More details about polymorphism and generality in GP can be found in [Yu and Clack, 1997].

The type system also performs type checking during the genetic operation of crossover and mutation. Thus, the structure of λ abstractions and implicit recursion can be preserved throughout the evolutionary process. We perform the "point-typing" structure-preserving crossover [Koza, 1994 pp. 532] in our programs: a point is first selected from the first parent program; depending on the source of the node (the main program or a λ abstraction body), a node with the same source is selected from the second parent program. Moreover, if the crossover point in the first parent program is inside a λ abstraction, the crossover point in the second parent program has to be inside a λ abstraction which has the same number and type of arguments as the λ abstraction where the first point was selected. This restriction assures that the offspring will not contain any unbound global variables.

Our crossover operation is also allowed to be performed on λ abstraction nodes; we term this "λ modular crossover" as it results the swapping of a λ abstraction module in one program with a λ abstraction module in another program. This operation is similar to the modular crossover in [Kinnear, Jr., 1994]. However, with our type system, the problem of arguments mismatching mentioned in Kinnear's work does not occur in our implementation. Each λ abstraction node is annotated with a type which indicates the number and type of its arguments. The type system restricts the λ modular crossover to be performed between two λ abstractions which have the same number and type of arguments.

4 Experiments

4.1 Parameters and Primitives

As in [Wong and Leung, 1996], we use a population size of 500 and a maximum generation of 50. Meanwhile, a maximum tree depth of 4 for the main program is imposed. A λ abstraction is considered as one single node in a program parse tree as it performs one task just like those primitive functions in the function set. The maximum tree depth allowed for a λ abstraction is also 4. The crossover rate is 100%.

A FOLDR expression inside another FOLDR expression creates nested recursion. Nested recursive programs require considerable amount of time and space to evaluate. We therefore limit the depth of the nested recursion to be 100, which we think is powerful enough to handle the Even-N-Parity problem. The following are the primitives and their types used in our experiments:

**Output Type:** bool
**Argument Type:** [bool]
**Terminal Set:** T = [L:: [bool]]

**Function Set:**
F = [HEAD:: [a] -> a,
TAIL:: [a] -> [a],
AND:: bool -> bool -> bool,
OR:: bool -> bool -> bool,
NAND:: bool -> bool -> bool,
NOR:: bool -> bool -> bool,
FOLDR:: (a->b->b)->b->[a]->b]

4.2 Selection of Fitness Cases

The fitness cases of the Even-2-Parity and the Even-3-Parity are selected to evaluate our programs. There are 2^2 + 2^3 = 12 fitness cases. A general Even-N-Parity program can handle any value of N, which may be either even or odd. The Even-2-Parity fitness cases help GP to learn to handle an input list with an even number of items while the Even-3-Parity fitness cases train GP to work on an input list with an odd number of items. With this set of fitness cases, we hope the generated programs can be general solutions which work for any value of N.

4.3 Design of Fitness Function

The fitness function used is the same as that used by Koza [Koza 1999, pp. 160] except that a run-time error is punished. Each program is evaluated against all of the fitness cases. When a correct result is produced for a fitness case, the program receives a 1; otherwise, it receives a 0. If run-time error has been flagged, fitness is reduced by 0.5. The fitness of a program is the sum of the fitness values for all of the fitness cases. The maximum fitness value of a program in this experiment is 12.

5 Results

We made 60 runs and 57 of them found a solution. Moreover, all 57 are general solutions which work for any N number of inputs. To facilitate direct comparison, we have followed Koza's method [Koza, 1992 Ch. 8] to measure the performance of our new strategy. Figure 1 shows the performance curves of the experiments.
Figure 1: Performance curves for the Even-N-Parity program with population size of 500.

The curve \( P(M,i) \) shows the cumulative probability of success to solve the problem by generation \( i \) using a population size of 500. The curve \( I(M,i,z) \) indicates the number of programs that have to be processed to produce a solution by generation \( i \) with probability \( z \). In this work, the probability \( z \) is set to 99%. The curve of \( I(M,i,z) \) reaches a minimum value of 14,000 at generation 3 (marked on the figure). This means that if this problem is run through to generation 3, processing a total of \( I(M,i,z) = I(500, 3, 0.99) = 14,000 \) individuals (i.e. 500 x 4 generations x 7 runs) is sufficient to yield a solution of this problem with 99% confidence. Since we use 12 fitness cases to test our programs, the number of fitness cases to be processed is 14,000 x 12 = 168,000. Compared with other related works by Koza using ADFs [Koza, 1994, pp. 196] and by Wong and Leung using a Generic Genetic Programming (GGP) system [Wong and Leung, 1996], our performance excels. Table 1 summarizes the performance of these 3 different approaches in evolving Even-N-Parity program.

<table>
<thead>
<tr>
<th>Results</th>
<th>Implicit Recursion + ( \lambda ) abstraction</th>
<th>GGP</th>
<th>GP with ADFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs</td>
<td>Even-N-Parity</td>
<td></td>
<td>Even-7-Parity</td>
</tr>
<tr>
<td>Runs/Sucess</td>
<td>60/57</td>
<td>60/17</td>
<td>29/10</td>
</tr>
<tr>
<td>Minimum ( I(M,i,z) )</td>
<td>12</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>Fitness Cases Processed</td>
<td>168,000</td>
<td>1,760,000</td>
<td>184,320,000</td>
</tr>
</tbody>
</table>

6 Analysis and Discussion

The results of our experiments indicate that by using the structure of \( \lambda \) abstractions and implicit recursion, GP is able to evolve very efficiently Even-N-Parity programs which work for any N. Koza’s ADFs provide a mechanism for module creation and reuse which has helped GP learn the Even-N-Parity program up to N=11 but its performance details are not reported. In Table 1, we use the performance information for Even-7-Parity on [Koza, 1994 pp.195]. The GGP system by Wong and Leung uses recursion to support program code reuse (however, no module creation mechanism is provided): the GGP system can learn the Even-N-Parity programs which work for any N more efficiently than the ADF approach. Our GP system supports both recursion and modules. With the structure of \( \lambda \) abstractions and implicit recursion, our system can learn the Even-N-Parity programs which work for any value of N and it can do this by processing a much smaller number of programs than the number required either by the ADF approach or by the GGP system (see Table 1). More than 50% of our 60 runs obtained a solution before generation 5 and two of them found a solution during generation 0 through random search under the constraints of the specified program structure. This is an exceptional performance compared with any other previous work with the same problem.

Besides the benefits of recursion and modules, we believe there is one more factor which contributes to such an exceptional performance:

**Higher-order functions provide structure abstraction in the program parse trees.** The type system protects this structure abstraction and helps GP to find good program structures during program evolution.

The ability of traditional GP to build good solutions from partial solutions hierarchically has been challenged [O’Reilly and Oppacher, 1995]. The module mechanisms of ADFs, MA and ARL can facilitate GP in hierarchical processing by abstracting program contents. Our module mechanism of \( \lambda \) abstractions promotes the use of hierarchy further by supporting program "structure abstraction". As an argument to a higher-order function, a \( \lambda \) abstraction is constrained to sit underneath the higher-order function in the program tree hierarchy. During program evolution, this two-layer-hierarchy program structure grouping is protected from disruption by our type system; crossover can only change its contents but not its structure. In other words, GP uses the two-layer-hierarchy structure as one unit to exploit the most advantageous program structure. Figure 2 shows three program structure groupings that have identical structure. Note that they may have different contents since the three \( \lambda \) abstractions may be different.

Figure 2: Program structure grouping for FOLDR.

We analyzed the data collected from 10 test runs to see whether structure abstraction is beneficial to GP. During generation 0, various program structures are created. All those programs with more than two FOLDRs had fitness value 0. A few programs with no FOLDR had above average fitness. But all the programs which contain either 1 or 2 occurrences of FOLDR receive better than average fitness. All 57 correct Even-N-Parity programs generated from our experiments also contain either 1 or 2 occurrences of FOLDR (see Table 2). This suggests that the structure of the Even-N-Parity program is generally determined at generation 0. Most evolutionary processes search for the correct program contents to fill in the program structure. Table 2 displays all 57 generated correct Even-N-Parity programs. Those functions in italics are generated functions represented as \( \lambda \) abstractions in the programs. They are anonymous functions in the programs but we provide them with names here for easy reference. The Truth Table for these generated \( \lambda \) functions is presented in Table 3. Note that those \( \lambda \) abstractions which compute xor might contain very different code. The values True and False in Table 2 indicate expressions which produce True or False under all conditions.
Table 2: Generated Correct Programs

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Even-N-Parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>nor (foldr xor (head L) (tail L)) False</td>
</tr>
<tr>
<td>9</td>
<td>foldr xor (nor (head L) (head L)) (tail L)</td>
</tr>
<tr>
<td>6</td>
<td>nor (foldr xor (head L) (tail L)) (foldr xor (head L) (tail L))</td>
</tr>
<tr>
<td>6</td>
<td>foldr xor (nand (head L) (head L)) (tail L)</td>
</tr>
<tr>
<td>6</td>
<td>nand (foldr or (head L) (tail L)) (foldr xor (head L) (tail L))</td>
</tr>
<tr>
<td>5</td>
<td>nand (foldr xor (head L) (tail L)) True</td>
</tr>
<tr>
<td>2</td>
<td>foldr xor (foldr xand (head L) (tail L)) (tail L)</td>
</tr>
<tr>
<td>1</td>
<td>nor (foldr xor (head L) (tail L)) (foldr xor (head L) (tail L))</td>
</tr>
</tbody>
</table>

Table 3: Truth Table

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>xor</th>
<th>exor</th>
<th>xand</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

Much research work has asserted that type constraints can reduce the number of ways to construct programs and improve GP performance [Montana, 1995; Haynes, Schoenefeld and Wainwright, 1996; Clack and Yu, 1997]. [Harris, 1997] used abstraction on user-defined types to enforce a hierarchy in the program parse tree, but in a specialized domain and with the hierarchy defined by the users' knowledge of that domain: this explicit program structuring method improves GP performance in his image template matching experiments. In our work, we use type constraints and higher-order functions in a general context to support structure abstraction and enhance GP performance. We anticipate that type constraints can assist GP learning in other ways yet to be discovered.

7 Conclusion

The structure of \( \lambda \) abstractions and implicit recursion provides GP with an effective mechanism to perform module creation and reuse. We have demonstrated its power by evolving Even-N-Parity programs. By incorporating this mechanism, GP is able to evolve a correct program by processing far fewer programs than the number required in any previous work. All evolved correct programs are general solutions which work well for any number of inputs. As we said in the introduction, three main factors have contributed to this result:

- Module creation is neither a random process nor a prefixed condition. Instead, modules are generated dynamically based on the recursion structures specified in advance by the users. This allows the exploration of beneficial program structures under the constraints of the users' specified conditions.
- Implicit recursion provides reuse without the possible side effect of infinite loops since there are no recursion semantics present in the program. This not only relieves GP from handling infinite loops in a program but also from measuring the semantic elements of recursive programs which would be used in directing genetic operation.

- The structure of \( \lambda \) abstractions and implicit recursion provides structure abstraction in the program. As the GP paradigm evolves program structure and contents simultaneously, abstraction of structure can reduce the search effort for good program structure. Most evolutionary effort is then focused on the search for correct program contents rather than the structure.

References


