Chapter 1
Introduction to Evolutionary Computation and Genetic Programming

Julian F. Miller

1.1 Evolutionary Computation

1.1.1 Origins

Evolutionary computation is the study of non-deterministic search algorithms that are based on aspects of Darwin's theory of evolution by natural selection [10]. The principal originators of evolutionary algorithms were John Holland, Ingo Rechenberg, Hans-Paul Schwefel and Lawrence Fogel. Holland proposed *genetic algorithms* and wrote about them in his 1975 book [19]. He emphasized the role of genetic recombination (often called 'crossover'). Ingo Rechenberg and Hans-Paul Schwefel worked on the optimization of physical shapes in fluids and, after trying a variety of classical optimization techniques, discovered that altering physical variables in a random manner (ensuring small modifications were more frequent than larger ones) proved to be a very effective technique. This gave rise to a form of evolutionary algorithm that they termed an *evolutionary strategy* [38, 40]. Lawrence Fogel investigated evolving finite state machines to predict symbol strings of symbols generated by Markov processes and non-stationary time series [15]. However, as is so often the case in science, various scientists considered or suggested search algorithms inspired by Darwinian evolution much earlier. David Fogel, Lawrence Fogel's son, offers a detailed account of such early pioneers in his book on the history of evolutionary computation [14]. However, it is interesting to note that the idea of artificial evolution was suggested by one of the founders of computer science, Alan Turing, in 1948. Turing wrote an essay while working on the construction of an electronic computer called the Automatic Computing Engine (ACE) at the National Physical Laboratory in the UK. His employer happened to be Sir Charles Darwin, the grandson of Charles Darwin, the author of 'On the Origin of Species'. Sir Charles dismissed the article as a "schoolboy essay"! It has since been recognized that in the article Turing not only proposed artificial neural networks but the field of artificial intelligence itself [44].

J.F. Miller (ed.), *Cartesian Genetic Programming*, Natural Computing Series,
DOI 10.1007/978-3-642-17310-3_1,

1.1.2 Illustrating Evolutionary Computation: The Travelling Salesman Problem

Solving many computational problems can be formulated as a problem of trying to find a string of symbols that lead to a solution. For instance, a well-known problem in computer science is called the travelling salesman problem (TSP). In this problem one wishes to find the shortest route that will visit a collection of cities. The route begins and ends on the same city and must visit every other city exactly once. Assuming that the cities are labelled with the symbols $C_1, C_2, C_3, \ldots, C_n$, then any solution is just a permutation of cities.

The way evolutionary algorithms would proceed on this problem is broadly as set out in Procedure 1.1. Firstly, it is traditional to refer to the string of symbols as a chromosome (or, sometimes, a genotype). The evolutionary algorithm starts by generating a number of chromosomes using randomness; this is called the initial population (step 1). Thus, after this step we would have a number of random permutations of routes (each one visiting each city once). The next step is to evaluate each chromosome in the population. This is referred to as determining the *fitness* of the members of the population (step 2). In the case of the TSP, the fitness of a chromosome is typically the distance covered by the route defined in the chromosome. Thus, in this case one is trying to minimize the fitness. The next step is to select the members of the population that will go forward to create the new population of potential solutions (step 3). Often these chromosomes are called 'parents' as they are used to create new chromosomes (often called 'children'). Generally, children are generated using two operations. The first is called *recombination* or *crossover*. A child is usually generated from two parents, by selecting genes (in this case cities) from each. For example, consider the two parent strings

$$c_2, c_9, c_1, c_8, c_5, c_7, c_3, c_6, c_4, c_{10} \text{ and } C_3, C_8, C_7, C_4, C_6, C_9, C_{10}, C_1, C_5, C_2.$$

Let us suppose that the child is created by taking the first five genes from one parent and the second five from the other. The child chromosome would look like this:

$$c_2, c_9, c_1, c_8, c_5, C_9, C_{10}, C_1, C_5, C_2.$$

However, when we inspect this we have an immediate problem. The child is not a permutation of the ten cities. In such cases, a *repair* procedure needs to be applied that takes the invalid chromosome and rearranges it in some way so that it becomes a valid permutation. In fact, there are a number of ways this can be done, each with advantages and disadvantages [13, 28]. We discuss briefly here an alternative method of recombination of permutation-based chromosomes which does not require a repair procedure. It was proposed by Anderson and Ashlock and is called 'merging crossover' (or MOX) [2]. It works as follows [28]. First, the two parent permutations are randomly merged (i.e. selecting with probability 0.5 a city from

either parent) to create a list of twice the size. For example, suppose we have six cities and the two parents are

$$p_1 = c_3, c_1, c_4, c_6, c_5, c_2 \text{ and } p_2 = C_6, C_4, C_2, C_1, C_5, C_3.$$

After random merging we might obtain

$$c_3, C_6, C_4, c_1, c_4, c_6, C_2, C_1, c_5, C_5, C_3, c_2.$$

After this, we split the extended list in such a way that the first occurrence (reading left to right) of each value in the merged list gives the ordering of cities in the first child, and the second occurrence does so in the second child. Thus we obtain the two child chromosomes

$$d_1 = c_3, C_6, C_4, c_1, C_2, c_5 \text{ and } d_2 = c_4, c_6, C_1, C_5, C_3, c_2.$$

In many other computational problems, solution strings may not be permutational in nature (i.e. strings of binary digits or floating-point numbers). The next step in the generic evolutionary algorithm (step 6) is to mutate some parents and offspring. Mutation means making a random alteration to the chromosome. If the chromosome is a permutation, we should make random changes in such a way that we preserve the permutational nature of the chromosome. One simple way of doing this would be to select two cities at random in the chromosome and swap their positions. For instance, taking d_1 above, and swapping the first city with the third, we obtain the mutated version

$$\tilde{d_1} = C_4, C_6, c_3, c_1, C_2, c_5.$$

Appropriate mutations depend on the nature of the chromosome representation. For instance, with binary chromosomes one usually defines a mutation to be a single inversion of a binary gene. Step 7 of the evolutionary algorithm forms the new population from some combination of parents, offspring and their mutated counterparts. In the optional step 8, some parents are promoted to the next generation without change. This is referred to as *elitism*; as we shall see later in this book, this is very often used in Cartesian genetic programming.

The purpose behind the recombination step, 5, is to combine elements of each parent into a new potential solution. The idea behind it is that as evolution progresses, partial solutions (sometimes referred to as *building blocks*) can be formed in chromosomes which, when recombined, are closer to the desired solution. It has the potential to produce chromosomes that are genetically intermediate between the parent chromosomes. Mutation is often thought of as a local random search operator. It makes a small change, thus moving a chromosome to another that is not very distant (in genotype space) from the parent. As we will see in Chap. 2, the degree of genetic change that occurs through the application of a mutation operator is strongly dependent on what the chromosome represents. In Cartesian genetic programming

Procedure 1.1 Evolutionary algorithm

1: Generate initial population of size p. Set number of generations, $g = 0$
2: **repeat**
3: Calculate the fitness of each member of the population
4: Select a number of parents according to quality
5: Recombine some, if not all, parents to create offspring chromosomes
6: Mutate some parents and offspring
7: Form new population from mutated parents and offspring
8: Optional: promote a number of unaltered parents from step 4 to the new population
9: Increment the number of generations $g \leftarrow g + 1$
10: **until** (g equals number of generations required) **OR** (fitness is acceptable)

it turns out that a single gene change can lead to a huge change in the underlying program encoded in the chromosome (the phenotype).

1.2 Genetic Programming

The automatic evolution of computer programs is known as genetic programming (GP). The origins of GP (and evolutionary computation) go back to the origins of evolutionary algorithms [14]. As early as 1958, Friedberg devised an algorithm that could evaluate the quality of a computer program, make some random changes to it and then test it again to check for improvements, and so on [17, 18]. Smith utilized a form of GP in his PhD thesis in 1980 [42]. In 1981, Forsyth advocated the use of GP for artificial intelligence. He evolved Boolean expressions[1] for three prediction problems: (a) whether heart patients would survive treatment in a hospital, (b) to distinguish athletes good at sprinting from those good at longer distances and (c) to predict British soccer results [16]. In 1985, Cramer evolved sequential programs in the computer languages JB and TB [9]. The latter have the form of symbolic expression trees. He used a few assembler-like functions (coded as positive integers, with integer arguments). Unaware of Cramer's work, also in 1985, Schmidhüber experimented with GP in LISP, later reimplementing it in a form of PROLOG [12, 39]. However, GP started to become more widely known after the publication of John Koza's book in 1992 [20]. One of the obvious difficulties in evolving computer programs is caused by the fact that computer programs are highly constrained and must obey a specific grammar in order to be compiled.

[1] Forsyth used the primitive functions AND, OR, NOT, EQ, NEQ, the comparatives GT, LT, GE, and LE, and the arithmetic operations +, minus, $\times$ and $\div$.

1.2.1 GP Representation in LISP

LISP was invented by John McCarthy in 1958 and is one of the oldest high-level computer languages [25]. Even today, it is widely used by researchers in artificial intelligence. In LISP, all programs consist of S-expressions of lists of symbols enclosed in parentheses. For instance, calls to functions are written as a list with the function name first, followed by its arguments. For example, a function `f` that takes four arguments would be written in LISP as `(f arg1 arg2 arg3 arg4)`. All LISP programs can be written in the form of data structures known as trees. This simplifies the task of obtaining valid programs by applying genetic operations. In 1992, John Koza published a comprehensive work on evolution of computer programs in the form of LISP expressions [20]. The example in Fig. 1.1 shows the tree representation of the LISP expression `(SIGMA (SET-SV (* (% X J))))` (Koza [20], p. 470). This happens to be an evolved S-expression that computes a solution to the differential equation (1.1), where the initial value $y_{initial}$ is 2.718, corresponding to an initial value $x_{initial}$ of 1.0:

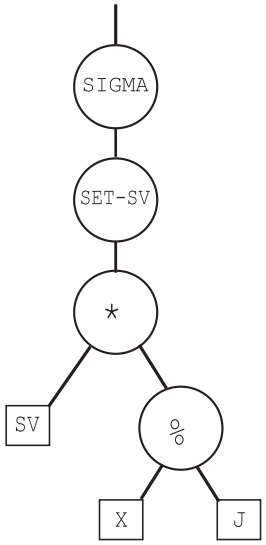


Fig. 1.1 Program tree that represents the S-expression `(SIGMA (SET-SV (* (% X J))))` [20].

$$\frac{dy}{dx} - y = 0. \tag{1.1}$$

The function `SIGMA` is a one-argument function that adds its argument to the value stored in a register called `SV`. It is defined to increment an indexing variable `J` each time it does this. There is a maximum allowed value for `J` of 15 (i.e. the maximum number of iterative summations is 15). The function `%` is a two-argument

function that implements protected division. It is protected, as it is defined to return 1 if the denominator is close to zero. The function `SET-SV` assigns its single argument to the register. `SV` and `J` are defined to have initial values of one. The program sums the successive arguments $x, x^2/2, x^3/3!, \ldots$, which approximates a solution to (1.1), $e^x - 1$.

LISP programs (and programs in general) are highly constrained and obey precisely defined syntax. Ensuring that one can generate random trees is a first step. As we saw, evolutionary algorithms generally use the operations of recombination and mutation to generate new candidate solutions. Recombination in tree-based GP involves exchanging subtrees between parent chromosomes. Mutation involves substituting a subtree by a randomly generated one. Detailed accounts of how this can be accomplished are well known [20, 5, 36, 37]. It should be noted that the size of chromosomes in tree-based GP are variable, as recombination and mutation can create offspring of different sizes.

Human programmers solve problems by divide-and-conquer; that is to say, we break down algorithmic problems into smaller subprocedures that are encapsulated as functions. Such subprocedures are usually reused many times by the main program code. Koza enabled his LISP GP system to automatically construct and use such subprocedures. He called them *automatically defined functions* (ADFs) [21]. In his second book, he showed how they can be used in a GP system and the many computational advantages they bring. ADFs become particularly important in solving harder instances of problems; thus they are seen as a very important in bringing scalability to GP. They also make programs more comprehensible.

1.2.2 Linear or Machine Code Genetic Programming

In linear genetic programming (LGP), programs are a constrained linear set of operations and terminals (inputs). Such programs are fairly similar to programs written in machine code. Originally, Banzhaf evolved chromosomes in the form of bitstrings of length 225 [3, 4]. He defined terminals and functions using a five-bit code; thus his programs could encode up to 45 program instructions. The function set comprised the eight functions `PLUS, MINUS, TIMES, DIV, POW, ABS, MOD and IFEQ`, where `DIV` is protected division, `IFEQ(a,b) = 1` if `a` equals `b` and zero otherwise, and `MOD(a,b)= a mod b` (protected). Five-bit codes with the most significant bit equal to zero code for one of the functions (the eight-function list is duplicated to make 16 entries). The remaining five-bit codes denote terminals (inputs). Banzhaf evolved solutions to the problem of integer sequence prediction. The terminal set consisted of `(I0, I1, 0, 1, 2)`. Randomly altered or generated chromosome strings would not necessarily parse into compilable code and so a simple repair strategy needed to be implemented. A simple example showing the process of translating a chromosome bit string into compilable code (phenotype) is shown in Table 1.1.

Table 1.1 The genotype–phenotype mapping process in linear GP

11110**0**1110**1**0011**1**1101**1**1011**1**1000**1**1100...
Transcribes to
`0 PLUS I0 1 I0 I0 I1...`
After repair
`PLUS PLUS I0 1 I0 I0 I1...`
After parsing
`PLUS(PLUS(I0,1),I0)`
After editing
`function z1(I0,I1)`
`return PLUS(PLUS(I0,1),I0)`

Peter Nordin with Wolfgang Banzhaf, considerably extended this earlier approach to produce a genetic algorithm that manipulates machine code instructions [29, 31, 30]. This technique allowed the use of most program constructs: arrays, arithmetic operators, subroutines, if–then–else, iteration, recursion, strings and list functions. It also allowed incorporating any C function into the GP function set. They used the SPARC instruction set operating on SUN workstations. In this, processor instructions have a 32-bit word length. Crossover operated in such a way that only whole 32-bit instructions were swapped between individuals (thus maintaining validity). Mutation operated by selecting an instruction at random and checked whether it had an embedded constant (a) or whether it was an operation involving only registers (b). If it was case (a), a mutation was a bit-flip in the constant; if case (b), then either the instruction data source or the destination registers were mutated. The resulting system was shown to be up to 2000 times faster than a LISP implementation on some problems.

A more recent form of LGP uses chromosomes representing variable-length strings composed of simple statements in the C programming language [6, 7]. Individual instructions are encoded in a four-component vector. For instance, the C assignment `v[i]= v[j] + c;` would be coded as `(id(+), i, j, c)`. Employing very few possible instructions has proved to be effective in many problems. These are shown in Table 1.2.

Table 1.2 Instruction types in linear GP: `op` refers to allowed arithmetic operations (i.e. +, minus, *, /) and `cmp` is typically $>, \leq$

Instruction type	Function definition
Arithmetic operation 1	`v[i] = v[j] op v[k]`
Arithmetic operation 2	`v[i] = v[j] op c`
Conditional branch 1	`if (v[i] cmp v[k])`
Conditional branch 2	`if (v[i] cmp c))`
Function call	`v[i]=f([v[k])`

Crossover swaps only a whole number of instructions. Mutation is responsible for changing individual instructions by randomly replacing the instruction identifier, a variable or the constant (if it exists) by equivalents from valid ranges. Constants are modified within a user-defined standard deviation from the current value.

1.2.3 Grammar-Based Approaches

Compilers use grammars to define the legal expressions of a computer language. Thus a natural approach to genetic programming is to explicitly evolve chromosomes that obey a given grammar. This would mean that evolving all kinds of constrained structures or languages could be tackled by using the same generic approach but choosing a different grammar. A recent review of such approaches can be found in [26]. In this section, we will briefly discuss perhaps the most widely known and published grammatical approach, namely *grammatical evolution* (GE) [32, 33].

In GE, variable-length binary-string genomes are used grouped into codons of eight bits. The integer value defined by the codon is used via a mapping function to select an appropriate production rule from a grammar defined using the Backus–Naur form (BNF). BNF grammars consist of *terminals* which are items that can appear in the language (i.e. +, *, x, sin, 3.14) and *non-terminals*, which can be expanded into one or more terminals and non-terminals. A grammar can be represented by the tuple {N, T, P, S}, where N is the set of non-terminals, T is a set of terminals, P is a set of production rules that maps the elements of N to T, and S is a start symbol that is a member of N. When there are a number of productions that can be applied, the choice is delimited with the OR symbol, ‘|’. An example is given in Table 1.3

Table 1.3 Example BNF form

Non-terminals			expr, op, pre-op	
Terminals			sin, +, -, /, *, x, 1.0, (,)	
Start symbol			<expr>	
Production rules	(1)	<expr>::	=<expr><op><expr>	(0)
			| (<expr><op><expr>)	(1)
			| <pre-op>(<expr>)	(2)
			| <var>	(3)
	(2)	<op>::=	+	(0)
			| -	(1)
			| /	(2)
			| *	(3)
	(3)	<pre-op>::=	sin	(0)
	(4)	<var>::=	x	(0)
			| 1.0	(1)

Table 1.4 An example genotype in GE. Here, the eight-bit binary codons have all been packed as integers for convenience

220	240	220	203	101	53	202	203	102	55	223	202	243	134	35	202	203	140	39	202	203	102

The mapping process of mapping the genotype shown in Table 1.4 is carried out as follows. The leftmost non-terminal is selected and the symbol noted (i.e. expr, op, pre-op). We denote the codon by C, and the number of production rules for a given expression is N. The rule to apply is $R = C \bmod N$. Then the symbol is rewritten according to this rule. The decoding process continues until no rules can be applied. Note than the genotype is assumed to be circular (it wraps around) so that the first codon follows the last codon. Table 1.5 shows the complete decoding process for the genotype shown in Table 1.4.

Table 1.5 Decoding the example genotype in Table 1.4. Given an expression, with a number of production rules N and a codon C, a rewriting rule is chosen according to $R = C \bmod N$. This is applied to build the next expression

Expression	C	N	R	Rule
<expr>	220	4	0	<expr>::=<expr><op><expr>
<expr><op><expr>	240	4	0	<expr>::=<expr><op><expr>
<expr><op><expr><op><expr>	220	4	0	<expr>::=<expr><op><expr>
<expr><op><expr><op><expr><op><expr>	203	4	3	<expr>::=<var>
<var><op><expr><op><expr><op><expr>	101	2	1	<var>::=1.0
1.0<op><expr><op><expr><op><expr>	53	4	1	<op>::= -
1.0-<expr><op><expr><op><expr>	202	4	2	<expr>::=<pre-op>(<expr>)
1.0-<pre-op>(<expr>)<op><expr><op><expr>				<pre-op>::=sin
1.0-sin(<expr>)<op><expr><op><expr>	203	4	3	<expr>::=<var>
1.0-sin(<var>)<op><expr><op><expr>	102	2	0	<var>::=x
1.0-sin(x)<op><expr><op><expr>	55	4	3	<op>::= *
1.0-sin(x)*<expr><op><expr>	223	4	3	<expr>::=<var>
1.0-sin(x)*<var><op><expr>	202	2	0	<var>::=x
1.0-sin(x)*x<op><expr>	243	4	3	<op>::= *
1.0-sin(x)*x*<expr>	134	4	2	<expr>::=<pre-op>(<expr>)
1.0-sin(x)*x*<pre-op>(<expr>)				<pre-op>::=sin
1.0-sin(x)*x*sin(<expr>)	35	4	3	<expr>::=<var>
1.0-sin(x)*x*sin(<var>)	202	2	0	<var>::=x
1.0-sin(x)*x*sin(x)				

Since, in GE, genotypes are binary strings, no special crossover or mutation operators are required. The genotype-to-phenotype mapping process will always generate syntactically correct individuals. In addition to the standard genetic operators of mutation (point) and crossover, a codon duplication operator is also used. Duplication involves randomly selecting a number of codons to duplicate, and the starting

position of the first codon in this set. The duplicated codons are placed at the end of the chromosome. So the genotype is of variable length.

1.2.4 PushGP

A stack-based computer language called Push has been developed by Lee Spector [43]. His GP system using Push (called PushGP) allows many advanced GP features, such as multiple data types, automatically defined subroutines and control structures. Push was also defined by Spector to support a self-adaptive form of evolutionary computation called *autoconstructive evolution*. An autoconstructive evolution system is an evolutionary computation system that adaptively constructs its own mechanisms of reproduction and diversification as it runs. That is to say, methods of recombination and mutation can be evolved in the system rather than, as is more usual, imposed from the start.

In stack-based computer languages,[2] arguments are passed to instructions via global data stacks. This contrasts with argument-passing techniques based on registers. In stack-based argument passing the programmer first specifies (or computes) arguments that are pushed onto the stack, and then executes an instruction using those arguments. For example, consider adding 2 and 7. This would be written in postfix notation as 2 7 +, and this code specifies that 2 and then 7 should be pushed onto the stack, and then the + instruction should be executed. The + instruction removes the top two elements of the stack, adds them together and pushes the result back onto the stack. If extra arguments are present, they are ignored automatically as each instruction takes only the arguments that it needs from the top of the stack. A stack instruction containing too few arguments would normally be signalled as a run-time error and require program termination; however in Push, an instruction with insufficient arguments is simply ignored (treated as a NOOP).

Push handles multiple data types by providing a stack for each type. There is a stack for integers, a stack for floating-point numbers, a stack for Boolean values, a stack for data types (called TYPE), a stack for program code (CODE) and others. Each instruction takes the inputs that it requires from appropriate stacks and pushes outputs onto appropriate stacks. Using the CODE stack allows Push to handle recursion and subprocedures. The CODE stack also allows evolved programs to push themselves or parts of themselves onto the CODE stack. It is this mechanism that allows programs to define new genetic operators (i.e. recombination and mutation) to create their own offspring.

A Push language reference can be found in [43]. [3]

[2] The language Forth is a well-known example.

[3] Source code is available for research versions of the Push interpreter and PushGP from Lee Spector's website.

1.2.5 Cartesian Graph-Based GP

Unlike trees, where there is always a unique path between any pair of nodes, graphs allow more than one path between any pair of nodes. If we assume all nodes carry out some computational function, representing functions in the form of graphs is more compact than trees since they allow the reuse of previously calculated sub-graphs. Graphs are also attractive representations, since they are widely used in many areas of computer science and engineering [1, 11, 8]. Indeed, neural networks are graphs.

It appears that the first person to evolve graph-based encodings using a Cartesian grid was Sushil Louis in 1990 [22, 23]. In a technical report, Louis described a binary genotype that encodes a network of digital logic gates, in which gates in each column can be connected to the gates in the previous column. Figure 1.2 shows an image extracted from Louis's 1993 PhD thesis [24].

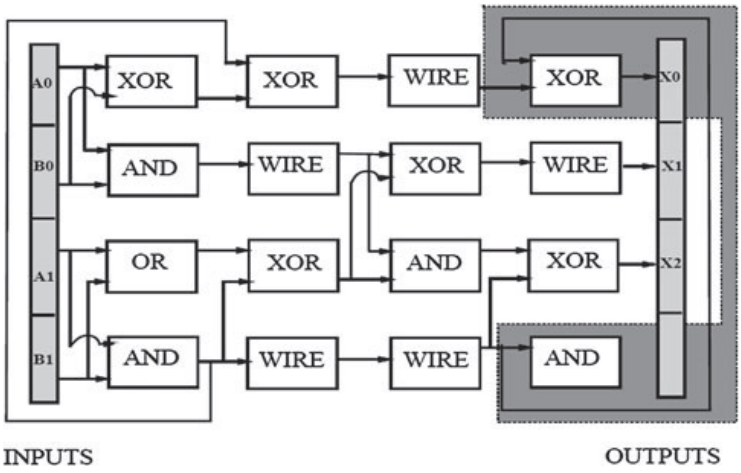


Fig. 1.2 Sushil Louis's evolved 2-bit adder. Image extracted from [24].

Independently, and inspired by neural networks, Poli proposed a graph-based form of GP called parallel distributed GP (PDGP) [34, 35]. PDGP, in principle, allows the evolution of standard tree-like programs, logic networks, neural networks, recurrent transition networks and finite state automata. Some of these are made possible by associating labels with the edges in the program graph. In addition to the usual function and terminal sets, PDGP requires the definition of a set of links that determine how nodes are connected. The labels on the links depend on what is to be evolved. For example, in neural networks, the link labels are numerical constants for the neural-network weights. An example of PDGP is given in Fig. 1.3.

In Fig. 1.3, the programs output is defined to be at coordinates (0,0). Connections point upwards and are allowed only between nodes belonging to adjacent rows. Like Louis, Poli included an identity function so that nodes in non-adjacent rows can still

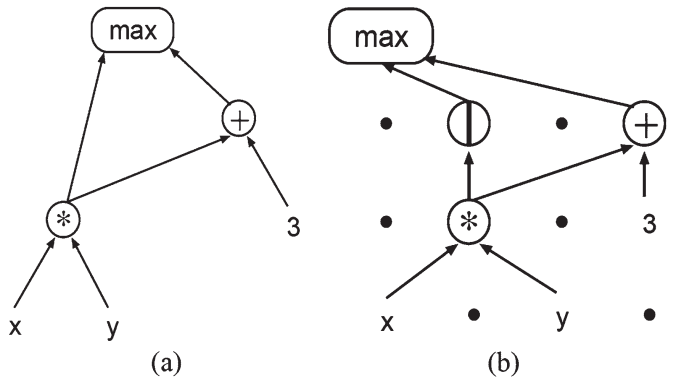


Fig. 1.3 (a) Graph-based representation of the expression max(x*y, 3+x*y), (b) Grid-based representation of the graph in (a). Image extracted from [35].

connect with each other (see the pass-through node in the figure). When PDGP is implemented, the program is represented as an array with the same topology as that of the grid. Each node contains a function label and the horizontal displacement of the nodes in the previous layer used as arguments for the function. The horizontal displacement is an offset from the position of the calling node. Functions or terminals are associated to every node in the grid even if they are not referenced in the program path. Such nodes are inactive (introns).

The basic crossover operator of PDGP is called subgraph active–active node (SAAN) crossover. It is a generalization for graphs of the crossover generally used in tree-based GP to recombine trees. SAAN crossover is defined below, and an example is shown in Fig. 1.4.

1. A random active node is selected in each parent (this is the crossover point),
2. A subgraph including all the active nodes which are used to compute the output value of the crossover point in the first parent is extracted,
3. The subgraph is inserted into the second parent to generate the offspring (if the x coordinate of the insertion node in the second parent is not compatible with the width of the subgraph, the subgraph is wrapped around).

Poli used two forms of mutation in PDGP. A *global* mutation inserts a randomly generated subgraph into an existing program. A *link* mutation changes a random connection in a graph by firstly selecting a random function node, then selecting a random input link of such a node and, finally, altering the offset associated with the link.

As we will see in Chap. 2, Cartesian GP (CGP) also encodes directed graphs; however, the genotype is just a one-dimensional string of integers. Also, CGP genetic operators operate directly on the chromosome, while in PDGP they act directly on the graph. Furthermore, CGP has almost always used mutation (in particular, Poli's link mutation) as its main search operation; however, as we will see later in

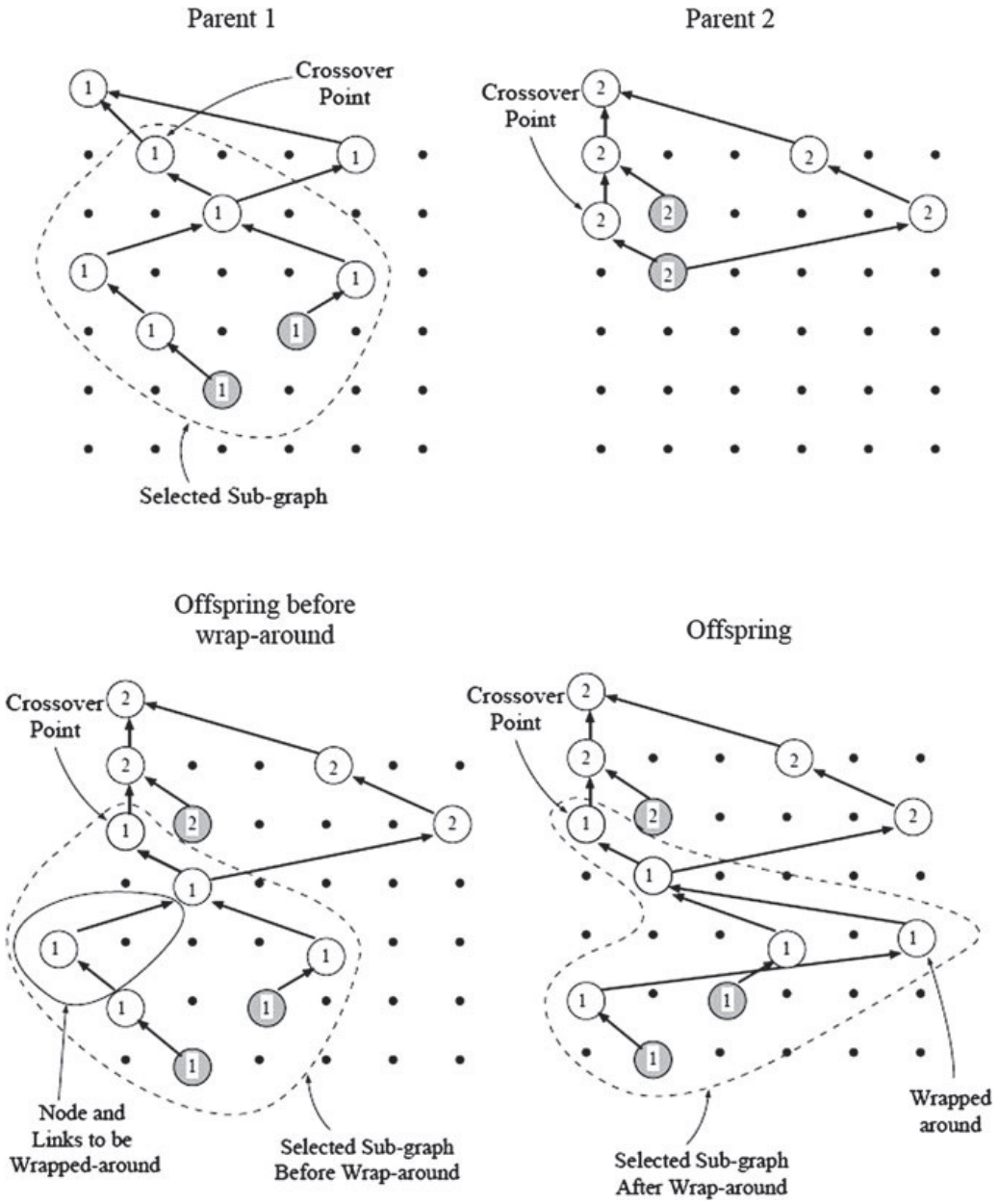


Fig. 1.4 An example of subgraph active–active node (SAAN) crossover [35].

this book, a number of crossover methods have been investigated for CGP. Rather than offsets, CGP uses absolute addresses to determine where nodes obtain their input data from. Graph connectivity is also controlled by a parameter called *levels-back* and nodes can obtain inputs from any of the previous nodes within the range of columns defined by this parameter. In addition, CGP evolutionary algorithms tend to be a form of evolutionary strategy using small populations and elitism.

1.2.6 Bloat

When evolutionary algorithms are applied to many representations of programs, a phenomenon called *bloat* happens. This is where, as the generations proceed, the chromosomes become larger and larger without any increase in fitness. Such programs generally have large sections of code that contain inefficient or redundant subexpressions. This can be a handicap, as it can mean that processing such bloated programs is time-consuming. Eventually, an evolved program could even exceed the memory capacity of the host computer. In addition, the evolved solutions can be very hard to understand and are very inelegant. There are many theories about the causes of bloat and many proposed practical remedies [36, 37, 41]. It is worth noting that Cartesian GP cannot suffer from genotype growth, as the genotype is of fixed size; in addition, it also appears not to suffer from phenotypic growth [27]. Indeed, it will be seen that program sizes remain small even when very large genotype lengths are allowed (see Sect. 2.7).

References

1. Aho, A.V., Ullman, J.D., Hopcroft, J.E.: Data Structures and Algorithms. Addison–Wesley (1983)
2. Anderson, P.G., Ashlock, D.A.: Advances in Ordered Greed. In: C.H. Dagli (ed.) Intelligent Engineering Systems Through Artificial Neural Networks, vol. 14, pp. 223–228. ASME Press (2004)
3. Banzhaf, W.: Genetic programming for pedestrians. In: S. Forrest (ed.) Proc. International Conference on Genetic Algorithms, p. 628. Morgan Kaufmann (1993)
4. Banzhaf, W.: Genetic Programming for pedestrians. Tech. Rep. 93-03, Mitsubishi Electric Research Labs, Cambridge, MA (1993)
5. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming: An Introduction. Morgan Kaufmann (1999)
6. 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)
7. Brameier, M.F., Banzhaf, W.: Linear Genetic Programming. Springer (2006)
8. Chartrand, G., Lesniak, L., Zhang, P.: Graphs and Digraphs, fifth edn. Chapman and Hall (2010)
9. Cramer, N.L.: A Representation for the Adaptive Generation of Simple Sequential Programs. In: J.J. Grefenstette (ed.) Proc. International Conference on Genetic Algorithms and their Applications. Carnegie Mellon University, USA (1985)

10. Darwin, C.: On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life. John Murray (1859)
11. Deo, N.: Graph Theory with Applications to Engineering and Computer Science. Prentice-Hall (2004)
12. Dickmanns, D., Schmidhüber, J., Winklhofer, A.: Der genetische Algorithmus: Eine Implementierung in Prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Technische Universität München (1987)
13. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer (2007)
14. Fogel, D.B.: Evolutionary Computation: The Fossil Record. Wiley-IEEE Press (1998)
15. Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial Intelligence through Simulated Evolution. John Wiley (1966)
16. Forsyth, R.: BEAGLE A Darwinian Approach to Pattern Recognition. Kybernetes **10**(3), 159–166 (1981)
17. Friedberg, R.: A learning machine: Part I. IBM Journal of Research and Development **2**, 2–13 (1958)
18. Friedberg, R., Dunham, B., North, J.: A learning machine: Part II. IBM Journal of Research and Development **3**, 282–287 (1959)
19. Holland, J.H.: Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. University of Michigan Press, Ann Arbor, MI (1975)
20. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press, Cambridge, Massachusetts, USA (1992)
21. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, Massachusetts (1994)
22. Louis, S., Rawlins, G.J.E.: Using Genetic Algorithms to Design Structures. Tech. Rep. 326, Department of Computer Science, Indiana University (1990)
23. Louis, S., Rawlins, G.J.E.: Designer Genetic Algorithms: Genetic Algorithms in Structure Design. In: Proc. International Conference on Genetic Algorithms, pp. 53–60. Morgan Kauffmann (1991)
24. Louis, S.J.: Genetic algorithms as a computational tool for design. Ph.D. thesis, Department of Computer Science, Indiana University (1993)
25. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Communications of the ACM, 184–195 (1960)
26. McKay, R., Hoai, N., Whigham, P., Shan, Y., O'Neill, M.: Grammar-based Genetic Programming: a survey. Genetic Programming and Evolvable Machines **11**(3), 365–396 (2010)
27. Miller, J.F., Smith, S.L.: Redundancy and Computational Efficiency in Cartesian Genetic Programming. IEEE Transactions on Evolutionary Computation **10**(2), 167–174 (2006)
28. Mumford, C.L.: New Order-Based Crossovers for the Graph Coloring Problem. In: T. Runarsson, H.G. Beyer, E. Burke, J. Merelo-Guervós, L. Whitley, X. Yao (eds.) Parallel Problem Solving from Nature - PPSN IX, *LNCS*, vol. 4193, pp. 880–889 (2006)
29. Nordin, P.: A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In: K.E. Kinnear (ed.) Advances in Genetic Programming, pp. 311–331. MIT Press (1994)
30. Nordin, P.: Evolutionary program induction of binary machine code and its applications. Ph.D. thesis, Department of Computer Science, University of Dortmund, Germany (1997)
31. Nordin, P., Banzhaf, W.: Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code. In: Proc. International Conference on Genetic Algorithms, pp. 318–327. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1995)
32. O'Neill, M., Ryan, C.: Grammatical Evolution. IEEE Transactions on Evolutionary Computation **5**(4), 349–358 (2001)
33. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Springer (2003)
34. Poli, R.: Parallel distributed genetic programming. Tech. Rep. CSRP-96-15, School of Computer Science, University of Birmingham (1996)

35. Poli, R.: Evolution of Graph-Like Programs with Parallel Distributed Genetic Programming. In: E. Goodman (ed.) Proc. International Conference on Genetic Algorithms, pp. 346–353. Morgan Kaufmann (1997)
36. Poli, R., Langdon, W.B.: Foundations of Genetic Programming. Springer (2002)
37. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (2008)
38. Rechenberg, I.: Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Ph.D. thesis, Technical University of Berlin, Germany (1971)
39. Schmidhüber, J.: Evolutionary principles in self-referential learning. Diploma thesis, Institut für Informatik, Technical University of München (1987)
40. Schwefel, H.P.: Numerische Optimierung von Computer-Modellen. Ph.D. thesis, Technical University of Berlin (1974)
41. Silva, S., Costa, E.: Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. Genetic Programming and Evolvable Machines **10**, 141–179 (2009)
42. Smith, S.F.: A Learning System Based on Genetic Adaptive Algorithms. Ph.D. thesis, University of Pittsburgh (1980)
43. Spector, L., Robinson, A.: Genetic Programming and Autoconstructive Evolution with the Push Programming Language. Genetic Programming and Evolvable Machines **3**, 7–40 (2002)
44. Turing, A.: Intelligent Machinery. In: D. Ince (ed.) Collected Works of A. M. Turing: Mechanical Intelligence. Elsevier Science (1992)

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

J.F. Miller (ed.), *Cartesian Genetic Programming*, Natural Computing Series,
DOI 10.1007/978-3-642-17310-3_2,

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

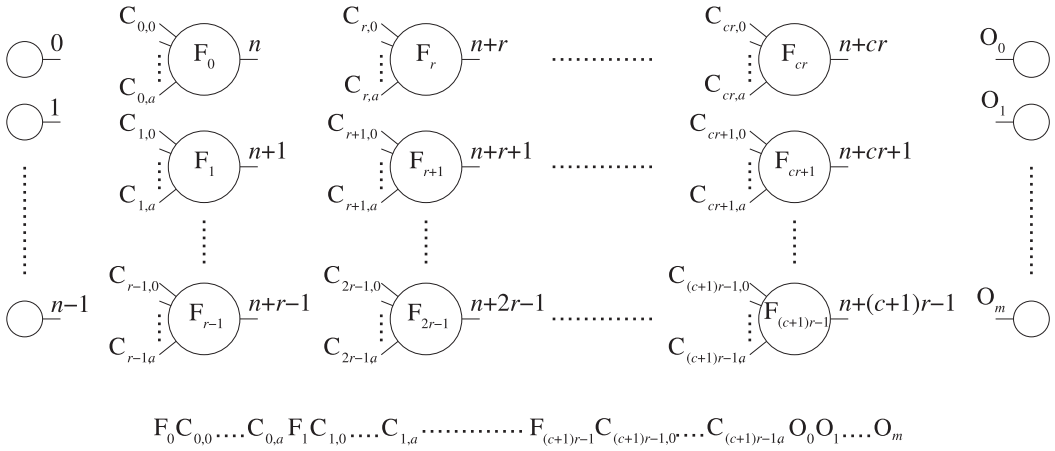


Fig. 2.1 General form of CGP. It is a grid of nodes whose functions are chosen from a set of primitive functions. The grid has n_c columns and n_r rows. The number of program inputs is n_i and the number of program outputs is n_o. Each node is assumed to take as many inputs as the maximum function arity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

$$0 \leq f_i \leq n_f. \tag{2.1}$$

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 $j \geq l$,

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

If $j < l$,

$$0 \leq C_{ij} \leq n_i + jn_r. \tag{2.3}$$

Output genes O_i can connect to any node or input:

$$0 \leq O_i < n_i + L_n. \tag{2.4}$$

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

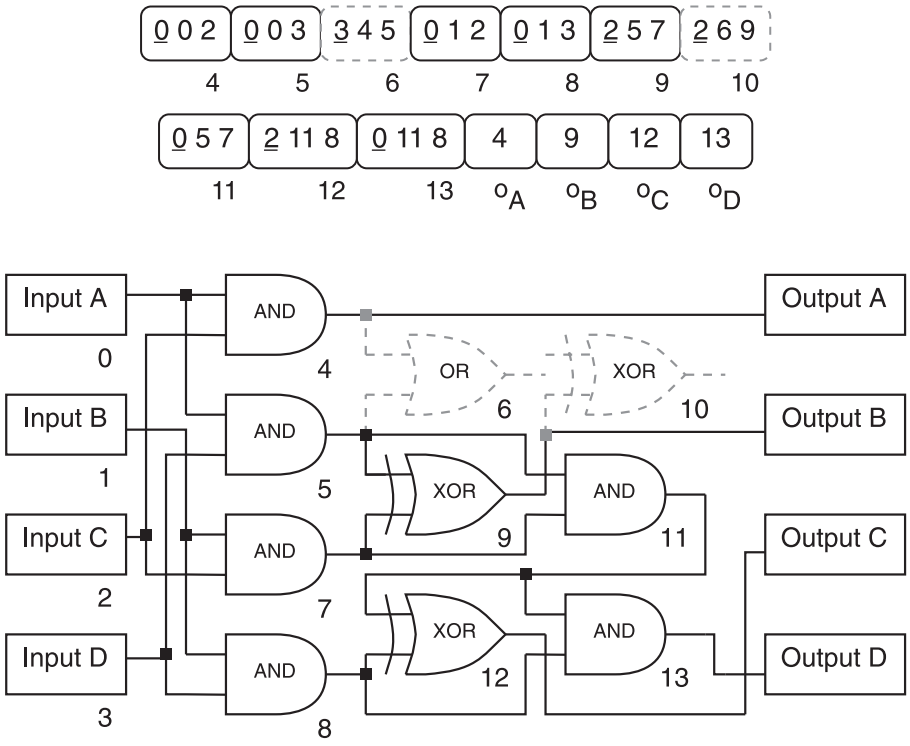


Fig. 2.2 A CGP genotype and corresponding phenotype for a two-bit multiplier circuit. The underlined genes in the genotype encode the function of each node. The function look-up table is AND (0), AND with one input inverted (1), XOR (2) and OR (3). The addresses are shown underneath each program input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

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

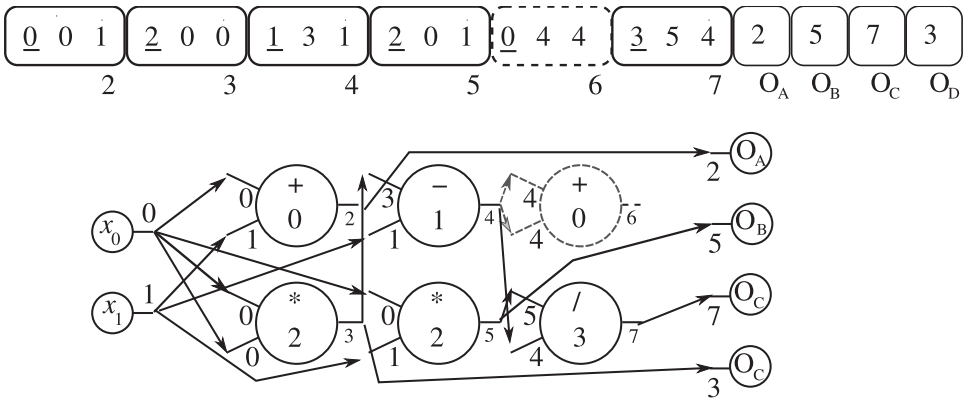


Fig. 2.3 A CGP genotype and corresponding schematic phenotype for a set of four mathematical equations. The underlined genes in the genotype encode the function of each node. The function look-up table is add (0), subtract (1), multiply (2) and divide (3). The addresses are shown underneath each program 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	$\sqrt{\lvert x - y \rvert}$
4	$255(\lvert \sin(\frac{2\pi}{255}x) + \cos(\frac{2\pi}{255}y) \rvert)/2$
5	$255(\lvert \cos(\frac{2\pi}{255}x) + \sin(\frac{2\pi}{255}y) \rvert)/2$
6	$255(\lvert \cos(\frac{3\pi}{255}x) + \sin(\frac{2\pi}{255}y) \rvert)/2$
7	$\exp(x + y) \pmod{256}$
8	$\lvert \sinh(x + y) \rvert \pmod{256}$
9	$\cosh(x + y) \pmod{256}$
10	$255 \lvert \tanh(x + y) \rvert$
11	$255(\lvert \sin(\frac{\pi}{255}(x + y)) \rvert$
12	$255(\lvert \cos(\frac{\pi}{255}(x + y)) \rvert$
13	$255(\lvert \tan(\frac{\pi}{8*255}(x + y)) \rvert$
14	$\sqrt{\frac{x^2 + y^2}{2}}$
15	$\lvert x \rvert^y \pmod{256}$
16	$\lvert x + y \rvert \pmod{256}$
17	$\lvert x - y \rvert \pmod{256}$
18	$xy/255$
19	$\begin{cases} x & y = 0 \\ x/y & y \neq 0 \end{cases}$

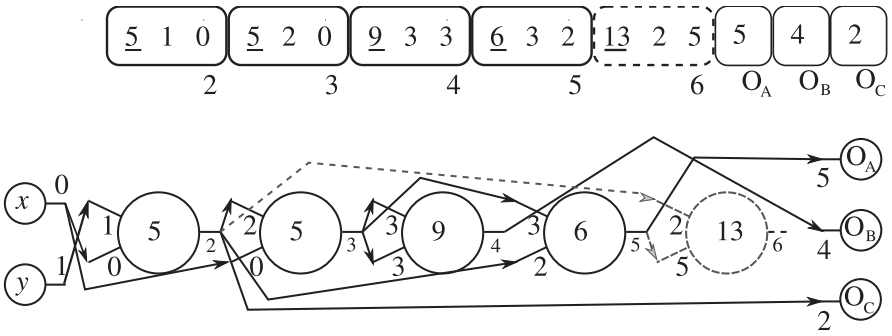


Fig. 2.4 A CGP genotype and corresponding schematic phenotype for a program that defines a picture. The underlined genes in the genotype encode the function of each node. The function look-up table is given in Table 2.1. The addresses are shown underneath each program input and node in the genotype and phenotype. The inactive areas of the genotype and phenotype are shown in grey dashes (node 6).

$$
\begin{aligned}
g_2 &= 255\left(\left|\cos\left(\frac{2\pi}{255}y\right) + \sin\left(\frac{2\pi}{255}x\right)\right|\right)\Big/2,\\
g_3 &= 255\left(\left|\cos\left(\frac{2\pi}{255}g_2\right) + \sin\left(\frac{2\pi}{255}x\right)\right|\right)\Big/2,\\
g_4 &= \cosh(2g_3) \pmod{256},\\
g_5 &= 255\left(\left|\cos\left(\frac{2\pi}{255}g_3\right) + \sin\left(\frac{2\pi}{255}g_2\right)\right|\right)\Big/2,\\
r &= g_5,\\
g &= g_4,\\
b &= g_2.
\end{aligned}
\tag{2.6}
$$

When these mathematical equations are executed for all 256^2 pixel locations, they produce the picture (the actual picture is in colour) shown in Fig. 2.5.

2.5 Decoding a CGP Genotype

So far, we have illustrated the genotype-decoding process in a diagrammatic way. However, the algorithmic process is recursive in nature and works from the output genes first. The process begins by looking at which nodes are 'activated' by being directly addressed by output genes. Then these nodes are examined to find out which nodes they in turn require. A detailed example of this decoding process is shown in Fig. 2.6.

It is important to observe that in the decoding process, non-coding nodes are not processed, so that having non-coding genes presents little computational overhead.

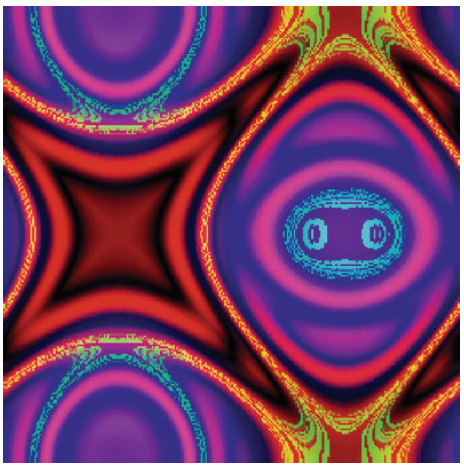


Fig. 2.5 Picture produced when the program encoded in the evolved genotype shown in Fig. 2.4 is executed. It arose in the sixth generation. The user decides which genotype will be the next parent.

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-

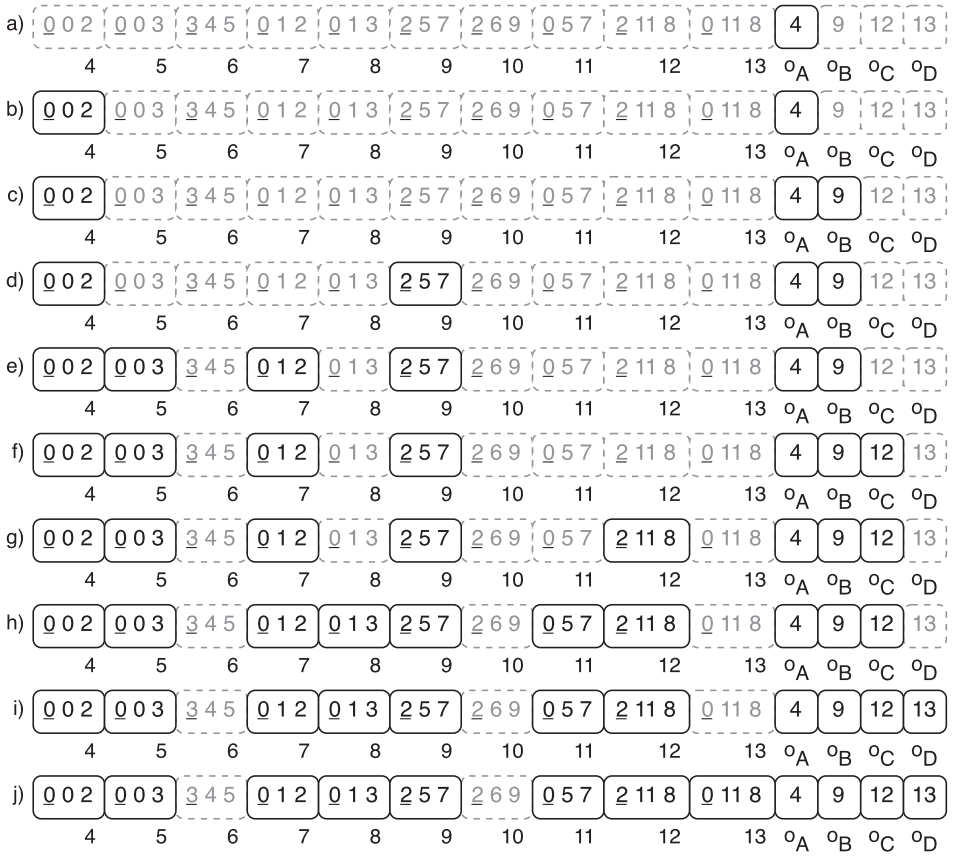


Fig. 2.6 The decoding procedure for a CGP genotype for the two-bit multiplier problem. (a) Output A (o_A) connects to the output of node 4; move to node 4. (b) Node 4 connects to the program inputs 0 and 2; therefore the output A is decoded. Move to output B. (c) Output B (o_B) connects to the output 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 ≤ i < M do
3:     NU[i] = FALSE
4: end for
5: for all i such that L_g − n_o ≤ i < L_g do
6:     NU[G[i]] ← TRUE
7: end for
8: for all i such that M − 1 ≥ i ≥ 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 ≤ 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 ≤ j < Arity(NG[n_n − 1]) do
15:            NU[NG[j]] ← TRUE
16:        end for
17:    end if
18: end for
19: n_u = 0
20: for all j such that n_i ≤ j < M do // store active node addresses in NP
21:    if NU[j] = TRUE then
22:        NP[n_u] ← j
23:        n_u ← n_u + 1
24:    end if
25: end for
26: return n_u
```

Procedure 2.2 Decoding CGP to get the output

```
1: DecodeCGP(G,DIN,O,n_u,NP,item)
2: for all i such that 0 ≤ i < n_i do
3:     o[i] ← DIN[item]
4: end for
5: for all j such that 0 ≤ j < n_u do
6:     n ← NP[j] − n_i
7:     g ← n_n n
8:     for all i such that 0 ≤ 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 + n_n − 1] // get function gene of node
12:    o[n + n_i] = NF(in, f) // calculate output of node
13: end for
14: for all j such that 0 ≤ j < n_o do
15:    O[j] ← o[G[L_g − n_o + 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

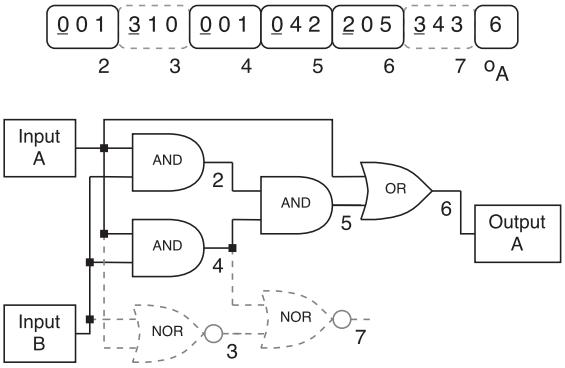
1: $FitnessCGP(G)$
2: $n_u \leftarrow NodesToProcess(G,NP)$
3: $fit \leftarrow 0$
4: **for all** i such that $0 \leq i < N_{fc}$ **do**
5: $\quad DecodeCGP(G,DIN,O,n_u,NP,item)$
6: $\quad f_i = EvaluateCGP(O,DOUT,i)$
7: $\quad fit \leftarrow fit + f_i$
8: **end for**

2.6 Evolution of CGP Genotypes

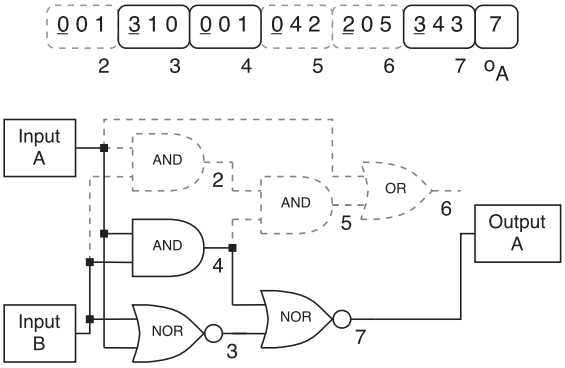
2.6.1 Mutation

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.



(a) Before mutation



(b) After mutation

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 ≤ 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 ≤ 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

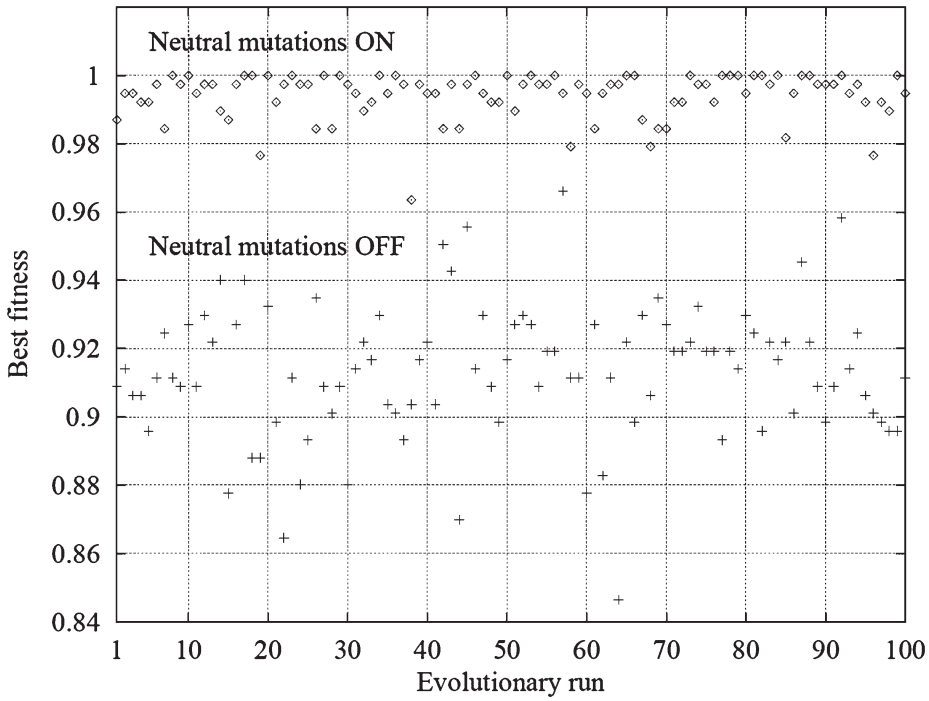


Fig. 2.8 The normalized best fitness value achieved in two sets of 100 independent evolutionary runs of 10 million generations. The target of evolution was to evolve a correct three-bit digital parallel multiplier. In one set of runs, neutral drift was allowed, and in the other, neutral drift was not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