# Self-Modifying Cartesian Genetic Programming

Simon Harding
Dept. of Computer Science
Memorial University
St John's, Canada
simonh@cs.mun.ca

Julian F. Miller
Dept. of Electronics
University of York
York, UK
jfm7@ohm.york.ac.uk

Wolfgang Banzhaf
Dept. of Computer Science
Memorial University
St John's, Canada
banzhaf@cs.mun.ca

## ABSTRACT

In nature, systems with enormous numbers of components (i.e. cells) are evolved from a relatively small genotype. It has not yet been demonstrated that artificial evolution is sufficient to make such a system evolvable. Consequently researchers have been investigating forms of computational development that may allow more evolvable systems. The approaches taken have largely used re-writing, multi- cellularity, or genetic regulation. In many cases it has been difficult to produce general purpose computation from such systems. In this paper we introduce computational development using a form of Cartesian Genetic Programming that includes self-modification operations. One advantage of this approach is that ab initio the system can be used to solve computational problems. We present results on a number of problems and demonstrate the characteristics and advantages that self-modification brings.

## Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: [Miscellaneous]

## General Terms

Algorithms

## 1. INTRODUCTION

In evolutionary computation there has been increasing interest in the notion of the genotype-phenotype mapping. The term mapping is really a synonym for the concept of a mathematical function, i.e. an algorithm that calculates an output from a set of inputs. A genotype-phenotype mapping therefore implies an algorithm that transforms an input string of numbers encoding a genotype into another string of numbers that comprises the phenotype of an individual. However, the inputs in the transformation of natural genotypes into their phenotypes are not merely the string of bases in the DNA. Such a view would be akin to the once dominant, but now obsolete view in molecular biology known as 'the central dogma of biology'. Nowadays, the process of transformation from genotype to phenotype is more properly regarded as a complex interaction in which a genotype, together with the cellular machinery and the environment gives rise to a stage of the phenotype, which itself influences the decoding of the genotype for the next stage [1]. One can regard this process as one of self-modification which could take place both at the genotype or cellular level. Implicit in this notion is the concept of time or iteration. In this paper we take the view that development can be equated to the time-dependent process whereby genotype and phenotype, in interaction with each other and an external environment, produce a phenotype that can be selected for. Kampis [9] has conducted an impressive philosophical analysis of the notion and importance of self-modification in biology and its relevance to 'emergent computation'. In our approach the modifications that can be made (within the scope of the self-modification functions defined) are entirely under the control of evolution. Indeed, it is possible for the genome to destroy itself or create copies of itself.

In section 2 we review work that has investigated the benefits of evolving developmental representations when compared with direct representations. To this day, however, it is still not clear how and whether developmental representations have advantages in a more general computational sense. This is the case because, firstly, investigations have concentrated on particular systems such as neural networks, structural design, digital circuits or sorting networks. Secondly, often by their own admission, authors have chosen rather naive direct representations in comparison with developmental representations. It is our aim in this paper to describe a developmental system that is capable of general computation in the sense of genetic programming. However in doing this we did not want to dictate in advance, whether the representation should be developmental or not. We wanted to leave that to be decided by evolution itself.

The approach we have taken is to introduce into a genetic programming technique, namely Cartesian Genetic Programming (CGP), operators that modify the computational genotype itself. In this way evolution is free to use self-modifying operators or not. In the latter case the developmental system would reduce to non-developmental CGP. In evolutionary computation, the idea of self-modification has its origins in the ontogenetic programming system of Spector and Stoffel [18], the graph re-writing system of Gruau [5] and the developmental method of evolving graphs and circuits of Miller [15].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*GECCO'07,* July 7–11, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-697-4/07/0007 ...$5.00.

## 2. RELATED WORK

Recently there has been increasing interest in the benefits of computational development [12] and its potential benefits for evolutionary computation. Many argue that some form of development will be necessary in order to make evolutionary techniques scale up to larger problems (see, e.g. [2]) and there have been a number of investigations that show that for particular applications developmental or generative techniques scale better.

Early on, Kitano developed a method for evolving the architecture of an artificial neural network using a matrix re-writing system that manipulated adjacency matrices [11]. Kitano found that his method produced superior results to direct methods, i.e., a fixed architecture, directly encoded and evolved. It was later claimed by another study that the two approaches would be really of equal quality [17].

Gruau devised a graph re-writing method called cellular encoding [5] for local graph transformations that control the division of cells growing into a artificial neural network. Connection strengths (weights), threshold value and the grammar tree that defines the graph re-writing rules were evolved using an evolutionary algorithm. This method was shown to be effective at optimizing both the architecture and weights at the same time, and scaled better, according to [6] than a direct encoding.

Bentley and Kumar examined a number of genotype to phenotype mappings on a problem of creating a tessellating tile pattern [3]. They found that the indirect developmental mapping could evolve tiling patterns much quicker than a variety of other representations (including direct) and further, that they could be subsequently grown to (iterated) much larger sized patterns.

Hornby and Pollack evolved context free L-systems to define three dimensional objects (table designs) [7]. They found that their generative system could produce fitter designs faster than direct methods.

Eggenberger investigated the relative merits of a direct versus a developmental genetic representation for the difficult problem of optical lens design [8]. He found that the direct method in which the location of optical elements was evolved scaled very badly when compared with the developmental approach.

Roggen and Federici compared evolving direct and developmental mappings for the task of producing specific two dimensional patterns of various sizes (the Norwegian Flag and a pattern produced by Wolfram 1D CA rule 90) [16]. They showed in both cases that as the pixel dimensions of the patterns increased the developmental methods outperformed the direct.

Gordon showed that evolved developmental representations were more scalable than direct representations for digital adders and parity functions [4].

Sekanina and Bidlo showed how a developmental approach could be evolved to design arbitrarily large sorting networks [13]. Kicinger investigated the problem of design in steel structures for tall buildings and found CA-based generative models produced better results quicker than direct representations and that the solutions were more compact [10].

So a variety of systems have demonstrated the strength of indirect encodings, although none of the systems mentioned performed general purpose computation.

## 3. SELF-MODIFYING CARTESIAN GENETIC PROGRAMMING (SMCGP)

### 3.1 Cartesian Genetic Programming (CGP)

Cartesian Genetic Programming was originally developed by Miller and Thomson [14] for the purpose of evolving digital circuits and represents a program as a directed graph. One of the benefits of this type of representation is the implicit re-use of nodes in the directed graph. 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.

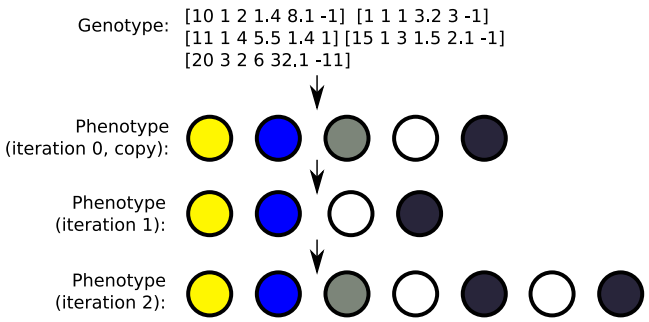


**Figure 1: The genotype maps directly to the initial graph of the phenotype. The genes control the number, type and connectivity of each of the nodes. The phenotype graph is then iterated to perform computation and produce subsequent graphs.**

### 3.2 SMCGP

In this paper, we use a slightly different genotype representation to previously published work using CGP. Each node in the directed graph represents a particular function and is encoded by a number of genes. The first gene encodes the function the node is representing, and the remaining genes encode the location where the node takes its inputs from, plus three parameters required for the function. Hence each node is specified by 6 genes.

An example genotype is shown in Figure 1. The 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 of 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 specifies a distance of 1 it 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 the relative distances are generated 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 an input.

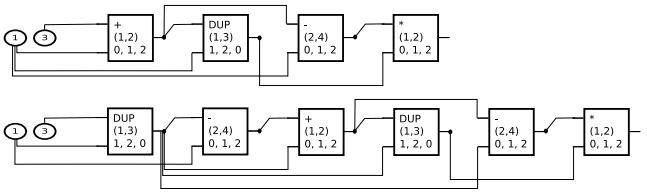


**Figure 2: Example program execution. Showing the DUP(licate) operator being activated, and inserting a copy of a section of the graph (itself and a neighboring functions on either side) into the beginning of the graph in the next iteration. Each node is labeled with a function, the relative address of the nodes to connect to and the parameters for the function (see Section 3.3. The circled numbers of the left are the numerical values taken by the two program inputs.**

The distance is converted to an input index (by taking the modulus of the value and the number of inputs). Hence, the graph automatically can use an arbitrary number of inputs. This encoding is demonstrated visually in Figures 2 and 3

The relative addressing used here attempts to allow for sub-graphs to be placed or duplicated in the graph whilst retaining their semantic validitity. This means that sub-graphs could represent the same sub-function, but acting on different inputs. This can be done without recalculating any node addresses thus maintaining validity of the whole graph. So sub-graphs can be used as functions in the sense of ADFs in standard GP. A node can be replaced, at run time, by a subgraph which takes the two inputs of the calling node (remember that this form of CGP handles arbitrary numbers of inputs without modification to the graph) as inputs. The subgraph is then executed with these inputs and the output of the final node in the subgraph becomes the output of the calling node.

Each node in the SMCGP graph is defined by a function that is represented internally as an integer. Associated with each function are genes denoting connected nodes and also a set of parameters that influence the function's behavior. These parameters are primarily used by functions that perform modification to the phenotype's graph. In some cases they are represented as real numbers but certain functions require that they be cast to integers.Table 8, at the end of this paper, details the available functions and any associated parameters.

## 3.3 Evaluation of the SMCGP graph

From a high level perspective, when a genotype is evaluated the process is as follows. The initial phenotype is a copy of the genotype. This graph is then executed, and if there are any modifications to be made, they alter the phenotype graph. This is illustrated in Figure 1.

Technically, we consider the genotype invariant during the entire evaluation of the individual and perform all modifications on the phenotype which started 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.

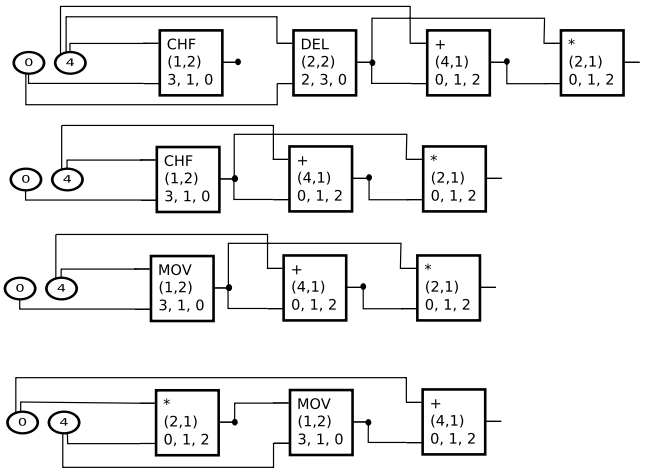


**Figure 3: Example program execution. Showing the DEL(eltion) operator being activated and causing itself to be deleted. On the next iteration, the CHF(change function) node is now connected, and is executed. It modifies it own function to become a MOV(e) operator, which changes the order of the graph. At each iteration, the program is different and outputs a different value - despite the fact the inputs remain constant. Each node is labeled with a function, the relative address of the nodes to connect to and the parameters for the function.**

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 are not processed and do not effect the behavior of the graph at that stage.

For function nodes, such as $+$,$-$ and $*$, the output value is the result of the mathematical operation on input values.

For graph manipulation function, the input values to the node are found and the behavior of that node is based on these input values. If the first input is greater or equal in value to the second, then the graph manipulation function is added to a "To Do" list of pending modifications. After each iteration, the "To Do" list is parsed, and all manipulations are performed. The parsing is done in order of the instructions being appended to the list, i.e. first in first out.

The length of the list can be limited as manipulations are relatively computationally expensive to perform. Here we limit the length to 100 instructions. There is a single "To Do" list for evaluation of each individual, and hence sub-procedures also share the same list. All graph manipulation functions require a number of parameters, as shown in table 9. These parameters are encoded in the genotype, and the necessary casts are made when the "To Do" list is parsed.

For efficiency, nodes are only evaluated once with the result cached, even if they are connected to multiple times. This is necessary to reduce computation especially when multiple outputs are used, as it is quite likely that there will be a large number of shared nodes. It has the side effect that nodes performing graph manipulation will only make one modification - despite repeated calls. Nodes reused by the PRC (procedure) function will be revaluated each time, and hence can repeatedly affect the manipulations.

Two examples of rewriting functions in use are shown in Figures 2 and 3. In the first example the use of DUP(licate) is shown which inserts a copy of a section of the graph (itself and its neighboring function on either side) into the beginning of the graph in the next iteration. Figure 3 shows the effect of the DEL(eltion) operator. On the second iteration, the CHF(change function) node is now connected to the other functions in the graph - as the connections of the graph are relative to the calling node. It modifies it own function to become a MOV(e) operator, which changes the order of the graph. Again, the effect of the relative connections can be seen.

# 4. EVOLUTIONARY ALGORITHM

We used a basic evolutionary system with a population of 50 individuals, elitism and with mutation only. The best 5 individuals in each generation were automatically promoted to the next. Other individuals were produced using selection and mutation. For selection, we used a tournament of size 5, with the best individual being selected. SMCGP, like normal CGP, allows for different mutation rates to effect different parts of the genotype. In these experiments, we chose to make all the rates the same. The parameters used SMCGP which are common to all of the experiments are shown in table 1, however we found that some experiments required slightly different parameters. These changes are described in the experiment details for each experiment. The parameter values have not been optimized, and we would expect performance increases if more suitable values were used.

| Parameter | Value |
|---|---|
| Population size | 50 |
| Initial genotype size | 500 |
| Tournament size | 5 |
| Elitism | 5 |
| Probability mutating a function | 0.001 |
| Probability mutating a connection | 0.001 |
| Probability mutating a parameter | 0.001 |

**Table 1: The common parameters used**

# 5. EXPERIMENTS

## 5.1 Squares

In this task, we ask that evolution find a program that generates a sequences 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 [18] . Hence, normal genetic programming, including CGP, will be unable to find a general solution to this problem. In these experiments, we find that SMCGP is able to evolve this sequence.

The input to the graph is the index number in the sequence which is essentially the number that it needs to square. Initial genotype graphs were limited to 50 nodes, with a maximum limit of 1000 nodes in the phenotype graphs. The function set contains the mathematical operators + and −, in addition the rewriting functions (excluding procedure calls). A maximum of 10,000,000 evaluations were permitted.

| Type | SMCGP |
|---|---|
| Mean | 19.53 |
| Mode | 20 |
| Std dev | 1.58 |
| Successes | 89% |
| Generalised | 66% |
| Mean evaluations to success | 1261422 |
| Std dev of evaluates to successs | 2040392 |

**Table 2: Statistical results for the squares sequence problem, based on 2600 independent runs.**

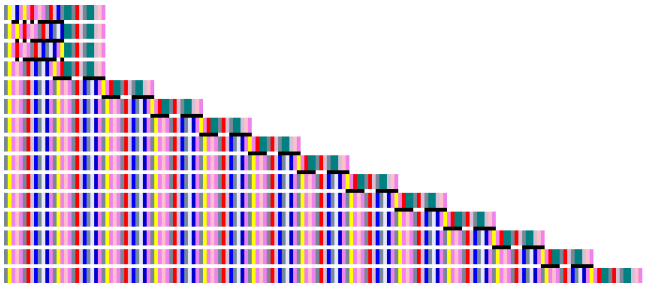

**Figure 4: Graphical view of graph at each stage of iteration. Each node is displayed as a bar, with the colour dependent on function. The black marks between strips indicate the genotype has changed at this point.**

The fitness was determined as the number of correct integers produced before a mistake was made, hence the greater the fitness the better the individual. Individuals were initially evaluated for 10 iterations, leading to a maximum fitness of 10. Individuals that achieved this fitness were then executed for 50 iterations to see if the evolved programs generalised. In many instances this was found to occur.

| Iteration ($i$) | Function | Result |
|---|---|---|
| 0 | $0+i$ | 0 |
| 1 | $0+i$ | 1 |
| 2 | $0+i+i$ | 4 |
| 3 | $0+i+i+i$ | 9 |
| 4 | $0+i+i+i+i$ | 16 |
| etc. | | |

**Table 3: Program that generates sequence of squares**

The results are summarised in table 2. We can see that the successful solutions able to generalise were found in 66% of the runs. Figure 4 shows the graph at each iteration of a successful individual. Each node is displayed as a bar, with the colour dependent on function. The black marks between stripes indicate that the phenotype has changed at this point. The evolved solution in this instance generalises, and will continue to output the next integer in the squares sequence at each iteration.

By examining the generated graph, we found that the developing program appends several new nodes, using the duplicate operator, at each iteration. Each duplicated section contains an addition appended to the end of the output function, as demonstrated in table 3. The duplication of the sections of the graph can be seen clearly in the Figure 4. Notice how the function for iteration 0 and 1 are the same function, and that the graph length does not change in these early iterations. After the 4th iteration, the graph increases at a constant rate. Analysis of several of the evolved programs shows the same behaviour, although implemented in slightly different ways.

Further analysis shows that these programs all fail to operate if the graph is run with modification requests disabled, which is to be expected as it is unclear how the programs could produce an approximation to the square sequence without self-modification.

| Type | CGP | SMCGP |
|---|---|---|
| Mean | 79.5 | 93.5 |
| Mode | 81 | 100 |
| Std dev | 8.24 | 11.95 |
| Successes | 0/100 | 72/100 |

**Table 4: Statistical results for the French Flag sequence problem. Statistically these results differ, with $p \leq 1.236X10^{-16}$**

## 5.2 Producing the French Flag as a sequence

The task in this experiment is to produce a sequence of 100 integers of a desired pattern, here one that superficially resembles a French Flag. Generating the French Flag pattern has become a standard demonstrator in developmental systems, and in cellular developmental systems a French Flag image is often used as the target sequence. Here, we will use sequence of integers as the output, although they could also be interpreted as an image. The sequence consists of a pattern of 20 0s, 20 1s, 20 2s, 20 3s and 20 0s, the second consists of 5 of each type, and repeated 4 times. In the visual interpretation of this, the 0s would represent 'dead cells' and the others particular flag colours.

We use two different approaches to construct the French Flag. In the first, we treat the sequence generation as a form of regression, where the task is to find a function (of an input) that produces the specified sequence. There is no necessity for graph rewriting in this instance, however it is an indicator of general evolvabiltiy. The inputs to the graph are the previous output state of the graph, and the index. Initial genotype graphs were limited to 50 nodes, with a maximum limit of 1000 nodes in the phenotype graphs.

| n | % success | Avg. evals | Std Dev |
|---|---|---|---|
| 2 | 100 | 210 | 259 |
| 3 | 100 | 1740 | 1972 |
| 4 | 100 | 28811 | 45774 |
| 5 | 100 | 58194 | 60052 |
| 6 | 96 | 191493 | 169527 |
| 7 | 84 | 352901 | 270877 |
| 8 | 15 | 583712 | 285074 |

**Table 5: Results for n-input parity**

The function set contains the mathematical operators $+$, $-$, $/$, $\times$, in addition to the rewriting functions (excluding procedure calls). A maximum of 10,000,000 evaluations were permitted.

We found that SMCGP is superior to normal CGP for this problem, despite there being no necessity for the ability to rewrite. This suggest that self-modification can assist evolution in problem solving. Results are based on 100 evaluations of each approach. Higher fitness is better, with 100 being the highest score. Results are summarised in table 4.

Analysis of a sample of the SMCGP programs shows that the rewriting functionality is important to their behaviour, as disabling it causes the programs to fail.

The second French Flag problem we investigated requires the use of self-modification and growth. This task is designed as a demonstrator for our system. The challenge is to find an SMCGP graph that, after execution, contains a French Flag sequence encoded as the output values on active nodes.

The initial graph size is 20 nodes, and we allow the individual to develop. Given an input of 0, we evaluate the graph once. After evaluation we iterate through the nodes, and if they are active, we record the absolute integer value. The values of each of the nodes are then compared to a list of target values, and the number of correct items in the list determines the fitness. We tested the system on various sequence lengths, ranging from 10 nodes to 100 nodes. Where the target length is less than the initial length, the program has to decrease its length.

The function of the evolved program is ignored, only the values of the nodes are of concern.

The results in table 6 show that we are unable to precisely evolve this complicated pattern, however we are able to get good approximations to the solutions. If we consider a solution with 95% correct nodes as a solution, we find that the number of found solutions remains quite high over all target lengths. We expect that changing the parameters used in the evolutionary algorithm could lead to improved results.

## 5.3 Digital Circuit: Parity

In this experiment we investigate the behaviour and performance of SMCGP in building even parity circuit. The output circuit was produced by iterating the graph by the number of times specified in the genotype. This mature graph was then tested with every combination of inputs, with the fitness being the number of correct output bits. The function set includes self-modification operators and the binary functions AND,OR,NAND and NOR.

We compare this approach to CGP and to Embedded-CGP[19](see table 7), and find that SMCGP is consistently better than standard CGP. For larger number of inputs, SMCGP performed worse than ECGP. However, for a small number of inputs SMCGP performed significantly better. The reason can be seen in table 5, where the success rate decreases at 6 inputs which increases the expected number of evaluations required to find a solution.

| Target | Avg. Fitness | P(Success) | Std Dev | Avg Evals |
|---|---|---|---|---|
| Fitness= =1 | | | | |
| 10 | 1 | 1 | 25295 | 18559 |
| 20 | 0.98 | 0.77 | 227685 | 233625 |
| 30 | 0.93 | 0.34 | 278680 | 339898 |
| 40 | 0.89 | 0.12 | 289284 | 369045 |
| 50 | 0.87 | 0.05 | 352544 | 669035 |
| 60 | 0.89 | 0.1 | 334116 | 434746 |
| 70 | 0.87 | 0.09 | 279193 | 418810 |
| 80 | 0.86 | 0.03 | 250057 | 386763 |
| 90 | 0.85 | 0.03 | 89325 | 210250 |
| 100 | 0.85 | 0.06 | 95019 | 235976 |
| Fitness≥0.95 | | | | |
| 10 | 1 | 1 | 25295 | 18559 |
| 20 | 0.98 | 0.91 | 241226 | 258042 |
| 30 | 0.93 | 0.52 | 286800 | 370543 |
| 40 | 0.89 | 0.31 | 300943 | 434336 |
| 50 | 0.87 | 0.2 | 254881 | 616181 |
| 60 | 0.89 | 0.41 | 297623 | 543669 |
| 70 | 0.87 | 0.18 | 239390 | 388756 |
| 80 | 0.86 | 0.18 | 281672 | 516349 |
| 90 | 0.85 | 0.14 | 278381 | 490767 |
| 100 | 0.85 | 0.22 | 306666 | 477681 |

**Table 6: Statistical results for French Flag expression problem.**

| N-inputs | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|
| CGP | 81728 | 293572 | 972420 | 3499532 | 10949256 |
| ECGP | 65296 | 181920 | 287764 | 311940 | 540224 |
| SMCGP | 28811 | 58194 | 191493 | 352901 | 583712 |
| SMCGP(Expected) | 28811 | 58194 | 199256 | 410128 | 1080656 |
| Speedup over ECGP | 2.27 | 3.13 | 1.44 | 0.76 | 0.5 |
| Speedup over CGP | 2.84 | 5.04 | 4.88 | 8.53 | 10.13 |

**Table 7: Results for n-input parity , showing the number of evaluations taken for various size parameters. We also calculate an expected number of evaluations for SMCGP to produce a solution, based on the experimental success rate.**

# 6. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced a form of Cartesian Genetic Programming that has the property of self-modification. The advantage of this approach is that it is a general automated technique for solving computational problems while at the same time being developmental in nature. This allows us to evaluate the advantages of self-modification on a wide range of computational problems and to compare the approach with traditional GP techniques. We presented first experimental results on evolving solutions to a number of problems. We have shown that the use of self-modification can perform tasks that a non-modifying system could not achieve, and that self-modification is even advantageous on problems where self-modification is not necessary.

The results obtained are not always better than a non-modifying encoding for the same problem. However, the encoding presented here can solve the same problems - and others that standard CGP cannot solve - without modification to the algorithm. We feel this feature makes the algorithm robust over many different problem domains, including those that we do not know a-priory if a self-modifying encoding is necessary or gives an advantage.

A similar encoding between the standard and the self-modifying CGP also allows us to compare behavior of the two systems fairly. We hope that further work will give insights into self-modifying and developmental encodings.

In future work we intend to investigate whether the evolutionary algorithm itself could be replaced by survival-based measures for a collection of interacting and self-modifying genomes. Our aim in this is to allow, to some extent, evolution to act on itself and the organization of useful genotypes. We believe that this forms a step in the direction of the computational evolution agenda proposed in [1].

## Acknowledgment

JM and SH acknowledge support from the visitor program of the Dept. of Computer Science at Memorial University. WB acknowledges support from NSERC NSERC under Discovery Grant RGPIN 283304-04.

| Function | Parameters | Description |
|---|---|---|
| NOP | None | Passes the first connection value to the output . |
| + | None | Returns the sum of the input values. |
| − | None | Returns the subtraction of the second input value from the first. |
| * | None | Returns the product of the input values. |
| DIV | None | Returns the first input values, divided by the second. |
| AND | None | Performs a logical AND of the input values |
| OR | None | Performs a logical OR of the input values |
| NAND | None | Performs a logical NAND of the input values |
| NOR | None | Performs a logical ANOR of the input values |
| CONST | Value | Returns the first parameter. |
| INP | InputIndex | Returns the (InputIndex modulo the number of inputs) input value |
| READ | Address | Returns the value stored in memory location (Address modulo memory size) |
| WRT | Address | Stores the first input value in memory location (Address modulo memory size) |
| PRC | Start, End | Executes the subgraph specified by Start and End as a separate graph with the calling nodes input values used as the graph inputs |

**Table 8: SMCGP Function set**

| Function | Parameters | Description |
|---|---|---|
| MOVE | Start, End, Insert | Moves each of the nodes between Start and End into the position specified by Insert |
| DUPE | Start, End, Insert | Inserts copies of the nodes between Start and End into the position specified by Insert |
| DELETE | Start, End | Deletes the nodes between Start and End indexes |
| ADD | Insert, Count | Adds Count number of NOP nodes at position Insert |
| CHF | Node, New Function | Changes the function of a specified node to the specified function |
| CHC | Node, Connection1, Connection2 | Changes the connections in the specified node |
| CHP | Node, ParameterIndex, New Value | Changes the specified parameter and a given node |
| FLR | None | Clears any entries in the pending modifications list |
| OVR | Start, End, Insert | Moves each of the nodes between Start and End into the position specified by Insert, overwriting existing nodes |
| DU2 | Start, End, Insert | Similar to DUPE, but connections are considered to absolute, rather than relative |

**Table 9: SMCGP overwriting functions set., The following nodes are executed if the first input is greater, or equal to the second. They output 1 if the they are executed, 0 otherwise**

# 7. 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*, 7:729–735, 2006.

[2] W. Banzhaf and J. Miller. The challenge of complexity. In A. Menon, editor, *Frontiers in Evolutionary Computation*, pages 243–260. Kluwer Academic, 2004.

[3] P. Bentley and S. Kumar. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 35–43, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[4] T. G. Gordon and P. J. Bentley. Development brings scalability to hardware evolution. In *Proceedings of the 2005 NASA/DoD Conference on Evolvable Hardware*, pages 272–279, 2005.

[5] F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, France, 1994.

[6] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Stanford University, CA, USA, 28–31 1996. MIT Press.

[7] G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 600–607, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 2001. IEEE Press.

[8] P. E. Hotz. Comparing direct and developmental encoding schemes in artificial evolution: A case study in evolving lens shapes. In *Congress on Evolutionary Computation, CEC 2004*, 2004.

[9] G. Kampis. Self-modifying systems in biology and cognitive science, 1991.

[10] R. Kicinger. Evolutionary development system for structural design. In *AAAI Fall Symposium in Developmental Systems*, 2006.

[11] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4(4):461–476, 1990.

[12] S. Kumar and P. J. Bentley. *On Growth, Form and Computers*. Academic Press Inc., US, 2003.

[13] S. Luks and B. Michal. Evolutionary design of arbitrarily large sorting networks using development. *Genetic Programming and Evolvable Machines*, 6(3):319–347, 2005.

[14] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 2000. Springer-Verlag.

[15] J. F. Miller and P. Thomson. A developmental method for growing graphs and circuits. In *Proceedings of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, volume 2606 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2003.

[16] D. Roggen and D. Federici. Multi-cellular development: is there scalability and robustness to gain? In X. Yao, E. Burke, and J. L. et al., editors, *proceedings of Parallel Problem Solving from Nature 8, Parallel Problem Solving from Nature (PPSN) 2004*, pages 391–400, 2004.

[17] A. Siddiqi and S. Lucas. A comparison of matrix rewriting versus direct encoding for evolving neural networks. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation, (Piscataway, NJ, USA)*, pages 392–397. IEEE Press, 1998.

[18] L. Spector and K. Stoffel. Ontogenetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 394–399, Stanford University, CA, USA, 28–31 1996. MIT Press.

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