

Chapter 2

Cartesian Genetic Programming

Julian F. Miller

2.1 Origins of CGP

Cartesian genetic programming grew from a method of evolving digital circuits developed by Miller et al. in 1997 [8]. However the term ‘Cartesian genetic programming’ first appeared in 1999 [5] and was proposed as a general form of genetic programming in 2000 [7]. It is called ‘Cartesian’ because it represents a program using a two-dimensional grid of nodes (see Sect. 2.2).

2.2 General Form of CGP

In CGP, programs are represented in the form of directed acyclic graphs. These graphs are represented as a two-dimensional grid of computational nodes. The genes that make up the genotype in CGP are integers that represent where a node gets its data, what operations the node performs on the data and where the output data required by the user is to be obtained. When the genotype is decoded, some nodes may be ignored. This happens when node outputs are not used in the calculation of output data. When this happens, we refer to the nodes and their genes as ‘non-coding’. We call the program that results from the decoding of a genotype a phenotype. The genotype in CGP has a fixed length. However, the size of the phenotype (in terms of the number of computational nodes) can be anything from zero nodes to the number of nodes defined in the genotype. A phenotype would have zero nodes if all the program outputs were directly connected to program inputs. A phenotype would have the same number of nodes as defined in the genotype when every node in the graph was required. The genotype–phenotype mapping used in CGP is one of its defining characteristics.

The types of computational node functions used in CGP are decided by the user and are listed in a function look-up table. In CGP, each node in the directed graph

represents a particular function and is encoded by a number of genes. One gene is the address of the computational node function in the function look-up table. We call this a *function gene*. The remaining node genes say where the node gets its data from. These genes represent addresses in a data structure (typically an array). We call these *connection genes*. Nodes take their inputs in a feed-forward manner from either the output of nodes in a previous column or from a program input (which is sometimes called a terminal). The number of connection genes a node has is chosen to be the maximum number of inputs (often called the arity) that any function in the function look-up table has. The program data inputs are given the absolute data addresses 0 to n_i minus 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 $n_i + L_n - 1$, where L_n is the user-determined upper bound of the number of nodes. The general form of a Cartesian genetic program is shown in Fig. 2.1.

If the problem requires n_o program outputs, then n_o integers are added to the end of the genotype. In general, there may be a number of output genes (O_i) which specify where the program outputs are taken from. 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. The graph 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. The structure of the genotype is seen below the schematic in Fig. 2.1. All node function genes f_i are integer addresses in a look-up table of functions. All connection genes C_{ij} are data addresses and are integers taking values between 0 and the address of the node at the bottom of the previous column of nodes.

CGP has three parameters that are chosen by the user. 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: $L_n = n_c n_r$. The parameter l 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 topologies. Choosing the number of columns to be small and the number of rows to be large results in graphs that are tall and thin. Choosing the number of columns to be large and the number of rows to be small results in short, wide graphs. Choosing levels-back to be one produces highly layered graphs in which calculations are carried out column by column. 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. In this case the genotype can represent any bounded directed graph where the upper bound is determined by the number of columns.

As we saw briefly in Chap. 1, one of the benefits of a graph-based representation of a program is that graphs, by definition, allow the implicit reuse of nodes, as a

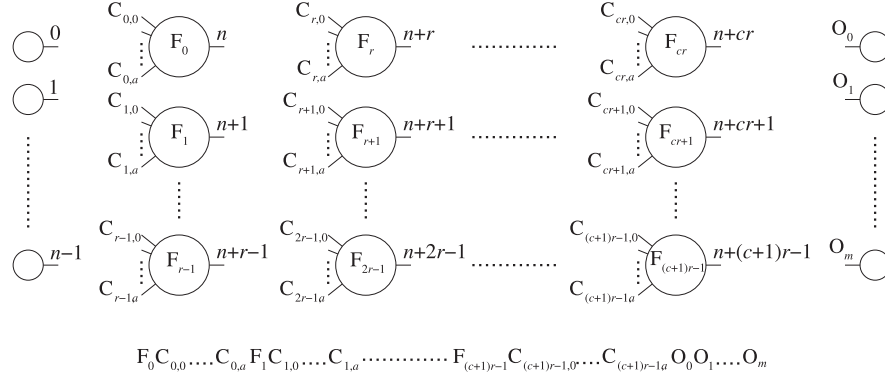


Fig. 2.1 General form of CGP. It is a grid of nodes whose functions are chosen from a set of primitive functions. The grid has n_c columns and n_r rows. The number of program inputs is n_i and the number of program outputs is n_o . Each node is assumed to take as many inputs as the maximum function arity 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).

node can be connected to the output of any previous node in the graph. In addition, CGP can represent programs having an arbitrary number of outputs. In Sect. 2.7, we will discuss the advantages of non-coding genes. This gives CGP a number of advantages over tree-based GP representations.

2.3 Allelic Constraints

In the previous section, we discussed the integer-based CGP genotype representation. The values that genes can take (i.e. alleles) are highly constrained in CGP. When genotypes are initialized or mutated, these constraints should be obeyed.

First of all, the alleles of function genes f_i must take valid address values in the look-up table of primitive functions. Let n_f represent the number of allowed primitive node functions. Then f_i must obey

$$0 \leq f_i \leq n_f. \quad (2.1)$$

Consider a node in column j . The values taken by the connection genes C_{ij} of all nodes in column j are as follows. If $j \geq l$,

$$n_i + (j - l)n_r \leq C_{ij} \leq n_i + jn_r. \quad (2.2)$$

If $j < l$,

$$0 \leq C_{ij} \leq n_i + jn_r. \quad (2.3)$$

Output genes O_i can connect to any node or input:

$$0 \leq O_i < n_i + L_n. \quad (2.4)$$

2.4 Examples

CGP can represent many different kinds of computational structures. In this section, we discuss three examples of this. The first example is where a CGP genotype encodes a digital circuit. In the second example, a CGP genotype represents a set of mathematical equations. In the third example, a CGP genotype represents a picture. In the first example, the type of data input is a single bit; in the second, it is real numbers. In the art example, it is unsigned eight-bit numbers.

2.4.1 A Digital Circuit

The evolved genotype shown in Fig. 2.2 arose using CGP genotype parameters $n_c = 10$, $n_r = 1$ and $l = 10$. It represents a digital combinational circuit called a two-bit parallel multiplier. It multiplies two two-bit numbers together, so it requires four inputs and four outputs. There are four primitive functions in the function set (logic gates). Let the first and second inputs to these gates be a and b . Then the four functions (with the function gene in parentheses) are AND(a,b)(0), AND(a ,NOT(b))(1), XOR(a,b)(2) and OR(a,b)(3). One can see in Fig. 2.2 that two nodes (with labels 6 and 10) are not used, since no circuit output requires them. These are non-coding nodes. They are shown in grey.

Figure 2.2 shows a CGP genotype and the corresponding phenotype.

2.4.2 Mathematical Equations

Suppose that the functions of nodes can be chosen from the arithmetic operations plus, minus, multiply and divide. We have allocated the function genes as follows. Plus is represented by the function gene being equal to zero, minus is represented

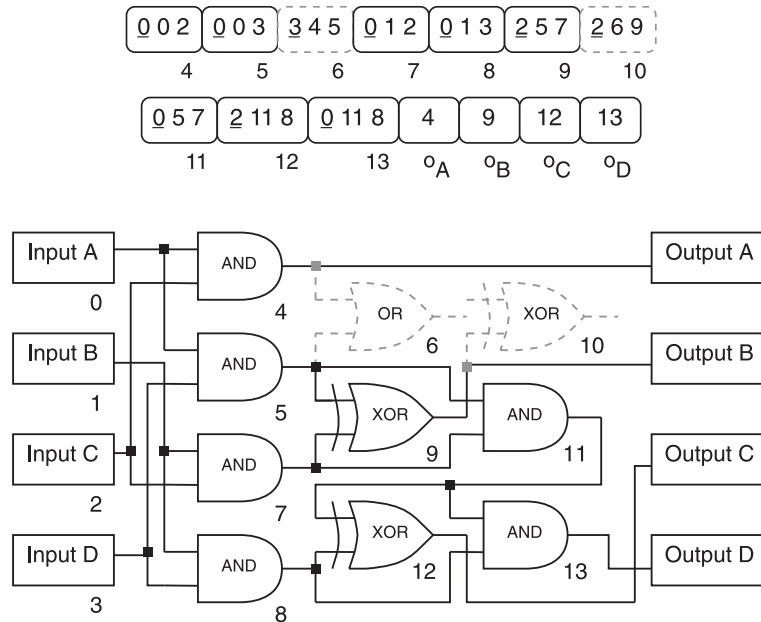


Fig. 2.2 A CGP genotype and corresponding phenotype for a two-bit multiplier circuit. The underlined genes in the genotype encode the function of each node. The function look-up table is AND (0), AND with one input inverted (1), XOR (2) and OR (3). The addresses are shown underneath each program input and node in the genotype and phenotype. The inactive areas of the genotype and phenotype are shown in grey dashes (nodes 6 and 10).

by one, multiply by two and divide by three. Let us suppose that our program has two real-valued inputs, which symbolically we denote by x_0 and x_1 . Let us suppose that we need four program outputs, which we denote O_A , O_B , O_C and O_D . We have chosen the number of columns n_c to be three and the number of rows n_r to be two. In this example, assume that levels-back, l is two. An example genotype and a schematic of the phenotype are shown in Fig. 2.3. The phenotype is the following set of equations:

$$\begin{aligned}
 O_A &= x_0 + x_1 \\
 O_B &= x_0 * x_1 \\
 O_C &= \frac{x_0 * x_1}{x_0^2 - x_1} \\
 O_D &= x_0^2.
 \end{aligned}
 \tag{2.5}$$

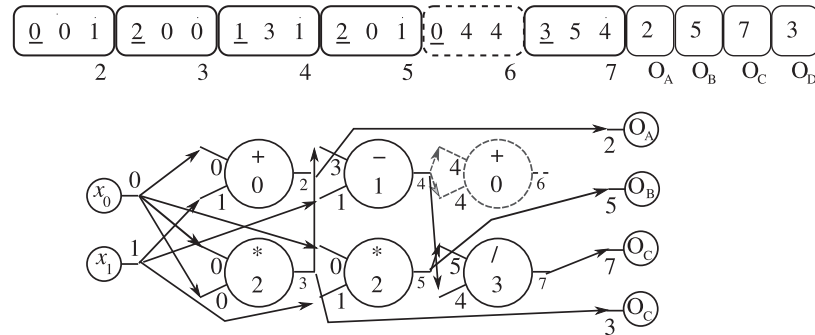


Fig. 2.3 A CGP genotype and corresponding schematic phenotype for a set of four mathematical equations. The underlined genes in the genotype encode the function of each node. The function look-up table is add (0), subtract (1), multiply (2) and divide (3). The addresses are shown underneath each program input and node in the genotype and phenotype. The inactive areas of the genotype and phenotype are shown in grey dashes (node 6).

2.4.3 Art

A simple way to generate pictures using CGP is to allow the integer pixel Cartesian coordinates to be the inputs to a CGP genotype. Three outputs can then be allowed which will be used to determine the red, green and blue components of the pixel's colour. The CGP outputs have to be mapped in some way so that they only take values between 0 and 255, so that valid pixel colours are defined. A CGP program is executed for all the pixel coordinates defining a two-dimensional region. In this way, a picture will be produced. In Fig. 2.4, a genotype is shown with a corresponding schematic of the phenotype. The set of function genes and corresponding node functions are shown in Table 2.1. In a later chapter, ways of developing art using CGP will be considered in detail.

The functions in Table 2.1 have been carefully chosen so that they will return an integer value between 0 and 255 when the inputs (Cartesian coordinates) x and y are both between 0 and 255. The evolved genotype shown in Fig. 2.4 uses only four function genes: 5, 9, 6 and 13. We denote the outputs of nodes by g_i , where i is the output address of the node. The red, green and blue channels of the pixel values (denoted r , g , b) are given as below:

Table 2.1 Primitive function set used in art example

Function gene	Function definition
0	x
1	y
2	$\sqrt{x + y}$
3	$\sqrt{ x - y }$
4	$255(\sin(\frac{2\pi}{255}x) + \cos(\frac{2\pi}{255}y))/2$
5	$255(\cos(\frac{2\pi}{255}x) + \sin(\frac{2\pi}{255}y))/2$
6	$255(\cos(\frac{3\pi}{255}x) + \sin(\frac{2\pi}{255}y))/2$
7	$\exp(x + y) \pmod{256}$
8	$ \sinh(x + y) \pmod{256}$
9	$\cosh(x + y) \pmod{256}$
10	$255 \tanh(x + y) $
11	$255(\sin(\frac{\pi}{255}(x + y)))$
12	$255(\cos(\frac{\pi}{255}(x + y)))$
13	$255(\tan(\frac{\pi}{8 \cdot 255}(x + y)))$
14	$\sqrt{\frac{x^2 + y^2}{2}}$
15	$ x ^y \pmod{256}$
16	$ x + y \pmod{256}$
17	$ x - y \pmod{256}$
18	$xy/255$
19	$\begin{cases} x & y = 0 \\ x/y & y \neq 0 \end{cases}$

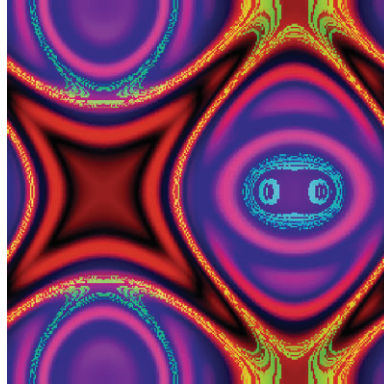


Fig. 2.5 Picture produced when the program encoded in the evolved genotype shown in Fig. 2.4 is executed. It arose in the sixth generation. The user decides which genotype will be the next parent.

There are various ways that this decoding process can be implemented. One way would be to do it recursively; another would be to determine which nodes are active (in a recursive way) and record them for future use, and only process those. Procedures 2.2 and 2.1 in the next section detail the latter. The possibility of improving efficiency by stripping out non-coding instructions prior to phenotype evaluation has also been suggested for Linear GP [1].

2.5.1 Algorithms for Decoding a CGP Genotype

In this section, we will present algorithmic procedures for decoding a CGP genotype. The algorithm has two main parts: the first is a procedure that determines how many nodes are used and the addresses of those nodes. This is shown in Procedure 2.1. The second presents the input data to the nodes and calculates the outputs for a single data input. We denote the maximum number of addresses in the CGP graph by $M = L_n + n_i$, the total number of genes in the genotype by L_g , the number of genes in a node by n_n , and the number of active or used nodes by n_u .

In the procedure, a number of arrays are mentioned. Firstly, it takes the CGP genotype stored in an array $G[L_g]$ as an argument. It passes back as an argument an array holding the addresses of the nodes in the genotype that are used. We denote this by NP . It also returns how many nodes are used. Internally, it uses a Boolean array holding whether any node is used, called $NU[M]$. This is initialized to *FALSE*. An array NG is used to store the node genes for any particular node. It also assumes that a function $Arity(F)$ returns the arity of any function in the function set.

Once we have the information about which nodes are used, we can efficiently decode the genotype G with some input data to find out what data the encoded pro-

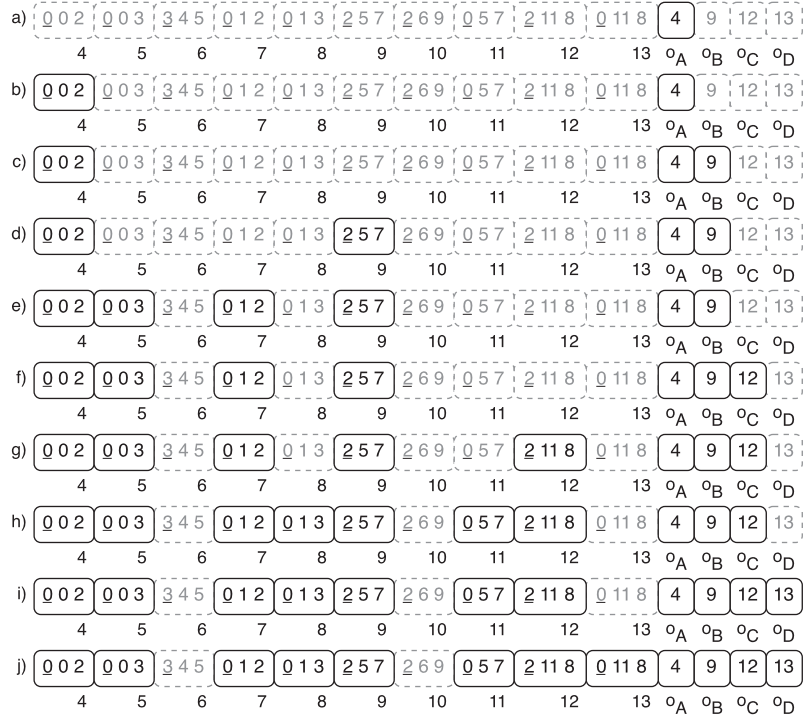


Fig. 2.6 The decoding procedure for a CGP genotype for the two-bit multiplier problem. (a) Output A (o_A) connects to the output of node 4; move to node 4. (b) Node 4 connects to the program inputs 0 and 2; therefore the output A is decoded. Move to output B. (c) Output B (o_B) connects to the output of node 9; move to node 9. (d) Node 9 connects to the output of nodes 5 and 7; move to nodes 5 and 7. (e) Nodes 5 and 7 connect to the program inputs 0, 3, 1 and 2; therefore output B is decoded. Move to output C. The procedure continues until output C (o_C) and output D (o_D) are decoded (steps (f) to (h) and steps (i) to (j) respectively). When all outputs are decoded, the genotype is fully decoded.

gram gives as an output. This procedure is shown in Procedure 2.2. It assumes that input data is stored in an array DIN , and the particular item of that data that is being used as input to the CGP genotype is $item$. The procedure returns the calculated output data in an array O . Internally, it uses two arrays: o , which stores the calculated outputs of used nodes and in , which stores the input data being presented to an individual node. The symbol g is the address in the genotype G of the first gene in a node. The symbol n is the address of a node in the array NP . The procedure assumes that a function NF implements the functions in the function look-up table.

The fitness function required for an evolution algorithm is given in Procedure 2.3. It is assumed that there is a procedure $EvaluateCGP$ that, given the CGP cal-

Procedure 2.1 Determining which nodes need to be processed

```

1: NodesToProcess(G, NP) // return the number of nodes to process
2: for all i such that  $0 \leq i < M$  do
3:   NU[i] = FALSE
4: end for
5: for all i such that  $L_g - n_o \leq i < L_g$  do
6:   NU[G[i]] ← TRUE
7: end for
8: for all i such that  $M - 1 \geq i \geq n_i$  do // Find active nodes
9:   if NU[i] ← TRUE then
10:    index ←  $n_n(i - n_i)$ 
11:    for all j such that  $0 \leq j < n_n$  do // store node genes in NG
12:      NG[j] ← G[index + j]
13:    end for
14:    for all j such that  $0 \leq j < \text{Arity}(\text{NG}[n_n - 1])$  do
15:      NU[NG[j]] ← TRUE
16:    end for
17:  end if
18: end for
19: nu = 0
20: for all j such that  $n_i \leq j < M$  do // store active node addresses in NP
21:   if NU[j] = TRUE then
22:     NP[nu] ← j
23:     nu ← nu + 1
24:   end if
25: end for
26: return nu

```

Procedure 2.2 Decoding CGP to get the output

```

1: DecodeCGP(G, DIN, O, nu, NP, item)
2: for all i such that  $0 \leq i < n_i$  do
3:   o[i] ← DIN[item]
4: end for
5: for all j such that  $0 \leq j < n_u$  do
6:   n ← NP[j] - ni
7:   g ←  $n_n n$ 
8:   for all i such that  $0 \leq i < n_n - 1$  do // store data needed by a node
9:     in[i] ← o[G[g + i]]
10:  end for
11:  f = G[g + nn - 1] // get function gene of node
12:  o[n + ni] = NF(in, f) // calculate output of node
13: end for
14: for all j such that  $0 \leq j < n_o$  do
15:   O[j] ← o[G[Lg - no + j]]
16: end for

```

culated outputs O and the desired program outputs $DOUT$, calculates the fitness of the genotype f_i for a single input data item. The procedure assumes that there are N_{fc} fitness cases that need to be considered. In digital-circuit evolution the usual number of fitness cases is $n_o 2^{n_i}$. Note, however, that Procedure 2.1 only needs to be executed once for a genotype, irrespective of the number of fitness cases.

Procedure 2.3 Calculating the fitness of a CGP genotype

```

1:  $FitnessCGP(G)$ 
2:  $n_u \leftarrow NodesToProcess(G, NP)$ 
3:  $fit \leftarrow 0$ 
4: for all  $i$  such that  $0 \leq i < N_{fc}$  do
5:    $DecodeCGP(G, DIN, O, n_u, NP, item)$ 
6:    $f_i = EvaluateCGP(O, DOUT, i)$ 
7:    $fit \leftarrow fit + f_i$ 
8: end for

```

2.6 Evolution of CGP Genotypes

2.6.1 Mutation

The mutation operator used in CGP is a point mutation operator. In a point mutation, an allele at a randomly chosen gene location is changed to another valid random value (see Sect. 2.3). If a function gene is chosen for mutation, then a valid value is the address of any function in the function set, whereas if an input gene is chosen for mutation, then a valid value is the address of the output of any previous node in the genotype or of any program input. Also, a valid value for a program output gene is the address of the output of any node in the genotype or the address of a program input. The number of genes in the genotype that can be mutated in a single application of the mutation operator is defined by the user, and is normally a percentage of the total number of genes in the genotype. We refer to the latter as the *mutation rate*, and will use the symbol μ_r to represent it. Often one wants to refer to the actual number of gene sites that could be mutated in a genotype of a given length L_g . We give this quantity the symbol μ_g , so that $\mu_g = \mu_r L_g$. We will talk about suitable choices for the parameters μ_r and μ_g in Sect. 2.8.

An example of the point mutation operator is shown in Fig. 2.7, which highlights how a small change in the genotype can sometimes produce a large change in the phenotype.

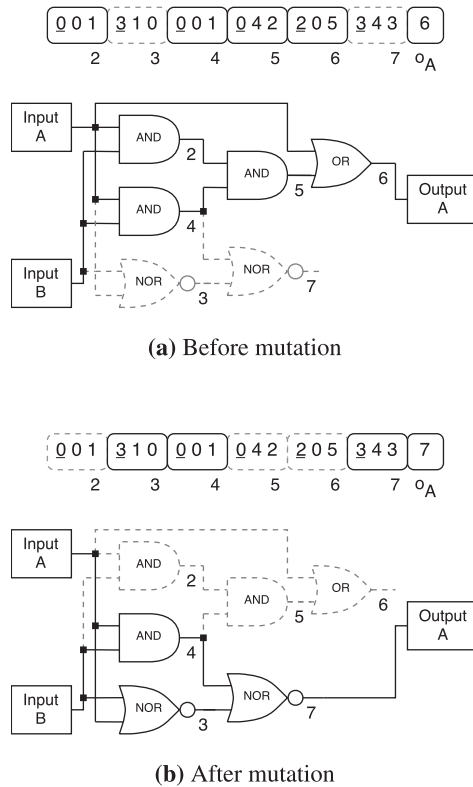


Fig. 2.7 An example of the point mutation operator before and after it is applied to a CGP genotype, and the corresponding phenotypes. A single point mutation occurs in the program output gene (o_A), changing the value from 6 to 7. This causes nodes 3 and 7 to become active, whilst making nodes 2, 5 and 6 inactive. The inactive areas are shown in grey dashes.

2.6.2 Recombination

Crossover operators have received relatively little attention in CGP. Originally, a one-point crossover operator was used in CGP (similar to the n -point crossover in genetic algorithms) but was found to be disruptive to the subgraphs within the chromosome, and had a detrimental affect on the performance of CGP [5]. Some work by Clegg et al. [2] has investigated crossover in CGP (and GP in general). Their approach uses a floating-point crossover operator, similar to that found in evolutionary programming, and also adds an extra layer of encoding to the genotype, in which all genes are encoded as a floating-point number in the range $[0, 1]$. A larger population and tournament selection were also used instead of the $(1 + 4)$ evolutionary strategy normally used in CGP, to try and improve the population diversity.

The results of the new approach appear promising when applied to two symbolic regression problems, but further work is required on a range of problems in order to assess its advantages [2]. Crossover has also been found to be useful in an image-processing application as discussed in Sect. 6.4.3. Crossover operators (cone-based crossover) have been devised for digital-circuit evolution (see Sect. 3.6.2). In situations where a CGP genotype is divided into a collection of chromosomes, crossover can be very effective. Sect. 3.8 discusses how new genotypes created by selecting the best chromosomes from parents' genotypes can produce *super-individuals*. This allows difficult multiple-output problems to be solved.

2.6.3 Evolutionary Algorithm

A variant on a simple evolutionary algorithm known as a $1 + \lambda$ evolutionary algorithm [9] is widely used for CGP. Usually λ is chosen to be 4. This has the form shown in Procedure 2.4.

Procedure 2.4 The (1 + 4) evolutionary strategy

```

1: for all  $i$  such that  $0 \leq i < 5$  do
2:   Randomly generate individual  $i$ 
3: end for
4: Select the fittest individual, which is promoted as the parent
5: while a solution is not found or the generation limit is not reached do
6:   for all  $i$  such that  $0 \leq i < 4$  do
7:     Mutate the parent to generate offspring  $i$ 
8:   end for
9:   Generate the fittest individual using the following rules:
10:  if an offspring genotype has a better or equal fitness than the parent then
11:    Offspring genotype is chosen as fittest
12:  else
13:    The parent chromosome remains the fittest
14:  end if
15: end while

```

On line 10 of the procedure there is an extra condition that when offspring genotypes in the population have the same fitness as the parent and there is no offspring that is better than the parent, in that case an *offspring* is chosen as the new parent. This is a very important feature of the algorithm, which makes good use of redundancy in CGP genotypes. This is discussed in Sect. 2.7.

2.7 Genetic Redundancy in CGP Genotypes

We have already seen that in a CGP genotype there may be genes that are entirely inactive, having no influence on the phenotype and hence on the fitness. Such inactive genes therefore have a neutral effect on genotype fitness. This phenomenon is often referred to as neutrality. CGP genotypes are dominated by redundant genes. For instance, Miller and Smith showed that in genotypes having 4000 nodes, the percentage of inactive nodes is approximately 95%! [6].

The influence of neutrality in CGP has been investigated in detail [7, 6, 10, 13, 14] and has been shown to be extremely beneficial to the efficiency of the evolutionary process on a range of test problems. Just how important neutral drift can be to the effectiveness of CGP is illustrated in Fig. 2.8.

This shows the normalized best fitness value achieved in two sets of 100 independent evolutionary runs of 10 million generations [10]. The target of evolution was to evolve a correct three-bit digital parallel multiplier. In the first set of runs, an offspring could replace a parent when it had the same fitness as the parent and there was no other population member with a better fitness (as in line 10 of Procedure 2.4). In the figure, the final fitness values are indicated by diamond symbols. In the second set, a parent was replaced only when an offspring had a strictly better fitness value. These results are indicated by plus symbols. In the case where neutral drift was allowed, 27 correct multipliers were evolved. Also, many of the other circuits were very nearly correct. In the case where no neutral drift was allowed, there were no runs that produced a correct multiplier, and the average fitness values are considerably lower.

It is possible that by analyzing within an evolutionary algorithm whether mutational offspring are phenotypically different from parents, one may be able to reduce the number of fitness evaluations. Since large amounts of non-coding genes are helpful to evolution, it is more likely that mutations will occur only in non-coding sections of the genotype; such genotypes will have the same fitness as their parents and do not need to be evaluated. To accomplish this would require a slight change to the evolutionary algorithm in Procedure 2.4. One would keep a record of the nodes that need to be processed in the genotype that is promoted (i.e. array NP in Sect. 2.5.1). Then, if an offspring had exactly the same nodes that were active as in the parent, it would be assigned the parent's fitness. Whether in practice this happens sufficiently often to warrant the extra processing required has not been investigated.

2.8 Parameter Settings for CGP

To arrive at good parameters for CGP normally requires some experimentation on the problem being considered. However, some general advice can be given. A suitable mutation rate depends on the length of the genotype (in nodes). As a rule of thumb, one should use about 1% mutation if a maximum of 100 nodes are used (i.e. $n_c n_r = 100$). Let us assume that all primitive functions have two connection genes

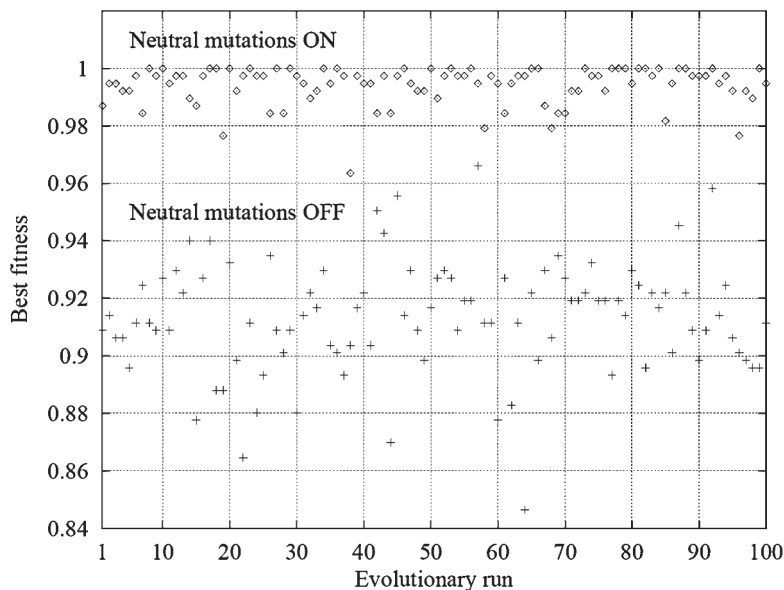


Fig. 2.8 The normalized best fitness value achieved in two sets of 100 independent evolutionary runs of 10 million generations. The target of evolution was to evolve a correct three-bit digital parallel multiplier. In one set of runs, neutral drift was allowed, and in the other, neutral drift was not allowed. The evolutionary algorithm was unable to produce a correct circuit in the second case.

and the problem has a single output. Then a genotype with a maximum of 100 nodes will require 301 genes. So 1% mutation would mean that up to three genes would be mutated in the parent to create an offspring. Experience shows that to achieve reasonably fast evolution one should arrange the mutation rate μ_r to be such that the number of gene locations chosen for mutation is a slowly growing function of the genotype length. For instance, in [6] it was found that $\mu_g = 90$ proved to be a good value when $L_g = 12,004$ (4000 two-input nodes and four outputs). This corresponds to $\mu_r = 0.75\%$. When $L_g = 154$ (50 two-input nodes and four outputs), a good value of μ_g was 6, which corresponds to $\mu_r = 4\%$. Even smaller genotypes usually require higher mutation rates still for fast evolution.

Generally speaking, when optimal mutation rates are used, larger genotypes require fewer fitness evaluations to achieve evolutionary success than do smaller genotypes. For instance, Miller and Smith found that the number of fitness evaluations (i.e. genotypes whose fitness is calculated) required to successfully evolve a two-bit multiplier circuit was lower for genotypes having 4000 nodes than for genotypes of smaller length [6]! The way to understand this is to think about the usefulness of neutral drift in the evolution of CGP genotypes. Larger CGP genotypes have a much larger percentage of non-coding genes than do smaller genotypes, so the potential

for neutral drift is much larger. This is another illustration of the great importance of neutral drift in evolutionary algorithms for CGP.

So, we have seen that large genotypes lead to more efficient evolution; however, given a certain genotype length, what is the optimal number of columns n_c and number of rows n_r ? The advice here is as follows. If there are no problems with implementing arbitrary directed graphs, then the recommended choice of these parameters is $n_r = 1$ with $l = n_c$. However, if for instance one is evolving circuits for implementation of evolved CGP genotypes on digital devices (with limited routing resources), it is often useful to choose $n_c = n_r$. It should be stressed that these recommendations are ‘rules of thumb’, as no detailed quantitative work on this aspect has been published.

CGP uses very small population sizes (5 in the case described in Sect. 2.6.3). So one should expect large numbers of generations to be used. Despite this, in numerous studies, it has been found that the average number of fitness evaluations required to solve many problems can be favourably compared with other forms of GP (see for instance [5, 12]).

2.9 Cyclic CGP

CGP has largely been used in an acyclic form, where graphs have no feedback. However, there is no fundamental reason for this. The representation of graphs used in CGP is easily adapted to encode cyclic graphs. One merely needs to remove the restriction that alleles for a particular node have to take values less than the position (address) of the node. However, despite this, there has been little published work where this restriction has been removed. One exception is the recent work of Khan et al., who have encoded artificial neural networks in CGP [3]. They allowed feedback and used CGP to evolve recurrent neural networks. Other exceptions are the work of Walker et al., who applied CGP to the evolution of machine code of sequential and parallel programs on a MOVE processor [11] and Liu et al., who proposed a dual-layer CGP genotype representation in which one layer encoded processor instructions and the other loop control parameters [4]. Also Sect. 5.6.2.1 describes how cyclic analogue circuits can be encoded in CGP.

Clearly, such an investigations would extend the expressivity of programs (since feedback implies either recursion or iteration). It would also allow both synchronous and asynchronous circuits to be evolved. A full investigation of this topic remains for the future.

References

1. Brameier, M., Banzhaf, W.: A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining. *IEEE Transactions on Evolutionary Computation* **5**(1), 17–26 (2001)

2. Clegg, J., Walker, J.A., Miller, J.F.: A New Crossover Technique for Cartesian Genetic Programming. In: Proc. Genetic and Evolutionary Computation Conference, pp. 1580–1587. ACM Press (2007)
3. Khan, M.M., Khan, G.M., Miller, J.F.: Efficient representation of Recurrent Neural Networks for Markovian/Non-Markovian Non-linear Control Problems. In: A.E. Hassanien, A. Abraham, F. Marcelloni, H. Hagnas, M. Antonelli, T.P. Hong (eds.) Proc. International Conference on Intelligent Systems Design and Applications, pp. 615–620. IEEE (2010)
4. Liu, Y., Tempesti, G., Walker, J.A., Timmis, J., Tyrrell, A.M., Bremner, P.: A Self-Scaling Instruction Generator Using Cartesian Genetic Programming. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 6621, pp. 299–310. Springer (2011)
5. Miller, J.F.: An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach. In: Proc. Genetic and Evolutionary Computation Conference, pp. 1135–1142. Morgan Kaufmann (1999)
6. Miller, J.F., Smith, S.L.: Redundancy and Computational Efficiency in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation* **10**(2), 167–174 (2006)
7. Miller, J.F., Thomson, P.: Cartesian Genetic Programming. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 1802, pp. 121–132. Springer (2000)
8. Miller, J.F., Thomson, P., Fogarty, T.C.: Designing Electronic Circuits Using Evolutionary Algorithms: Arithmetic Circuits: A Case Study. In: D. Quagliarella, J. Periaux, C. Poloni, G. Winter (eds.) Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications, pp. 105–131. Wiley (1998)
9. Rechenberg, I.: Evolutionsstrategie – Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Ph.D. thesis, Technical University of Berlin, Germany (1971)
10. Vassilev, V.K., Miller, J.F.: The Advantages of Landscape Neutrality in Digital Circuit Evolution. In: Proc. International Conference on Evolvable Systems, *LNCS*, vol. 1801, pp. 252–263. Springer (2000)
11. Walker, J.A., Liu, Y., Tempesti, G., Tyrrell, A.M.: Automatic Code Generation on a MOVE Processor Using Cartesian Genetic Programming. In: Proc. International Conference on Evolvable Systems: From Biology to Hardware, *LNCS*, vol. 6274, pp. 238–249. Springer (2010)
12. Walker, J.A., Miller, J.F.: Automatic Acquisition, Evolution and Re-use of Modules in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation* **12**, 397–417 (2008)
13. Yu, T., Miller, J.F.: Neutrality and the evolvability of Boolean function landscape. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 2038, pp. 204–217. Springer (2001)
14. Yu, T., Miller, J.F.: Finding Needles in Haystacks is not Hard with Neutrality. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 2278, pp. 13–25. Springer (2002)