Evolution, Development and Learning Using 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

Self-Modifying Cartesian Genetic Programming (SMCGP) is a form of genetic programming that integrates developmental (self-modifying) features as a genotype-phenotype mapping. This paper asks: Is it possible to evolve a learning algorithm using SMCGP?

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Learning; I.2 [**Artificial Intelligence**]: Automatic Programming

General Terms

Algorithms

Keywords

Developmental systems, genetic programming

1. INTRODUCTION

In natural evolution the notion of time plays an important, if not essential role. At a low level, information stored long-term in DNA is transcribed into messenger RNA, a process that is controlled by time-dependent transcription control that in itself requires time. RNA is subsequently subject to numerous time-dependent process before being translated into proteins which themselves have a finite lifetime. At a higher level, time dependent networks of gene-protein interactions take place and these patterns of activity characterize and cause the differentiation cells into different types. At a higher level still, in multicellular development, cells replicate and differentiate in ways that depend on time and their environmental surroundings. Moving a few levels still higher, neurons adjust and form new connections over time, in response to their own internal communications and to external environmental signals, a process that leads to learning. In artificial neural networks, learning often occurs through a time independent algorithm which adjusts the strengths of connections (weights). This approach is used even if evolutionary algorithms control the adjustment of weights. However, natural evolution does not work in this way, instead evolution operates at the very much lower level of genetic code and learning in organisms is an emergent consequence of many underlying processes and interactions with an external environment. The essential point here is that all learning occurs in the *lifetime* of the organism, a fact recently emphasized in [4, 5]. From the point of view of this paper, the key point is that evolution has invented learning algorithms (inasmuch as physical learning processes can be simulated by algorithms). In our view, the term 'development' means that there is a genotype-phenotype mapping that unfolds over time.

Previously we proposed a developmental mapping in the context of Cartesian Genetic Programming (CGP) by introducing self-modification (SM) [3]. However, as far as we can tell it was Spector and Stoffel who first introduced self-modification into genetic programming (GP) [8]. Self-modification functions are functions that carry out simple computations but also act as instructions to alter the phenotype. This approach to computational development (i.e. by augmenting a GP function set with SM functions) has a distinct advantage, in that the phenotypes are automatically computational entities. This means that the computational advantage of development can be assessed in a unified way and arbitrary additional mappings do not have to be devised to go from a phenotype to a computation (i.e. mapping a arrangement of cells to a computation). Another advantage is that if a problem does not require a time-dependent solution, it can automatically be achieved through the complete removal of SM instructions (i.e. the phenotype can shut down its own development). Recently, the efficiency of SMCGP has been shown to surpass CGP on a number of problems: generating a Fibonnaci sequence, a sequence of squares, sum of all inputs and a power function [1]. In addition, SMCGP has been shown to be able to find general solutions to parity [2] a problem that cannot be solved with CGP (since it has a bounded genotype).

In this paper, we show how SMCGP can invent a learning algorithm. We do this by evolving a genotype which, when developed and iterated with an error signal, can learn any desired two-input Boolean function. We have devised the problem so that the evolved program cannot generate all two input Boolean functions in one program, instead it learns *purely through an error signal* how to become the desired function.

GECCO'09, July 8–12, 2009, Montreal Quebec, Canada.

2. DEMONSTRATING THE ABILITY TO LEARN

Our aim is to demonstrate that we can evolve programs that have the ability to learn post evolution. This implies that we must be sure that changes in the behaviour (i.e. learning) happen only at run time and are not resident somehow in the genotype (i.e. directly discovered by evolution). To show that it is possible to evolve a learning algorithm, means that it is necessary to untangle the contribution to solving a problem that derives from each of the components (evolution, learning and development).

Unfortunately, it is not possible to just implement a framework that could allow for learning and then consider the end fitness score on an *arbitrary* task. To clarify this, consider the following scenario where the task is to evolve the controller for a soccer playing robot, with the intention that it can learn to play soccer at run time. The hope being that the robot gets better at playing over time when the evolved program is executed.

If it is observed that (after evolution) the player gets better the more games it plays, there are at least two possible reasons. It may be because the evolved player is learning soccer, and that a successful learning algorithm has been evolved. However, it may just be because the program is time dependent (perhaps the speed of the robot is a function of time). Here, the program may look like it is learning - but it is not.

It may be possible to prevent evolution from being the learning mechanism by testing the evolved program on a number of different problems. But does the ability to generalize reduce the chance that the learning is occurring at run time or during evolution? If a learning algorithm is evolved, it may be expected that it should be able to learn problems not seen during evolution.

However, just because something can learn to do one thing (e.g. sing in a girl band) does not mean that it can learn to do something else (e.g. act). So, perhaps even if a program is learning certain problems, our unseen learning examples may not be learnable to that particular program. Alternatively, it however is likely that learning to do things that are similar, or at least have some over lapping skills, (such as learning both piano and violin) are linked. There is an intuition here that a learning system should be able to learn more than one task - especially when they are similar. Therefore, under some circumstances, the ability to solve (or not) unseen problems is evidence of learning at run time.

Another possible way to determine where the problem solving ability comes from is to inspect the genotype and the phenotype. From this it may be possible to deduce the learning algorithm or even reverse engineer it. However, evolved programs are often convoluted and hard for a human to understand.

These observations indicate that designing a suitable fitness task for evolving a learning algorithm is difficult. To address many of these issues we have formulated a learning problem where the phenotype must learn a truth table (two-input) using the two Boolean inputs and an error signal. We are searching for a program that can modify itself so that it can correctly output any two-input Boolean function. We emphasize that we are not looking for a program that can output one two-input Boolean function (this is trivial) but rather a program that is capable of becoming any of the 16 possible two-input Boolean functions when presented with an error signal. The program we are looking for cannot merely encode a function that output the correct response for all two input Boolean functions simultaneously as we present each problem separately. So it *must* utilize the error signal. Thus we are trying to evolve a learning algorithm itself rather than a fixed function. As far as we are aware this problem has not been formulated before in the GP literature.

The type of learning we are investigating is similar in some ways to that investigated in artificial neural networks. However, in our approach, primitive functions can be any computational functions and our programs can alter their own topology and functionality at run time. The SMCGP approach is very general and neural networks could be seen as a special case.

3. SELF MODIFYING CGP

3.1 Cartesian Genetic Programming (CGP)

Cartesian Genetic Programming is a graph-based representation of programs [7]. Graphs are attractive data structures because of they allow multiple paths between nodes (as opposed to trees) so that sub-functions can be re-used. 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 a directed graph.

CGP uses a genotype-phenotype mapping that does not require all of the nodes to be connected to each other, resulting in a bounded variable length phenotype. This allows areas of the genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This type of neutrality has been investigated in detail [7, 10, 11] and found to be extremely beneficial to the evolutionary process on the problems studied.

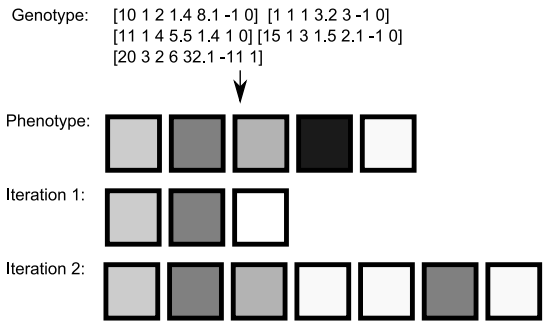

Figure 1: Conceptual view of SMCGP. The genotype maps directly to the initial graph of the phenotype. The genes control the number, type and connectivity of each of the nodes. The phenotype graph is then iterated to perform computation and produce subsequent graphs.

3.2 SMCGP

As in CGP, in SMCGP each node in the directed graph represents a particular function and is encoded by a number of genes. The first gene encodes the function of the node. This is followed by a number of connection genes (as in

CGP) that indicate the location in the graph where the node takes its inputs from. However SMCGP also has three real-valued genes which encode parameters that may be required for the function (primarily SM functions use these and in many cases they are truncated to integers when necessary, see later). It also has a binary gene that indicates if the node should be used as an output. In this paper all nodes take two inputs, hence each node is specified by seven genes. An example genotype is shown in Figure 1.

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

If a gene specifies a connection pointing outside of the graph, i.e. with a larger relative address than there are nodes to connect to, then this is treated as connecting to zero value. Inputs arise in the graph through special functions. This is described in section 3.3.

This encoding is demonstrated visually in Figure 2. The relative addressing allows sub-graphs to be placed or duplicated in the graph (through SM operations) whilst retaining their semantic validity. This means that sub-graphs could represent the same sub-function, but acting on different inputs.

Section 5.1 details the available functions and any associated parameters.

3.3 Inputs and outputs

If connection genes address nodes beyond the start of the graph they return false (or 0 for non-binary versions of SMCGP). Inputs are acquired to programs through the use of special node functions, that we call INP. These are used in the following way. The leftmost INP node outputs the first program input, each successive INP node gets the next input from the available set of inputs. If the INP node is called more times than there are inputs, the counting starts from the beginning again, and the first node is used.

Outputs are handled in a slightly different way than inputs. Each SMCGP node has a binary gene that defines if the phenotype should attempt to use that node as a program output. When an individual is evaluated, the first stage is to identify the nodes in the graph that have their output gene set to 1. Once these are found, the graph can be evaluated from each of these nodes in a recursive manner.

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

3.4 Evaluation of the SMCGP graph

The outline of the process of evaluating a genotype 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.

The genotype is invariant during the entire evaluation of the individual as perform all modifications on the phenotype beginning with a copy of the genotype. In subsequent iterations, the phenotype will usually gradually diverge from the genotype.

The encoded graph is executed in the same manner as standard CGP, but with changes to allow for self-modification. The graph is executed by recursion, starting from the output nodes down through the functions, to the input nodes. In this way, nodes that are unconnected are not processed and do not affect the behavior of the graph at that stage.

For function nodes (e.g. AND, OR, XOR) the output value is the result of the mathematical operation on input values.

Each active (non-junk) graph manipulation function (starting on the leftmost node of the graph) is added to a "To Do" list of pending modifications. After each iteration, the "To Do" list is parsed, and all manipulations are performed (provided they do not exceed the number of operations specified in the user defined "To Do" list length). The parsing is done in order of the instructions being appended to the list, i.e. first in is first to be executed.

The length of the list can be limited as manipulations are relatively computationally expensive to perform. Here we limit the length to just 2 instructions. All graph manipulation functions use extra genes as parameters. This is described in section 5.1.

4. EVOLUTIONARY ALGORITHM AND PARAMETERS

We use an (1+4) evolutionary strategy for the experiments in this paper. This has been shown to be an effective evolutionary algorithm in CGP [7]. However, the initial population consists of 50 random individuals. We then select the best individual and generate four offspring. We test these new individuals, and use the best of these to generate the next population. During initial experiments, it was observed that the length of the phenotype became extremely long and that this would hinder understanding the programs. Selection was therefore modified to select the shorter of two phenotypes for individuals with the same fitness.

We have used a relatively high (for CGP) mutation rate of 0.1. This means that each gene has a probability of 0.1 of being mutated. SMCGP, like normal CGP, allows for different mutation rates to affect different parts of the genotype (for example functions and connections could have different mutation rates). In these experiments, for simplicity, we chose to make all the rates the same. Mutations for the function type and relative addresses themselves are unbiased; a gene can be mutated to any other valid value.

For the real-valued genes, the mutation operator can choose to randomize the value (with probability 0.1) or add noise (normally distributed, sigma 20). The evolutionary parameters we have used have not been optimized in any way, so we expect to find much better values through empirical investigations.

Evolution is limited to 10,000,000 evaluations. Trials that fail to find a solution in this time are considered to have failed. It is important to note we are not evolving a function but a learning algorithm (a SMCGP program) capable of

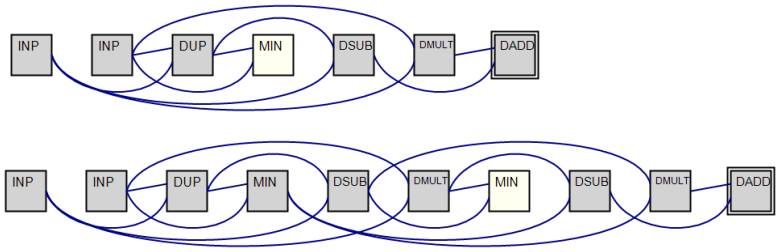

Figure 2: Example program execution. Showing the DUP(licate) operator being activated, and inserting a copy of a section of the graph (itself and a neighboring functions on either side) elsewhere in the graph in next iteration. Each node is labeled with a function, the relative address of the nodes to connect to and the parameters for the function (see Section 3.4).

self-adapting to be any two-input Boolean function. Since this problem has not been investigated before we do not know how difficult a problem this is, but our experience in using SMCGP on other problems indicates that it does have a high level of difficulty.

These evolutionary parameter values have not been optimized.

5. FUNCTION SET

The function set is defined in two parts. The computational operations as defined in table 1. The other part is the set of modification operators. These are common to all data types used in SMCGP.

5.1 Definition of Modification Functions

The self-modifying genotype (and phenotype) nodes contain three double precision numbers, called "parameters". In the following discussion we denote these P_0,P_1,P_2. We denote the integer position of the node in the phenotype graph that contained the self modifying function (i.e. the leftmost node is position 0), by x. In the definitions of the SM functions we often need to refer to the values taken by node connection genes (which are all relative addresses). We denote the jth connection gene on node at position i, by c_{ij}. There are several rules that decide how addresses and parameters are treated:

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

The functions (with the short-hand name) are defined as follows:

Duplicate and scale addresses (DU4) Starting from position $(P_0 + x)$ copy (P_1) nodes and insert after the node at position $(P_0 + x + P_1)$. During the copy, c_{ij} of copied nodes are multiplied by P_2.

Shift Connections (SHIFTCONNECTION) Starting at node index $(P_0 + x)$, add P_2 to the values of the c_{ij} of next P_1.

ShiftByMultConnections (MULTCONNECTION) Starting at node index $(P_0 + \mathrm{x})$, multiply the c_{ij} of the next P_1 nodes by P_2.

Move (MOV) Move the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$.

Duplication (DUP) Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$.

DuplicatePreservingConnections (DU3) Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$. When copying, this function modifies the c_{ij} of the copied nodes so that they continue to point to the original nodes.

Delete (DEL) Delete the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$.

Add (ADD) Add P_1 new random nodes after $(P_0 + x)$.

Change Function (CHF) Change the function of node P_0 to the function associated with P_1.

Change Connection (CHC) Change the $(P_1 mod 3)$th connection of node P_0 to P_2.

Change Parameter (CHP) Change the $(P_1 mod 3)$th parameter of node P_0 to P_2.

Overwrite (OVR) Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ to position $(P_0 + x + P_2)$, replacing existing nodes in the target position.

Copy To Stop (COPYTOSTOP) Copy from x to the next "COPYTOSTOP" function node, "STOP" node or the end of the graph. Nodes are inserted at the position the operator stops at.

6. FITNESS FUNCTION

The evolved SMCGP programs operate on floating point numbers (C♯ doubles). The function set is shown in table 1.

Each program is tested on its ability to learn a number of different problems: 2-input truth tables. Each of the sixteen 2-input truth tables is mapped to a numeric table in which true is replaced with 1.0 and false by -1.0. Each of the 16 phenotype program derived from the genotype has to independently give the correct outputs for each of the four rows in the table it has been assigned to. If the real valued program outputs a number that is less than zero we say it has output false, otherwise we say it outputs true.

Each table is referred to by the number represented by its binary string output (i.e. table 0 outputs all zero. table 3 outputs 0011 etc).

Function name	Description
No operation (NOP)	Passes through the first input.
Add, Subtract, Multiply, Divide (DADD, DSUB, DMULT, DDIV)	Performs the relevant mathematical operation on the two inputs.
Const (CONST)	Returns a numeric constant as defined in parameter P_0.
Average (AVG)	Returns the average of the two inputs.
Node index (INDX)	Returns the index of the current node. 0 being the first node.
Input count (INCOUNT)	Returns the number of program inputs.
Min, Max (MIN, MAX)	Returns the minimum/maximum of the two inputs.

Table 1: Numeric and other functions.

The fitness function tests only 12 of the 16 programs to see if they can learn a different truth table. The remaining 4 tables, tables 1, 6, 11, and 12 are reserved for testing generality (the validation stage). The tables were chosen so that the expected outputs of the training and validation sets both contained 50% of TRUE bits.

The fitness score is the sum of the error of the output for each truth table. We refer to each truth table as a test set. A row in a truth table is a test case.

Figure 3 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). This 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.

An error is calculated for each test set, with the fitness for the individual being the sum of the errors of all the test sets. Hence, a lower fitness value is better.

6.1 Presenting the data to the evolved learner

During the fitness evaluation stage, a truth table is presented to a copy of the evolved phenotype (Figure 3.e). The algorithm for that presents the data to the learning algorithm is:

```
For each row in truth table:
   Repeat 10 times:
     Apply row of truth table and
     last error to evolved program.
     Iterate program, and get output.
     Calculate error.
     Execute self modification
     operators ('ToDo' list).
     If the error for that row is 0
     then break.
```

7. DISCUSSION OF THE FITNESS FUNCTION

The fitness function (described in section 6) aims to address some of the issues discussed in section 2. For example:

The evolved program is tested on its ability to learn a number of different, yet related problems. Each truth table performs a different computation, however they are all taken from the same 'set' of problems. Each truth table also has other tables that are very similar (e.g. 1 bit difference in the output, or the outputs are simply the inverse of another tables.)

The task itself is simple for a human. It is therefore hoped that the evolved programs are relatively easy to understand (but of course this is not guaranteed).

There are unseen problems from the same, related set that can be tried. The fitness function breaks the full set of truth tables into two parts (learning and validation), it is therefore possible to see if the algorithm can generalize. Importantly, the problems are related and therefore it would be expected that generalization is possible. We do not test for generalization on an unrelated problem e.g. a robot control or game playing, where generalization would be unlikely.

However, there are still some potential problems with the approach:

The evolved program may cheat. It may discover a program that based on the error can pick between evolved solutions to each of the problem. In other words, the learning may be some form of genetic memory. However, the genotype used here contains only 20 nodes. It is hoped that rather than attempt to compress 16 different truth table representations into this space, evolution will instead opt to produce a learning algorithm.

The program may be too hard to understand to deduce the learning rules. In previous work with SMCGP, it has been possible to understand the evolved programs [2]. However, not all of the evolved solutions were easy to follow. Finding a human comprehensible solution may be unlikely, especially considering the complexity of the problem shown here.

8. RESULTS

111 experiments were completed. Of these, 18 experiments were successfully able to produce programs that correctly learned the 12 tables. On average for the successful runs, 4.4 million evaluations were required (minimum 0.41 million evaluations, standard deviation 2.64 million). On average, 8.6 tables were correctly learned.

None of the evolved programs were able to generalize to learn all the unseen truth tables. The best result had 2 errors (out of a possible 16). The worst had 9, with 50% TRUEs in the outputs, this result is worse than guessing. On average, solutions that learned the initial 12 tables had 5.67 (std. dev. 1.8) errors in the unseen tables.

In this section the behaviour of the best generalizing solution is examined. This individual can successfully learn 14 tables (and makes one mistake in each of the other tables).

In the figures, unconnected (or 'junk') nodes in the phenotype are represented as smaller shapes and their connections

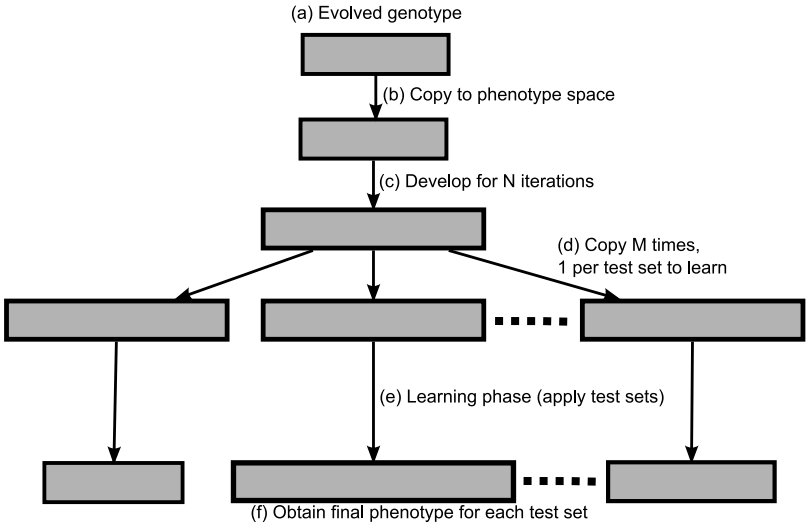

Figure 3: Fitness function flow chart, as described in section 6.

are omitted for clarity. The colours of each node correspond to their function type. For space reasons, all the labeling of the graphs cannot be shown. However, the figures do illustrate the general form of the graph.

Figure 4 shows the phenotype (that is the same graph as represented by the genotype) and the phenotype after the initial development stage. In this example, the phenotype and developed phenotype are the same, however a visual inspection of other results show that this is atypical.

It should be noted that the genotype contains 20 nodes, with only 8 of those used in the initial phenotype. The self modification and learning therefore gives a very good compression (14 accurate tables, or 62 bits out of 64). The length of the phenotype does not appear to be related to the complexity of the table to be learned.

After development has finished the learning stage begins. Figure 7 shows each change to the phenotype during the learning stage for two example tables, with table 6 on the left and table 10 on the right.

Tables 6 and 10 both have similar output patterns (FTFT and FTTF respectively), where the first two outputs are the same. Looking at figure 7 it is possible to see how the phenotype diverges during learning. The first row in the illustration shows the original phenotype. The next row in the figure shows result of the first row of the truth table being presented once. The second row of the truth table is presented 4 times, and we have shown the changes that happen to the graph in the next four rows. After this, the graphs begin to diverge. For table, 6 the third row is presented once but when learning table 10 it is presented twice. When learning table 6, the final row is also presented just once - with the final row being presented 4 times when learning table 10. The final row in the figure shows the final phenotype that represents that truth table.

Figure 5 shows the phenotype graph for each of the 12 tables tested during the evolutionary stage. Figure 6 shows the phenotypes for the validation tables. We see that the phenotypes vary greatly in length and also in structure. Table 2 compares the output of the programs on the unseen truth tables. Of particular interest is that table 6 (XOR) was correctly learned. Both bad tables have 1 bit wrong.

Table	Expected	Actual	Correct?
1 (NOR)	TFFF	TFFT	YYYN
6 (XOR)	FTTF	FTTF	YYYY
11	TTFT	TTFT	YYYY
12	FFTT	FFTF	YYYN

Table 2: Output from validation (unseen) programs. (T=TRUE, F=FALSE, Y=Yes, N=No).

9. CONCLUSIONS

Although we failed to evolve a program that was a perfect Boolean function learner we did evolve programs that could learn up to 14 out of the sixteen possible two-input truth tables. The mechanism of the evolved algorithm is unclear. One goal was to remove all predefined notions of how a learning algorithm should work. However we concede that the method of presenting the data and error to the phenotype may bias the search to some form of supervised learning. In future, we would like to devise a system that does not have this bias.

In the work described the learning signal was the error after applying a truth table. However, the learning signal could be viewed as an environmental condition. Perhaps this approach is also suitable for learning tasks where there is no natural notion of error?

Figure 5 shows that the generated phenotypes vary greatly in form. This shows that this approach is capable of producing a diverse set of phenotypes not only in respect to form but to function. We believe that this should improve evolvablity as it means that the evolved genotype is very good at producing a wide variety of phenotypes. It is also interesting to note that there does not appear to be any relationship between the length of the phenotype and complexity of the truth table (i.e. whether the output of the truth table depends on either none, one or both inputs).

Previous developmental systems (such as [6]) have shown the ability to produce a range of phenotypes for a single problem under different environmental conditions. Others have evolved genotypes that can take on two different phe-

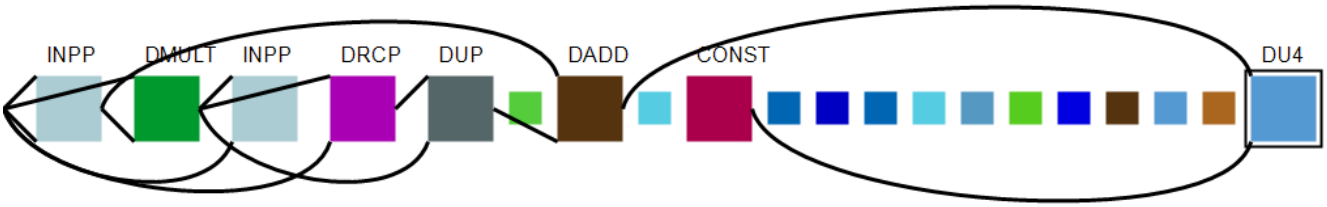

Figure 4: Initial phenotype (and genotype).

Figure 5: Phenotypes for each of the tables learned during evolution. The remaining tables (1, 6, 11, and 12) were reserved for validation on unseen examples.

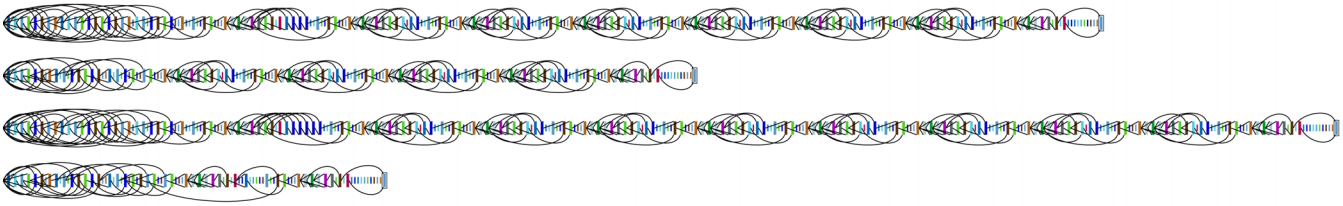

Figure 6: Phenotypes for each of the tables (1, 6, 11, and 12) tested during validation. The ability for the phenotype to generate these (mostly correct) programs helps demonstrate that there is some form of run time learning occuring.

Figure 7: Detail of the learning stage for tables 6 (FTTF) and 10 (FTFT). The phenotype graphs are identical until the truth table outputs diverge and a different error signal is returned. See section 8 for a description.

notypes depending on the environment [9]. In this paper we have shown the ability to produce a range of phenotypes that solve many different problems. Our aim is to show that a single phenotype is able to solve all the different possible instances (i.e. all input combinations) of all different possible problems (i.e. all possible 2 input truth tables).

We believe that one of the benefits of SMCGP is the consistency of the representation through evolution, development and now learning. Our current implementation of the evolutionary algorithm also uses the same self modifying operators that are used during evolution/development on the genotype graph during crossover, mutation and other house keeping tasks. This leads to the obvious next step of attempting to use SMCGP to evolve its own evolutionary algorithm (and by extension developmental/learning). One could also use a different set of self-modifying operations for development and learning.

10. REFERENCES

[1] S. Harding, J. F. Miller, and W. Banzhaf. Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing. In *Accepted for publication in EuroGP 2009*, 2009.

[2] S. Harding, J. F. Miller, and W. Banzhaf. Self modifying cartesian genetic programming: Parity. In *Accepted for publication in CEC 2009*, 2009.

[3] S. L. Harding, J. F. Miller, and W. Banzhaf. Self-modifying cartesian genetic programming. In D. Thierens and H.-G. Beyer et al, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 1, pages 1021–1028, London, 7-11 July 2007. ACM Press.

[4] G. M. Khan, J. F. Miller, and D. M. Halliday. Coevolution of intelligent agents using cartesian genetic programming. In D. Thierens and H.-G. Beyer et al, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 1, pages 269–276, London, 7-11 July 2007. ACM Press.

[5] G. M. Khan, J. F. Miller, and D. M. Halliday. Breaking the synaptic dogma: Evolving a neuro-inspired developmental network. In X. Li, M. Kirley, and M. Zhang et al., editors, *SEAL*, volume 5361 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2008.

[6] T. Kowaliw, P. Grogono, and N. Kharma. Environment as a spatial constraint on the growth of structural form. In D. Thierens and H.-G. Beyer et al, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 1, pages 1037–1044, London, 7-11 July 2007. ACM Press.

[7] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli and W. Banzhaf, et al.,, editors, *Proc. of EuroGP 2000*, volume 1802 of *LNCS*, pages 121–132. Springer-Verlag, 2000.

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

[9] G. Tufte and P. Haddow. Extending artificial development: Exploiting environmental information for the achievement of phenotypic plasticity. In L. L. Kang, Y. Liu, and S. Zeng, editors, *Proceedings of International Conference on Evolvable Systems*, volume 4684 of *LNCS*, pages 297–308. Springer-Verlag, 2007.

[10] V. K. Vassilev and J. F. Miller. The advantages of landscape neutrality in digital circuit evolution. In *ICES '00: Proceedings of the Third International Conference on Evolvable Systems*, volume 1801, pages 252–263. Springer-Verlag, 2000.

[11] T. Yu and J. Miller. Neutrality and the evolvability of boolean function landscape. In J. F. Miller and M. Tomassini et al., editors, *Proceedings of EuroGP 2001*, volume 2038 of *LNCS*, pages 204–217. Springer-Verlag, 2001.