Self Modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing

Simon Harding[1], Julian F. Miller[2], and Wolfgang Banzhaf[1]

[1] Department of Computer Science, Memorial University, Canada,
(simonh,banzhaf)@cs.mun.ca,
http://www.cs.mun.ca
[2] Department Of Electronics, University of York, UK,
jfm7@ohm.york.ac.uk,
http://www.elec.york.ac.uk

Abstract. Self Modifying CGP (SMCGP) is a developmental form of Cartesian Genetic Programming(CGP). It is able to modify its own phenotype during execution of the evolved program. This is done by the inclusion of modification operators in the function set. Here we present the use of the technique on several different sequence generation and regression problems.

Key words: Genetic programming, developmental systems

1 Introduction

In biology, the process whereby genotypes gives rise to a phenotype can be regarded as a form of self-modification. At each decoding stage, the expressed genes, the local environment and the cellular machinery influence the subsequent genetic expression [1, 2]. The concept of self-modification can be a unifying way of looking at development that allows the inclusion of genetic regulatory processes inside single cells, graph re-writing and multi-cellular systems.

In evolutionary computation self-modification has not been widely considered, but Spector and Stoffel examined its potential in their ontogenetic programming paper [3]. It also has been studied in the the graph re-writing system of Gruau [4] and was implicitly considered in Miller [5]. However, recently, much work in computational development has concentrated at a multi-cellular level and the aim has been to show that evolution could produce developmental cellular programs that could construct various cellular pattern [6]. A common motivation for evolving developmental programs is that they may allow the evolution of arbitrarily large systems which are infeasible to evolve using a direct genetic representation. However many of the proposed developmental approaches are not *explicitly* computational in that often one must apply some other mapping process from the developed cellular structure into a computation. Further discussion of our motivation, and how it relates to previous work, can be found in [7].

In our previous work, we showed that by utilizing self-modification operations within an existing computational method (a form of genetic programming, called Cartesian Genetic Programming, CGP) we could obtain a system that (a) could develop over time in interaction with environmental inputs and (b) would at every stage provide a computational function [7]. It could stop its own development, if required, without external input. Another interesting feature of the approach is that, in principle, programs could be evolved which allow the replication of the original code.

Here we demonstrate SMCGP on a number of different tasks. The problems illustrate different aspects and capabilities of SMCGP, and are intended to be representative of the types of problems we will investigate in future.

2 Self modifying CGP

2.1 Cartesian Genetic Programming (CGP)

Cartesian Genetic Programming represents programs as directed graphs [8]. 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 always chose the number of rows to be one, thus giving a one-dimensional topology, as used in this paper. In CGP, the genotype is a fixed-length representation and consists of a list of integers which encode the function and connections of each node in the directed graph.

CGP uses a genotype-phenotype mapping that does not require all of the nodes to be connected to each other, resulting in a bounded variable length phenotype. 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 type of neutrality has been investigated in detail [8–10] and found to be beneficial to the evolutionary process on the problems studied.

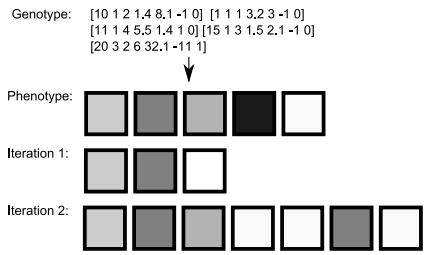

Fig. 1. The genotype maps directly to the initial graph of the phenotype. The genes control the number, type and connectivity of each of the nodes. The phenotype graph is then iterated to perform computation and produce subsequent graphs.

2.2 SMCGP

In this paper, we use a slightly different genotype representation to previously published work using CGP.

As in CGP an integer gene (in our case the first) encodes the function the node represents. This is followed by a number of integer connection genes that indicate the location in the graph where the node takes its inputs from. However in SMCGP there are also three real-valued genes that encode parameters required for the node function. Also there is a binary gene that indicates if the node should be used as an program output. In this paper all nodes take two inputs, hence each node is specified by 7 genes. An example genotype is shown in Figure 1.

Like CGP, nodes take their inputs in a feed-forward manner from either the output of a previous node or from a program input (terminal). The actual number of inputs to a node is dictated by the arity of its function. However, unlike previous implementations of CGP, nodes are addressed *relatively* and specify how many nodes back in the graph they are connected to. Hence, if the connection gene is 1 it means that the node will connect to the previous node in the list, if the gene has value 2 then the node connects 2 nodes back and so on. All such genes are constrained to be greater than 0, to avoid nodes referring directly or indirectly to themselves.

If a gene specifies a connection pointing outside of the graph, i.e. with a larger relative address than there are nodes to connect to, then this is treated as connecting to zero value. Unlike classic CGP, inputs arise in the graph through special functions. This is described in section 2.3. The relative addressing of connection genes allows sub-graphs to be placed or duplicated elsewhere in the graph whilst retaining their semantic validity.

This encoding is demonstrated visually in Figure 2.

The three node function parameter genes are primarily used in performing modification to the phenotype. In the genotype they are represented as real numbers but be cast (truncated) to integers if certain functions require it.

Section 4 details the available functions and any associated parameters.

2.3 Inputs and outputs

The way we handled inputs in our original paper on SMCGP was flawed. We found, it did not scale well as sub-graphs became disconnected from inputs, as self-modifying functions moved them away from the beginning of the graph and they lost their semantic validity. The new input strategy required two simple changes from conventional CGP and our previous work in SMCGP.

The first, was to make all negative addressing return false (or 0 for non-binary versions of SMCGP). In previous work [7], we used negative addresses to connect nodes to input values.

The second was to change how the INPUT function works. When a node is of type INP (shorthand for INPUT), each successive call gets the next input from

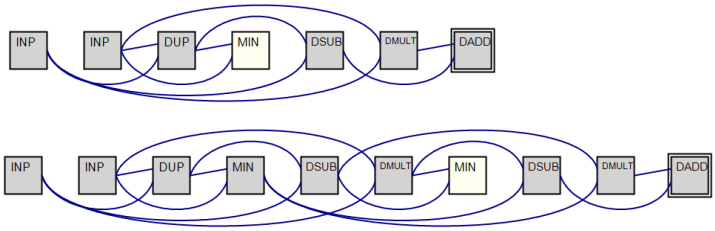

Fig. 2. Example program execution. Showing the DUP(licate) operator being activated, and inserting a copy of a section of the graph (itself and a neighboring functions on either side) elsewhere in the graph in next iteration. Each node is labeled with a function, the relative address of the nodes to connect to and the parameters for the function (see Section 2.4).

the available set of inputs. If the INP node is called more times than there are inputs, the counting starts from the beginning again, and the first node is used.

Outputs are handled slightly differently to inputs. We added another gene to the SMCGP node that decides whether the phenotype could use that node as an output. In previous work, we used the last n-nodes in the graph to represent the n-outputs. However, as with the inputs, we felt this approach did not scale as the graph changes size. When an individual is evaluated, the first stage is to identify the nodes in the graph that have their output gene set to 1. Once these are found, the graph can be evaluated from each of these nodes in a recursive manner.

If no nodes are flagged as outputs, the last n nodes in the graph are used as the n-outputs. Essentially, this reverts the system back to the previous approach. If there are more nodes flagged as outputs than are required, then the leftmost nodes that have flagged outputs are used until the required number of outputs is reached. If there are fewer nodes in the graph than required outputs, the individual is deemed to be corrupt and it is not evaluated (it is given a bad fitness score to ensure that it is not selected for).

2.4 Evaluation of the SMCGP graph

From a high level perspective, when a genotype is evaluated the process is as follows. The initial phenotype graph is a copy of the genotype graph. This graph is then executed, and if there are any modifications to be made, they alter the phenotype graph.

The genotype is invariant during the entire evaluation of the individual. All modifications are made to the phenotype which starts out as a copy of the genotype. In subsequent iterations, the phenotype will usually gradually diverge from the genotype.

The encoded graph is executed in the same manner as standard CGP, but with changes to allow for self-modification. The graph is executed by recursion,

starting from the output nodes down through the functions, to the input nodes. In this way, nodes that are unconnected ('junk') are not processed and do not affect the behavior of the graph at that stage.

For non-self modification function nodes the output value, as in GP in general, is the result of the mathematical operation on input values.

On executing a self-modification node, a comparison is made of the two input values. If the second value is less than the first, the second value is passed through. Otherwise, the node is activated and the first value passed through and the self-modification function in that node is added to a "To Do" list of pending modifications. This makes the execution of the self modifying function dependent on the data passing through the program.

After each iteration, the "To Do" list is parsed, and all manipulations are performed (provided they do not exceed the number of operations specified in the user defined "To Do" list length). The parsing is done in order of the instructions being appended to the list, i.e. first in is first to be executed.

The length of the list can be limited as manipulations are relatively computationally expensive to perform. Here, we limit the length to just 2 instructions. All graph manipulation functions require a number of parameters, as described in section 4.

3 Evolutionary algorithm and parameters

We use an (1+4) evolutionary strategy for the experiments in this paper. We bootstrap the process by testing a population of 50 random individuals. We then select the best individual and generate four offspring. We test these new individuals, and use the best of these to generate the next population. If there is more than one best in the population and one of these is the parent, we always choose the *offspring* to encourage neutral drift (see section 2.1).

We have used a relatively high (for CGP) mutation rate of 0.1. This means that each gene has a probability of 0.1 of being mutated. Mutations for the function type and relative addresses themselves are unbiased; a gene can be mutated to any other valid value.

For the real-valued genes, the mutation operator can choose to randomize the value (with probability 0.1) or add noise (normally distributed, sigma 20).

Evolution is limited to 10,000,000 evaluations. Trials that fail to find a solution in this time are considered to have failed.

The evolutionary parameter values have not been optimized, and we would expect performance increases if more suitable values were used.

4 Function set

The function set is defined in two parts. The first is a set of modification operators, as shown in table 1. These are common to all data types used in SMCGP. The functions chosen are intended to give coverage to as many modification

operations as possible. The remainder of the set are the computational operations. The data type these functions manipulate is determined by the problem definition. Table 2 contains definitions for all the numerical (i.e. non-modifying) operators that are available. Depending on the experiment, different sub-sets of this set are used.

The way self modifying functions act is defined by 4 variables. The three genes that are double precision numbers, here we call them "parameters" and denote them P_0,P_1,P_2. The other variable is the integer position of the node in the phenotype graph that contained the self modifying function (i.e. the leftmost node is position 0), we denote this x. In the definitions of the SM functions we often need to refer to the values taken by node connection genes (which are all relative addresses). We denote the jth connection gene on node at position i, by c_{ij}.
There are several rules that decide how addresses and parameters are treated:

- When P_i are added to the x, the result is treated as an integer.
- Address indexes are corrected if they are not within bounds. Addresses below 0 are treated as 0. Addresses that reach beyond the end of the graph are truncated to the graph length.
- Start and end indexes are sorted into ascending order (if appropriate).
- Operations that are redundant (e.g. copying 0 nodes) are ignored, however they are taken into account in the ToDo list length.

5 Experiments

5.1 Mathematical functions and Sequences

We can use SMCGP to produce numeric sequences, where the program provides the first number on the first iteration, the 2nd on the next and continues to generate the next number in a sequence as we iterate the program. For these sequences, the input value to the program is fixed to 1. This forces the program to modify itself to produce a new program that produces the next digit. We demonstrate this ability on two sequences of integers; Fibonacci and a list of square numbers.

Squares In this task, a program is evolved that generates a sequence of squares 0,1,2,4,9,16,25,... without using multiplication or division operations. As Spector (who first devised this problem) points out, this task can only be successfully performed if the program can modify itself - as it needs to add new functionality in the form of additions to produce the next integer in the sequence [3]. Hence, conventional genetic programming, including CGP, will be unable to find a general solution to this problem.

The function set for this experiment includes all the self modification functions and DADD, CONST, INP, MIN, MAX, INDX, SORT and INCOUNT.

Function name	Description
Duplicate and scale addresses (DU4)	Starting from position $(P_0 + x)$ copy (P_1) nodes and insert after the node at position $(P_0 + x + P_1)$. During the copy, c_{ij} of copied nodes are multiplied by P_2.
Shift Connections (SHIFTCONNECTION)	Starting at node index $(P_0 + x)$, add P_2 to the values of the c_{ij} of next P_1 nodes.
Shift By Mult Connections (MULTCONNECTION)	Starting at node index (P_0 + x), multiply the c_{ij} of the next P_1 nodes by P_2 nodes.
Move (MOV)	Move the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$.
Duplication (DUP)	Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$.
Duplicate, Preserving Connections (DU3)	Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$. When copying, this function modifies the c_{ij} of the copied nodes so that they continue to point to the original nodes.
Delete (DEL)	Delete the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$.
Add (ADD)	Add P_1 new random nodes after $(P_0 + x)$.
Change Function (CHF)	Change the function of node P_0 to the function associated with P_1.
Change Connection (CHC)	Change the $(P_1 mod 3)$th connection of node P_0 to P_2.
Change Parameter (CHP)	Change the $(P_1 mod 3)$th parameter of node P_0 to P_2.
Overwrite (OVR)	Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ to $(P_0 + x + P_2)$, replacing existing nodes in the target position.
Copy To Stop (COPYTOSTOP)	Copy from x to the next "COPYTOSTOP" function node, "STOP" node or the end of the graph. Nodes are inserted at the position the operator stops at.

Table 1. Self modification functions.

Function name	Description
No operation (NOP)	Passes through the first input.
Add, Subtract, Multiply, Divide (DADD, DSUB, DMULT, DDIV)	Performs the relevant mathematical operation on the two inputs.
Const (CONST)	Returns a numeric constant as defined in parameter P_0.
$\sqrt{x}, \frac{1}{\sqrt{x}}$ Cos, Sin, TanH, Absolute (SQRT, DRCP, COS, SIN, TANH, DABS)	Performs the relevant operation on the first input (ignoring the second input).
Average (AVG)	Returns the average of the two inputs.
Node index (INDX)	Returns the index of the current node. 0 being the first node.
Input count (INCOUNT)	Returns the number of program inputs.
Min, Max (MIN, MAX)	Returns the minimum/maximum of the two inputs.

Table 2. Numeric and other functions.

Table 3 shows a summary of results for the squares problem (based on 110 runs). Programs were evolved to produce the first 10 terms in the sequence, with the fitness score being the number of correctly produced numbers. After successfully evolving a program that generates this sequence, the programs were tested on their ability to generalize to produce the first 100 numbers in the sequence. It was found that 84.3% of solutions generalised.

We found that as the squaring program iterates, the length of the phenotype graph increased linearly. However, the number of active nodes inside the graph fluctuated a lot on early iterations but stabilized after about 15 iterations.

Avg. Evaluations	Std. dev.	Min. evaluations	Max. evaluation	% generalize
141,846	513,008	392	3,358,477	84.3

Table 3. Evaluations required to evolve a program that can generate the squares sequence

Fibonacci Koza demonstrated that recursive tree structures could be evolved that generate the Fibonacci sequence [11]. Huelsbergen evolved machine language programs in approximately 1 million evaluations, and found that all his evolved programs were able to generalise [12]. Nishiguchi [13] successfully evolved recursive solutions with a success rate of 70%. The algorithm quickly evolves solutions in approximately 200,000 evaluations. However, the authors do not appear to test for generalisation. More recently, Agapitos and Lucas used a form of object oriented GP to solve this problem [14]. They tested their programs on the first 12 numbers in the sequence, and tested for generality. Generalization was achieved with 25% success and on average required 20 million evaluations. In addition, they note that their approach is computationally expensive. In [15] a graph based GP (similar to CGP) was demonstrated on this problem, however the approach achieved a success rate of only 8% on the training portion (first 12 integers) of the sequence. Wilson and Heywood evolved recursive programs using Linear GP that solved up to order-3 Fibonacci sequences [16]. On average solutions were discovered in 2.12×10^5 evaluations, with a generalization rate of 83%. We evolve for both the first 12 and 50 numbers in the sequence and test for generality to 74 numbers (after which the value exceeds a long int).

Table 4 shows the success rate and number of evaluations required to evolve programs that produce the Fibonacci sequence (based on 287 runs). As with the squares problem, the fitness function is the number of correctly outputted numbers. The starting condition (either 0,1 or 1,2) and the length of the target sequence appear to have little influence on the success rate of time taken. The results show that sequences that are evolved to produce the first 50 numbers do better at generalizing to produce the first 74 numbers. However, the starting condition again makes no difference.

Sum of numbers Here we wished to evolve a program that could sum an arbitrarily long list of numbers. At the n-th iteration, the evolved program should

Start	Max. Iterations	% Success	Avg Evals	% Generalise
0 1	12	89.1	1,019,981	88.6
0 1	50	87.4	774,808	94.5
1 2	12	86.9	965,005	90.4
1 2	50	90.8	983,972	94.4

Table 4. Success and evaluations required to evolve programs that generate the Fibonacci sequence. The starting condition and the length appear to have little influence on the the success rate of time taken. Percentage of solutions that generalize to solve up to 74 terms.

be able to take n inputs and compute the sum of all the inputs. We devised this problem because we thought it would be difficult for genetic programming, but relatively easy for a technique such as neural networks. The premise being, that neural networks appear to perform well when combining input values and genetic programming seems to prefer feature selection on the inputs.

Input vectors consist of random sequences of integers. The fitness is defined as the absolute cumulative error between the output of the program and the expected sum of the values. We evolved programs which were evaluated on input sequences of 2 to 10 numbers. The function sets consists of the self modifying functions and just the ADD operator.

Table 5 summarizes the results this problem. All experiments were found to be successful, in that they evolved programs that could sum between 2 and 10 numbers (depending on the number of iterations the program is iterated). Table 6 shows the number of evaluations required to reach the nth sum (where n is from 2 to 10).

After evolution, the best individual for each run was tested to see how well it generalized. This test involved summing a sequence of 100 numbers. It was found that most solutions generalized, however in 0.07% of cases, they did not.

We also tested the ability of conventional CGP to sum a set of numbers. Here CGP could only be evolved for a given size of set of input numbers. The results (based on 500 runs) are also shown in table 6. It is revealed that CGP is able to solve this problem only for a smaller sets of numbers. This shows a clear benefit of the self-modification approach in comparison with the direct encoding.

Average	Minimum	Maximum	Std. Deviation
6,922	2,27	2,9603	4,998

Table 5. Evaluations required to evolve a program that can add a set of numbers.

5.2 Regression and Classification

Bioinformatics Classification In this experiment, SMCGP is applied to the protein classification problem described in [17]. The task is to predict the location

Size of set	Average	Minimum	Maximum	Std. dev	% CGP
2	50	50	50	0	100
3	618	54	3,248	566	80
4	1,266	64	9,334	1,185	95.8
5	1,957	116	9,935	1,699	48
6	2,564	120	11,252	2,151	38.1
7	3,399	130	17,798	2,827	0
8	4,177	184	17,908	3,208	0
9	5,138	190	18,276	3,871	0
10	5,918	201	22,204	4,401	0

Table 6. Evaluations required to evolve a SMCGP program that can add a set of numbers of a given size. 100% of SMCGP experiments were successful. The % success rate for conventional CGP is also shown.

of a protein in a cell, from the amino acids in the particular protein. The entire dataset was used for the training set. The set consisted of 2,427 entries, with 19 variables each and 1 output. The function set for SMCGP includes all the mathematical operators in addition to the self modifying command set. The CGP function set contained just the mathematical operators (see section 4). For this type of problem, it is not clear that a self-modification approach would have advantages compared with classic CGP. Also, we added the number of iterations to the genotype so that the phenotype is iterated that many times before being executed on the training set.

Table 7 shows the summary of results for this problem (based on 100 runs of each representation). Both CGP and SMCGP perform similarly. The addition of the self modification operations does not appear to hinder evolvability - despite the increase in search space size.

-	CGP	SMCGP
Average fitness (training)	66.81	66.83
Std. dev. fitness (training)	6.35	6.45
Average fitness (validation)	66.10	66.18
Std. dev. fitness (validation)	5.96	6.46
Avg. evaluations to best fitness (training)	7,679	7,071
Std. dev. evaluations to best fitness (training)	2,452	2,644
Avg. evaluations to best fitness (validation)	7,357	7,161
Std. dev. evaluations to best fitness (validation)	2,386	2,597

Table 7. Results summary for the bio informatics classification problem.

Powers Regression A problem was devised that tests the ability of SMCGP to learn a 'modular' regression problem. The task is to evolve a program that, depending on the iteration, approximates the expression x^n where n is the iteration number. The fitness function applies x as integers from 0 to 20. The fitness is defined as the number of wrong outputs (i.e. lower is better).

The function set contains all the modification operators (section 4) and NOP, DADD, DSUB, NOP, DADD, DSUB, DMULT, DDIV, CONST, INDX and INCOUNT from the numeric operations (section 4).

Programs are evolved to $n = 10$ and then tested for generality up to $n = 20$. As with other experiments, the program is evolved incrementally. Where it first tries to solves n=1 and if successful, is evaluated and evolved for n=1 and n=2 until eventually it is evaluated on n=1 to 10.

Table 8 shows the results summary for the powers regression problem. All 337 runs were successful. In this instance, there is an interesting difference between the two starting conditions. If the fitness function starts with $n = 1$ to 5, it is found that that fewer evaluations are required to reach $n = 10$. However, this leads to reduced generalization. Using a Kolmogorov-Smirnov test, it was found that the difference in the evaluations required is statistically significant (p=0.0).

Number of Initial Test Sets	Average Evaluations	Std Dev.	Percentage Generalize
1	687,156	869,699	60.4
5	527,334	600,800	55.6

Table 8. Summary of results for the powers regression problem

6 Conclusions

We have examined and discussed a developmental form of Cartesian Genetic Programming called Self Modifying CGP and evaluated and compared it with classic CGP on a number of diverse problems. We found that it is more efficient than classic CGP at solving four of the test problems: Fibonacci sequence, sequence of squares, sum of inputs, and a power function. In addition it appears that it was able to obtain general solutions for all these problems, although full confirmation of this will require further analysis of evolved programs. On a fourth problem, classification it performed no worse than CGP despite a larger search space.

Other approaches to solving problems, such as Fibonacci, produce a computer program. Instead, at each iteration we produce a structure that produces the output. This could be considered as unrolling the loops (or recursion) in a program. In a related paper [18], we use this structure building approach to construct digital circuits. In future work, we will investigate SMCGP when used as a continuous program. We believe combining both approaches will be beneficial.

References

1. W. Banzhaf, G. Beslon, S. Christensen, J. A. Foster, F. Kps, V. Lefort, J. F. Miller, M. Radman, and J. J. Ramsden, "From artificial evolution to computational evolution: A research agenda," *Nature Reviews Genetics*, vol. 7, pp. 729–735, 2006.
2. G. Kampis, *Self-modifying Systems in Biology and Cognitive Science*. Pergamon Press, 1991.

3. L. Spector and K. Stoffel, "Ontogenetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Stanford University, CA, USA: MIT Press, 28–31 1996, pp. 394–399.
4. F. Gruau, "Neural network synthesis using cellular encoding and the genetic algorithm." Ph.D. dissertation, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France, 1994.
5. J. F. Miller and P. Thomson, "A developmental method for growing graphs and circuits." in *ICES*, ser. Lecture Notes in Computer Science, A. M. Tyrrell, P. C. Haddow, and J. Torresen, Eds., vol. 2606. Springer, 2003, pp. 93–104.
6. S. Kumar and P. Bentley, *On Growth, Form and Computers*. Academic Press, 2003.
7. S. L. Harding, J. F. Miller, and W. Banzhaf, "Self-modifying cartesian genetic programming," in *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, D. Thierens and H.-G. Beyer et al, Eds., vol. 1. London: ACM Press, 7-11 Jul. 2007, pp. 1021–1028.
8. J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Proc. of EuroGP 2000*, ser. LNCS, R. Poli and W. Banzhaf, et al.,, Eds., vol. 1802. Springer-Verlag, 2000, pp. 121–132.
9. V. K. Vassilev and J. F. Miller, "The advantages of landscape neutrality in digital circuit evolution," in *Proc. of ICES*. Springer-Verlag, 2000, vol. 1801, pp. 252–263.
10. T. Yu and J. Miller, "Neutrality and the evolvability of boolean function landscape," in *Proc. of EuroGP 2001*, ser. LNCS, J. F. Miller and M. T. et al., Eds., vol. 2038. Springer-Verlag, 2001, pp. 204–217.
11. J. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*. Cambridge, Massachusetts, USA: MIT Press, 1992.
12. L. Huelsbergen, "Learning recursive sequences via evolution of machine-language programs," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza and K. Deb et al., Eds. Stanford University, CA, USA: Morgan Kaufmann, 13-16 Jul. 1997, pp. 186–194.
13. M. Nishiguchi and Y. Fujimoto, "Evolution of recursive programs with multi-niche genetic programming (mnGP)," in *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence*, 1998, pp. 247–252.
14. A. Agapitos and S. M. Lucas, "Learning recursive functions with object oriented genetic programming," in *EuroGP*, ser. Lecture Notes in Computer Science, P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, Eds., vol. 3905. Springer, 2006, pp. 166–177.
15. S. Shirakawa, S. Ogino, and T. Nagao, "Graph structured program evolution," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. London, England: ACM, 2007, pp. 1686–1693.
16. G. Wilson and M. Heywood, "Learning recursive programs with cooperative coevolution of genetic code mapping and genotype," in *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2007, pp. 1053–1061.
17. W. B. Langdon and W. Banzhaf, "Repeated sequences in linear genetic programming genomes," *Complex Systems*, vol. 15, no. 4, pp. 285–306, 2005.
18. S. Harding, J. F. Miller, and W. Banzhaf, "Self modifying cartesian genetic programming: Parity," in *Submitted to CEC 2009*, 2009.