Chapter 1

A SURVEY OF SELF MODIFYING CARTESIAN GENETIC PROGRAMMING

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

[1]*Department Of Computer Science, Memorial University, Canada;* [2]*Department Of Electronics, University of York, UK;*

Abstract

Self-Modifying Cartesian Genetic Programming (SMCGP) is a general purpose, graph-based, developmental form of Cartesian Genetic Programming. In addition to the usual computational functions found in CGP, SMCGP includes functions that can modify the evolved program at run time. This means that programs can be iterated to produce an infinite sequence of phenotypes from a single evolved genotype. Here, we discuss the results of using SMCGP on a variety of different problems, and see that SMCGP is able to solve tasks that require scalability and plasticity. We see how SMCGP is able to produce results that would be impossible for conventional, static Genetic Programming techniques.

Keywords: Cartesian genetic programming, developmental systems

1.   Introduction

In evolutionary computation (EC) scalability has always been an important issue. An evolutionary technique is scalable if the generational time it takes to evolve a satisfactory solution to a problem increases relatively weakly with increasing problem size. As in EC, scalability is an important issue in Genetic Programming (GP). In GP important methods for improving scalability are modularity and re-use. Modularity is introduced through sub-functions or sub-procedures. These are often called Automatically Defined Functions (ADFs) (Koza, 1994a). The use of ADFs improves the scalability of GP by allowing solutions of larger or more difficult instances of particular classes of problems to be evolved. However, GP methods in general have largely employed genotype representations whose length (number of genes) is proportional to the size of the anticipated problem solutions. This has meant that *evolutionary operators* (e.g. crossover or mutation) have been used as the mechanism for building large genotypes. The same idea underlies approaches to evolve artificial neural networks. For instance, a well known method called NEAT uses evolutionary operators to introduce new neurons and connections, thus expanding the size of the genotype (Stanley and Miikkulainen, 2002).

It is interesting to contrast these approaches to mechanisms employed in evolution of biological organisms. Multicellular organisms, having possibly enormous phenotypes, are developed from *relatively* simple genotypes. Development implies an unfolding in space and time. It is clearly promising to consider employing an analogue of biological development in genetic programming (Banzhaf and Miller, 2004). There are, of course, many possible aspects of developmental biology that could be adopted to construct a developmental GP method. In this chapter we discuss one such approach. It is called Self Modifying Cartesian Genetic Programming (SMCGP). It is based on a simple underlying idea. Namely, that a phenotype can unfold over time from a genotype by allowing the genotype to include primitive functions which act on the genotype itself. We refer to this as self-modification. As far as the authors are aware, self-modification is included in only one existing GP system: Lee Spector's Push GP language (Spector and Robinson, 2002). One of the attractive aspects of introducing primitive self-modification functions is that it is relatively easy to include them in any GP system.

Since 2007, SMCGP has been applied to a variety of computational problems. In the ensuing time the actual details of the SMCGP implementation have changed, however the key concepts and philosophy have remained the same. Here we present the latest version. We explain the

essentials of how SMCGP works in section 2. Section 3 discusses briefly examples of previous work with SMCGP. In section 4 we compare and contrast the way other GP systems include iteration with the iterative unrolling that occurs in SMCGP. We end the chapter with conclusions and suggestions for future work.

2. Self Modifying Cartesian Genetic Programming

As the name suggests, SMCGP is based on the Cartesian Genetic Programming technique. In CGP, programs are encoded in a partly connected, feed forward graph. A full description can be found in (Miller and Thomson, 2000) . The genotype encodes this graph. Associated with each node in the graph are genes that represent the node function and genes representing connections to either other nodes or terminals. The representation has a number of interesting features. Firstly, not all of the nodes in the genotype need to be connected to the output, so there is a degree of neutrality which has been shown to be very useful (Miller and Thomson, 2000; Vassilev and Miller, 2000; Yu and Miller, 2001; Miller and Smith, 2006). Secondly, as the genotype encodes a graph there is reuse of nodes, which makes the representation very compact and also distinct from tree based GP.

Although CGP has been used in various ways in developmental systems (Miller, 2004; Miller and Thomson, 2003; Khan et al., 2007), the programs that it produces are not themselves developmental. Instead, these approaches used a fixed length genotype to represent the programs defining the behaviour of cells.

SMCGP's representation is similar to CGP in some ways, but has extensions that allow it to have the self modifying features. SMCGP genotypes are a linear string of nodes. That is to say, only one row of nodes is used (in contrast to CGP which can have a rectangular grid of nodes). In contrast to CGP in which connection genes are absolute addresses, indicating where the data supplied to a node is to be obtained, SMCGP uses *relative* addressing. Each node obtains its data inputs from its connection genes by counting back from its position in the graph. To prevent cycles, nodes can only connect to previous nodes (on their left). The relative addressing allows section of the graph to be moved, duplicated, deleted etc without breaking constraints of the structure whilst allowing some sort of modularity. In addition to CGP, SMCGP has some extra genes that are used by self-modification functions to identify parts or characteristics of the graph that will be changed.

Another change from CGP is the way SMCGP handles inputs and outputs. Terminals are acquired through special functions (called INP, INPP, SKIPINP) and program outputs are taken from a special function called OUTPUT. This is an important change as it enables SMCGP programs to obtain and deliver as many inputs or outputs as required by the problem domain, during program execution. This allows the possibility of evolving general solutions to problems. For example, to find a program that can compute even-n parity, where n is arbitrary, one needs to be able to acquire an arbitrary number of inputs or terminals.

In summary: Each node in the SMCGP graph contains a number of evolvable elements:

- The function. Represented in the genotype as an integer.

- A list of (relative) connections addresses, again represented as integers.

- A set of 3 floating point number arguments used by self-modification functions.

There are also primitive *functions* that acquire or deliver inputs and outputs.

As with CGP, the number of nodes in the genotype is typically kept constant through an experiment. However, this means care has to be taken to ensure that the genotype is large enough to store the target program.

Executing a SMCGP Individual

SMCGP individuals are evaluated in a multi-step process, with the evolved program (the phenotype) executed several times. The evolved program in SMCGP initially has the same structure as the genotype, hence the first step is to make a copy of the genotype and call it the phenotype. This graph is to be the 'working copy' of the program.

Each time the program is executed, the graph is first run and then any self modification operations required are invoked. The graph is executed in the following manner.

First, the node (or nodes) to be used as outputs are identified. This is done by reading through the graph looking at which nodes are of type OUTPUT. Once a sufficient number of these nodes has been found, the various nodes that they connect to are identified. If not enough output nodes are found, then the last n nodes in the graph are used, where n is the number of outputs required. If there are not enough nodes to satisfy this requirement, then the execution is aborted, and the individual is discarded.

At this point in the decoding, all the nodes that are actually used by the program have been identified and so their values can be calculated (the other nodes can simply be ignored). For the mathematical and binary operators, these functions are performed in the usual manner. However, as mentioned before SMCGP has a number of special functions. Table 1-1 shows an example of some of the functions used in previous work (see section 3).

The first special functions are the INP and INPP functions. Each time the INP function is called it returns the next available input (starting with the first, and returning to the first after reading the last input). The INPP function is similar, but moves backwards through the inputs. SKIPINP allows a number of inputs to be ignored, and then returns the next input. These functions help SMCGP to scale to handle increasing numbers of inputs through development. This also applies to the use of the OUTPUT function, which allows the number of outputs to change over time.

If a function is a self modification function, then it may be activated depending on the following rules. For binary functions they are always activated. For numeric function nodes, if the 1st input is larger than the 2nd input the node is activated. The self modification operation from an activated node is added to a list of pending operations - the 'ToDo' list. The maximum length of the list is a parameter of the system. After execution, the self modification functions on the ToDo list are applied to the current graph. The ToDo list is operated as a FIFO list in which the leftmost activated SM function is the first to be executed (and so on).

The self modification functions require arguments defining which parts of the phenotype the function operates on. These are taken from the arguments of the calling node. Many of the arguments are integers, so they may need to be cast. The arguments may be treated as an address (depending on the function) and like all SMCGP operations, these are relative addresses. The program can now be iterated again, if necessary.

3. Summary of Previous Work in SMCGP

Early experiments

There are very few benchmark problems in the developmental system literature. In the first paper on SMCGP (Harding et al., 2007), we identified two possible challenges that had been described previously.

The first was to find a program that generates a sequence of squares (i.e. 0,1,2,4,9,16,25...) using a restricted set of mathematical operators such as $+$ and $-$, but not multiplication or power. Without some form

Basic	
Delete (DEL)	Delete the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$.
Add (ADD)	Add P_1 new random nodes after $(P_0 + x)$.
Move (MOV)	Move the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$.
Duplication	
Overwrite (OVR)	Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ to position $(P_0 + x + P_2)$, replacing existing nodes in the target position.
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.
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.
Copy To Stop (COPYTOSTOP)	Copy from x to the next "COPYTOSTOP" or 'STOP" function node, or the end of the graph. Nodes are inserted at the position the operator stops at.
Stop Marker (STOP)	Marks the end of a COPYTOSTOP section.
Connection modification	
Shift Connections (SHIFTCONNEC-TION)	Starting at node index $(P_0 + x)$, add P_2 to the values of the c_{ij} of next P_1.
Shift Connections 2 (MULTCONNEC-TION)	Starting at node index $(P_0 + x)$, multiply the c_{ij} of the next P_1 nodes by P_2.
Change Connection (CHC)	Change the $(P_1 mod 3)$th connection of node P_0 to P_2.
Function modification	
Change Function (CHF)	Change the function of node P_0 to the function associated with P_1.
Change Parameter (CHP)	Change the $(P_1 mod 3)$th parameter of node P_0 to P_2.
Miscellaneous	
Flush (FLR)	Clears the contents of the ToDo list

Table 1-1. Self modification functions. x represents the absolute position of the node in the graph, where the leftmost node has position 0. P_N are evolved parameters stored in each node.

of self modification this challenge would be impossible to solve (Spector and Stoffel, 1996). SMCGP was easily able to solve this problem (89% success rate), and a large number of different solutions were found.

Typical solutions were similar to the program in table 1-2, where the program grew in length by adding new terms.

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 1-2.   Program that generates sequence of squares. The program was found by reverse engineering a SMCGP phenotype. i, the current iteration, is the only input to the program.

During evolution, solutions were only tested up to the first 10 iterations. However, after evolution the solutions were tested for generality by increasing the number of iterations to 50. 66% of the solutions are correct to 50 iterations. Thus SMCGP was able to find general solutions.

The next benchmark problem was the French Flag (FF) problem. Several developmental systems have been tested on generating the FF pattern (Miller, 2003; Miller and Banzhaf, 2003; Miller, 2004), and it is one of the few common problems tackled. In this problem, the task is to evolve a program that can assign the states of cells (represented as colours) into three distinct regions so that the complete set of cells looks like a French Flag. However, the design goals of SMCGP are very different to those the FF task demands. Many developmental systems are built around the idea of multi-cellularity and although they are capable of producing cellular patterns or even concentrations of simulated proteins, they are not explicitly computational in the sense of Genetic Programming. Often researchers have to devise somewhat arbitrary mappings from developmental outputs (i.e. cell states and protein levels) to those required for some computational application. SMCGP is designed to be an explicitly computational developmental system from the outset.

Typically, the FF is produced via a type of cellular automaton (CA), where each cell 'alive' contains a copy of an evolved program or set of update rules. We could have taken this approach with SMCGP, but we decided on a more abstract interpretation of the problem. In the CA version, each cell in the CA is analogous to a biological cell. In SMCGP,

the biological abstractions are blurred, and the SMCGP phenotype itself could be viewed as a collection of cells. One way of viewing cells in SMCGP is to break the phenotype into 'modules' and then define these as the cells. In this way, SMCGP cells duplicate and differentiate using the various modifying functions. In a static program, this concept of cellularity does not exist.

To tackle the FF problem with SMCGP, we defined the target pattern to be a string of integers that could be visually interpreted as a French Flag pattern. In the CA model, the pattern would be taken as the output of the program at each cell. Here, since we can view SMCGP phenotypes as a collection of cells, we took the output pattern as the set of outputs from all the active (connected) nodes in the phenotype graph. The fitness of an individual is the count of how many of the sequence it got right after a certain number of iterations.

As the phenotype can change length when it is iterated, the number of active nodes can change and the length of the output pattern can also change. The value of the output of active nodes is dependent on the calculation it (and the nodes before it) does. So the French Flag pattern is effectively the side effect of some mathematical expression.

It was found that this approach was largely successful, but only in generating approximations to the flag. No exact solutions were found, which is similar to the findings of the CA solutions where exact results are uncommon.

The final task we explored in this paper was generating parity circuits, a challenge we return to in the next section.

Digital Circuits

Digital circuits have often been studied in genetic programming (Koza, 1994b; Koza, 1992b), and some systems have been used to produce 'general' solutions (Huelsbergen, 1998; Wong and Leung, 1996; Wong, 2005). A general solution in this sense is a program that can output a digital circuit for an arbitrary number of inputs, for example it may generate a parity circuit of any size [1]. Conveniently, many digital circuits are modular and hierarchical - and this fits the model of development that SMCGP implements.

In our first paper, we successfully produced parity circuits up to 8 inputs (Harding et al., 2007). We stopped at this size because, at the time, this was the maximum size we could find conventional CGP so-

---

[1]An even parity circuit takes a set of binary inputs and outputs true if an even number of the inputs are true, and false otherwise.

lutions for. In a subsequent paper (Harding et al., 2009a), we revisited the problem (using the latest version of SMCGP), and found that not only could we evolve larger parity circuits, but we could rapidly and consistently evolve provably general parity circuits.

We used an incremental fitness function to find programs that on the first iteration would solve 2 input parity, then 3 input parity on the next iteration and continue up to a maximum number of inputs. The fitness of an individual is the number of correct output bits over all iterations. To keep the computational costs down, we limited the evolution to 2 to 20 inputs, and then tested the final programs for generality by running up to 24 bits of input. We also stopped iterating programs if they failed to correctly produce all the output bits for the current table.

Note how if an individual fails to be successful on a particular iteration the evaluation is canceled. Not only did this reduce the computation time, but we hoped it would also help with producing generalized solutions. Our function set consisted of all the two-input Boolean functions and the self modifying functions. In 251 evolutionary runs we found that the average number of evaluations required to successfully solve the parity problems was (number of inputs in parentheses) are as follows: 1,429(2), 4,013 (3), 43,817 (6), 82, 936 (8), 107,586 (10), 110,216 (17). Here we have given an incomplete list that just illustrates the trend in problem difficulty.

We found that the number of evaluations stabilizes when the number of inputs is about 10. This is because after evolution has solved to a given number of inputs the solutions typically become generalized. We found that by the time that evolution had solved 5 inputs, more than half the solutions were generalizable up to 20 inputs, and by 10 inputs this was up to 90%. The percentage of runs that correctly computed even-parity 22 to 24 was approximately 96%. However, without analysis of the programs it was difficult to know whether they were truly general solutions.

The evolved programs can be relatively compact, especially when we place constraints on the initial size, the number of self modification operations allowed on the ToDo list and the overall length of the program. Figure 1-1 shows an example of an evolved parity circuit generating a program which we were able to prove is a general solution to even-parity.

In recent work (to be published in (Harding et al., 2010a)) we have also shown general solutions for the digital adder circuit. A digital adder circuit of size n adds two binary n bit numbers together. This problem is much more complicated than parity, as the number of inputs scales twice as fast (i.e. it has to produce 1 bit+1 bit, 2+2, 3+3) and the number of outputs also grows with the number of inputs.
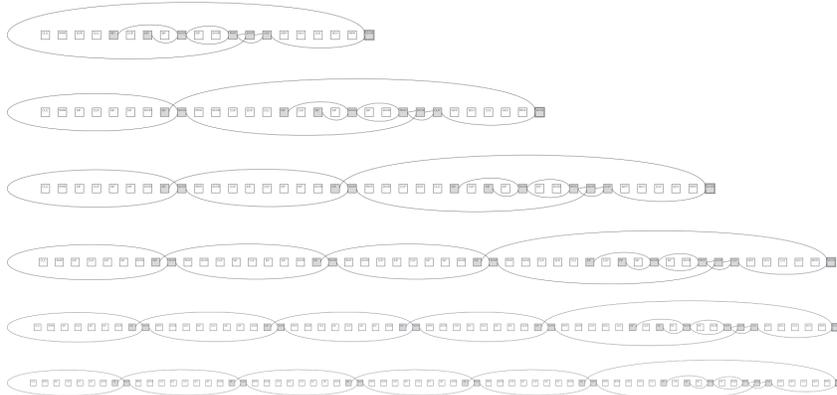
Figure 1-1.   An example of the development of a parity circuit. Each line shows the phenotype graph at a given time step. The first graph solves the 2-input parity, the second solves 3-input and continues to 7-bits. The graph has been tested to generalise through to 24 inputs. This pattern of growth is typical of the programs investigated.

Mathematical problems

SMCGP has been applied to a variety of mathematical problems (Harding et al., 2009c; Harding et al., 2010b).

For the Fibonacci sequence, the fitness function is the number of correctly calculated Fibonacci numbers in a sequence of 50. The first two Fibonacci numbers are given as fixed inputs (these were 0 and 1). Thus the phenotypes are iterated 48 times. Evolved solutions were tested for generality by iterating up to 72 times (after which the numbers exceeds the long int). A success rate of 87.4% was acheived on 287 runs and 94.5% of these correctly calculated the suceeeding 24 Fibonacci numbers. We found that the average number of evaluations of 774,808 compared favourably with previously published methods and that the generalization rate was higher.

In the "list summation problem" we evolved programs that could sum an arbitrarily long list of numbers. At the n-th iteration, the evolved program should 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 without the addition of an explicit summation command. Koza used a summation operator called SIGMA that repeatedly evaluates its sole input until a predefined termination condition is realised (Koza, 1992a).

Input vectors consisted 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 set consisted of the self modifying functions and just the ADD operator. All 500 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). On average it took 6,922 evaluations to solve this problem. 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 99.03% solutions generalized. When conventional CGP was used it could only sum up to 7 numbers.

We also studied how SMCGP performed on a "Powers 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). Programs were evolved to $n = 10$ and then tested for generality up to $n = 20$. As with many of the other experiments, the program is evolved with an incremental fitness function. We obtained 100% correct solutions (in 337 runs). The average number of evalutions was 869,699.

More recently we have looked at whether SMCGP could produce algorithms that can compute mathematical constants, like π and e, to arbitrary precision (Harding et al., 2010b). We were able to prove that two of the evolved formulae (one for π and one for e) rapidly converged to the constants in the limit of large iterations. We consider this work to be significant as evolving provable mathematical results is a rarity in evolutionary computation.

The fitness function was designed to produce a program where subsequent iterations of the program produced more accurate approximation to π or e. Programs were allowed to iterate for a maximum of 10 iterations. If the output after an iteration did not better approximate π, evaluation was stopped and a large fitness penalty applied. Note that it is possible that after the 10 iterations the output value diverges from the constant and the quality of the result would therefore worsen.

We analyzed one of the solutions that accurately converges to π. It had the generating function:

$$f(i) = \begin{cases} cos(sin(cos(sin(0)))) & i = 0 \\ f(i-1) + sin(f(i-1)) & i > 0 \end{cases} \qquad (1.1)$$

Equation 1.1 is a nonlinear recurrence relation and it can be proven formally that it is an exact solution in that it rapidly approaches π in the limit of large i.

Using the same fitness function as with π, evolving solutions for e was found to be significantly harder. In our experiments we chose the initial genotype to have 20 nodes and the ToDo list length to be 2. This meant that only two SM functions were used in each phenotype. We allowed the iteration number it as the sole program input. Defining $x = 4^{it}$ and $y = 4x = 4^{it+1}$ we evolved the solution for the output, z as

$$z = (1 + \frac{1}{y})^y \sqrt{1 + \frac{1}{y}} \tag{1.2}$$

Eqn 1.2 tends to the form of a well-known Bernoulli formula.

$$\lim_{y \to \infty} (1 + \frac{1}{y})^y \tag{1.3}$$

Evolving to Learn

In nature, we are used to the idea that plasticity (e.g., in the brain) can be used to learn during the lifetime of an organism. In the brain, the 'self-modification rules' are ultimately encoded in the genome. In (Harding et al., 2009b), we set out to use SMCGP to evolve a learning algorithm that could act on itself. The basic question being whether SMCGP can evolve a program that can learn - during the development phase - how to perform a given task. We chose the task of getting the same phenotype to learn all possible 2-input boolean truth tables. We took 16 copies of the same phenotype, and then tried to train each copy on a different truth table, with the fitness being how well the programs (after the learning phase) did at calculating the correct value based on a pair of inputs.

In SMCGP, the activation of a self modifying node is dependent on the values that it reads as inputs. Combined with the various mathematical operators, this allows the phenotype to develop differently in the presence of different sets of inputs. To support the mathematical operators, the Boolean tables were represented (and interpreted) as numbers, with -1.0 being false, +1.0 being true.

Figure 1-2 illustrates the process. The evolved genotype (a) is copied into phenotype space (b) where it can be executed. The phenotype is allowed to develop for a number of iterations (c). The number of iterations is defined by a special gene in the genotype. Copies of the developed phenotype are made (d) and each copy is assigned a different

truth table to learn. The test set data is applied (e) as described in the following section. After learning (f) the phenotype can now be tested, and its fitness found. During (f), the individual is treated as a static individual - and is no longer allowed to modify itself. This fixed program is then tested for accuracy, and its fitness used as a component in the final fitness score of that individual.
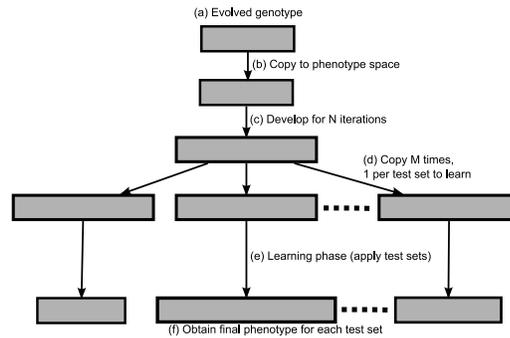

Figure 1-2. Fitness function flow chart, as described in section 3.0.

During the fitness evaluation stage, each row of the truth table is presented to a copy of the evolved phenotype (Figure 1-2.e). During this presentation, the error between the expected and actual output is fed back into the SMCGP program, in order to provide some sort of feedback. Full details of how this was implemented can be found in (Harding et al., 2009b).

During fitness calculation, we tested all 16 tables. However, we split the tables into two sets, one for deriving the fitness score (12 tables) and the other for a validation score (4 tables). It was found that 16% of experimental runs were successfully able to produce programs that correctly learned the 12 tables. None of the evolved programs was able to generalize to learn all the unseen truth tables. However, the system did come close with the best result having only 2 errors (out of a possible 16).

Figure 1-3 shows the form of the final phenotypes for the programs for each of the fitness truth tables. We can see both modularity and a high degree of variation - with the graphs for each table looking quite different from one another. This is in contrast to previous examples, such as the parity circuits, where we generally only see regular forms.

Figure 1-3.   Phenotypes for each of the tables learned during evolution.

4.   Iteration in SMCGP and GP

One of the unique properties of SMCGP is how it handles iteration. Iteration is not new in genetic programming and there are several different forms. The most obvious form of GP with iteration is Linear Genetic Programming (LGP), where evolved programs can execute inside a kind of virtual machine in which the program counter can be modified using jump operations. LGP operates on registers (as in a CPU), and uses this memory to store state between iterations of the same section of program. It is also worth noting that in LGP sub-sections of code are executed repeatedly. This is different from most implementations of tree-based GP (and we restrict our discussion to the simple, common varieties found in the literature), as the tree represents an expression, and so any iteration has to be applied externally. Tree-based GP also typically does not have a concept of working registers to store state between iterations, so these must be added to the function set, or previous state information passed back via the tree's inputs. Tree-based GP normally only has one output, and no intermediate state information. So additional mechanisms would be required to select what information to store and pass to subsequent iterations. In LGP termination can be controlled by the evolved program itself, whereby with external iteration another mechanism needs to be defined - perhaps by enforcing a limit to the number of iterations or some form of conditional.

SMCGP handles its iteration in a very different manner. It can be viewed as something analogous to loop-unrolling in a compiler, whereby

the contents of the loop are explicitly rewritten a number of times. In SMCGP, the duplication operator unrolls the phenotype. State information is passed between iterations by the connections made in the duplicated blocks. In compilers, it is done for program efficiency and is typically only done for small loops. In SMCGP, if the unrolling is excessive it will exceed the maximum permitted phenotype length. We speculate that this may help to evolve more efficient modularization. Because the activation of self modifying functions is determined by both the size of the ToDo list and the inputs to self modifying nodes, it is possible for SMCGP to self-limit when sections of code should be unrolled.

SMCGP's unrolling also has the possibility to grow exponentially, which forms a different kind of loop. For example, imagine a duplication operator that copied every node to its left and inserted it before itself : e.g NODE0 NODE1 DUPLICATE. On the next iteration it would produce NODE0 NODE1 NODE0 NODE1 DUPLICATE, then NODE0 NODE1 NODE0 NODE1 NODE0 NODE1 NODE0 NODE1 DUPLI-CATE and so on. Hence the program length almost doubles at each time. Similarly, the arguments for the duplication operation may only replicate part of the previously inserted module, so the phenotype would grow a different, smaller rate each time. Other growth progressions are also possible, especially when several duplication-style operators are at work on the same section of phenotype. This makes the iteration capabilities of SMCGP very rich and implies that it can also do a form of recursion unrolling - removing the need for explicit procedures in a similar way to the lack of need for loop instructions.

5. Conclusions and Further Work

Self modification in Genetic Programming seems to be a useful property. With SMCGP we have shown that the implementation of such a system can be relatively straightforward, and that very good results can be achieved. In upcoming work, we will be demonstrating SMCGP on several other problems including generalized digital adders and a structural design problem.

Here we have discussed problems that require some sort of developmental process, as the problems require a scaling ability. One benefit of SMCGP is that if the problem does not need self modification, evolution can stop using it. When this happens, the representation reverts to something similar to classical CGP. In (Harding et al., 2009c), we showed that on a bio-informatics classification problem where there should be no benefit in using self modification, SMCGP behaved similarly to CGP.

This result lets us be confident that in future work we can by default use SMCGP and automatically gain any advantages that development might bring.

The SMCGP representation has changed over time, whilst maintaining the same design philosophy. In future work we consider other variants as well. Currently we are investigating ways to simplify the genotype to make it easier for humans to understand. This should allow us to be able to prove general cases more easily, and perhaps explain how processes like the evolved learning algorithm function.

A whole world of self modifying systems seems to have become available now that the principle has been shown work successfully. We plan to investigate this world further and also encourage others to consider self modification in their systems.

References

Banzhaf, W. and Miller, J. F. (2004). *The Challenge of Complexity*. Kluwer Academic.

Harding, S., Miller, J. F., and Banzhaf, W. (2009a). Self modifying cartesian genetic programming: Parity. In Tyrrell, Andy, editor, *2009 IEEE Congress on Evolutionary Computation*, pages 285–292, Trondheim, Norway. IEEE Computational Intelligence Society, IEEE Press.

Harding, Simon, Miller, Julian F., and Banzhaf, Wolfgang (2009b). Evolution, development and learning with self modifying cartesian genetic programming. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 699–706, New York, NY, USA. ACM.

Harding, Simon, Miller, Julian F., and Banzhaf, Wolfgang (2010a). Developments in cartesian genetic programming: Self-modifying cgp. *To be published in Genetic Programming and Evolvable Machines.*

Harding, Simon, Miller, Julian F., and Banzhaf, Wolfgang (2010b). Self modifying cartesian genetic programming: Finding algorithms that calculate pi and e to arbitrary precision. In *Genetic and Evolutionary Computation Conference, GECCO 2010. Accepted for publication.*

Harding, Simon, Miller, Julian Francis, and Banzhaf, Wolfgang (2009c). Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing. In Vanneschi, Leonardo, Gustafson, Steven, et al., editors, *Genetic Programming, 12th European Conference, EuroGP 2009, Tübingen, Germany, April 15-17, 2009, Proceedings*, volume 5481 of *Lecture Notes in Computer Science*, pages 133–144. Springer.

Harding, Simon L., Miller, Julian F., and Banzhaf, Wolfgang (2007). Self-modifying cartesian genetic programming. In Thierens, Dirk, Beyer, Hans-Georg, Bongard, Josh, Branke, Jurgen, Clark, John Andrew, Cliff, Dave, Congdon, Clare Bates, Deb, Kalyanmoy, Doerr, Benjamin, Kovacs, Tim, Kumar, Sanjeev, Miller, Julian F., Moore, Jason, Neumann, Frank, Pelikan, Martin, Poli, Riccardo, Sastry, Kumara, Stanley, Kenneth Owen, Stutzle, Thomas, Watson, Richard A, and Wegener, Ingo, editors, *GECCO '07: Proceedings of the 9th annual con-*

ference on Genetic and evolutionary computation, volume 1, pages 1021–1028, London. ACM Press.

Huelsbergen, Lorenz (1998). Finding general solutions to the parity problem by evolving machine-language representations. In Koza, John R., Banzhaf, Wolfgang, Chellapilla, Kumar, Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max H., Goldberg, David E., Iba, Hitoshi, and Riolo, Rick, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 158–166, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann.

Khan, G.M., Miller, J.F, and Halliday, D.M. (2007). Coevolution of intelligent agents using cartesian genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 269 – 276.

Koza, J. R. (1994a). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press.

Koza, John R. (1992a). A genetic approach to the truck backer upper problem and the inter-twined spiral problem. In *Proceedings of IJCNN International Joint Conference on Neural Networks*, volume IV, pages 310–318. IEEE Press.

Koza, John R. (1994b). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts.

Koza, J.R. (1992b). *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, Cambridge, Massachusetts, USA.

Miller, J. F. and Smith, S. L. (2006). Redundancy and computational efficiency in cartesian genetic programming. In *IEEE Transactions on Evoluationary Computation*, volume 10, pages 167–174.

Miller, Julian F. (2003). Evolving developmental programs for adaptation, morphogenesis, and self-repair. In Banzhaf, Wolfgang, Christaller, Thomas, Dittrich, Peter, Kim, Jan T., and Ziegler, Jens, editors, *Advances in Artificial Life. 7th European Conference on Artificial Life*, volume 2801 of *Lecture Notes in Artificial Intelligence*, pages 256–265, Dortmund, Germany. Springer.

Miller, Julian F. and Banzhaf, Wolfgang (2003). Evolving the program for a cell: from french flags to boolean circuits. In Kumar, Sanjeev and Bentley, Peter J., editors, *On Growth, Form and Computers*. Academic Press.

Miller, Julian F. and Thomson, Peter (2000). Cartesian genetic programming. In Poli, Riccardo, Banzhaf, Wolfgang, Langdon, William B., Miller, Julian F., Nordin, Peter, and Fogarty, Terence C., editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh. Springer-Verlag.

Miller, Julian F. and Thomson, Peter (2003). 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.

Miller, Julian Francis (2004). Evolving a self-repairing, self-regulating, french flag organism. In Deb, Kalyanmoy, Poli, Riccardo, Banzhaf, Wolfgang, Beyer, Hans-Georg, Burke, Edmund K., Darwen, Paul J., Dasgupta, Dipankar, Floreano, Dario, Foster, James A., Harman, Mark, Holland, Owen, Lanzi, Pier Luca, Spector, Lee, Tettamanzi, Andrea, Thierens, Dirk, and Tyrrell, Andrew M., editors, *GECCO (1)*, volume 3102 of *Lecture Notes in Computer Science*, pages 129–139. Springer.

Spector, L. and Robinson, A. (2002). Genetic programming and auto-constructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3:7–40.

Spector, Lee and Stoffel, Kilian (1996). Ontogenetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 394–399, Stanford University, CA, USA. MIT Press.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

Vassilev, Vesselin K. and Miller, Julian F. (2000). The advantages of landscape neutrality in digital circuit evolution. In *Proceedings of the Third International Conference on Evolvable Systems*, pages 252–263. Springer-Verlag.

Wong, Man Leung (2005). Evolving recursive programs by using adaptive grammar based genetic programming. *Genetic Programming and Evolvable Machines*, 6(4):421–455.

Wong, Man Leung and Leung, Kwong Sak (1996). Evolving recursive functions for the even-parity problem using genetic programming. In Angeline, Peter J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA.

Yu, Tina and Miller, Julian (2001). Neutrality and the evolvability of boolean function landscape. In Miller, Julian F., Tomassini, Marco, Lanzi, Pier Luca, Ryan, Conor, Tettamanzi, Andrea G. B., and Langdon, William B., editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 204–217, Lake Como, Italy. Springer-Verlag.