Genetic Programming and Evolvable Machines (2020) 21:129–168
https://doi.org/10.1007/s10710-019-09360-6

Cartesian genetic programming: its status and future

Julian Francis Miller[1]

Received: 12 October 2018 / Revised: 7 April 2019 / Published online: 6 August 2019

Abstract
Cartesian genetic programming, a well-established method of genetic programming, is approximately 20 years old. It represents solutions to computational problems as graphs. Its genetic encoding includes explicitly redundant genes which are well-known to assist in effective evolutionary search. In this article, we review and compare many of the important aspects of the method and findings discussed since its inception. In the process, we make many suggestions for further work which could improve the efficiency of the CGP for solving computational problems.

Keywords Cartesian genetic programming · Genetic programming · Evolutionary algorithms

1 Introduction

The term "Cartesian genetic programming" (CGP)[1] first appeared in 1999 [65]. Although, it was a generalisation of a method of encoding and evolving electronic circuits that was first described in 1997 [71]. In 2000, it became established as a new form of genetic programming [70]. Since that time CGP has been adopted by many researchers, adapted by others and applied to many applications areas. This article provides an extensive review of the different variants of CGP, and an analysis of many important aspects of CGP. It discusses numerous open issues and questions. The article contains many suggestions for further work which could lead to improvements in the efficiency of the CGP for solving computational problems.

In contrast to tree-based genetic programming [53, 80], CGP encodes computational structures as directed graphs. Its invention was heavily influenced by earlier work on creating and evolving genetic representations of circuits. These are most naturally encoded as graphs and can be described using structures called netlists. CGP uses a netlist-inspired address-based genotype consisting of integers pointers to an

[1] cartesiangp.com.

✉ Julian Francis Miller
julian.miller@york.ac.uk

[1] Department of Electronic Engineering, The University of York, York, UK

either an array of primitive functions or data locations. Encoding graphs, rather than trees has advantages in that nodes can be multiply used and graphs can have multiple outputs. It is a very adaptable representation as it can easily represent many types of computational structures, such as systems of equations, state-machines, neural networks, algorithms and electronic circuits. Another defining characteristic of CGP is that the genotype can include non-coding genes. The genotype is recursively decoded from program outputs to inputs and in so doing nodes can be ignored, these are non-coding. As we shall the presence of non-coding genes has strongly influenced the choice of search algorithm that is most suitable for CGP (see Sect. 2).

In the article, we discuss many variants of CGP that have been proposed. We begin with a graph-based form of GP called parallel distributed GP (PDGP) which was developed by Poli prior to CGP [77, 78]. This has many similarities with CGP but also uses various graph-based crossover and mutation operators and restricted graph connectivity. Modular CGP (MCGP) is based on standard CGP but has additional mutation operators which allow CGP-encoded sub-functions (called modules) to be captured, re-used or modified. This was motivated by Koza's work on automatically defined functions in tree-based GP [54]. In real-valued CGP (RVCGP) all genes are real-valued and a decoding step translates these into standard CGP genes. This is attractive as it allows many methods for evolving genotypes used in evolutionary algorithms work to be used to evolve programs (e.g. "flat" crossover see Sect. 5).

Implicit-context CGP (ICCGP) removed the positional dependence of genes in CGP by representing components by evolved entities called enzymes which are self-assembled to form CGP-like phenotypes. Self-modifying CGP (SMCGP) extended CGP by introducing new kinds of primitive functions (self-modifying) which carried out transformations of the phenotype, this allowed *sequences* of phenotypes to be evolved from a single genotype. It also required genes in CGP to use *relative* addressing (also used in PDGP). Mixed-type CGP (MTCGP) borrowed some features from SMCGP and also allowed CGP to handle different data types (i.e. scalar and vector). Recurrent CGP (RCGP) is a simple extension to standard CGP which allowed connections between primitive functions to be both feed-forward or feedback (i.e. breaking acyclicity). Iterative CGP (ICGP) introduced conditional loop-genes into CGP which allowed CGP to encode algorithms.

Differentiable CGP (DCGP) is a form of CGP in which phenotypes are differentiable. Primitive functions are truncated Taylor's series and graph connections are weighted. This allows gradient descent to be used to arrive at highly-tuned graphs. We also discuss a recently developed form of graph-based GP called evolving graphs by graph programming (EGGP) in which phenotypes are encoded using a graph description language. This allows graphs to be manipulated more freely than is possible in the standard form of CGP. We also discuss the recently proposed positional CGP (PCGP). This a real-valued form of CGP form in which graph nodes have evolvable positions, and connections are made to the nearest node.

We also discuss in depth various promising forms of crossover, mutation and search algorithms that have been suggested in the literature of CGP. It should be noted that very recently, another review of CGP has appeared. It discusses CGP and

its variants and focuses in detail on their representational differences and evolutionary operators [62].

The plan of the article is as follows. First we review CGP-encoded representations of programs that have been suggested in the literature (Sect. 3). We follow this with a discussion of work done in mutation (Sect. 4) and crossover (Sect. 5). Section 6 discusses various search algorithms that have been investigated for CGP. In Sect. 7 we examine ways in which CGP can be made faster by using hardware. Section 8 discusses major applications of CGP including CGP encoded artificial neural networks. In Sect. 9 we discuss the relatively high comprehensibility of CGP programs. We give comparisons of the efficiency and human-competitiveness of CGP to other machine learning techniques in Sect. 10. The growing amount of publicly available software for CGP is reviewed in Sect. 11. We close with sections on open questions and suggestions for further investigation (Sect. 12) and conclusions. For reference and completeness we begin with a description of the standard form of CGP.

2 Aspects of standard CGP

2.1 Representation

In its standard form CGP programs are representations of directed acyclic graphs. These graphs are represented using a two-dimensional grid of computational nodes. Hence the term "Cartesian".

Each *node* in the directed graph represents a particular function and is encoded by a number of integer genes. Each node has a *function gene* which is the address of the computational function of the node in a user-defined look-up table of functions. The remaining node genes are *connection genes* and say where the node gets its data from. These genes represent addresses in a data structure (typically an array). Nodes take their inputs from either the output of a node (in acyclic CGP, from a previous column) or from a program input. The number of connection genes a node has is chosen to be the maximum number of inputs that any function in the function look-up table has. If any node function requires less inputs, the remaining connection genes are ignored. The program data inputs are given the *absolute* data addresses 0 to $n_i - 1$ where n_i is the number of program inputs. The data outputs of nodes in the genotype are given addresses sequentially, column by column, starting from n_i to $n_i + n_n - 1$, where n_n is the user-determined upper bound on the number of nodes. A schematic is shown in Fig. 1.

If the problem requires n_o program outputs, the genotype is augmented with a number of output genes ($O_i = n_o$). Each of these is an address of a node where the program output data is taken from. Nodes in columns cannot be connected to each other. In its standard form the graph in CGP is directed and feed-forward; this means that a node may only have its inputs connected to either input data or the output of a node in a previous column. However, as we will see in Sects. 3.7 and 3.8, this restriction can be relaxed to allow recurrent or cyclic graphs. A schematic of the genotype is shown in Fig. 1.

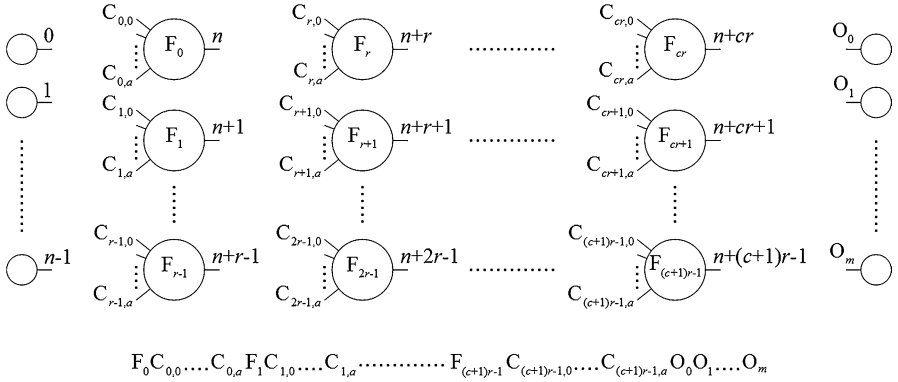

Fig. 1 Standard form of CGP representation. It is a grid of nodes whose functions are chosen from a set of primitive functions. The grid has n_c columns and n_r rows. The number of program inputs $n = n_i$ and the number of program outputs $m = n_o$. Each node is assumed to take as many inputs as the maximum function arity a. Every data input and node output is 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 the outputs of inputs and nodes)

CGP has three user-selectable graph parameters that define the dimensionality and connectivity of the encoded graphs. These are the *number of columns*, the *number of rows* and *levels-back*. These are denoted by n_c, n_r and l, respectively. The product of the first two parameters determine the maximum number of computational nodes allowed: $n_n = n_c n_r$. The parameter l, called levels-back, controls the connectivity of the graph encoded. Levels-back constrains which columns a node can get its inputs from. If $l = 1$, a node can get its inputs only from a node in the column on its immediate left or from a primary input. If $l = 2$, a node can have its inputs connected to the outputs of any nodes in the immediate left two columns of nodes or a primary input. If one wishes to allow nodes to connect to any nodes on their left, then $l = n_c$. Varying these parameters can result in various kinds of graph topology. An important special case of these three parameters occurs when the number of rows is chosen to be one and levels-back is set to be the number of columns (one dimensional topology). In this case the genotype can represent any bounded directed graph where the upper bound is determined by the number of columns. The length of a CGP genotype is given by $L_{cgp} = n_n(a + 1) + n_o$.

$$L_{cgp} = n_n(a + 1) + n_o \tag{1}$$

2.2 Search algorithm

Standard CGP uses a search algorithm (Algorithm 1) that is inspired by the $1 + \lambda$ evolutionary strategy [83] in which a single parent genotype is mutated to create λ offspring. An offspring that has a fitness better or equal to the parent becomes the new parent for the next generation. This is still the most widely used search algorithm for CGP. We discuss work suggesting other search algorithms in Sect. 6.

Algorithm 1 CGP search algorithm

```
1: for all i such that 0 ≤ i < 1 + λ do
2:     Randomly generate individual i
3: end for
4: Select the fittest individual, which is promoted as the parent
5: while Solution is not found or the generation limit is not reached do
6:     for all i such that 0 ≤ i < λ do
7:         Mutate the parent to generate offspring i
8:     end for
9:     Generate the new parent using the following rules:
10:    if A single offspring has a better fitness than any other member of the population then
11:        The offspring is chosen as parent
12:    else if One or more offspring have an equal fitness to the parent then
13:        Randomly choose one of these as parent
14:    else
15:        The parent chromosome remains the same as before
16:    end if
17: end while
```

2.3 Non-coding genes

The reason that such a simple search strategy operating on such a small population is effective is strongly connected with the presence of non-coding genes in the genotype. Non-coding genes arise in CGP when the outputs of nodes are not referenced in the flow of data from program inputs to program outputs. It is this which leads to both variable length phenotypes and evolutionary neutral drift in the genotype. It should be noted that the genotype-phenotype mapping process in CGP incurs very little computational overhead as it is only carried out once per genotype. The $1 + \lambda$ -inspired algorithm promotes neutral drift since often mutational offspring differ only in inactive genes and so have equal fitness to their parent [69, 110, 119, 128, 129]. Such identical offspring can be detected before fitness evaluation so incurring no fitness calculation. Thus, even when it appears that the algorithm is stuck on a local optimum fitness value the search algorithm is exploring many possible different solutions by mutating different genotypes. The maximum allowed size of the genotype is highly influential here and Turner et al. investigated the relationship between it and performance on a suite of hard benchmarks [110]. They found that the relationship was approximately quadratic with a clear optimum value for n_n. Often the optimum number of nodes was quite large (hundreds or thousands of nodes).

2.4 Absence of bloat

CGP does not suffer from a phenomenon called bloat [105], in which over evolutionary time, programs become larger [91]. Bloat has been described as "program growth without (significant) return in terms of fitness" [80]. It was

thought that the lack of bloat in CGP was caused by either neutral genetic drift (NGD) [66] or length bias (LB) [18]. The former argued that since NGD is beneficial to search, it is beneficial to maintain a large number of inactive genes. This provides a pressure to limit the number of active genes. LB argues that the lack of bloat is a consequence of the genotype representation which has the property that nodes closer to the program inputs are much more likely to be active since they can be connected to by many nodes on their right. This causes a strong bias towards small program sizes. Interestingly, both these hypotheses for the lack of bloat in CGP have been disproved [105] where it was shown that when NGD is disallowed in CGP bloat still does not occur. LB cannot be the reason why there is no bloat since Recurrent CGP (see Sect. 3.7) has no length bias and yet it too does not exhibit bloat. Thus the cause of lack of bloat in CGP is still an open question.

3 Alternative genotype representations

3.1 Parallel distributed genetic programming

For completeness, and because of later discussion, we include a brief discussion of a form of graph-based genetic programming that has a representation that is similar to CGP. It is called parallel distributed GP. Poli proposed this in 1996 [77, 79]. Like CGP, PDGP represented graphs using a Cartesian grid of nodes. Connection to nodes were restricted to the previous layer (i.e. in CGP terms, with levels-back equal to 1) and the connection genes were relative (as in SMCGP see Sect. 3.5) and counted back from a node position to the source of node input. A pass-through (or wire) function was allowed so that nodes could be connected to deeper layers. Poli devised various crossover operators. They are all based around the idea of swapping sub-graphs defined by selecting random crossover points in the parents to create offspring. One can choose the first crossover point at random and the other must respect the limitation that the graphs have a maximum depth (in CGP terms, a maximum number of columns). By choosing crossover points either in either active or inactive nodes or both, he described a number of types of crossover. He also discussed two forms of mutation, one he called *global* in which a random sub-graph is generated and replaces an existing sub-graph. Actually, he states that global mutation was implemented by crossing over an individual with a randomly generated new individual [79]. The second mutation operator he called *link* which is the same as point mutation in CGP. Using an evolutionary algorithm with population sizes of around 1000 and tournament selection with crossover and both types of mutation Poli showed that PDGP was markedly more efficient at solving problems than standard tree-based GP (even with ADFs) and often the evolutionary algorithm was most efficient with small grid sizes. PDGP was also used to evolve artificial neural networks [81].

3.2 Modular

Koza demonstrated the usefulness of automatically-defined functions (ADFs) in tree-based GP [54]. Walker and Miller introduced an equivalent to ADFs for CGP called modules [122, 123]. In modular CGP sub-functions or modules can be captured or destroyed by mutation operators. Capture or acquisition happens via two random positions in the genotype. The genotype is extracted and placed in a module list and the removal site replaced by a module containing the re-labelled sub-genotype so that the meaning of the genotype is identical to the original genotype before module acquisition. Empirical comparisons of results with and without ADFs were made on a number of benchmarks and appeared to show that CGP with module acquisition and evolution obtained solutions in much fewer numbers of evaluations that standard CGP, particularly on hard problems. However, it should be noted that the comparisons with CGP were unfair in the sense that the total size of allowed genotype in nodes with modules was much greater than the number allowed in standard CGP. Thus, it still remains an open question as to whether module acquisition is beneficial.

Noting that in modular CGP modules were acquired or destroyed randomly (i.e. via mutation), Kaufmann and Platzner introduced some new techniques for creating modules: age-based and cone-based [38]. The age-based module creation operator identifies primitives nodes that have remained unchanged for a number of generations and places these into modules (only primitive nodes can reside in a module). The module is given an age that is the average of the ages of the primitive nodes within it. Two candidates are generated using this operator and the one that is older is chosen. In contrast to standard or age-based module creation, cone-based module acquisition (MA) aggregates only primitive nodes that are within a structure called cone (see [38] for details). Cones are a widely-used concept in circuit synthesis. They compare the computational effort (as defined in [53]) of the original modular CGP with versions that allow either age-based or cone-based MA on circuit synthesis and classification problems. They found that in almost all cases the age-based technique was superior to the original MA. However, cone-based MA largely proved superior only on circuit problems.

3.3 Real-valued

Clegg et al. devised a genotype representation for CGP in which all genes are floating point numbers in the interval [0.0, 1.0] [10]. We refer to this as a real-valued CGP representation (RVCGP). The motivation for devising this representation was that crossover in CGP might be more effective using a real-valued representation. We will discuss this in more detail in Sect. 5. Real-valued CGP has an additional decoding step in which a standard CGP genotype is obtained from the real-valued genotype. The real-valued genotype still has genes grouped by nodes consisting of a function gene and a number of connection genes.

Assume that the node function look-up table has n_f functions. Standard CGP node function integer genes, ig_f are obtained from the floating point gene, rg_f, by examining which of n_f segments of the interval [0.0, 1.0], rg_f lies within. This is accomplished with Eq. 2.

$$ig_f = floor(rg_f n_f) \tag{2}$$

For instance assume there are four node functions. Then if rg_f is less than 0.25 then $ig_f = 0$ while if rg_f is greater than 0.25 and less than 0.5, $ig_f = 1$, and so on.

Real-valued connection genes, rg_c corresponding to node address, n_p are decoded to standard integer connection genes, ig_c by dividing the unit interval into n_p segments. The equation that accomplishes this given in Eq. 3.

$$ig_c = floor(rg_c n_p) \tag{3}$$

For instance, suppose that $n_p = 6$, and there are two program inputs and four nodes. Then the unit interval is divided into as many segments as the node address (i.e. 6). If rg_c is between 0 and 1/6 then $ig_c = 0$, so that the first input to node 6 is the first program input (whose address is 0), if rg_c is between 5/6 and 1 then $ig_c = 5$ and the first input of node 6 is connected to the output of node 5 (the previous node). Output genes are decoded in the same way as the connection genes corresponding to node address of one more than the highest node address.

Using floating point values as genes for CGP has the potential to make the genotype more evolvable since when small changes are made to the genes the gene can move *gradually* to a value which will result in genotype change (assuming Gaussian mutation), whereas in standard CGP random mutations either abruptly moves a node input connection to an entirely different node, or changes the function of the node. Also, another motivation was to transform the discrete nature of CGP genotypes to smooth functions of n variables [10]. Finally, it allowed the prospect that much of the research into the optimization of real-valued vectors using evolutionary algorithms could be applied to CGP and hence the evolution of programs or other computational structures. Indeed, Clegg (see Sect. 5) used a form of crossover showing it appeared to be beneficial for symbolic regression problems. However, in his master's thesis, Turner [101] examined RVCGP on three additional classes of computational problems, digital circuit synthesis, function optimisation and agent-based wall avoidance. On these problems, it was found that RVCGP together with the crossover operation performed worse than standard CGP. However, he found that implementing the RVCGP representation but with the same selection and mutation methods as CGP gave equivalent performance to CGP.

Meier et al. interpreted the RVCGP genes as mean values in a multivariate Gaussian distribution [63]. From these distributions new genotypes can be sampled. They defined an operator called *forking* which decides whether a genotype should be interpreted either as a point or as a distribution in genotype space. The decision to fork uses population statistics based on an analysis of phenotypes (called fingerprints). Early in evolution an individual's phenotype is likely to be rare in the population, in which case the forking operator is more likely to interpret the individual as a point. As evolution progresses, individuals focus on fewer regions so that the current individual's phenotype may be shared by other individuals. In this case, its phenotype fingerprint

frequency increases and the forking operator will be more likely to interpret the current individual as a distribution, so that sampling happens more frequently. They evaluated their approach on four symbolic regression problems and found the forking operator reduced the number of generations to converge to high fitness and the computation time per run, as compared with standard RVCGP.

Walker et al. modified the RVCGP representation to encode transistor circuits [121]. Six floating point genes were used to specify each CGP node. Four genes defined the transistor characteristics while two specify the node inputs. A three-stage genotype to phenotype mapping was employed to obtain a valid circuit simulator netlists that could be simulated using the SPICE simulator.

Wilson et al. used a recursive variant of the real-valued representation to evolve highly effective Atari game playing agents in [125]. In addition, each node also had a parameter gene which used used by some node functions and was also used to weight the node output (as used in [52]).

3.4 Implicit context

In the Sect. 2 we saw that in CGP all nodes and inputs have an address which expresses where they are positioned in the genotype. Lones created a form of GP called enzyme GP in which the structure of a program is not given explicitly (as in most forms of GP) but is derived from connection choices made by each component of the program in a bottom-up fashion. He argued that in biological genomes the location of genes is relatively unimportant and criticized GP systems for being positionally dependent [60]. However, this point of view is at best debatable since some biological studies indicate that genes are located in positional neighbourhoods [12]. Furthermore, it has been discovered that a gene's location in a chromosome does play a significant role in shaping how an organism's traits vary and evolve [84].

Smith et al. [95] adopted many aspects of Enzyme GP and proposed a new representation of CGP called implicit context CGP (ICCGP). In enzyme GP and ICCGP program nodes are called enzymes. Each enzyme has a type (referred as 'activity'), a number of binding vectors ($\underline{b_1}$, $\underline{b_2}$), an output vector called a shape, $\underline{s}$ and a function gene (see Fig. 2). The elements of the vectors are integers in the range 0 to 255 (though in some implementations real numbers between 0 and 1 are used). Type merely indicates whether the enzyme is a program input, computational node or a program output. The number of elements that a binding or shape vector have is given by the sum of the number of external program inputs, n_i and the number of possible computational functions in the function look-up table, n_f.

As with CGP, the number of bindings (inputs) an enzyme has, is defined by the the arity of the enzyme function. Enzyme's of type 0 have only a shape as these represent external inputs and enzyme's of type 2 only have a single binding vector as these represent external outputs. An enzyme's shape is defined as:

$$\underline{s} = 0.25 * \underline{b_1} + 0.25 * \underline{b_1} + 0.5 * \underline{f} \tag{4}$$

where $\underline{f}$ is a vector of the same length as the binding vectors, but whose only non-zero element is 255 at position $n_i + f$ where f is the enzyme's function gene. The

multiplying constants ensure that the elements of the shape vector are in the range 0 to 255. The idea behind shape is that it represents the *affinity* that a component has to form connections with other components.

Genes are initialised randomly. An assembly process creates the phenotype. It starts at the type 2 enzymes (i.e. outputs). These bind to other enzymes (types 0 or 1). Binding is how connections between computational nodes occur. The binding vectors define whether and how strongly an enzyme can bind to another enzyme's shape. This is determined by the degree of matching between the binding of one enzyme with the shape of another. Input enzymes have a shape, outputs a single input binding. Connections between enzymes, inputs and outputs are determined by which bindings form the strongest match with shapes. The smallest difference between the elements of the binding vector and shape indicates the strongest match. Construction of the phenotype is similar to standard CGP. It begins with outputs binding to component or input shapes. Then these components bind and so on, until all enzyme's are bound. At this point a CGP-like phenotype can be obtained. The genotype has $L_{iccgp} = (n_i + n_n + n_o)(n_i + n_f)$ elements. Genotypes are mutated via point mutation and bindings and enzyme functions can be mutated at different rates.

The length of the genotype representation in ICCGP is much greater than in CGP as each enzyme has $(n_i + n_f)$ elements compared with $n_a + 1$ in CGP. For instance, suppose we choose 100 nodes or enzymes (a modest figure in CGP). Assume also that the problems we are trying to solve have many inputs (e.g. Boolean circuits with say 50 inputs and five outputs) and there are many possible node functions (say 16) then the genotype length would be $(50 + 100 + 5)(50 + 16) = 10{,}230$ genes compared with 305 in standard GP! As a consequence, ICCGP has only been applied with a very modest numbers of enzymes.

To date, as far as the author is aware, there has been only a single comparison of the computational efficiency of ICCGP with CGP and this was in the very limited context of the three input parity function over a range of rectangular topology parameters [8]. Thus, it remains unclear whether ICCGP offers any computational benefits. Despite this ICCGP has been extensively and successfully applied to various problems in medical diagnostics [56, 58, 59, 93, 94, 96–98].

3.5 Self-modifying

Self-modifying CGP introduced another type of node, a self-modifying (SM) node, one that refers to the code itself [26, 28, 29]. Such nodes modify the *phenotype*. For instance, a node may be a deletion operation, which deletes CGP nodes between two positions in the phenotype. Such operations imply that the phenotype is iterated in something akin to a developmental process. The process is as follows. The genotype is first duplicated to create the first phenotype. So here phenotypes mean genotype-like strings of numbers in the same format as CGP. The active self-modification nodes in the phenotype are applied one after the other to produce a new phenotype. This process defines one iteration. This is repeated until either there are no SM nodes in a phenotype or until a user-defined limit on the number of iterations is reached.

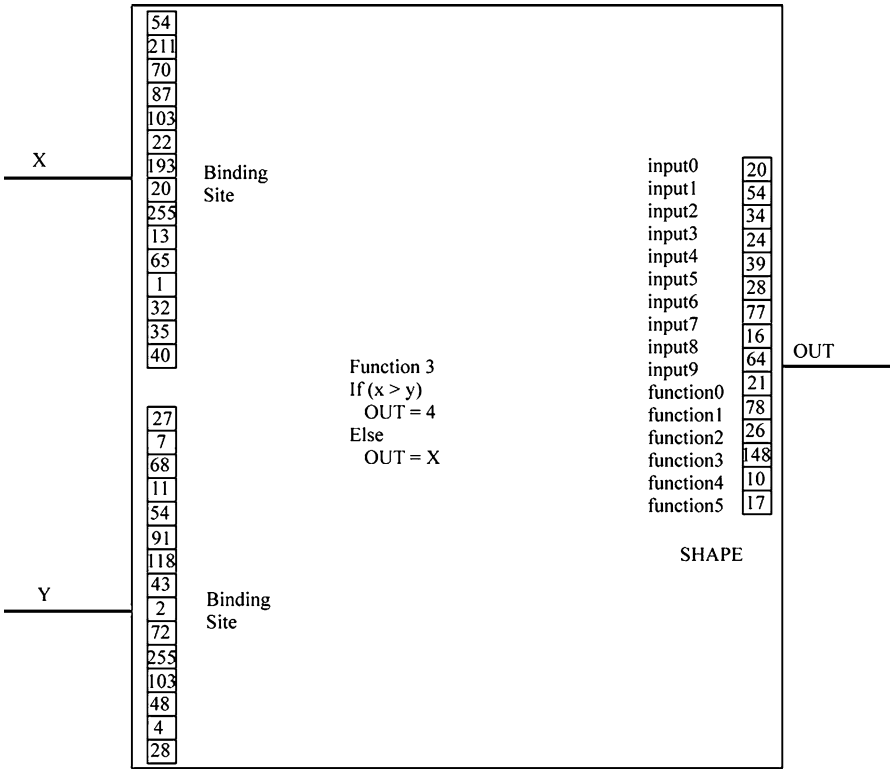

Fig. 2 In ICCGP nodes are called enzymes. An enzyme has two binding vectors and a shape vector. Each binding or shape vector has a number of elements equal to the sum of the number of program inputs and the number of possible node functions. The shape vector is a function of the binding vectors and a vector representing the enzyme function (Eq. 4). The binding and shape vectors determine how enzymes will bind to other enzymes, and hence, assemble the CGP graph. The computational inputs are X and Y and the enzyme has one computational output. In this example, the enzyme function is a simple if statement

The presence of SM nodes makes it more appropriate to replace absolute node addressing as in standard CGP with *relative* addressing. The latter is where connection genes count back from the node position. In addition, *functions* were introduced that provided external inputs (via a circular register) or wrote calculations to external outputs. This was done so that SM operations could change the number of inputs and outputs. SM nodes also require parameters which dictate where they operate in the phenotype (i.e. copy nodes between n and m and insert them at position p).

SMCGP can produce a sequence of phenotypes, each of which could solve a different computational problem (e.g. produce a series of parity functions of increasing number of inputs). This allows SMCGP to be applied to classes of problems that non-developmental encodings can not solve. Indeed, it has been shown that SMCGP could find provably general solutions to certain classes of problems: parity, binary addition [26], computation of π and e [27]

It is interesting to note that even without the presence of SM nodes, SMCGP introduced a possible modification of the standard CGP genotype representation (relative addressing and input/output functions). This was actually used in place of CGP in a number of papers [23, 24, 55]. However, to date, there has been no quantitative comparative studies of the two CGP representations and their efficacy.

3.6 Mixed-type

Harding et al. [23] introduced a form of CGP called mixed-type CGP (MTCGP) in which the data which flowed through the CGP graphs could have different types. In MTCGP node functions inspect the types of the input values being passed to it, and determine the most suitable operation to perform. This in turn determines the output type of the function. Node functions return a default value in cases where there is no suitable operation for the types being passed to the function. In MTCGP inputs can be variable length vectors of reals or individual values. Since MTCGP can handle multiple and mixed data types it allows a much wider range of node functions to be used. They used relative addressing and input gathering and output producing functions (as in SMCGP).

MTCGP was applied to a number of classification tasks in the UCI Machine learning repository: Wisconsin breast cancer, phoneme, diabetes, and heart datasets. For each of the classification tasks, the inputs presented to the program were both the entire attribute vector as well as the individual attribute values of the vector. They used a large set of primitive functions ranging over list processing, mathematical and statistical and compared results with a suite of well-known classification methods producing highly respectable results. Often also, the evolved CGP programs were very small and readable.

Wilson et al. in their work on evolving human-readable Atari game playing programs, extended multiple data types handling node functions to include also a matrix type [125].

3.7 Recurrent

CGP is usually described as a representation of acyclic graphs. However, it is straightforward to extend it to recurrent or cyclic graphs [108, 109]. To do this, the restriction that connection genes of a node must have values less than its position is lifted, so that it can connect to itself and any other node. The genotype is decoded in a very similar way to standard CGP to produce a list of active nodes. First the active nodes need to be determined. This can be done recursively from output to inputs in the usual way for CGP except that one only adds new nodes to the active list, not ones already present (this breaks cycles). After this the following steps are carried out

1. set all active nodes to output zero
2. apply the next set of program inputs
3. update all active nodes *once* from program inputs to program outputs

4. read the program outputs
5. repeat from 2 until all program input sets have been applied.

Of course, in step 3 one could choose to update active nodes more than once and use some function of the output values (i.e. the average). However, there would always be the issue of when to stop.

To control the numbers of recurrent versus non-recurrent connections, Turner et al. introduced a additional parameter called recurrent connection probability which controlled the likelihood of recurrent and non-recurrent connections. The performance of RCGP was compared with CGP on a number of benchmarks [108], the Artificial Ant, Sunspot prediction and a number of integer sequences [109]. In all cases RCGP outperformed CGP significantly. In addition RCGP outperformed various published methods for the Fibonacci sequence. RCGP should be regarded as "standard" CGP in that when the recurrent connection probability is zero it becomes the original form of CGP.

3.8 Iterative

In iterative CGP (ICGP) conditional cyclic loops can be represented [86]. To accomplish this a linear CGP geometry is assumed (i.e. one row and multiple columns) with *levels-back* set to be the number of columns. All nodes have four genes. The first gene is a node function gene, the next two are connection genes and the last gene represents a Boolean condition which determines whether a loop should be continued or terminated. The first connection of a node always refers to a node previous to the node (i.e. a standard feed-forward CGP connection). The second connection gene is either ignored or represents a cycle. The gene is ignored if it refers to a previous node or input. However, it may refer to a subsequent node (or itself) in which case it defines a loop. The nodes between the position of the calling node and the forward connection are then in a cycle. The condition gene is an address in a look-up table of possible loop exit conditions. If it exits then the next instruction is decided by the node immediately after the end of the loop, if it does not exit it executes the instruction in the next node on the right. There is a single output gene (O_A) which points to the last executed node (10). An example genotype is shown in Fig. 3.

In the example, the nodes are executed in the following order: 1, 2, 3, 6, 7, 2, 8, 1, 9, 10. We have assumed that on the second call of node 2, condition 1 is met, so execution passes to the next node after the loop (node 8) and thence to 1. We also have assumed in this example that loop condition of node 1 (2) is also met thus causing program execution to move to the next node (9) after the loop terminates at node (8). There are some rules required to enforce valid loops:

1. For any nodes inside an existing loop, branching genes can only connect to either any previous node or input (acyclic) or a node with a higher index that is *inside* the current loop (cyclic). For instance, in Fig. 3, the branching gene of nodes 3–6 can be valid if its value is lower than 7. However, any branching genes greater

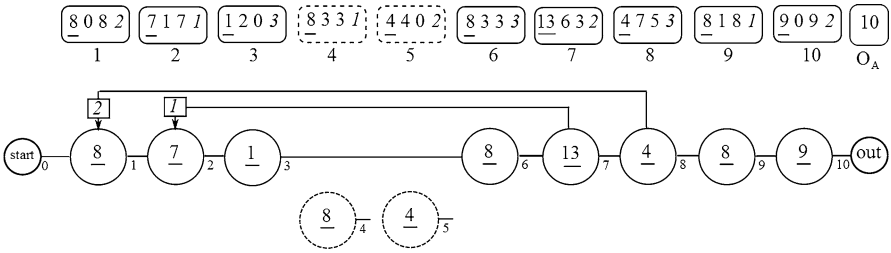

Fig. 3 An example of an iterative CGP genotype. There are ten nodes each with four genes. The node function is shown underlined. The second gene represents a feed-forward connection. The third gene can either refer to a previous node (in which case it is ignored) or refers to a subsequent node in which case it represents a cycle or loop. The fourth node gene (in italics) is the address of a Boolean loop exit condition. It is shown on the cycles. In this example there are two nodes that are not referred to so are ignored (shown dashed)

than the node label would create loops within the already existing loop starting at node 2 and ending at node 7.

2. For any nodes outside an existing loop, their branching genes can only connect to a node that is *outside* any existing loops. For instance, because the nodes 2–7 are already in a loop, the branching gene of node 1 can only point to either the input or nodes 8, 9, or 10.

Ryser-Welch applied iterative CGP to three classes of problems: travelling salesman, mimicry and nurse-rostering [85, 86]. For these problems, the node operations were very sophisticated and applied existing heuristic algorithms rather than merely a mathematical function of input data. In this way it was shown that new human-readable, standalone problem-solving algorithms could be produced.

3.9 Differentiable CGP

Genetic programming is normally considered as a derivative-free method, however in an impressive paper Izzo et al. [33] show how it is possible to obtain a complete representation of the differential properties of a program encoded by a genetic programming expression. They applied this to CGP creating a new form of it called called differentiable CGP (dCGP). In dCGP all connections have weights (like artificial neural networks) and node functions are represented as truncated Taylor expansions of a given order. They show that this allows arithmetic operators +, −, *, / to be extended to operate on truncated Taylor expansions. For instance, using this idea they show that a CGP encoded function $O_0 = sigmoid(yz + 1)/x$ can be written also as second order Taylor expansion of differentials,

$$\begin{aligned} O_0 = {} & 0.881 - 0.881dx + 0.105dy + 0.105dz + 0.881dx^2 \\ & - 0.0400dy^2 - 0.0400dz^2 - 0.105dxdz + 0.025dydz - 0.105dxdy \end{aligned} \tag{5}$$

The dCGP approach means that not only can the output function of a CGP program be computed at a given point but also all its derivatives up to a given order.

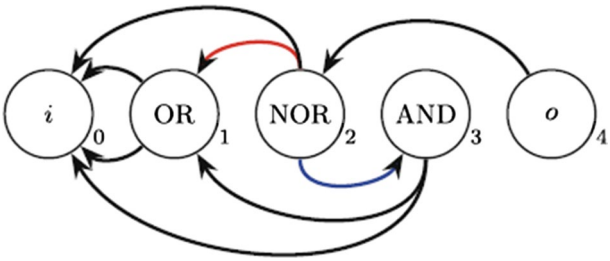

Fig. 4 Example of illegal CGP mutation but allowable graph transformation of a Boolean function. An EGGP mutation which changes a connection (red) from node 2 to node 1 is replaced with a connection (blue) directed to node 3. This mutation produces a valid circuit but is impossible in standard CGP as mutations have values less than the node position (to guarantee having a feedforward) property Taken from [7] (Color figure online)

Their approach makes it possible to apply concepts such a back propagation to CGP encoded programs.

They evaluate the dCGP technique on a suite of varied symbolic regression problems in which π and e appear. Newton's method was used to back-propagate the errors on ephemeral constants in evolved expressions to determine the constants in the Taylor expansions. The fitness is the final error. They showed that they could evolve the target formulae exactly (with all weights set to 1). However, they note a drawback to this method: the number of ephemeral constants used as additional terminals needs to be pre-determined. However, by associating weights to every edge of the graph, the differential properties of the error with respect to weights can be determined. This allows symbolic regression to be carried out with no extra terminal inputs but the values for all the weights has to be determined. Since there can be many weights, the dCGP method selects random weights and then iteratively back-propagates the errors. They used a $1 + 4$ evolutionary strategy to evolve the dCGP expression and interestingly, each offspring, $i = 1$ to 4 received i mutations. They also showed that dCGP could be applied to solve partial, ordinary differential equations and to search for expressions that are prime integrals of sets of differential equations

3.10 Graph programming

Recently, an interesting new method of evolving graphs has been proposed. It is called evolving graphs by graph programming (EGGP) [7]. This method evolves graphs directly rather than using linear or grid-based genotypes. Using a probabilistic extension to the graph programming language, GP2, Atkinson et al. are able to evolve graphs. They show firstly that any CGP individual can be represented as an EGGP individual, whereas the converse may not always hold when the number of rows in a CGP individual is greater than one. Secondly they note that some feed-forward preserving mutations in EGGP are not possible in the standard form of CGP. Figure 4 shows how an allowable mutation in EGGP is not allowed in CGP. The

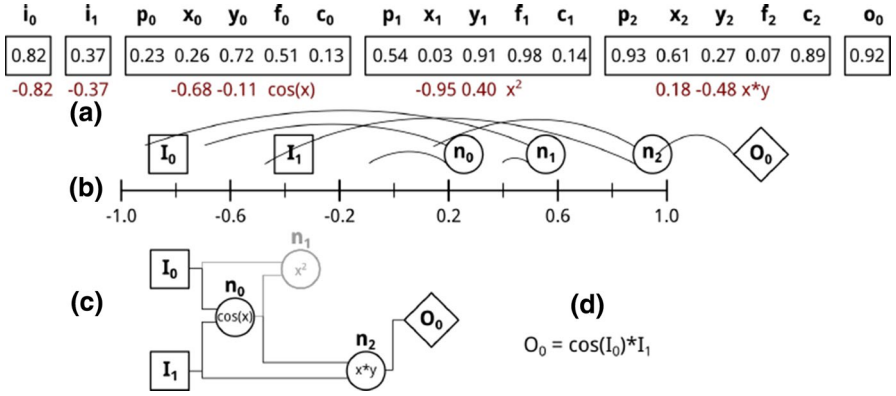

Fig. 5 Example of PCGP genotype. A PCGP genome (**a**), including input i_n and positional p_n genes. These are translated to input and node positions (**b**) and connection positions "snap" to the nearest node. As in CGP, a resultant graph (**c**) and output program (**d**) are then extracted. Taken from [126]

EGGP approach is shown to significantly out-perform CGP on a collection of circuit benchmarks (particularly the harder benchmarks).

However, it should be noted that since recurrent CGP allows both feedforward and feedback connections the mutations which are illegal in standard CGP but possible in EGGP are also legal in recurrent GP. It may also be possible to introduce a new mutation operator to make these currently illegal mutations possible in standard CGP.

3.11 Positional CGP

Positional CGP [126] is an interesting new real-valued representation of CGP in which inputs, and nodes have *evolvable* positions. All node genes are floating point numbers in range [0, 1], which correspond to the connections of each node n, x_n and y_n, the node function f_n, and a parameter gene c_n which can be used to weight the output of nodes or act as a parameter of a function. In addition, every node has a position gene p_n which determines the position of the node. Connections are formed by converting the connection genes x_n and y_n to coordinates by multiplying the genes by the node position, p_n and then "snapping" these connections to the nearest node. Inputs also have evolvable position genes, i_n but are constrained to the interval $[-1, 0]$. Output genes do not have positions and take values in the interval [0, 1]. A small example of the PCGP representation is shown in Fig. 5.

Since node positions are evolved, it is highly unlikely that two will occupy the same position, even between different genotypes. Furthermore, over evolution, nodes which are connected can have positional genes and connection genes which are highly related. Finally, a node's connection positions depend only on its position, which is in its genes, rather than the node's placement in the genotype. This allows node genes to be exportable; the same genes in a different individual will form connections in the same place. If multiple genes are exported together, entire sections of the graph can be migrated between individuals. In PCGP, nodes can be added or

removed from a genome without disturbing the existing connection scheme, unlike in CGP, where a node addition and deletion causes a shift in all downstream node positions. Experiments were carried out comparing PCGP with CGP on nine benchmarks (three each of classification, symbolic regression and reinforcement learning). Results showed PCGP to be superior to RVCGP on classification and regression benchmarks but inferior on reinforcement problems.

3.12 Genotype-phenotype complexity

It is important to realise that translating a CGP genotype to a phenotype is done once per genotype. Evaluating the fitness of a genotype is expected to far outweigh this decoding time. However, on problems where fitness evaluation is very fast, the decoding time for different CGP variants may be significant.

The decoding of genotype to phenotype in standard and PDGP is straightforward. One begins at the output nodes and recursively activates nodes that are required until inputs are encountered. Along the way, a list of active nodes and their connections are stored for later use (i.e. the phenotype). This decoding step only needs to be carried out once per genotype. Data is only presented and fitness calculated using the stored phenotype. In modular CGP, the decoding is more complex as the phenotype is a collection of CGP programs consisting of the main program and the CGP code of all the modules. When modules are captured, the captured code is re-written so that module code refers to its inputs in the same way as standard CGP refers to program inputs. Also when modules are destroyed, the modular code has to be translated back into standard CGP format. This "book-keeping" increases the time complexity of the decoding step. A large number of modules would increase the time taken for the decoding step. However, it is expected that for problems where modules are beneficial the evolution time would be shortened. The decoding step of RVCGP is little different from standard CGP, one merely requires an initial pass to convert the real-valued representation into standard CGP. In positional CGP one also needs to identify which nodes are located closest to the positions of the inputs to the nodes, this is an extra step so will increase genotype-phenotype decoding time. Self-modifying CGP requires self-modifying operations to be applied to the old phenotype to generate the new one. This obviously adds to the genotype-phenotype decoding time. However, by using a limited list of self-modifying operations one can control this complexity to acceptable limits. The decoding step in recurrent and mixed-type CGP is little different to standard CGP unless in the former case one recurses over the genotype multiple times (which is not done in practice, see Sect. 3.7). The complexity of the genotype-phenotype mapping in iterative CGP is slightly more complex due to branching conditions. Differentiable CGP has a genotype phenotype decoding that is little different from CGP, however gradient descent of weights would certainly increase the time required for fitness evaluation. Implicit context CGP as we have seen requires much larger genotype sizes and also undergoes matching operations to find out which components bind to each other, so genotype-phenotype mapping time would be longer than standard CGP.

4 Mutation

In standard CGP mutation either point or probabilistic mutation can be used. In the former, the user decides the percentage of the total number of genes of a parent genotype to be mutated to create an offspring. In probabilistic mutation every gene is considered for mutation according to a user-defined probability. Point mutation is easier to implement and more efficient than using a probabilistic mutation as one does not need to linearly walk through the genes to decide which ones to mutate. However, choosing a discrete number of genes to mutate means that only certain mutation rates can be chosen. Probabilistic mutation is continuous and very low mutation probabilities can be chosen and investigated. It is also sometimes useful to use different mutation rates for different classes of genes (connection, function, output). However, the utility of this has not been investigated in detail.

One of the very interesting aspects of the CGP representation is that a large variety of phenotypes can be found by applying mutation. Since many genes in CGP are redundant, often mutations occur only in the redundant regions, which means that the mutated genotype has the same phenotype as its parent. In such cases one does not need to carry out a fitness evaluation (see below). However, other mutations might change an output gene. In this case the program output comes from a formerly redundant node, which in turn may connect to previously redundant genes. This can cause large changes in the phenotype.

Goldman and Punch compared several mutation strategies on a range of circuit benchmarks [19]: *Normal*, *Skip*, *Accumulate* and *Single*. *Normal* is just standard mutation (probabilistic) with no check for offspring having identical genotypes to their parent. *Skip* checks offspring to see if the phenotype is identical to the parent (by comparing active genes) and if so returns the fitness of the parent. *Accumulate* continues to mutate an offspring until some of its active genes are different from the parent. *Single* mutates the offspring until one active gene is changed. They found that the performance of *Skip* and *Accumulate* were fairly insensitive to mutation rate whereas *Normal's* performance was very sensitive to mutation probability. Overall, the performance of *Single* which has no mutation parameter was close to the best performance of the other strategies.

There is a considerable length bias in standard CGP in favour of small phenotypes [18]. In addition there is a strong positional bias in CGP in that it is much more likely that nodes on the left side of the genotype (i.e. close to the inputs) will be active. This is simply because the inputs of *any* node on the right of a given node are allowed to connect to it. For instance the first node can be connected to by the input of any node on its right. While the penultimate node on the right can only be connected to by either the last node or an external output. These biases mean that the location of inactive nodes in the genotype are not distributed evenly and nodes toward the right (towards the outputs) are likely to have many inactive nodes between them.

In a comprehensive and detailed study of CGP, Goldman and Punch investigated two strategies for compensating for positional bias [20]: *Reorder* and *DAG*.

In the former, prior to mutation of a parent, the position of nodes in the parent are shuffled without changing the semantics of the phenotype. Reorder is used once each generation to shuffle the nodes of the parent's genome. As the shuffling does not semantically change the parent, it does not require re-evaluation. Reorder causes active nodes to be, on average, more likely to be located halfway between the input and output ends of the genotype. The DAG strategy allows connections to be feedforward or feedback (as long as no cycle is created) and removes the positional bias in CGP. In an extensive series of experiments they concluded that *Reorder* had the best performance overall. Of course, this should also be combined with *Skip*. Further, when Reorder (or DAG) was not used, the number of nodes *never used during evolution* was between 43% or higher (depending on benchmark problem)! They also found that large numbers of nodes compute a constant value irrespective of the program input. They suggested also that the advantage of large amounts of inactive nodes is purely that they provide mutation with random nodes for exploratory purposes (i.e. useful previously found substructures are *not* stored in inactive code). This accords with the author's unpublished finding that randomising inactive genes prior to mutation appears not to cause any performance impact. It would be interesting to reduce the rate at which inactive nodes were randomised by mutation to see if allowing memory of past useful structures in inactive code gives some advantages.

It would also be interesting to investigate how mutation could be defined so that the chance of any node being connected would be equal for all nodes. This would require mutation to be dependent on position so that the connection gene is much more likely to be changed to connect to nodes that are nearer to it. A simple way to achieve this would be to record the number of mutations that have occurred so far at each node location during an entire evolutionary run. A new mutation could be restricted to choosing to mutate only the least mutated node. This would ensure that any node was equally likely to be connected to.

Kalkreuth [34] introduced an interesting new pair of mutation operators into standard CGP: inactive node activation and active node deactivation. He referred to these as "insertion" and "deletion" respectively. An insertion mutation chooses an inactive node and changes one or more connection genes in the genotype to make it active. Conversely, deletion alters connections to an active node so that the node becomes inactive. He examined the impact of the new mutation operators (operating together with the standard point mutation) on three Boolean benchmarks and a suite of symbolic regression problems. On all problems, the two operators gave improved performance. More detailed studies are required to confirm and strengthen these findings.

Vašíček and Sekanina made a breakthrough in the optimisation of digital circuits by showing how circuits with many inputs could be optimised using CGP by starting with a state-of-the-art logically correct reference circuit and employing a SAT solver algorithm to decide if an evolved circuit was logically correct. If it is incorrect it receives a fitness of zero, if correct the fitness is the difference between the genotype size in nodes and the number of gates utilised. This technique has allowed the optimisation of industrial-sized digital circuits [120].

In this technique, Vašíček [120] discovered the surprising result that there is no need to utilise inactive nodes in CGP when optimising fully functional digital circuits! He defined a "zero neutrality" search process by ensuring that mutation respects the following conditions: (a) inactive gates are unchanged (b) active gates (or primary output) can not connect to an inactive gates and finally, (c) the second input connection of single-input gates are not mutated. He compared the performance of this mutation against standard CGP on a suite of 100 industrial-sized circuits and found that the performance of the two were statistically indistinguishable. This needs more investigation on a wider set of problems.

The genotypes in the initial population were exactly the size of the circuits synthesised by the ABC algorithm.[2] In other words there was no redundancy in the initial circuits and redundancy could only arise by some gates not being required (i.e. by successful optimisation). It would be interesting to see what the results would have if the initial genotype had been larger than that produced by ABC by adding inactive gates. Vassilev et al. [118] found that smaller three-bit multipliers were found by encoding a working multiplier with a small amount of redundancy. This is also supported in the work of Gajda and Sekanina [16] who discovered while optimising digital circuits that in long evolutionary runs strictly selecting circuits which had fewer gates did not produce as small circuits as merely choosing circuits that were functionally correct (that is with lower fitness). They showed that the frequency of occurrence of correct circuits was greater using the weaker fitness criterion. This highlights the importance of neutral drift for exploration and the importance of maintaining redundancy.

5 Crossover

In standard CGP crossover is not used. In his original paper on CGP [65], Miller found that crossover appeared to have little effect on the efficiency of CGP and for the most part, subsequent work ignored crossover.

In real-valued CGP Clegg et al. [10] used a simple method of real-valued genotype crossover called "flat" crossover [82] which generated two offspring, o_i from two parent genotypes, p_i using Eq. 6, where $0 < r_i < 1$ is randomly generated and $i = 0, 1$

$$o_i = (1 - r_i)p_1 + r_i p_2 \tag{6}$$

Kalkreuth et al. [36] investigated RVCGP using adaptive crossover, mutation and selection. Adaptation of these operators is based on maximising the diversity of *phenotypes* in the population. They compared their technique with Clegg's RVCGP on a number of symbolic regression problems showing that the new approach improved performance.

[2] Berkley Logic Synthesis and Verification Group: ABC: A System for Sequential Synthesis and verification. http://www.eecs.berkeley.edu/*alanmi/abc/.

Walker et al. devised a multi-chromosome representation which could be applied to special classes of problems [124]. They looked at seven multiple output digital circuit problems and instead of allocating as many output genes as circuit outputs they divided the genotype into as many chromosomes as the number of outputs. For the problems chosen (all digital circuits) the fitness of each chromosomes could be assessed independently. The parent genotype in the $1 + \lambda$ EA was constructed by choosing each of the fittest chromosomes. For all benchmarks this produced spectacularly better results than the one chromosome version. Indeed for one problem it was possible to evolve a solution approximately 392 faster than using a single chromosome with the same total number of nodes.

However, most computational problems do not have the property that multiple outputs can be evaluated for fitness independently. It would be interesting to investigate a multi-chromosome form of CGP in such cases. One way this could be done would be to allow multiple chromosomes each providing a single output and also having an additional "coordinator"chromosome which uses the outputs of the other chromosomes as inputs. Fitness would be determined from the coordinator chromosome only. This would allow parent genotypes to produce offspring by crossover of non-coordinator chromosomes. Indeed, the coordinator chromosome would be free to utilise any or all of the non-coordinator chromosomes.

Slaný and Sekanina examined how various crossover operators and standard mutation affected the smoothness and ruggedness of the series of fitness values of the best population member in each generation. They examined this in the domain of CGP applied to image filter design. They found that either point mutation (with $\lambda = 7$) or single-point crossover where only one offspring of the crossover operation is mutated and included into the new population, generated the smoothest fitness landscape.

In their work on new techniques for acquiring modules in modular CGP (see Sect. 3.2), Kaufmann and Platzner also investigated a cone-based crossover operator to be used with a genetic algorithm (rather than $1 + \lambda$-ES). This generates a recombined chromosome by transplanting a cone of a donor chromosome into a clone of a recipient chromosome. They investigated the utility of the crossover operator using various population sizes using a genetic algorithm. They found smaller population sizes worked best in the new approach but it only performed better than the original modular CGP technique on two of the six benchmark problems (two and three-bit multipliers).

Kalkreuth et al. have recently investigated sub-graph crossover in CGP [35]. Sub-graph crossover is like single-point crossover except that the active nodes both sides of the crossover point are preserved. The crossover point is always chosen so that it lies between nodes. This idea of sub-graph crossover is that it should reduce the disruption caused by single-point crossover in standard CGP and truly recombine meaningful sub-graphs. If after single-point crossover active genes would change then the connection genes on the right of the crossover point are randomly re-generated to preserve the active genes on the left of the crossover point. They used a standard genetic algorithm with population size 50 and tournament selection. They evaluated the utility of the new crossover operator for a range of crossover rates on a suite of benchmark problems in circuit design, symbolic regression and image filter

design. Often high crossover rates were beneficial and in all cases a nonzero crossover rate performed better than without crossover. However, they did not compare their results with standard CGP so it still remains unclear whether crossover has utility over purely mutational CGP.

Kalkreuth and Husa have recently proposed *block* crossover [32]. This is defined using the one-dimensional representation of CGP. First before carrying out block crossover blocks must be identified in two parent genotypes. Blocks are groups of nodes that meet the following criteria: (1) The block contains a desired number of nodes, (2) All nodes in the block are directly linked through their inputs or outputs, (3) All nodes in the block are part of the genotype's active path. Block crossover then randomly selects one block from each genotype and swaps them. The position of the nodes transferred as part of the block may change inside the new genotype. However, their mutual links are preserved and the function performed by the block stays the same. After block crossover, point mutation is applied. They conducted parameter sweeps with standard CGP $(1 + \lambda)$ versus genetic algorithms using block crossover on a suite of Boolean functions and symbolic regression problems. They found that there was no single set of crossover parameters that worked best over the problems. They also found that the best value of λ varied with problem. Although it is possible for crossover operators to outperform the standard $1 + \lambda$ strategy, if both methods have their parameters fine-tuned, the $1 + \lambda$ strategy usually remains as the overall best strategy.

6 Search algorithms

6.1 Evolutionary strategy and hybrid algorithms

Although the most usual search algorithm used in CGP is $\mu + \lambda$ evolutionary strategy with $\mu = 1$ and $\lambda = 4$, sometimes larger values of λ have been chosen. For instance $\lambda = 14$ [16] was used in digital circuit optimisation and $\lambda = 8$ in the evolution of image filters [88].

Recently Milano et al. [64] examined the robustness to mutation and the evolvability of CGP genotypes encoding digital circuits. Mutational robustness is capability of a system to preserve its functionality after mutations. Evolvability is the likelihood of producing adaptive heritable phenotypic variations as a result of mutation (we are not considering crossover). Their experiments used a 20×20 array of two input logic gates (AND, OR, NAND, NOR) and the aim was to find a functionally correct even-5 parity function (a moderately difficult task).

They studied two $\mu + \lambda$ evolutionary strategies. The first uses $\mu + 1$ and the second is $1 + \lambda$. In $\mu + 1$ each of μ parent are mutated to produce a single offspring, then the best μ individuals from the 2μ genotypes are chosen as the new population. Having $\mu > 1$ allows the population to have a greater genetic diversity, since multiple mutational searches take place around the multiple parents. However, they observe that in this method the offspring of individuals that are more robust to mutations have more chance to be selected than the offspring of individuals that are less

robust. They show that this drives evolution to produce small phenotypes and it has markedly worse performance than $1 + \lambda$.

They proposed a new algorithm which they call a Parallel Stochastic Hill Climber (PSHC) is a combination of $\mu + 1$ and $1 + 1$. The aim is to preserve population diversity while maintaining strong selection pressure. In this algorithm each of the μ parents, p_i is adapted using a $1 + 1$ evolutionary strategy for a certain number of steps. The best candidate solution obtained after this phase, $\acute{p_i}$ is then used to replace p_i and, also with a certain low probability, the worst individual of the population. Additionally, they allow a certain degree of stochasticity in the $1 + 1$ phase by adding a random variation to the fitness, however at the end of this phase the decision whether or not to replace p_i by $\acute{p_i}$ is on the basis of the true fitness.

They found that performance of the PSHC algorithm is statistically significantly better than the $1 + \lambda$ algorithm.

Hybrid search algorithms have also received attention in the work of Kaufmann and Platzner [39]. They investigated in particular hybrids of $1 + \lambda$ (as a form of local search) with either of the multi-objective algorithms, NSGAII [13] and SPEA [131]. They found that for some multi-objective benchmarks and for all digital circuit benchmarks the hybrid algorithms outperform NSGAII and SPEA.

6.2 Simulated annealing and unusual $\mu + \lambda$ evolutionary strategies

Kaufmann et al. recently conducted a large and detailed empirical study of the performance of CGP various $\mu + \lambda$ with a wide range of values of μ and λ and also looked at simulated annealing (SA) [37]. They also used Iterated Race for Automatic Algorithm Configuration (iRace) [61] to optimise CGP parameters, n_c, n_r, μ, λ and the mutation rate. They found that for harder digital circuit problems (e.g. 3-bit multiplier, and even-parity with higher than six inputs) SA was the best performing search algorithm. SA worked best with unusual topologies (e.g. 300 columns and 10 rows, or 150 columns and 8 rows). They also examined the performance of various $\mu + \lambda$ strategies on a suite of symbolic regression problems where they found that large values of μ and λ performed very well (e.g. $\mu = 22$, $\lambda = 4096$). However, unfortunately, they did not choose a fixed maximum number of nodes ($n_n = k$, where k is a constant). n_n is an extremely important parameter for CGP. It is easy to see why. Choosing n_n to be small constrains neutral drift markedly and it is well understood that neutral drift in CGP is an extremely important search mechanism (Sect. 2.3).

6.3 Balanced CGP

Yazdani and Shanbehzadeh [127] pointed out that CGP does not have any possibility of sharing information among solutions. To achieve this they suggested incorporating features from biogeography-based optimization (BBO) [92] and opposition-based learning (OBL) [100]. They applied BBO's migration operator which allows information sharing between individuals. They also introduced a new mutation operator inspired by OBL. They evaluated their approach on five symbolic regression

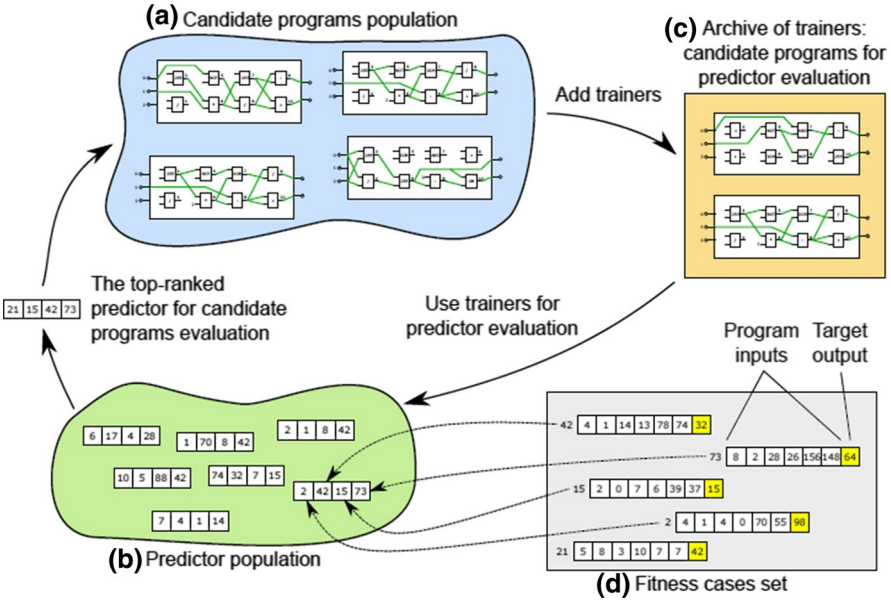

Fig. 6 Coevolution of populations of CGP programs (**a**) and fitness predictors (**b**) [14]. Some of the objective fitness evaluations are replaced with an alternative fitness calculated using a fitness predictor. The population of solutions is evaluated with the current best fitness predictor. The population of fitness predictors evolves to minimize the difference between the true fitness and the predicted fitness. An archive of fitness trainers (**c**) is used by the predictor population to evaluate evolved fitness predictors. Fitness predictors are represented using variable size array of pointers to elements of a subset of the training data (**d**)

problems. The found that the proposed BCGP method outperforms traditional CGP in terms of accuracy and the convergence speed.

6.4 Coevolution

Šikulová et al. [14, 90] developed a combination of fitness prediction with coevolution in CGP to reduce the number of expensive full fitness evaluations. The method replaces some of the objective fitness evaluations with an alternative fitness calculated using a fitness predictor. The fitness predictors are coevolved in a second population in order to provide accurate fitness predictions. The population of solutions is evaluated with the current best fitness predictor while the population of fitness predictors evolves to minimize the difference between the true fitness and the predicted fitness when measured using the current population of solutions.

Fitness predictors are represented as an adapted variable size array of pointers to elements of a subset of the training data. The population of candidate programs are evolved with the usual $1 + \lambda$ evolutionary strategy and the population of fitness predictors are evolved with a simple genetic algorithm. They also use two archives one of fitness trainers and one containing the best evolved fitness predictors. The archive

of fitness trainers is used by the predictor population for the evaluation of evolved fitness predictors. It contains copies of selected candidate programs obtained during the evolution. The fitness predictor from the other archive is used to evaluate candidate programs. The overall methodology is depicted in Fig. 6.

Drahosova et al. [14] evaluated their method on suites of varied symbolic regression problems and image filter design problems. They showed that the coevolutionary approach outperforms standard CGP in terms of CPU time required to converge.

7 Acceleration of CGP

Vašíček and Slaný developed an efficient acceleration technique designed to speedup the evaluation of candidate solutions in CGP. The method translates a CGP phenotype to machine code that is consequently executed. An attractive feature of the method is that the translation mechanism requires only a marginal knowledge of target CPU instruction set. They exemplified the technique by applying it to a symbolic regression problem. It was shown that for a cost of small changes in a common CGP implementation, a significant speedup can be obtained even on a common desktop CPU.

Vašíček and Sekanina have also accelerated CGP for circuits, symbolic regression [113] and image filter design [114]. In the former, they designed and built an application-specific virtual reconfigurable circuit (VRC) and fitness unit, obtaining speedups of 40 times optimized software implementations. In image filter design CGP was implemented on a single FPGA and gave a significant speedup (170) in comparison with a software implementation.

Coevolutionary CGP (Sect. 6.4) has also been implemented in dedicated hardware (FPGA) achieving 58 times speedups over highly optimized software implementations [31] .

Cartesian genetic programming (CGP) was one of the first genetic programming representations to take advantage of the general purpose computing capabilities of modern graphics processing units (GPUs) [22]. The implementation of CGP on GPUs was benchmarked on regression and Boolean problems and when compared to a naive, CPU-based C# implementation, was able to execute evolved programs hundreds of times faster. Harding et al. also implemented CGP on clusters of GPUs. This made it possible to evaluate the population in parallel, which in turn increased the speed at which the population could be evaluated [25].

8 Applications

Here we highlight some of the major application areas of CGP which are likely to continue to grow in the future.

Vašíček gives a thoughtful and insightful analysis concerning some of the problems that remain in moving CGP digital synthesis to industry [112]. He notes that when using CGP to optimise logically correct circuits a large number of new candidate solutions need to be generated and evaluated before a new candidate is found

with the same or better fitness (about 180 on average). He attributes this to the fact that random point mutation is very inefficient and that some form of mutation which is invested with domain knowledge could be much more efficient. Perhaps one can define mutation operations that replace small sub-structures in a phenotype with alternative but logically equivalent sub-structures?

In the synthesis of approximate circuits and programs Vašíček and Sekanina opened up an area of research which is rich with research opportunities [89, 116]. In many application areas (e.g. digital filters, artificial neural network) one can trade-off accuracy with power consumption, speed, or other properties of digital circuits. However, to evolve large-scale approximate circuits with CGP requires an efficient method to determine the approximation error. They solved this using an equivalence checking algorithm operating over binary decision diagrams (BDD) [117]. Recent work has used CGP successfully to synthesize approximate circuits with formal guarantees of approximation error [9]. Impressively, they have been able to synthesise aproximate 32-bit multipliers and 128-bit adders.

CGP has been successfully used in the design of various kinds of image filters [24, 74, 88]. Sekanina pioneered the use of CGP for image filter design [87] and showed that image filters could be automatically designed that were competitive with conventionally designed filters both in image quality and hardware cost. Harding demonstrated the advantages of employing extensive domain knowledge in CGP. He used a large function set (over 50) which included many Open Computer Vision operations.[3] Using an island-based parallel implementation of CGP he examined its performance on various image related problems: noise reduction, recognising cell mitosis[4] and object recognition for the iCub Humanoid Robot. He obtained excellent results on all three problems and in particular the evolved iCub robot image filters were often extremely small and could track moving objects in live video stream under a range of lighting conditions. CGP using OpenCV is now being used commercially to detect faults in machined metal.[5]

Artificial neural networks can be easily encoded in CGP by allowing all connections to have weights [46, 48, 49]. These are referred to as CGPANNs. Evolving ANNs with CGP allows the benefits of the CGP representation and techniques to be carried over to the evolution of neural networks (e.g. heterogeneous neural activation functions [107]). Also, it has been shown that evolving ANNs with CGP produced networks whose performance is much less sensitive to topology choices than fixed topology evolved ANNs [104]. Comparisons of CGPANNs with other neuroevolutionary methods on reinforcement learning tasks consistently show them to be competitive with many other neuroevolutionary methods [103, 111]. Recurrent CGPANNs also show promise in time-series prediction [111]. It should be noted that, in his PhD thesis, Turner [102] used R package implementations[6] of many machine learning techniques and compared their performance with CGPANNs on

[3] https://opencv.org/.

[4] This was a competition at the 2012 International Conference on Pattern Recognition (ICPR).

[5] machineintelligence.co.uk.

[6] caret: Classification and Regression Training, 2014. R package version 6.0-37.

classification problems. This appeared to show that although CGPANNs performed reasonably well they were not as good as many other methods. There is an oddity here, as on one particular dataset (breast cancer) CGPANNs were compared with *published* results for 31 other machine learning papers [103] and CGPANNs proved to give better results than 21 other techniques. It is not clear why CGPANNs appear not to perform so well on classification problems. Clearly, this issue needs further investigation.

Various other types of ANNs have been evolved using CGP including convolutional ANNs [99] and wavelet ANNs [47, 50]. There is now a growing literature on CGPANNs and they have been applied to many other applications including financial [130],medical [1–4], client prediction [5], load forecasting [41, 42, 45], internet traffic estimation [44] and signal reconstruction [51].

Other promising areas of application of CGP are in multi-step forecasting [15], and cryptography [75, 76]. Also further discussion of CGP applications is presented in [62].

9 Comprehensibility of CGP solutions

CGP phenotypes are often very small and interpretable, this is partly connected to the bias that CGP has towards small programs (see discussion in Sect. 4). This often means that CGP programs can be understood and offer insights and understanding into problem features and aspects that can be interesting and useful at the application level. This was seen in the small size of the object recognition filters (see Sect. 8), data classifiers [23], the understandable successful Atari game-playing strategies found by Wilson et al. [125] and the usefulness of CGP interpretable results to medical practitioners using the results of ICCGP [56, 58, 59]. CGP has also been used to create small readable data filters for drug discovery [17] and understandable intrusion detection programs [6]. This white-box property of CGP can be highly valuable.

10 Comparisons of CGP methods with other techniques

CGP has been applied to many application areas and benchmark problems. In Table 1, we summarize the comparative results published for CGP variants with other methods. The fourth column gives evidence for the high performance of CGP techniques. This is either by the ratio of minimum computational effort of tree-based GP (TGP) to CGP (above 1 means CGP is superior), or performance ranking compared with other machine learning techniques, or by type of medal received at the annual human competitive workshop at the GECCO conference.[7] CGP clearly excels in many domains but particularly in optimised circuits, image processing, classification, reinforcement-learning, time-series prediction and sequence induction.

[7] http://www.human-competitive.org/awards.

Table 1 Comparative performance of some CGP variants with other techniques

Type of CGP	Computational problem	Compared with	Speed-up, ranking, evidence
Standard	Even-5 parity [65]	TGP	13.64
	Approx. circuits [9]	–	Bronze 2018
	Approx. circuits [116]	–	Gold 2015
	Bent circuits [30]	–	Bronze 2014
	Optimised circuits [115]	–	Silver 2011
	Impulse burst noise [88]	6 types of conventional median filter	Highest PSNR[a] in 12/15 images
	Image noise recovery [88]	conventional median filter	Superior MDPP[b] in all cases
Modular [123]	Even-6 parity	TGP	1.37
	Sextic polynomial regr.	TGP	2.16
	Quintic polynomial regr.	TGP	33.43
	Lawn mower (size = 96)	TGP	12.5
	HIFF function	GA	71.0
ICCGP [57]	Parkinson's disease dyskinesia pred. model	–	Gold medal
SMCGP [26]	Even-7 parity	TGP	4.54
Mixed-type [23]	Cancer classification	11 methods	1st
	Phoneme classification	9 methods	7th
	Diabetes classification	12 methods	4th
	Heart classification	5 methods	5th
Recurrent [109]	Fibonacci series induction	6 methods	1st
PCGP [125]	Atari game strategy 56 games	9 methods	1st on 9 games

Table 1 (continued)

Type of CGP	Computational problem	Compared with	Speed-up, ranking, evidence
CGPANN [103, 111]	Laser time-series	9 methods	1st
	Mackey-glass time-series	9 methods	1st
	Sunspot time-series	9 methods	1st
	Pole-balancing	13 methods	4th
	Arm-throwing	3 methods	1st
	Monks classification	31 methods	9th
	Breast cancer [4] classification	21 methods	6th
	Client requests for [5] resource allocation	11 methods	1st
	Foreign currency exchange rate prediction [130]	8 methods	1st
	Electrical load forecasting [42]	8 methods	1st
	Very short term load forecasting [45]	10 methods	1st
	Audio signal reconstruction [51]	8 methods	1st

[a]Peak signal to noise ratio

[b]Mean difference per pixel

11 Software

CGP implementations are available in a number of languages: C, C++, Java, Matlab, Python, Julia.[8] The three most extensive packages are described below.

A cross-platform, open source, extensive and extensible software library[9] has been written in C by Andrew Turner [106]. The package includes standard and recurrent CGP including Artificial Neural Networks. It has been designed to be simple to use and adapt. It defines a well documented CGP Application Programming Interface (API). This means that the user does not need to understand or edit the underlying implementation in order to use the CGP library. Also users can benefit from backwards compatible updates to the library. A compiled library has the advantage that it can be used natively by the C and C++ programming languages but also imported into other languages including Python. The CGP library can also be compiled for a wide range of operating systems as it only depend upon standard C libraries.

Differentiable CGP [33] is available in C++ and Python.[10] It includes examples and tutorials on a number of problem types.

Recently, a freely downloadable flexible toolbox called CGP4Matlab[11] has been developed [73] that allows CGP to be run within MATLAB. The toolbox is particularly suited to signal and image processing. As a demonstration Miragaia et al. used the toolbox for pitch estimation and obtained results comparable with the state-of-the-art.

12 Open questions and issues

We discuss a number of ideas, questions and open issues that are worthy of further investigation. It is very important when new algorithms and methods are proposed that they are evaluated on suites of different computational problems using benchmarks recommended by the wider research community. At present although the many operators and algorithmic variants look promising and interesting, further investigation is required to establish whether they have general advantages. Conducting experiments on the basis of both a fixed number of fitness evaluations and a fixed budget of active nodes processed appears offer promise for fair comparisons with different parameters or methods. Indeed, this could be adopted generally in the GP community.

[8] cartesiangp.com.
[9] http://www.cgplibrary.co.uk/files2/About-txt.html.
[10] http://darioizzo.github.io/d-CGP/index.html.
[11] https://github.com/tiagoinacio/cgp4matlab.

12.1 Modularity

The design of an effective form of modularity in CGP is still an open question. Although, Koza showed that automatically defined functions were very useful in tree-based GP and particularly so in harder computational problems [54]. Their usefulness in CGP is less clear. Since CGP encodes graphs, nodes can be multiply used, perhaps this makes independent modules less necessary. We have already discussed a form of multichromosome CGP in which the outputs of independent secondary chromosomes can be used by a primary chromosome. This links with the idea of modules as one could easily view the secondary chromosomes as possible modules. It will be interesting to see if this improves the efficiency of CGP on a range of computational problems.

12.2 Representation

The way CGP programs are represented in a genotype is an important issue. We have examined a number of these discussed in the literature. Since RVCGP and PCGP are more general than standard CGP they both look promising to investigate in much more detail. They allow types of mutation and crossover that are difficult to express in standard integer-based CGP. The addition of evolvable node positions in PCGP is an interesting aspect that requires further investigation.

12.3 Mutation

Mutation is extremely important in CGP. More sophisticated mutation operators that take account of the presence of inactive nodes appear to be more effective (i.e. Goldman and Punch's Reorder and Kalkreuth's insertion and deletion). Mutation operators that eliminate the length bias and the non-uniform location of inactive nodes also have the promise of being more effective. Making node outputs on the right of nodes accessible, via mutation, to nodes on the left (even when no recurrence is desired) as discussed in Sect. 3.10 is clearly important. The merits of mutating different numbers of genes in the generation of mutational offspring (as used by Izzo, see Sect. 3.9) needs also to be investigated. Turner et al. showed that never mutating explicitly inactive genes significantly worsens the evolutionary search compared with allowing mutations to both active and inactive genes [110]. They also speculated that having two different mutation rates for inactive and active genes may be more effective than a single rate, indeed if the role of inactive material is solely to provide mutation with a supply of purely random alterations then a high inactive mutation rate may lead to a more effective search. Finally, in biology there are many forms of mutation (e.g. gene transposing and duplication), perhaps this may inspire new types of mutation in CGP.

12.4 Crossover

Crossover is still very underdeveloped in CGP, this may partly be because the mutation of genotypes with inactive genes already allows mutation to make both small changes and large changes. Nevertheless, Poli suggested a number of forms of crossover for PDGP which could be transferred and investigated in CGP. In addition multi-chromosome crossover looks like a good candidate to investigate. Kalkreuth's crossover operators need to be evaluated on a wide range of problems.

12.5 Weighted CGP

We saw that in differentiable CGP connections were augmented with weights. This begs the following question. How well would weighted CGP work in comparison with unweighted? Weighted CGP could be seen as a generalization of artificial neural networks where node functions could be chosen from a much larger set of mathematical functions (possibly including neural functions also). Of course, evolved solutions using such a large set may make these solutions uninterpretable rather like standard neural networks, however weighted GP has not been investigated in the literature. It remains to be seen whether it will increase the efficiency in evolving solutions to computational problems. However, introducing weights would certainly negatively impact on the comprehensibility of CGP evolved solutions.

12.6 Applying CGP to general optimisation problems

Genetic programming and evolutionary algorithms are usually applied to different classes of computational problems. Since CGP allows multiple outputs, these outputs could be interpreted as a potential solution to a computational problem. This would mean that CGP could be applied to many kinds of optimisation problems not tackled with GP. Using a function set of mathematical operations, it would be possible to use it to generate from a set of constant inputs, a vector of real-valued numbers. Alternatively, one could use CGP to generate a new solution vector from a previous one. The elements of these vectors could be mapped into the required ranges for many fixed length search problems. This would allow CGP to be used as a search algorithm for many problems where evolutionary algorithms are currently used (e.g. Function optimisation, TSP, Bin-packing etc). This is interesting as one would be evolving *programs* to generate vectors rather than evolving the vectors directly. This might have advantages, for instance, some vector elements could be functions of each other (thus indicating a lower dimensional problem) something that is difficult to discover using standard evolutionary algorithms. In addition, evolution would be working in a completely different genotype space which may have advantages. This idea has already received preliminary investigation for TSP[12] [11] and function optimisation [68].

[12] To obtain a permutation from an indexed vector of real-numbers one merely has to sort the numbers and the re-arranged indexes form a permutation.

12.7 CGP encoded ANNs

Using backpropagation to adjust weights of CGP encoded ANNs has the potential to produce compact highly optimal neural networks. Of course, this could be computationally very expensive. It would be interesting to investigate hybrid algorithms using standard CGP search together with phases of local search with backpropagation. One of the advantanges of using CGP to encode and evolve neural networks is that discoveries that make CGP more efficient have the potential to make CGPANNs a more efficient technique. In principle, the beneficial relationship could work the other way too. In the light of Turner's findings on CGPANNs for classification, further investigation needs to be carried to discover the best way of doing classification with CGPANNs. Another interesting possibility is to use the techniques of module acquisition to create modular CGPANNs. Finally, using SMCGP and choosing all primitive functions to be neural (i.e. sigmoid or hyperbolic tangent) would lead to self-modifying ANNs [67]. This would mean that one could evolve CGPANNs that could modify themselves producing a sequence of neural networks each solving a different problem.

12.8 Search algorithms and real-Valued CGP

One of the advantages of the real-valued CGP representation is that it could be evolved using evolutionary optimisation solvers such as CMAES [21] and other approaches. It allows many methods found to be useful in the genetic and evolutionary algorithms community to be brought to bear on the evolution of programs. Indeed, even CGPANNs could be represented using RVCGP.

12.9 Evolving "brain-like" multiple problem solvers

CGP has been used to produce programs that develop new types of ANNs, these are referred to as developmental ANNs [40, 43, 72]. Khan et al. evolved genotypes that produced a dynamic neural network in which neurons, connections and weights changed during problem solving [40, 43]. They applied the technique to the evolution of intelligent agents, checkers and maze solving. Recently, Miller et al. have simplified the model in [43] so that two evolved programs (one for a neuron soma and the other for the dendrite) can solve multiple problems simultaneously [72]. The eventual aim is the moonshot challenge of producing programs that can build an artificial brain that can learn for itself and solve an arbitrary number of different problems.

13 Conclusion

We have examined the current status of Cartesian genetic programming and discussed a number of different or enhanced representations of the genotype. We also looked at issues with, and current thinking on mutation, crossover and the evolutionary algorithms used to evolve programs. We have made many suggestions for future investigations that may lead to more efficient CGP techniques in the future. CGP is

in a healthy state with an increasing number of researchers taking it up, improving and analyzing it, and applying it to new classes of problems.

Acknowledgements Thanks to the anonymous reviewers and to Dennis Wilson for their helpful comments.

References

1. A.M. Ahmad, G.M. Khan, Bio-signal processing using Cartesian genetic programming evolved artificial neural network (CGPANN), in *2012 10th International Conference on Frontiers of Information Technology (FIT)* (IEEE, 2012), pp. 261–268
2. A.M. Ahmad, G.M. Khan, S.A. Mahmud, Classification of arrhythmia types using Cartesian genetic programming evolved artificial neural networks, in *Engineering Applications of Neural Networks*, ed. by L. Iliadis, C. Jayne (Springer, Berlin, 2013), pp. 282–291
3. A.M. Ahmad, G.M. Khan, S.A. Mahmud, Classification of mammograms using Cartesian genetic programming evolved artificial neural networks, in *AIAI, IFIP Advances in Information and Communication Technology*, vol. 436 (Springer, 2014), pp. 203–213
4. A.M. Ahmad, G.M. Khan, S.A. Mahmud, J.F. Miller, Breast cancer detection using Cartesian genetic programming evolved artificial neural networks, in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation* (2012), pp. 1031–1038
5. J. Ali, F. Zafari, G.M. Khan, S.A. Mahmud, Future clients' requests estimation for dynamic resource allocation in cloud data center using CGPANN, in *2013 12th International Conference on Machine Learning and Applications (ICMLA)*, vol. 2 (IEEE, 2013), pp. 331–334
6. H. Alyasiri, J. Clark, D. Kudenko, Applying Cartesian genetic programming to evolve rules for intrusion detection system, in *Proceedings of the 10th International Joint Conference on Computational Intelligence—Volume 1: IJCCI* (SciTePress, 2018), pp. 176–183
7. T. Atkinson, D. Plump, S. Stepney, Evolving graphs by graph programming, in *Proceedings of the European Conference on Genetic Programming, LNCS*, vol. 10781 (2018), pp. 35–51
8. X. Cai, S.L. Smith, A.M. Tyrrell, Positional independence and recombination in Cartesian Genetic programming, in *European Conference on Genetic Programming, LNCS*, vol. 3905 (2006), pp. 351–360
9. M. Češka, J. Matyáš, V. Mrazek, L. Sekanina, Z. Vašíček, T. Vojnar, Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished, in *Proceedings of the 36th International Conference on Computer-Aided Design, ICCAD '17* (IEEE Press, 2017), pp. 416–423
10. J. Clegg, J.A. Walker, J.F. Miller, A new crossover technique for Cartesian genetic programming, in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (ACM, 2007), pp. 1580–1587
11. K.D. Clegg, J.F. Miller, K. Massey, M. Petty, Travelling salesman problem solved 'in materio' by evolved carbon nanotube device, in *Parallel Problem Solving from Nature—PPSN XIII* (Springer, 2014), pp. 692–701
12. S. De, M. Babu, Genomic neighbourhood and the regulation of gene expression. Curr. Opin. Cell Biol. **22**, 326–333 (2010)
13. K. Deb, S. Agrawal, A. Pratap, T. Meyarivan, A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II, in *Parallel Problem Solving from Nature PPSN VI, LNCS*, vol. 1917 (2000), pp. 849–858
14. M. Drahošová, L. Sekanina, M. Wiglasz, Adaptive fitness predictors in coevolutionary Cartesian genetic programming. Evolut. Comput. **26**(4), 1–27 (2018)

15. I. Dzalbs, T. Kalganova, Multi-step ahead forecasting using Cartesian genetic programming, in *Inspired by Nature: Essays PresInspired by Nature: Essays Presented to Julian F. Miller on the Occasion of his 60th Birthday*, ed. by S. Stepney, A. Adamatzky (Springer, Berlin, 2018), pp. 235–246
16. Z. Gajda, L. Sekanina, An efficient selection strategy for digital circuit evolution, in *Evolvable Systems: From Biology to Hardware, LNCS*, vol. 6274 (2010), pp. 13–24
17. A.B. Garmendia-Doval, J.F. Miller, S.D. Morley, Cartesian genetic programming and the post docking filtering problem, in *Genetic Programming Theory and Practice II*, ed. by U.M. O'Reilly, T. Yu, R. Riolo, B. Worzel (Springer, New York, 2005), pp. 225–244
18. B.W. Goldman, W.F. Punch, Length bias and search limitations in Cartesian genetic programming, in *Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference* (ACM, 2013), pp. 933–940
19. B.W. Goldman, W.F. Punch, Reducing wasted evaluations in Cartesian genetic programming, in *Proceedings of the European Conference on Genetic Programming*, vol. 7831 (Springer, 2013), pp. 61–72
20. B.W. Goldman, W.F. Punch, Analysis of Cartesian genetic programming's evolutionary mechanisms. IEEE Trans. Evolut. Comput. **19**(3), 359–373 (2015)
21. N. Hansen, A. Ostermeier, Completely derandomized self-adaptation in evolution strategies. Evolut. Comput. **9**(2), 159–195 (2001)
22. S. Harding, W. Banzhaf, Fast genetic programming on GPUS, in *Proceedings of the European Conference on Genetic Programming, LNCS*, vol. 4445 (2007), pp. 90–101
23. S. Harding, V. Graziano, J. Leitner, J. Schmidhuber, MT-CGP: mixed type Cartesian genetic programming, in *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference* (ACM, 2012), pp. 751–758
24. S. Harding, J. Leitner, J. Schmidhuber, *Genetic Programming Theory and Practice X. Cartesian Genetic Programming for Image Processing* (Springer, Berlin, 2013), pp. 31–44
25. S. Harding, J.F. Miller, *Cartesian Genetic Programming on the GPU* (Springer, Berlin, 2013), pp. 249–266
26. S. Harding, J.F. Miller, W. Banzhaf, Developments in Cartesian genetic programming: self-modifying CGP. Genet. Program. Evolvable Mach. **11**(3–4), 397–439 (2010)
27. S. Harding, J.F. Miller, W. Banzhaf, Self modifying Cartesian genetic programming: finding algorithms that calculate pi and e to arbitrary precision, in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation* (ACM, 2010), pp. 579–586
28. S.L. Harding, J.F. Miller, W. Banzhaf, Self-modifying Cartesian genetic programming, in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO '07* (2007), pp. 1021–1028
29. S.L. Harding, J.F. Miller, W. Banzhaf, *Self-Modifying Cartesian Genetic Programming* (Springer, Berlin, 2011), pp. 101–124
30. R. Hrbacek, V. Dvorak, Bent function synthesis by means of Cartesian genetic programming, in *Parallel Problem Solving from Nature—PPSN XIII*, ed. by T. Bartz-Beielstein, J. Branke, B. Filipič, J. Smith (Springer, Berlin, 2014), pp. 414–423
31. R. Hrbacek, M. Šikulová, Coevolutionary Cartesian genetic programming in FPGA, in *Proceedings of the Conference on Artificial Life* (2013), pp. 431–438
32. J. Husa, R. Kalkreuth, A comparative study on crossover in Cartesian genetic programming, in *Proceedings of the European Conference on Genetic Programming, LNCS*, vol. 10781 (2018), pp. 203–219
33. D. Izzo, F. Biscani, A. Mereta, Differentiable genetic programming, in *Proceedings of the European Conference on Genetic Programming, Lecture Notes in Computer Science*, vol. 10196 (2017), pp. 35–51
34. R. Kalkreuth, Towards Advanced Phenotypic Mutations in Cartesian Genetic Programming (2018). CoRR arXiv:abs/1803.06127
35. R. Kalkreuth, G. Rudolph, A. Droschinsky, A new subgraph crossover for Cartesian genetic programming, in *Proceedings of the European Conference Genetic Programming, LNCS*, vol. 10196 (2017), pp. 294–310
36. R. Kalkreuth, G. Rudolph, J. Krone, Improving convergence in Cartesian genetic programming using adaptive crossover, mutation and selection, in *2015 IEEE Symposium Series on Computational Intelligence* (2015), pp. 1415–1422

37. P. Kaufmann, R. Kalkreuth, Parametrizing Cartesian genetic programming: an empirical study, in *KI 2017: Advances in Artificial Intelligence, LNCS*, vol. 10505 (2017), pp. 316–322
38. P. Kaufmann, M. Platzner, Advanced techniques for the creation and propagation of modules in Cartesian genetic programming, in *Proceedings of the Conference on Genetic and Evolutionary Computation* (2008), pp. 1219–1226
39. P. Kaufmann, M. Platzner, Combining local and global search: a multi-objective evolutionary algorithm for Cartesian genetic programming, in *Inspired by Nature: Essays Presented to Julian F. Miller on the Occasion of his 60th Birthday*, ed. by S. Stepney, A. Adamatzky (Springer, Berlin, 2018), pp. 175–194
40. G.M. Khan, *Evolution of Artificial Neural Development—In Search of Learning Genes, Studies in Computational Intelligence*, vol. 725 (Springer, Berlin, 2018)
41. G.M. Khan, S. Khan, F. Ullah, Short-term daily peak load forecasting using fast learning neural network, in *2011 11th International Conference on Intelligent Systems Design and Applications (ISDA)* (IEEE, 2011), pp. 843–848
42. G.M. Khan, A.R. Khattak, F. Zafari, S.A. Mahmud, Electrical load forecasting using fast learning recurrent neural networks, in *The 2013 International Joint Conference on Neural Networks (IJCNN)* (IEEE, 2013), pp. 1–6
43. G.M. Khan, J.F. Miller, D.M. Halliday, Evolution of Cartesian genetic programs for development of learning neural architecture. Evolut. Comput. **19**(3), 469–523 (2011)
44. G.M. Khan, F. Ullah, S.A. Mahmud, MPEG-4 internet traffic estimation using recurrent CGPANN, in *Engineering Applications of Neural Networks*, ed. by L. Iliadis, H. Papadopoulos, C. Jayne (Springer, Berlin, 2013), pp. 22–31
45. G.M. Khan, F. Zafari, S.A. Mahmud, Very short term load forecasting using Cartesian genetic programming evolved recurrent neural networks (CGPRNN), in *2013 12th International Conference on Machine Learning and Applications (ICMLA)*, vol. 2 (IEEE, 2013), pp. 152–155
46. M.M. Khan, A.M. Ahmad, G.M. Khan, J.F. Miller, Fast learning neural networks using Cartesian genetic programming. Neurocomputing **121**, 274–289 (2013)
47. M.M. Khan, S.K. Chalup, A. Mendes, Parkinson's disease data classification using evolvable wavelet neural networks, in *Proceedings of Second Australasian Conference on Artificial Life and Computational Intelligence* (2016), pp. 113–124
48. M.M. Khan, G.M. Khan, J.F. Miller, Evolution of neural networks using Cartesian genetic programming, in *Proceedings of the IEEE Congress on Evolutionary Computation, CEC* (2010), pp. 1–8
49. M.M. Khan, G.M. Khan, J.F. Miller, Evolution of optimal ANNs for non-linear control problems using Cartesian genetic programming, in *Proceedings of the 2010 International Conference on Artificial Intelligence* (2010), pp. 339–346
50. M.M. Khan, A. Mendes, P. Zhang, S.K. Chalup, Evolving multi-dimensional wavelet neural networks for classification using Cartesian genetic programming. Neurocomputing **247**, 39–58 (2017)
51. N.M. Khan, G.M. Khan, Audio signal reconstruction using Cartesian genetic programming evolved artificial neural network (CGPANN), in *ICMLA* (IEEE, 2017), pp. 568–573
52. K. Knezevic, S. Picek, J.F. Miller, Amplitude-oriented mixed-type CGP classification, in *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2017), pp. 1415–1418
53. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (MIT Press, Cambridge, 1992)
54. J.R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT Press, Cambridge, 1994)
55. J. Leitner, S. Harding, A. Förster, J. Schmidhuber, Mars terrain image classification using Cartesian genetic programming, in *11th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)* (2012)
56. M. Lones, J.E. Alty, P. Duggan-Carter, A.J. Turner, D.R. Jamieson, S.L. Smith, Classification and characterisation of movement patterns during levodopa therapy for Parkinson's disease, in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation, GECCO Comp '14* (2014), pp. 1321–1328
57. M.A. Lones, J.E. Alty, J. Cosgrove, P. Duggan-Carter, S. Jamieson, R.F. Naylor, A.J. Turner, S.L. Smith, A new evolutionary algorithm-based home monitoring device for Parkinson's dyskinesia. J. Med. Syst. **41**(11), 176 (2017)

58. M.A. Lones, S.L. Smith, J.E. Alty, S.E. Lacy, K.L. Possin, D.S. Jamieson, A.M. Tyrrell, Evolving classifiers to recognize the movement characteristics of Parkinson's disease patients. IEEE Trans. Evolut. Comput. **18**(4), 559–576 (2014)
59. M.A. Lones, S.L. Smith, A.T. Harris, A.S. High, S.E. Fisher, D.A. Smith, J. Kirkham, Discriminating normal and cancerous thyroid cell lines using implicit context representation Cartesian genetic programming, in *IEEE Congress on Evolutionary Computation* (2010), pp. 1–6
60. M.A. Lones, A.M. Tyrrell, Biomimetic representation with enzyme genetic programming. Genet. Program. Evolvable Mach. **3**(3), 315–315 (2002)
61. M. Lopez-Ibanez, J. Dubois-Lacoste, L.P. Cáceres, M. Birattari, T. Stützle, The irace package: iterated racing for automatic algorithm configuration. Oper. Res. Perspect. **3**, 43–58 (2016)
62. A. Manazir, K. Raza, Recent developments in Cartesian genetic programming and its variants. ACM Comput. Surv. **51**(6), 122:1–122:29 (2019)
63. A. Meier, M. Gonter, R. Kruse, Accelerating convergence in Cartesian genetic programming by using a new genetic operator, in *Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference* (ACM, 2013), pp. 981–988
64. N. Milano, P. Pagliuca, S. Nolfi, Robustness, Evolvability and Phenotypic Complexity: Insights from Evolving Digital Circuits (2017). arXiv:1712.04254
65. J.F. Miller, An empirical study of the efficiency of learning Boolean functions using a Cartesian genetic programming approach, in *Proceedings of the Genetic and Evolutionary Computation Conference*, vol. 2 (1999), pp. 1135–1142
66. J.F. Miller, What bloat? Cartesian genetic programming on Boolean problems, in *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers* (2001), pp. 295–302
67. J.F. Miller, Chapter 8: Neuro-centric and holocentric approaches to the evolution of developmental neural networks, in *Growing Adaptive Machines: Combining Development and Learning in Artificial Neural Networks*, ed. by T. Kowaliw, N. Bredeche, R. Doursat (Springer, Berlin, 2014), pp. 227–249
68. J.F. Miller, M. Mohid, Function optimization using Cartesian genetic programming, in *Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion* (ACM, 2013), pp. 147–148
69. J.F. Miller, S. Smith, Redundancy and computational efficiency in Cartesian genetic programming. IEEE Trans Evolut. Comput. **10**(2), 167–174 (2006)
70. J.F. Miller, P. Thomson, Cartesian genetic programming, in *Proceedings of the European Conference on Genetic Programming*, vol. 1820 (Springer, 2000), pp. 121–132
71. J.F. Miller, P. Thomson, T. Fogarty, Chapter 6: Designing electronic circuits using evolutionary algorithms. Arithmetic circuits: a case study, in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications*, ed. by D. Quagliarella, J. Periaux, C. Poloni, G. Winter (Wiley, Hoboken, 1997)
72. J.F. Miller, D.G. Wilson, S. Cussat-Blanc, Chapter 8: Evolving developmental programs that build neural networks for solving multiple problems, in *Genetic Programming Theory and Practice XVI*, ed. by W. Banzhaf, L. Spector, L. Sheneman (Springer, Berlin, 2019), pp. 137–176
73. R. Miragaia, G. Reis, F. Fernandéz, T. Inácio, C. Grilo, CGP4Matlab—a Cartesian genetic programming MATLAB toolbox for audio and image processing, in *Applications of Evolutionary Computation, LNCS*, vol. 10784 (Springer, 2018), pp. 455–471
74. P.C.D. Paris, E.C. Pedrino, M.C. Nicoletti, Automatic learning of image filters using Cartesian genetic programming. Integr. Comput. Aided Eng. **22**(2), 135–151 (2015)
75. S. Picek, C. Carlet, S. Guilley, J.F. Miller, D. Jakobovic, Evolutionary algorithms for Boolean functions in diverse domains of cryptography. Evolut. Comput. **24**(4), 667–694 (2016)
76. S. Picek, D. Jakobovic, J.F. Miller, L. Batina, M. Cupic, Cryptographic Boolean functions. Appl. Soft Comput. **40**(C), 635–653 (2016)
77. R. Poli, Parallel distributed genetic programming. Technical Report CSRP-96-15, Department of Computer Science, University of Birmingham, UK (1996)
78. R. Poli, Some steps towards a form of parallel distributed genetic programming, in *Proceedings of the First On-line Workshop on Soft Computing* (1996), pp. 290–295
79. R. Poli, Parallel distributed genetic programming, in *New Ideas in Optimization*, ed. by M. Dorigo, D. Corne, F.W. Glover (McGraw-Hill Ltd., London, 1999), pp. 403–432
80. R. Poli, W.B. Langdon, McN.F. Phee, A field guide to genetic programming (2008). Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk. Accessed Apr 2019

81. J. Pujol, R. Poli, Evolving the topology and the weights of neural networks using a dual representation. Appl. Intell. **8**(1), 73–84 (1998)
82. N.J. Radcliffe, Equivalence class analysis of genetic algorithms. Complex Syst. **5**, 183–205 (1991)
83. I. Rechenberg, Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Ph.D. Thesis, Technical University of Berlin, Germany (1971)
84. M.V. Rockman, S.S. Skrovanek, L. Kruglyak, Selection at linked sites shapes heritable phenotypic variation in *C. elegans*. Science **330**(6002), 372–376 (2010)
85. P. Ryser-Welch, Evolving comprehensible and scalable solvers using CGP for solving some real-world inspired problems. Ph.D. Thesis, Department of Electronic Engineering, University of York (2017). http://etheses.whiterose.ac.uk/19011/1/finalThesisv3.pdf. Accessed Apr 2019
86. P. Ryser-Welch, J.F. Miller, J. Swan, M.A. Trefzer, Iterative Cartesian genetic programming: creating general algorithms for solving travelling salesman problems, in *Proceedings of the European Conference on Genetic Programming, LNCS*, vol. 9594 (2016), pp. 294–310
87. L. Sekanina, Image filter design with evolvable hardware, in *Applications of Evolutionary Computing, LNCS*, vol. 2279, ed. by S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf, G.R. Raidl (Springer, Berlin, 2002), pp. 255–266
88. L. Sekanina, S.L. Harding, W. Banzhaf, T. Kowaliw, *Image Processing and CGP* (Springer, Berlin, 2011), pp. 181–215
89. M. Shafique, R. Hafiz, M.U. Javed, S. Abbas, L. Sekanina, Z. Vašíček, V. Mrazek, Adaptive and energy-efficient architectures for machine learning: challenges, opportunities, and research roadmap, in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2017), pp. 627–632
90. M. Šikulová, L. Sekanina, Coevolution in Cartesian genetic programming, in *Proceedings of the European Conference on Genetic Programming, LNCS*, vol. 7244 (2012), pp. 182–193
91. S. Silva, E. Costa, Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. Genet. Program. Evolvable Mach. **10**(2), 141–179 (2009)
92. D. Simon, Biogeography-based optimization. IEEE Trans. Evolut. Comput. **12**, 702–713 (2008)
93. S.L. Smith, Cartesian genetic programming and its application to medical diagnosis. IEEE Comput. Intell. Mag. **6**(4), 56–67 (2011)
94. S.L. Smith, P. Gaughan, D.M. Halliday, Q. Ju, N.M. Aly, J.R. Playfer, Diagnosis of Parkinson's disease using evolutionary algorithms. Genet. Program. Evolvable Mach. **8**(4), 433–447 (2007)
95. S.L. Smith, S. Leggett, A.M. Tyrrell, An implicit context representation for evolving image processing filters. Appl. Evolut. Comput. **3449**, 407–416 (2005)
96. S.L. Smith, M.A. Lones, Medical applications of Cartesian genetic programming, in *Inspired by Nature: Essays Presented to Julian F. Miller on the Occasion of his 60th Birthday*, ed. by S. Stepney, A. Adamatzky (Springer, Berlin, 2018), pp. 247–266
97. S.L. Smith, M.A. Lones, M. Bedder, J.E. Alty, R. Cosgrove, R.J. Maguire, M.E. Pownall, D. Ivanoiu, C. Lyle, A. Cording, C.J. Elliott, Computational approaches for understanding the diagnosis and treatment of Parkinson's disease. IET Syst. Biol. **9**(6), 226–23 (2015)
98. S.L. Smith, J.A. Walker, J.F. Miller, *Medical Applications of Cartesian Genetic Programming* (Springer, Berlin, 2011), pp. 309–336
99. M. Suganuma, S. Shirakawa, T. Nagao, A genetic programming approach to designing convolutional neural network architectures, in *Proceedings of the Genetic and Evolutionary Computation Conference* (2017), pp. 497–504
100. H. Tizhoosh, Opposition-based learning: a new scheme for machine intelligence, in *Proceedings of International Conference on Computational Intelligence for Modeling Control and Automation*, vol. 1 (2005), pp. 695–701
101. A.J. Turner, Improving crossover techniques in a genetic program. Masters Thesis, Department of Electronics, University of York (2012). http://www.andrewjamesturner.co.uk. Accessed Apr 2019
102. A.J. Turner, Evolving artificial neural networks using Cartesian genetic programming. Ph.D. Thesis, Department of Electronic Engineering, University of York (2017). http://etheses.whiterose.ac.uk/12035/. Accessed Apr 2019
103. A.J. Turner, J.F. Miller, Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks, in *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO-13)* (2013), pp. 1005–1012
104. A.J. Turner, J.F. Miller, The importance of topology evolution in neuroevolution: a case study using Cartesian genetic programming of artificial neural networks, in *Research and Development in Intelligent Systems XXX*, ed. by M. Bramer, M. Petridis (Springer, Berlin, 2013), pp. 213–226

105. A.J. Turner, J.F. Miller, Cartesian genetic programming: Why no bloat?, in *Proceedings of the European Conference on Genetic Programming, LNCS*, vol. 8599 (2014), pp. 193–204
106. A.J. Turner, J.F. Miller, Introducing a cross platform open source Cartesian genetic programming library. Genet. Program. Evolvable Mach. **16**(1), 83–91 (2014)
107. A.J. Turner, J.F. Miller, NeuroEvolution: the importance of transfer function evolution and heterogeneous networks, in *Proceedings of the 50th Anniversary Convention of the AISB* (2014), pp. 158–165
108. A.J. Turner, J.F. Miller, Recurrent Cartesian genetic programming, in *13th International Conference on Parallel Problem Solving from Nature (PPSN 2014), LNCS*, vol. 8672 (2014), pp. 476–486
109. A.J. Turner, J.F. Miller, Recurrent Cartesian genetic programming applied to famous mathematical sequences, in *Proceedings of the Seventh York Doctoral Symposium on Computer Science & Electronics* (2014), pp. 37–46
110. A.J. Turner, J.F. Miller, Neutral genetic drift: an investigation using Cartesian genetic programming. Genet. Program. Evolvable Mach. **16**(4), 531–558 (2015)
111. A.J. Turner, J.F. Miller, Recurrent Cartesian genetic programming of artificial neural networks. Genet. Program. Evolvable Mach. **18**(2), 185–212 (2017)
112. Z. Vašíček, Bridging the gap between evolvable hardware and industry using Cartesian genetic programming, in *Inspired by Nature: Essays Presented to Julian F. Miller on the Occasion of his 60th Birthday*, ed. by S. Stepney, A. Adamatzky (Springer, Berlin, 2018), pp. 39–55
113. Z. Vašíček, L. Sekanina, Hardware accelerators for Cartesian genetic programming, in *Proceedings of the European Conference on Genetic Programming, LNCS*, vol. 4971 (2008), pp. 230–241
114. Z. Vašíček, L. Sekanina, Hardware accelerator of Cartesian genetic programming with multiple fitness units. Comput. Inform. **29**, 1359–1371 (2010)
115. Z. Vašíček, L. Sekanina, Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. Genet. Program. Evolvable Mach. **12**(3), 305–327 (2011)
116. Z. Vašíček, L. Sekanina, Evolutionary approach to approximate digital circuits design. IEEE Trans. Evolut. Comput. **19**(3), 432–444 (2015)
117. Z. Vašíček, L. Sekanina, Evolutionary design of complex approximate combinational circuits. Genet. Program. Evolvable Mach. **17**(2), 169–192 (2016)
118. V.K. Vassilev, J.F. Miller, Embedding landscape neutrality to build a bridge from the conventional to a more efficient three-bit multiplier circuit, in *Proceedings of the Genetic and Evolutionary Computation Conference* (2000), p. 539. http://cartesiangp.com/julian-miller. Accessed Apr 2019
119. V.K. Vassilev, J.F. Miller, The advantages of landscape neutrality in digital circuit evolution, in *Proceedings of International Conference on Evolvable Systems, LNCS*, vol. 1801 (Springer, 2000), pp. 252–263
120. Z. Vašíček, Cartesian GP in optimization of combinational circuits with hundreds of inputs and thousands of gates, in *Proceedings of European Conference on Genetic Programming, LNCS*, vol. 9025 (2015), pp. 139–150
121. J.A. Walker, J.A. Hilder, A.M. Tyrrell, Evolving variability-tolerant CMOS designs, in *Evolvable Systems: From Biology to Hardware, LNCS*, vol. 5216, ed. by M. Sipper, D. Mange, A. Pérez-Uribe (Springer, Berlin, 2008), pp. 308–319
122. J.A. Walker, J.F. Miller, Evolution and acquisition of modules in Cartesian genetic programming, in *Proceedings of European Conference on Genetic Programming*, vol. 3003 (2004), pp. 187–197
123. J.A. Walker, J.F. Miller, The automatic acquisition, evolution and reuse of modules in Cartesian genetic programming. IEEE Trans. Evolut. Comput. **12**(4), 397–417 (2008)
124. J.A. Walker, K. Völk, S.L. Smith, J.F. Miller, Parallel evolution using multi-chromosome Cartesian genetic programming. Genet. Program. Evolvable Mach. **10**(4), 417–445 (2009)
125. D.G. Wilson, S. Cussat-Blanc, H. Luga, J.F. Miller, Evolving simple programs for playing Atari games, in *Proceedings of the Genetic and Evolutionary Computation Conference* (2018), pp. 229–236
126. D.G. Wilson, J.F. Miller, S. Cussat-Blanc, H. Luga, Positional Cartesian Genetic Programming (2018). arXiv:1810.04119
127. S. Yazdani, J. Shanbehzadeh, Balanced Cartesian genetic programming via migration and opposition-based learning: application to symbolic regression. Genet. Program. Evolvable Mach. **16**(2), 133–150 (2015)
128. T. Yu, J. Miller, Neutrality and the evolvability of Boolean function landscape, in *Genetic Programming, Lecture Notes in Computer Science*, vol. 2038, ed. by L. Sekanina, T. Hu, N. Lourenço, H. Richter, P. García-Sánchez (Springer, Berlin, 2001), pp. 204–217
129. T. Yu, J.F. Miller, Through the interaction of neutral and adaptive mutations, evolutionary search finds a way. Artif. Life **12**(4), 525–551 (2006)

130. F. Zafari, G.M. Khan, M. Rehman, S.A. Mahmud, Evolving recurrent neural network using Cartesian genetic programming to predict the trend in foreign currency exchange rates. Appl. Artif. Intell. **28**(6), 597–628 (2014)
131. E. Zitzler, M. Laumanns, L. Thiele, SPEA2: improving the strength Pareto evolutionary algorithm. Technical Report 103, ETH Zurich (2001)