Genet Program Evolvable Mach (2010) 11:397–439
DOI 10.1007/s10710-010-9114-1

CONTRIBUTED ARTICLE

Developments in Cartesian Genetic Programming: self-modifying CGP

Simon Harding · Julian F. Miller · Wolfgang Banzhaf

Received: 27 November 2009 / Revised: 26 April 2010 / Published online: 25 June 2010

Abstract Self-modifying Cartesian Genetic Programming (SMCGP) is a general purpose, graph-based, developmental form of Genetic Programming founded on Cartesian Genetic Programming. In addition to the usual computational functions, it includes functions that can modify the program encoded in the genotype. This means that programs can be iterated to produce an infinite sequence of programs (phenotypes) from a single evolved genotype. It also allows programs to acquire more inputs and produce more outputs during this iteration. We discuss how SMCGP can be used and the results obtained in several different problem domains, including digital circuits, generation of patterns and sequences, and mathematical problems. We find that SMCGP can efficiently solve all the problems studied. In addition, we prove mathematically that evolved programs can provide general solutions to a number of problems: n-input even-parity, n-input adder, and sequence approximation to π.

Keywords Cartesian Genetic Programming · Developmental systems

S. Harding (✉) · W. Banzhaf
Department of Computer Science, Memorial University of Newfoundland,
St. John's, NL A1B3X5, Canada
e-mail: simonh@mun.ca

W. Banzhaf
e-mail: banzhaf@mun.ca

J. F. Miller
Department of Electronics, University of York, York YO10 5DD, UK
e-mail: jfm7@ohm.york.ac.uk

1 Introduction

In genetic programming (GP), representations of programs (genotypes) are evolved that solve computational problems. However, in such approaches the programs are almost always static and do not change over time, or in response to environmental inputs. In this paper, we propose a form of genetic programming in which the programs can, over time, change, acquire new inputs or produce new outputs. The work has been influenced by ideas expressed in the field of generative and developmental systems, where researchers evolve genotypes that can lead to arbitrarily large phenotypes with time dependent behaviour. Many such approaches utilize the concept of a cell in which a *fixed* genotype resides. The phenotypes arise through cell replication. The multicellular structure is finally mapped to an appropriate computation in the application domain. We wished to be able to evolve genotypes that could be iterated to arbitrarily sized phenotypes but in such a way that the phenotypes were always interpretable as a program.

In biology, the phenotypes arise from genotypes through a complex interaction in which a genotype, along with the cellular machinery and the environment gives rise to a stage of the phenotype, which itself influences the decoding of the phenotype for subsequent stages [5]. Regulatory mechanisms have been found to play a key role in this transformation. In the approach described in this paper abstractions of "regulatory" mechanisms determine whether and how self-modification operations will be applied. In computer science, this "unrolling" of the phenotype is often restricted and considered analogous to self-modification or re-writing. However, in genetic programming this notion has received only a little attention (see Sect. 2).

In the method we describe, a genotype decodes to a potentially infinite sequence of phenotypes (which themselves are programs). Such an approach has a number of advantages, not least, that a genotype may represent a solution to an entire class of problems. For instance, we show later that genotypes can be evolved which encode programs that can build parity functions with an arbitrary number of inputs, and others that can exactly compute π in the limit of large iterations.

To accomplish this we have introduced extra functions into a GP function set that cause modifications to the executed code itself. The technique we discuss is based on Cartesian Genetic Programming (CGP), so we refer to it as self-modifying CGP (SMCGP). The representation used in SMCGP is very flexible. For instance, a genotype that has no SM functions is essentially identical to standard CGP. On the other hand it is possible for SM functions to arise that cause the entire duplication of the genotype. This could be seen as a kind of multi-cellular development, albeit without the notion of a Euclidean space in which cells have to position themselves and occupy space.

We chose to base our approach on GP, as opposed to neural networks or other representations, as it allows the technique to be used in many different applications requiring algorithms, e.g., mathematical regression or circuit design. The approach we have taken is very general and in principle could encompass many types of developmental computational systems. We also desired to devise a representation that can work in many different domains without greatly modifying the basic working principle. This way we are able to test our approach on a wide variety of problems that appear in the literature.

Another of our motivations, particularly when contrasted with developmental systems, is that we wanted to be able to understand the evolved program. This is especially important when demonstrating the generality of a solution. For practical reasons, the developmental process needs to be computationally cheap. We feel that such goals largely rule out devising a developmental GP method that emulates too many aspects of biological systems.

SMCGP allows us not only to solve problems that cannot be solved using GP or CGP but it allows us to find general solutions to some classes of problems. In particular, we are able to find general solutions to problems that previously had only been solved through a combination of CGP and human inspection. For instance, in the first edition of the Genetic Programming and Evolvable Machines Journal, Miller et al. [40] investigated the “digital adder problem” and showed a series of evolved designs for adders that could by human inspection be generalized to produce a design capable of adding *n*-bit binary numbers. In Sect. 5.4 we show how SMCGP can obtain a general solution automatically.

The plan of the paper is as follows: We review the origins of self-modification in computer science in Sect. 2. In Sect. 3, we review re-writing or developmental methods, particularly those that have compared the evolution of developmental genotypes with direct representations. We explain how SMCGP works in Sect. 4, showing in detail how we define and use SM functions both in a self-modifying and computational sense. We also describe how inputs and outputs are handled. The evolutionary algorithm used and its operators and parameters are also discussed in that section. We have devised and used a variety of different fitness function types and primitives sets in the many experiments we have undertaken. In Sect. 4.9 we show which function sets have been used for the various problems studied. The experiments and results are described in Sects. 5, 6 and 7.

2 Review of self-modification

In this section we briefly review the existing body of work in the area of self-modification. The first distinction we are going to make for our purposes is the distinction between self-modified code of computers and all other kinds of self-modification, like self-modification of organismic behaviour through learning [50] or self-modification of brains through changes to their wiring pattern [11], or even machines [2]. In fact, ACM/IEEE Computer Society list, under their keywords for classification in the class Theory of Computation under point 6.1.1. “Self-modifying machines” [23]. Thus, the term has a prominent place in computing, which has to do with its history.

Early on in the design of electronic computers, it was recognized that the distinction between data and programs really was an artificial one for the purpose of information storage. This led von Neumann and others to the conclusion that one should store programs and data in the same type of devices. Preceding this development was the recognition of Turing, embodied in the famous Turing machine that a machine could not only modify data stored on a tape, but, if that tape contained its program, start to modify its own program. Before Turing, Gödel had realized with his numbering system a similar idea that would ultimately lead him to the recognition of the incompleteness problem.

With von Neumann's computer architecture came the possibility of manipulating program code in the same way as one would manipulate data, and it was quickly realized that one could write code that modified itself. In these early days of computing, this was not only a possibility, but an important feature due to limited resources available for storage. Self-modification allowed more compact code, if less understandable one, but with some precautions, it was possible to use the technique to optimize space requirements in memory.

One of the main uses of self-modifying code until today has been the runtime generation of code and of code compression [4, 27]. Both of these applications allow better use of computing resources, and therefore allow run-time or memory usage improvements over their non-modified counterparts. In the compression domain, other aspects like security from reverse engineering also play a role.

Another line of reasoning for the usefulness of self-modification starts with adaptation. In the context of Virtual Machines used heavily today, this has taken hold [3]. Here, the question is again how to make intelligent use of resources.

It is only a small step from here to the idea of evolution of self-modifying models of computation, as it was proposed repeatedly in the last decades for automata [49], for hardware (FPGAs) [12] or computer code [44]. The latter development took place within the field of Genetic Programming [31] which demonstrated the evolvability of computer algorithms and paved the way for an entire new field of algorithm development. Spector and Stoffel explicitly used self-modification in a GP approach called "ontogenetic programming" [54]. Spector's later developed the GP language "Push" so that "autoconstructive evolution" could take place [53]. This is where evolved genotypes are responsible for the production of their own offspring, rather than it being coded into an evolutionary algorithm explicitly. Push allows evolved code to manipulate itself. So that programs could, in principle have "morphological" phases during which they develop into "mature" code which is then executed to solve a problem. Alternatively, such programs might continue to develop as they run, exhibiting "ontogeny" more in the manner of living organisms. Self-modification was also implicit in the graph re-writing system proposed by Gruau [14], which will be described in the next section. Miller [43] also considered a form of self-modification in his developmental method of evolving graphs and circuits. McPhee [38] used an N-gram based GP system to produce programs that could solve regression problems, where development was linked to an incremental fitness function.

In a series of contributions and works, Kampis [24, 25, 26] pointed out that self-modification is extremely important for living organisms, and indeed might constitute their defining properties. Already prior to Kampis' work, Maturana and Varela [36] had proposed the concept of autopoiesis as a key feature of living systems. Self-modification has also been considered in artificial organisms as they were examined in the field of Artificial Life. Major contributions were made by the introduction of Coreworld [46] and the TIERRA [47] and its variants [1]. In this line of work, the emphasis is, however, more on the observation of emergent effects in self-modifying systems than on their use in computation.[1]

Since we are interested here in moving toward more life-like behaviour of algorithms, that exhibit adaptivity and efficiency, self-modification is taken to be one of the key properties we want to include in our system.

3 Developmental genotypes versus direct mappings

Recently there has been an increasing interest in generative and developmental systems (GDS) [34] and their potential benefits for evolutionary computation. Many argue that GDS will be necessary in order to make evolutionary techniques scale up to larger problems (see, e.g., [6]).

Kitano used a developmental method to define the architecture of an artificial neural network (ANN). The technique used a matrix re-writing system that manipulated adjacency matrices [29]. He claimed that he could evolve better ANNs more quickly using the developmental approach than by direct methods, i.e., a fixed architecture, directly encoded and evolved. However, a later paper by Siddiqi and Lucas [52] made a more detailed study and concluded that the two approaches were of equal quality.

Gruau also investigated an evolutionary developmental approach for ANNs. He devised a graph re-writing method called cellular encoding [14] for local graph transformations that control the division of cells growing into an artificial neural network. Connection strengths (weights), threshold values and the grammar tree that defines the graph re-writing rules were evolved using an evolutionary algorithm. This method was shown to be effective at optimizing both the architecture and weights at the same time, and scaled better than a direct encoding [15].

Bentley and Kumar [8] looked at a number of genotype to phenotype mappings on a problem of creating a tessellating tile pattern. They found that what they termed "implicit" developmental mapping could evolve tiling patterns much quicker than a variety of other representations (including direct) and further, that they could be subsequently grown (iterated) to much larger sized patterns.

Hornby and Pollack [20] evolved context-free L-systems to define three dimensional objects (table designs). They found that their generative system could produce fitter designs faster than direct methods. The generative approach produced more structures with more regularity and symmetry than direct methods.

Eggenberger investigated evolving developmental and non-developmental genetic representations on the difficult problem of optical lens design [21]. He found that the direct method scaled very badly when compared with the developmental approach.

Roggen and Federici [48] compared evolving direct and developmental mappings for the task of producing specific two dimensional patterns of various sizes (the Norwegian Flag and a pattern produced by Wolfram 1D CA rule 90). They showed

[1] With the exception of recent work in AVIDA, which takes a more utilitarian approach [37].

in both cases that as the pixel dimensions of the patterns increased the developmental methods out performed the direct.

Gordon [13] showed that evolved developmental representations were more scalable than direct representations for digital adders and parity functions.

Sekanina and Bidlo [51] showed how a developmental approach could be evolved to design arbitrarily large sorting networks. Kicinger [28] investigated the problem of design in steel structures for tall buildings and found that CA-based generative models produced better results quicker than direct representations and that the solutions were more compact.

Clune et al. [10] investigated the use of an indirect mapping to encode weights in an ANN and compared with a direct ANN mapping for leg control in simulated quadruped robots. They found the indirect mapping evolved faster to produce better robot locomotion and it also produced much more regular gaits. The indirect mapping used to encode ANN weights is called compositional pattern producing networks (CPPNs). It generates neural weights between neurons in planar arrays by evolving a mapping from the Cartesian coordinates of the two neurons to a weight [55]. Stanley refers to such a mapping as a “novel abstraction of natural development”. However, unlike biological development the technique does not involve time or iteration.

Clearly, the evidence is growing that GDS representations may be more scalable than more direct genotype representations. Despite this, in general it is still not clear how and whether developmental representations have advantages in a more general computational sense. This is because, firstly, investigations have concentrated on particular systems such as neural networks, structural design, digital circuits or sorting networks. Secondly, in some cases the demonstrations of greater scalability of GDS are questionable, since authors, sometimes by their own admission, have chosen rather naive direct representations in comparison with developmental representations.

Arguably, there has been little focus on actual computation in GDS. Instead, much work has concentrated on pattern formation. This requires a mapping stage where the abstract pattern is mapped to a program, design or circuit. Since the SMCGP approach is computational from the outset, a mapping from a phenotype to a computation is already provided. The unified representation of SMCGP includes both developmental and non-developmental functions. As a result, comparisons of the computational efficiencies of explicitly non-developmental (CGP) and developmental mappings (SMCGP) are more meaningful in such a setting.

4 Self-modifying CGP

4.1 Overview

SMCGP has three distinct aspects: the underlying genotype representation, the evolutionary algorithm and the developmental process.

Algorithm 1 gives a high-level overview of the process of mapping a genotype to a phenotype through a process of development. The first stage of the mapping is the modification of the genotype. This happens through the use of evolutionary operators acting on the genotype. The developmental steps in the mapping are outlined in lines 3–8 if the algorithm. The first step is to make an exact copy of the genotype and call it the phenotype at iteration 0. After this the self-modification operators are applied to produce the phenotype at the next iteration. Development stops when either a predefined iteration limit is achieved or it turns out that the phenotype has no self-modifications operations that are active. It is important to note that there are various ways in which there may be no active self-modification operations. Firstly, no self-modification operations may exist in the phenotype. Secondly, self-modification operations are present but they are non-coding. Thirdly the self-modification operations may not be "triggered" when the instructions encoded in the phenotype are executed. These various conditions will be discussed in the detailed description in the following sections.

Algorithm 1 Overview of genotype, phenotype and development

```
1: Generate genotype
2: Copy genotype to phenotype. Iteration, $i = 0$
3: repeat
4:     Apply self-modification operations to phenotype $i$
5:     increment $i$
6:     Calculate fitness increment, $f_i$
7: until ($i$ equals number of iterations required) OR (No self-modification functions to do)
8: Evaluate phenotype fitness $F$ from fitness increments, $f_i$
```

The representation of the genotype is described in detail in Sect. 4.3. It is based on Cartesian Genetic Programming but includes a number of new features that support the self-modification operators. The evolutionary algorithm that operates on this representation is simple evolutionary strategy, and is described in Sect. 4.4.

4.2 Cartesian Genetic Programming (CGP)

The term "Cartesian Genetic Programming" (CGP) first arose in a paper 11 years ago on the evolution of digital circuits published in the first conference on "Genetic and Evolutionary Computation" [39]. The following year, in the first edition of Genetic Programming and Evolvable Machines, Miller et al. [40] examined how CGP could be used to find novel digital circuits and how general digital design principles could be deduced. Also in 2000 the method was proposed as a new and complete method of genetic programming [42].

CGP represents programs as directed graphs. One of the benefits of this type of representation is the implicit re-use of nodes that is characteristic of graphs. In CGP, the genotype is a fixed-length representation where each node in the directed graph represents a particular function and is encoded by a number of genes. One gene encodes the function that the node represents, and the remaining genes encode where in the graph the node obtains its inputs from. The nodes take their inputs in a feed-forward manner from either the output of nodes in a previous column or from a program input (terminal). Also, the number of inputs that a node has, is dictated by the arity of the function it represents. The program data inputs are given the *absolute* data addresses 0 to $n - 1$ where n is the number of program inputs. The data outputs of nodes in the genotype are given sequential addresses, column by column, starting from n to $n + m - 1$ where m is the user-determined upper bound of the number of nodes (equal to the number of rows multiplied by the number of columns). If the problem requires k program outputs, then k integers are added to the end of the genotype, each one being the *absolute* address of the output of a node where the program output is taken from. The two dimensional general form of a Cartesian Genetic Program is shown in Fig. 1.

CGP uses a genotype-phenotype mapping that does not require all of the nodes to be connected to each other. So the phenotypes can have a length from zero to the maximum number of nodes encoded in the genotype. Thus, areas of the genotype can be inactive and have no influence on the phenotype. This means that many genotypes decode to exactly the same phenotype and consequently their fitnesses are identical. This genetic redundancy (often referred to as neutrality) has been shown to highly beneficial to the evolutionary search of CGP genotypes [41, 42, 56, 64].

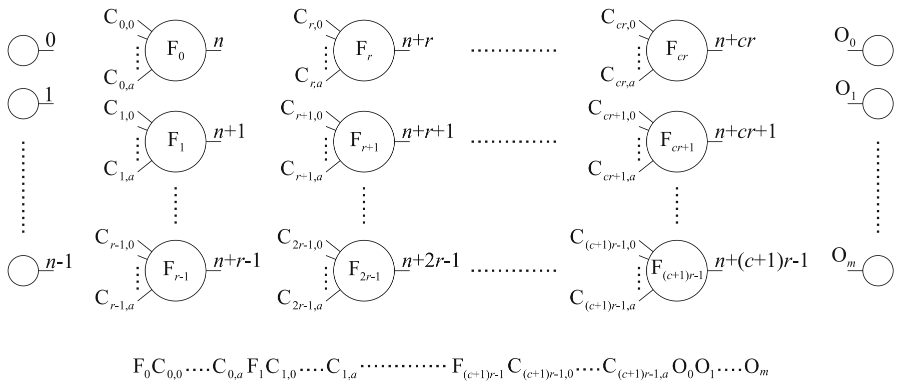
0
1
n-1
C0,0 F0 C0,a n
C1,0 F1 C1,a n+1
Cr-1,0 Fr-1 Cr-1,a n+r-1
Cr,0 Fr Cr,a n+r
Cr+1,0 Fr+1 Cr+1,a n+r+1
C2r-1,0 F2r-1 C2r-1,a n+2r-1
Ccr,0 Fcr Ccr,a n+cr
Ccr+1,0 Fcr+1 Ccr+1,a n+cr+1
C(c+1)r-1,0 F(c+1)r-1 C(c+1)r-1,a n+(c+1)r-1
O0
O1
Om
F0C0,0….C0,a F1C1,0….C1,a………… F(c+1)r-1 C(c+1)r-1,0….C(c+1)r-1,a O0O1….Om

Fig. 1 General form of two-dimensional CGP. It is a grid of nodes whose functions are chosen from a set of primitive functions. Two parameters c, and r, respectively, define the number of columns and rows in the grid. Each node is assumed to take as many inputs as the maximum function arity a. Every data input and node output are labeled consecutively (starting at 0) which gives it a unique data address which specifies where the input data or node output value can be accessed (shown in the figure on outputs of inputs and nodes). Nodes in the same column cannot be connected to each other. The graph is directed so that a node may only have its inputs connected to either input data or the output of a node in a previous column. In general there may be a number of output genes (O_i) which specify where the program outputs are taken from. The structure of the genotype is seen below the schematic. All node function genes F_i are integer addresses in a look-up table of functions. All connection genes C_{ij} are *absolute* data addresses and are integers taking values between 0 and the address of the node at the bottom of the previous column of nodes

The original form of CGP did not include Automatically Defined Functions [32]. However, later Walker and Miller [57, 58], utilizing ideas from a technique known as module acquisition, showed how sub-functions could be evolved and re-used in CGP. Finally, it was shown that CGP is, under certain conditions, equivalent to a particular form of linear GP [59].

4.3 The SMCGP representation

SMCGP's representation, though similar to CGP, has some important differences. SMCGP genotypes represent a linear string of nodes. That is to say, only one row of nodes are used (in contrast to CGP which can have a rectangular grid of nodes). Another important difference is how SMCGP represents connection genes. In CGP, connection genes are *absolute* addresses, indicating where the data supplied to a node is to be obtained, whereas SMCGP uses *relative* addressing. Each node obtains its data via its connection genes by counting back from its position in the graph. As in CGP, to prevent cycles, nodes can only connect to previous nodes (on their left). The relative addressing allows sections of the graph to be easily moved, duplicated, or deleted without breaking constraints of the directed graphical structure. Self-modification also require extra genes that are used to identify parts or characteristics of the graph that will be changed.

Another change from CGP, and previously published work on SMCGP is the way SMCGP handles inputs and outputs. Terminals are acquired through special functions (called INP, INPP, SKIPINP) and program outputs can be 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.

To summarize, 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 three floating point number arguments used by self-modification functions.

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

An example genotype is given in Fig. 2. The figure also shows purely schematically some phenotypes arising at different iterations.

The actual number of inputs of a node is dictated by the arity of its function, and in this paper there is a maximum of two inputs. If the connection gene specifies a distance of 1 it will connect to the previous node in the list, if the gene has value 2 then the node connects 2 nodes back and so on. All the relative distances are generated so that they are greater than zero, to avoid nodes referring directly or indirectly to themselves.

If a gene specifies a connection pointing outside the graph, i.e., with a larger relative address than there are nodes to connect to, then this is treated as connecting to a null input. Terminals and outputs themselves are obtained by using special

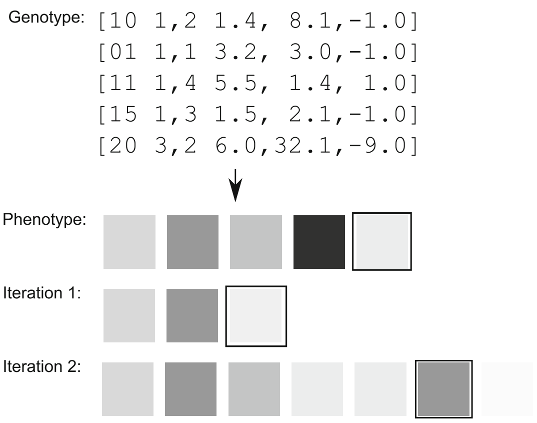

Fig. 2 The genotype maps directly to the initial graph of the phenotype. The genes control the number, type and connectivity of each of the nodes. The phenotype graph is then iterated to perform computation and produce subsequent graphs. The nodes in the phenotype that are acting as outputs are outlined

functions. This is described in detail in Sect. 4. This encoding is shown visually in Sect. 4.7.

The relative addressing used allows, sub-graphs to be placed or duplicated in the graph whilst retaining their semantic validity. This means that sub-graphs could represent the same sub-function, but acting on different inputs. This can be done without recalculating any node addresses thus maintaining validity of the whole graph. So sub-graphs can be used as functions in the sense of ADFs in standard GP (or modules in CGP [58]).

The three floating point arguments are used as arguments for the self-modification functions, or for functions that return constant values. It is important to note that depending on the function using them, the value may be truncated to an integer. Section 4.8 details the available functions and their associated arguments.

Functions that are not explicitly computational (i.e., SM functions and output functions) pass on the computations presented to them. This is discussed in Sect. 4.5.

4.4 Evolutionary algorithm

In CGP, a (1 + 4) evolutionary strategy is often used. We do the same in this paper. However, we begin the process by testing a population of 50 random individuals. This helps to boot-strap the evolutionary algorithm and increases the chance of obtaining a viable individual from which to build from. We then select the best individual and generate four offspring by mutation. We test these new individuals, and use the best of these to generate the next population (and if there are two or more equally best, we pick the newer).

In the experiments in this paper, 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. Similarly, when functions are mutated (or during the initial generation of individuals), all functions in the set of functions chosen for a particular experiment, have the same probability of being selected. We have yet to investigate the effect of biasing factors such as the ratio of self-modifying to normal functions.

For the arguments, the mutation operator can choose to randomize the value (with probability 0.1), or with probability 0.9 add noise (normally distributed, with a standard deviation of 20).

The argument values have not been optimized and are based on our experiences with CGP. We would expect performance increases if more suitable values were used.

4.5 Evaluation of the SMCGP graph

The phenotype is executed in the same manner as standard CGP, so that the computational output of the graph is obtained by recursion, starting from the output nodes down through the functions, to the input nodes. In this way, nodes that are unconnected are not processed and do not effect the behavior of the graph at that stage.

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

For graph manipulation functions (self-modifying), the graph is parsed from left to right. The input values to nodes are found and the behavior of the node is based on these input values. If the first input is greater or equal to the second, then the graph manipulation function is added to a "To Do" list of pending modifications and the node returns the first input. If a graph manipulation function is not added to the "To Do" list it returns the value of its second input. This means that the programs self-modifying behaviour is dependent on the particular data passing through the graph. For Boolean functions, we add the operation to the "To Do" regardless of the inputs.

The length of the list is usually limited as manipulations are relatively computationally expensive to perform. In this paper we have limited the length to just two instructions, unless stated to the contrary.

All graph manipulation functions require a number of arguments (evolved), that determine how they operate on the graph. These are described in Sect. 4.8.

4.6 Inputs and outputs

In the classical implementation of CGP, inputs are defined as *absolute* addresses that nodes can connect to. It was ensured that all node connection genes were always a valid address. In the early versions of SMCGP [16], with the addition of relative addressing, measures needed to be taken to deal with situations where node connection addresses did not refer to any nodes on the graph (i.e., when addresses went negative). To ensure that this could not happen the addresses were taken modulo the number of inputs. This ensured that such connections always obtained a valid program input. In this way, as the graphs grew, the addresses could reach more and more inputs without having to have the connections explicitly encoded (as would be the case in CGP).

We found, however, that inputs still did not scale particularly well with problem size, so in subsequent papers [17, 18, 19] we examined another strategy: Three special input functions are now added to the function set: INP, INPP and SKIPINP. When decoding the phenotype graph, a pointer is maintained that refers to an input. If the first occurrence of an active input function is INP it returns the first program input. If the first occurrence is INPP, the last program input is returned. After INP is called it increments the input pointer. INPP decrements the input pointer. SKIPINP allows the pointer to jump more than one input, and returns the current input before moving the pointer by an amount given by the first argument of SKIPINP, P_0. This is truncated to an integer and decides whether to increment or decrement the input pointer according to the sign of the argument. When the pointer is asked to move beyond the last (or first) input, it simply wraps around to the other side of the input list. This ensures that there is always an input available to be read.[2]

Also in earlier work we included an extra binary gene with every node which flagged whether the node could provide a program output [17, 18, 19]. However, in the work for this paper we have removed output genes and instead introduced a primitive *function* called OUTPUT that provides a program output. This was partly done because we thought, like the introduction of input functions, it would improve the ability of the approach to scale to larger problems with different numbers of outputs and also to make the input-output mechanisms more consistent.

A number of measures need to be taken when the number of OUTPUT nodes does not match the number of program outputs.

– 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.

4.7 Examples

The figures used throughout this paper are generated automatically by the SMCGP program. When reading the graphs there are several important things to note:

– Each function has a different colour (or shade of grey), however, the same colour may be used differently in different function sets.

[2] In this model, if a node wishes to connect to a negative address, a default value is returned. For binary problems this value is FALSE, for numeric problems the default value is 0. INP, INPP and SKIPINP are all terminal functions with no inputs. So connection genes are ignored

- The pictured graphs cannot be converted back into phenotypes. For clarity many details have been omitted (such as the values for parameters).
- Unconnected nodes are drawn with a smaller square and without their function type as a label.
- In figures with more than one phenotype, each graph represents one iteration.

Figure 3 shows a phenotype with one output node and two INP (input) nodes. The output of the program will be the Binary XOR (BXOR) of the two input nodes. INP returns the next available input. If the program only had one input, both of the INP nodes would return the same value. If there are two inputs, the INP node on the left will return the first input, the next INP node will return the second input. If there were more than two inputs, the additional inputs would be ignored.

Figure 4 introduces the INPP node. Suppose that the program has two inputs x_0 and x_1. The leftmost INP function returns x_0, the next INP function returns x_1. The INPP returns x_1 also, this is because the second INP function would have left the input pointer pointing to x_0 (due to the list of inputs being a circular list). Thus the first BXOR function returns $x_0 \oplus x_1$ and the second BXOR function returns $x_1 \oplus x_0 \oplus x_1$. Since the OUTPUT function is directly connected to this it returns the same.

Figure 5 shows a simple phenotype with two outputs. The first output value is equal to the first input. The second output is the exclusive-OR of the first two input values.

Figure 6 also shows a phenotype with two outputs (the nodes used as outputs are drawn with a box around them). However, here there is only one OUTPUT node in the phenotype. The SMCGP interpreter attempts to find other nodes to use as outputs. Here the next node has been selected to be used as an output.

Figure 7 demonstrates the use of the DUP (duplicate) operator. The duplication operator here duplicates a section of the graph made of three nodes BXOR, INP, and BXOR. It inserts them next to the DUP node. The source of these three nodes is the BXOR node used in the first iteration and it's neighbouring two nodes to the right. This demonstrates two things. The first is how the duplication operator can make

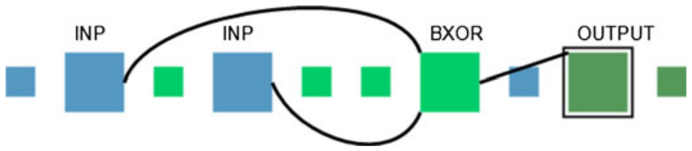

Fig. 3 Example showing the a simple phenotype with 1 output and 2 input nodes

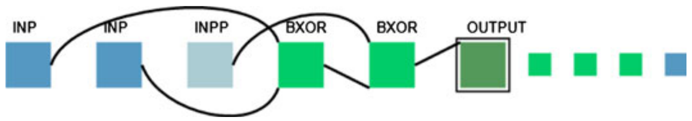

Fig. 4 Example showing the a simple phenotype with 1 output and 3 input nodes

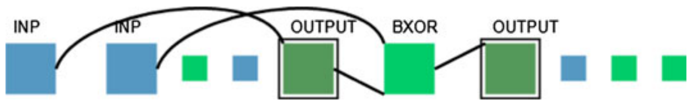

Fig. 5 Example showing multiple outputs

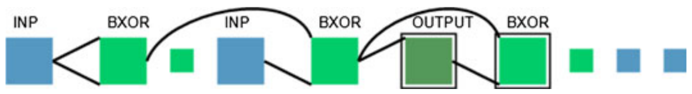

Fig. 6 Example showing a phenotype with multiple outputs, but only one OUTPUT node

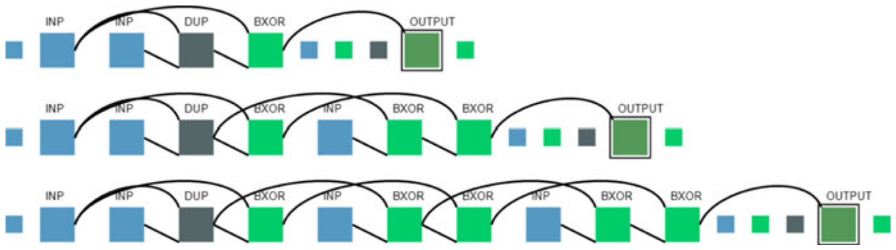

Fig. 7 Example showing the use of DUP (duplicate operator)

programs grow by inserting copies of other parts of the phenotype into the next iteration. The other observation is that SMCGP can move nodes that were previously unconnected and connect them to form part of the program.

Phenotypes in SMCGP can also shrink. Figure 8 shows an example of the DEL (deletion) operator in use. In the first iteration, the program uses three inputs. In the next iteration, the DEL node has removed the first node. The left most BXOR no longer connects to an input, but instead receives copies of the “default” value. The INP (input nodes) in the first iteration use inputs 0, 1 and 2. In the next iteration the nodes would use inputs 0 and 1. In a further iteration, the first node (BXOR) is removed. Each time the DEL occurs the program’s functionality is also changed.

4.8 Self-modification functions

The way self-modifying functions act is defined by four variables. Three of them are the argument genes that are double precision numbers. We denote them P_0, P_1, and P_2. The other variable is the integer position in the phenotype of the self-modifying node (i.e., the leftmost node is position 0). We denote this 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 arguments are treated:

– When P_i are added to the x, the result is treated as an integer.

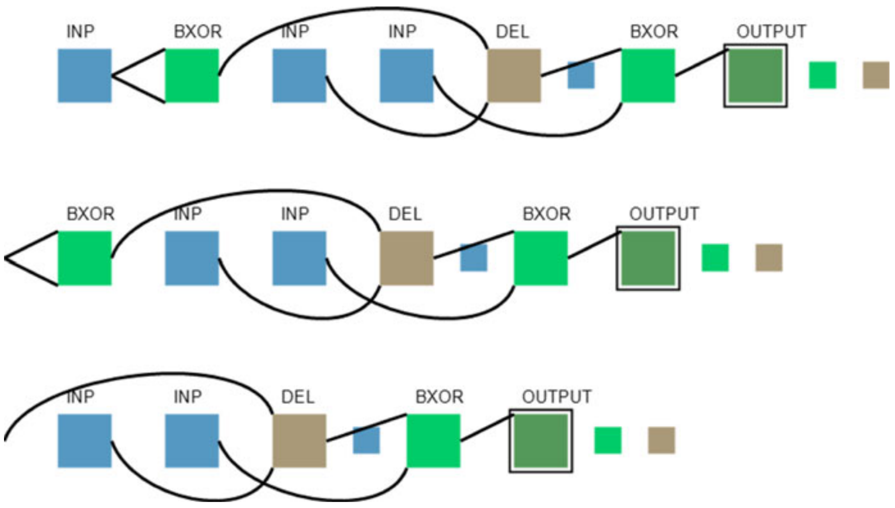

Fig. 8 Example showing the use of DEL (deletion operator)

– 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 "To Do" list length.

The exact rules obeyed by various graph manipulation functions are shown in Table 1.

The list of self-modification functions in Table 1 is large and some are quite complex, however, it is at this stage unclear what the minimal useful set of self-modifications should be. To some extent this question may only be answerable through extensive experimentation using evolution.

4.9 Function sets for experiments

There are a number of different function sets used in these experiments, so they have been grouped into sets as in Table 5. Table 6 contains the set of functions in each of these groups. The choice of function sets is determined by the problem type, and by any previous work that we wish to compare with. Tables 2, 3 and 4 detail the various individual functions.

5 Experiments: digital circuits

5.1 Fitness function for parity and adder

In this section we describe how to use SMCGP to evolve programs that generate adder and parity circuits having an arbitrary number of inputs. The aim is to evolve

Table 1 Definition of the self-modification functions

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 (DU2)	Copy the nodes between (P_0) and $(P_0 + P_1)$ and insert after $(P_0 + P_2)$
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 (SHIFTCONNECTION)	Starting at node index $(P_0 + x)$, add P_2 to the values of the c_{ij} of next P_1 nodes
Shift connections 2 (MULTCONNECTION)	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 "To Do" list

P_i are the evolved arguments of the self-modification functions. x represents the absolute position of the node in the graph, where the leftmost node has position 0. c_{ij} is the jth connection gene on node at position i

a program that on each iteration, produces the next larger circuit by taking more inputs and performing the appropriate function (even-parity or bitwise addition).

Even parity circuits consist of n inputs, and a single output that is true when there are an even number of True bits in the input. For the adder, the circuits take two n-bit, binary encoded integers and return one $n + 1$-bit number that is the numerical sum of the two inputs.

Digital circuits have often been studied in genetic programming [30, 32], and some systems have been used to produce "general" solutions [22, 60, 61, 63]. A general solution 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.

Table 2 Binary functions

Function	Operation
BAND	a AND b
BOR	a OR b
BNAND	NOT (a AND b)
BXOR	a XOR B
BNOR	NOT (a OR b)
BNOT	NOT a
BIAND	(NOT a) AND b
BF0	FALSE
BF1	(a AND b)
BF2	a AND (NOT b)
BF3	(a AND (NOT b)) or (a AND b)
BF4	(NOT a) AND b
BF5	((NOT a) AND b) OR (a AND b)
BF6	((NOT a) AND b) OR (a AND (NOT b))
BF7	((NOT a) AND b) OR (a AND NOT(b)) OR (a AND b)
BF8	((NOT a) AND (NOT b))
BF9	((NOT a) AND (NOT b)) OR (a AND b)
BF10	((NOT a) AND (NOT b)) OR (a AND NOT (b))
BF11	((NOT a) AND (NOT b) OR a AND (NOT b) OR a AND b)
BF12	((NOT a) AND (NOT b) OR (NOT a) AND b)
BF13	((NOT a) AND (NOT b) OR (NOT a) AND b OR a AND b)
BF14	((NOT a) AND (NOT b) OR (NOT a) AND b OR a AND (NOT b))
BF15	((NOT a) AND (NOT b) OR (NOT a) AND b OR a AND (NOT b) OR a AND b)

In the case of parity, to begin with, we evolve for two input bits. When a successful solution is found, the fitness function requires that the program produces a two bit circuit, followed by a three bit circuit. Then when a genotype has been found that solves the two bit problem and on iteration solves the three bit problem, the fitness function changes so that now in addition to the previous behaviour the genotype, when iterated twice, produces a phenotype that solves the four-bit problem. The process continues in this way until we obtain a phenotype that correctly implements the required function with 20 inputs. We refer to the application of each parity or adder as a test case.

Fitness is computed as the number of correctly predicted bits over all test cases. If the candidate solution fails to find a totally correct solution for a given input size, it is not tested on other input sizes. We evolve for 19 test cases (2 inputs to 20 inputs).

The fitness function is designed to force the SMCGP to find a solution that grows through each test case to the next. In this way, the chance of a general solution is maximised. The fitness function pseudo code is shown in Algorithm 2.

Table 3 Mathematical functions

Function	Operation
NOP	No operation
DADD	a + b
DSUB	a − b
DMULT	a * b
DDIV	a / b
CONST	a constant, defined by P_0
AVG	(a + b) / 2
DSQRT	Square root of a
DRCP	1 / square root of a
DABS	Absolute value of a
TANH	tanh(a)
TANHNN	tanh(a + b)
FACT	Factorial
POW	a^b
COS	cosine(a)
SIN	sine(a)
MIN	min(a, b)
MAX	max(a, b)
IFLTE	If ($a < 0$) return b, else 0
INDX	Current node index
INCOUNT	Number of inputs

Table 4 Input and output functions

Function	Operation
INP	Return input pointed to by current_input, increment current_input
INPP	Return input pointed to by current_input, decrement current_input
SKIPINP	Return input pointed to by current_input, current_input = current_input + P_0
OUTPUT	Return data provided

P_0 is the first argument gene

Table 5 The function set used in each experiment

Experiment	Function set
Adder	A
Parity (full)	B
Parity (reduced)	C
Fibonacci	D
Pi (patterns)	E
Approximating Pi	F
Power regression	F
Classification	F
Squares	D
Sum of numbers	D

See Table 6 for the definition of each group

Algorithm 2 Fitness function

```
1: Fitness, $F = 0$
2: Copy genotype to phenotype. Iteration, $i = 0$
3: repeat
4:     $BREAK = FALSE$
5:     Apply self-modification operations to phenotype $i$
6:     increment $i$
7:     Calculate fitness on test case, $f_i$, by counting number of incorrect bits
8:     if $f_i \neq 0$ then
9:         $BREAK = TRUE$
10:    end if
11:    $F = F + f_i$
12: until $i = LIMIT$ OR $BREAK$
```

For the parity case *LIMIT* = 19, so the largest parity function tested has 20 inputs. However, we test the solutions for generality by testing to 24 bits. We chose 24 bits for two reasons. First, the largest evolved parity circuit we found in the literature was 22 bits [45]. It should be noted that Poli and Page used all 16 two input Boolean functions in their function set, whereas we just use AND, NAND, OR and NOR. Secondly, this is the largest circuit we can test within reasonable time and reasonable memory requirements.

Essentially, the fitness function used for evolving both types of circuit is the same, except that for the adder fitness function, the number of inputs *and* outputs grow each time. So the genotype should add two binary inputs, the phenotype at the first iteration, should add two two-bit binary numbers, and so on. The *LIMIT* = 6, but we tested evolved solutions further in order to check for generalization. Due to the demand on computational resources, we stop exhaustively evaluating the circuits at 10 bits (i.e., the addition of two ten bit numbers). For larger sizes, we sample the input space by testing with 10,000 random numbers having up to 1,000 inputs.

For the adder, function set A (shown in Table 6) is used. For the parity problem, two different function sets were compared. One contains all two-input Boolean functions, set B in Table 6 and the other which contains a reduced set of Boolean functions, set C. Both sets of functions have been used in work by other authors.

5.2 Parity results

Table 7 shows the average number of evaluations required to evolve for a given number of inputs. The success rate was 100%. Results are based on 50 trials per function set. Although, Poli and Page evolved solutions to even-parity up to 22

Table 6 Function sets per experiment

Function name	A	B	C	D	E	F
OUTPUT	X	X	X	X	X	X
INP	X	X	X	X	X	X
INPP	X	X	X		X	X
SM	X	X	X	X	X	X
NOP					X	X
DADD				X	X	X
DSUB					X	X
DMULT					X	X
DDIV					X	X
CONST					X	X
DSQRT					X	X
POW					X	X
COS					X	X
SIN					X	X
MIN					X	X
MAX					X	X
AVG					X	X
DRCP					X	X
DABS					X	X
TANH					X	X
LOG					X	X
LN					X	X
INCOUNT						X
BAND	X		X			
BNAND	X		X			
BXOR	X					
BNOR	X		X			
NOR	X		X			
BIAND	X					
BF0 to BF15	X	X				

inputs, they only gave numbers of evaluations for a single evolutionary run when the number of inputs was 13, 15, 17, 20 and 22 [45].

In Table 8, the results are compared to previous CGP representations and Koza's figures for GP with ADFs [32]. The SMCGP results are clearly highly competitive. We have included Koza's figures for reference. Koza calculated the computational effort for a 99% success rate and so represent the number of evaluations assuming the most favourable number of runs and numbers of generations. More detailed comparisons between CGP and other methods have been previously published in [58]. There it was seen that Embedded Cartesian Genetic Programming (ECGP) was highly competitive with other GP methods. It

Table 7 Average evaluations required to find a program that will solve parity up to a given number of bits (50 runs)

Input bits	Reduced	Full
3	247,753	37,276
4	275,663	41,697
5	278,635	43,016
6	298,104	43,593
7	318,376	150,719
8	322,843	150,721
9	322,843	150,722
10	322,843	150,722
11	322,851	150,722
12	322,851	150,722
13	322,866	150,722
14	322,866	150,722
15	322,866	150,722
16	322,866	150,722
17	322,870	150,722
18	322,870	150,722
19	322,874	150,722
20	322,874	150,722

Results are for both function sets used

Table 8 Comparison with previous work on evolving parity

Input bits	Reduced	Full	SMCGP 2007	CGP	ECGP	GP
4	275,663	41,697	28,811	81,728	65,296	176,000
5	278,635	43,016	58,194	293,572	181,920	464,000
6	298,104	43,593	199,256	972,420	287,764	1,344,000
7	318,376	150,719	410,128	3,499,532	311,940	1,440,000
8	322,843	150,721	1,080,656	10,949,256	540,224	–

GP is Koza's tree GP (with ADFs), ECGP is embedded CGP and CGP is conventional CGP. With the exception of the figures of Koza, the figures show average evaluations required to find a given sized parity circuit. Results for higher numbers of inputs are not available for CGP or ECGP. The figures for Koza represent computational effort so they represent a minimum number of evaluations required to achieve 99% success. The minimum is selected from the "ideal" number of runs and number of generations. The blank symbol indicates that figures are unavailable

should be noted that SMCGP is solving a different problem than CGP and ECGP. SMCGP were evolved to solve not just one instance of the parity problem, but a sequence of parity problems.

After evolution, solutions were tested for inputs of up to 24 bits. It was found that all solutions generalized to problems of this size. This is an improvement over our previous work, where it was found that 96% of solutions generalised to all tested problems.

For both sets of functions, most solutions apparently had generalized when even-5 parity is reached. Since, when the genotype can solve 3, 4 and 5 input problems it very often was able to solve all other tested parity sizes. This is reflected in the results in Table 7, where the number of evaluations required to solve a problem stops increasing, because once a solution is found to generalize, no further evolution is required.

Spector also examined the even-parity problem, however, he used a different function set [53]. He found better scaling behaviour than Koza on even-parity functions up to six inputs.

Other approaches have looked at finding general solutions. Huelsbergen [22] evolved machine language level programs that could iterate over the bits in a string and from this parity could be easily determined. The solutions would be suitable for any length bit string. Recursion has also been successfully used to solve the parity problem [61, 62, 63]. These approaches produced programs rather than circuits to solve the problem. They also used high level programming constructs rather than purely boolean logical primitives.

5.3 A general solution to parity

The genotype in Fig. 9 was evolved with a "To Do" list length of 1. The 20 node genotype only has 7 active nodes. The inactive nodes are shown as unconnected smaller squares. Nodes INPP at positions 0 and 2, obtain inputs x_1 and x_0, respectively. Three Boolean functions BNOR, BAND and BOR appear at positions 4, 5, and 6. The OUTPUT function obtains the single output from the BOR node. The only active SM node is DUP at position 1. It carries arguments which cause it to copy eight nodes beginning at the node on its left (INPP) and insert them immediately after itself. The action of the DUP node is shown using the curved line with an arrow emanating from the box. Since the genotype has no connections that are left of the first node, when DUP copies it disconnects the first two nodes in the generated phenotype. These appear at the beginning (left) of the new phenotype (iteration 1) at positions 0 and 1. It is important to note that in this phenotype a previously inactive node (BNAND) at position 9 becomes active.

We can see that the genotype computes even-2 parity as follows. Denote the outputs of node, i by z_i. Note $\oplus$ denotes the exclusive-OR operation. When two or more Boolean arguments are side by side (as if being multiplied), it is assumed that the Boolean AND (BAND) operation is applied to the arguments (e.g., xy is equivalent to BAND (x, y)). Overbar represents inversion.

$$\begin{aligned}
z_0 &= x_1 \\
z_1 &= x_1 \\
z_2 &= x_0 \\
z_4 &= \mathrm{BNOR}(z_2, z_1) = \overline{x_0 + x_1} = \overline{x_0}\,\overline{x_1} = (1 \oplus x_0)(1 \oplus x_1) \\
z_5 &= \mathrm{BAND}(z_1, z_2) = x_1 x_0 \\
z_6 &= \mathrm{BOR}(z_5, z_4) = z_5 + z_4 = z_5 \oplus z_4 \oplus z_4 z_5
\end{aligned} \tag{1}$$

Substituting for z_5 and z_4, expanding and then canceling terms we obtain

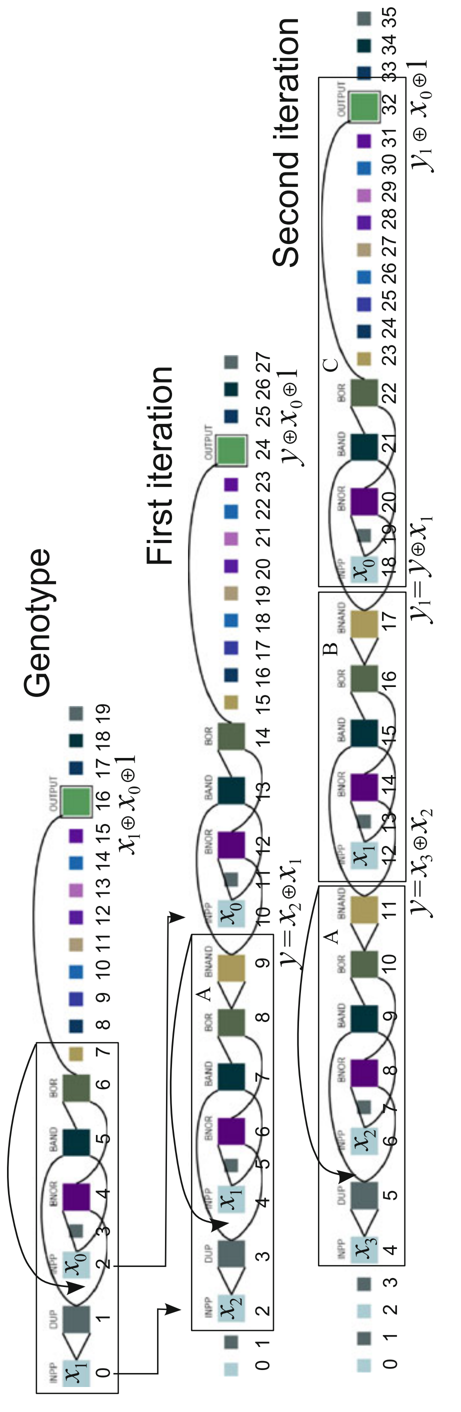

Fig. 9 An evolved genotype iterated twice that computes even-parity

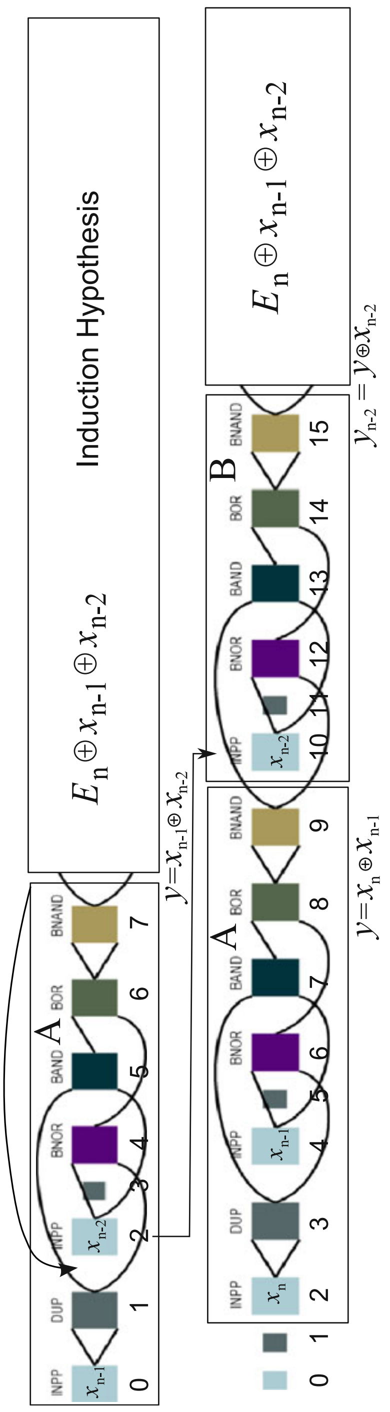

Fig. 10 The inductive hypothesis in diagrammatic form

Table 9 Evaluations required to evolve to each size

No. of bits in each pair	Average Evals	% Success
1	2,415	100.0
2	952,965	94.0
3	1,043,732	88.0
4	1,083,890	86.0
5	1,237,723	86.0
6	1,439,856	86.0

An n input adder adds two n-bit numbers

$$\begin{aligned} z_6 &= x_1x_0 \oplus (1 \oplus x_0)(1 \oplus x_1) \oplus x_1x_0(1 \oplus x_0)(1 \oplus x_1) \\ z_6 &= x_1x_0 \oplus 1 \oplus x_1 \oplus x_0 \oplus x_1x_0 \oplus x_1x_0(1 \oplus x_1 \oplus x_0 \oplus x_1x_0) \\ z_6 &= x_1 \oplus x_0 \oplus 1 \end{aligned} \tag{2}$$

Thus $z_{16} = z_6$ is the even-3 parity function. When the eight duplicated nodes are inserted into the genotype just before position 2 they cause the activation of the BNAND node at position 9 in the new phenotype. This inverts the function computed by the eight duplicated nodes in the genotype. So the output of this block of nodes (denoted A), is $x_2 \oplus x_1$ since the INPP functions return the inputs in descending order.

Now we turn out attention to the second iteration. When DUP inserts nodes 2 to 9 after itself, the nodes 4 to 9 are shifted right (becoming nodes 12–17) in the second iteration phenotype (enclosed in a box labeled B in the Fig. 9). We now prove that this set of nodes carries out the exclusive OR of its input (emanating from the NAND node, which we call y) with the input variable (in this case x_1) (Fig. 10).

$$\begin{aligned} z_{12} &= x_1 \\ z_{14} &= \mathrm{BNOR}(x_{12}, y) = \mathrm{BNOR}(x_1, y) = (1 \oplus x_1)(1 \oplus y) \\ z_{15} &= \mathrm{BAND}(y, z_{12}) = \mathrm{BAND}(y, x_1) = x_1y \\ z_{16} &= \mathrm{BOR}(z_{15}, z_{14}) = z_{15} + z_{14} = z_{15} \oplus z_{14} \oplus z_{14}z_{15} \\ z_{17} &= \mathrm{BNAND}(z_{16}, z16) = z_{16} \oplus 1 \end{aligned} \tag{3}$$

Substituting for z_{14} and z_{15} in z_{16} and then noting that in the last term when $x_1\ y$ multiplies $(1 \oplus x_1)$ we obtain $(x_1y \oplus x_1y)$ which is zero, thus we can simplify

$$\begin{aligned} z_{16} &= x_1y \oplus (1 \oplus x_1)(1 \oplus y) \oplus x_1y(1 \oplus x_1)(1 \oplus y) \\ z_{16} &= x_1y \oplus (1 \oplus x_1)(1 \oplus y) = x_1y \oplus 1 \oplus x_1 \oplus y \oplus x_1y \\ z_{16} &= 1 \oplus x_1 \oplus y \\ z_{17} &= x_1 \oplus y \end{aligned} \tag{4}$$

Since we have seen that the nodes in section C compute the odd parity of the supplied input y and the acquired input (by INPP) we find that at iteration two the phenotype computes $y_1 \oplus x_0 \oplus 1 = y \oplus x_1 \oplus x_0 \oplus 1 = x_3 \oplus x_2 \oplus x_1 \oplus x_0 \oplus 1$. This is the even-4 parity function. To construct a proof by induction we will assume that for n inputs the phenotype computes even-n parity.

The upper diagram shows the even-n parity function E_n. This is the inductive hypothesis. We have already seen that the function enclosed in box A produces at the next iteration the two disconnected nodes and the function in A, $y = x_n \oplus x_{n-1}$ followed immediately by the function in box B, $y_{n-2} = y \oplus x_{n-2}$, thus the new phenotype, E_{n+1} generates the function,

$$\begin{aligned} E_{n+1} &= y_{n-2} \oplus E_n \oplus x_{n-1} \oplus x_{n-2} \\ E_{n+1} &= y \oplus x_{n-2} \oplus E_n \oplus x_{n-1} \oplus x_{n-2} \\ E_{n+1} &= x_n \oplus x_{n-1} \oplus x_{n-2} \oplus E_n \oplus x_{n-1} \oplus x_{n-2} \\ E_{n+1} &= x_n \oplus E_n \end{aligned} \tag{5}$$

Thus the inductive hypothesis also applies for the $n + 1$th iteration. We have already seen that at iteration two, the form of the phenotype obeys the inductive hypothesis. Hence the general case is proved.

5.4 Adder results

Table 9 shows the average number of evaluations required to evolve a program that could grow to an adder of a given size (via intermediate sizes). We analysed when the adder solutions began to generalise (i.e., could solve up to 6 bits addition, even though they were evolved to solve a smaller problem).

After evolving to a 6 + 6 bit adder, the successful solutions were tested on larger problems. Adders were tested up to 1,000 + 1,000 bits, and remarkably, all were found to successfully generalise. Again, we had to sample the input space for larger problem sizes. We took 1,000 different input patterns for every input size from 10 to 1000 bits.

5.5 A general solution to n-bit binary addition

We prove formally that an evolved genotype produces a general adder. The genotype was evolved with a “To Do” list of length equal to 1. It is 20 nodes long but only 7 nodes are active. This can be seen in the first row of Fig. 11. There are two input obtaining nodes INP at positions 0 and 1. There are three Boolean functions used BF6 (Exclusive-OR), BF11 and BF1 appearing at positions 3, 4, and 6. The OUTPUT function receives the output from the BXOR node at position 3. The only active SM node is DUP at position 2. Its arguments cause it to copy 14 nodes beginning at the node on its left (BF6) and insert them (at the next iteration) immediately before the node at 17. The action of the DUP node is shown using an arrow emanating from the box containing the 14 nodes that will be re-inserted. After insertion in the phenotype after node 16 the formerly inactive nodes, 7, 10, 11, 14, 15, 16 (see box B) are activated by nodes on the right (in box A). After the copying operation the nodes in box A in the genotype, beginning with node 3, become the nodes in the new phenotype beginning at node 17 (box A). As a result, new inputs (inside box B) are obtained through the action of INP nodes at positions 10 and 11 and two more SM functions DU2 and DUP, located at positions 7 and 14, respectively, become active. However, because the length of the “To Do” list is 1

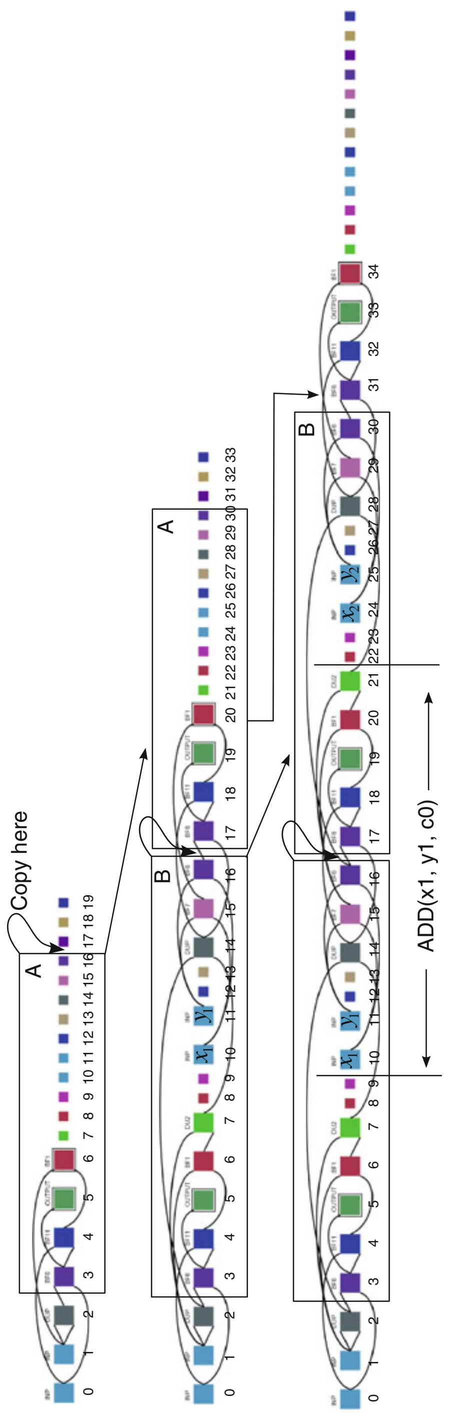

Fig. 11 A schematic of an evolved genotype and two subsequent phenotypes which represent a general binary adder

Table 10 Percentage of solutions that generalise to various, un-evolved input lengths

No. Of inputs	% Success
10	80
50	76
100	76
250	76
500	76
750	76
1000	76

these instructions have no self-modifying role. Instead they have the passive computational role of passing their second input. When the phenotype at the first iteration is executed the DUP function (at position 2) cause fourteen more nodes to be copied after node 16. This is shown in box B in the second iteration (Table 10).

To establish that these operations when repeated will construct an adder of an arbitrary size we need to consider the way the phenotypes can be divided into recognizable modules (i.e., a series of simple adders). In Fig. 12 we show how the phenotypes can be decomposed into a series of one-bit adder modules. Initially the genotype implements a one bit adder without carry, ADD(x0, y0). A one bit adder with carry-in, C_{in}, and carry-out, C_{out} is defined by the truth table shown in Table 11.

The equations for the sum and carry bits, S and C_{out} are given below.

$$\begin{aligned} S &= x \oplus y \oplus C_{in} \\ C_{out} &= xy \oplus C_{in}(x \oplus y) \end{aligned} \tag{6}$$

Let us examine the nodes in the box labeled ADD(x0,y) in Fig. 12. First note that it uses functions BF_6, BF_{11} and BF_1 (see 2). BF_6 is the exclusive-OR function, BF_1 is the AND function and $BF_{11}(a,b) = 1 \oplus b \oplus ab$. The equations below show the outputs z_i of all the active nodes with labels i.

$$\begin{aligned} z_0 &= x_0 \\ z_1 &= y_0 \\ z_2 &= y_0 \\ z_3 &= x_0 \oplus y_0 \\ z_4 &= 1 \oplus x_0 \oplus y_0 \oplus x_0(x_0 \oplus y_0) = 1 \oplus y_0 \oplus x_0 y_0 \\ z_5 &= x_0 \oplus y_0 \\ z_6 &= y_0(1 \oplus y_0 \oplus x_0 y_0) = x_0 y_0 \end{aligned} \tag{7}$$

When $C_{in} = 0$ in Eq. 6 and the resulting equations compared with Eq. 7 one easily sees that z_5 and z_6 are identical with S and C_{out}, respectively. Thus the active nodes in the genotype implement a one-bit adder with no carry-in. The next thing we need to show is that the nodes in Fig. 12 in the box labeled ADD(x1, y1, c0) do indeed implement a one bit adder with carry-in and carry-out (Eq. 6). Let us examine the

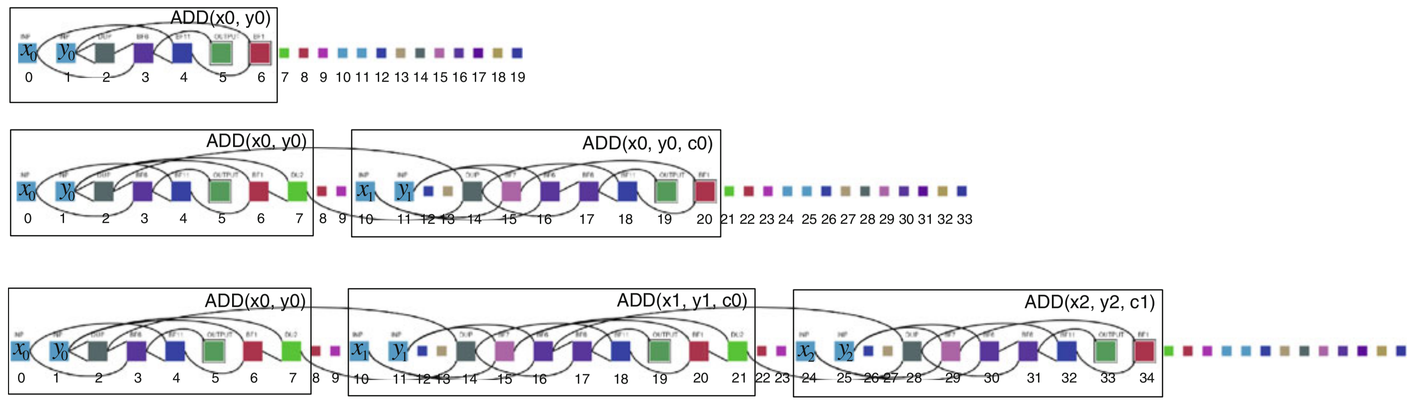

Fig. 12 After each duplication (box B in Fig. 11) a one-bit adder with carry ADD(x, y, c) is created which receives new inputs through two activated INP functions and the carry from the previous stage through a passive DU2 node. It in turn passes the computed sum to an OUTPUT node and the new carry to a DU2 node

Table 11 Truth table of a one-bit adder with carry-in and carry-out

C_{in}	x	y	C_{out}	S	C_{in}	x	y	C_{out}	S
0	0	0	0	0	0	0	1	0	1
0	1	0	0	1	0	1	1	1	0
1	0	0	0	1	1	0	1	1	0
1	1	0	1	0	1	1	1	1	1

nodes 10–20. One more additional Boolean function, $BF_7(a,b) = a \oplus b \oplus ab$ is used. Once again we write equations for the output of all nodes.

$$\begin{aligned}
z_{10} &= x_1\\
z_{11} &= y_1\\
z_{14} &= c_0\\
z_{15} &= BF_7(y_1, x_1) = x_1 \oplus y_1 \oplus x_1y_1\\
z_{16} &= x_1 \oplus y_1\\
z_{17} &= x_1 \oplus y_1 \oplus c_0\\
z_{18} &= BF_{11}(c_0, x1 \oplus y_1 \oplus c_0) = 1 \oplus (x1 \oplus y_1 \oplus c_0) \oplus c_0(x_1 \oplus y_1 \oplus c_0)\\
&= 1 \oplus (x_1 \oplus y_1) \oplus c_0(x_1 \oplus y_1)\\
z_{19} &= x_1 \oplus y_1 \oplus c_0\\
z_{20} &= BF1(x_1 \oplus y_1 \oplus x_1y_1, 1 \oplus (x_1 \oplus y_1) \oplus c_0(x_1 \oplus y_1))\\
&= ((x_1 \oplus y_1) \oplus x_1y_1)(1 \oplus (x_1 \oplus y_1) \oplus c_0(x_1 \oplus y_1))\\
&= ((x_1 \oplus y_1) \oplus (x_1 \oplus y_1) \oplus c_0(x_1 \oplus y_1) \oplus x_1y_1 \oplus x_1y_1(x_1 \oplus y_1) \oplus c_0x_1y_1 \oplus c_0x_1y_1\\
&= x1y1 \oplus c_0(x_1 \oplus y_1)
\end{aligned} \tag{8}$$

Comparing with Eq. 6 we see that the output S is correctly obtained from z_{19} and the carry-out is obtained from z_{20}. In the phenotype at iteration 2 the carry out is passed to the next module by the SM node DU2 (node 21).

So to summarize, at the first iteration. the duplicated section of the phenotype activates previously inactive nodes which form the front section of a one-bit adder and also activates a SM node which passes the carry of the previous adder (via a DUP node) into the newly formed adder. In the second iteration the phenotype consists of three adders connected in the manner of a ripple carry adder. The first adder block originated in the original genotype (first phenotype). It passes out the sum bit computed to an OUTPUT node and passes out its carry (ADD(x0, y0)) to the next adder block, which is a full one bit adder (with carry-in and carry-out). This is shown as ADD(x1, y1, c0). This in turn passes its carry (c1) to the final adder block, ADD(x2, y2, c1) which passes out the two most significant sum bits. The process is entirely regular and it can be easily seen that carrying out further iterations adds a new adder block, ADD(x, y, c) into the existing adder. Thus we can see that the iterated genotype represents an arbitrary bit adder.

We have many different solutions for adders which appear to be general and it is highly likely that some of these are very innovative ways of building general adders. Such analysis remains for the future.

6 Experiments: patterns and sequences

6.1 Digits of π

The task here is to find a program that on each iteration will output the next digit of π, i.e., the first time the program is executed it outputs 3. Then after self-modification is applied, the program encoded in the phenotype again should output 1. Then 4, 1, 5, and so on.

The program inputs are the iteration, i, and the previous output value.

An integer data type was used. Again, function set E was used (see Table 6). The fitness function terminated iteration when an incorrect digit was given as output. Fitness is defined as the total number of correct digits that were output before making a mistake. Evolution was allowed to continue for 10,000,000 evaluations (or 100 digits of π).

6.2 Digits of π results

The experiment was repeated 310 times. The longest sequence found was 31 digits. The shortest was 5 digits, and the average number of correct digits was 14. The best evolved solution produced 31 digits of π, before outputting an incorrect digit. The evolved output sequence and the expected output sequence are shown in Table 12.

Figure 13 shows the development stages of the first ten iterations of this program. Each iteration outputs the next digit of π, starting with 3 in the first step. The program consistently uses the final node, a copy to stop (CTS) function, as its output. This is because no OUTPUT nodes were used, so the graph runner defaulted to using the last node in the graph as output.

The program unfolds as follows. In iteration 0, CTS (Copy To Stop) returns the constant 3.29, but because the program is interpreted as integers, the value is truncated to 3. In iteration 1, INP returns the first input, which is the current iteration (1). The CTS node returns the DSQRT (square root) of 1. In iteration 2, CTS returns the truncated value of the constant, i.e., 4. In iteration 3, the CTS node returns the square root of the square root of the current iteration (3). As a truncated integer, this

Table 12 Output from the π generator compared to actual digits

Evolved output	Correct digits
3141592653 5897932384 6264338327	3141592653 5897932384 6264338327
9 653334444 4444444444 4444444444	9 502884197 1693993751 0582097494
4444555555 5555555555 5555555555	4592307816 4062862089 9862803482
5555555555	5342117068

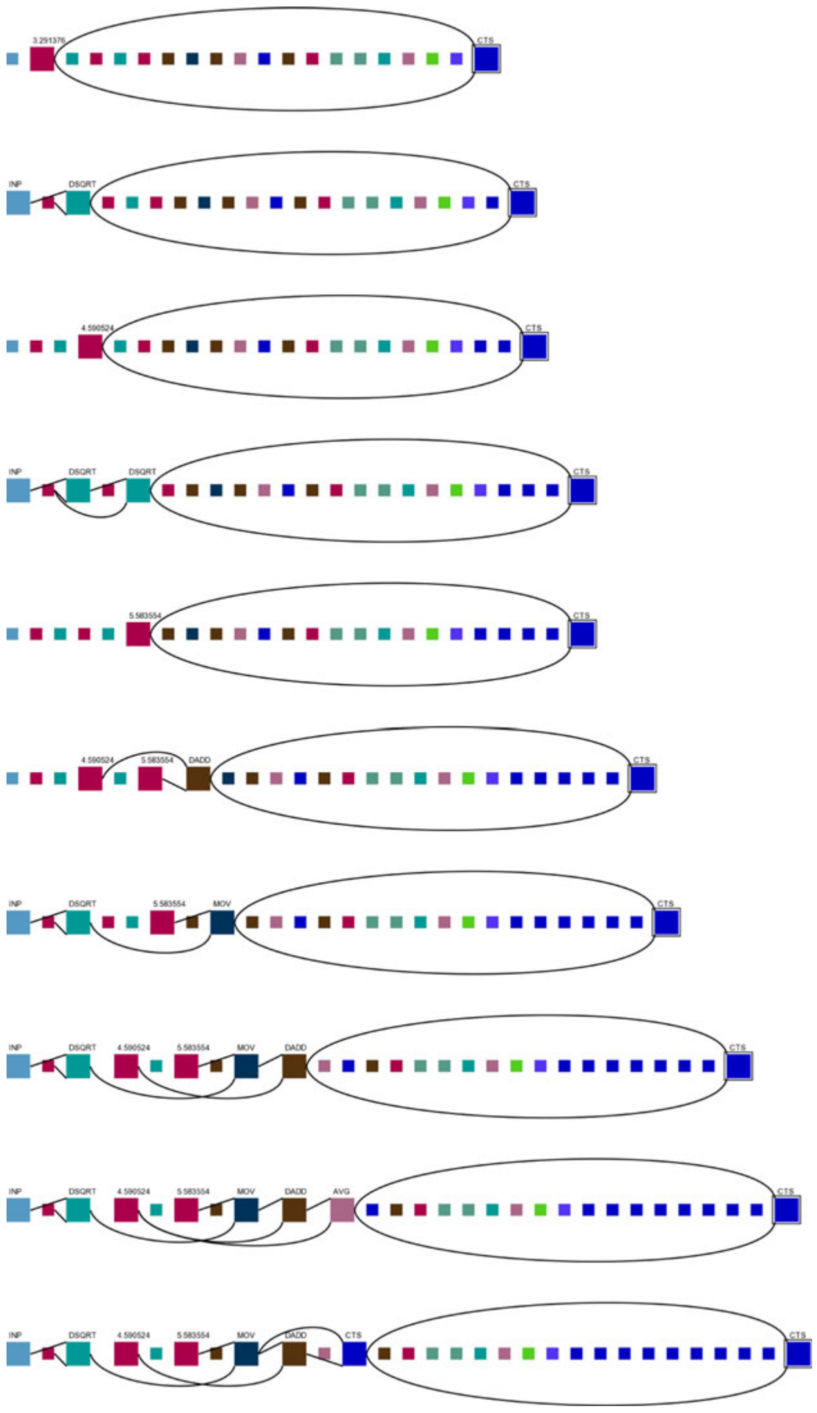

Fig. 13 The first ten developmental steps of a program that produces a π digits sequence

is 1. In iteration 4, again, the CTS node returns a value (5) from a constant. In iteration 5, the output comes from adding two constants 4 and 5, to return 9. In iteration 6, here the CTS node connects to a MOV (Move) function which is returning the square root of 6 (as integer), i.e., 2. In iteration 7, the output (6) is the sum (DADD) of 4 and 2 (which is the integer square root of 7). In iteration 8, the output comes from the average (AVG) of 4 and 6 (via the same calculations as iteration 7), to get 5. In iteration 9, the input value is 9, so the square root function now outputs 3. The left most CTS function returns 3 (via the MOV node connected to the top input). This is because of the order of modification nodes has reached a limit on the "To Do" list, and the operation has failed—changing which of the inputs is selected as the output. The program continues in this fashion for the first 31 digits.

6.3 Squares

In this task, we ask that evolution finds a program that generates a sequences of squares 0, 1, 4, 9, 16, 25, ... without using multiplication or division operations. As Spector [54] who first devised this problem points out, this task can only be successfully performed if the program can modify itself—as it needs to add new functionality in the form of additions to produce the next integer in the sequence. Hence, genetic programming, without iteration, will be unable to find a general solution to this problem.

6.4 Squares results

Programs were evolved using a very restricted function set. Programs have one input: the current iteration. The initial graph size was set to 20 nodes. Mutation rate 0.1. Function set D from Table 6 was used.

Out of 50 trials and a maximum of 1,000,000 evaluations, all successfully evolved the first ten outputs correctly. The average number of evaluations needed was 35,224. The minimum and maximum number of evaluations are 135 and 249,969. With a standard deviation of 53,609. Of these, 84% were found to successfully generalise to the first 100 values.

6.5 Fibonacci

In a task similar to the squares problem (see Sect. 6.3), we evolve a program that when iterated produces the Fibonacci sequence. Again, we limit the function set to force evolution to find a solution that requires self-modification.

We evolve for both the first *n* and *m* numbers in the sequence and test for generality to 42 numbers (after which the value exceeds a long int). We have previously noted that the programs produced by evolution generalize although they have an irregular pattern to begin with (see Sect. 6.3). We were intrigued to see the behaviour when starting the base case for Fibonacci at either the arbitrary start of 1, 1 or at the next step of that sequence 1, 2.

6.6 Fibonacci results

Programs were evolved using a very restricted function set. Programs have one input, the numeric constant 1. The initial graph size was set to 20 nodes. Mutation rate 0.1. Function set D was used (see Table 6).

Out of 50 trials and a maximum of 1,000,000 evaluations, all successfully evolved the first ten outputs correctly. The average number of evaluations needed was 71,812. The minimum and maximum number of evaluations are 630 and 924,333. With a standard deviation of 162,998. Of these, 46% were found to successfully generalise to the first 100 values.

7 Experiments: mathematical

7.1 Approximating π

There exist several iterative approaches to approximating π [9]. For example, the Gregory-Leibniz series calculates:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} \ldots$$

This series is simple, but requires a large number of iterations to reach good accuracy. Another method[3] uses recursion to find an approximation:

$$\pi = n(\tan(\pi/n) - \frac{\tan^3(\pi/n)}{3} + \frac{\tan^5(\pi/n)}{5} - \frac{\tan^7(\pi/n)}{7} \ldots)$$

for $n = 1, 2, \ldots$. Curiously, there has been little work on evolving approximators to π, despite it being a well defined problem with many human designed solutions to compare against.

Krohn et al. [33] employed an artificial developmental system based on fractal proteins [7] to produce approximations to π using two different approaches. The first approach was to generate the digits as a binary sequence. The second, and more successful, approach was to use the output of the developmental system to provide values for the equation:

$$\sum_{i=1}^{I} \frac{\sum_{n=2}^{N} B_{n,i}}{\prod_{t=1}^{i} B_{1,i}}$$

where I is the number of developmental iterations, N is the number of behavioural genes (an output of the evolved program) and $B_{n,i}$ is the output of the nth behavioural gene at iteration i.

Our fitness function was designed to produce a program where subsequent iterations of the program would produce more accurate approximation to π. Programs were allowed to iterate for a maximum of ten iterations. If the output after an iteration did not approximate π more closely, evaluation was stopped and a large

[3] http://www.ams.org/featurecolumn/archive/pi-calc.html.

Table 13 Variants of the fitness function with different input strategies

Config.	Inputs	Description
A	One input: the current iteration	Functions can be built using the current iteration counter as a parameter
B	One input: numeric constant (1)	The program has no real input, and therefore has to build a structure that performs the iterative process
C	Two inputs: the current iteration and last outputted value	This form can, in some sense, be viewed as recurrent, as programs can depend on the previous output
D	Two inputs: numeric constant (1) and last outputted value	This form can also be viewed as recurrent, as programs can depend on the previous output

fitness penalty applied. Note that it is possible that after the ten iterations the output value diverges from π, and the quality of the result would therefore worsen.

The fitness score of an individual is defined as the absolute error of the last output. In addition, the string representation of the output was checked to ensure that all digits matched correctly. We were limited to a precision of 14 decimal places (i.e., 3.14159265358979), due to using double precision representation. Four variants of the fitness function were tested, each with different configurations of inputs given to the programs, these are described in Table 13.

7.2 Approximating π results

The statistical results for these experiments are shown in Table 14. Each experiment was conducted approximately 150 times. The standard deviations are large and overlap, which means that the algorithms appear to perform similarly.

7.3 Example π generator

Figure 14 shows the output of an evolved SMCGP program that accurately converges to π. Table 15 shows the output of the program at each time step. As the program is relatively short, it was possible to extract the evolved generating function:

$$f(i) = \begin{cases} \cos(\sin(\cos(\sin(0)))) & i = 0 \\ f(i-1) + \sin(f(i-1)) & i > 0 \end{cases} \tag{9}$$

Equation 9 is a nonlinear recurrence relation. However, it can be shown that it converges rapidly to π. When $i = 10$, the output matches the first 2,048 digits of π.[4] We can note that the value of π is a fixed point of Eq. 9 since $x = x + \sin(x)$ is obeyed when $x = \pi$. It is stable since $f'(x) = 1 + \cos(x) = 0$ when $x = \pi$. How rapidly it converges to π can be seen from the following argument. Suppose at some

[4] Tested using the mpmath library for Python: http://www.code.google.com/p/mpmath/.

Table 14 Finding π using various inputs to the evolved programs

Config.	% Success	Avg. Evals	Min	SD	Min., Avg. iterations
A	96.7	8,952,441	6,731	17,766,584	3, 5.95
B	99.4	2,905,673	526	8,826,492	3, 5.49
C	96.2	4,953,518	869	13,674,006	3, 6.78
D	98.7	2,146,348	642	9,515,520	3, 5.90

Experiment types: A, One input, the current iteration; B, One input, numeric constant (1); C, Two inputs, the current iteration and last outputted value; D, Two inputs, numeric constant (1) and the last outputted value. Iterations refers to the number of times the program has to be iterated before it reaches π to 14 decimal places

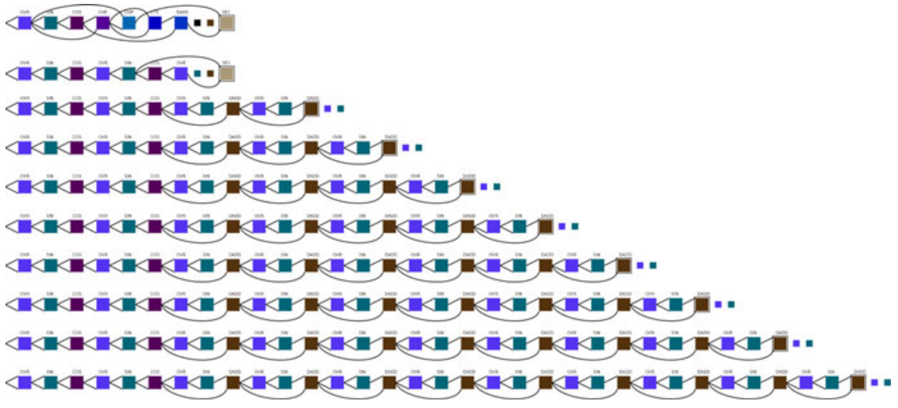

Fig. 14 SMCGP program that produces an approximation to π. Each row is a different time step

iteration m, $f(m)$ is close to π. Then we can write $f(m) = \pi - \delta$, where δ is a small quantity. Then from Eq. 9 $f(m+1) = \pi - \delta + \sin(\pi - \delta) \approx \pi - \frac{\delta^3}{3!}$.

7.4 Summing numbers

Here we wished to evolve a program 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, but relatively easy for a technique such as neural networks. The premise being, that neural networks appear to perform well when combining input values and genetic programming seems to work well using feature selection on the inputs.

Input vectors consist 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 sets consists of the self-modifying functions and just the ADD function.

Table 15 Output from program shown in Fig. 14

Iteration	Output error	Output	Correct digits
0	3.14159265358979	0	0
1	2.47522590819691	0.666366745392881	0
2	0.897795232223359	2.24379742136643	0
3	0.115840730988874	3.02575192260092	0
4	0.000258905467357184	3.14133374812244	3
5	2.89235302375346E-12	3.1415926535869	11
6	0	3.14159265358979	14
7	0	3.14159265358979	14
8	0	3.14159265358979	14
9	0	3.14159265358979	14

Output is error is the difference between π (.Net's Math.PI) and the output from the program. Correct digits is the count of the correctly matching digits after the decimal point. Errors will appear to be 0 when the actual error is very small

7.5 Summing numbers results

All 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). Table 16 shows the number of evaluations required to reach the n-th sum (where n runs from 2 to 10).

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 most solutions generalized, however, in 1% of cases, they did not.

We also tested the ability of conventional CGP to sum a set of numbers. Here CGP could only be evolved for a given size of set of input numbers. The results (based on 200 runs) are also shown in Table 16. This experiment revealed that CGP is able to solve this problem only for a smaller sets of numbers. This shows a clear benefit of the self-modification approach in comparison with the direct encoding.

7.6 Regression

We devised a new problem that tests the ability of SMCGP to learn a "modular" 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). Function set F was used, as detailed in Table 6. As with the squares problem, without self-modification, it would be impossible for GP to produce a general solution to this problem.

We evolved to $n = 10$ and then tested for generality up to $n = 20$. As with other experiments, we evolved incrementally. We first required the programs to solve $n = 1$. When that was successful, we evolved for $n = 1$ and $n = 2$. Next for $n = 1, 2, 3$ and so on.

Table 16 Evaluations required to evolve a SMCGP program that can add a set of numbers of a given size

Size of set	Average	Minimum	Maximum	SD	% CGP
2	50	50	50	0	100
3	1,208	54	6,764	987	80
4	2,338	62	19,307	2,025	95.8
5	3,120	87	23,149	2,549	48
6	4,026	126	42,168	4,068	38.1
7	5,010	204	48,824	5,447	0
8	5,931	213	68,201	7,033	0
9	6,788	231	87,949	8,348	0
10	7,434	246	87,976	8,779	0

Hundred percent of SMCGP experiments were successful. The % success rate for conventional CGP is also shown

7.7 Regression results

Table 17 shows the results summary for the powers regression problem. All runs were successful. In this instance, we see an interesting difference between the two starting conditions. If the fitness function starts with $n = 1, ..., 5$, we find that fewer evaluations are required to reach $n = 10$. However, this leads to reduced generalization. Using a Kolmogorov-Smirnov test, we find that the difference in the evaluations required is statistically significant ($P < 0.01$).

7.8 Classification

In this experiment we wanted to investigated the behaviour of SMCGP on a problem in which there appeared to be no clear benefit for self-modification. The problem we chose is a protein classification problem—as described in [35]. The task is to predict the location of a protein in a cell, from the amino acids in the particular protein. We used the entire dataset as training set. The set consisted of 2,427 entries, with 19 variables each and 1 output. The function set for SMCGP includes all the mathematical operators in addition to the self-modifying command set. The CGP function set contained just the mathematical operators.

We allowed the phenotype to iterate the number of times specified in the genotype before we tested the program on the training set. Function set F (see Table 6) was used for the SMCGP function set, and for CGP the same set was used but without the self-modification functions.

Table 17 Summary of results for the powers regression problem

Number of initial test sets	Average evaluations	SD	Percentage generalize
1	687156	869699	60.4
5	527334	600800	55.6

7.9 Classification results

Table 18 shows the summary of results for the protein localization problem. We see under these conditions, that both CGP and SMCGP perform similarly.

This is encouraging as it suggests that using SMCGP does not worsen performance (compared with CGP) on a problem where there is no clear need for self-modification. Of course, further work is needed to confirm this on a wider collection of problems.

8 Conclusions and further work

It is 10 years since the birth of Cartesian Genetic Programming and it is fitting that we should now be reporting on an improved form of it called Self-Modifying CGP. The new technique has borrowed some concepts from developmental biology. It extends the capability of CGP. Unlike CGP, SMCGP allows us to evolve solutions to whole classes of problems rather than specific instances with a fixed number of inputs and outputs.

We have also shown that it is suitable for a wide range of computational problems, and that it can out perform previous approaches on many of these problems. Given the remarkable success of the technique, it might be worth investigating whether other methods of GP couldn't be extended following a similar route. We speculate that in both linear GP an in tree-based GP, it should be possible to implement analogous operations, perhaps with similar effects.

There remains much to be investigated. For example, it is unclear what the best parameter configuration should be. Here we used arbitrary parameters, and tried to maintain consistency throughout experiments, however, it is likely that these were sub optimal. Parameter settings seem to be very different from CGP. For example, here we use small genotypes and large mutation rates, whereas CGP appears to work best with large genotypes and small mutation rates.

The available function set is also an area that needs investigating. The current set of self-modifying functions is unlikely to be optimal. However, it is very hard to predict what functions are actually most useful. When examining the evolved

Table 18 Results summary for the bioinformatics classification problem

-	CGP	SMCGP
Average fitness (training)	66.81	66.83
SD fitness (training)	6.35	6.45
Average fitness (validation)	66.10	66.18
SD fitness (validation)	5.96	6.46
Avg. evaluations to best fitness (training)	7,679	7,071
SD evaluations to best fitness (training)	2,452	2,644
Avg. evaluations to best fitness (validation)	7,357	7,161
SD evaluations to best fitness (validation)	2,386	2,597

programs, our intuition has sometimes been proved wrong about whether certain functions would be beneficial to evolution.

The methods used to activate self-modifying functions depending on the values of numerical inputs need to be investigated further to ascertain how useful they are. At present they are designed with bi-arity functions in mind. It would be interesting to consider activation mechanisms for self-modifying functions inspired by epigenetics in biology.

Although SMCGP is a relatively complex technique, conceptually it is simple, and it is straight forward to implement. SMCGP execution can be implemented very efficiently, and graphs containing thousands of nodes are easily handled. We have yet to explore the possibilities that this brings.

A significant aspect of SMCGP is that it can produce mathematically provable solutions to general classes of problems. It seems possible that it could produce hitherto unknown general solutions to some problems. These solutions may have utility in a number of research domains. We intend to continue to investigate such areas. Utilizing a theorem prover in the fitness function, rather than carrying out post-hoc proofs though desirable, is likely to be computationally intractable as theorem provers, even if they could be used, have poor time complexity.

The form of SMCGP we have described here uses a one-dimensional graph representation, however, we have also been investigating a form of SMCGP in which programs can be developed that are a two-dimensional sheet of nodes. Early experiments indicate that this allows solutions to problems to be evolved even faster than the one-dimensional form.

References

1. C. Adami, C. Brown, Evolutionary learning in the 2d artificial life system AVIDA, in *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. (MIT Press, 1994), pp. 377–381
2. I. Aleksander, *Neural Computing Architectures: The Design of Brain-Like Machines* (MIT Press, Cambridge, 1989)
3. M. Arnold, S. Fink, D. Grove, M. Hind, P. Sweeney, A survey of adaptive optimization in virtual machines. Proc. IEEE **93**, 449–466 (2005)
4. J. Aycock, A brief history of just-in-time. ACM Comput. Surv. (CSUR) **35**, 97–113 (2003)
5. W. Banzhaf, G. Beslon, S. Christensen, J.A. Foster, F. Képès, V. Lefort, J.F. Miller, M. Radman, J.J. Ramsden, From artificial evolution to computational evolution: a research agenda. Nat. Rev. Genet. **7**, 729–735 (2006)
6. W. Banzhaf, J. Miller, The challenge of complexity, in *Frontiers in Evolutionary Computation*, ed. by A. Menon (Kluwer Academic, Boston, MA, 2004), pp. 243–260
7. P. Bentley, Fractal proteins. Genet. Program. Evol. Mach. **5**(1), 71–101 (2004)
8. P. Bentley, S. Kumar, Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem in *Proceedings of the Genetic and Evolutionary Computation Conference*, ed. by W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, R.E. Smith, vol 1 (Morgan Kaufmann, Orlando, 13–17 1999), pp. 35–43
9. J. Borwein, D. Bailey, R. Girgensohn, *Experimentation in Mathematics—Computational Paths to Discovery* (A. K. Peters, Ltd, Ellesley, MA, 2003)
10. J. Clune, B.E. Beckmann, C. Ofria, R.T. Pennock, Evolving coordinated quadruped gaits with the hyperneat generative encoding in *Proceedings of the IEEE Congress on Evolutionary Computing* (2009), pp. 2764–2771

11. N. Doidge, *The Brain that Changes Itself: Stories of Personal Triumph from the Frontiers of Brain Science* (Penguin Group, USA, 2007)
12. A. Donlin, Self modifying circuitry—a platform for tractable virtual circuitry in *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm* (Springer, London, 1998), pp. 199–208
13. T.G. Gordon, P.J. Bentley, Development brings scalability to hardware evolution in *EH '05: Proceedings of the 2005 NASA/DoD Conference on Evolvable Hardware* (IEEE Computer Society, Washington, DC, 2005), pp. 272–279
14. F. Gruau, *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France, 1994
15. F. Gruau, D. Whitley, L. Pyeatt, A comparison between cellular encoding and direct encoding for genetic neural networks in *Genetic Programming 1996: Proceedings of the First Annual Conference*, ed. by J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo. (MIT Press, Stanford University, CA, 28–31 1996), pp. 81–89
16. S. Harding, J.F. Miller, W. Banzhaf, Self-modifying cartesian genetic programming in *GECCO*, ed. by H. Lipson (ACM, 2007), pp. 1021–1028
17. S. Harding, J.F. Miller, W. Banzhaf, Evolution, development and learning with self modifying cartesian genetic programming in *Genetic and Evolutionary Computation Conference, GECCO 2009. Accepted for publication.* (2009)
18. S. Harding, J.F. Miller, W. Banzhaf, Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing in *EuroGP '09: Proceedings of the 12th European Conference on Genetic Programming* (Springer, Berlin, 2009), pp. 133–144
19. S. Harding, J.F. Miller, W. Banzhaf, Self modifying cartesian genetic programming: parity in *2009 IEEE Congress on Evolutionary Computation*, ed. by A. Tyrrell (IEEE Computational Intelligence Society, IEEE Press, Trondheim, Norway, 18–21 May 2009), pp. 285–292
20. G.S. Hornby, J.B. Pollack,The advantages of generative grammatical encodings for physical design in *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001* (IEEE Press, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27–30 2001), pp. 600–607
21. P.E. Hotz, Comparing direct and developmental encoding schemes in artificial evolution: a case study in evolving lens shapes in *Congress on Evolutionary Computation, CEC 2004* (2004)
22. L. Huelsbergen, Finding general solutions to the parity problem by evolving machine-language representations in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, ed. by J.R. Koza, W. Banzhaf et al (Morgan Kaufmann, University of Wisconsin, Madison, 22–25 July 1998), pp. 158–166
23. IEEE Computer Society, Keywords, http://www.computer.org/portal/web/publications/acmtaxonomy
24. G. Kampis, *Self-Modifying Systems in Biology and Cognitive Science: A New Framework for Dynamics, Information, and Complexity* (Pergamon, Oxford, 1991)
25. G. Kampis, *Life-Like Computing Beyond the Machine Metaphor* (Chapman and Hall, London, 1993)
26. G. Kampis, Self-modifying systems: a model for the constructive origin of information. BioSystems **38**, 119–125 (1996)
27. Y. Kanzaki, A. Monden, M. Nakamura, K. Matsumoto, Exploiting self-modification mechanism for program protection in *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings of 27th Annual International* (2003), pp. 170–179
28. R. Kicinger, Evolutionary development system for structural design in *AAAI Fall Symposium in Developmental Systems* (2006)
29. H. Kitano, Designing neural networks using genetic algorithms with graph generation system. Complex Syst. **4**(4), 461–476 (1990)
30. J. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection* (MIT Press, Cambridge, 1992)
31. J. Koza, J. Rice, *Genetic Programming* (MIT Press, Cambridge, 1992)
32. J.R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT Press, Cambridge, 1994)
33. J. Krohn, P.J. Bentley, H. Shayani, The challenge of irrationality: fractal protein recipes for pi in *GECCO*, ed. by F. Rothlauf (ACM, 2009), pp. 715–722
34. S. Kumar, P. Bentley, *On Growth, Form and Computers* (Academic Press, London, 2003)
35. W.B. Langdon, W. Banzhaf, Repeated sequences in linear genetic programming genomes. Complex Syst. **15**(4), 285–306 (2005)

36. H. Maturana, F. Varela, *Autopoiesis and Cognition: The Realization of the Living* (Springer, New York, 1980)
37. P. McKinley, B. Cheng, C. Ofria, D. Knoester, B. Beckmann, H. Goldsby, Harnessing digital evolution. IEEE Comput. **41**(1), 54 (2008)
38. N.F. McPhee, E.F. Crane, S.E. Lahr, R. Poli, Developmental plasticity in linear genetic programming (ACM, Nominated for best paper award in the GP track in *GECCO '09: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ed. by G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C.B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.-G. Beyer, K. Stanley, J.F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. Di Penta, B. Doerr, T. Jansen, R. Poli, E. Alba. (Montreal, 8–12 July 2009), pp. 1019–1026
39. 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)* (Morgan Kaufmann, Orlando, 1999), pp. 1135–1142
40. J.F. Miller, D. Job, V.K. Vassilev, Principles in the evolutionary design of digital circuits—part I. Genet. Program. Evol. Mach. **1**(1), 8–35 (2000)
41. J.F. Miller, S.L. Smith, Redundancy and computational efficiency in cartesian genetic programming in *IEEE Transactions on Evoluationary Computation*, vol 10 (2006), pp. 167–174
42. J.F. Miller, P. Thomson, in *Proceedings of EuroGP 2000*, ed. by R. Poli, W. Banzhaf et al. Cartesian genetic programming. LNCS, vol 1802 (Springer, 2000), pp. 121–132
43. J.F. Miller, P. Thomson, Lecture Notes in Computer Science in *ICES*, ed. by A.M. Tyrrell, P.C. Haddow, J. Torresen. A developmental method for growing graphs and circuits, vol 2606 (Springer, 2003), pp. 93–104
44. P. Nordin, W. Banzhaf, Evolving turing-complete programs for a register machine with self-modifying code in *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95).* (1995), pp. 318–325
45. R. Poli, J. Page, Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code gp and demes. Genet. Program. Evol. Mach. **1**(1/2), 37–56 (2000)
46. S. Rasmussen, C. Knudsen, R. Feldberg, M. Hindsholm, The coreworld: emergence and evolution of cooperative structures in a computational chemistry. Phys. D Nonlinear Phenom. **42**(1–3), 111–134 (1990)
47. T. Ray, An evolutionary approach to synthetic biology: zen and the art of creating life. Artif. Life **1**(1–2), 179–209 (1993)
48. D. Roggen, D. Federici Multi-cellular development: is there scalability and robustness to gain?, in *Proceedings of Parallel Problem Solving from Nature 8, PPSN 2004*, ed. by X. Yao, E. Burke, J. Lozano et al. (2004), pp. 391–400
49. R. Rubinstein, J. Shutt, Self-modifying finite automata: an introduction. Inf. Process. Lett. **56**(4), 185–190 (1995)
50. J. Schmidhuber, J. Zhao, N. Schraudolph, Reinforcement learning with self-modifying policies, in *Learning to learn*, ed. by S. Thrun, L. Pratt (Kluwer, 1997), pp. 293–309
51. L. Sekanina, M. Bidlo, Evolutionary design of arbitrarily large sorting networks using development. Genet. Program. Evol. Mach. **6**(3), 319–347 (2005)
52. A. Siddiqi, S. Lucas. A comparison of matrix rewriting versus direct encoding for evolving neural networks (1998)
53. L. Spector, A. Robinson, Genetic programming and autoconstructive evolution with the push programming language. Genet. Program. Evol. Mach. **3**, 7–40 (2002)
54. L. Spector, K. Stoffel, Ontogenetic programming in *Genetic Programming 1996: Proceedings of the First Annual Conference*, ed. by J.R. Koza, D.E. Goldberg et al. (MIT Press, Stanford University, CA, 28–31 1996), pp. 394–399
55. K.O. Stanley, Compositional pattern producing networks: a novel abstraction of development. Genet. Program. Evol. Mach. **8**, 131–162 (2007)
56. V.K. Vassilev, J.F. Miller, The advantages of landscape neutrality in digital circuit evolution in *Proceedings of ICES*, vol 1801 (Springer, 2000), pp. 252–263
57. J.A. Walker, J.F. Miller, in *Proceedings of the 7th European Conference on Genetic Programming (EuroGP).* Evolution and acquisition of modules in cartesian genetic programming. Lecture Notes in Computer Science, vol 3003 (Springer, 2004), pp. 187–197
58. J.A. Walker, J.F. Miller, Automatic acquisition, evolution and re-use of modules in cartesian genetic programming. IEEE Trans. Evol. Comput. **12**, 397–417 (2008)

59. G.C. Wilson, W. Banzhaf, A comparison of cartesian genetic programming and linear genetic programming in *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, ed. by M. O'Neill, L. Vanneschi, S. Gustafson, A.I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, E. Tarantino. Lecture Notes in Computer Science, vol 4971 (Springer, Naples, 26–28 Mar 2008), pp. 182–193
60. M.L. Wong, Evolving recursive programs by using adaptive grammar based genetic programming. Genet. Program. Evol. Mach. **6**(4), 421–455 (2005)
61. M.L. Wong, K.S. Leung, Evolving recursive functions for the even-parity problem using genetic programming in *Advances in Genetic Programming 2*, ed. by P.J. Angeline, K.E.E. Kinnear Jr., chapter 11 (MIT Press, Cambridge, 1996), pp. 221–240
62. M.L. Wong, T. Mun, Evolving recursive programs by using adaptive grammar based genetic programming. Genet. Program. Evol. Mach. **6**(4), 421–455 (2005)
63. T. Yu, Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. Genet. Program. Evol. Mach. **2**(4), 345–380 (2001)
64. T. Yu, J. Miller, Neutrality and the evolvability of boolean function landscape in *Proceedings of EuroGP 2001*, ed. by J.F. Miller, M. Tomassini et al. LNCS, vol 2038 (Springer, 2001), pp. 204–217