ME-CGP: Multi Expression Cartesian Genetic Programming

Phil T. Cattani and Colin G. Johnson

Abstract— **Cartesian Genetic Programming (CGP) is a form of Genetic Programming that uses directed graphs to represent programs. In this paper we propose a way of structuring a CGP algorithm to make use of the multiple phenotypes which are implicitly encoded in a genome string. We show that this leads to a large increase in efficiency compared with standard CGP where genomes are translated into only one phenotype. We call this method Multi Expression CGP (ME-CGP), based on Mihai Oltean's work on Multi Expression Programming using linear GP.**

I. Introduction

Cartesian Genetic Programming (CGP) [13] is a form of Genetic Programming (GP) that uses directed graphs to represent programs. In a typical CGP program, the last node in the graph returns the output value for the program. A fixed-length genome of a certain length may not be appropriate for the problem at hand. If the genome is too long, this can lead to inefficiencies in the time it takes to evolve a solution. If the genome is too short, it may not be able to solve the problem at all. In this paper we propose a way of structuring a CGP algorithm to implicitly use a genome string as if it contained multiple lengths at once. We call this method Multi Expression CGP (ME-CGP), as it is based on the Multi Expression Programming concept of Oltean [15], [16]. We implement an ME-CGP program and test it on two types of problems: a set of symbolic regression problems and a 6-bit Multiplexer. We show that using ME-CGP leads to a large increase in efficiency compared with using CGP with fixed single-length genomes.

II. Related Work

Researchers in this field have tried to address the problem of using the most efficient length genome string for a given problem in various ways.

In work on genetic algorithms, the use of variable-length genomes is not uncommon (see e.g. [7], [12] for some early examples). In tree-based GP [11], the evolved structures (program parse trees) are usually of variable size—both because the initially generated trees are of variable size, and because mutation and crossover operators can change the size of a parse tree.

However, for some other GP representations, in particular CGP, the genome is typically of fixed length. This has the potential to lead to the problems mentioned in the introduction concerning genomes that are too short or too long. A number of approaches have been taken to address this issue.

Phil T. Cattani and Colin G. Johnson are with the School of Computing, University of Kent, Canterbury, England (email: C.G.Johnson@kent.ac.uk)

One approach is to evolve the choice of exit node(s) from the CGP system [5]. That is, additional information is included within the genome to specify whether a particular node is to be used as an output node or not.

A more sophisticated approach to this issue is taken by the algorithm called *Self-Modifying Cartesian Genetic Programming* (SMCGP) [4], [5], [3]. In this method, the genome does not specify a fixed program, but rather specifies a phenotype that is able to modify itself using developmental rules, based on the inputs to the program. This has been tested on a range of problems and has proven particularly useful in evolving programs that generalise to a wider variety of input values than are used in training. For example [5], evolving a program to carry out the *square* function using just addition and subtraction operations.

In the work discussed above, the choice of which node to use as an exit node is *evolved*. That is, each time a population member is evaluated, only one exit node (or, for problems requiring multiple values, one node for each value required) is considered.

Similarly, other approaches have attempted to improve the choice of exit node via an operator, for example in *hoist mutation* [9], [8], which takes a tree-based genome and uses a mutation operator which chooses a non-exit node in one generation and re-roots the parse tree at that node, therefore treating it as an exit node in the following generation.

Another analogy is with the idea of *multiple reading frames* in genetics. This is the phenomenon whereby a single genome string can be read from a number of different starting and end positions, thus producing different proteins from a single DNA strand. This analogy has been used by a number of authors in producing genetic algorithm variants [20], [10], [21]. In this paper we extend these ideas to GP.

Paladugu et al. [19] make use of a similar technique in the evolution of modules in biochemical networks, where the input and output nodes are chosen randomly from a set of candidate nodes within each network in the population.

However, these approaches miss an opportunity to exploit information that has *already been calculated* as part of the program evaluation process. In the remainder of this paper, we explore the idea that, rather than throwing away the values at all other nodes than the exit node, we *compare* these values (a cheap operation) and *choose* the best.

This idea was first introduced by Oltean and Groşan in 2003 [17], who applied these methods to function optimization problems. Further papers applied this to the creation of executable structures: boolean logic programs [14] and electronic circuits [18]. These ideas has subsequently been applied to more complex problems still, for example the evolution of classification rules in data mining [2] and evolving

978-1-4244-8126-2/10/$26.00 ©2010 IEEE

intrusion detection systems for computer security [1]. These applications have used variants on Linear GP; in the work below we extend this to Cartesian GP.

III. A CGP Program with Multiple Exit Nodes

A. A Simple CGP Program

We created a simple Cartesian Genetic Programming program suitable for solving a variety of problems. Our program uses a directed graph with one row and a number of columns (s). This is the standard graph structure used in most CGP experiments [22].

The genome string is a sequence of integer numbers, which can be considered as a sequence of tuples of integers of length n (each tuple is called a gene), each tuple describing one node in the final program. The first $n-1$ values indicate which nodes are the input nodes to that node, and the final value indicates which function is used. The population size used in all experiments was 100.

Every node refers to a number of input nodes within the range of $[0..s-1]$, however, the input nodes must both also have node locations which are less than the current node. If the function requires just two inputs, and the number of inputs into the node is three, then the last input is ignored. For example, a node with graph location 10 must have input nodes with locations between 0 and 9 inclusive. A node is also permitted to have identical input nodes. Each node in the graph contains one function from the problem-dependent function set.

After the genome string is translated to the phenotype, the CGP program is executed using forward propagation. That is to say, firstly the node at position 1 collects values from its list of input nodes. These values are then processed by the function within that node, and the output from that function is stored as the output from the node. This procedure is then carried out for each consecutive node on the graph starting with the first non-input node at position 1, and finishing at position $s-1$.

After a generation is run, b of the fittest genomes are selected for reproduction. The remaining $(PopulationSize - b)$ genomes are all mutations of these b fittest genomes. Typically, this value of b is very small compared with the population size. This strong selection pressure is usual within CGP [13]. The fitness function will depend on the problem being tackled.

Our mutation algorithm mutates genome strings at a constant rate $m.$ per gene. If a gene is chosen for mutation, all four integers within that gene are replaced by random valid integer values (including the original value).

B. Multi Expression Genomes: Introducing ME-CGP

For CGP programs which are trying to solve for one variable only, the last node is typically used as the exit node, and this is where the output value for the program is obtained. For programs which are trying to solve for v variables, either the last v nodes in the graph are used, or there is a binary variable encoded in the genome string for each node which determines whether it will be used as an output node [4]; Harding and Miller show that the latter approach scales better when the graph changes size.

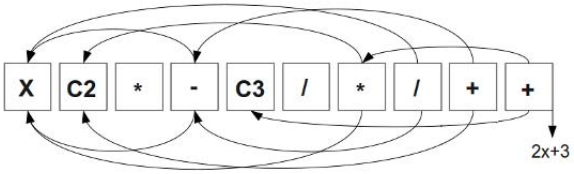

Fig. 1. A decoded genome string with one exit node representing one program

We propose to use multiple exit nodes for solving problems where we are only trying to solve for one variable. That is, we calculate the values at all nodes as normal for each fitness case, and then calculate an aggregate fitness value for each of the e potential exit nodes by summing the fitness values for all fitness cases (e is a parameter). This method is illustrated in figures 1 and 2. We call this approach *Multi Expression CGP* (ME-CGP), and is based on Oltean's work called *Multi Expression Programming* for linear GP [15], [16].

ME-CGP is illustrated in the diagrams in Figures 1 and 2; for clarity, not all node connections in the above diagrams were shown; only those connections which determine the last three output values are shown. Figure 1 represents a traditional CGP program where the value of the last node in the graph is used as the output value of the program. Figure 2 represents a CGP program where multiple exit nodes are used. In Figure 2, the last three nodes of the graph are used, but this could be any arbitrary number, appropriate to the size of the graph. Using exit nodes before the last node on the graph will obviously reduce the potential size of the program, thus implicitly exploring programs of different sizes within a fixed-length genome. However, the law of diminishing returns applies, and at a certain point, the genome string will become too short, so that its expressive power is too limited to solve a problem of a sufficient complexity with its given functions. Therefore, the number of potential exit nodes is a parameter of the ME-CGP process.

We believe this redundancy leads to algorithmic efficiency by implicitly implementing multiple program phenotypes from the same genome string. The question we want to answer is this: does the additional computational cost of doing these calculations outweigh the potential gain in fitness per generation?

C. Evolution Using Multiple Fitness Values

In our experiments we use two complementary processes that involve multiple exit nodes. In the first, the aggregate fitness values are used to determine whether any program has been reached that satisfies the problem-dependent stopping condition. The fitness values are evaluated starting from the

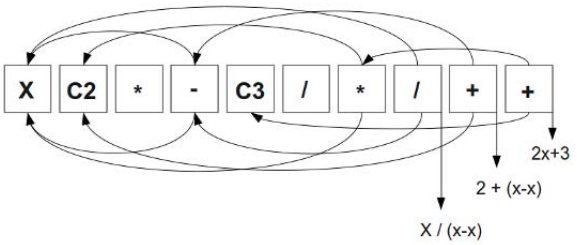

Fig. 2. A decoded genome string with three exit nodes, implicitly representing three programs

lowest exit node location $s - e - 1$ and finishing at node location $s - 1$.

If *any* of the potential exit nodes satisfy the stopping condition, then the program terminates and the current exit node location and phenotype is returned. If this condition is not satisfied, a fitness value is assigned to the genome, consisting of the best fitness value from the set of exit nodes considered.

For example, if we examine the decoded genome string shown in Figure 2, we see that it has 10 nodes. The ME-CGP program is using this string in such a way that it uses the last three nodes as potential exit nodes. The string therefore codes for *three* programs, each of which will have its own fitness value.

In the remainder of this paper we contrast this process (ME-CGP) with a standard single-exit CGP algorithm, which we refer to as the *Baseline CGP*.

IV. Theory

One question to address is whether the cost of processing the multiple exits costs more than processing a population of separate individuals.

We can estimate this cost in the following way. Consider a CGP expression that is made up of f functions. Furthermore, call the average cost evaluating each function c elementary arithmetic or logic operations. In calculating the fitness term, the cost of processing the error for one fitness case is 3 elementary arithmetic operations (EAOs): one to subtract the error from the target value, one to form the absolute value or square of that error, and one to add this to an aggregate error score.

If we consider a standard one exit processing, then the cost per fitness case is $fc + 3$, i.e. the cost of calculating each of the functions in the expression plus 3 for adding the error value to the aggregate error score for that set of fitness cases.

If we consider a multi-expression CGP with e exits, then the cost per fitness case is $(((f - e)c) + e(3 + c))/e$, i.e. the cost of evaluating the first $f - e$ exits and not adding them to the error measure, plus the cost of evaluating the remaining exits *and* adding them to the error measure, divided by the number of different exits that are evaluated.

Consider the parameters used in many of the experiments below, i.e. $s = 30$ and $e = 5$. The graph in figure 3 shows

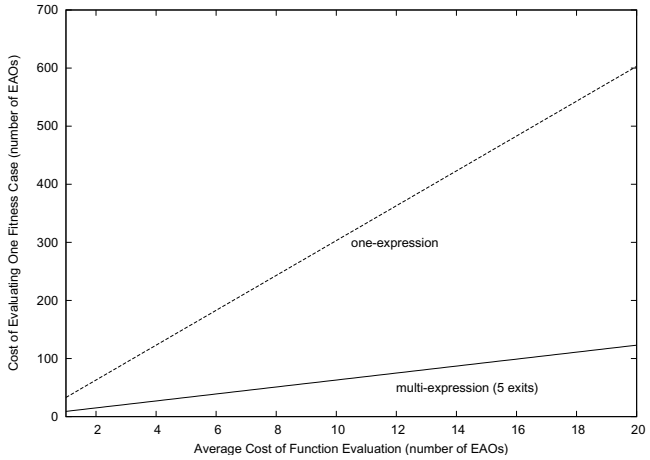

Fig. 3. Comparing a model of the computational cost of one-expression and multi-expression CGP for various function-evaluation costs (c).

the estimated cost of processing one fitness case for various function costs. It is notable that even for the case where the average function cost is 1 (as in the examples below), the multi-expression model is better; for applications where the cost of evaluating a function is very high, the benefit is very large. This latter case could apply, for example, to applications such as image processing.

Furthermore, there is an additional benefit to the multi-expression variant, in that the number of CGP expressions created per genome is higher, and therefore there is potentially a substantial saving of computational effort from having a smaller population size yet having the same number of expressions evaluated.

V. Experimental Methods and Test Problems

In this section we define the parameters used in our experiments and define the various test problems and CGP functions and terminal sets.

A. Parameters and Setup

Using a script, we ran all experiments two hundred times in order to generate statistically meaningful results. We measured the average number of generations as well as the time it took (in milliseconds) to solve each of the symbolic regression problems and the 6-bit Multiplexer problem. Included with each average value is the standard deviation.

We used a population size of 100, and the size of the set of fittest genomes selected in each generation b is 5. We use a mutation rate m of 0.2, and the number of columns s is 30. For the symbolic regression problems the number of inputs to each node is 2 and for the multiplexer problem the number is 3.

Our ME-CGP program uses 5 exit nodes. This value of 5 was chosen as a sensible compromise value between comparing all 30 possible values and having too few. In this study, we did not thoroughly test different values for the number of exit nodes. However, we did a non-exhaustive test of changing the number of exit nodes to ten just for the

test problem of Euler's formula. We discuss these results in the next section. Clearly, a detailed study of this parameter would be a useful piece of future work.

B. Symbolic Regression

We applied ME-CGP program to four symbolic regression problems. The first three were the following expressions: (i) $3x + 2$ (ii) $x^2 + 3x + 2$ (iii) $4x^2 + 3x + 2$. In addition, we included a special case problem of symbolic regression in using a variant on Euler's formula for generating primes [6]: (iv) $x^2 - x + 41$ This formula will generate prime numbers when $0 \leq x < 41$.

The input values for x when testing each evolved program the first two expressions was the integer set: {-100, -50, -10, -5, -4, -3, -2, -1, 0, 1, 2, 3, 5, 10, 50, 100}. While Euler's formula was tested using the first 16 integers starting from zero: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}.

The function set for this problems contains the following functions: {StarterModule, Constant0, Constant1, Constant2, Constant3, Constant4, Constant10, Add, Subtract, Multiply, Divide}. StarterModule is only ever found once in the graph and at position 0. It returns the inputted value for x. The six Constant$X$ functions simply return constants with their respective nominal values. The values obtained from the input nodes for the constant functions are ignored by the function. The divide function is protected so that a divide by zero operation returns 0 instead of an error.

Our fitness function is a function which computes the sum of the differences between each output value, with a given a set of input values, of the phenotype program, and the mathematical expression we are trying to evolve. For example, for the expression $2x + 1$, the correct output value for the input value of 5 would be 11 and for 10 it would be 21. If the program produces the output values 15 and 25 respectively, then the fitness value for that genome string would be $|15-11|+|25-21| = 4+4 = 8$. In this case, lower fitness values are better, and a fitness value of 0 implies the problem has been solved. Note that for each input value, the fitness function takes the absolute difference between the two values.

C. 6-bit Multiplexer

The second type of problem we applied our program to was a 6-bit Multiplexer (6 MUX). The 6-bit Multiplexer has a 2 bit address space and a 4 bit data space. The task of the multiplexer is to return the bit held in the 4 bit data space designated by the value in address space. For example, starting from the left, the address bit 01 would return the bit held in the second data space.

For this problem the function set is the set of boolean operators: {AND, IF, OR, NOT}, where the IF operator has an input arity of three. The IF function is defined by the following behaviour: if the first input is 1, then the value of the second input is passed to the output; if the value of the first input is 0, then the value of the third input is passed to the output.

TABLE I

AVERAGE NUMBER (AND STANDARD DEVIATION) OF GENERATIONS REQUIRED TO SOLVE EACH PROBLEM

Experiment	Baseline CGP	ME-CGP
$3x + 2$	59.54±104.78	8.83±9.54
$x^2 + 3x + 2$	139.04±286.31	23.73±25.44
$4x^2 + 3x + 2$	2795.46±6182.99	121.34±256.25
$x^2 - x + 41$	3892.34±6486.79	582.59±1011.83
6 MUX	6708.53±13133.04	1350.36±2694.77

TABLE II

AVERAGE TIME (AND STANDARD DEVIATION) IN MS REQUIRED TO SOLVE EACH PROBLEM

Experiment	Baseline CGP	ME-CGP
$3x + 2$	851.71±1061.47	241.02±173.67
$x^2 + 3x + 2$	1525.89±2401.29	529.09±374.39
$4x^2 + 3x + 2$	22963.32±49071.94	1539.93±2263.57
$x^2 - x + 41$	32306.19±52424.23	5670.83±8485.80
6 MUX	94478.78±182722.59	20498.11±39113.35

Our input test set was the set of four possible combinations of address space: {00, 01, 10, 11}, with each value tested against the sixteen possible combinations of data space: {0000, 0001,... 1111}. The fitness was calculated by testing each pair from the address space against every possible 4-tuple in the data space, and counting the number of output values where the wrong output was generated by the phenotype program.

VI. RESULTS

Table I, Table II and Table III show our results. Table I indicates the average number of *generations* it took to solve each expression or problem, Table II indicates the average *time* (in milliseconds) it took. Table III compares the average time (in milliseconds) it took to solve the expression x^2-x+41 for five exit nodes and ten exit nodes. With the exception of Table III, all tables in this paper show results obtained using five exit nodes.

Each value is the average (mean) over two hundred trials. Outliers have been removed by taking away the ten trials with the largest generation count for each experiment. This has been done because there were a small number of trials which took an uncharacteristically large number of generations, which distorted the averages. This can, furthermore, be justified in terms of algorithm development, as a practical algorithm could readily include a cap on the maximum number of generations allowed.

The second column in each table represents the Baseline CGP case, whilst the third column represents the ME-CGP case.

VII. DISCUSSION

A. Polynomial Expressions

As we can see in Table I and Table II, using ME-CGP both decreased the number of generations required and the time it took to solve each of our problems.

TABLE III

AVERAGE TIME (AND STANDARD DEVIATION) IN MS REQUIRED TO SOLVE THE EXPRESSION $x^2 - x + 41$ WHEN USING *ten* EXIT NODES.

No. of Exit Nodes	BaselineCGP	ME-CGP
5	32306.19±52424.23	5670.83±8485.80
10	46834.67±87867.58	2808.09±3306.18

TABLE IV

EFFICIENCY GAINS FOR ALL THE EXPERIMENTS WHEN USING *ME-CGP* COMPARED WITH THE RESPECTIVE BASELINES.

Experiment	Generation Eff. Gain	Time Eff. Gain
$3x + 2$	6.74	3.53
$x^2 + 3x + 2$	5.86	2.88
$4x^2 + 3x + 2$	23.04	14.91
$x^2 - x + 41$	6.68	5.70
6 MUX	4.97	4.61

It is clear that, although these techniques may add some computational overhead with each generation, the increase in overall efficiency greatly overrides any cost associated with this overhead.

Table IV shows the efficiency gain (i.e. the ratio of ME-CGP value to Baseline CGP value) for both the generation and time values. This demonstrates the effectiveness of the ME-CGP algorithm.

B. Euler's Prime-generating Formula

Similarly, for the Euler formula, the data in Tables I and II both show a substantial improvement for the ME-CGP over the Baseline CGP. This is summarised in the efficiency gains shown in Table IV.

As seen in Table III, there was a good increase in performance when using ten exit nodes while using ME-CGP, solving the problem in roughly half the time on average compared with ME-CGP with five exits.

C. 6-Bit Multiplexer

Finally, for the 6-bit Multiplexer the data in tables I and II both show a substantial improvement for the ME-CGP over the Baseline CGP. This is also summarised in the efficiency gains shown in Table IV. It is interesting to note that this method also works well for an example in a different domain, i.e. evolving boolean logic functions.

D. Statistical Analysis

The statistical significance of the results has been tested using the Mann-Whitney U test, i.e. testing the hypothesis that the two results are drawn from populations with the same medians. The nonparametric test has been chosen as we cannot readily show that the underlying distributions are normal—indeed, the high values of the standard deviations, alongside informal observations of visualisations of the data, suggest a long-tailed distribution. For each set of experiments, the results for the data sets (with outliers removed) have been analysed to compare the results from the baseline CGP algorithm with those of the ME-CGP algorithm.

TABLE V

p-VALUES FROM MANN-WHITNEY U TESTS

Fitness Function	Measurement	Baseline CGP vs ME-CGP
$3x + 2$	generations	< 0.0001
	time	< 0.0001
$x^2 + 3x + 2$	generations	< 0.0001
	time	< 0.0001
$4x^2 + 3x + 2$	generations	< 0.0001
	time	< 0.0001
$x^2 - x + 41$	generations	< 0.0001
	time	< 0.0001
6 MUX	generations	< 0.0001
	time	< 0.0001

TABLE VI

RATIO OF AVERAGE TO STANDARD DEVIATION (AVG/STDEV) WITH RESPECT TO TIME IN MILLISECONDS, WHEN COMPARING *ME-CGP* TO THE BASELINE.

Expression	Baseline CGP	ME-CGP
$3x + 2$	0.54	1.39
$x^2 + 3x + 2$	0.64	1.41
$4x^2 + 3x + 2$	0.47	0.68
$x^2 - x + 41$	0.61	0.66
6 MUX	0.52	0.52

The results are presented in table V. In all cases the p-value is less than 0.0001. The p-value for the time difference between 10 exits and 5 exits is also < 0.0001; the p-value between the two sets of Baseline CGP runs in the 5 vs. 10 exit experiments (which should be equivalent) is 0.8654.

E. The ratio of the mean to the standard deviation

An additional unexpected benefit which we discovered after compiling our experimental results was that when using ME-CGP, the ratio of the mean value to the standard deviation increased on all the experiments except the 6-bit Multiplexer, where it stayed the same. Thus, using ME-CGP appears to provide more consistent results for symbolic regression problems. The results are shown in Table VI.

VIII. CONCLUSIONS AND FUTURE WORK

We believe the results in this paper represent significant efficiency gains to CGP programs when the techniques of ME-CGP are applied. In a simple CGP program which uses forward propagation, the value of every node is executed starting from the beginning node to the last node. By using the values of the last e nodes as output values we can achieve significant gains in efficiency despite the small overhead cost. Similarly, ME-CGP can be applied to CGP programs which use backwards propagation by only computing the values of a subset of the nodes described by the phenotype. The values of each of the nodes in the subset can be used instead of the last e nodes as in the case with feed forward programs.

We have not yet tested this technique with feed backward networks.

Finally, the more consistent results shown by the increase in ratio of mean to standard deviation in test times is an added not insignificant benefit to using the two techniques outlined in this paper.

REFERENCES

[1] A. Abraham and C. Groşan. Evolving intrusion detection systems. In Nadia Nedjah et al., editor, *Genetic Systems Programming*, pages 57–80. Springer Verlag, 2006. Studies in Fuzziness and Soft Computing.

[2] Adil Baykasoglu and Lale Özbakir. MEPAR-miner : Multi-expression programming for classification rule mining. *European Journal of Operational Research*, 183(2):767–784, 2007.

[3] S. Harding, J.F. Miller, and W. Banzhaf. Self modifying Cartesian genetic programming: Parity. In *Proceedings of the 2009 Congress on Evolutionary Computation (CEC'09)*, pages 285–292. IEEE Press, 2009.

[4] Simon Harding, Julian F. Miller, and Wolfgang Banzhaf. Self-modifying Cartesian genetic programming. In *Proceedings of the 2007 Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1021–1028, 2007.

[5] Simon Harding, Julian F. Miller, and Wolfgang Banzhaf. Self-modifying cartesian genetic programming: Fibonacci, squares, regression, and summing. In *Genetic Programming: Proceedings of the 2009 European Conference*, pages 133–144. Springer, 2009.

[6] G.H. Hardy and E.M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 1979.

[7] I. Harvey. The SAGA cross: The mechanics of crossover for variable-length genetic algorithms. In *Parallel Problem Solving from Nature 2*, pages 269–278. Elsevier, 1992.

[8] K.E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, volume 1, pages 142–147. IEEE Press, 1994.

[9] Kenneth E. Kinnear, Jr. Alternatives in automatic function definition: A comparison of performance. In *Advances in Genetic Programming*, pages 119–141, 1994.

[10] Satoshi Kobayashi and Yasubumi Sakakibara. Multiple splicing systems and the universal computability. *Theor. Comput. Sci.*, 264(1):3–23, 2001.

[11] John R. Koza. *Genetic Programming : On the Programming of Computers by means of Natural Selection*. Series in Complex Adaptive Systems. MIT Press, 1992.

[12] C.-Y. Lee and E.K. Antonsson. Variable length genomes for evolutionary algorithms. In *Proceedings of the 2000 Genetic and Evolutionary Computation Conference (GECCO 2000)*, pages 806–812, 2000.

[13] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian Miller, Peter Nordin, and Terence C. Fogarty, editors, *Proceedings of the 2000 European Conference on Genetic Programming*, pages 121–132. Springer, 2000. LNCS 1802.

[14] Mihai Oltean. Solving even-parity problems using multi expression programming. In Ken Chen et al., editor, *Proceedings of the 5th International Workshop on Frontiers in Evolutionary Algorithms*, pages 315–318, 2003. Part of the 7th Joint Conference on Information Sciences.

[15] Mihai Oltean. Improving the search by encoding multiple solutions in a chromosome. In N. Nedjah and L. de Mourrelo, editors, *Evolutionary Machine Design*, New York, 2004. Nova Science Publishers. Chapter 15.

[16] Mihai Oltean. Multi expression programming. Technical report, Babes-Bolyai Univ, Romania, 2006.

[17] Mihai Oltean and Crina Groşan. Evolving evolutionary algorithms using multi expression programming. In *Proceedings of the Seventh European Conference on Artificial Life*, pages 651–658. Springer-Verlag, 2003. LNAI Vol. 2801.

[18] Mihai Oltean and Crina Groşan. Evolving digital circuits using multi expression programming. In R. Zebulum et al., editor, *NASA/DoD Conference on Evolvable Hardware*. IEEE Press, 2004.

[19] S. Paladugu, V. Chickarmane, A. Deckard, J. Frumkin, M. McCormack, and H.M. Sauro. In-silico evolution of functional modules in biochemical networks. *IEE Proceedings Systems Biology*, 153(4):223–235, 2006.

[20] Michael Schmidt and Hod Lipson. Comparison of tree and graph encodings as function of problem complexity. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1674–1679, New York, NY, USA, 2007. ACM.

[21] T. Soule and A.E. Ball. A genetic algorithm with multiple reading frames. In *Proc. Genetic and Evolutionary Computation Conf.* Citeseer, 2001.

[22] T. Yu and J.F. Miller. Neutrality and the evolvability of boolean function landscape. In *Proceedings of the 4th European Conference on Genetic Programming (EuroGP 2001),*, pages 204–217. Springer-Verlag, 2001. Vol. 2038 on Lecture Notes in Computer Science.