SMCGP2: Self Modifying Cartesian Genetic Programming in Two Dimensions

Simon Harding
Department of Computer Science
Memorial University
Newfoundland, Canada
simonh@cs.mun.ca

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

Wolfgang Banzhaf
Department of Computer Science
Memorial University
Newfoundland, Canada
banzhaf@cs.mun.ca

ABSTRACT

Self Modifying Cartesian Genetic Programming is a general purpose, graph-based, developmental form of Cartesian Genetic Programming. Using a combination of computational functions and special functions that can modify the phenotype at runtime, it has been employed to find general solutions to certain Boolean circuits and mathematical problems. In the present work, a new version, of SMCGP is proposed and demonstrated. Compared to the original SMCGP both the representation and the function set have been simplified. However, the new representation is also two-dimensional and it allows evolution and development to have more ways to solve a given problem. Under most situations we show that the new method makes the evolution of solutions to even parity and binary addition faster than with previous version of SMCGP.

Categories and Subject Descriptors

I.2.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming; D.1.2 [**Software**]: Automatic Programming

General Terms

Algorithms

Keywords

Genetic programming, developmental systems

1. INTRODUCTION

Self Modifying Cartesian Genetic Programming (SMCGP) is a generalization and modification of the graph-based form of Genetic Programming (GP) known as Cartesian Genetic Programming (CGP) [8],[10]. In addition to usual computational functions, this generalization includes functions that can modify the program encoded in the genotype. SMCGP is developmental in nature in that evolved programs encoded in the genotype can be iterated to produce an infinite sequence of programs (a developing phenotype). The fitness of a genotype can be measured incrementally over time (after each iteration). In addition, during iterations programs (phenotypes) are able to acquire more inputs and produce more outputs. It has been shown that this allows genotypes to be evolved that represent *general* solutions to computational problems. For instance, the technique has been shown to be able to find general solutions to parity, binary addition, and also algorithms that compute mathematical constants that are exact in the limit (i.e. π and e) [4]. SMCGP has hitherto been based on a one-dimensional form of CGP (i.e. one in which there is only one row of nodes). In this paper we describe a two-dimensional form of SMCGP (SMCGP2). The new technique is applied to two well known problems: building even-parity solutions and binary addition. We show that SMCGP2 can find programs that appear to be general solutions, to the degree tested, faster than SMCGP or any other previously known technique.

GECCO'11, July 12–16, 2011, Dublin, Ireland.

2. FROM CGP TO SMCGP

CGP encodes (generally acyclic) computational structures in the form a two-dimensional grid of primitive functions (nodes) and the connections between them. The genotype is a list of integers of three kinds: node connections, node functions, and program outputs. The node connection genes encode the absolute addresses in a data array where the node gets its input data from. The node function genes encode the absolute addresses of function types defined in a lookup table. The program output genes hold the data addresses where the program outputs are to be taken from. CGP has been shown to be a highly competitive form of genetic programming [13].

Unlike CGP, SMCGP encodes connections between nodes using relative addressing [2]. That means that a node with connection gene, g, at node position p in a genotype, obtains its input data from a node at position $p - g$. It retains function addresses. The major change from CGP is that SMCGP includes functions that cause changes to the representation itself. These function require extra genes which encode information about how these changes are to be carried out. The extra genes are called 'arguments'. Later versions of SMCGP introduced a new mechanism for acquiring inputs and delivering outputs [4]. This new version retains many of these features, however there are some significant differences, and is fully described in the next section.

Gene	Type	Description
Function	Integer address of function type	Computational function of the node.
Constant	Real	Floating point number that is used to encode a value.
Input connections	Address x 2	Relative addresses of the inputs to the node function.
SM Location	Address	Relative address where SM operations should occur.
SM Size	Address	Size of the associated SM operator.

Table 1: Genes needed to encode a node in SMCGP2.

3. REPRESENTATION

The representation used in SMCGP was designed to be compact. However, this lead to the overloading of the argument genes in each node, where the same gene could be interpreted in different ways. This reduced the ability for the programs to be human readable. In the new version, the representation has been significantly changed to improve readability.

Like standard CGP, SMCGP2 genotypes represent computation using a 2D grid of nodes. As with CGP there are two user defined parameters, height and width which are respectively synonymous with what are termed in CGP, number of columns and number of rows. If the height is set to 1, then the representation becomes 1D, and similar to the previous version of SMCGP. The content of each node is described in Table 1. As with the previous version of SMCGP, relative addresses are used for the node connections of the graph. The address type is a pair of integers, representing how many nodes back in the horizontal and vertical dimensions the address refers to. The horizontal address must always be greater than 0. This prevents cycles in the program as nodes can only connect to nodes in columns to their left.

If the function in a node is a SM operator, the SM location and size genes are used. For the SM location, the addresses can be negative. This means the SM operations can be applied anywhere in the grid. The SM size address specifies the height and width of the SM operator.

The constant gene is used to store a real numbered value that can be used by the node's function. For example, if the node is the 'Const' function, then the value returned by that node is the constant gene's value.

All genes within each node are under evolutionary control, and can be mutated with equal probability.

Some of the SM operators add empty space into the program. Empty space did not exist in the previous version of SMCGP, and is a natural consequence of operating in 2D. Empty space can be converted to nodes in two ways: either overwritten by SM functions, or mutated into functional nodes. If a mutation operator is applied to an empty node, it gets converted into a randomly generated node. Similarly, a node can be mutated to an empty space. Genes encoding empty space are redundant and are not used in calculating fitness.

3.1 Interpreting a SMCGP2 Genotype

Interpreting the genotype in SMCGP2 is similar to that in SMCGP. First, the phenotype is created by making a copy of the genotype. All SM operations apply to this phenotype and do not modify the genotype.

Next, the output nodes need to be identified. This is done by scanning the phenotype from either top-left to bottom-right (forwards) or from bottom-right to top-left (backwards), and remembering which nodes have the function type 'Output'. The direction of the parse is a user defined parameter of the experiment, and we investigate its effect later in this paper.

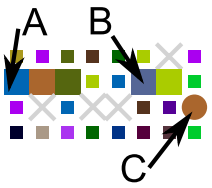

Figure 1: Example showing how empty space affects computing relative addresses. Empty nodes are represented by X. From C to B, the relative address is 2,1 - meaning 2 nodes to the left, and one node up. From C to A, the relative address is 4,1. Note how the empty nodes are not counted when computing how many nodes back to connect.

Once the output nodes have been found, the values of the nodes that connect to them can be computed. This is done by following the relative addresses, in a recursive manner, through the genotype. If a node contains a self-modifying operation, then a copy of the node is appended to a 'To Do' list. After each iteration, the SM operations listed in the 'To Do' list are processed to produce the next iteration of the phenotype. To prevent the phenotype from growing too quickly and to improve human readability, the length of the list is limited (here to just 2 operations). Hence, at most two SM operations will occur per iteration.[1]. SM nodes have a passive role in computation, and if they are called they pass through the first input to the calling node.

Empty space is taken into consideration when computing where relative addresses connect to. The relative address is computed using only non-empty nodes. Figure 1 shows an example of two address computations.

As with CGP and SMCGP, only nodes that are connected to the outputs need to have their values computed. Even with large genotypes a relatively small number of nodes are in use, and hence the computation is efficient. Further, some nodes may be connected to more than once, and their value only has to be computed once.

3.2 Self Modifying Function Set

In work with the previous version of SMCGP, it was determined that many of the self-modifying (SM) operators were rarely or not effectively used. In the new version of SMCGP, the set of SM operators has been simplified.

[1]As with SMCGP, it is also possible to make the insertion onto the 'To Do' list conditional. For example, with numeric problems, a node may only be appended when the first input value is greater than the second.

The set of SM operators is described in Table 2. SM functions are now consistent in how they interpret the location and size of their action.

SM operations have two parameters: SM Location (SML) and SM Size (SMS). The location helps specify where in the phenotype modifications will occur, whilst the size determines how many nodes the operation may use. Where in the phenotype the SM operation occurs is relative to the location of the calling node (NL). For example, if we consider the duplication function ('DUP'), the nodes that are duplicated are found by adding the location address of the calling node to the SM Location (i.e. $SML + NL$). The number of nodes to copy is specified by SMS, which determines a region of a given height and width to copy. The size of this region includes empty nodes. Nodes are then duplicated and inserted at $NL + SML$.

Other SM operations work in a similar fashion.

3.3 Program Input and Output functions

SMCGP also introduced other functions, and as these were found to be beneficial they have been retained in SMCGP2. These functions are listed in Table 2.

When a program is executed, the interpreter maintains a pointer which points to a possible input of the program. Calling INP returns the value at the current position, and then moves the pointer to the next entry in the input list. Using the INPP function returns the input at the current position, and moves the pointer backwards. The pointer loops around the input list, so that there is always a valid input available. The SKIP function takes the numeric constant encoded in the node, and moves the pointer that many steps - again wrapping around as appropriate. It then returns the input value at that position. This allows a program to jump to any input in one call. Having functions that return input values in this manner allows the program to vary the number of program inputs with each iteration - it just needs to add appropriate calls to one of these functions. INP, INPP and SKIP do not connect to other nodes, i.e. they have arity zero.

Similarly, the nodes used as outputs for the program have a special OUTPUT function. The interpreter looks for OUTPUT nodes when determining the program stored in the phenotype. The number of OUTPUT nodes can change as the program is iterated, and this allows the number of program outputs to vary with iteration. Various measures need to be taken if the number of OUTPUT nodes does not equal the number of outputs defined by the task. If there are no OUTPUT nodes in the graph, then the last n nodes in the graph are used. If there are more OUTPUT nodes than are required, then the right-most OUTPUT nodes are used until the required number of outputs is reached. If the graph has fewer OUTPUT nodes than are required graph, then nodes are chosen as outputs by moving forwards from the right-most node flagged as an output. If there is a condition where not enough nodes can be used as outputs (as there are not enough nodes in the graph), the individual is labeled as corrupt and is given a bad fitness score to prevent selection.

4. EVOLUTIONARY ALGORITHM

As with SMCGP, a simple, 1+4 evolutionary strategy is applied. The best individual in the population is used to produce 4 offspring. The top ranked offspring replaces the parent. Where there is no measured improvement in fitness, the parent is replaced by an offspring of equal fitness. If all offspring are worse than the parent, the parent is retained. Crossover is not used in our examples, however SMCGP has been tested with crossover and appears to function correctly. Mutation used is probabilistic, with each gene having an equal probability of being altered.

5. TEST PROBLEM: PARITY

Parity is a well studied problem in both genetic programming and developmental systems. A previous form of developmental CGP was able to find solutions up to 5 inputs, but was unable to find general solutions [11]. In the first SMCGP paper, parity circuits of up to 8 inputs were evolved - however they were not tested for generalization [2]. Rule and grammar based systems have also been used to grow parity solutions. For example, L-systems have been used to generate a grammar for modifying GP trees that solved parity up to 12 inputs [6]. Using artificial protein rules, circuits up to 12 inputs were found [1]. In [1] the authors also compared to direct (i.e. non developmental) encodings, and were unable to produce circuits with more than 4 inputs. In general, direct encodings are unable to solve large instances of the parity circuit. In [12] circuits with 22 inputs are directly evolved using a novel crossover operator. General solutions have also been found using GP, [7] evolved machine language programs that could iterate over the bits in the input string and [14, 15] used recursion. These approaches produced programs rather than circuits to solve the problem. In contrast, the technique presented in this paper evolves programs that produce circuits.

Here, SMCGP2 is evaluated on the same problem and compared to SMCGP. SMCGP has been used to find programs that can generate even parity circuits of any size [3, 4], where the task is to evolve programs that iteratively generate circuits of different numbers of inputs. On the first iteration, the program implements a 3 input parity circuit. On the next iteration, after the SM operations have been applied, it implements 4 input parity. Each subsequent iteration produces a parity circuit of the next size. For this to function, the SM operations have to add new Boolean functions to perform the computation and also new functions to obtain the additional inputs. Fixed sized programs can only solve for a specific number of inputs. SMCGP was found to outperform more traditional approaches when the number of inputs was relatively large for fixed sized programs, e.g., with more than 5 inputs. SMCGP, however, was able to obtain general solutions, which could solve a problem of any size.

The same fitness function is employed here. In the case of parity, to begin with, we evolve for three input bits. When a successful solution is found, the fitness function requires that the program produces a three input circuit, followed by a four input circuit. The process continues in this way until we obtain a phenotype that correctly implements a parity circuit with twenty inputs. Fitness is defined as the number of correctly predicted output bits over all circuit sizes tested. The fitness function does not continue to test circuits that fail to produce a parity circuit of a given size.

Two different function sets are compared. Both contain all of the functions listed in Table 2. In addition, the "full" set contains all possible 2-input Boolean functions. The "reduced" set contains AND, NAND, OR and NOR.

To investigate the behaviour of SMCGP2 under various

SM Function	Description
DUP	Extracts a section, SML+NL to SML+NL+SMS, and inserts at SML+NL. This resizes the graph, extra space is left empty.
OVR	Extracts a section, SML+NL to SML+NL+SMS, and copies in SML+NL. Nodes may be overwritten, and the graph may be expanded if copied nodes go beyond the current graph size.
DEL	Deletes a section, SML+NL to SML+NL+SMS by replacing the nodes with empty spaces.
CROP	Crops a section, SML+NL to SML+NL+SMS. All other nodes are deleted.
DELROW	Deletes the row at vertical component of SML+NL.
DELCOL	Deletes the column at horizontal component of SML+NL.
ADDROW	Adds a row at vertical component of SML+NL.
ADDCOL	Adds a column at horizontal component of SML+NL.
Input/Output Functions	**Description**
INP	Returns the next input.
INPP	Returns the previous input.
SKIP	Skips N inputs, where N is the integer value of the constant stored in that node.
OUTPUT	Flags a node as being used as an output.

Table 2: Description of SM and Input/Output functions. NL = Location of calling node, SML = Location of SM action (i.e. the offset where the self-modification will occur), SMS = SM size (i.e. the size of the modification). NL, SML and SMS are 2D-addresses.

conditions, the parity problem was evaluated under a large number of conditions. This step allows us to chose the 'best' configuration for the algorithm, and allows us to compare SMCGP2 to other approaches more easily.

All combinations of the following parameters were tested: Genotype Width = 5, 10, 15, 20; Genotype Height = 1, 5, 10, 15, 20; Function sets = Full, Reduced; SM Size = 10, 100, 1000, 10000; Mutation Rates = 0.005, 0.01, 0.05, 0.1 .

Programs were evolved to solve parity from 2 to 20 inputs, and the individual experiment was considered successful if a program functioned correctly. Programs were then tested on their ability to generalize by solving 2 to 24 inputs.

Experiments were repeated 10 times, for each of the 1,280 different parameter configurations. Evolution was limited to 10 million evaluations.

6. RESULTS

The overall success rate, over all parameter configurations, was found to be 68.1%. 59.3% of all runs resulted in programs that generalised to solve all sizes of parity circuit tested. Table 3 shows the configurations with the best performance in terms of their ability to generalize, the number of evaluations required, with the different function sets. All circuits were always able to generalize to the larger parity circuits. Progams that are 1 node high do the worst, and the smallest configuration 1 high x 5 wide is always unsuccessful.

Table 3 shows the top ten best configurations for both the full and reduced function sets. The best configurations have similar performance, but the parameters vary considerably. This shows that SMCGP2 may be tolerant to parameter choices. Table 4 shows the average number of evaluations to evolve to a given sized parity circuit (or more specifically to find a program that when iterated produces all parity circuits up to that number of inputs). The results are based on the best ten configurations for both function sets. It should be noted that the criterion for judging the quality was the degree to which the evolved solutions generalized. The results for the full function set show that SMCGP2 is better than previously published results in SMCGP [4]. Since extensive comparisons were made there to other approaches, as far as we can tell, the SMCGP2 results are the most efficient yet seen. The results also indicate that general solutions are found at around 6 or 7 inputs, as the number of evaluations required to solve further problems remains fairly constant. The results for the reduced function set show that SMCGP2 does not perform as well as SMCGP. The reasons for this are currently unclear, but we expect that it may be the result of the larger genotypes and larger search spaces that SMCGP2 employs. Additionally, the self modifying functions are different in SMCGP2, and it is possible that a useful function has not been implemented. SMCGP2 also appears to find general solutions at 6 or 7 inputs with the reduced function set. However, it is important to remember that these techniques solve for a single instance of the parity circuit problem - and do not find general solutions.

Table 3 shows the best configurations, but it is also interesting to investigate the effect of each parameter in isolation. In Table 5, the results show that using the full function set is preferable to the reduced function set. Even though the search space is now larger (due to more functions), the solution space is also likely to be much larger.

From Table 5, it appears that for the parity problem the parse direction (i.e. if the graph interpreter starts looking for outputs at the top-left (forwards) or from bottom-right (backwards)) makes little difference to the success rate. From Table 3 it is clear that working backwards is common with the most successful and fastest configurations. In previous work with SMCGP both directions were examined, and forwards was found to be better in some circumstances. It is expected that this may be a problem dependent parameter. Working forwards tends to produce more compact solutions, as there are less active nodes in the phenotype. It is likely that parity is too simple a problem to run into problems with phenotypic bloat. In Table 9, the sizes of the graph are kept fairly compact, despite the fact that the genotypes are large and are interpreted backwards.

In SMCGP, a mutation rate of 0.1 was typically applied, a

Average evaluations	Std.dev	Height	Width	Mutation Rate	SM Size	Parse Direction
Reduced Function Set						
187,695	197,600	1	15	0.05	1000	Backwards
257,291	243,255	1	20	0.1	1000	Forwards
393,402	254,737	1	15	0.1	1000	Backwards
415,524	419,690	1	20	0.05	10000	Backwards
417,126	343,312	15	20	0.05	1000	Backwards
465,178	540,770	5	15	0.1	10000	Backwards
465,792	603,180	1	15	0.05	1000	Forwards
495,061	370,903	10	15	0.1	10000	Backwards
552,197	360,228	15	10	0.05	10000	Backwards
559,342	721,085	1	20	0.05	1000	Forwards
Full Function Set						
10,973	8,477	15	10	0.1	1000	Backwards
13,442	14,066	1	20	0.05	10000	Backwards
15,867	9,647	1	20	0.05	1000	Forwards
16,506	18,575	1	20	0.1	10000	Backwards
20,074	12,546	1	15	0.1	1000	Backwards
22,029	18,990	5	10	0.1	10	Backwards
22,486	42,504	1	20	0.1	1000	Backwards
25,766	40,991	5	20	0.1	10	Backwards
30,755	44,223	5	20	0.1	1000	Forwards
34,096	49,501	5	15	0.05	10	Backwards

Table 3: The top 10 performing configurations per function set. These configurations were able to solve the seen problems with the fewest evaluations and with 100% success.

	SMCGP2, Full		SMCGP2, Reduced		SMCGP, Full	SMCGP, Red.
Inputs	Avg	Std Dev	Avg	Std Dev	Avg	Avg
3	9,166	9,103	382,963	417,212	37,276	247,753
4	12,412	10,786	449,040	456,939	41,697	275,663
5	18,875	22,925	547,709	700,569	43,016	278,635
6	21,548	28,948	553,292	703,260	43,593	298,104
7	22,247	29,807	557,285	702,668	150,719	318,376
8	22,250	29,806	558,313	702,876	150,721	322,843
9	22,332	29,781	558,406	702,901	150,722	322,843
10	22,332	29,781	559,196	703,167	150,722	322,843
11	22,332	29,781	559,205	703,163	150,722	322,851
12	22,333	29,782	560,724	702,408	150,722	322,851
13	23,729	32,547	560,728	702,415	150,722	322,866
14	23,729	32,547	560,734	702,417	150,722	322,866
15	23,731	32,548	560,744	702,411	150,722	322,866
16	23,732	32,547	560,744	702,411	150,722	322,866
17	23,732	32,547	560,744	702,411	150,722	322,870
18	23,732	32,547	560,744	702,411	150,722	322,870
19	23,733	32,547	560,745	702,410		
20	23,733	32,547	560,745	702,410		

Table 4: Number of evaluations to solve parity. For SMCGP and SMCGP2 the evolved programs are developmental. For SMCGP2, the results are the average performance of the top 10 best configurations. Note best refers to best in terms of generalization, rather than speed of evolution.

Func. Set	Parse Direction	% Success	% Generalize
Full	Backwards	87.5	75.3
Full	Forwards	88.4	75.0
Full	Combined	88.0	75.1
Reduced	Backwards	59.8	54.5
Reduced	Forwards	41.6	36.4
Reduced	Combined	50.6	45.4

Table 5: Success rate and rate of generalization by the direction (or where results from both directions are combined) of the decoding of the phenotype and by function set type. The relationship is unclear, and this is again reflected in table 3. The combined results show that the full function set is much better at both evolving given problems as well as producing generalized solutions.

Func. Set	Mutation Rate	% Success	% Generalize
Full	0.005	84.2	74.1
Full	0.01	87.2	74.0
Full	0.05	89.9	75.1
Full	0.1	90.1	77.0
Reduced	0.005	33.2	30.3
Reduced	0.01	45.4	41.1
Reduced	0.05	63.6	56.5
Reduced	0.1	59.0	52.4

Table 6: Success rate and rate of generalization by mutation rate and function set. It generally appears that higher mutation rates are preferable. In previous SMCGP work, a mutation rate of 0.1 was typically used.

value that was chosen after some informal experimentation. In Table 6, it appears that for SMCGP2 a higher mutation rate is generally better. This is in contrast to classical CGP, where very low mutation rates have been found to be optimal [9].

SMCGP2 introduces the new parameter of 'SM size'. It was important to investigate how this affects results, as there was concern setting it too high would lead to phenotypic bloat. However, the results from Table 7 show that maximum larger sizes for the SM operations are preferable. This is somewhat surprising as Table 9 shows that the solutions are compact, and this is despite the ability for the SM operations to make very large changes at each step.

Table 8 shows the effect of the initial height and width of the genotype on evolution. The results for parity indicate that there exists a sweet spot where the genotype is neither too small to encode a solution (e.g. 1 node high by 5 wide) or too big to encode a compact program. For all widths, at height 20, there is a drop in the success rate for both the evolution and for the ability of the programs to generalize.

Table 3 indicated that the best height for the genotype is 1 node. This makes the genotype one dimensional, and therefore superficially similar to the SMCGP representation. In Table 9 the average change in size of the phenotype at different iterations are shown (other results omitted for space). The results show that SMCGP2 uses the 2D representation during phenotypic development, and this suggests that the 2D properties are important to SMCGP2. The results also show that the growth is fairly well managed

Function Set	SM Size	% Success	% Generalize
Full	10	78.8	78.5
Full	100	88.4	45.6
Full	1,000	93.1	87.6
Full	10,000	92.4	91.6
Reduced	10	49.2	49.0
Reduced	100	26.2	12.8
Reduced	1,000	63.7	57.2
Reduced	10,000	65.8	65.2

Table 7: Success rate and rate of generalization by SM size and function set. In general, the larger the amount of phenotype can change, the better the Success rate.

in these cases, and there is no excessive bloat - despite the fact the phenotype is limited to 1 million nodes. In future work, it will be interesting to observe optimal genotype dimensions for other problems.

As an illustration, Figure 2 shows an example of parity circuit development. The left most program solves for 2 input parity. The next programs solve for 3,4 and 5 inputs. The right-most program shows the phenotype solving for 12 input parity. Although there are obvious regularities, it is interesting to note that the behavior of the active nodes on the left differs throughout the developmental stages. In previous work on parity with SMCGP, irregular growth was not observed. We therefore speculate that SMCGP2 may have interesting properties in producing irregular, complex patterns.

7. ADDER

To verify that SMCGP2 can also function on other Boolean problems, evolution was used to evolve a digital adder. Previously SMCGP has been used to find solutions, including general solutions, to this problem. Unlike parity the number of outputs also increases as the number if inputs increases. For brevity, the adder evolution was performed using only the best configuration from the parity results. This is unlikely to be the best configuration, and in future work we will explore the parameter space for this problem. The experiment was repeated 50 times.

Table 10 shows the average number of evaluations required to solve to a given sized parity circuit. Evolution was allowed to find programs that scaled from 1 + 1 bit adder to 8+8 bit adder. All runs were successful. Further, all but one run successfully generalized to solve up to 10+10 bit adder. The results with SMCGP2 are highly competitive with previously published results in SMCGP [4].

8. CONCLUSIONS

This paper has introduced SMCGP2, and has shown that it is capable of efficiently evolving what appear to be general solutions for parity and adder circuits. The results are either better, or highly competitive, with previous work.

SMCGP2 has many advantages over SMCGP. The simplified representation and function set should make programs more human readable. The use of a 2D representation allows for much greater expressiveness of the phenotype. This may allow evolution to find solutions to more complicated problems than was previously possible. We invite suggestions for potential challenges.

		Full Function Set		Reduced Function Set	
Width	Height	% Success	% Generalize	% Success	% Generalize
5	1	0.0	0.0	0.0	0.0
5	5	95.9	82.8	27.3	22.5
5	10	98.4	83.3	25.1	20.8
5	15	97.3	87.7	22.3	18.3
5	20	96.3	83.6	15.9	15.2
10	1	89.4	79.9	45.0	44.0
10	5	98.8	84.5	76.0	65.4
10	10	99.5	87.7	64.9	58.0
10	15	100.0	83.2	64.0	57.5
10	20	99.0	84.8	52.8	46.4
15	1	75.4	64.4	51.2	48.4
15	5	96.7	81.3	71.0	64.1
15	10	99.5	84.0	76.4	68.6
15	15	99.4	82.2	66.8	56.6
15	20	98.9	84.3	50.6	46.6
20	1	69.4	55.7	51.5	47.4
20	5	99.2	85.2	76.1	72.9
20	10	98.4	83.9	78.3	67.2
20	15	98.2	82.7	62.6	55.5
20	20	99.4	81.5	56.5	50.0

Table 8: Success rate and rate of generalization by function set type and by genotype size for evolution of parity. Evolution is unable to find solutions when given a 5 x 1 genotype, which is probably because the genotype is too small. For the full function set, the taller the genotype, the better the results. For the reduced genotype, this does not appear to be the case. Here, heights 5 and 10 seem better for each width.

	Full Function Set				Reduced Function Set			
	Change between iterations		Change from start		Change between iterations		Change from start	
Inputs	Width	Height	Width	Height	Width	Height	Width	Height
2	23.5	3.8	6.0	1.2	22.5	7.9	6.5	1.9
3	12.7	1.6	22.3	2.9	16.4	2.6	29.6	4.3
4	16.4	1.7	39.9	4.4	18.7	1.8	47.7	4.1
5	18.3	2.3	54.1	7.0	22.6	1.6	69.0	5.1
6	24.8	3.1	77.8	9.4	32.3	2.1	102.1	6.6
...	...	...	...	...	...	...	...	...
19	9.9	1.2	175.6	21.3	13.3	1.8	253.8	27.6
20	9.9	1.2	185.4	22.5	13.3	1.8	267.1	29.4

Table 9: Results showing how the phenotype height and width changes between iterations and from the initial starting size.

	SMCGP2		SMCGP
n	Avg	Std Dev	Avg
1	15,259	12,138	2,415
2	83,805	95,332	952,965
3	240,375	921,138	1,043,732
4	241,031	921,007	1,083,890
5	241,041	921,005	1,237,723
6	241,047	921,005	1,439,856
7	241,047	921,005	
8	241,047	921,005	

Table 10: Evaluations required to find an adder capable of adding two n-bit numbers. Apart from the 1-bit adder, SMCGP2 outperforms SMCGP [4].

9. FUTURE WORK

In future work, we will explore the capabilities of SMCGP2 further, and attempt to identify problems that have previously been impossible to solve using evolution and development. Another goal for future work is to explore problems that do not have such obvious regularity as digital circuits. Towards this goal, there are current results that use SMCGP2 to further previous work on finding general solutions to mathematical problems [5]. Here, the improved human-readability is of particular help.

We expect SMCGP2 to be at least as expressive as SMCGP, and as can be seen by previous work with SMCGP, the technique should be capable of solving a wide range of different problems. The ability for SMCGP, and by extension SMCGP2, to revert to a non-developmental GP is also a useful property of the representation.

10. ACKNOWLEDGMENTS

WB acknowledges funding from Atlantic Canada's HPC network ACENET and by NSERC under the Discovery Grant Program RGPIN 283304-07.

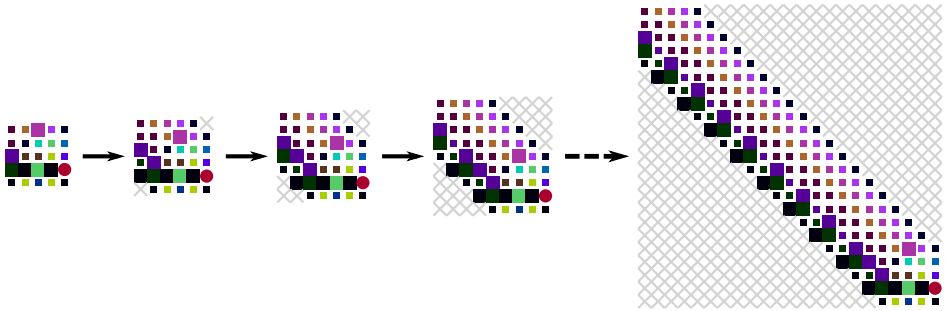

Figure 2: An example of a parity circuit generating program in SMCGP2. Each sub-image shows the phenotype at a different stage in the development. The left most program solves for 2 input parity. The next programs solve for 3,4 and 5 inputs. The right-most program shows the phenotype solving for 12 inputs. Although there are obvious regularities, it is interesting to note that the behavior of the active nodes on the left differs throughout the developmental stages.

11. REFERENCES

[1] 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.

[2] S. Harding, J. F. Miller, and W. Banzhaf. Self-modifying cartesian genetic programming. In H. Lipson, editor, *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, pages 1021–1028. ACM, 2007.

[3] S. Harding, J. F. Miller, and W. Banzhaf. Self modifying cartesian genetic programming: Parity. In A. Tyrrell, editor, *2009 IEEE Congress on Evolutionary Computation*, pages 285–292, Trondheim, Norway, 18-21 May 2009. IEEE Computational Intelligence Society, IEEE Press.

[4] S. Harding, J. F. Miller, and W. Banzhaf. Developments in cartesian genetic programming: Self-modifying CGP. *Genetic Programming and Evolvable Machines*, 11:397–439, 2010.

[5] S. Harding, J. F. Miller, and W. Banzhaf. SMCGP2: Finding Algorithms That Approximate Numerical Constants Using Quaternions and Complex Numbers. *Accepted for publication in GECCO 2011*, 2011.

[6] T.-H. Hoang, R. McKay, D. Essam, and X. H. Nguyen. Developmental evaluation in genetic programming: A position paper. *Frontiers in the Convergence of Bioscience and Information Technologies, 2007. FBIT 2007*, pages 773–778, Oct. 2007.

[7] L. Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In J. R. Koza, W. Banzhaf, and et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 158–166, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[8] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference (GECCO)*, pages 1135–1142, Orlando, Florida, 1999. Morgan Kaufmann.

[9] J. F. Miller and S. L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computing*, 10:167–174, 2006.

[10] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli and et. al, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[11] 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.

[12] R. Poli and J. Page. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code gp and demes. *Genetic Programming and Evolvable Machines*, 1(1-2):37–56, 2000.

[13] J. A. Walker and J. F. Miller. Automatic acquisition, evolution and re-use of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12:397–417, 2008.

[14] M. L. Wong and K. S. Leung. Evolving recursive functions for the even-parity problem using genetic programming. In P. J. Angeline and K. E. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[15] M. L. Wong and T. Mun. Evolving recursive programs by using adaptive grammar based genetic programming. *Genetic Programming and Evolvable Machines*, 6(4):421–455, 2005.