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].

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

$c_2, c_9, c_1, c_8, c_5, c_7, c_3, c_6, c_4, c_{10}$ 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:

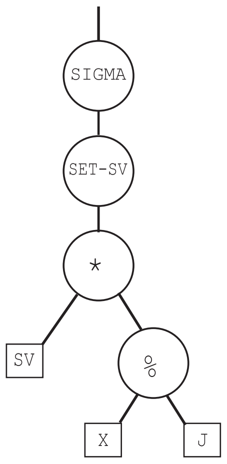

Fig. 1.1 Program tree that represents the S-expression `(SIGMA (SET-SV (* (% X J))))` [20].

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

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].

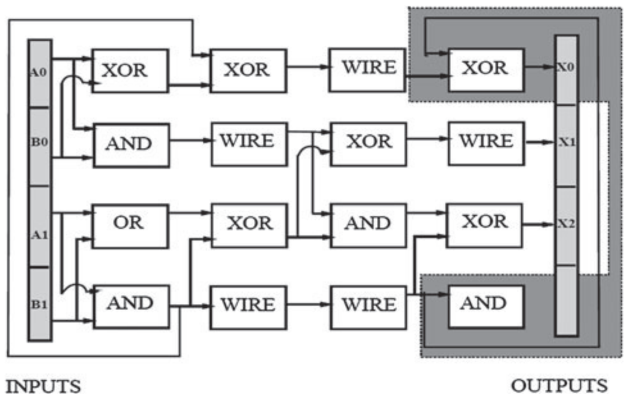

Fig. 1.2 Sushil Louis's evolved 2-bit adder. Image extracted from [24].

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

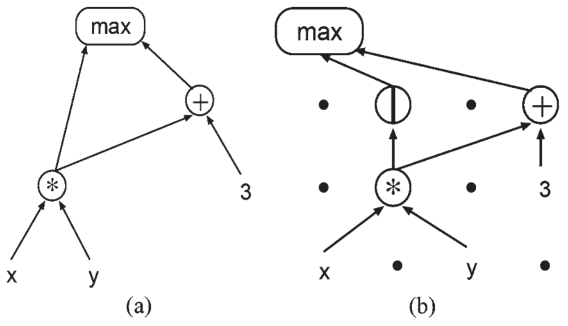

Fig. 1.3 (a) Graph-based representation of the expression max(x*y, 3+x*y), (b) Grid-based representation of the graph in (a). Image extracted from [35].

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

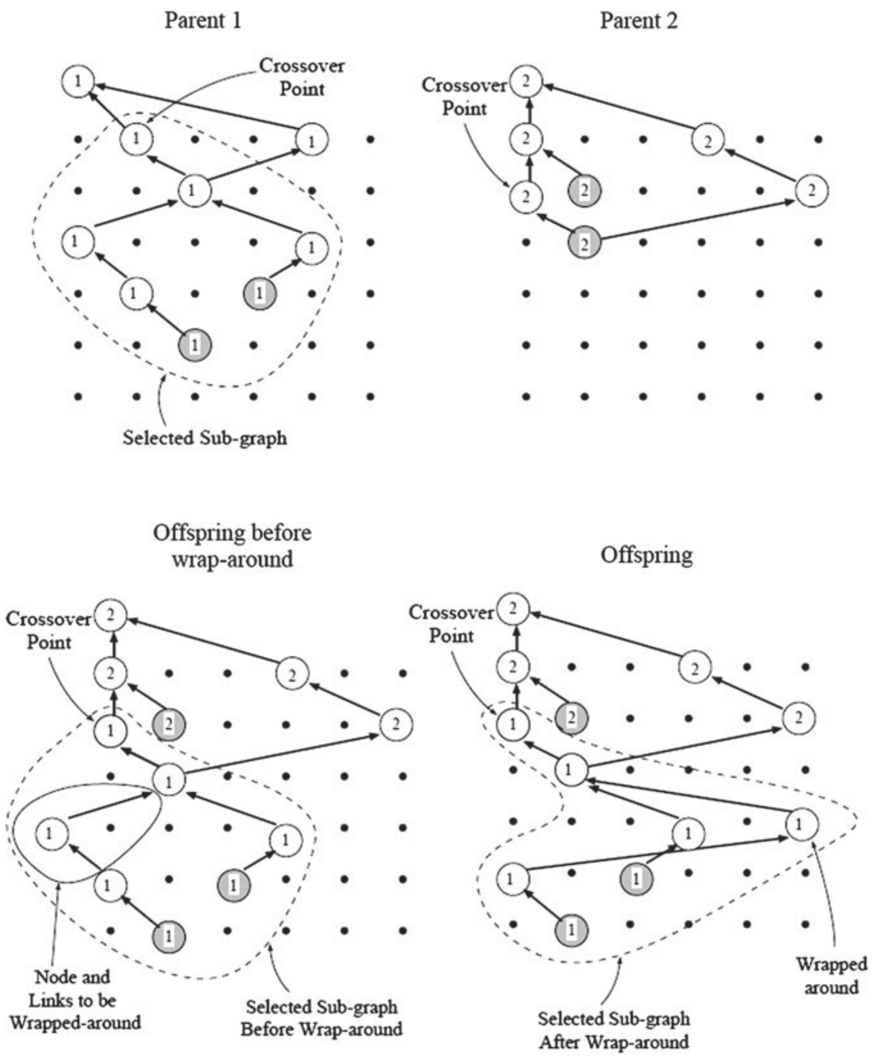
Parent 1
Parent 2
Crossover Point
Crossover Point
Selected Sub-graph
Offspring before wrap-around
Offspring
Crossover Point
Crossover Point
Node and Links to be Wrapped-around
Selected Sub-graph Before Wrap-around
Selected Sub-graph After Wrap-around
Wrapped around

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)