# Solving Real-valued Optimisation Problems using Cartesian Genetic Programming

Genetic Programming Track

James Alfred Walker
Intelligent Systems Group, Department of Electronics
University of York, Heslington
York, YO10 5DD, UK
jaw500@ohm.york.ac.uk

Julian Francis Miller
Intelligent Systems Group, Department of Electronics
University of York, Heslington
York, YO10 5DD, UK
jfm7@ohm.york.ac.uk

## ABSTRACT

Classical Evolutionary Programming (CEP) and Fast Evolutionary Programming (FEP) have been applied to real-valued function optimisation. Both of these techniques directly evolve the real-values that are the arguments of the real-valued function. In this paper we have applied a form of genetic programming called Cartesian Genetic Programming (CGP) to a number of real-valued optimisation benchmark problems. The approach we have taken is to evolve a computer program that controls a writing-head, which moves along and interacts with a finite set of symbols that are interpreted as real numbers, instead of manipulating the real numbers directly. In other studies, CGP has already been shown to benefit from a high degree of neutrality. We hope to exploit this for real-valued function optimisation problems to avoid being trapped on local optima. We have also used an extended form of CGP called Embedded CGP (ECGP) which allows the acquisition, evolution and re-use of modules. The effectiveness of CGP and ECGP are compared and contrasted with CEP and FEP on the benchmark problems. Results show that the new techniques are very effective.

## Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search

## General Terms

Algorithms, Design, Performance

## Keywords

Cartesian Genetic Programming, Real-valued Function Optimisation, Modules, Evolutionary Programming


*GECCO* '07 July 7-11, 2007, London, England, United Kingdom.


## 1. INTRODUCTION

In the past, Evolutionary Programming (EP) [5] has been successfully applied to many numerical and combinatorial optimisation problems [2, 3, 4]. However, one of the key problems with methods for tackling such problems is concerned with avoiding slow convergence to the optimum or convergence to sub-optima. In a technique called Fast Evolutionary Programming (FEP), Yao and Liu found that it was more effective to using a Cauchy mutation operator rather than the more usual Gaussian mutation used in EP [17, 18]. They showed that FEP improved convergence to the optimum on a series of multi-modal functions with many local minima in comparison with Fogel's EP (which Yao and Liu refer to as classical EP (CEP)) [17, 18].

In this paper, we have drastically changed the representation of the genotype for these problems. Instead of directly manipulating the real values themselves, we evolve a computer program that writes out the real-value arguments to the optimisation functions. Our motivation for doing this is the following. Firstly, we hope that the change in dimensionality and structure of the search space will be changed and perhaps, may make some of these problems easier to solve. Secondly, we hope to exploit a characteristic of the CGP representation of programs. CGP has been shown to benefit from the form and degree of redundancy that is naturally present in its genotype. Thirdly, CGP has been shown to be a highly effecient technique (in comparsion with other GP techniques) on a number of GP benchmark problems. Our final motivation was to look at the performance of an extended form of CGP, called Embedded CGP (ECGP) [12] in comparison with CGP, EP and FEP.

ECGP incorporates ideas from Module Acquisition [1], that allows the automatic acquisition, evolution and re-use of partial solutions in the form of modules. Previous work [13, 15] has shown ECGP to be more computationally efficient than CGP on a range of digital circuit problems and the speedup grows with problem difficulty.

Recently, CGP and ECGP have been applied to the Genetic Algorithm (GA) based Hierarchical-if-and-only-if (H-IFF) [14], one-max and order-3 deceptive problems [16]. CGP and ECGP found solutions to the H-IFF problem more easily than published attempts using a GA. CGP also found solutions to the one-max and order-3 problems more easily than simple and generational GAs and the GAuGE [11] and LINKGAuGE [9] systems. This paper builds on the work

from [14, 16] by modifying the technique to produce real-valued numbers instead of binary strings, so that it can be applied to real-valued optimisation problems.

The plan for the paper is as follows: Sections 2 and 3 give an overview of CGP and ECGP. In section 4, we describe our approach of applying CGP and ECGP to real-valued optimisation problems. The details of our experiments are shown in section 5 followed by the results and comparisons in section 6. Section 7 gives conclusions and suggestions for future work.

## 2. CARTESIAN GENETIC PROGRAMMING (CGP)

Cartesian Genetic Programming is a form of Genetic Programming (GP) [6] invented by Miller and Thomson [8], for the purpose of evolving digital circuits. However, unlike the conventional tree-based GP, CGP represents a program as a directed graph (that for feed-forward functions is acyclic), which is only modified by mutation. The benefit of this type of representation is that it allows the implicit re-use of nodes in the directed graph. CGP is also similar to another technique called Parallel Distributed GP, which was independently developed by Poli [10]. Originally CGP used a program topology defined by a rectangular grid of nodes with a user-defined number of rows and columns. However, later work on CGP showed that it was more effective when the number of rows is chosen to be one [19]. This one-dimensional topology is used throughout the work we report in this paper.

CGP uses a fixed length representation, where the genotype consists of a list of integers, encoding the function and connections of each node in the directed graph. However, the number of nodes in the directed graph (phenotype) can vary but is bounded, as every node encoded in the genotype does not have to be connected. This allows areas of the genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This unique type of neutrality has been investigated in detail [8] and found to be extremely beneficial to the evolutionary process on the problems studied.

Each node in the directed graph is encoded in the genotype by a number of genes, determined by the arity of the function the node represents. For each encoded node, the first gene encodes the node's function (using values from a lookup table) and the remaining genes encode the node's input connections (using the index label of the node or program input). The nodes take their inputs in a feed forward manner from either the output of a previous node in the directed graph or from the program inputs (terminals). The program inputs are labelled from 0 to *n-1* where $n$ is the number of program inputs. The nodes in the directed graph are also labelled sequentially starting from $n$ to $n+m-1$ where $m$ is the number of nodes in the directed graph. If the problem requires $k$ program outputs then $k$ integers are added to the end of the genotype, each one encoding the index of the node in the directed graph where the program output is taken from. These $k$ integers are initially set as pointers to the outputs of the last $k$ nodes encoded in the genotype. Figure 1 shows a CGP genotype and corresponding phenotype for the 8-bit one-max problem, whilst Figure 2 illustrates the decoding procedure of a CGP genotype.

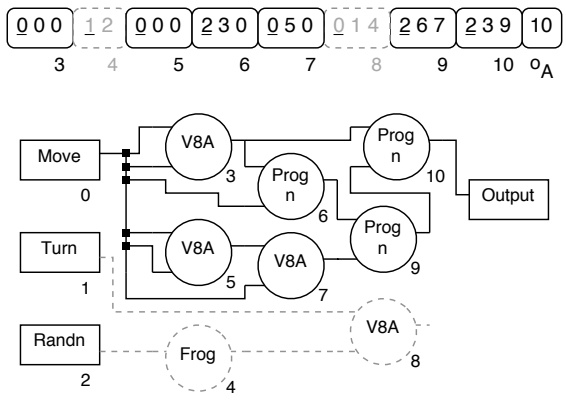


**Figure 1: CGP genotype and corresponding phenotype for the 8-bit H-IFF problem. The underlined genes encode the function of each node using the lookup table: V8A(0), Frog(1), Progn(2). See Section 4 for function details. The index labels are shown underneath each program input and node. The inactive areas of the genotype and phenotype are shown in grey dashes.**

## 3. EMBEDDED CARTESIAN GENETIC PROGRAMMING (ECGP)

ECGP incorporates ideas from Module Acquisition [1] with CGP, to allow the automatic acquisition, evolution and re-use of partial solutions (referred to as modules) [13]. Thereby giving CGP a form of Automatically Defined Function (ADF) [7]. This paper only gives a brief overview of ECGP due to space restrictions. For information on the technical details of ECGP, please refer to [13].

ECGP uses a modified CGP *genotype* making it a bounded variable length representation (in terms of the number of encoded nodes in the genotype and the number of genes used to encode each node). The number of nodes encoded in the genotype decreases when sections of the genotype are encapsulated into modules (when modules are created by the compress operator) and increases when modules are expanded back into sections of the genotype (when modules are destroyed by the expand operator). The number of genes used to encode the inputs of a node in the genotype can vary as a result of either module mutation increasing or decreasing the number of module inputs (therefore affecting the number of genes required to encode the module), or a module being introduced into the genotype (requiring extra genes to encode all of the module inputs).

Modules are capable of having multiple outputs, but the CGP representation only encodes nodes with single outputs, therefore each gene is now represented using a pair of integers rather than just a single integer, as in CGP. For each gene encoding a node input, the first integer encodes the node index (as in CGP), whilst the second integer encodes the function output used.

Using a pair of integers to encode each function gene allows the introduction of node types into the ECGP representation. Node types allow the identification of nodes encoded in the genotype representing: primitive functions (node type 0), modules that contain an original section of the genotype (node type I) and modules that contain a re-used section

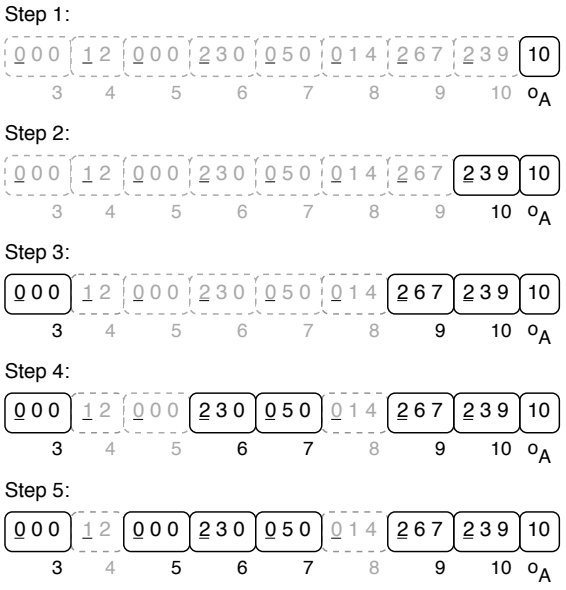


**Figure 2: Decoding the CGP genotype from Figure 1. Step 1: Output A ($o_A$) connects to the output of node 10, move to node 10. Step 2: Node 10 connects to the output of nodes 3 and 9, move to nodes 3 and 9. Step 3: Nodes 3 and 9 connect to the output of nodes 6 and 7, and program input 0, move to nodes 6 an 7. Step 4: Nodes 6 and 7 connect to the output of nodes 3 and 5, and program input 0, move to node 5 (as node 3 has already been processed). Step 5: Node 5 only connects to program input 0, therefore the genotype is now decoded.**

of the genotype (node type II). Operators act differently on the nodes encoded in the genotype depending on their node type. Node types are encoded as the second integer of the function gene of every node, the first integer encodes the primitive function (as in CGP) or module (using values from a lookup table). Figure 3 illustrates the differences between the CGP and ECGP representations.

Modules are represented using a modified ECGP representation, which also encodes structural information about the module. Four extra integers are added to the beginning of the module genotype to encode the module identifier, the number of inputs and outputs of the module, and the number of nodes the module contains. Currently, a module can only contain nodes of type 0, to prevent bloat inside the module. Once a module is created, it is added to the module list (a dynamic extension of the function list) and can be re-used whilst the module remains in the module list, along with the primitive functions. The module list is updated every generation to contain the module list of the fittest individual in the population (in accordance with the 1 + 4 evolutionary strategy).

The module genotype can be evolved by the module mutation operators independently of the ECGP genotype. Either a structural mutation can occur, which affects the number of module inputs and outputs, or a point-mutation can occur, which affects the nodes contained in the module (as mutation would occur in CGP).

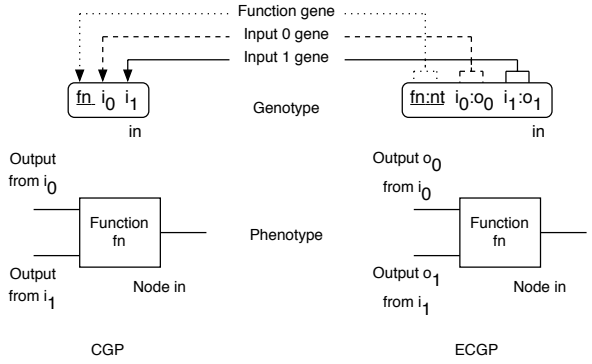


**Figure 3: CGP and ECGP genotypes and corresponding phenotypes for a single node. The components of each gene are labelled as follows: function (*fn*), node type (*nt*), node indexes that the node inputs are taken from ($i_0$, $i_1$), node outputs that the node inputs are taken from ($o_0$, $o_1$), index of this node (*in*).**

# 4. APPLYING CGP AND ECGP TO REAL-VALUED OPTIMISATION PROBLEMS

Previous work by Walker and Miller [15, 14, 16] has shown that CGP and ECGP can be used to evolve solutions to problems traditionally associated with Genetic Algortihms (GAs). The method chosen was heavily influenced by a GP benchmark problem called the Lawnmower Problem [7]. In the lawnmower problem, GP is used to evolve a set of commands to move a lawnmower around a lawn, which has been divided into a $n$ x $m$ grid of squares, where $n$ and $m$ denote the width and height of the lawn respectively. The lawnmower cuts the grass in each square it visits, with a solution being found when the lawnmower has visited every square of the grid, therefore cutting all the grass.

The approach used in the lawnmower problem was modified so that instead of evolving a set of instructions to control a lawnmower moving on a 2-dimensional lawn, a set of commands for a moving a tape head on a 1-dimensional tape was evolved. In a similar way to the manner of discretizing the lawn in the lawnmower problem, the tape is divided into $n$ squares, where $n$ was the number of bits in the GA. Initially, all squares on the tape are blank, and the tape head is positioned in the centre square of the tape (similar to the lawnmower starting in the centre square of the lawn). In a single command, the tape head can move one square or jump a number of squares in the direction the tape head is facing (left or right). If the tape head moves off one end of the tape, it re-appears in the square at the opposite end of the tape (just as the lawnmower would in the lawnmower problem). When the tape head visits a square, the value of the square is changed according to the rule in Equation 1.

$$if(square == blank \;||\; square == 1), \quad square = 0 \qquad (1)$$
$$if(square == 0), \quad square = 1$$

Therefore, the tape head behaves like the bit-flip operator found in GAs. Once the set of commands has been executed, the tape head will have produced a bit-string of length $n$ containing the symbols: - (blank), 0 and 1, which can be

evaluated as an individual in a GA. A blank (-) in the bit-string does not contribute towards the fitness score, as we only want to generate bit-strings containing 0's and 1's.

In the work described in this paper, we have modified the approached used previously so that it can be applied to real-valued optimisation problems. The approach still evolves a sequence of commands, which affect the movement of a tape-head on a piece of tape, as in the [15, 14, 16]. However, the main difference in our approach is that we have changed the rule that alters the value of a square on the tape. Instead of just altering the value of a square between 0 and 1, as in Equation 1, the new rule permits a square to have an integer value between 0 and 9. This allows the tape to produce a real-valued number, where each square represents a significant figure of a real-valued number. If more than one real-valued number is to be generated, each real-valued number is allocated a set number of squares on the tape, which is determined by the user, and also determines the overall length of the tape. When the tape head visits a square, the square's value is either incremented or decremented by 1. We keep the values written in squares in the range 0 to 9 by applying a modulo 10 operation. The new rule is given in Equation 2.

$$\begin{aligned} &if(increment == true) \\ &\qquad square = (square + 1) \,\%\, modulusValue \\ &if(increment == false), \\ &\qquad square = (square - 1) \,\%\, modulusValue \end{aligned} \tag{2}$$

The range of the real-valued numbers produced on the tape can also be restricted by altering the modulus value for specific squares. For example, this allows us to limit the first square on the tape to allow only 0's and 1's by setting the modulus value of the first square equal to 2. If a modulus value of 2 was set for every square on the tape, the rule in Equation 2 would be equivalent to the old rule in Equation 1 and the tape would produce a binary string rather than a real-valued number. Another purpose of restricting the range of a square's value using modulus 2, would be to represent the sign of the real-valued number produced on the tape. This restriction is applied to the first square allocated to every real-valued number represented on the tape (so that we can deal with the sign of numbers). If this square's value is 0 it represents a positive number, whilst a 1 represents a negative number. The position of the decimal point on the tape is a user-defined parameter, as this is related to the problem to be solved and the range of real values allowed in the optimisation problem.

Once a set of commands has been executed, the tape head will have produced a string containing the symbols between 0 and 9 (possibly less depending on the user-defined range restrictions), as shown in Figure 4. The tape is then decoded into a series of real-valued numbers, as shown in Figure 5, which can be evaluated as an individual in an EP system. The fitness function used here is described in Section 5.

Although the proposed approach changes the nature of the real-valued function optimisation problems, we hope that changing the dimensionality and neutral interconnectedness of the genotype space may alleviate problems typical of some forms of EP - early convergence on sub-optima. By using a CGP representation, we have allowed the possibility that small changes to the genotype can produce a big change in the real-valued numbers produced on the tape. This acts a little like having an implicit variable rate mutation operator. We hope that this approach might alleviate the strong sensitivity of algorithm peformance to the value of the mutation probability that EP approaches commonly suffer from.

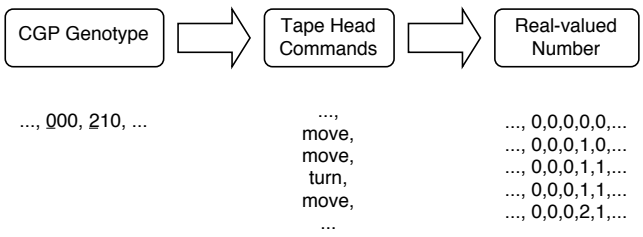


**Figure 4: The three step procedure for producing an real-valued number data on the tape from the CGP genotype, via a set of tape head commands.**

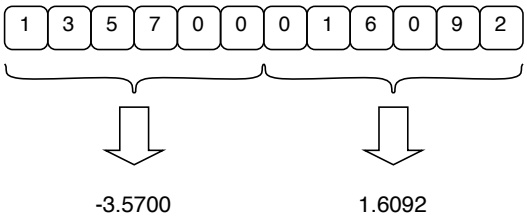


**Figure 5: Decoding the real-valued number data on a tape (consisting of 12 squares) into two real-valued numbers (each consisting of 6 squares) suitable for evaluation. The decimal point occurs after the second square in each number.**

The CGP program has four program inputs, three of which are constrained versions of those used in the lawnmower problem: *move* - moves the tape head one square in the direction it is facing and changes the value of the new square according to Equation 1, *turn* - alters the direction the tape head travels along the tape from right to left or vice versa and *random constant* - a random number, $r$, chosen at the start of each independent run, where $0 <= r < n$. The other program input is *transition*, which alters Equation 2 to either increment or decrement the value of a square when it is visited by the tape-head. Move, turn and transition also return a constant, 0, so mathematical operations can also be performed on the program inputs.

The function set used contains the same functions as the lawnmower problem: *progn* - a program node, which executes the graph connected to its first input, followed by the graph connected to its second input and returns the result of the second input, *v8a* - performs addition on its two inputs and returns the result, and *frog* - moves the tape head by a number of squares specified by its input in the direction it is facing and changes the value of the new square according to Equation 2.

## 5. EXPERIMENT DETAILS

In this paper, CGP and ECGP are applied to real-valued function optimisation problems $f_8(x)$, $f_{16}(x)$ and $f_{18}(x)$ from Yao and Liu's paper [17] and are shown in Table 1. The number of real-valued inputs and the range allowed for each

**Table 1: The three multi-modal functions used to test CGP and ECGP.**

| Function | Dimension | Range | $f_{min}$ |
|---|---|---|---|
| $f_8(x) = \sum_{i=1}^{n} -x_i \sin(\sqrt{x_i})$ | 30 | [-500, 500] | -12569.5 |
| $f_{16}(x) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4_2^4$ | 2 | [-5, 5] | -1.0316285 |
| $f_{18}(x) = [1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)] \times [30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)]$ | 2 | [-2, 2] | 3 |

input is also shown for each function, along with the known minimum value of the function. All three functions are multi-modal functions with many local minima, thereby allowing many possibilities for algorithms to converge early on local minima. Also, the three functions chosen allow us to see how CGP and ECGP perform on low and high dimensional functions and also functions with small and large ranges for each dimension.

For each function, the length of tape used by the CGP program had to be decided upon. One square is needed to represent the sign of each function input and one square is needed for each significant figure in the range of each function. The evolved function inputs need to be relatively accurate, so an extra 8 decimal places (each represented by a tape square) were allowed, in addition to the number of decimal places present in the known minimum value of each function. The calculation of the number of tape squares required for each function input is summarised in Equation 3. The total length of the tape is calculated using Equation 4.

$$\begin{aligned} squaresPerFunctionInput &= sign \\ &+ significantFiguresOfRange \\ &+ decimalPlacesOfMin + 8 \end{aligned} \quad (3)$$

$$\begin{aligned} tapeLength &= functionDimension \\ &\times squaresPerFunctionInput \end{aligned} \quad (4)$$

The total length of tape we used for each function is shown in Table 2, as well as the modulus values for each tape square of a number, the position where the decimal point occurs (in tape squares) and the number of generations allowed for a each experiment. The number of generations allowed for each function differs from those found in [17], as a different population size was used. Instead, we calculated the total number of evaluations that the FEP approach used [17]. We ensured that the number of function evaluations used for CGP and ECGP was the same.

The parameters used by CGP and ECGP are shown in Table 3. These are taken from [13]. The probabilities and rates of the various mutation operators were shown to be optimal in a series of previous experiments.

The fitness function used by CGP and ECGP is the same as that used in EP and is simply a minimisation function, which awards higher fitness depending on the closeness to the known global minimum value of the function.

## 6. RESULTS AND DISCUSSION

For each experiment, the best of generation minimum fitness was taken when the total number of generations was reached. The average and standard deviation was calculated from the minimum fitness figures from all 50 independent runs and are shown in Table 4 along with the corresponding figures for CEP and FEP from [17]

**Table 3: The parameter settings used for CGP and ECGP (* - ECGP only). The mutation rate is expressed as a percentage of the genotype length. The operator rates and probabilities are per generation.**

| Parameter | Value |
|---|---|
| Population size | 5 |
| Number of nodes | 2,000 |
| Genotype point mutation rate | 2% (40 Genes) |
| Compress/Expand probability * | 0.1/0.2 |
| Module point mutation probability * | 0.04 |
| Add/Remove input probability * | 0.01/0.02 |
| Add/Remove output probability * | 0.01/0.02 |
| Maximum module size * | 5 nodes |
| Number of independent runs | 50 |

In the 50 independent runs, CGP found the global minimum value in runs 5, 4 and 11, and ECGP found the global minimum value in runs 7, 5 and 11 for functions $f_8(x)$, $f_{16}(x)$ and $f_{18}(x)$ respectively. This shows that both CGP and ECGP are capable of finding the optimal minimum value for all functions tested. We anticipate that other runs didn't end in finding the global minimum because of the restricted number of evaluations used (not because of early convergence). Previous findings concerning CGP and ECGP [12, 13, 14, 16] have shown that both CGP and ECGP are capable of producing a 100% success rate on all runs that are long enough. This is because the neutrality in the representation appears to allow the techniques to avoid convergence. In future work, we intend to evaluate the lack of convergence by continuing the evolution until the runs have found the global minimum (providing the global minimum value is known), rather than stopping the runs by a certain generation.

Firstly, the results show both CGP and ECGP can both be successfully applied to real-valued function optimisation problems. Comparing the results of CGP and ECGP with CEP and FEP on functions $f_{16}(x)$ and $f_{18}(x)$ shows that all techniques are capable of finding the global minimum value, within a certain degree of error. In fact, the mean for CGP and ECGP is fractionally lower than the mean for FEP on the function $f_{18}(x)$. However, this is unlikely to be statistically significant. CGP and ECGP give better results for $f_8(x)$, than CEP (which is converged) but is not as good as FEP.

The main difference in our results for all three functions is that CGP and ECGP have a much larger standard deviation than CEP and FEP. This indicates the distribution of the best solutions found from each independent run in CGP and ECGP is more unpredictable than the distributions of best

**Table 2: The tape parameters used for each function.**

| Function | Tape Squares per Function Input | Tape Length | Modulus Value per Tape Square | Decimal Point Position | Total Number of Generations |
|---|---|---|---|---|---|
| $f_8(x)$ | 13 | 390 | [2,5,10,10, ..., 10] | 4 | 225,025 |
| $f_{16}(x)$ | 17 | 34 | [2,5,10,10, ..., 10] | 2 | 2,525 |
| $f_{18}(x)$ | 10 | 20 | [2,2,10,10, ..., 10] | 2 | 2,525 |

**Table 4: Average minimum fitness and standard deviation figures of CGP, ECGP, CEP and FEP applied to functions $f_8(x)$, $f_{16}(x)$ and $f_{18}(x)$.**

| | CGP | | ECGP | | CEP | | FEP | |
|---|---|---|---|---|---|---|---|---|
| Function | Mean | Std Dev | Mean | Std Dev | Mean | Std Dev | Mean | Std Dev |
| $f_8(x)$ | -11239.7 | 1620.9 | -11290.6 | 1493.2 | -7917.1 | 634.5 | -12554.5 | 52.6 |
| $f_{16}(x)$ | -1.03 | $6.55 \times 10^{-4}$ | -1.03 | $6.08 \times 10^{-4}$ | -1.03 | $4.9 \times 10^{-7}$ | -1.03 | $4.9 \times 10^{-7}$ |
| $f_{18}(x)$ | 3.0 | $4.64 \times 10^{-3}$ | 3.01 | $9.03 \times 10^{-3}$ | 3.0 | 0 | 3.02 | 0.11 |

solutions for CEP and FEP. This means that it is harder to guarantee that a run of CGP and ECGP will always produce a good set of optimal inputs for each function. This is not really surprising given the small populations used.

Comparing the results of CGP and ECGP shows that both techniques perform on a par with the some of the best techniques. However, it is clear that ECGP does not appear to offer any real advantages over CGP on these problems. This suggest that good programs that control the tape writing head may not be modular. However, further work is needed to clarify this further. It accords also with our comparative results for a number of binary GA problems [16].

# 7. CONCLUSION AND FUTURE WORK

In this paper, we have presented for the first time an approach to real-valued function optimisation using CGP and ECGP. The approach uses CGP and ECGP to evolve a list of commands for a tape-head, which produces a series of real-valued numbers on a tape by executing the commands. The results show that CGP and ECGP are both capable of producing results which are comparable to, if not better than CEP and FEP. However, we are not directly comparing the performance of CGP and ECGP with CEP and FEP, as the approach used by CGP and ECGP changes the nature of the test problems. Instead, we are using the comparison to judge whether the benefits of the CGP and ECGP representation (such as neutrality) could help avoid convergence on local minima and also whether CGP and ECGP could provide a feasible alternative to EP for real-valued function optimisation.

In the experiments in this paper, all runs were stopped at a particular generation so comparisons could be made. However, we have found before, that the nature of the CGP algorithm usually allows it to continue and not converge until a solution is found. In future work, we intend to allow CGP to run until it has reached a certain error threshold from the global minimum or maximum (depending on which the function requires), as in real-world optimisation problems, the optimal value may not always be known. Also, CGP will be applied to further real-valued optimisation problems to substantiate the results in this paper and to understand more about the tape and tape-head approach, as it appears to be quite sensitive to numerical values of the parameters chosen.

There are of course a number of ways that we could have chosen to represent real-valued numbers on the tape. We could have allowed the tape to hold only binary strings and used the 2's compliment method to convert the binary string into a real-valued number. Each real-valued number would then have been represented by 32 or 64 squares on the tape, depending on the precision required. This approach was not used in this paper, as this would introduce binary strings that can not be converted to any real-valued number. Also, there would be the issue of how to specify a range in which all real-valued numbers produced on the tape must be within. In addition we would have to decide how to deal with those that fall outside of the specified range. All these issues will be addressed in future work.

# 8. REFERENCES


[1] P. J. Angeline and J. Pollack. Evolutionary module acquisition. In *Proc. of the 2nd Annual Conference on Evolutionary Programming*, pages 154–163, 1993.

[2] D. Fogel. *System Identification Through Simulated Evolution: A Machine Learning Approach to Modelling*. Ginn Press, 1991.

[3] D. Fogel. *Evolving Artificial Intelligence*. PhD thesis, University of California, San Diego, 1992.

[4] D. Fogel. Applying evolutionary programming to selected travelling salesman problems. *Cybernetics and Systems*, 24:27–36, 1993.

[5] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence Through Simulated Evolution*. John Wiley and Sons, 1966.

[6] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, USA, 1992.

[7] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, USA, 1994.

[8] J. F. Miller and P. Thomson. Cartesian genetic programming. In *Proc. of the 3rd EuroGP*, volume 1802 of *LNCS*, pages 121–132. Springer, 2000.

[9] M. Nicolau and C. Ryan. Linkgauge: Tackiling hard deceptive problems with a new linkage learning genetic algortihm. In *Proc. of GECCO*, pages 488–494. AAAI, 2002.

[10] R. Poli. Parallel Distributed Genetic Programming. Technical Report CSRP-96-15, School of Computer Science, University of Birmingham, B15 2TT, UK, Sept. 1996.

[11] C. Ryan, M. Nicolau, and M. O'Neill. Genetic algorithms uing grammatical evolution. In *Proc. of the 5th EuroGP*, volume 2278 of *LNCS*, pages 278–287. Springer, 2002.

[12] J. A. Walker and J. F. Miller. Evolution and acquisition of modules in cartesian genetic programming. In *Proc. of the 7th EuroGP*, volume 3003 of *LNCS*, pages 187–197. Springer, 2004.

[13] J. A. Walker and J. F. Miller. Investigating the performance of module acquisition in cartesian genetic programming. In *Proc. of GECCO*, volume 2, pages 1649–1656. ACM, 2005.

[14] J. A. Walker and J. F. Miller. Embedded cartesian genetic programming and the lawnmower and hierarchical-if-and-only-if problems. In *Proc. of GECCO*. ACM, 2006.

[15] J. A. Walker and J. F. Miller. Automatic acquisition, evolution and re-use of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 2007.

[16] J. A. Walker and J. F. Miller. Changing the genospace: Solving ga problems with cartesian genetic programming. In *Proc. of EuroGP 2007*. Springer, 2007.

[17] X. Yao and Y. Liu. Fast evolutionary programming. In *Proceedings of the Fifth Annual Conference on Evolutionary Programming (EP'96)*, pages 451–460, San Diego, CA, USA, 1996. MIT Press.

[18] X. Yao, Y. Liu, and G. Liu. Evolutionary programming made faster. *IEEE Trans. on Evolutionary Computation*, 3(2):82–102, 1999.

[19] T. Yu and J. F. Miller. Neutrality and the evolvability of boolean function landscape. In *Proc. of the 4th EuroGP*, volume 2038 of *LNCS*, pages 204–217. Springer, 2001.