Changing the Genospace: Solving GA Problems with Cartesian Genetic Programming

James Alfred Walker and Julian Francis Miller

Intelligent Systems Group, Department of Electronics, University of York,
Heslington, York, YO10 5DD, UK
{jaw500,jfm7}@ohm.york.ac.uk

Abstract. Embedded Cartesian Genetic Programming (ECGP) is an extension of Cartesian Genetic Programming (CGP) capable of acquiring, evolving and re-using partial solutions. In this paper, we apply for the first time CGP and ECGP to the ones-max and order-3 deceptive problems, which are normally associated with Genetic Algorithms. Our approach uses CGP and ECGP to evolve a sequence of commands for a tape-head, which produces an arbitrary length binary string on a piece of tape. Computational effort figures are calculated for CGP and ECGP and our results compare favourably with those of Genetic Algorithms.

1 Introduction

Embedded Cartesian Genetic Programming (ECGP) is an extension of Cartesian Genetic Programming (CGP) incorporating ideas from Module Acquisition [1], which allows the automatic acquisition, evolution and re-use of partial solutions in the form of modules. Previous work [2] 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) problem [3]. CGP and ECGP found solutions to the H-IFF problem more easily than published attempts using a GA. This paper builds on the work from [3] by applying the same technique to two GA benchmarks; the one-max problem and the order-3 deceptive problem.

The one-max problem [4] was originally used to test the generality of hill-climbing search algorithms but is now more commonly used as a GA benchmark [5]. The objective of the problem is to find a binary string of length n, which contains all 1's. The order-3 deceptive problem was proposed by Goldberg [6] and has also been widely adopted as a challenging problem for GAs. The problem analyses similarities in a binary string using 3-bit schemata. The aim of the problem is to find a binary string containing all 1's, therefore consisting only of the 3-bit schema containing all 1's. This schema is associated with the highest fitness. The only other fitness rewards are associated with schemata containing all 0's or a single 1. This leads the GA away from the global optimum and towards the global minimum, and is the reason why the problem is classed as deceptive.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 261–270, 2007.

Some work already exists on evolving GAs using alternate forms of evolutionary computation. Miller and Yu [7] implemented a form of CGP to evolve binary strings to the one-max problem when they were investigating the properties and utility of neutrality. Unlike this approach, in this paper the link between the number of nodes encoded in the genotype and length of the binary string is indirect and uncorrelated, we anticipate that this will allowing better scaling on large problem instances. Ryan et al [8] have developed a technique called GAuGE, which extends Grammatical Evolution (GE) to form a position independent GA. GAuGE has been applied to the one-max problem and an extension of GAuGE called LINKGAuGE, which employs tight linkage between the genes of the GA [9], has been applied to the order-3 deceptive problem. Another GE based approach is the meta-Grammar Genetic Algorithm (mGGA) [10], which allows the construction of small bit-strings that are re-used when forming the solution bit-string.

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 GA 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)

CGP is a form of GP based on acyclic directed graphs, which is only modified by mutation [11]. 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 [11] 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

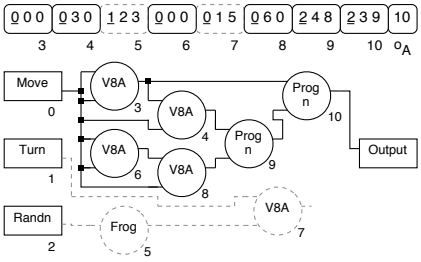

Fig. 1. CGP genotype and corresponding phenotype for the 8-bit one-max problem. The underlined genes encode the function of each node using the lookup table: V8A(0), Frog(1), Progn(2). See Section 4 for function details. The index labels are shown underneath each program input and node. The inactive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

last k nodes encoded in the genotype. Fig. 1 shows a CGP genotype and how it is decoded for the 8-bit one-max problem.

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) [2]. Thereby giving CGP a form of Automatically Defined Function (ADF) [12]. This paper only gives a brief overview of ECGP due to space restrictions. For information on the technical details of ECGP, please refer to [2].

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

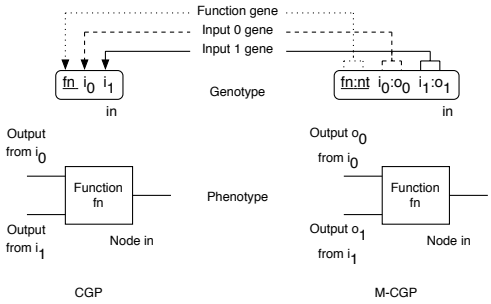

Fig. 2. CGP and ECGP genotypes and corresponding phenotypes for a single node. The components of each gene are labelled as follows: function (fn), node type (nt), node indexes that the node inputs are taken from (i_0, i_1), node outputs that the node inputs are taken from (o_0, o_1), index of this node (in).

modules that contain a re-used section 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 2 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).

4 Applying CGP and ECGP to GAs

One of the main issues faced was deciding how to apply CGP to GA problems. A method was needed which would scale well for different length bit strings and would not require changes to the number or type of program inputs. The method chosen in this paper was heavily influenced by a GP benchmark problem called the Lawnmower Problem [12]. In the lawnmower problem, GP is used to evolve

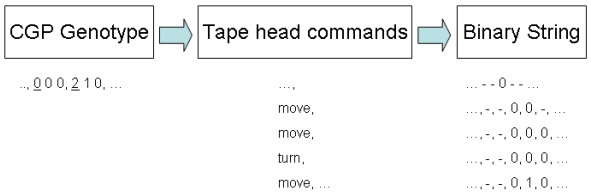

Fig. 3. The three step procedure for producing a GA bit-string from the CGP genotype, via a set of tape head commands

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.

In this paper, instead of evolving a set of instructions for a lawnmower on a 2-dimensional lawn, a set of commands for a moving a tape head on a 1-dimensional tape is evolved. Similar to the lawn in the lawnmower problem, the tape is divided into n squares, where n is 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:

$$\begin{aligned} if(square == blank \mid\mid square == 1),\ square = 0 \\ if(square == 0),\ square = 1 \end{aligned} \tag{1}$$

Therefore, the tape head behaves like the bit-flip operator found in GAs, once a tape square does not contain a blank. 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. An example of the approach is shown in Figure 3.

Although the proposed approach changes the nature of the GA test problems, it does allow us to investigate whether the proposed approach can evolve solutions to GAs whilst taking advantage of the benefits of CGP, such as neutral drift. We believe changing the dimensionality and neutral interconnectedness of the genotype space may alleviate problems typical of GAs - early convergence on sub-optima. Due to the nature of the approach, small changes to the genotype

Table 1. 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 and taken from [2].

Parameter	Value
Population size	5
Genotype point mutation rate	3% (18 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 *	3 or 5 nodes
Number of independent runs	50

can produce a big change in the bit-string produced on the tape. Thereby acting like an implicitly defined variable rate mutation operator, which could reduce the mutation parameter sensitivity associated with GAs.

The CGP program has three program inputs, 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 $0 <= r < n$. Both move and turn 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 1.

5 Experiment Details

The parameter settings used for CGP and ECGP on the 100, 250, 500, 1000, 2000 and 4000-bit one-max problem (using 100 nodes) and the 30-bit order-3 deceptive problem (using 25, 50, 75 and 100 nodes) are shown in Table 1.

The fitness functions used for both problems are the same as those used by GA researchers. For the one-max problem, the fitness function is the total number of ones present in the bit-string and for the order-3 deceptive problem, the fitness function is defined by the schema from [6] shown in Table 2. Any schema containing a blank(-) is awarded a fitness score of zero.

Table 2. The schema for the order-3 deceptive problem and their fitness values

String	000	001	010	011	100	101	110	111
Fitness	28	26	22	0	14	0	0	30

6 Results

Computational effort was calculated using the formula from [3] and shown in Equation 2 with $z = 99\%$. The computational effort figures for CGP and ECGP applied to the one-max and order-3 deceptive problems are shown in Tables 3 and 4 respectively. Various run statistics are also included in both tables to allow comparisons with other techniques and to illustrate how computational effort is a better measure to use than the average number of fitness evaluations, as it is more resilient to outliers in the data. A modified standard deviation is used, as the results are not normally distributed. The modified standard deviation is a statistic in which 68% of all the results lie either side of the mean.

$$P(M,i) = \frac{N_s(i)}{N_{total}},\ R(z) = ceil\left(\frac{\log(1-z)}{\log(1-P(M,i))}\right),\ I(M,i,z) = MR(z)\,i+1 \tag{2}$$

Table 3. Computational effort (CE) figures and various statistics for CGP and ECGP applied to the one-max problem for bit-strings of various lengths (NB). The statistics gathered include: average number of evaluations (AE), modified standard deviation (MSD), the quartiles (Q0-Q4), the limits for mild and extreme outliers (MO and EO) and the number of each outlier present in the data is shown in brackets.

	NB	AE	MSD	Q0	Q1	Q2	Q3	Q4	MO	EO	CE
CGP	100	1,684	1,143	197	583	895	1,906	10,405	3,891(3)	5,875(2)	5,766
	250	2,175	1,598	329	659	981	1,876	13,237	3,702(8)	5,527(5)	6,405
	500	4,850	4,074	321	869	1,471	5,026	47,013	11,262(4)	17,497(3)	9,606
	1000	2,006	1,293	441	818	1,071	1,592	23,405	2,753(6)	3,914(4)	6,120
	2000	2,146	1,165	493	1,145	1,455	2,232	15,989	3,863(5)	5,493(3)	7,203
	4000	3,340	2,312	593	1,193	1,377	2,186	33,417	3,676(8)	5,165(7)	7,203
ECGP(3)	100	5,695	5,210	201	703	1,561	6,645	45,125	15,558(4)	24,471(2)	9,610
	250	8,411	7,750	237	1,169	2,589	11,758	71,745	27,642(4)	43,525(2)	16,326
	500	6,505	5,117	541	1,405	2,443	5,209	68,429	10,915(5)	16,621(4)	16,326
	1000	39,529	37,797	637	1,700	4,665	12,873	1,290,565	29,633(5)	46,392(4)	24,010
	2000	14,186	12,151	793	2,101	3,955	10,539	169,021	23,196(7)	35,853(4)	26,888
	4000	15,125	12,592	445	2,594	5,227	11,012	256,661	23,639(5)	36,266(4)	30,728
ECGP(5)	100	11,472	10,807	301	675	1,593	5,990	220,325	13,963(6)	21,935(4)	10,248
	250	16,839	15,762	353	1,254	2,487	10,378	282,553	24,064(8)	37,750(5)	15,368
	500	14,061	12,787	701	1,425	3,333	7,778	183,753	17,308(7)	26,837(5)	20,810
	1000	22,024	19,676	613	2,064	3,461	7,794	798,149	16,389(5)	24,984(3)	23,527
	2000	19,139	17,327	661	2,329	5,407	14,366	129,725	32,422(8)	50,477(6)	32,652
	4000	19,417	15,612	873	3,866	6,469	15,577	248,457	33,144(5)	50,710(4)	42,248

Table 4. Computational effort (CE) figures and various statistics for CGP and ECGP applied to the 30-bit order-3 deceptive problem for various genotype lengths (ND). The statistics gathered include: average number of evaluations (AE), modified standard deviation (MSD), the quartiles (Q0-Q4), the limits for mild and extreme outliers (MO and EO) and the number of each outlier present in the data is shown in brackets.

	ND	AE	MSD	Q0	Q1	Q2	Q3	Q4	MO	EO	CE
CGP	25	3,814	2,898	221	1,078	2,537	5,445	18,697	12,021(4)	18,586(1)	14,088
	50	5,998	4,874	249	1,139	2,357	5,032	72,717	10,872(4)	16,711(3)	15,368
	75	118,649	117,272	153	1,329	2,535	7,600	3,918,661	17,007(9)	26,413(7)	16,648
	100	279,444	278,372	173	1,171	3,037	10,882	9,066,769	25,449(8)	40,015(8)	15,219
ECGP(3)	25	10,313	9,176	261	1,036	1,859	5,172	303,133	11,376(6)	17,580(4)	12,005
	50	48,374	47,126	401	1,266	2,715	8,049	1,390,337	18,224(9)	28,398(7)	16,648
	75	14,571	13,782	201	912	2,155	5,569	301,137	12,555(6)	19,540(6)	12,808
	100	64,164	62,735	129	1,629	2,691	7,645	1,315,681	16,669(7)	25,693(6)	15,364
ECGP(5)	25	21,548	20,132	385	1,421	2,567	4,755	681,717	9,756(8)	14,757(7)	14,724
	50	70,738	69,797	329	899	2,845	8,486	1,572,621	19,867(8)	31,247(6)	14,415
	75	16,314	15,649	233	696	1,629	13,016	255,513	31,496(7)	49,976(4)	10,413
	100	44,965	43,612	205	1,495	3,651	10,893	1,169,005	24,990(7)	39,087(7)	19,208

For both problems, all fifty independent runs of CGP and ECGP produced 100% successful solutions.

The computational effort figures for the one-max problem show CGP performs better than ECGP regardless of the maximum module size, for all lengths of bit-string. As the length of the bit-string increases, the computational effort required by CGP increases only slightly, indicating that CGP scales particularly well with problem difficulty. This suggests that CGP may perform comparatively better on larger bit-strings. In ECGP, the automatic acquisition, evolution and re-use of modules could be hindering the search performance, possibly due to a lack of modularity in the problem. Alternatively, the problem could be too simple, so by the time a useful module has been discovered, CGP has already found a solution to the problem.

The results in Table 4 show the computational effort figures for CGP and ECGP are similar, as the number of nodes increases but ECGP is capable of performing better than CGP, depending on the maximum module size chosen. This suggests ECGP is exploiting any modularity in the problem that makes it less susceptible to deception, such as the re-use of a module that creates the schema containing three ones. However, the average number of evaluations figures contradict the computational effort figures on a number of occasions. On analysis, the quartiles, Q0-Q3, for CGP and ECGP show a similar trend to the computational effort figures. However, the quartile, Q4, is quite erratic as it contains numerous mild and extreme outliers. The outliers are the reason for the contradiction between the average number of evaluation and computational effort figures, therefore showing computational effort is less influenced by the presence of outliers.

In general, the computational effort figures for CGP and ECGP increase with the number of nodes, suggesting using smaller genotypes produces better results.

Table 5. The average number of evaluations (AE) for other techniques applied to the 100-bit One-Max and 30-bit Order-3 Deceptive Problems

	100-bit One-Max			30-bit Order-3 Deceptive		
Technique	Gen-GA	Simple-GA	GAuGE	Gen-GA	Messy-GA	LinkGAuGE
AE	7,714	4,000	4,000	4,484	10,000	20,000

The larger the genotype, the longer the list of commands for the tape head. Therefore, a small change in a large genotype could drastically alter the number of commands for the tape head, making it harder to find a solution when you are only a few bits away.

The results of CGP and ECGP for the two problems are compared with other techniques found in Table 5. The generational-GA results were taken from [5], the simple-GA and GAuGE results are approximated from [8] and the messy-GA and LinkGAuGE results are approximated from [9].

For the 100-bit One-Max problem, CGP performs better than the other three techniques and ECGP (with a maximum module size of 3) performs better than the generational-GA but worse than the simple-GA and GAuGE. However, it is notable that CGP also solves the 4000-bit One-Max problem slightly faster than the simple-GA and GAuGE on the 100-bit One-Max problem.

Comparing the results of CGP and ECGP (with 25 nodes) and the other techniques on the 30-bit Order-3 Deceptive problem, once again CGP performs better than the other three techniques and ECGP performs better than LinkGAuGE, and has approximately equal performance to the messy-GA but is worse than the generational-GA.

Out of curiosity, the CGP and ECGP solutions found to the One-max problem were applied to the one-max problem with different length of bit-strings than those used to evolve the solution. The results showed that the majority of the solutions found on the original problem solved the One-max problem for all lengths of bit-string from 1-bit up to the length it was originally trained on, and also on some longer bit-strings. In one case, a CGP solution to the 100-bit One-max problem solved all One-max problems up to a length of 264-bits. This implies the solution had learned something about the form of the general solution to the One-Max problem. This was also noticed with the solutions to the order-3 deceptive problem, except that the original solution either solved all the order-3 deceptive problems up to a length of 30-bits, or it solved the order-3 deceptive problems that were a factor of the 30-bit problem, such as the 3, 6 and 15-bit problems. We intend to investigate this further in future work.

7 Conclusion

We have presented the application of CGP and ECGP to two classic GA problems: the one-max and order-3 deceptive problems. CGP was shown to perform better than ECGP on the one-max problem for various length bit-strings and was

also shown to scale well with problem difficulty. The performance of CGP and ECGP was similar on the order-3 deceptive problem, however ECGP is capable of performing better than CGP but is dependant on the relationship between the maximum module size chosen and genotype length. Comparing CGP, a simple GA, a generational GA and GAuGE on the one-max problem showed CGP to perform the best and to scale better on problem size than the others. Comparing CGP, a generational GA, a messy-GA and LINKGAuGE on the order-3 deceptive problem also showed CGP to perform the best. This could possibly indicate that the method employed in this paper not only drastically alters the search space but also takes advantage of the benefits associated with CGP (such as neutral drift) and transfers them to the GA.

Preliminary results for initialising the tape with different values (all 0's or 0's and 1's at random) have shown a decrease in the performance of CGP and ECGP, and will be investigated further in future work. It would be interesting to see if the approach described in this paper can be modified to produce floating point numbers and be applied to real-valued optimisation problems associated with classical evolutionary programming. This approach could also be used in a variety of real-world problems, such as protein folding and protein sequence comparison from the field of bioinformatics.

References

1. Angeline, P.J., Pollack, J.: Evolutionary module acquisition. In: Proc. of the 2nd Annual Conference on Evolutionary Programming. (1993) 154–163
2. Walker, J.A., Miller, J.F.: Investigating the performance of module acquisition in cartesian genetic programming. In: Proc. of GECCO. Volume 2., ACM (2005) 1649–1656
3. Walker, J.A., Miller, J.F.: Embedded cartesian genetic programming and the lawnmower and hierarchical-if-and-only-if problems. In: Proc. of GECCO, ACM (2006)
4. Ackley, D.H.: A connectionist Machine for Genetic Hillclimbing. Kluwer (1987)
5. Tuson, A., Ross, P.: Adapting operator settings in genetic algorithms. Evolutionary Computation **6**(2) (1998)
6. Goldberg, D.E., Deb, K., Korb, B.: Messy genetic algortihms: Motivation, analysis and first results. Complex Systems **3**(5) (1989)
7. Yu, T., Miller, J.F.: The role of neutral and adaptive mutation in an evolutionary search on the onemax problem. In: Late Breaking Papers at GECCO, AAAI (2002) 512–519
8. Ryan, C., Nicolau, M., O'Neill, M.: Genetic algorithms uing grammatical evolution. In: Proc. of the 5th EuroGP. Volume 2278 of LNCS., Springer (2002) 278–287
9. Nicolau, M., Ryan, C.: Linkgauge: Tackiling hard deceptive problems with a new linkage learning genetic algortihm. In: Proc. of GECCO, AAAI (2002) 488–494
10. O'Neill, M., Brabazon, A.: mGGA: The meta-grammar genetic algorithm. In: Proc. of the 8th EuroGP. Volume 3447 of LNCS., Springer (2005) 311–320
11. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proc. of the 3rd EuroGP. Volume 1802 of LNCS., Springer (2000) 121–132
12. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, USA (1994)